CONF-9310206--

Proceedings

Sisal '93

John T. Feo
editor

October 1993

Lawrence Livermore National Laboratory

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Proceedings

Sisal '93

Editor

John T. Feo
Lawrence Livermore National Laboratory
Livermore, California

Sponsored by

Computer Research Group
Lawrence Livermore National Laboratory
Livermore, CA

October 3-5, 1993

San Diego, California

[illegible]

Contents

Programmability and Performance Issues: the Case of an Iterative Partial Differential Equation Solver*

Chinhyun Kim   Jean-Luc Gaudiot

Electrical Engineering-Systems Dept.
University of Southern California
Los Angeles, CA 90089-2563

Wlodek Proskurowski

Mathematics Dept.
University of Southern California
Los Angeles, CA 90089-1113

Abstract

In this paper, we use a specific example to discuss the viability of functional programming in the context of parallel computing. The traditional argument for functional languages has been programmability. Indeed, due to high-level abstractions and the implicit parallelism provided by functional languages, programmers are free to concentrate on the implementation of the algorithm at hand without being burdened with low-level machine execution details. We further report that it is possible to deliver both programmability and performance through functional programming. Some quantitative results from an experiment which consists of developing a multigrid elliptic Partial Differential Equation (PDE) solver are presented.

1 Introduction

In current parallel programming style using imperative languages such as FORTRAN or C, an applications programmer needs to be aware of the architectural details of the target machine in order to generate an efficient program [7, 10]. This is due to the execution model (von Neumann) of most existing programming languages. Such practice makes writing parallel programs difficult. Furthermore, once written, porting a program to a machine with a different architecture virtually means rewriting the whole program. At present, however, no other programming languages can compete with imperative languages in performance.

Functional languages such as SISAL [5, 8] provide higher-level abstractions so that underlying machine architecture is transparent to a programmer. In addition, representation of parallel operations is implicit in the language semantics. These features allow a programmer to concentrate on the implementation details oı the algorithm at hand without worrying about parallelization and other low-level machine mechanisms. The drawback of functional languages, however, have been performance. Programming in functional languages can be a double-edged sword. That is, while high-level abstractions free programmers from low-level details, it could be difficult to achieve good performance.

Much work has been done in compilation techniques to improve the performance of functional languages [2]. It has been shown that the latest SISAL compiler called the Optimizing SISAL Compiler (OSC) [3] can compete with the FORTRAN compiler on CRAY machines [4]. The CRAY FORTRAN compiler can be considered one of the best commercially available optimizing compilers. Thanks to OSC, using the performance issue in the argument against functional programming has been substantially weakened. On the other hand, the alleged functional language feature of programmability issue has not been well substantiated.

A desirable (parallel) programming environment is one that which shields a programmer from the low-level machine details without sacrificing performance. The objective of this paper, therefore, is to address the issues of programmability and performance of functional programming together by presenting some empirical results from an experiment. The experiment is based on a one semester graduate level course on numerical methods of elliptic Partial Differential Equations (PDE). In the course, four different algorithms for numerically solving elliptic PDEs are presented and each student is required to implement the PDE solvers within some specified time. In the course, students are free to use a programming language of his/her choice. The four iterative PDE solvers covered in the course are based on :

*This work was supported in part by the National Science Foundation under grant No. CCR-9013965.

1. Basic iterative methods : Jacobi, Gauss-Seidel, and Succesive Over Relaxation (SOR),
2. Multigrid method,
3. Preconditioned Conjugate Gradient (PCG) method (and Fast Poisson solvers),
4. Domain Decomposition method.

The experiment consists of participating in the course and implementing the PDE solvers in SISAL within the assigned date. Once the PDE solvers are written, their performance on various types of parallel machines are measured in addition to uniprocessor machines. There is to be no modifications made to the programs that run on various machines. Note that implementing the PDE solvers for parallel machines is not part of the course. In the course, students were to write programs only for sequential machines.

The programmer participating in the experiment had the following background at the start of the experiment:

- Understood programming language issues in general, but has not written any substantial SISAL programs prior to the experiment.
- Had no knowledge of the numerical PDE solver algorithms covered in the class prior to the experiment.

Due to space limitation, this paper concentrates the discussion on the implementation of a multigrid method. In section 2, the model problem used in the experiment is discussed. In addition, characteristics of the multigrid algorithm is described in some detail. Section 3 describes the implementation of a multigrid algorithm. In section 4, performance measurements of the implemented multigrid solver on sequential and parallel machines are described. In addition, the development time of each solver is described. Section 5 ends with some concluding remarks.

2 Description of the Problem

In this section, the model problem used in the experiment is discussed. In addition, two multigrid algorithms are described in some detail.

2.1 The Model Problem

The model problem used in the experiment is the following two-dimensional self-adjoint elliptic equation in the unit square with proper Dirichlet (static) boundary conditions :

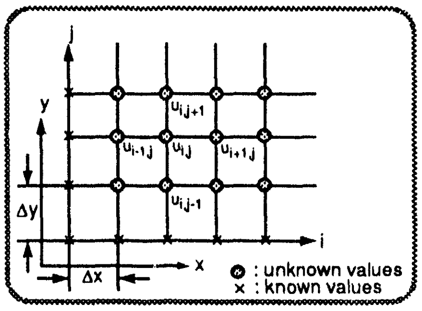


Figure 1: The finite difference method discretizes a continuous region into a finite number of grid points by dividing the region of interest into equal grid sizes.

$$-\textbf{div (k grad u)} = \text{f}$$

Two different values for the diffusion function $k(x,y)$ are used. The first case is when $k(x,y) = 1$. This results in the well-known Poisson's equation :

$$-[\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2}] = f(x,y)$$

In general, the resulting equation is of the following form :

$$-\frac{\partial}{\partial x}[k(x,y)\frac{\partial u(x,y)}{\partial x}] - \frac{\partial}{\partial y}[k(x,y)\frac{\partial u(x,y)}{\partial y}] = f(x,y)$$

In order to solve the problem numerically, the continuous partial differential equation needs to be discretized. In other words, solution of the dependent variables are determined only at discrete points within the problem domain although the variables vary continuously throughout the domain. In the experiment, the partial differential equation is discretized using the finite difference method. The finite difference method is based on Taylor series expansion in which the order of the *truncation error* depends on the number of terms selected from the Taylor series. In the experiment, second order approximation is used, *i.e.*, truncation error is of order $O((\Delta x)^2, (\Delta y)^2)$; Δx and Δy are the grid spaces in the x and y axis, respectively (Figure 1). If $\Delta x = \Delta y = h$, an unknown variable u at discrete point x_i and y_j (when $k(x,y) = 1$) can be approximated as in the following equation :

$$u_{i,j} = \frac{u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} + h^2 f_{i,j}}{4}$$

$$\text{where } 0 \le i \le N, \text{ and } 0 \le j \le M$$

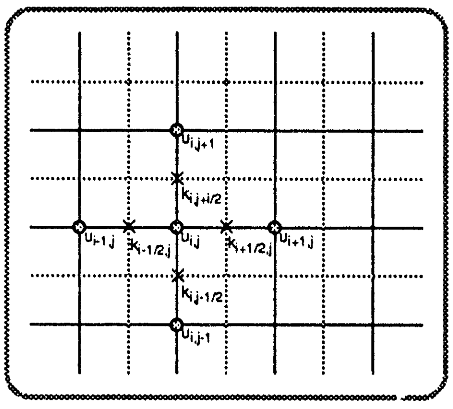


Figure 2: When the diffusion function is not constant, using a staggered grid scheme results in a symmetric coefficient matrix.

The variables N and M are the number of grid points in the x and y directions, respectively. The $u_{i,j}$ and $f_{i,j}$ represent the variables $u(x,y)$ and $f(x,y)$ at discrete grid points x_i and y_j.

When $k(x,y)$ is a function of x and y, *staggered grid* method is used so that the resulting coefficient matrix is symmetric and the $O(h^2)$ accuracy is retained. The resulting difference equation looks like the folowing :

$$\begin{aligned}
u_{i,j} &= \frac{K_1 u_{i-1,j} + K_2 u_{i+1,j} + K_3 u_{i,j-1} + K_4 u_{i,j+1}}{K_0} \\
&\quad + \frac{h^2 f_{i,j}}{K_0} \\
K_0 &= k_{i-1/2,j} + k_{i+1/2,j} + k_{i,j-1/2} + k_{i,j+1/2} \\
K_1 &= k_{i-1/2,j} \\
K_2 &= k_{i+1/2,j} \\
K_3 &= k_{i,j-1/2} \\
K_4 &= k_{i,j+1/2}
\end{aligned}$$

Once the partial differential equation is discretized, a system of linear equations results which can be written in a vector form as shown below :

$$\mathbf{A}_{(n\times n)}\mathbf{u}_{(n\times 1)} = \mathbf{f}_{(n\times 1)}$$

- **A** is the coefficient matrix. Its size is $n \times n$ and has the characteristics of being sparse and symmetric.
- **u** is a vector of unknown variables.
- **f** is a vector of the values of $f(x,y)$ at discrete points.

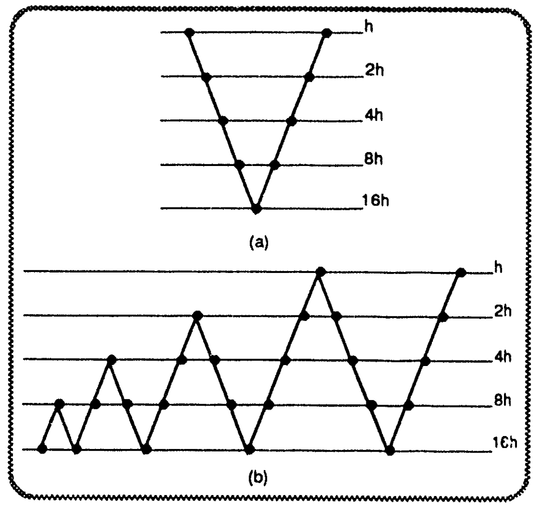


Figure 3: The top figure shows the V-cycle and the bottom figure shows the FMV-cycle.

2.2 Multigrid Method

The disadvantage of basic iterative methods is their slow rate of convergence. Of the three methods listed, SOR has the best performance. This, however, is based on the assumption that the optimum value of the weighting parameter ω_{opt} is known [6]. Unfortunately, the value of ω_{opt} is usually unknown. Conceptually, the slowness of the basic iterative methods is attributed to the fact that they act as a low-pass filter with a fixed cutoff frequency. Initially, the convergence rate is fast because the high frequency error components are quickly filtered out. However, once the high-frequency error components are filtered out and only the low-frequency (or smooth) error components are left, the convergence rate becomes very slow.

The basic idea behind the Multigrid method is to take advantage of the fact that low-frequency error components of a fine grid becomes high-frequency error components in a coarser grid [1]. Thus, the strategy is to move down to a coarser grid once the convergence rate at the current grid saturates. We can think of this as a low-pass filter whose cutoff frequency can vary. That is, once the high-frequency components are filtered out, the cutoff frequency can be further moved down so that the error components which could not be filtered out in the previous setting can be filtered. Thus, fast convergence rate can be sustained by moving through different grid levels.

The basic Multigrid scheme forms a V-cycle in which the downward path computes the residual error while the upward path is the correction path which updates the old estimation with a new approximation. The algorithm of the V-cycle in a recursive form is as follows [1] :

- $\mathbf{A}^h$ is a coefficient matrix at grid level h.
- $\mathbf{f}^h$ is a vector of the values of $f(x,y)$ at grid level h.
- $\mathbf{v}^h$ is a vector of the unknown variable approximations at grid level h.
- I_h^{2h} is an interpolation function mapping from a fine grid to a coarse grid. Also called, *restriction*.
- I_{2h}^h is an interpolation function mapping from a coarse grid to a fine grid.

Algorithm MV : $\mathbf{v}^h \leftarrow MV^h(\mathbf{v}^h, \mathbf{f}^h)$

```
1. Relax $\nu_1$ times on $\mathbf{A}^h\mathbf{v}^h = \mathbf{f}^h$ with
   initial guess $\mathbf{v}^h$.
2. If $\Omega^h$ is $\Omega^H$ (coarsest grid) then go to 4.
   Else $\mathbf{f}^{2h} \leftarrow I_h^{2h}(\mathbf{f}^h - \mathbf{A}^h\mathbf{v}^h)$
        $\mathbf{v}^{2h} \leftarrow 0$ (zero as initial guess, for error)
        $\mathbf{v}^{2h} \leftarrow MV^{2h}(\mathbf{v}^{2h}, \mathbf{f}^{2h})$
   End if
3. Correct $\mathbf{v}^h \leftarrow \mathbf{v}^h + I_{2h}^h\mathbf{v}^{2h}$.
4. Relax $\nu_2$ times on $\mathbf{A}^h\mathbf{v}^h = \mathbf{f}^h$ with $\mathbf{v}^h$ as
   initial guess.
```

A more efficient multigrid scheme called the *full multigrid* (FMV) computes the initial guess on the finest level by performing the V-cycle at every grid level using the corrected value of v at the coarser level as the new initial guess. Its algorithm in a recursive form is as follows [1] :

Algorithm FMV : $\mathbf{v}^h \leftarrow FMV^h(\mathbf{v}^h, \mathbf{f}^h)$

```
1. If $\Omega^h$ is $\Omega^H$ then go to 3.
   Else $\mathbf{f}^{2h} \leftarrow I_h^{2h}(\mathbf{f}^h - \mathbf{A}^h\mathbf{v}^h)$
        $\mathbf{v}^{2h} \leftarrow 0$
        $\mathbf{v}^{2h} \leftarrow FMV^{2h}(\mathbf{v}^{2h}, \mathbf{f}^{2h})$
   End if
2. Correct $\mathbf{v}^h \leftarrow \mathbf{v}^h + I_{2h}^h\mathbf{v}^{2h}$.
3. $\mathbf{v}^h \leftarrow MV^h(\mathbf{v}^h, \mathbf{f}^h)$ $\nu_0$ times.
```

3 Implementation

In the experiment, a full multigrid algorithm is implemented. As shown in Figure 3 (b), this method starts out at the coarsest grid in which each grid points are computed to exact values. These grid points are then interpolated to the next finer grid. Then a V-cycle is performed on these interpolated grid points. This is repeated at every grid level until the finest grid level is reached. On the finest level the MV algorithm is performed ν_0 times. This section describes the SISAL implementation of the algorithm and various functions performed as part of the multigrid operations.

```
function MultiV (N,n1,n2:integer; V,F : TwoDim returns TwoDim)
   let
      NewGrid := Relax(N,n1,V,F);
      UpdateGrid := if N = 2 then
         Relax(N,1,NewGrid,F)
      else
         let
            Residue := ComputeRes(N,F,NewGrid);
            CoarseF := Restrict(N/2,Residue);
            CoarseErrorG := MultiV(N/2,n1,n2,InitVal(N/2),CoarseF);
            ErrorGrid := InterP(N,CoarseErrorG);
            CorrectedV := Correction(N,NewGrid,ErrorGrid)
         in
            Relax(N,n2,CorrectedV,F)
         end let
      end if
   in
      UpdatedGrid
   end let
end function
```

Figure 4: The function **MultiV** is a SISAL implementation of a recursive algorithm which performs a V-cycle.

The function **MultiV** shown in Figure 4 is a SISAL implementation that performs a V-cycle. The function is written in a recursive style and closely resembles the Algorithm MV description. It has five input parameters and one output parameter which is a two dimensional array. The data type **TwoDim** is a user defined data type which is really a two dimensional array of double precision floating point numbers. The first three input parameters **N,n1,n2** of type **integer** are the grid size, the number of relaxations in the downward V-cycle, and the number of relaxations in the upward V-cycle, respectively. Among the two input parameters of type **TwoDim**, **V** is the current approximation of $\mathbf{u}$ and **F** is $\mathbf{f}$ in the equation $\mathbf{Au} = \mathbf{f}$. In the current implementation of the multigrid method, the coarsest grid is when the grid is 2 by 2, *i.e.*, when the number of unknowns become one. At this time, the value of the unknown can be computed to the exact value.

The function **MultiV** is called by the function **FMV** which performs the FMV-cycle. In FMV-cycle, computation starts from the coarsest grid level and moves up one level at a time. At each higher (finer) grid level, a V-cycle is performed (Figure 3 (b)). Figure 5 is a SISAL implementation of the function **FMV**. This

```
function FMV (N,n1,n2:integer; V,F:TwoDim returns TwoDim)
   let
      Grid := if N = 2 then
         ExactSolve(F)
      else
         let
            Residue := ComputeRes(N,F,V);
            CoarseF := Restrict(N/2,Residue);
            CoarseUpdatedV := FMV(N/2,n1,n2,BndVal(N/2),CoarseF);
            UpdateV := InterP(N,CoarseUpdatedV);
            CorrectedV := Correction(N,V,UpdatedV)
         in
            MultiV(N,n1,n2,CorrectedV,F)
         end let
      end if
   in
      Grid
   end let
end function
```

Figure 5: The function **FMV** is a SISAL implementation of a recursive algorithm which performs a FMV-cycle.

function is also written in a recursive style and closely resembles the Algorithm *FMV* description in the previous section.

The functions **MultiV** and **FMV** call the following five functions. These are the core functions of the multigrid algorithm. A short description of each function is as follows :

- **Relax** : One of the iterative methods such as Jacobi, Gauss-Seidel, etc. This function is discussed in more detail in subsequent paragraphs.
- **ComputeRes** : This function computes *residual* $\mathbf{r}$, *i.e.* $\mathbf{r} = \mathbf{f} - \mathbf{A}\mathbf{v}$. This function contains (two-level) nested forall loops only.
- **Restrict** : This function performs an interpolation from a grid of size N to $N/2$. It is used in the downward path of the V-cycle and contains nested forall loops.
- **InterP** : This function performs an interpolation from a coarse grid of size $N/2$ to a fine grid of size N. This function also contains only forall loops.
- **Correction** : This function modifies the previously approximated unknown variables by adding the correction values. This function contains forall loops.

Two points were considered in deciding the kind of iterative method to be used for relaxation. The first consideration is the convergence rate. As discussed previously, SOR performs the best if ω_{opt} can be determined. Since this value cannot be determined in general, the next best choice is the Gauss-Seidel iteration. The second consideration is the amount of parallelism available in the algorithm. In this respect, Jacobi method has the most parallelism. In Jacobi iteration, a new approximation of a grid point is only a function of grid points from previous approximations. Therefore, all grid points can be updated in parallel. In Gauss-Seidel method, on the other hand, a new approximation of a grid point depends partly on the most recently approximated grid points. Due to this data dependency in the algorithm, Gauss-Seidel method is inherently sequential.

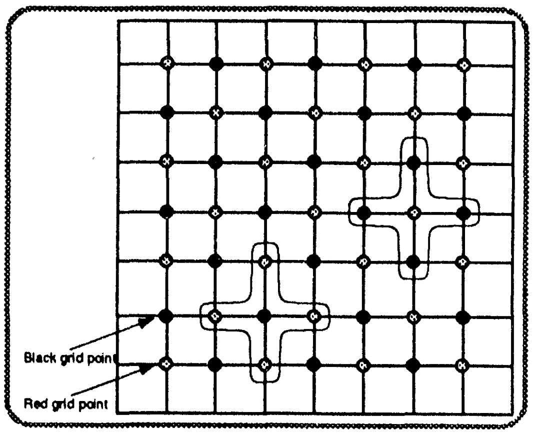


Figure 6: The Red-Black Gauss-Seidel is a parallel version of the otherwise sequential Gauss-Seidel iterative method. Notice the way the red and the black grid points are divided. The cross-like regions represent the 5-point stencil used in the approximation.

Fortunately, a parallel version of the Gauss-Seidel method exists. It is called the ***Red-Black*** (R-B) Gauss-Seidel method [9] and is shown in Figure 6. This method is not fully parallel as the Jacobi method. Instead, grid points are updated in two sequential steps. That is, one half of the grid points are updated first and then the other half are updated next. At each step, however, grid points can be updated in parallel. Although the amount of parallelism available in the Red-Black Gauss-Seidel method is only half that of the Jacobi method, its superior convergence rate (twice faster than Jacobi) makes it a better iterative scheme.

4 Experimental Results

In this section, we first discuss the programmability issue. We then describe the performance of the implemented program. First, the amount of parallelism existing in the SISAL implementation of a multigrid

Solvers	Develop. Time (days)
Basic iterative methods	29
Multigrid	23
Precond. Conj. Gradient	21
Domain Decomposition	30

Table 1: Time spent to learn and implement each PDE solvers presented in the course.

algorithm is analyzed. Then we present the actual performance of the program on a number of different parallel machines.

4.1 Programmability

Table 1 shows the development time of every PDE solver implemented during the course. The development time shown in the second column of the table includes the class lectures explaining the algorithms as well as the days spent in actual program development. On the average, half of the time was devoted to the discussion of the algorithm and the other half to the actual implementation. Although the first assignment consists of simple programs, additional time was needed to become familiar with writing SISAL programs.

In the case of the multigrid PDE solver, the SISAL implementation consists of approximately 350 source lines consisting of 17 functions. Two functions (`MultiV` and `FMV`) are written in a recursive style. There are 25 loops in the program in which three loops are written in a sequential loop construct and 22 loops are written in a parallel loop construct. Approximately two weeks (six lectures) were spent in discussing the algorithm and one week was spent in actual program development.

Once a working program is written, that same program was used for performance measurements on various parallel machines without any modifications. Therefore, on the average, a parallel PDE solver is written in two weeks which runs on various parallel machines in addition to average single processor workstations.

4.2 Performance

This section first discusses the amount of parallelism available in the SISAL implementation of the multigrid algorithm. Then actual performance measured by executing the program on different parallel machines is presented.

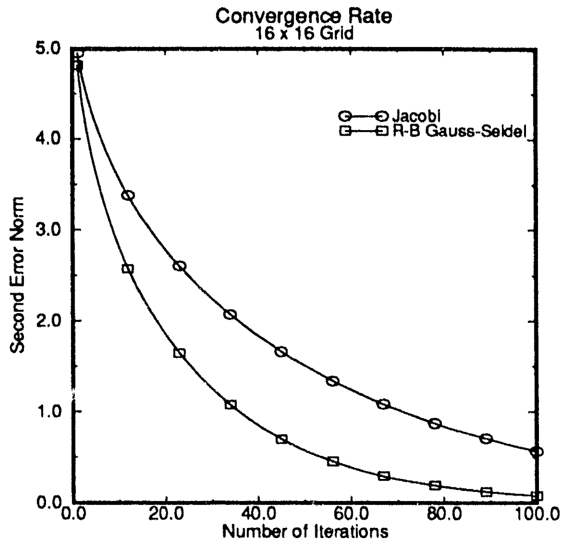


Figure 7: Red-Black Gauss-Seidel iteration has a superior convergence rate over that of the Jacobi iteration.

4.2.1 Parallelism Profile of the Multigrid Implementation

For performance measurements, an implementation of a full multigrid algorithm is used. In the program, the diffusion function of $k(x,y) = e^{(x+y)}$ is used. Throughout the measurements, the number of relaxations ν_1 in the downward path of the V-cycle is set to two and the number of relaxations ν_2 in the upward path of the V-cycle is set to one.

The most expensive operation in each grid level is the relaxation operation. As mentioned in the previous section, the Red-Black Gauss-Seidel relaxation scheme is used. This scheme has a superior convergence rate over that of the Jacobi relaxation while still providing parallelism. Figure 7 shows the convergence rate of the two schemes for a two dimensional grid of 16 by 16. The initial guess for the unknowns were set to zero for both schemes. Table 2 shows the execution time of the two iterations on a Silicon Graphics four processor machine. It shows that both schemes have a close to linear speedups indicating that both schemes contain enough parallelism. Note that the execution time per iteration of the Jacobi iteration is slightly faster. This is due to the fact that the Red-Black Gauss-Seidel iteration updates grid points in a two-step sequence while the Jacobi iteration does it in a single step.

The relaxation operation along with other operations such as interpolation, restriction, residue calculation and error correction is performed at every grid

512 x 512, 10 iterations					
Jacobi			R-B Gauss-Seidel		
PE	Exec.Time(sec)	Speedup	PE	Exec.time(sec)	Speedup
1	202.80	1	1	223.13	1
2	103.19	1.97	2	113.54	1.97
3	69.24	2.93	3	76.33	2.92
4	52.01	3.90	4	57.54	3.88

Table 2: Both the Jacobi and the Red-Black Gauss-Seidel iterations contain enough parallelism to provide close to linear speedups.

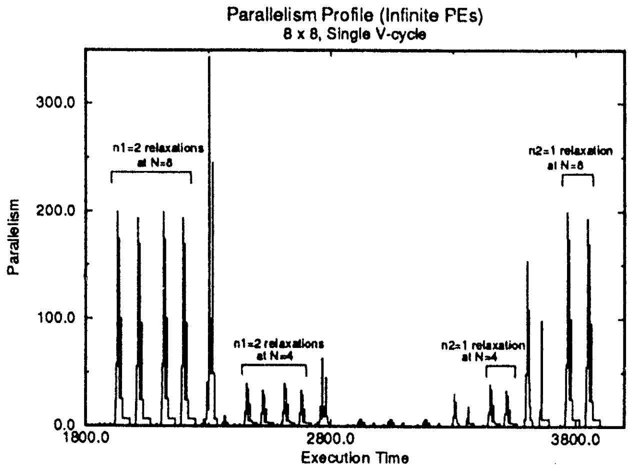


Figure 8: The ideal parallelism profile of a single V-cycle where the grid size is 8 by 8. $\nu_1 = 2$ and $\nu_2 = 1$. (See Algorithm *MV*)

level. Althouth there can be many variations of moving around different grid levels, they are all based on the V-cycle. Figure 8 shows the parallelism profile of the V-cycle for an 8 by 8 grid using the Red-Black Gauss-Seidel iteration as the relaxation scheme. The 2 by 2 grid is the coarsest grid in the V-cycle. The parallelism profile shown is for an ideal case which assumes infinite number of processors and no communication overhead.

The amount of parallelism is computed by counting the number of executable nodes at each time interval. The nodes are part of the intermediate-level representation of the program which is a directed acyclic graph called Intermediate Form 1 (IF1) [11]. The nodes represent instructions and the edges connecting the nodes represent data dependency relationships among the nodes. The reason that the parallelism profile looks like a cluster of impulses is because it is assumed that infinite number of processors is available. That is, all instructions that become executable are assumed to be executed together at the same time. The parallelism profile shows that parallelism decreases by one fourth until the grid size of 2 by 2 is reached and increases back to the original level. The reason the profile is not exactly symmetric is because the relaxation is performed twice going down the grid level ($\nu_1 = 2$), but only once coming up the grid level ($\nu_2 = 1$). Note that a single relaxation produces two spikes because in Red-Black Gauss-Seidel iteration, grid points are updated in two sequential steps.

In the current multigrid implementation, a full multigrid V-cycle (FMV-cycle) is used once in the initialization stage followed by regular V-cycles in which the number of repetitions is specified by an input parameter. An FMV-cycle starts from the coarsest grid and moves up to the finest grid. At each grid level, a V-cycle is performed. The parallelism profiles for an FMV-cycle is shown in Figure 9 for infinite processors. As expected, we see repeated V-cycle profiles of different sizes. The rightmost pattern is the V-cycle parallelism profile for an 8 by 8 grid. At far left, parallelism profile for a 2 by 2 grid can barely be seen.

Figure 13 shows the parallelism profile of the full multigrid scheme doing 5 iterations. As mentioned already, the first iteration is the FMV-cycle, and the next 5 iterations are the V-cycles.

4.2.2 Actual Performance on Parallel Machines

Performance of the multigrid implementation is measured on three MIMD type parallel computers. They are,

- CRAY Y-MP (4 PEs)
- Silicon Graphics (4 PEs)
- Sequent Balance (16 PEs)

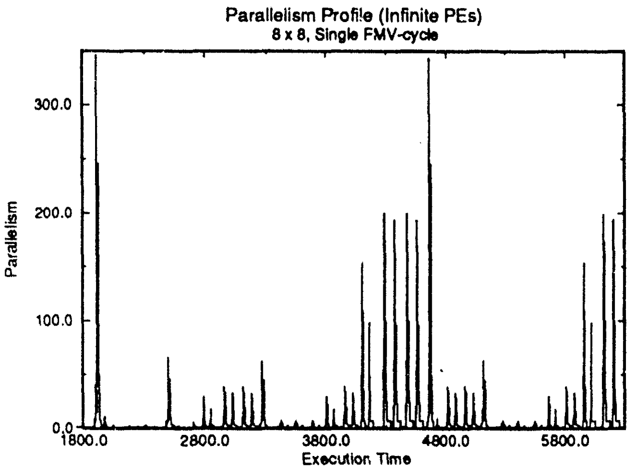


Figure 9: The ideal parallelism profile of a single FMV-cycle where the grid size is 8 by 8. $\nu_1 = 2$ and $\nu_2 = 1$. (See Algorithm *FMV*)

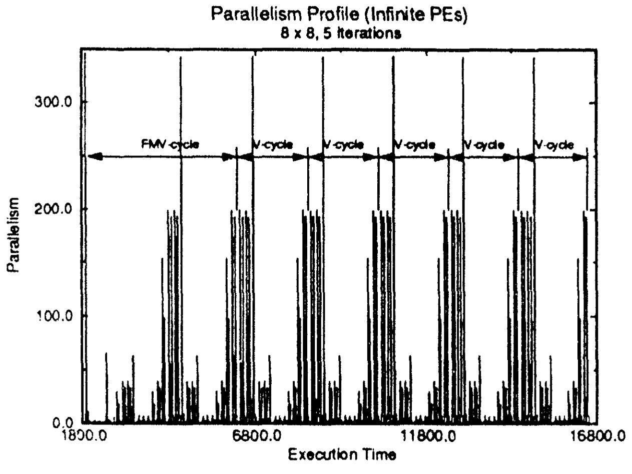


Figure 10: The ideal parallelism profile of a full multigrid scheme doing 5 iterations. The grid size is 8 by 8. $\nu_1 = 2$ and $\nu_2 = 1$.

512 x 512, 5 iterations		
PE	CRAY (sec.)	SGI (sec.)
1	15.89	457.78
2	8.76	287.14
3	6.39	192.69
4	5.34	146.00

Table 3: Execution time of a multigrid program on a CRAY Y-MP and a Silicon Graphics machines.

256 x 256, 5 iterations	
PE	Balance (sec.)
1	3989.20
2	1993.17
4	1014.61
8	516.11
12	362.29
16	274.89

Table 4: Execution time of a multigrid program on a Sequent Balance Machine.

CRAY Y-MP runs UNICOS which is a Unix-like operating system and has vector execution units in each processor. The Silicon Graphics machine is also a Unix-based and is built on MIPS R3000 processor chips. Sequent Balance utilizes National Semiconductor's NS32032 processor chip and is a slow machine by today's standard.

In the performance measurement, a grid size of 512 by 512 is used and each run consists of 5 iterations. Note that by 5 iterations we actually mean one FMV-cycle followed by five V-cycles. Since the full multigrid scheme utilizes a FMV-cycle with V-cycles, it is helpful to measure the speedup of these two cycles separately before measuring the speedup of the whole program. Figure 11 shows a graph which compares the speedup of one FMV-cycle and one V-cycle. It shows that V-cycle results in a better speedup. This is expected since FMV-cycle spends some time performing V-cycles at coarser grids. This results in less processor utilization and thus produces lower speedup. The graph shows that V-cycle results in a close to linear speedup.

From Figure 11, we expect the speedup of the multigrid implementation to be somewhere between the speedup reached by the FMV-cycle and the V-cycle. Figure 12 shows the speedup of the multigrid implementation on CRAY Y-MP and Silicon Graphics machines. Figure 13 shows the speedup on a Sequent Balance. Tables 3 and 4 show the execution times.

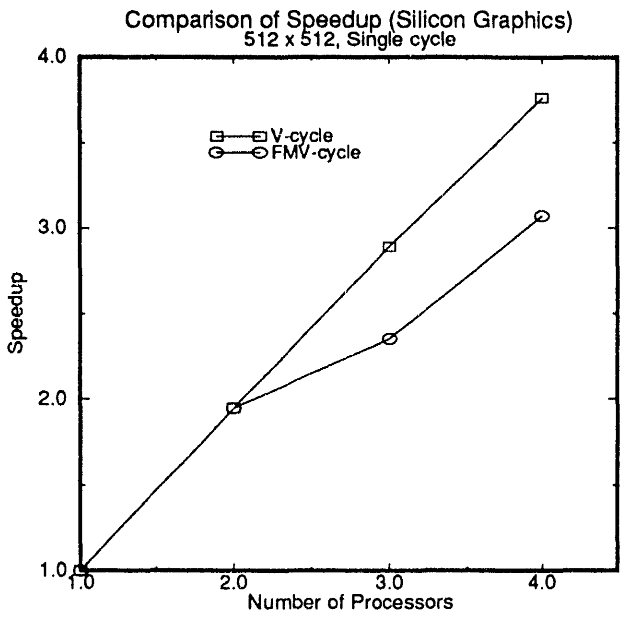


Figure 11: Speedup comparison of a single FMV-cycle and a single V-cycle: 512 by 512 grid.

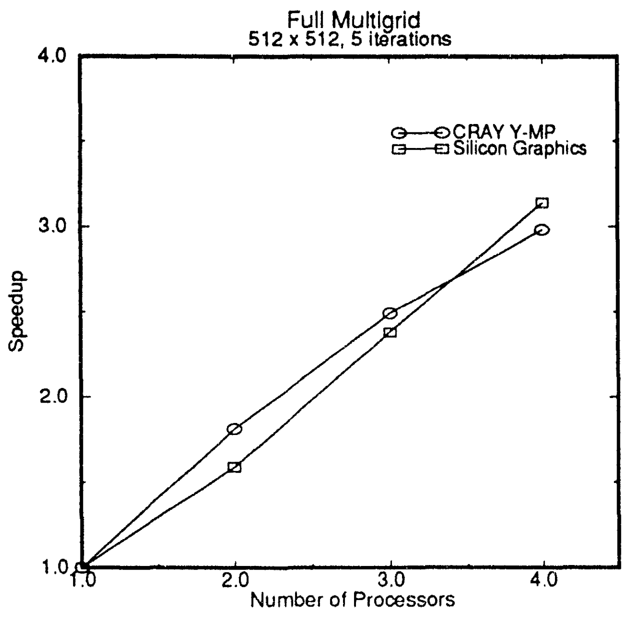


Figure 12: Speedup of a full multigrid implementation on a 512 by 512 grid.

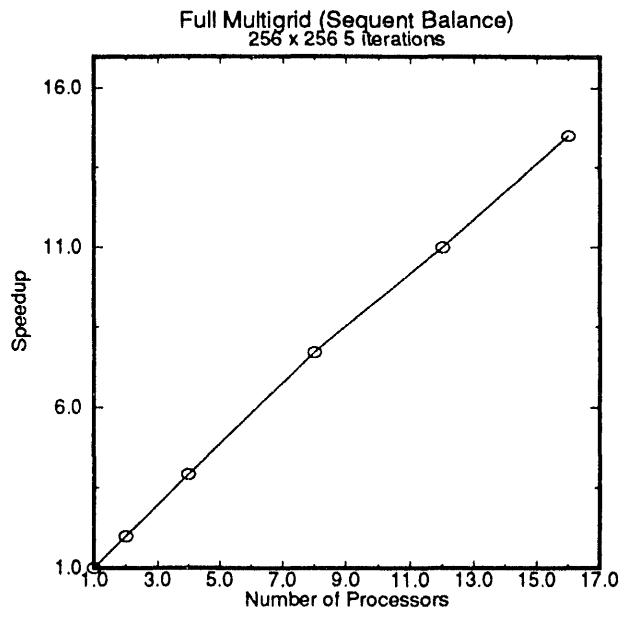


Figure 13: Speedup of a full multigrid implementation on a 256 by 256 grid.

We see that the OSC compiler does a good job of concurrentization by observing the speedup curve of each machine. In addition to concurrentization, CRAY also utilizes hardware vector facilities. The OSC recommends innermost parallel loops for vectorization. In the multigrid implementation used in the performance measurements, the OSC recommended vectorization of 12 loops in which all were vectorized by the native C compiler. A parallel loop in function **Norm** which has reduction operations (**sum** and **greatest**) in the return clause were neither recommended by the OSC nor vectorized by the CRAY C compiler.

5 Conclusion

We have shown that functional programming is indeed a viable approach to parallel computing providing both programmability and performance. Our program which has originally been written for a sequential machine efficiently executed on a number of parallel machines without requiring the programmer to manually parallelize the code. The implicit parallelism of SISAL, therefore, has allowed the programmer to concentrate on the implementation of the algorithm without having to worry about low-level execution details. Once a program is verified to work correctly on a sequential machine, it can be run on various parallel machines without program modification.

However, the current version of Osc is tailored for execution on a *shared global address space* machines which currently employ a relatively small number of processors (< 30). On the other hand, there is a growing number of parallel machines already introduced or being introduced which employ a large number of processors (in the hundreds). Logically, some machines have shared global address space and some do not. Physically, however, all large machines have distributed memory spread across the processors making memory accesses nonuniform and latency a serious issue to consider. To achieve good performance on such machines, a new execution model needs to be developed for the next generation of the SISAL compiler.

Acknowledgements

We would like to thank our colleagues, especially Dr. John Feo at the Computing Research Group of the Lawrence Livermore National Laboratory for letting us use the CRAY and the Silicon Graphics parallel machines. We also thank Professor Walid Najjar of Colorado State University for letting us use the Sequent Balance machine.

References

[1] W. Briggs. *A Multigrid Tutorial.* SIAM, 1987.

[2] D. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, 1989.

[3] D. Cann. *The Optimizing SISAL Compiler: Version 12.0.* Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, CA 94550, 1992.

[4] D. Cann. Retire Fortran: A debate rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[5] J. Feo, D. Cann, and R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10:349–366, December 1990.

[6] G. Golub and C. van Loan. *Matrix Computations.* Johns Hopkins University Press, 1989.

[7] M. Kallstrom and S. Thakkar. Programming three parallel computers. *IEEE Software*, pages 11–22, January 1988.

[8] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. *SISAL Language Reference Manual Version 1.2*, March 1985.

[9] J. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems.* Frontiers of Computer Science. Plenum Press, 1988.

[10] A. Osterhaug. *Guide to Parallel Programming on Sequent Computer Systems.* Sequent Computer Systems, Inc., second edition, 1987.

[11] S. Skedzielewski and J. Glauert. *IF1 An Intermediate Form for Applicative Languages.* Computing Research Group, Lawrence Livermore National Laboratory, P.O. Box 808, L-306, Livermore, CA 94550, 1985.

Implementing the Kernel of the Australian Region Weather Prediction Model in SISAL

G.K. Egan
Laboratory for Concurrent Computing Systems
Swinburne University of Technology
John Street, Hawthorn 3122, Australia

Abstract

The SISAL implicit parallel programming language has been implemented on a number of platforms ranging from scientific workstations through medium cost multiprocessors to high end parallel super computers and recently massively parallel processors. No changes to source code are required to obtain good performance across these platforms and it has been claimed that SISAL exhibits similar uniprocessor performance to FORTRAN while providing significant speedup compared to FORTRAN on multiprocessors.

The Australian Region Weather Prediction Model is an experimental FORTRAN code which uses a variable resolution nesting scheme to provide higher resolution predictions over important areas of the Australian continent such as cities and coastal fisheries. In this preliminary study we explore the performance of the SISAL implicit parallel programming language on a significant scientific application by recoding the kernel subroutine of the Model in SISAL. Results are presented for a low end SPARC workstation, an entry level Cray Y-MP EL and a high end Cray C90.

1 Introduction

The Australian Region Weather Prediction Model (ARPE) was developed by the Australian Bureau of Meteorology Research Centre [1] for short-term weather forecasting up to 36 hours. ARPE draws upon the work of Arakawa, Lamb and Miyakoda [2][3] for its formulation and is intended to be a production code for the prediction of weather over the Australian region. This paper will concentrate on the implementation of the core subroutine of the ARPE in the SISAL language and readers are directed to reference [1] for a detailed description of the model. The work is part of a continuing long term international study of SISAL being conducted in collaboration with the Lawrence Livermore National Laboratory.

2 The SISAL language

SISAL is a functional language for numerical computation [4]. The developers of SISAL have been able to demonstrate performance comparable with FORTRAN on a number of computing platforms including the Cray Research multiprocessors [5].

SISAL prohibits by design the ability to express constructions which lead to the side effects that make compilation for parallel computer systems extremely difficult. Examples of side effects include those which occur through the COMMON and EQUIVALENCE statements in FORTRAN and SISAL has neither of these constructs. SISAL is block structured and superficially resembles a number of modern languages. The single assignment nature of SISAL means variables have values assigned to them once. This requires some departure from a common style of programming where variables are re-used in programs sometimes for unrelated computations. Translation of FORTRAN programs into SISAL is not necessarily a simple process and can be complicated significantly if the program being re-expressed has been the subject of undisciplined maintenance or construction. This may be compounded if there is no original formulation of the mathematical model available. Direct transliteration of well written FORTRAN code can yield satisfactory results.

Most comparative studies to date have involved the complete recoding of an application in SISAL. In this study the mixed language facility of the current (V12.9.1) Optimising SISAL Compiler is used with an initial core subroutine being recoded.

3 The weather prediction model

The Weather Prediction Model code (ARPE) consists of some 10,000 lines of FORTRAN source code. Its pre-processors and ancillary code constitute perhaps another 5,000 lines of code. The code is generally well written with disciplined use of COMMON and EQUIVALENCE statements. The kernel routines make almost no use of subroutines although the structure of the code suggests they should be used. ARPE then is a reasonable example of a code where inlining has occurred from the outset in an attempt to obtain improved performance. It predates modern FORTRAN pre-processors which automatically inline selected subroutines.

4 FORTRAN

The Cray Research FORTRAN tool suite used [6] runs under X Windows and is a marked advance on those generally available only a few years ago. The tool set comprises: a profiler (flowview) which identifies key subroutines and subroutines which are candidates for inlining; a pre-processor which performs inlining and attempts to identify and annotate parallel regions; an assistant for explicit parallel annotation (atscope); and a parallelism estimator (atexpert).

Routine Name	Tot Time	Calls	Avg Time	Percentage	Accum%
INNER2	2.52E+01	9	2.80E+00	42.24	42.24
LIE	1.09E+01	24	4.53E-01	18.23	60.47
PHYS	6.15E+00	5	1.23E+00	10.31	70.79
LIEBIG	5.61E+00	12	4.68E-01	9.42	80.21
LIEH	5.50E+00	12	4.58E-01	9.23	89.44
LIEBH	1.69E+00	9	1.88E-01	2.84	92.28
SEMIMP	1.48E+00	9	1.64E-01	2.48	94.76
VMODES	1.08E+00	4	2.71E-01	1.82	96.57
INNER	9.56E-01	9	1.06E-01	1.60	98.18
DADADJ	4.40E-01	11470	3.84E-05	0.74	98.91
LAMLL	1.43E-01	2600	5.50E-05	0.24	99.15

Table 1: Execution Profile (5 iterations Y-MP EL)

The original program was profiled using flowtrace to identify the core subroutines. For reasons already stated flowtrace did not identify any subroutines eligible for inlining.

The INNER2 subroutine was chosen as the starting point for this study but as it represents only 42% of the run time contribution no significant speedup is to be expected. The LIE and PHYS subroutines will be translated in due course. Our interest here is to confirm that the run time is not adversely affected and that underlying concurrency is uncovered by the OSC compiler.

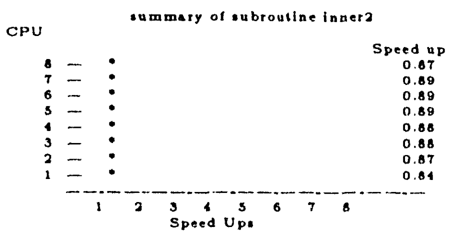


Figure 1: speedup of INNER2 predicted by atexpert

4.1 Results for FORTRAN

The automatic parallel annotator was used to annotate the INNER2 subroutine. No attempt was made to resolve data dependencies in the original FORTRAN in this part of the study although this is intended later. The atexpert measurement tool was used to examine individual DO loops for predicted speedup. Atexpert is claimed to accurately predict performance for dedicated systems. The tool provides parallelism profiles and allows routines associated with parallel or sequential regions to be examined and analysed interactively.

It can be seen in Figure 1 that fpp failed to discover significant parallel regions in INNER2.

5 SISAL

5.1 Mixed language compilation

The osc compiler compiles and links modules written in FORTRAN and SISAL. In this FORTRAN is invoking a SISAL function. To do this the original INNER2 subroutine was replaced by a FORTRAN shell. The shell initialises the array descriptors required by SISAL and calls the replacement INNER2 written in SISAL [7].

Fortunately the array descriptors may be re-used for other arrays which have an identical shape. The ability to specify an offset for returned data structures could be used to avoid the often clumsy process of dealing with boundary values. The current descriptor mechanism unfortunately sets to zero the elements not written to.

5.2 The transliteration process

Although the mathematical formulation was available it did not provide significant assistance in the transliteration process. The INNER2 subroutine was directly transliterated into SISAL with no restructuring being attempted. A number of unintentional out of bound accesses were discovered in the FORTRAN program during this transliteration.

The transliteration process was significantly complicated by the size of the INNER2 subroutine. While the SISAL debugger (sdbx) gave some assistance there were many cases where sdbx was not able to determine the original source line causing the error. Other minor difficulties which would case irritation for programmers used to imperative styles also arose. In this case even though the author has a reasonable understanding of SISAL the passage of time since writing his previous SISAL program still led him to be caught by the following:

```
for initial
  ....
  k:=0;
while k < kz repeat
  k:= old k +1;
returns ......u[k].....
```

Most programmers will expect k to be 1 when the variable u is accessed on the first loop iteration rather than zero as stated by the for initial clause.

Transliteration and debugging took approximately 35 hours.

5.3 Results for SISAL

The results for one call of INNER2 in FORTRAN and SISAL are shown in Table 2. In their current form both versions are several hundred lines long and the interleaving of initialisation, the calculation of primary meteorological variables and common working variables makes their inner workings difficult to comprehend (Appendices).

Language	Sparc	EL (1-cpu)	C90 (1-cpu)	C90 (4-cpu)
f77 -O	6.6+0.7			
cf77 -Zp		3.01+0.48	0.39+0.01	
osc -O	7.2+1.0	6.57+0.25	1.08+0.01	0.29+0.01

Table 2: Run Times for FORTRAN and SISAL

It may be noted that although the run times on the SPARC workstation for FORTRAN and SISAL are comparable performance on the Cray systems is not as good. It is believed that the transliteration resulted in a SISAL style which caused difficulty for the SISAL optimisers; this is currently being resolved.

6 Conclusions

A modest amount of difficulty was encountered in the transliteration of the kernel INNER2 subroutine into SISAL. The run time for this first SISAL implementation relative to FORTRAN is acceptable. Good speedup has been achieved with the SISAL version's runtime falling below that for FORTRAN at four processors. Given this promising start the study will now refine the version of INNER2 and move to the other dominant kernel subroutines LIE and PHYS. The PHYS subroutine is dominated by conditionally executed code as are many other weather codes. It is anticipated that this will produce a more demanding test for SISAL.

Acknowledgements

The author thanks the Australian Bureau of Meteorology Research Centre for access to the ARPE code. The author also thanks the members of the Laboratory for Concurrent Computing Systems for their contributions to the work presented in this paper.

Appendices

INNER2.F

The original code of INNER2 has been stripped out and replaced with descriptor initialisation and call to sinner2.

```
      SUBROUTINE INNER2
C
C     INNER2 CALCULATES THE RH SIDES OF THE MAIN SEMI-
IMPLICIT EQUATIONS
C
      include 'arpe.inc'
      PARAMETER
     + (
     +  I2=I1+1, I3=I1+2  I4=I1+3, ILM=IL-1, ILN=IL-2
     +  ,J2=J1+1, J3=J1+2, J4=J1+3, JLM=JL-1, JLN=JL-2
     +  ,KZM1=KZ-1, KZP1=KZ+1
     +  ,CP=1.00464E7, G=980.6, HL=2.501E10, PBAR=1.E6, R=2.87E6
     +  ,RV=4.61E+6
     + )
C
      COMMON
    +    /DTDS/  DT,DS,DTI,DSI,DSI2,DSSQ,TDSI,HDTDS,BET65,DTMAX
        +          ,/INTGRL/ PRECP, PRECTA, CKS, EKE, PE, PS-
BAR, TRHAT, VROMG
     +     ,/KTAU/  KTAU
C
      COMMON
     +      /CDIFF/  CDIFF(IL,JL)
     +     ,/CORP/   CORP(IL,JL)
     +     ,/DNORM/  DNORM(KZ)
     +     ,/DQ/     DQ(KZ)
     +     ,/DTODQ/  DTODQ(KZ)
     +     ,/EM/     EM(IL,JL)
     +     ,/EMSQ/   EMSQ(IL,JL)
     +     ,/EMSQI/  EMSQI(IL,JL)
     +     ,/GAMA/   GAMA(KZ)
     +     ,/OMEGA/  OMEGA(KZ,IL,JL)
     +     ,/PHI/    PHI(KZ,IL,JL)
     +     ,/PS/     PSM(IL,JL), PS(IL,JL), PSP(IL,JL)
     +     ,/Q/      Q(KZ)
     +     ,/QPH/    QPH(KZ)
      COMMON
     +      /RM/     RMM(KZ,IL,JL), RM(KZ,IL,JL), RMP(KZ,IL,JL)
     +     ,/RTBAR/  RTBAR(KZ)
     +     ,/SIGDOT/ SIGDOT(KZ,IL,JL)
     +     ,/T/      TM(KZ,IL,JL), T(KZ,IL,JL), TP(KZ,IL,JL)
     +     ,/TBAR/   TBAR(KZ)
     +     ,/U/      UM(KZ,IL,JL), U(KZ,IL,JL), UP(KZ,IL,JL)
     +     ,/V/      VM(KZ,IL,JL), V(KZ,IL,JL), VP(KZ,IL,JL)
     +     ,/ZS/     ZS(IL,JL)
C
      REAL Q
      integer ik(100),iij(100),ikij(100)
      DIMENSION RMPR(KZ,IL,JL)
      DIMENSION TFLEV(KZ),DTFDQ(KZ),WVEL(KZ)
      DIMENSION VADVU(KZP1),VADVV(KZP1),VADVRM(KZP1)
      DATA VADVU/KZP1*0./,VADVV/KZP1*0./,VADVRM/KZP1*0./
      DATA OMG / 0.0 /
C
c SISAL array descriptors
c one dimension
      ik(1)=0
      ik(2)=0
      ik(3)=0
c
      ik(4)=1
      ik(5)=kz
      ik(6)=1
      ik(7)=kz
      ik(8)=1
c
c two dimensions
      iij(1)=0
      iij(2)=0
      iij(3)=0
c
      iij(4)=i1
      iij(5)=il
      iij(6)=i1
      iij(7)=il
      iij(8)=1
c
      iij(9)=j1
      iij(10)=jl
      iij(11)=j1
      iij(12)=jl
      iij(13)=1
c
c three dimensions
      ikij(1)=0
      ikij(2)=0
      ikij(3)=0
c
      ikij(4)=1
      ikij(5)=kz
      ikij(6)=1
      ikij(7)=kz
      ikij(8)=1
c
      ikij(9)=i1
      ikij(10)=il
      ikij(11)=i1
      ikij(12)=il
      ikij(13)=1
c
      ikij(14)=j1
      ikij(15)=jl
      ikij(16)=j1
      ikij(17)=jl
      ikij(19)=1
c
      call sinner2(
     +dt,ds,dsi,dsi2,tdsi,dtmax,cks, eke, pe, psbar, trhat, vromg,
     +ktau,
     +cdiff,iij,
     +corp,iij,
     +dnorm,ik,
     +dq,ik,
     +dtodq,ik,
     +em,iij,
     +emsq,iij,
     +emsqi,iij,
     +gama,ik,
     +omega,ik,
     +phi,ikij,
     +psm, iij,ps,iij,
     +q,ik,
     +qph,ik,
     +rmm, ikij,rm, ikij,rmp,ikij,
     +rtbar,ik,
     +sigdot,ikij,
     +tm,ikij, t,ikij, tp,ikij,
     +tbar,ik,
     +um, ikij,u, ikij,up,ikij,
     +vm, ikij,v, ikij,vp,ikij,
     +zs ,iij,
c     returns
     +rm,ikij,
     +sigdot,ikij,
     +up,ikij,
     +new'tp,ikij,
     +new'rmp,ikij,
     +vp,ikij,
     +eke,
     +cks,
     +trhat,
     +pe,
     +psbar,
     +vmrong)
c
      RETURN
      END
```

inner2.sis

```
define sinner2

% G.K. Egan 1993

type OneDReal = array[real];
type TwoDReal = array[OneDReal];
type ThreeDReal = array[TwoDReal];

global log(a:real returns real)

global sqrt(a:real returns real)

function boundary'ceil(i,i1,il,j,j1,jl:integer returns boolean)
  ((i = i1)—(i = il)—(j = j1)—(j = jl))
end function

function divergence'sums(
          i,j,kz,i1,il,j1,jl:integer; dsi:real;
          dq:OneDReal;u,v,t:ThreeDReal;emsq:TwoDReal
        returns
          real, real, real, OneDreal, OneDReal,
          OneDReal, OneDReal, OneDReal)
  for initial
    sumu:=0.0;
    sumv:=0.0;
    sumx:=0.0;
    k:=1;
  while (k < kz) repeat
    k:=old k +1;
    sumu, sumv, sumx := (
    if boundary'cell(i,i1,il,j,j1,jl) then
      old sumu, old sumv, old sumx
    else
      old sumu+dq[k]*(u[k,i+1,j]-u[k,i-1,j]
        +v[k,i,j]+v[k,i+1,j]-v[k,i,j-1]-v[k,i+1,j-1]),
      old sumv+dq[k]*(u[k,i,j]+u[k,i,j+1]
        -u[k,i-1,j]-u[k,i-1,j+1]+v[k,i,j+1]-v[k,i,j-1]),
      old sumx+dq[k]*(u[k,i,j]-u[k,i-1,j]+v[k,i,j]-v[k,i,j-1])
    end if)
  returns
    value of sumu
    value of sumv
    value of sumx
    array of sumu
    array of sumv
    array of sumx
    array of (-emsq[i,j]*sumx*dsi)
    array of t[k,i,j]
  end for
end function

function sinner2(
          dt,ds,dsi,dsi2,tdsi,dtmax,cks, eke, pe, psbar, trhat, vromg:real;
          ktau:integer;
          cdiff:TwoDReal;
          corp:TwoDReal;
          dnorm:OneDReal;
          dq:OneDReal;
          dtodq:OneDReal;
          em:TwoDReal;
          emsq:TwoDReal;
          emsqi:TwoDReal;
          gama:OneDReal;
          omega:OneDReal;
          phi:ThreeDReal;
          psm, ps:TwoDReal;
          q:OneDReal;
          qph:OneDReal;
          rmm, rm, rmp:ThreeDReal;
          rtbar:OneDReal;
          tm, t, tp:ThreeDReal;
          tbar:OneDReal;
          um, u, up:ThreeDReal;
          vm, v, vp:ThreeDReal;
          zs:TwoDReal
        returns
          ThreeDReal,%new'rm
          ThreeDReal,%new'sigdot
          ThreeDReal,%new'up
          ThreeDReal,%new'tp
          ThreeDReal,%new'rmp
          ThreeDReal,%new'vp
          real,%new'eke
          real,%new'cks
          real,%new'trhat
          real,%new'pe
          real,%new'psbar
          real%new'vmrong
         )

 let
```

```
kz:=15;
il:=65;
jl:=40;
i1:=1;
j1:=1;
i2:=i1+1;
i3:=i1+2;
i4:=i1+3;
ilm:=il-1;
j2:=j1+1;
j3:=j1+2;
j4:=j1+3;
jlm:=jl-1;
kzm1:=kz-1;
kzp1:=kz+1;
cp:=1.00464e7;
g:=980.6;
hl:=2.501e10;
pbar:=1.e6;
r:=2.87e6;
rv:=4.61e+6;

dtt:=
  if (ktau = 1) then
    dt
  else
    2.0 *dt
  end if;

new'rm, rmpr:=
for k in 1,kz cross i in i1,il cross j in j1,jl
  t'rm,
  t'rmpr:=
    if (rm[k,i,j] > 0.0) then
      rm[k,i,j],
      rm[k,i,j] / (ps[i,j]+pbar)
    else
      0.0,
      0.0
    end if
returns
  array of t'rm
  array of t'rmpr
end for;

dmonp:=0.0;
emonp:=0.0;

new'tp, new'up, new'vp, new'rmp, new'sigdot,
new'eke, new'cks, new'trhat,
new'pe, new'psbar, new'romg:=
for i in i1,il cross j in j1,jl

  psijc:=ps[i,j]+pbar;
  psijci:=1.0/psijc;
  corf1,corf2:=

  if boundary'cell(i,i1,il,j,j1,jl) then
    0.0,0.0
  else
    0.125*(corp[i,j]+corp[i+1,j]),
    0.125*(corp[i,j]+corp[i,j+1])
  end if;

  emthad := emsq[i,j]*psijci*tdsi;
  em2tps := emthad*psijci*r / cp;

  emhad1,emhad2:=
  if boundary'cell(i,i1,il,j,j1,jl) then
    0.0,0.0
  else
    0.25*tdsi*(em[i,j]+em[i+1,j])
    0.25*tdsi*(em[i,j]+em[i,j+1])
  end if;

  amonp:=emonp; % 0.0 then cycle'emonp
  bmonp:=dmonp; % 0.0 then cycle'dmonp

  cycle'dmonp, fmonp:=
  if boundary'cell(i,i1,il,j,j1,jl) then
    0.0,0.0
  else
    em[i+1,j]/(ps[i+1,j]+pbar),
    em[i,j+1]/(ps[i,j+1]+pbar)
  end if;

  new'bmonp:=
  if (i = i2) & (~((j = j1)—(j = jl))) then
    em[i,j]*psijci
  else
    cycle'dmonp
  end if;

  new'amonp:=
  if (i = i2) & (~((j = j1)—(j = jl))) then
    0.25*(new'bmonp+fmonp+em[i-1,j] / (ps[i-1,j]+pbar)
     +em[i-1,j+1] / (ps[i-1,j+1]+pbar))
  else
    emonp
  end if;

  cmonp:=new'bmonp+cycle'dmonp;
  cycle'emonp, new'cmonp:=
  if boundary'cell(i,i1,il,j,j1,jl) then
    0.0, cmonp
  else
    0.25*(cmonp+em[i+1,j+1]/(ps[i+1,j+1]+pbar)+fmonp),
    0.25* (cmonp+em[i+1,j-1] / (ps[i+1,j-1]+pbar)
      +em[i,j-1] / (ps[i,j-1]+pbar))
  end if;

  pse, psn:=
  if boundary'cell(i,i1,il,j,j1,jl) then
    0.0, 0.0
  else
    0.5*(ps[i,j]+ps[i+1,j])+pbar,
    0.5*(ps[i,j]+ps[i,j+1])+pbar
  end if;
%
%  extra variables for evaluating p.grad terms logarithmically
%
  psrmi, psrmj, psldi, psldj, zspdi, zspdj, emudsi, emvdsi, emrdsi:=
  if boundary'cell(i,i1,il,j,j1,jl) then
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
  else
    ps[i+1,j]-ps[i,j]-psm[i+1,j]+psm[i,j],
    ps[i,j+1]-ps[i,j]-psm[i,j+1]+psm[i,j],
    log(ps[i+1,j]+pbar)-log(psijc),
    log(ps[i,j+1]+pbar)-log(psijc),
    pbar*(zs[i+1,j]-zs[i,j]),
    pbar*(zs[i,j+1]-zs[i,j]),
    emsq[i,j]/(4.0*ds*pse),
    emsq[i,j]/(4.0*ds*psn),
    emsq[i,j]/(4.0*ds*psijc)
  end if;
%
%  compute total divergence
%
  sumu, sumv, sumx, vadvu, vadvv, vadvrm, wvel, tflev:=
    divergence'sums(i,j,kz,i1,il,j1,jl,dsi,dq,u,v,t,emsq);

  sigdot'k:=
  for k in 1,kz
  returns array of (
    if (k = 1) then
      0.0
    else
      wvel[k-1]-qph[k-1]*wvel[kz]
    end if)
  end for;

  vadvrm'k, vadvu'k, vadvv'k:=
  for l in 1,kz
    ll:=l+1;
    t'vadvrm, t'vadvu, t'vadvv :=
    if (l = kz) then
      0.0, 0.0, 0.0
    else
      emrdsi*(qph[ll]*sumx-vadvrm[ll])
       *(new'rm[ll,i,j]+new'rm[l,i,j]
       +2.0*sqrt (new'rm[ll,i,j]*new'rm[l,i,j])),
      emudsi*(qph[ll]*sumu-vadvu[ll])
       *(u[ll,i,j]+u[l,i,j]),
      emvdsi*(qph[ll]*sumv-vadvv[ll])
       *(v[ll,i,j]+v[l,i,j])
    end if
  returns
    array of t'vadvrm
    array of t'vadvu
    array of t'vadvv
  end for;
%
%  set up temperature difference terms
%
  dtfdq:=
  for k in 1, kz
  returns array of (
    if ((k = 1)—boundary'cell(i,i1,il,j,j1,jl)) then
      0.0
    else
      if (k = kz) then
        dtmax*(tflev[kz]-tflev[kzm1])+dtodq[kz]
      else
        0.5*(tflev[k+1]-tflev[k-1]) / dq[k]+dtodq[k]
      end if
    end if)
  end for;
%
 psmue, psmuw, psmun, psmus, psmu, psmvn, psmvs, psmve, psmvw, psmv:=
  if ((j = j2)—boundary'cell(i,i1,il,j,j1,jl)) then
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
  else
    if (i = ilm) then
      1.5*psm[il,j]-0.5*psm[ilm,j]+pbar
    else
      0.5*(psm[i+1,j]+psm[i+2,j])+pbar
    end if,

    0.5*(psm[i-1,j]+psm[i,j])+pbar,
    0.5*(psm[i,j+1]+psm[i+1,j+1])+pbar,
    0.5*(psm[i,j-1]+psm[i+1,j-1])+pbar,
    0.5*(psm[i,j]+psm[i+1,j])+pbar,

    if (j = jlm) then
      1.5*psm[i,jl]-0.5*psm[i,jlm]+pbar
    else
      0.5*(psm[i,j+1]+psm[i,j+2])+pbar
    end if,
      0.5*(psm[i,j-1]+psm[i,j])+pbar,
      0.5*(psm[i+1,j]+psm[i+1,j+1])+pbar,
      0.5*(psm[i-1,j]+psm[i-1,j+1])+pbar,
      0.5*(psm[i,j]+psm[i,j+1])+pbar
    end if;
%
%   commence vertical level loop
%
    tp'k, up'k, vp'k, rmp'k, omega'k,
    new'eke, new'ppe, new'pvromg, new'ptrhat:=
    for k in 1,kz
%
%     compute vertical advection contribs. in rhs of mtm. = ns.
%
%     compute horizontal advection terms associated with rhs of mtm. = ns.
%
      up'k:=
      if ((j = j2)—boundary'cell(i,i1,il,j,j1,jl)) then
        up[k,i,j]
      else
        let
          vat1:=
          if (k = kz) then
            0.0 %gke
          else
            -(vadvu[k+1]-vadvu[k])/dq[k]
          end if;
          ub:=u[k,i,j]+u[k,i-1,j];
          uc:=u[k,i,j]+u[k,i,j-1];
          ud:=u[k,i,j]+u[k,i+1,j];
          ue:=u[k,i,j]+u[k,i,j+1];
          vb:=v[k,i,j]+v[k,i,j-1];
          vc:=v[k,i,j-1]+v[k,i+1,j-1];
          ve:=v[k,i,j]+v[k,i+1,j];
          hadv1:=emhad1* ( ud*ud*cycle'dmonp-ub*ub*new'bmonp
           +ue*ve*cycle'emonp-uc*vc*new'cmonp);
%
%          compute pressure gradient terms on rhs of mtm. = ns.
%          logarithmically
%
          pg1:= ((pse-pbar)*(phi[k,i+1,j]-phi[k,i,j])
           +zspdi-rtbar[k]*psrmi
           +pse*psldi*( r*0.5* (t[k,i+1,j]+t[k,i,j])
           +rtbar[k]))*dsi;
          ct1:= corf1*(vc+ve);
          ubdiff:=
          if (dnorm[k]  =  0.0) then
            0.0
          else
            cdiff[i,j]*dnorm[k]*dsi2
             *( um[k,i+1,j]/psmue+um[k,i-1,j]/psmuw
             +um[k,i,j+1]/psmun+um[k,i,j-1]/psmus
             -4.0*um[k,i,j]/psmu )*psmu
          end if
        in
          (ct1+vat1-hadv1-pg1+ubdiff)*dt+um[k,i,j]
        end let
      end if;
%
      t'rmp'k, tp'k, omega'k:=
      if ((i = i2)—boundary'cell(i,i1,il,j,j1,jl)) then
        rmp[k,i,j],
        tp[k,i,j],
        omega[k]
      else
%
%       calculate horizontal advection term in the temp. = ation
%
        let
          wvelav:=
          if (k > 1) then
            0.5*(wvel[k-1]+wvel[k])
          else
            0.5*wvel[1]
          end if;
          thadv1:=  u[k,i,j]*(t[k,i+1,j]-t[k,i,j])
           +u[k,i-1,j]*(t[k,i,j]-t[k,i-1,j]);
          thadv2:=  v[k,i,j]*(t[k,i,j+1]-t[k,i,j])
           +v[k,i,j-1]*(t[k,i,j]-t[k,i,j-1]);
          thadv:= emthad*(thadv1+thadv2),
          tfull:= tflev[k]+tbar[k];
          phadv1:=  u[k,i,j]*(ps[i+1,j]-ps[i,j])
           +u[k,i-1,j]*(ps[i,j]-ps[i-1,j]);
          phadv2:=  v[k,i,j]*(ps[i,j+1]-ps[i,j])
           +v[k,i,j-1]*(ps[i,j]-ps[i,j-1]);
          t'omg:= em2tps*(phadv1+phadv2);
          tpstar:= t'omg*tfull;
          omg:= wvelav+q[k]*psijc*t'omg*cp / r;
          gamapr:= (r / cp)*tfull / q[k]-dtfdq[k];
          thdiff:=
          if (dnorm[k]  =  0.0) then
            0.0
          else
            cdiff[i,j]*dnorm[k]*dsi2
             *( tm[k,i+1,j]+tm[k,i-1,j]
             +tm[k,i,j+1]+tm[k,i,j-1]-4.0*tm[k,i,j])
          end if;
%
          tp'k:= (tpstar-thadv+thdiff
           +(wvelav*gamapr+q[k]*dtfdq[k]*wvel[kz])*psijci
           -(wvelav*gama[k]+q[k]*dtodq[k]*wvel[kz]) / pbar
            )*dt+tm[k,i,j];
%
%         moisture
%
          rme:=rmpr[k,i,j]+rmpr[k,i+1,j];
          rmw:=rmpr[k,i,j]+rmpr[k,i-1,j];
          rmn:=rmpr[k,i,j]+rmpr[k,i,j+1];
          rms:=rmpr[k,i,j]+rmpr[k,i,j-1];
          rmvad:=
          if (k = kz) then
            0.0 %gke
          else
            -(vadvrm[k+1]-vadvrm[k]) / dq[k]
          end if;
          rmhad:=-emsq[i,j]*tdsi* ( u[k,i,j]*rme-u[k,i-1,j]*rmw
           +v[k,i,j]*rmn-v[k,i,j-1]*rms);
          rmme := rmm[k,i+1,j]/(psm[i+1,j]+pbar);
          rmmw := rmm[k,i-1,j]/(psm[i-1,j]+pbar);
          rmmn := rmm[k,i,j+1]/(psm[i,j+1]+pbar);
          rmms := rmm[k,i,j-1]/(psm[i,j-1]+pbar);
          rmmij := rmm[k,i,j]/(psm[i,j]+pbar);
          rmhdif:=
          if (dnorm[k]  =  0.0) then
            0.0
          else
            cdiff[i,j]*dnorm[k]*dsi2
             *(rmme+rmmw+rmmn+rmms-4.0*rmmij)
             *(psm[i,j]+pbar)
          end if;
          t'rmp'k:= rmm[k,i,j]+dtt*(rmhad+rmvad+rmhdif);

%         suppress negative mixing ratios
%
          rmp'k:=
          if (t'rmp'k < 1.0E-20) then
            0.0
          else
            t'rmp'k
          end if
        in
          rmp'k,
          tp'k,
          omg
        end let
      end if;
%
      vp'k,
      new'eke'k,
      new'ppe'k,
      new'pvromg'k:=
      if ((i = i2)—boundary'cell(i,i1,il,j,j1,jl)) then
        vp[k,i,j],
        0.0, %eke
        0.0, %ppe
        0.0 %pvromg
      else
%
        let
          ua:=u[k,i-1,j]+u[k,i-1,j+1];
          ue:=u[k,i,j]+u[k,i,j+1];
          va:=v[k,i,j]+v[k,i-1,j];
          vb:=v[k,i,j]+v[k,i,j-1];
          ve:=v[k,i,j]+v[k,i+1,j];
          vf:=v[k,i,j]+v[k,i,j+1];
%
%         calculate v velocity component logarithmically
%
          ct2:=-corf2*(ua+ue);
          hadv2:= emhad2*( ue*ve*cycle'emonp-ua*va*new'amonp
           +vf*vf*fmonp-vb*vb*new'bmonp);
          pg2:= ((psn-pbar)*(phi[k,i,j+1]-phi[k,i,j])
           +zspdj-rtbar[k]*psrmj
           +psn*psldj* ( r*0.5* (t[k,i,j+1]+t[k,i,j])
           +rtbar[k]))*dsi;
          vbdiff:=
          if (dnorm[k]  =  0.0) then
            0.0
          else
            cdiff[i,j]*dnorm[k]*dsi2
             *( vm[k,i+1,j]/psmve+vm[k,i-1,j]/psmvw
             +vm[k,i,j+1]/psmvn+vm[k,i,j-1]/psmvs
             -4.0*vm[k,i,j]/psmv)*psmv
          end if;
          vat2:=
          if (k = kz) then
            0.0 %gke
          else
            -(vadvv[k+1]-vadvv[k])/dq[k]
          end if;
          t'vp:= (ct2+vat2-hadv2-pg2+vbdiff)*dt+vm[k,i,j]
%
%          calculate contributions to integrals
%
        in
          t'vp,
          dq[k]*psijci* (u[k,i,j]*u[k,i,j]+ v[k,i,j]*v[k,i,j]),
          tflev[k]*dq[k],
          (omega'k*omega'k)*dq[k]
        end let
      end if;
    returns
      array of tp'k
      array of up'k
      array of vp'k
      array of t'rmp'k
      array of omega'k
      value of sum (dq[k]*psijci*(up'k*up'k+vp'k*vp'k)) %new'eke'k
      value of sum (tflev[k]*dq[k]) %new'ppe'k
      value of sum (omega'k*omega'k*dq[k]) %new'pvromg'k
      value of sum (t'rmp'k*dq[k])%ptrhat
    end for; % k
    t'new'ptrhat := new'ptrhat+emsqi[i,j];
    t'new'epe := new'ppe*psijc*emsqi[i,j];
    t'new'vromg := new'pvromg*emsqi[i,j];
  returns
    array of tp'k
    array of up'k
    array of vp'k
    array of rmp'k
    array of sigdot'k
    value of sum new'eke
    value of sum (emsqi[i,j]) %new'cks
    value of sum (t'new'ptrhat*emsqi[i,j])
    value of sum t'new'epe
    value of sum ((psijc-0.988e6)*emsqi[i,j]) % psbar
    value of sum t'new'vromg
  end for % i,j
 in
  new'rm,
  new'sigdot,
  new'up,
  new'tp,
  new'rmp,
  new'vp,
  new'eke,
  new'cks,
  new'trhat,
  new'pe,
  new'psbar,
  new'romg
 end let
end function
```

References

[1] Leslie, L.M. et al., "A High Resolution Primitive Equations NWP Model for Operations and Research", pp 11-35, Australian Meteorological Magazine, No. 33, Mar 1985.

[2] Arakawa, A. and Lamb, V.R., "Computatioal Design of the Basic Dynamical Processes of the UCLA General Circulation Model," pp 174-256, 337, Methods of Computational Physics, Vol. 17, Academic Press, 1977.

[3] Miyakoda, K., "Cumulative Results of Testing a Meteorological-Mathematical Model," pp 99-130, Royal Irish Academy Proceedings, July 1973.

[4] McGraw et al., "SISAL: Streams and Iteration in a Single Assignment Language," Language Reference Manual Version 1.2, Lawrence Livermore National Laboratory, March 1, 1985.

[5] Feo, J.T., and Cann, D.C., "A Report on the SISAL Language Project," *Journal of Parallel and Distributed Computing*, Vol. 10, pp 349-366, 1990.

[6] Cray Research, "CF77 Compiling System Volume 4: Parallel Processing Guide," Cray Research, Incorporated, SG-3074 5.0, 1991.

[7] Cann, D.C., "The Optimising SISAL Compiler: Version 12.0," Computing Research Group, L-306, Lawrence Livermore National Laboratory, Livermore, 1992.

Even and Quarter-even Prime Length Symmetric FFTs and their SISAL Implementations *

Jaime Seguel

Department of Mathematics
University of Puerto Rico
Mayaguez, Puerto Rico PR 00681

Dorothy Bollman

Department of Mathematics
University of Puerto Rico
Mayaguez, Puerto Rico PR 00681

Abstract

Even and quarter-even symmetric DFTs are variants of the discrete Fourier transform (DFT) in which all redundant operations induced on the DFT equations by the presence of either an even or quarter-even symmetry in the input data have been eliminated. These kinds of transforms appear frequently in image processing and in the core procedures of some direct methods for the numerical solution of the Poisson equation. Fast methods for computing even and the quarter-even DFTs when the number of data samples is a power of two have been proposed by Swarztrauber [8] and Briggs [1]. Their methods are generalizable to any factorizable number of data samples. In this article, following the basic mathematical techniques used by Rader [7] to derive a fast prime length FFT, we introduce fast methods for computing the even and the quarter-even symmetric DFT for a prime number of data samples. The expression of these methods in terms of matrix algebra facilitates their implementations in SISAL.

1 Introduction.

Since its rediscovery in 1965 by Cooley and Tukey [2], the fast Fourier transform (FFT) has become one of the most widely used computational tools in science and engineering. The term FFT, initially associated with the Cooley-Tukey FFT for sequences of period $N = 2^k$, has become, after the efforts of many researchers over the years, the generic name of a whole family of efficient discrete Fourier transform (DFT) numerical methods. Each member in the FFT family is specialized to computing the DFT of a particular class of periodic sequences. This period is also referred to as the transform length and the DFT (FFT) of length N is usually called an N-point DFT (FFT). An important member of this family is actually an extension of Cooley and Tukey's idea to N-point DFT's where N is factorizable. These N-point FFTs compute the N-point DFT through nested sequences of DFT's whose lengths are the factors of N. The Good-Thomas algorithm [5] improves the extended Cooley-Tukey FFT for transform lengths that are highly composite. Rader's algorithm [7], in turn, is designed for computing prime length DFT's. These algorithms, all members of the family of traditional FFTs, reduce the N-point DFT arithmetic complexity from $O(N^2)$ to $O(N \log N)$.

A second family of fast DFT algorithms, called symmetric FFTs, was started with an article by Cooley, Lewis and Welch [3] in 1970. A symmetric FFT uses the symmetries of the input sequence to improve over its traditional FFT counterpart in terms of computational complexity and memory storage requirements. Especially important for their use in image processing and in fast Poisson solvers design are the even-symmetric FFT (E) and the quarter-even symmetric FFT (QE). The Cooley-Lewis-Welch algorithm computes the N-point DFT of a real (E) sequence using a $N/2$-FFT as a core procedure. This algorithm involves, however, a numerically unstable pre-process. Dollimore [4] redesigned the Cooley-Lewis-Welch algorithm improving on its numerical stability properties. Swarztrauber [8], who coined the term symmetric FFT, found a family of algorithms for computing real (E) and (QE) symmetric FFTs of any factorizable length. The main strategy in Swarztrauber's approach is to eliminate data redundancies induced by the symmetry of the input sequence in the intermediate steps of the traditional extended Cooley-Tukey FFT. Swarztrauber's algorithm does not consist of a core procedure separated from pre- or post-processes but its data flow is not as regular as its traditional counterpart.

*This work was supported by NSF grant RII-8905080 and the Computational Mathematics Group of Puerto Rico EPSCoR II grant.

Briggs's algorithm [1] is in the same spirit of Swarztrauber's but with a more regular data flow.

The purpose of this article is to propose new algorithms for computing the DFT of (E) and (QE) symmetric FFTs of prime length and test their Sisal implementations. These algorithms fill a gap in the symmetric FFT family since none of the above mentioned methods can be used for computing symmetric DFT's of prime length.

2 Background

The discrete Fourier transform of an N-periodic complex sequence $x = (x_n)$ is the N-periodic complex sequence $\tilde{X} = (\tilde{X}_k)$ determined by the equations

$$\tilde{X}_k = \sum_{n=0}^{N-1} x_n \omega_N^{kn}, \quad k = 0, ..., N-1, \tag{1}$$

where $\omega_N = \exp(-2\pi i/N)$ and $i = \sqrt{-1}$. Since these sequences are periodic of period N, the indices n and k range over Z/N, the set of integers modulo N. Also, the sequences x and $\tilde{X}$ can be represented by the vectors $x = (x_0, ..., x_{N-1})$ and $\tilde{X} = (\tilde{X}_0, ..., \tilde{X}_{N-1})$. This gives the following matrix formulation of equation (1)

$$\tilde{X} = F_N x$$

where $F_N = [\omega_N^{kn}]$, $0 \leq k, n \leq N-1$ is an $N \times N$ complex matrix.

An N-periodic sequence $x = (x_n)$ is said to be:

$$\begin{aligned} \text{even}-\text{symmetric (E)} \quad & if x_n = x_{-n} \text{ and} \\ \text{quarter}-\text{even symmetric (QE)} \quad & if x_n = x_{-(n+1)}. \end{aligned}$$

For an odd number N, a fundamental set of indices for the (E) symmetry is any subset S of Z/N satisfying:

$$\begin{aligned} (E1) \quad S \cup (-S) \ &= Z/N \quad \text{and} \\ (E2) \quad S \cap (-S) \ &= \{0\} \end{aligned}$$

where $-S = \{-n : n \in S\}$. A fundamental set of indices S for the (QE) symmetry, in turn, is any subset of Z/N satisfying:

$$\begin{aligned} (QE1) \quad S \cup -(S+1) \ &= Z/N \quad \text{and} \\ (QE2) \quad S \cap -(S+1) \ &= \frac{N-1}{2}, \end{aligned}$$

where $S+1 = \{n+1 : n \in S\}$. The set $\{0, 1, ..., \frac{N-1}{2}\}$ is a fundamental set of indices for both symmetries. The restriction of an (E) ((QE)) symmetric sequence to an (E) ((QE)) fundamental set of indices eliminates redundant data. If $x = (x_n)$ is (E) symmetric, its discrete Fourier transform $\tilde{X} = (\tilde{X}_k)$ is also (E) symmetric and so, equation (1) can be reduced to

$$\tilde{X}_k = x_0 + \sum_{n=1}^{\frac{N-1}{2}} x_n(\omega_N^{kn} + \omega_N^{-kn}),$$

$k = 0, 1, \ldots, \frac{N-1}{2}$ for N odd.

This equation defines the even discrete Fourier transform ((E) DFT). Since $\omega_N^{kn} + \omega_N^{-kn} = 2\cos(2\pi kn/N)$, the (E) DFT is sometimes called the cosine transform. Its matrix representation is the $\frac{N-1}{2}+1 \times \frac{N-1}{2}+1$ real matrix

$$F_N^{(e)} = \begin{bmatrix} 1 & 2 & 2 & \ldots & 2 \\ 1 & c_1 & c_2 & \ldots & c_{\frac{N-1}{2}} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & c_k & c_{2k} & \ldots & c_{(\frac{N-1}{2})k} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & c_{\frac{N-1}{2}} & c_{2(\frac{N-1}{2})} & \ldots & c_{\frac{(N-1)^2}{2}} \end{bmatrix}$$

where $c_m = 2\cos(2\pi\frac{m}{N})$. On the other hand, if (x_n) is (QE) symmetric, equation (1) can be reduced to

$$\tilde{X}_k = x_{\frac{N-1}{2}}\omega_N^{\frac{k(N-1)}{2}} + \sum_{n=0}^{\frac{N-1}{2}-1} x_n(\omega_N^{kn} + \omega_N^{-k(n+1)}),$$

The (QE) symmetry does not induce any reduction in the number of DFT outputs, Therefore. the matrix representation of the (QE) DFT is an $N \times \frac{N-1}{2}+1$ complex matrix. We denote this matrix $F_N^{(qe)}$.

Rader's prime length complex FFT is based on the identification of an $N-1 \times N-1$ block in F_N which can be transformed into a Hankel-circulant by means of appropriate row and column permutations. In general, an $M \times M$ matrix A is a Hankel-circulant if it can be written as

$$A = \begin{bmatrix} a_0 & a_1 & a_2 & \ldots & a_{M-1} \\ a_1 & a_2 & & \ldots & a_0 \\ a_2 & & & \ldots & a_1 \\ \vdots & \vdots & \vdots & \ldots & \vdots \\ a_{M-1} & a_0 & a_1 & \ldots & a_{M-2} \end{bmatrix}$$

We use the notation $A = Circ(a_0, ..., a_{M-1})$ and $G_A = (a_0, a_1, \ldots, a_{M-1})^T$, the "generator" of A.

Hankel-circulants admit the following interesting matrix factorization: Let A be an $M \times M$ Hankel-circulant and let $\Omega(A)$ be the diagonal matrix whose main diagonal is the inverse DFT of the first row of A. Then,

$$A = F_M \Omega(A) F_M$$

Thus, the multiplication of a Hankel-circulant A of order M by a vector can be computed in terms of M-point FFTs, in fact, in terms of 2^k-point FFTs by embedding A in a circulant $\bar{A}$ of order 2^k defined as follows: Let $A = Circ(a_0, a_1, \ldots, a_{M-1})$ and let k be the minimum m for which $2^m \geq 2M - 1$. Define

$$\bar{A} = Circ(a_0, ..., a_{M-1}, a_0, a_1, ..., a_{M-2}, 0, \ldots, 0)$$

where the number of zeros inserted is $2^k - 2M + 1$ and for any vector x of length M, let $\bar{x}$ be the vector of length 2^k obtained from x by padding the last $2^k - 2M + 1$ positions with zeros. Then Ax is the vector consisting of the first M components of $\bar{A}\bar{x}$.

In order to identify the Hankel-circulant block in F_N let us rewrite the N-point DFT matrix as

$$F_N = \begin{bmatrix} 1 & e^T \\ e & W_N \end{bmatrix}$$

where e is the column vector of ones of length $N - 1$ and e^T is its transpose. The $N - 1 \times N - 1$ complex block can be transformed into a Hankel-circulant by pre- and post-multiplacations by permutation matrices whose definition rely on the field structure of Z/N. Indeed, if N is prime, then Z/N is a field and the multiplicative group $U(N) = Z/N - \{0\}$ is a cyclic group generated by a primitive root g modulo N. For example, $U(5) = \{1, 2, 3, 4\}$ is generated by $g = 2$ since

$$\begin{aligned} < 2^0 >_N &= 1 \\ < 2^1 >_N &= 2 \\ < 2^2 >_N &= 4 \\ < 2^3 >_N &= 3 \end{aligned}$$

where $< \cdot >_N$ denotes the least positive residue congruent to g^k modulo N. Thus, if g is a primitive root modulo N we define $P_{N,g}$ as a matrix representation of the permutation $y_k \leftarrow x_{<g^{k-1}>_N}$, $1 \leq k \leq N - 1$. It can be easily shown that

$$\tilde{W}_N = P_{N,g} W_N P_{N,g}^{-1}$$

where $\tilde{W}_N = Circ(\omega_N, \omega_N^{<g>_N}, \ldots, \omega_N^{<g^{N-2}>_N})$. Hence,

$$F_N x = \begin{bmatrix} 1 & e^T \\ e & P_{N,g}^{-1} \tilde{W}_N P_{N,g} \end{bmatrix} x$$

where e is a column vector of 1's and e^T is its transpose. Thus,

$$\begin{aligned} F_N x &= y + \begin{bmatrix} 0 \\ P_{N,g}^{-1} \tilde{W}_N P_{N,g} x' \end{bmatrix} \\ &= y + \begin{bmatrix} 0 \\ P_{N,g}^{-1} F_M (F_M^{-1} G_{\tilde{W}_N} \circ F_M P_{N,g} x') \end{bmatrix} \end{aligned}$$

where $M = N - 1$, $\circ$ denotes component-wise multiplication,

$$y = \begin{bmatrix} x_0 + x_1 + \ldots + x_{N-1} \\ x_0 \\ \vdots \\ x_0 \end{bmatrix}$$

and x' is the result of deleting the first component of x. This matrix expression is essentially Rader's algorithm. The same factorization but using $\bar{W_N}$ instead of W_N gives what we call the extended Rader algorithm.

In the rest of this paper, we show how, as in the extended Rader algorithm, the core procedures for computing even and quarter-even prime length symmetric FFTs can be written as the product of a Hankel-circulant by a vector. The efficiency of these algorithms, as well as the Rader algorithm or its extended version, depends on the availability of efficient algorithms for computing FFTs.

3 Prime length (E) symmetric FFTs

Let us rewrite the (E) DFT matrix as

$$F_N^{(e)} = \begin{bmatrix} 1 & 2e^T \\ e & C_N \end{bmatrix}$$

where $C_N = [c_{kn}]$, $1 \leq k, n \leq \frac{N-1}{2}$, an $\frac{N-1}{2} \times \frac{N-1}{2}$ real matrix. Now let g be a primitive root modulo N and define $P_{N,g}^{(e)}$ to be the matrix representation of the permutation on $1 \leq n \leq \frac{N-1}{2}$, defined by the map

$$p_{N,g}^{(e)}(n) = \begin{cases} < g^{n-1} >_N, & \text{if } < g^{n-1} >_N \leq \frac{N-1}{2} \\ < N - g^{n-1} >_N, & \text{otherwise} \end{cases}$$

Then we have

$$F_N^{(e)} x = \begin{bmatrix} 1 & 2e^T \\ e & {P_{N,g}^{(e)}}^{-1} \tilde{C}_N P_{N,g}^{(e)} x \end{bmatrix}$$

where

$$\tilde{C}_N = Circ(c_{p_{N,g}^{(e)}(1)}, c_{p_{N,g}^{(e)}(2)}, \ldots, c_{p_{N,g}^{(e)}(\frac{N-1}{2})}).$$

Thus, in this case we have

$$F_N^{(e)}x = y + \begin{bmatrix} 0 \\ P_{N,g}^{(e)\,-1} F_M(F_M^{-1}G_{\tilde{C}_N} \circ P_{N,g}^{(e)}x') \end{bmatrix}$$

where $M = N-1$,

$$y = \begin{bmatrix} x_0 + 2(x_1 + x_2 + \ldots + x_{\frac{N-1}{2}} \\ x_0 \\ \vdots \\ x_0 \end{bmatrix}$$

and

$$x' = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{\frac{N-1}{2}} \end{bmatrix}$$

For example, suppose $N = 7$. The group of units in $Z/7$ is generated by $g = 3$. In fact,

$$\begin{aligned} <3^0>_7 &= 1 \\ <3^1>_7 &= 3 \\ <3^2>_7 &= 2 \\ <3^3>_7 &= 6 = -1 \\ <3^4>_7 &= 4 = -3 \\ <3^5>_7 &= 5 = -2. \end{aligned}$$

Clearly $\{1,3,2\}$ is a fundamental set and in this case

$$\tilde{C}_7 = \begin{bmatrix} c_1 & c_3 & c_2 \\ c_3 & c_2 & c_6 \\ c_2 & c_6 & c_4 \end{bmatrix} = \begin{bmatrix} c_1 & c_3 & c_2 \\ c_3 & c_2 & c_1 \\ c_2 & c_1 & c_3 \end{bmatrix}$$

The last equality is due to the fact that the cosine function is even.

4 Prime length (QE) symmetric FFTs

Hankel-circulant representations for blocks in the core submatrix $Q_N = [\omega_N^{kn} + \omega_N^{-k(n+1)}]$, $0 \leq k \leq N-1$ and $0 \leq n \leq \frac{N-1}{2} - 1$ of $F_N^{(qe)}$ require some extra work since neither the upper nor the lower $\frac{N-1}{2} \times \frac{N-1}{2}$ square blocks in Q_N can be transformed into Hankel-circulant through any row-column permutation. A way around this difficulty is based on the function

$$\psi : Z/N \rightarrow Z/2N, \quad \psi(n) = 2n+1,$$

which maps (QE) symmetries into (E) symmetries in the sense that

$$\psi(-(n+1)) = -\psi(n).$$

In fact, using this function we get

$$\begin{aligned} \omega_N^{kn} + \omega_N^{-k(n+1)} &= \omega_{2N}^{-k}(\omega_{2N}^{k(2n+1)} + \omega_{2N}^{-k(2n+1)}) \\ &= \omega_{2N}^{-k}\, 2\cos(2\pi k(2n+1)/2N) \end{aligned}$$

for $k \neq 0$. Therefore, if

$$D_N = \begin{bmatrix} \omega_{2N}^{-1} & & & \\ & \omega_{2N}^{-2} & & \\ & & \ddots & \\ & & & \omega_{2N}^{-(N-1)} \end{bmatrix}$$

and

$$T_N = \begin{bmatrix} t_1 & t_3 & \cdots & t_{N-2} \\ t_2 & t_6 & \cdots & t_{2(N-2)} \\ \vdots & \vdots & \ddots & \vdots \\ t_{N-1} & t_{3(N-1)} & \cdots & t_{(N-1)(N-2)} \end{bmatrix}$$

where $t_s = 2\cos(2\pi s/2N)$, then we have,

$$Q_N = D_N T_N.$$

Since the input indexing set has been embedded into $Z/2N$ while the output indexing set remains a subset of Z/N, a slight adaptation of the Hankel-circulant representation technique is required. This adaptation is based in the following observations: first of all, since N is an odd prime $U(2N)$ is isomorphic to $U(N)$ and in particular, $U(2N)$ is cyclic of order $N-1$. Furthermore, no even number belongs to $U(2N)$ and so, $U(2N) \subset \psi(Z/N)$. By way of example, let us consider the case $N = 7$. As pointed out earlier, the group $U(7)$ is the output indexing set for the (QE) core procedure. Since ψ transforms (QE) symmetries into (E) symmetries, an (E) fundamental subset of $\psi(Z/7)$ will be the image of a (QE) fundamental indexing set in $Z/7$. Now, $U(14)$) is generated by $g = 3$. In fact,

$$\begin{aligned} <3^0>_{14} &= 1 \\ <3^1>_{14} &= 3 \\ <3^2>_{14} &= 9 \\ <3^3>_{14} &= 13 \\ <3^4>_{14} &= 11 \\ <3^5>_{14} &= 5 \end{aligned}$$

The missing odd number in the above list is $7 = \psi(3)$. But x_3 is not an input for the core computation. A natural choice for an input (E) fundamental set is $\{1,3,9\}$, which is the image under ψ of the (QE) fundamental set $\{0,1,4\}$. The rearranged matrix is now,

$$R_7 = \begin{bmatrix} t_1 & t_3 & t_9 \\ t_3 & t_9 & t_{13} \\ t_2 & t_6 & t_4 \\ t_6 & t_4 & t_{12} \\ t_4 & t_{12} & t_8 \\ t_5 & t_1 & t_3 \end{bmatrix}$$

By using the properties $t_j = t_{j'}$, if $j + j' = 0 \bmod 2N$ and $t_j = -t_{j'}$ whenever $j + j' = 7k$, k odd, we see that

$$\begin{aligned} R_7 &= \begin{bmatrix} 1 & & \\ & 1 & \\ & & -1 \\ -1 & & \\ & -1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} t_1 & t_3 & t_9 \\ t_3 & t_9 & t_1 \\ t_9 & t_1 & t_3 \end{bmatrix} \\ &= \begin{bmatrix} \Lambda_7 \\ -\Lambda_7 \end{bmatrix} \tilde{T}_7 \end{aligned}$$

which is a Hankel-circulant based matrix factorization for T_7. It is crucial to note that the same integer $g = 3$ has been used to generate both $U(7)$ and $U(14)$. Such a common integer exists for any pair of grc ,ps $U(N)$ and $U(2N)$ provided that N is an odd prime. In fact, for N an odd prime and g odd, g is a primitive root modulo N if and only if g is a primitive root modulo $2N$.

Now positive signs correspond precisely to the cases $< g^n >_{2N} \leq N$. Also, if $n < \frac{N-1}{2}$, then $< g^n >_{2N} < N$ if and only if $< g^{n+\frac{N-1}{2}} >_{2N} > N$, each sign pattern will be the negative of the other.

The general case can be described as follows: Let g be an odd primitive root modulo N (and hence a primitive root modulo $2N$). For each $j = 0, 1, \ldots, \frac{(N-3)}{2}$, let

$$\lambda_j = \begin{cases} 1, & \text{if } < g^j >_{2N} \leq N \\ -1, & \text{otherwise.} \end{cases}$$

Let $\Lambda_N = \text{diag}(\lambda_0, \ldots, \lambda_{\frac{N-3}{2}})$. Also, let $P_{N,g}^{(qe)}$ be the matrix representation of the map defined by

$$P_{N,g}^{(qe)}(n) = \begin{cases} < g^n >_{2N}, & \text{if } < g^n >_{2N} \leq N \\ 2N - < g^n >_{2N}, & \text{otherwise,} \end{cases}$$

where $n = 0, 1, \ldots, N-2$, and let $Q_{N,g}^{(qe)}$ be the matrix representation of the map defined by $Q_{N,g}^{(qe)}(n) = \psi^{-1}(P_{N,g}^{(qe)}(n))$, $n = 0, 1, \ldots, \frac{(N-3)}{2}$. Then we have

$$\begin{aligned} F_N^{(qe)} x &= y + \begin{bmatrix} 0 \\ D_N \circ P_{N,g}^{-1} \begin{bmatrix} \Lambda_N \\ -\Lambda_N \end{bmatrix} \tilde{T}_N Q_{N,g}^{(qe)} x' \end{bmatrix} \\ &= y + \begin{bmatrix} 0 \\ D_N \circ P_{N,g}^{-1} \begin{bmatrix} \Lambda_N \\ -\Lambda_N \end{bmatrix} F_{N-1} H x' \end{bmatrix} \end{aligned}$$

where

$$\tilde{T}_N = Circ(t_{P_{N,g}^{(qe)}(0)}, t_{P_{N,g}^{(qe)}(1)}, \ldots, t_{P_{N,g}^{(qe)}(\frac{N-3}{2})})$$

$$Hx' = F_{\frac{(N-1)}{2}}^{-1} G_{\tilde{T}_N} \circ F_{\frac{(N-1)}{2}} Q_{N,g}^{(qe)} x',$$

$$y = \begin{bmatrix} x_{\frac{N-1}{2}} + 2(x_0 + x_1 + \ldots + x_{\frac{N-3}{2}}) \\ x_{\frac{N-1}{2}} Z \end{bmatrix},$$

$$Z = \begin{bmatrix} \omega_N^{\frac{(N-1)}{2}} \\ \omega_N^{\frac{2(N-1)}{2}} \\ \vdots \\ \omega_N^{\frac{(N-1)^2}{2}} \end{bmatrix},$$

$$D_N = \begin{bmatrix} \omega_{2N}^{-1} \\ \omega_{2N}^{-2} \\ \vdots \\ \omega_{2N}^{-(N-1)} \end{bmatrix}, \text{ and } x' = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{\frac{N-3}{2}} \end{bmatrix}.$$

5 Implementations.

All three of the algorithms developed in this work, Rader, even, and quarter-even, require the computation of the product of a circulant matrix A by a permuted vector Z, which we compute by a cyclic convolution $F(F^{-1}G_A \circ FZ)$. Thus the efficiency of all three algorithms depends on an efficient FFT and its inverse for vectors of size $N-1$ or $(N-1)/2$ where N is the length of the original input vector. When N is of the form $2^k + 1$, the FFTs are of size 2^k. In all other cases, we embed the circulant A in a circulant of size 2^k, where k is the least number m for which $2^m \geq 2size(A) - 1$. power of 2.

The three algorithms were programmed in Sisal 12.9.1, using double precision, and tested on a silicon graphics computer (S.G.I.) with four processors. Cyclic convolution was implemented using a radix 4 version of Stockham's algorithm, which we found to be the fastest FFT on the S.G.I. when compared to other standard FFTs, both with and without digit-reversal.

Initialization for each of the three algorithms includes the computation of:

- a primitive root g modulo N
- vectors of indices needed for permuting the input and output vectors
- the size M of the cyclic convolution and a boolean variable indicating whether embedding necessary.
- twiddle factors for Stockham's radix 4 FFT
- the inverse DFT of the first row of the circulant

Initialization for the quarter-even case also requires

- the vector D_N of negative powers of ω_{2N}
- the vector Z of powers of $\omega_N^{\frac{N-1}{2}}$
- the vector Λ_N

The following two functions, expressed in pseudo Sisal, are used for all three algorithms:

```
function circXvector(M:integer;
                     invfft_circ,x:array[complex];
                     returns array[complex])
% twiddle factor arguments have been suppressed
% array[complex] is actually implemented by a pair
% of real vectors
% vecXvec returns the component-wise product
% of two vectors
let
 z := vecXvec(M,invfft_circ,fft(M,x))
in
 fft(M,z)
end let
end function %circXvector
```

```
function core(M,N:integer;emb:boolean
              index1:array[integer];
              invfft_circ,x:array[complex]
              returns array[complex])
% perm returns a vector with its components in the
% order given by a vector of indices
let
 z := perm(N - 1,index1,array_setl(x,0));
 embedded_z := if ēmb then z
                    else embed(N - 1,M,z)
in
 circXvector(M,invfft_circ,embedded_z)
end let
end function %core
```

Each of the three algorithms requires an additional function, called "rader","even", and "qeven", respectively. For example, for Rader's algorithm, we use

```
function rader(M,N:integer;emb:boolean;
               index1,index2:array[integer];
               invfft_circ,x:array[complex]
               returns array[complex]
%scalar_plus_vector returns the result of adding a
%scalar component-wise to a vector
let
 z := perm(N - 1,index2,
        core(M,N,emb,index1,invfft_circ,x));
 y_0 := for i in 0, N - 1
       returns value of sum x[i]
       end for
in
 array_addl(scalar_plus_vector(N - 1,x[0],z),y_0)
end let
end function % rader
```

The function even is almost identical to rader except for a factor of 2.0 in the computation of y_0. The function qeven is given by

```
function qeven(M,N:integer;
               emb:boolean;
               index1,index2:array[integer];
               Λ:array[double_real];
               invfft_circulant,Z,D:array[complex]
               returns array[complex])
% scalarXvec_plus_vec(c,x,y) returns cx + y
% where c is a complex scalar and x and y are
% complex vectors
% cadd returns the sum of two complex scalars
let
 r :=  core(M,(N + 1)/2,emb,index1,invfft_circulant,x);
 u1,u2 := for i in 0,(N - 3)/2
          returns array of Λ[i]*r[i]
                  array of -Λ[i]*r[i]
          end for;
 v := vecXvec(N - 1,D,perm(N - 1,index2,u1|| u2));
 w := scalarXvec_plus_vec(N - 1,x[(N - 1)/2],Z,v);
 y_0 := cadd(x[(N - 1)/2],
          for i in 1,(N - 1)/2
           returns value of sum 2.0d0*x[i]
          end for)
in
 array_addl(array_setl(w,1),y_0)
end let
end function %qeven
```

Since the even and quarter-even algorithms reduce the problem size by approximately one-half, we can expect that running times for each of these two algorithms to be about one-half of the running time for the Rader algorithm, no matter the value of N. In practice, however, we can do even better. For example, running times for the largest prime less than 2^{14} and the smallest prime greater than 2^{14} are:

$N = 16,381$

	1 CPU	2 CPU's	4 CPU's
Rader	1.92	1.36	1.10
even	0.78	0.55	0.48
qeven	0.91	0.67	0.53

$N = 16,411$

	1 CPU	2 CPU's	4 CPU's
Rader	4.17	2.99	2.61
even	1.85	1.28	1.04
qeven	1.98	1.38	1.09

We note that in these two examples, even and qeven run more than twice as fast as Rader. This will be true for any N not of the form $2^k + 1$ since the even and quarter-even algorithms will compute the FFT in terms of a cyclic convolution which is half the size of the cyclic convolution used by the Rader algorithm; however, running times for cyclic convolution more than double with double the problem size.

References

[1] W. Briggs, "Further Symmetries of In-place FFTs," *SIAM J. Sci. Stat. Comp.*, Vol. 8, pp. 644-654, 1987.

[2] J. Cooley and J. Tukey, "An Algorithm for the Machine Caclulation of the Complex Fourier Series," *IEEE Trans. Comput.*, Vol. AC-28, pp. 819-830, 1965.

[3] J. Cooley, P. Lewis, and P. Welch, "The Fast Fourier Transform Algorithm: Programming Considerations in the Calculation of Sine, Cosine and Laplace Transforms," *J. Sound Vib.*, Vol. 12, pp. 315-337, 1970.

[4] J. Dollimore, "Some Algorithms for use with the Fast Fourier Transform," *J. Inst. Math. Appl.*, Vol. 12, pp. 115-117, 1973.

[5] I. Good, "The Interaction Algorithm and Practical Fourier Analysis," *J. Royal Stat. Soc.*, Ser. B 20, pp. 361-375, 1958.

[6] J. Otto, "Prime Factor Symmetric FFTs," manuscript.

[7] C. Rader, "Discrete Fourier Transforms when the Number of Data Points is Prime," *Proc. IEEE*, Vol. 56, pp. 1107-1108, 1968.

[8] P. Swarztrauber, "Symmetric FFTs," *Math. Comp.*, Vol. 47, pp. 323-346, 1986.

Top-Down Thread Generation for Sisal *

Bhanu Shankar A.P.W.Böhm W.A.Najjar
Computer Science Department
Colorado State University
FortCollins, CO 80521

Abstract

In this paper we present a model of coarse grain dataflow execution. We present a top down method for generating machine independent multithreaded code, called MIDC. We define MIDC. We discuss the relevant phases in the Sisal to MIDC compilation process, and present some example compilations. We quantify the number of threads, number of inputs per thread, and average thread size for Livermore and Purdue benchmarks.

Keywords: *Hybrid von Neumann/Dataflow, threads, code generation algorithm.*

1 Introduction

Hybrid dataflow machines execute *threads* of von Neumann RISC code, where the threads are enabled by the availability of data. Thread enabling is either implemented by efficient matching using explicit token storage and presence bits, or by pools of "waiting" and "ready" threads with hardware support to move threads from and to these pools. A *strict* firing rule allows a thread to execute only when all its inputs are available, avoiding threads to block but potentially increasing latency and decreasing thread size. Conversely, a *non-strict* firing rule allows a thread to execute when some of its inputs are not available. In this case threads can become larger, but the architecture must cope with blocking threads, which may increase the complexity of synchronization, and may consequently require a larger, replicated, processor state. It appears that the matching unit oriented machines, such as the monsoon [PC90] and EM-4[SKS+92], with little processor state, would favor non-blocking threads, whereas the pool oriented machines with large processor state, such as the TERA machine [ACC+90], would favor blocking threads, but this remains to be investigated. In this study we will restrict ourselves to non-blocking threads.

In this paper we present the compilation of a functional language, Sisal [MSA+85], into machine independent coarse grain dataflow code (MIDC). We define MIDC, outline the relevant compilation stages, and measure the total number of threads executed, the average thread size, and the number of inputs per thread for the Livermore and Purdue benchmarks.

The rest of the paper is organized as follows. In section two we introduce the MIDC model of computation and outline MIDC generation. In section three we present some example programs and their intermediate forms. In section four we analyze the dynamic properties of our cluster generation strategy.

2 The MIDC Model of Computation

MIDC nodes are *threads* of von-Neumann instructions. A node is scheduled as a unit on one processor. An MIDC program is a data-driven graph of these clusters. Synchronization (matching) occurs at the cluster level, and once a cluster is enabled, it runs to completion without blocking and has a deterministic execution time. This implies that an instruction that issues a memory request cannot be in the same cluster as the instructions that use the value returned by the split-phase read. In our model, threads executes *strictly*, that is, a thread is enabled only when all the input tokens to the cluster are available. The limited fine grain parallelism internal to the cluster could be exploited, for instance, by a superscalar or VLIW processor. The construction of clusters is guided by the following objectives:

- Minimizing the internal thread parallelism, so that the inter thread parallelism is maximized.
- Ensuring deadlock-free execution of threads.
- Minimizing matching and synchronization overhead by maximizing the locality and making clus-

*This work is supported by NSF Grant MIP-9113268

ters as large as possible without violating the first two objectives.

- Minimizing the input latency caused by large numbers of inputs to clusters.

It should be noted that there is a trade-off between thread size and input latency. The larger a thread, the more inputs it may require. It is therefore not always advantageous to have as large as possible threads.

2.1 Compiler Structure

We are designing a Sisal compiler for coarse grain dataflow machines by targeting to MIDC, which should be easily mappable to machines such as monsoon, EM-4, *T, and Tera. IF1 [SG85] is used as a common intermediate form for all Sisal compilers, and decouples the front end of the compiler from the code generator. There are four components of the graph form: nodes, edges, types and sub-graphs. Nodes denote operations, edges represent values that are passed from node to node, types are attached to edges and functions. To provide block structure, some set of nodes and edges can be encapsulated in a sub-graph. For control structures such as loops and conditionals, sub-graphs are encapsulated in compound nodes.

The functional semantics of IF1 prohibits the expression of copy-avoiding optimizations. An extension of IF1, called IF2 [WSYR86], allows operations that explicitly allocate and manipulate memory. A class of AT operations is introduced, which are similar to their IF1 counterparts but have additional information indicating where in memory their results should be constructed. Artificial dependence edges have also been added to introduce synchronization where this may be useful, for instance to facilitate update in place optimization [Can89]. The concept of a *buffer* in IF2 provides a machine independent way of describing addressable memory. A buffer comprises two parts: 1) a buffer pointer into a contiguous block of memory, and 2) an element descriptor that defines the elements of the buffer, which may be arrays, streams, records or basic types. IF2 makes two assumptions: 1) all scalar values are operated by value and therefore copied to wherever they are needed, and 2) all of the fanout edges of a structured type are assumed to reference the same buffer, that is, each edge is not assumed to represent a distinct copy of the data. IF2 edges are decorated by pragmas. For instance, there is a "update-in-place" pragma that indicates that a certain replacement of an array element can be done without copying the other elements of the array, but by destructively updating the particular element. This dramatically improves the run time performance of the system.

The Optimizing Sisal Compiler (OSC) [Can89] performs some transformations that are useful for the machines that it currently targets, but not necessarily for MIDC. Sometimes analysis is needed to decide whether such transformations need to be undone. One such example is the transformation of `AScatter` nodes in the `Generator` sub-graph of `Forall` nodes to a `RangeGenerate` and the addition of `AElement` nodes to read the array in the `Forall Body`. This transformation is turned off for the purpose of MIDC code generation.

2.2 Cluster Generation

The top down cluster generation process transforms IF2 into a flat machine independent dataflow code MIDC where the nodes are clusters of straight line von Neumann code, or basic blocks, and the edges represent data paths. A *node header* provides a node-label, the number of input ports, the number of registers used, and the destinations for all outputs. Outputs can be conditional, i.e. only sent if a register, specified in the output directive, contains true. The node header is followed by a stream of instructions. Instruction operands may be node input values, registers or literals. Tokens travelling through the graph are tagged with an *activation name*, which can be interpreted as a pointer to a stack frame as in [CP90] or as a color in a more classical dataflow sense.

An MIDC program consists of a number of function definitions, one of which is called *main* and communicates with the outside world. A function consists of a header and a body. A function header provides the interface between calls and called functions. The MIDC syntax definitions are presented in Table 1.

The IF2 to MIDC translation process starts with a graph analysis and splitting phase, which breaks up the nested IF2 graphs such that threads can be generated. Innermost loops are identified, as they are candidates for vectorization, which is important not only for machines with vector capabilities, but maybe even more so for block (pre)fetching of conglomerate data. Initial values for reduction operators are generated in the appropriate threads. Threads terminate at control graph interfaces for loops and conditionals, and at nodes for which the execution time is not statically determinable, such as a function calls and remote memory accesses. Terminal nodes are identified and the IF2 graphs are split along this seam.

Function interfaces are then set up for all the functions that have survived function inlining. A function

Function Interface:	F FName Node# #ins Output
FName:	string
Output:	< (Destination) >
Destination:	node# port#
Function Body:	Node ...
Node:	N Node# #regs #ins OutList Instruction ...
OutList:	< (CondReg ValueReg Destination MatchFn) ... >
Instruction:	Targets = Operation
Targets:	Target Register | Target Register , Targets
Target Register:	$R_{\#}$ | $V_{\#}$
Operation:	Operator Operands
Operands:	Operand | Operand , Operands
Operand:	Operand Register | Literal
Operand Register:	$V_{\#}$ | $R_{\#}$ | $I_{\#}$
Literal:	"Type Value"
CondReg:	Operand Register
ValueReg:	Operand Register
MatchFn:	Normal | Queue
#: number or number of	%: comment ...: a sequence

Table 1: MIDC Syntax Definitions

interface consists of input and output directives for a function call. It connects input parameters to nodes in the function body, and combines return values with return information provided by the caller. A trigger input is used to activate code that has no inputs.

IF2 function bodies consist of simple nodes, function calls, and compound nodes. In the case of simple nodes, such arithmetic and logical operators, the translation scheme from IF2 to MIDC is straightforward. Simple nodes are merged to form threads. A function call is translated in code that connects the call site to the function interface, where the input values and return contexts are given a new activation name. The return contexts combine the caller's activation name and the return destination and are tagged (as are the input values) with the new activation name. The function interface returns function results using dynamic arcs.

Compound nodes, representing conditionals and loops contain subgraphs. The IF2 to MIDC compiler generates threads for all sub-graphs and "glue" code to link them together. They are wired up sequentially, such that activation names and other resources for these loops can be kept to a minimum. Forall loops with data independent body graphs and a loop count known before the loop is executed, consist of a *generator*, a *body*, and a *returns* graph. They are wired up so that all parallelism in these loops can be exploited. To that end, all ordering in the reduction operators has been removed. This is valid as Sisal reduction operators, such as *sum*, *product*, *least*, and *greatest* are commutative as well as associative.

3 Examples

Three example SISAL programs will be used to exemplify our code generation process. Please refer to the appendix. Livermore Loop 3 in figure 1 is a vector inproduct function, figure 5 shows a double recursive binary integration function using the trapezium rule, and figure 9 shows a bubble sort function. Function Inlining, common sub-expression elimination and other classical compiler optimizations along with update in place and copy elimination gives rise to the second intermediate form IF2. The IF2 forms of the Livermore Loop 3, binary integration and sort programs are shown in figures 2 6 and 10 respectively. The programs are compiled into the MIDC form shown in figures 3, 7 and 11. To facilitate easier reading, comments have been added by hand to reference the source code or provide an insight into the "glue" code needed.

3.1 Livermore Loop 3

During IF2 analysis we find that the loop is an *innermost* loop, and that there are no other structures that impede vectorization present in the graph. The generator graph for the loop contains a set of **AScatter** nodes. In keeping with the need for partial reductions the **Reduce** node is duplicated in the body. Since MIDC threads have no state, the initial value for the reduction has to be supplied from outside the **Returns** graph, and all the intermediate reduction values have to be looped back to the **Returns** graph. The corresponding edges are added in the **Generator** graph. The graph is now ready for code generation.

Figure 3 shows the code generated and figure 4 shows the MIDC graph structure for Livermore Loop 3.

1. The **RangeChecK (RCK)** operator in node 3 checks whether the sizes of all the array inputs to the loop are the same. Node 3 also contains code to calculate the number of chunks and the size of the odd sized chunk, if present.

2. A stream of colors to drive the loop bodies in parallel will be generated using the **Generate Activation Name Stream (GANS)** instruction in MIDC node 4, along with the outer context and the the position of the chunk in the array using the **SetColorSetinXed (SCSX)** instruction. The loop set up is completed in MIDC node 4 by the Proliferation of the loop body inputs and the generation of input values for any reduction operation necessary.

3. Code for the loop body is generated in two version: one that pertains to full chunks and one for the odd sized chunk. This is in keeping with our desire to reduce the number of tokens to a minimum. For each array that is present, a **FetchCHunk (FCH)** operation is issued. This is done after computing its address in the array, see Nodes 6 & 8. This particular loop body is encoded using vector instructions. Partial reductions and array gather are generated. To allow out of order execution of the final reduction operation, MIDC node 7 recolors partial results to the outer context using a **Label (LAB)** instruction.

4. The **Returns** graph is used as a barrier as the partial reductions are reduced further to give the final reduction. The **Returns** graph is run on the color of the outer context. To implement out of order evaluation, a *Queue Matching Function* is added. As its name implies if two or more tokens of the same color appear at the same port, the tokens are queued in non-deterministic order for processing.

3.2 Binary Integration

The analysis phase finds a **Select** node in the function body. Since control can flow either to the *Else* or the *Then* sub-graphs the outputs of the sub-graphs needs to be merged, and consequently, the MIDC graph needs to be split at this merge point. Further analysis shows that the **Select** node feeds the output of the function and not any other internal nodes, and the results of the *Then* and *Else* graphs can be linked directly to the function output interface. The **Else** graph contains two independent call nodes, which can be a part of the same thread. The **Else** graph is split broken at the outputs of the call nodes, as their latency is indeterminate.

Figure 7 shows the MIDC code generated and figure 8 shows the MIDC graph structure. Code is generated as follows.

1. The input and output function interfaces are generated.

2. For the function body, code generation is straightforward. Most IF2 nodes are simple, and have a corresponding operator in MIDC. For the select node, tokens have to be directed to the corresponding subgraphs, using conditional outputs driven by the boolean input to the select node. The **Then** branch is straightforward, as it is empty and only acts as a redirection node. In the **Else** branch, we come across two call nodes. Their function interfaces are identified using the function name in the

call node. A new activation name is generated using the **GenerateActivationName (GAN)** instruction. All the parameters are recolored using the **SetActivationName (SAN)** instruction. Then the return address along with the old activation name is provided. This is done in MIDC node 6 using the **REColor (REC)** instruction.

3.3 Bubble Sort

The analysis phase finds that the IF2 function graph contains a **LoopB** in node 3, the output of which links to the output of the function graph. No internal nodes use the output of this node. Thus, the function graph need not be split. The **LoopB** node 3 in turn contains another **LoopB**, node 4. As this node does not feed other internal nodes of the graph, splitting need not be performed. The **AElement** operations in node 4 are independent of each other and can be a part of the same MIDC thread. The graph is split at the outputs of the **AElement** nodes. We also find a **Select** node in the **LoopB**. The graph is split in the same manner as described in the binary integration example above.

Figure 11 shows the MIDC code generated and figure 12 shows the MIDC graph structure. Code generation is straightforward once analysis has been performed.

1. The upper limit of the array required by the **ALimH** in node 3 is computed using the *array descriptor*. The size and lower bound of the array is computed.

2. When the **Select** graphs are being encoded, we find that some threads do not have any input. Since a thread cannot start execution if it does not have any inputs, the MIDC nodes have to be triggered. This is done by the **TRG** instruction, see MIDC Node 11.

4 Evaluation

In this section we present the dynamic properties of our top down generation strategy, by running the Livermore and Purdue benchmark codes on our coarse grain dataflow graph simulator [Roh92]. The objectives of these measurements are to evaluate the total work, in terms of number of clusters, the average number of instructions per cluster (S_c) and average number of inputs per cluster (I_c) and the average number of inputs per instruction (I_i).

Table 2 gives the total number of clusters, instructions and matches for the Livermore and Purdue benchmarks.

References

[ACC+90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Portfield, and B. Smith. The Tera computer system. In *Proc. Int. Conf. on Supercomputing*, pages 1–6. ACM Press, June 1990.

[Can89] D. C. Cann. Compilation techniques for high performance applicative computation. Technical Report CS-89-108, Colorado State University, 1989.

[CP90] D. E. Culler and G. M. Papadopoulos. The explicit token store. *Journal of Parallel and Distributed Computing*, 10(4), 1990.

[MSA+85] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and Iteration in a Single Assignment Language: reference manual version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[PC90] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Int. Ann. Symp. on Computer Architecture*, June 1990.

[Roh92] Lucas Roh. IDIAS: a dataflow machine simulator. Technical Report CS-92-112, Colorado State University, Computer Science Department, March 1992.

[SG85] S. K. Skedzielewski and John Glauert. IF1: An intermediate form for applicative languages reference manual, version 1. 0. Technical Report TR M-170, Lawrence Livermore National Laboratory, July 1985.

[SKS+92] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura. Thread-based programming for the EM-4 hybrid dataflow machine. In *Int. Ann. Symp. on Computer Architecture*, 1992.

[WSYR86] M. Welcome, S. Skedzielewski, R. K. Yates, and J. Ranelleti. IF2: An applicative language intermediate form with explicit memory management. Technical Report TR M-195, University of California - Lawrence Livermore Laboratory, December 1986.

Program	Size	Clusters	Instructions	Matches	S_c	I_c	I_i
l1	20	123	1031	838	8.38	6.81	0.81
l2	20	895	8638	5652	9.65	6.32	0.65
l3	20	84	308	242	3.67	2.88	0.79
l4	100	86	727	648	8.45	7.53	0.89
l5	20	102	685	668	6.72	6.55	0.98
l6	20	1345	6696	5206	4.98	3.87	0.78
l7	20	129	2403	1080	18.63	8.37	0.45
l8	20	1053	6019	3776	5.72	3.59	0.63
l9	20	634	11744	9494	18.52	14.97	0.81
l10	50	1556	53624	31525	34.46	20.26	0.59
l11s	20	98	543	540	5.54	5.51	0.99
l12	20	603	3547	2558	5.88	4.24	0.72
l13	50	355	4401	3960	12.40	11.15	0.90
l14	20	231	2750	1979	11.90	8.57	0.72
l15	20	990	7863	5368	7.94	5.42	0.68
l16	20	539	2272	2609	4.22	4.84	1.15
l17	20	138	1632	1497	11.83	10.85	0.92
l19	20	1645	15407	14488	9.37	8.81	0.94
l21	25	25686	113764	88022	4.43	3.43	0.77
l22	20	1833	7790	5934	4.25	3.24	0.76
l24	20	116	455	428	3.92	3.69	0.94
p1	100	202	1007	705	4.99	3.49	0.70
p2	20	1263	5825	3366	4.61	2.67	0.58
p3v	20	283	1228	644	4.34	2.28	0.52
p3	20	1283	4032	3404	3.14	2.65	0.84
p4v	20	13	64	31	4.92	2.38	0.48
p4	20	63	272	184	4.32	2.92	0.68
p5v	20	3209	13536	10876	4.22	3.39	0.80
p5	20	3366	17138	16113	5.09	4.79	0.94
p7	20	2063	7447	7408	3.61	3.59	0.99
p8	20	690	5247	3562	7.60	5.16	0.68
p9	20	3713	40870	25180	11.01	6.78	0.62
p10	20	1458	8254	8068	5.66	5.53	0.98
p11v	20	325	1562	927	4.81	2.85	0.59
p11	20	375	1645	1332	4.39	3.55	0.81
p12	20	2879	10481	12185	3.64	4.23	1.16
p13	20	123	643	592	5.23	4.81	0.92
p14	20	667	14233	4920	21.34	7.38	0.35
p15	10	4995	55407	29206	11.09	5.85	0.53
Total	66941	449608	321697	6.72	4.81	0.72	

Table 2: Dynamic counts for Livermore loops (l) and Purdue benchmarks (p). v at the end of the program name signifies vector execution. S_c - Instructions/Cluster, I_c - Matches/Cluster, I_i - Matches/Instruction.

A Livermore Loop 3

```
function Loop3p(v1, v2: array[integer] returns Integer)
    for e1 in v1 dot e2 in v2
        x := e1 * e2;
    returns value of sum x
    end for
 end function % Loop3p
```

Figure 1: Sisal Code for Livermore Loop 3.

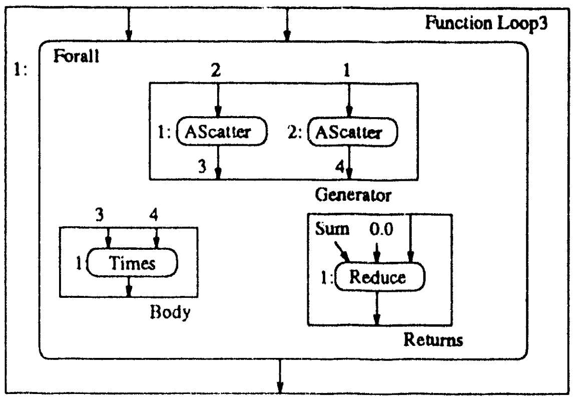


Figure 2: IF2 representation of Livermore Loop 3.

```
% Entry Code for Function loop3.
% Input Interface.
F loop3 1 2 <(3 1)(3 2)>

% Output Interface.
F loop3 2 2 <()>

% Body of Function loop3.
N 3 7 2 <(!R0 I1 4 1 0)(!R0 I2 4 2 0)
(!R0 R3 4 3 0)(!R0 R5 4 4 0)
(R4 R6 4 5 0)(!R4 R7 4 5 0)>
R0, R0, R1 = UPK I1
R0, R0, R2 = UPK I2
RCK R1 R2 ; Make sure arrays are of equal lengths.
R3 = MOI R1 "8" ; Chunk size = 8
R5 = SUI R1 R3
R6 = DVI R1 "8" ; Calculate the number of chunks.
R4 = EQI R3 R0
R7 = ADI R6 "1"

N 4 14 5 <(!R12 R4 9 4 0)(!R0 R5 7 3 0)
(!R0 R6 6 1 0)(!R12 R7 8 1 0)
(!R0 R8 6 2 0)(!R12 R9 8 2 0)
(!R12 R11 8 3 0)(!R12 R11 9 3 0)
(!R0 R13 5 3 0)(!R0 R14 5 2 0)> ; Start of Forall loop.
R3 = EIX I1 ; Extract loop index for future use.
R4 = SIX R3 I4 ; Set loop index for the odd chunk.
R12 = EQI I3 R0
R1 = SUI I5 "1"
R2 = GANS "1" R1 "8" ; Generate an Activation Name stream.
R5 = SCSX R2 R3 ; Save the old color and index information.
R6 = POL I1 R2 ; Proliferate the loop arguments.
R7 = SIX I1 I4
R8 = POL I2 R2
R9 = SIX I2 I4
R10 = EQU I3
R11 = SIX R10 I4
R13 = EQU I5 ; Number of chunks needed for the reduce.
R14 = EQU "0.0D0" ; Set initial value for the reduce operation.

N 5 3 3 <(R3 R1 2 1 0)(!R3 R1 5 2 0)
(!R3 R2 5 3 0)>
R1 = ADD I2 I1 ; reduce the partial sums.
R2 = SUI I3 "1" ; loop till #chunks = 0
R3 = EQI R2 R0

N 6 3 2 <()>
R2, R0, R0 = UPK I1
R1 = EIX I1
FCH R2 R1 "8" "7D1" ; fetch chunk for vector X.
R3, R0, R0 = UPK I2
FCH R3 R1 "8" "7D2" ; fetch chunk for vector Y.

N 7 3 3 <(!R0 R3 5 1 3)>
V1 = MUVD I1 I2 ; Multiply vectors.
R2 = RSM V1 "8" ; reduce vector sum.
R3 = LAB R2 I3 ; recolor the result.

N 8 3 3 <()>
R2, R0, R0 = UPK I1
R1 = EIX I1
FCH R2 R1 I3 "9D1" ; fetch odd size chunk for vector X.
R3, R0, R0 = UPK I2
FCH R3 R1 I3 "9D2" ; fetch odd size chunk for vector Y.

N 9 3 4 <(!R0 R3 5 1 3)>
V1 = MUVD I1 I2 ; odd size chunk is reduced as above.
R2 = RSM V1 I3
R3 = SIX R2 I4
```

Figure 3: MIDC code for Livermore Loop 3.

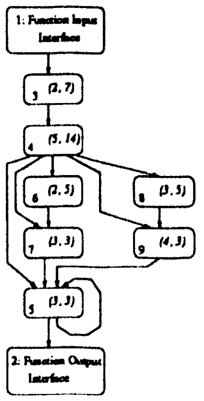


Figure 4: MIDC graphs structure for Livermore Loop 3. Legend in Nodes: Node number (#inputs, #instructions).

B Binary Integration

```
function F(X: real   returns real)
    3.0*X*X*X + 2.0*X*X + 5.0
end function  %  F

function Trap(L, R : real  returns real)
    (R-L) * (F(L) + F(R))/2.0
end function % Trap

function Area(L,R,Est,Tol: real returns real)
    let
        Mid := (L + R)/2.0;
        A1 := Trap(L, Mid); A2 := Trap(Mid, R);
        Newest := A1 + A2
    in
        if abs(Est - Newest) < Tol
        then newest
        else Area(L, Mid, A1, Tol/2.0) + Area(Mid, R, A2, Tol/2.0)
        end if
    end let
end function % Area
```

Figure 5: Sisal Code for Binary Integration.

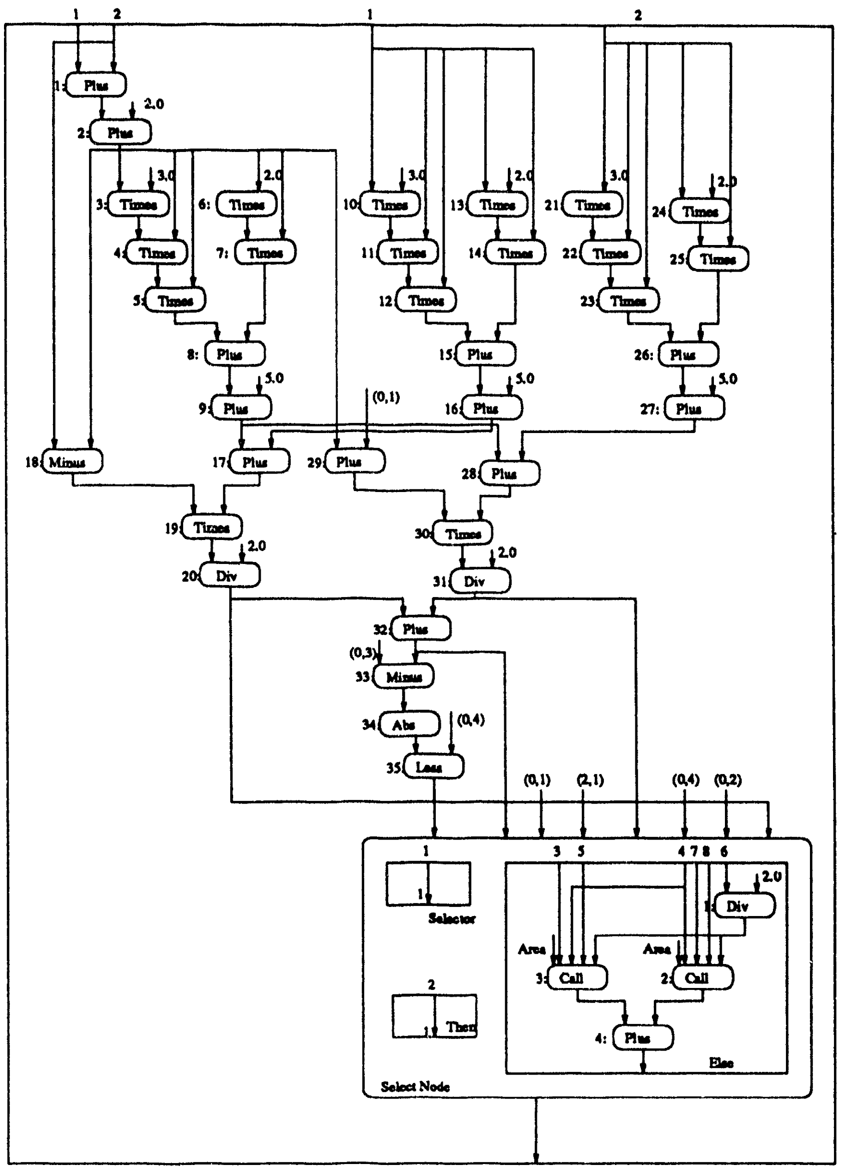


Figure 6: IF2 for Binary Integration.

```
% Function Interface for Function Area.
% Input Interface.
F area 1 4 <(5 1)(5 2)(5 3)(5 4)>

% Output Interface.
F area 2 2 <()>

% Body of Function Area.
N 5 35 4 <(R35 R32 8 1 0)(!R35 I1 6 3 0)
(!R35 R2 6 2 0)(!R35 R20 6 1 0)
(!R35 I4 6 6 0)(!R35 I2 6 5 0)
(!R35 R31 6 4 0)>
R1 = ADR I1 I2 ; T1 = L + R
R2 = DVR R1 "2.0" ; Mid = (L + R)/2
R3 = SUR R2 I1 ; Trap(L, Mid), inlined.
R4 = MUR I1 "3.0"
R5 = MUR R4 I1
R6 = MUR R5 I1
R7 = MUR I1 "2.0"
R8 = MUR R7 I1
R9 = ADR R6 R8
R10 = ADR R9 "5.0" ; F(L), inlined.
R11 = MUR R2 "3.0"
R12 = MUR R11 R2
R13 = MUR R12 R2
R14 = MUR R2 "2.0"
R15 = MUR R14 R2
R16 = ADR R13 R15
R17 = ADR R16 "5.0" ; F(Mid), inlined.
R18 = ADR R10 R17
R19 = MUR R3 R18
R20 = DVR R19 "2.0" ; Trap(L, Mid) completes.
R21 = SUR I2 R2
R22 = MUR I2 "3.0"
R23 = MUR R22 I2
R24 = MUR R23 I2
R25 = MUR I2 "2.0"
R26 = MUR R25 I2
R27 = ADR R24 R26
R28 = ADR R27 "5.0" ; F(R), inlined.
R29 = ADR R17 R28
R30 = MUR R21 R29
R31 = DVR R30 "2.0" ; Trap(Mid, R) completes.
R32 = ADR R20 R31 ; Newest = A1 + A2
R33 = SUR I3 R32
R34 = ABR R33
R35 = LTR R34 I4 ; T2 = abs(Est - Newest).
```

```
N 6 13 6 <(!R0 R3 1 1 0)(!R0 R4 1 2 0)
(!R0 R5 1 3 0)(!R0 R6 1 4 0)
(!R0 R7 2 2 0)(!R0 R9 1 1 0)
(!R0 R10 1 2 0)(!R0 R11 1 3 0)
(!R0 R12 1 4 0)(!R0 R13 2 2 0)> ; Else branch.
R2 = GAN I1 ; Function Call to Area.
R3 = SAN I3 R2 ; Arguments given a new color.
R4 = SAN I2 R2
R5 = SAN I1 R2
R1 = DVR I6 "2.0" ; New tolerance calculated.
R6 = SAN R1 R2
R7 = REC R2 "7D1" ; Destination + old color.
R8 = GAN I1
R9 = SAN I2 R8
R10 = SAN I5 R8
R11 = SAN I4 R8
R12 = SAN R1 R8
R13 = REC R8 "7D2"

N 7 1 2 <(!R0 R1 2 1 0)> ; Else Continuation.
R1 = ADR I1 I2

N 8 0 1 <(!R0 I1 2 1 0)> ; Then Branch return Newest.
```

Figure 7: MIDC code for the Binary Integration Function Area.

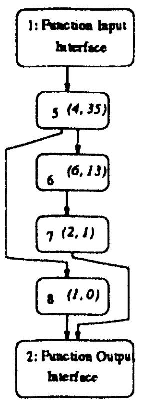


Figure 8: MIDC graph structure for Binary Integration Function Area.

C Bubble Sort

```
function Sort (Data: array[integer] returns array[integer])
    for initial
        Limit := array_limh(Data);
        B := Data;
        exchange := true
    while  Limit > array_liml(Data) & exchange repeat
        Limit := old Limit - 1;
        B, exchange :=
            for  initial
                J := array_liml(old B) - 1;
                X := old B;
                exch := false
            while J < Limit repeat
                J := old J + 1;
                Y := old X[J];
                X, exch :=
                if ( Y > old X[J + 1]) then
                    old X[J:old X[J+1]; J+1: Y], true
                elseif old exch then old X, true
                else old X, false
                end if
            returns value of X
                    value of exch
            end for
        returns value of B
        end for
end function %Sort
```

Figure 9: Sisal Code for Bubble Sort.

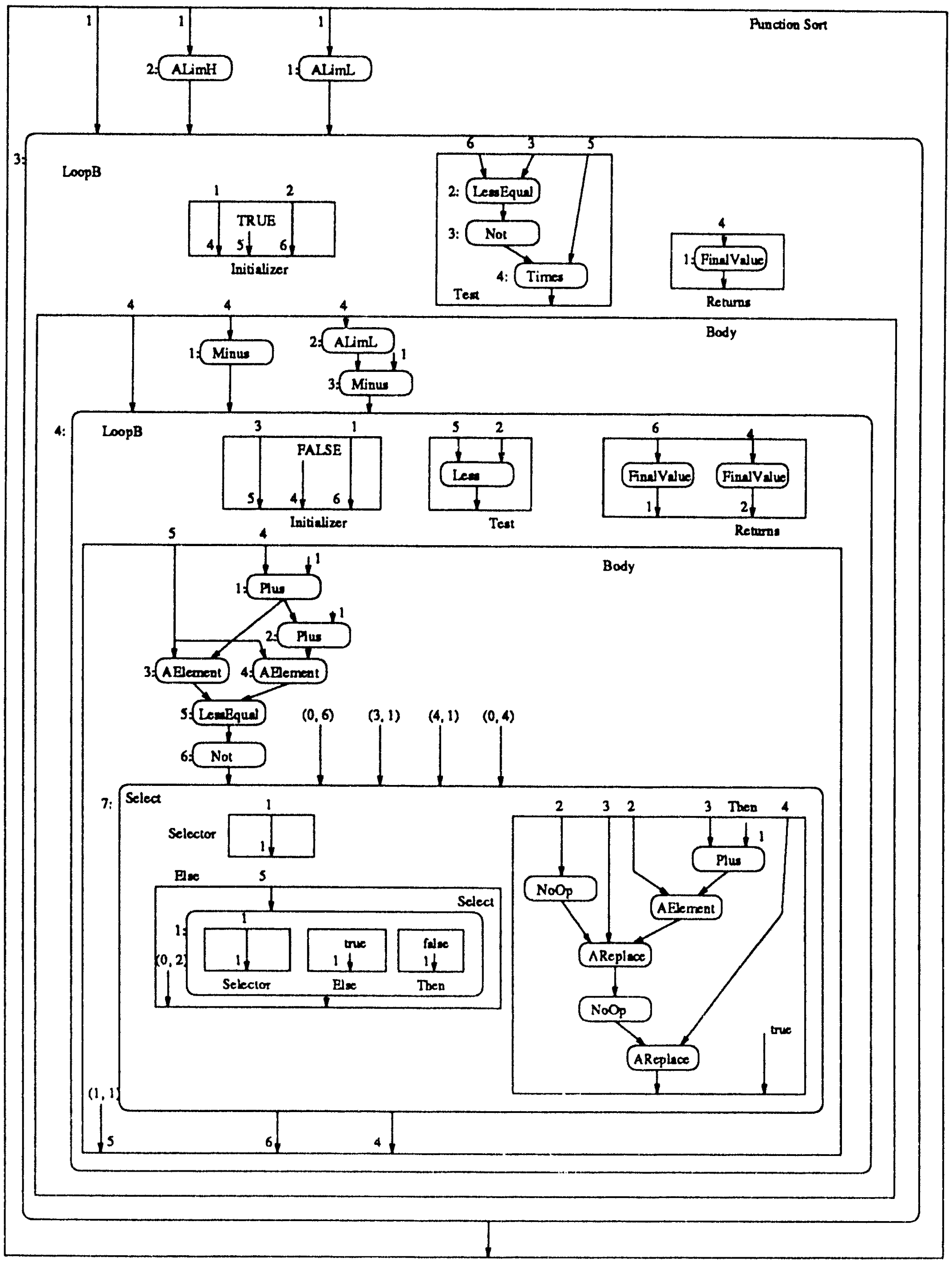


Figure 10: IF2 for Function Sort.

```
% Function Interface for Function Sort.
% Input Interface.
F sort 1 1 0 <(3 1)>

% Output Interface.
F sort 2 2 0 <()>

% Body of Function Sort.
N 3 12 1 <(!R0 R7 4 2 0)(!R0 R8 4 1 0)
(!R0 R9 4 3 0)(!R0 R10 4 4 0)
(!R0 R7 5 2 0)(!R0 R8 5 1 0)
(!R0 R9 5 3 0)(!R0 R10 5 4 0)
(!R0 R12 5 5 0)>
R7 = IIL I1 ; Init graph of outer loop.
R0, R5, R0 = UPK I1 ; ALimL of the Input array.
R8 = IIL R5 ; Increment Iteration Level
R6 = EQU "TRUE"
R9 = IIL R6
R0, R1, R2 = UPK I1
R3 = ADI R2 R1 ; ALimH of the Input array.
R4 = SUI R3 "1"
R10 = IIL R4
R11 = EIX I1 ; Extract Index of outer loop.
R12 = IIL R11

N 4 7 4 <(!R0 R3 5 6 0)(R3 R4 6 1 0)
(R3 R5 6 2 0)(R3 R6 6 3 0)
(R3 R7 6 4 0)>
R1 = LEI I4 I1 ; Test Graph of outer loop.
R2 = NOT R1
R3 = AND R2 I3
R4 = INX I1 ; Increment Index for the
R5 = INX I2 ; next iteration.
R6 = INX I3
R7 = INX I4

N 5 5 6 <(!I6 R3 2 1 0)(I6 R5 5 5 0)>
R1 = DIL I1 ; Returns graph of outer loop.
R3 = SIX R1 I5 ; Set index value for the outer value.
R5 = INX I5 ; Increment Index for next iteration.

N 6 10 4 <(!R0 I1 4 1 0)(!R0 R5 7 2 0)
(!R0 R6 7 1 0)(!R0 R7 7 3 0)
(!R0 R8 7 4 0)(!R0 R5 8 2 0)
(!R0 R6 8 1 0)(!R0 R7 8 3 0)
(!R0 R8 8 4 0)(!R0 R10 8 5 0)
(!R0 R1 4 4 0)(!R0 I1 5 1 0)
(!R0 R1 5 4 0)>
R4 = EQU "FALSE" ; Body of the outer loop.
R5 = IIL R4 ; Also, the Init for the inner loop.
R1 = SUI I4 "1"
R6 = IIL R1
R0, R2, R0 = UPK I2
R3 = SUI R2 "1"
R7 = IIL R3
R8 = IIL I2
R9 = EIX I1
R10 = IIL R9

N 7 5 4 <(!R0 R1 8 6 0)(R1 R2 9 1 0)
(R1 R3 9 2 0)(R1 R4 9 3 0)
(R1 R5 9 4 0)>
R1 = LTI I3 I1 ; Test graph of inner loop.
R2 = INX I1
R3 = INX I2
R4 = INX I3
R5 = INX I4

N 7 5 4 <(!R0 R1 8 6 0)(R1 R2 9 1 0)
(R1 R3 9 2 0)(R1 R4 9 3 0)
(R1 R5 9 4 0)>
R1 = LTI I3 I1 ; Test graph of inner loop.
R2 = INX I1
R3 = INX I2
R4 = INX I3
R5 = INX I4

N 8 7 6 <(!I6 R4 4 2 0)(!I6 R5 4 3 0)
(I6 R7 8 5 0)(!I6 R4 5 2 0)
(!I6 R5 5 3 0)>
R1 = DIL I1 ; Returns Graph of inner loop.
R4 = SIX R1 I5
R2 = DIL I4
R5 = SIX R2 I5
R7 = INX I5

N 9 10 4 <(!R0 I4 10 4 0)(!R0 I3 10 3 0)
(!R0 I2 10 2 0)(!R0 I1 10 1 0)
(!R0 R1 10 7 0)>
R1 = ADI I3 "1" ; Body of inner loop.
R2, R3, R4 = UPK I4
R5 = SUI R1 R3
RSS R2 R5 "10D5" ; AElement, node 3.
R7, R8, R9 = UPK I4
R6 = ADI I3 "2"
R10 = SUI R6 R8
RSS R7 R10 "10D6" ; AElement, node 4.

N 10 2 7 <(!R0 I1 7 1 0)(!R2 I4 11 1 0)
(R2 I4 12 1 0)(R2 I7 12 2 0)
(R2 I5 12 3 0)(!R2 I2 11 2 0)
(!R0 I1 8 1 0)>
R1 = LEI I5 I6 ; Compute conditional.
R2 = NOT R1 ; nodes, 5 & 6.

N 11 2 2 <(!R0 I1 7 4 0)(!I2 R1 14 1 0)
(I2 R2 15 1 0)(!R0 I1 8 4 0)>
R1 = TRG I1 ; Select Node, selector graph.
R2 = TRG I1 ; Trigger one of the paths.

N 12 5 3 <(!R0 R1 13 4 0)(!R0 I1 13 2 0)
(!R0 I2 13 3 0)(!R0 I3 13 5 0)>
R1 = ADI I2 "1" ; Then path of outer select.
R2, R3, R4 = UPK I1
R5 = SUI R1 R3
RSS R2 R5 "13D1"

N 13 13 5 <(!R0 R12 7 4 0)(!R0 R13 7 2 0)
(!R0 R12 8 4 0)(!R0 R13 8 2 0)>
R1 = EQU I2 ; continuation of then path
R2, R3, R4 = UPK R1 ; from MIDC node 12.
R5 = SUI I3 R3
WSS R2 R5 I1 ; AReplace
R6 = PAK R2 R3 R4
R7 = EQU R6
R8, R9, R10 = UPK R7
R11 = SUI I4 R9
WSS R8 R11 I5 ; AReplace
R12 = PAK R8 R9 R10 ; Repack the array descriptor.
R13 = EQU "TRUE"

N 14 1 1 <(!R0 R1 7 2 0)(!R0 R1 8 2 0)>
R1 = EQU "FALSE" ; Else path of the inner select.

N 15 1 1 <(!R0 R1 7 2 0)(!R0 R1 8 2 0)>
R1 = EQU "TRUE" ; Then path of the inner select.
```

Figure 11: MIDC code for the Sort.

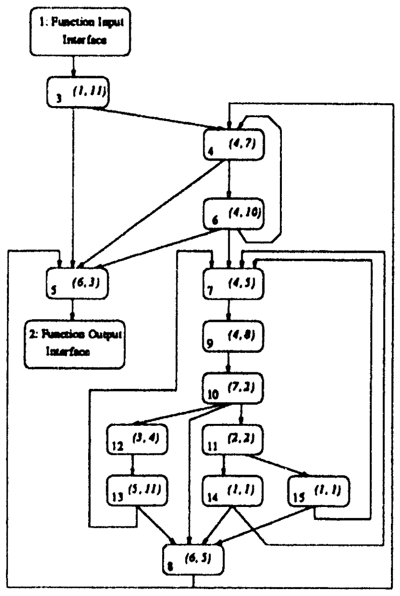


Figure 12: MIDC graph structure for Function Bubble Sort.

Overlapping Communications and Computations on NUMA Architectures

Rich Wolski and John Feo
Lawrence Livermore National Laboratory
Livermore, CA 94551

Abstract

Many of the currently available multiprocessing systems are built from commodity RISC processor chip sets. These processors reduce the latency associated with accessing memory by supporting non-strict load and store operations. These operations complete before the data is actually present in registers or transferred from registers to memory. Effectively overlapping communication and computations is the key to achieving high-performance on these systems. Present partitioning methods do not accurately model non-strict load and store operations; consequently, they generate non-optimal schedules for RISC-based machines. In this paper, we define two new nodes, WAt and RAt, to model the execution semantics of load and store operations. We show that by including these nodes in program dependency graphs with memory nodes, we can effectively overlap communications and computations, thereby, reducing critical path lengths. We present performance results for a RISC-based NUMA machine. Our study demonstrates the importance of the new nodes and the versatility of program dependency graphs with memory nodes.

1 Introduction

While older, CISC (complex instruction set computer) architectures implement strict instruction semantics, modern parallel computer systems tend to be built using RISC (reduced instruction set computer) processor technology. Many RISC processors are pipelined, load/store architectures in which all computational instructions operate on register operands. Data is moved between registers and memory by explicit load and store instructions rather than as part of each computation. Since the systems are typically pipelined, the load and store operations are non-strict; that is, control is returned to the processor before the operations complete. Moreover, several emerging new NUMA (non-uniform memory access) multiprocessor architectures implement processor multithreading as a mechanism for latency tolerance [Alv91, Nik91]. These systems hide memory latencies by rapidly switching between runnable threads when a reference occurs. Each load or store is overlapped with computation from another runnable thread.

Current partitioning methods do not adequately model load and store operations; consequently, they generate poor partitions for RISC-base systems. In this paper, we define two new nodes, WAt (write-at) and RAt (read-at), to model memory access operations. When we include these nodes in data dependency graphs with memory nodes [WolFeo92], we find that conventional partitioning algorithms, such HEF (heavy-edge first), can effectively overlap communications and computations. Most importantly, the partitions generated have shorter critical path lengths.

In Section 2, we introduce the new nodes, describe the advantages of program dependency graphs with memory nodes, and describe changes to the HEF algorithm to accommodate WAts and RAts. In Section 3, we give performance results for a RISC-based, NUMA architecture. In Section 4, we conclude and describe future work.

2 Partitioning

Parallel programs are naturally represented as data dependency graphs [Kuck81]. Within such graphs, nodes represent computations, and directed edges represent the conveyance of data between computations. The advantage of a graphical representation is that parallelism is immediately visible. Nodes contained within independent paths through a program may be executed in parallel. Traditionally, partitioning methods have assumed the macro-actor model of computation described in [Sarkar87]. The model assumes the lowest level operations to be atomic and functional. Each fundamental unit of computation executes to completion once initiated, and produces results based solely on its inputs. Fundamental computations are assumed to be strict with respect to their inputs and outputs. An operation may not execute before all of its inputs are present, and no output may be consumed until all outputs have been produced. The macro-actor model represents the communication of data from producer to consumer as a single atomic event.

In [WolFeo92], we showed the advantage of including memory nodes in dependency graphs when partitioning for NUMA architectures. The graphs more accurately reflect the complex memory structure of these machines, and enable partitioning algorithms to optimize the use of memory. The graphs identify explicitly the three distinct phases of communications [Figure 1a]:

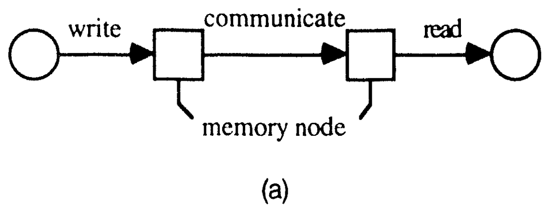
1) the producer writes its results to an accessible memory location,

2) the data is transferred by an intermediate communication facility to a memory location accessible by the consumer, and

3) the consumer reads the data from the accessible memory location.

If the producer and consumer write and read the same memory, then we may merge the two memory nodes eliminating the communication phase [Figure 1b]. By providing an architectural model that specifies the performance characteristics for each type of memory and communication channel, the compilation system can estimate execution delays accurately, and generate better program partitions and schedules.

During the communication phase, if it exists, both processors are free to execute other ready operations. Thus, data dependency graphs with memory nodes help to identify certain opportunities for overlapping communications and computations. But, for RISC-based and multithreaded architectures that can overlap every memory access operation with computations, the actual number of opportunities is much greater than that identified.



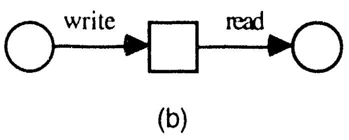


Figure 1

Therefore, to model load and store operations, we introduce two new nodes:

WriteAT: executed by a producer to transfer data from register to memory (abbreviated *WAt*).

ReadAt: executed by a consumer to transfer data from memory to register (abbreviated *RAt*).

Figure 2 illustrates the use of WAt and RAt nodes. In the figure, the computation node is marked "C", each RAt is marked "R", and each WAt is marked "W". The memory nodes read and written by the computation are colored to indicate that they have been assigned to registers. The memory nodes between the WAts and RAts are colored to indicate that they have been assigned to some memory type. [Since our focus is NUMA architectures, we have eliminated the communication phase of each memory transfer.]

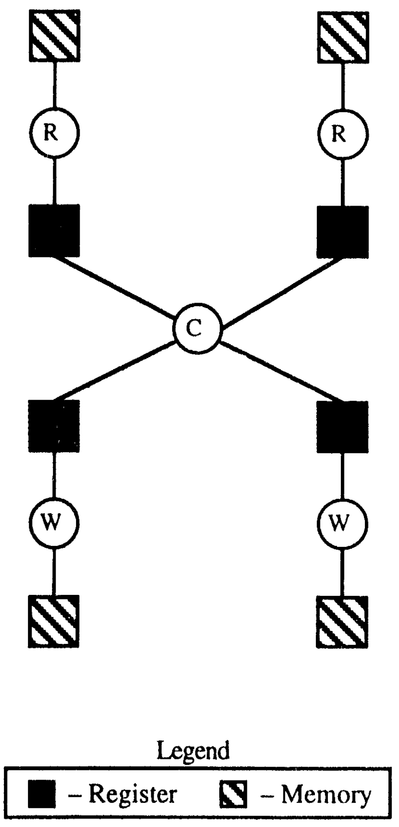


Figure 2

Unlike other nodes, however, WAt and RAt nodes may or may not imply execution semantics. The definition of their execution semantics depends on the architecture and language system. If the target architecture supports a load/store instruction set (as is the case with most RISC processors), then the RAt and WAt nodes correspond to load and store instructions respectively. Conversely, if the architecture and language system support direct memory addressing at the instruction level, then the RAts and WAts simply carry cost information but no execution semantics.

In RISC-based NUMA architectures, computations read and write only registers, and are strict. No computation may fire until all of its register inputs are available, and no register output may be consumed until all have been produced. However, load and store operations are not strict. Once executed, the processor is free to continue execution of independent instructions even though the inputs and outputs of these instructions are not yet stable. Since a WAt and RAt have execution semantics, our architectural model defines an execution cost for each node.

The heavy-edge-first (HEF) algorithm attempts to reduce program critical path by assigning the edges carrying the largest volumes of data (the heaviest edges) to the fastest memories. All memory nodes are initially colored with a default memory or communication type and sorted according to volume. The memory node incident on each edge (considered one at a time in the order specified by the sort) is speculatively colored with every other possible memory type. After each coloring, the graph's critical path is calculated, and the coloring that yields the shortest critical path is accepted. The algorithm terminates when all edges have been examined. Pseudocode for HEF is

```
HEF(graph)

{ assign all memory nodes in graph the
  default memory color;

  sorted_list = SORT(edges by volume);

  while(sorted_list is not empty)
    { best_color = default color;
      edge = remove_biggest(sorted_list);
      for(each memory color except default)
        { color memory node incident on edge;
          calculate critical path length;
          if(path length is better)
             best_color = color;
        }
      color memory node with best_color;
    }
}
```

The complexity of HEF is $O(k\ N^2)$ where k is the number of different possible memory types. Each node is colored with k different memory types, and after each coloring, the graph distances are adjusted. The recalculation of graph distances is $O(N)$, there are $O(k\ N)$ operations performed per node. Since each node is accessed once, the overall complexity of $O(k\ N^2)$.

Similar to the work outlined in [Sarkar87], HEF focuses on the edges carrying the greatest communication volume as being the sources of the greatest execution overhead. These edges will be considered for the fastest memory colors first thereby reducing the greatest amount of communication overhead. If a private color is selected for a given memory node, the threads containing its producer and consumer are merged according to the merge heuristic in [WolFeo92]. Briefly, the two threads to be merged are considered pre-sorted lists. The merge algorithm constitutes essentially a single iteration of a merge sort where the output sorted list is the new thread.

The semantics of load and store operations, as implemented by most architectures, also requires that the cost statistics for RAts and WAts be calculated differently than for other nodes. Specifically, for RAts and WAts we must charge the cost of accessing a non-register colored memory node after the execution of the RAt or WAt since that is how delayed memory operations are typically implemented. A load instruction can be thought of as reading a memory location and writing a register. The delay until the load executes, however, is not incurred as a result of its input memory type. Rather, the node executes and some time later its output register type contains a copy of the data in its input memory type. Similarly, a WAt does not read a register and write a memory location in terms of the cost model. The WAt fires and then data is moved from register to memory. Graphically, we depict this relationship in Figure 3.

Notice that neither the RAt nor the WAt incur an input cost; both of their read costs are zero. In both cases, we charge the latency associated with their incident memory type after their execution.

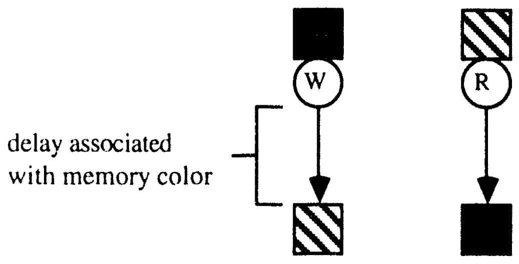


Figure 3

3 Results

For this study, we used a communication topology similar to that provided by the BBN TC2000 [BBN90]. The TC2000 supports essentially 4 forms of memory: registers, local private, block shared, and interleaved. We assumed 100 iterations per loop and a spawn cost of 50 clock cycles. A lock per processor must be assumed, and a pointer reference is incurred as each loop slice is allocated making 50 clocks a reasonable estimate.

As a cost metric, we use the mean percentage improvement in the length of the critical path before and after partitioning over a set of 100 test graphs. We use a combination of IF1 [Skedz85] and IF2 [Ran87] (referred to as IFX) to graphically represent parallel programs. Initially, we assume each computation is assigned to its own processor, and that all communications use shared memory as the default type. We define the length of the critical path in the initial graph as *the initial length*. We then partition the graph using a particular partitioning algorithm, and calculate the critical path length of the final graph (*the final length*). The percentage improvement is

$$\frac{\textit{initial path length} - \textit{final path length}}{\textit{initial path length}} * 100$$

For the memory characteristics of the TC2000, we obtained the improvement results shown in Table I. HEF in combination with our methods for assigning costs throughout the IFX hierarchy extracts reasonable critical path improvements.

Program	HEF Improvement
GJ	89.9 %
PIC	86.2 %
CG	88.1 %
RICARD	79.6 %
SIMPLE	67.7 %

Table I

3.1 Overlap

To effectively exploit RISC and multithreaded architectures, the partitioning and scheduling system must be able to schedule load and store operations so that the communication latencies are masked. Within each thread, computations that take and produce local data values should be executed during the delay between when a remote fetch is initiated, and its consuming computation executes. The notion of initiating a non-strict load before its data is required and then "filling" the resulting gap is occasionally referred to as *prefetching* in the literature.

To expose the effectiveness of our approach for prefetching, we define the following three statistics for each node:

Overlap(n): the amount of computation that takes place between the initiation of n's earliest load, and the execution of n itself,

Latency(n): the amount of computation that takes place between the initiation of n's earliest read, and the time when n's last input is available,

Delay(n): the amount of time between when n's last input is available and when n actually executes.

We assume that each node has been assigned to some thread, and that communications local to the thread cannot be overlapped (i.e. local communication takes place in registers). We do not impose limits on the number of registers available, nor do we assume that memory is scarce. Graphically, the overlap, latency, and delay statistics are depicted in Figure 4.

Intuitively, the *overlap* is the computation that takes place between the earliest prefetch and the actual execution of a node. It is a measure of the degree to which the partitioning methods are filling the latency gaps. However, since a node may be artificially delayed due to partitioning, we measure the amount of computation between the prefetch and the earliest time the node could execute (i.e. when its last input is ready). In the ideal case, each node would execute as soon as all of its inputs were available and all of the communication necessary to fetch those inputs would be overlapped with computation. The *latency* measure captures the degree to which our methods achieve this goal. The *delay* is the time between its earliest possible execution and when it actually executes. That is, the delay measures how long each computation is executed after its earliest possible start time. We would like to see the delay values for nodes on the program critical path be as small as possible. The algorithms should seek to push node delays off the critical path so critical nodes start as soon as possible. Note, however, that if a node is delayed due to partitioning, its start time may actually be shorter than if it were executed in parallel. Delay, therefore, does not necessarily indicate that a node should not have been sequentialized. Rather the absence of delay indicates that a node starts as soon as it is able.

Figure 5 shows the overlap, latency, and critical delay (delay values for nodes on the critical path) for graphs from a Conjugate Gradient Program (CG) The absence of any value for a graph implies either that the entire graph has been sequentialized, or that the graph is empty.

For the graphs where values are recorded, the latency and overlap values are either very close together or the critical delay value is very low. In the former case, all of the overlap is covering latency. That is, none of the overlap is covering delay induced by partitioning. In the latter case, there is delay incurred by various nodes, but it is not on the graph critical path. To allow the figure to scale properly, we omit the data for two graphs from Figure 5. Those values are:

Overlap	Latency	Critical Delay
57990	6702	2468
200500	10560	239

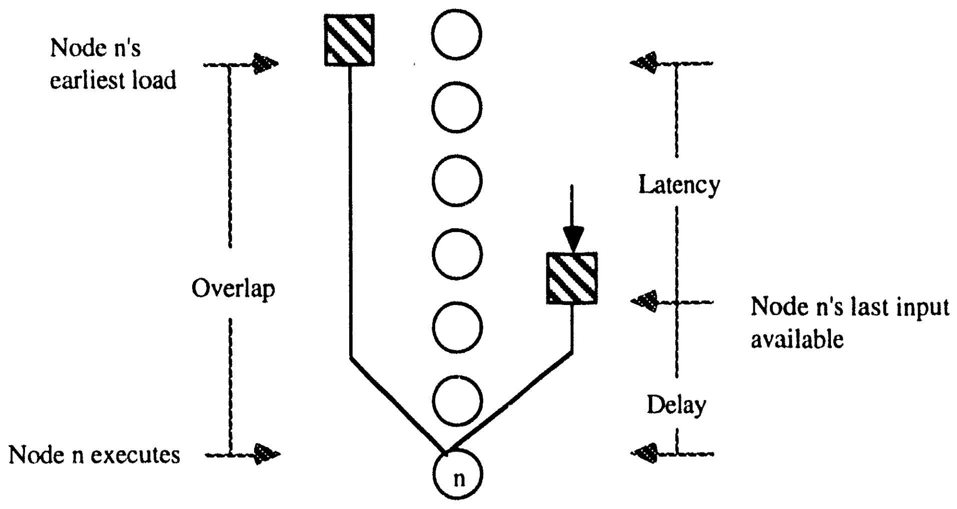


Figure 4

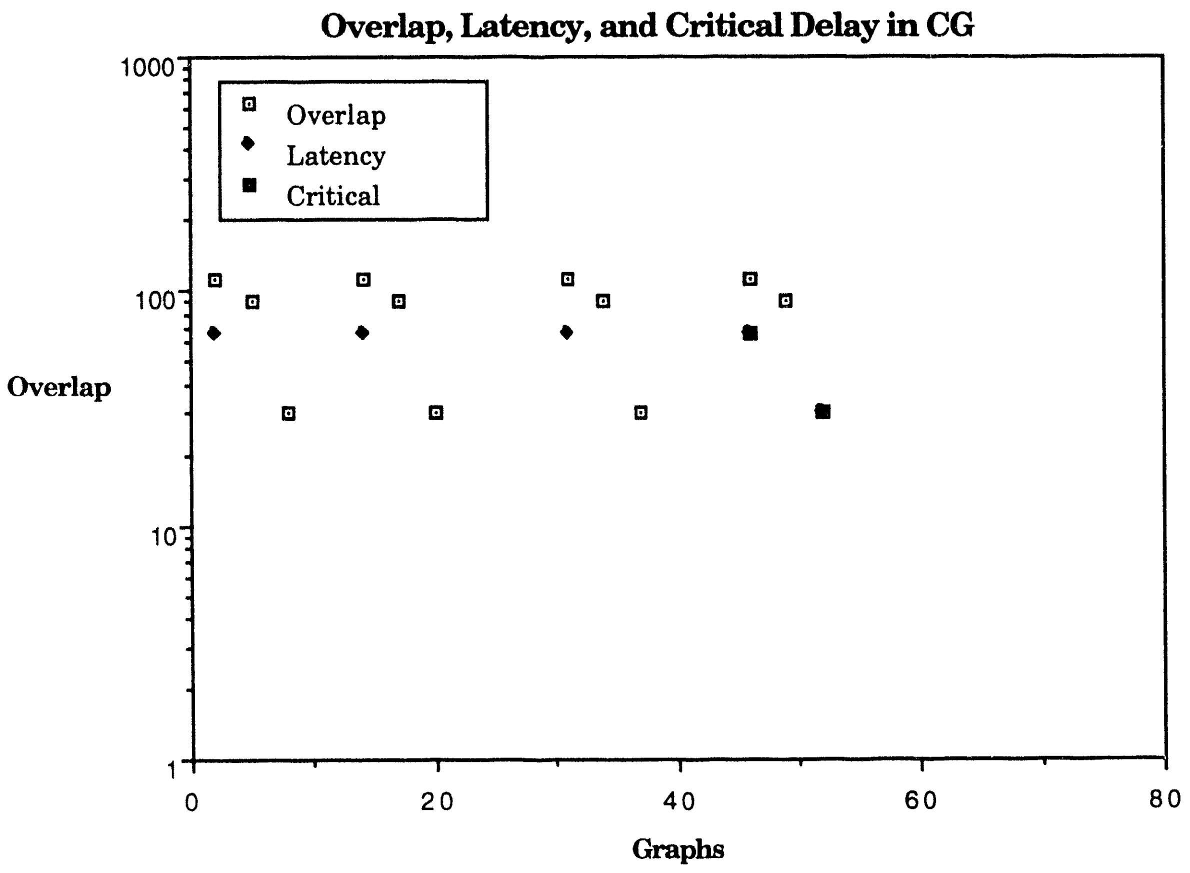


Figure 5

These figures show particularly good latency tolerance and non-critical delay values. Clearly, for CG the HEF partitioning algorithm is masking latency.

3.2 Gaps

Another important question concerns the idle time present in each thread. If that idle time is reasonably substantial, the overhead associated with multithreading might prove tolerable. Further, even in the presence of very fast multithreading support (i.e. supported by hardware), the presence of large gaps indicates that a large number of threads may be assigned to a single processor thereby reducing the processor requirements for a program. Multithreading has the additional advantage that it responds well in the presence of incomplete or erroneous analysis.

The gap within each thread is calculated as the difference between the completion time of each node, and the start of its successor. If the successor does not begin executing immediately, the processor executing that thread becomes idle [Figure 6].

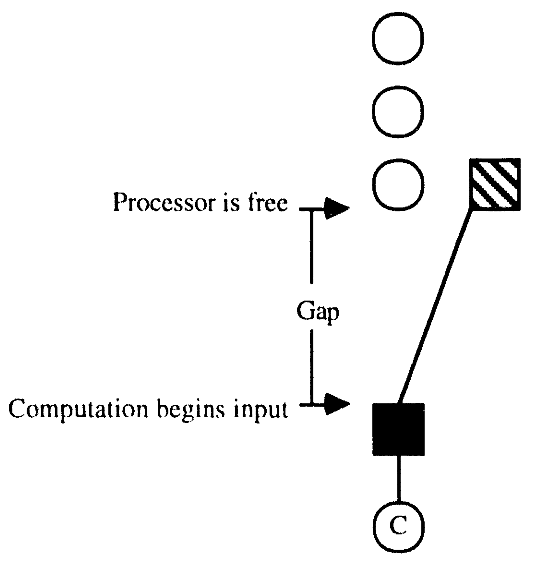


Figure 6

The processor becomes free after the last output from a computation node is written to a register. After the execution of a WAt or a RAt, the processor is immediately free since these instructions are not executed strictly. Similarly, the processor becomes busy when a computation begins reading its first input from a register. Again, RAts and WAts have node read cost so they busy their processors only during execution.

Figure 7 shows the same data for PIC when partitioned with HEF.

We, again, omit two outlying data points of 17,560 cycles and 52,690 cycles. We contend, then, that multithreading support is desirable since it can be exploited by compile-time partitioning and adapts better in the face of data dependent execution. Software multithreading may prove beneficial in the cases where large gaps are found. In either case, the partitioning and scheduling system is able to, at the very least, locate the portions of the code where multithreading might be profitable.

4 Conclusions

Non-strict instruction execution, either in the context of RISC processing or a multithreaded architecture, allows for greater overlap of computation and communication. Further, in order to extract the maximum possible performance from these systems, parallel programs must use the non-strictness to tolerate communication latencies. Partitioning algorithms must take into account the effects of non-strict execution so that the compilation system may exploit the underlying machine on behalf of the programmer. To assist the algorithms, we introduced two new nodes: WAt and RAt.

When we examine the amount of communication that is overlapped with computation, we notice that some of the overlap is due to sequentialization by the partitioning system. Therefore, we differentiate between overlap and latency. For many graphs, the overlap and tolerated latency are identical showing that HEF is effectively exploiting non-strict instruction execution. In the cases where the two statistics differ, the resulting delay is generally not on the critical path indicating that critical computations are starting as early as possible. Further, the overall percentage decrease in critical path is reasonably large for out test codes. We conclude that HEF is an effective partitioning methodology for non-strict NUMA systems.

Finally, we investigate the opportunities for multithreading by exposing thread idle time. For the communication granularity and the amount of processor resource available on the TC2000, some opportunities exist, but the majority of the parallelism must be extracted from coarser parallel constructs such as loops.

We plan to implement these algorithms as part of the OSC optimization chain for NUMA systems. Currently, the output of HEF is partitioned IFX suitable for scheduling and code generation. To realize actual running programs, we need to develop a scheduling methodology and augment the OSC code generator to produce executable threads.

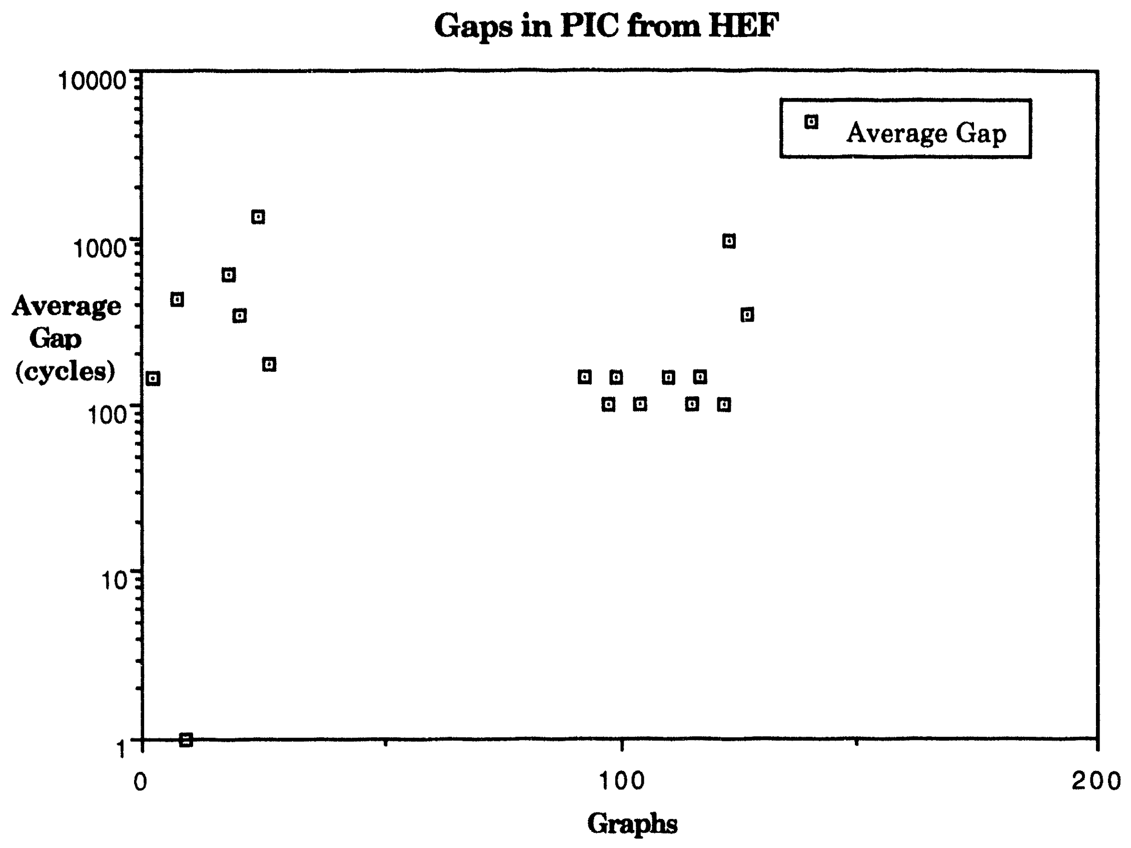


Figure 7

References

[Alv91] Alverson, G. et. al. Exploiting heterogeneous parallelism on a multi-thread multiprocessor. *Proc. Workshop on Multithreaded Computers*, Supercomputing '91, Albuquerque, NM, November 1991.

[BBN90] *Inside the TC2000, Computer*, BBN Advanced Computers Inc., 10 Fawcett St., Cambridge, MA, 02138, 1990.

[Cann89] Cann, D. C. Compilation Techniques for High Performance Applicative Computation. Ph.D. thesis, Department of Computer Science, Colorado State University, 1989.

[Kuck81] Kuck, D.J. *et al.* Dependence Graphs and Compiler Optimizations. *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pp. 207-218. January, 1981.

[McGraw85] McGraw, J. R. et. al. *Sisal: Streams and iterations in a single-assignment language, Language Reference Manual, Version 1.2.* Lawrence Livermore National Laboratory Manual M-146 (Rev. 1), Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[Nik91] Nikhil, R.S. et. al. *T: A multithreaded massively parallel architecture. *Proc. Workshop on Multithreaded Computers*, Supercomputing '91, Albuquerque, NM, November 1991.

[Ran87] Ranelletti, J. E. Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages. Ph.D. thesis, Department of Computer Science, University of California at Davis/Livermore, 1987.

[Sarkar87] Sarkar, V., *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* Stanford University Technical Report No. CSL-TR-87-328, Stanford University, 1987.

[Skedz85] Skedzielewski, S. K. and J. Glauert. *IF1 - An intermediate form for applicative languages.* Lawrence Livermore National Laboratory Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985.

[WolFeo92] Wolski, R. and Feo, J., Program Partitioning for NUMA Multiprocessor Computer Systems, Proceedings of the 2nd Sisal Users' Conference, Lawrence Livermore National Laboratory CONF-9210270, pp. 111-137. December 1992.

Compiling Technique Based on Dataflow Analysis for Functional Programming Language *Valid*

Eiichi Takahashi Rin-ichiro Taniguchi Makoto Amamiya

Department of Information Systems,
Kyushu University
Kasuga-shi, Fukuoka-ken 816, Japan

Abstract

This paper presents a compiling method to translate the functional programming language Valid into object code which is executable on a commercially available shared memory multiprocessor, Sequent Symmetry S2000. Since process management overhead in such a machine is very high, our compiling strategy is to exploit coarse-grain parallelism at function application level, and the function application level parallelism is implemented by a fork-join mechanism. The compiler translates Valid source programs into controlflow graphs based on dataflow analysis, and then serializes instructions within graphs according to flow arcs such that function applications, which have no data dependency, are executed in parallel. We report the results of performance evaluation of the compiled Valid programs on a Sequent S2000 and discuss the usefulness of our method by comparing it with C and SISAL compilers.

1 Introduction

Many programming languages for parallel processing have been proposed recently. Among those languages, functional programming languages have various attractive features, due to their pure formal semantics, for writing short and clear programs as well as verifying and transforming programs automatically. These merits of functional programs are more evident in writing programs for parallel processing [1]. For the efficient execution of functional programs, several compiling techniques based on dataflow and reduction models, have been proposed [2, 3] and implemented on commercially available parallel machines or simulators [4, 5, 6], such as the $\langle\nu, G\rangle$-machine by Thomas Johnsson [4], the Process-Oriented Dataflow System by Lubomir Bic [6] and the SISAL system [7]. The $\langle\nu, G\rangle$-machine is an implementation of a graph reduction machine, which is an abstract machine on a shared memory multiprocessor. $\langle\nu, G\rangle$-machine code, which is the object code of the $\langle\nu, G\rangle$-machine, is generated by compiling programs in the functional programming language Lazy ML. PODS is an implementation of a thread level dataflow model on a hypercube multicomputer. The object code of PODS is generated by compiling programs in the dataflow programming language Id. SISAL is a research language for investigating issues in parallel processing, especially for numerical computing. SISAL runs on (or is in development for) conventional sequential machines, shared memory multiprocessors, and vector processors, as well as the Manchester dataflow machine. The SISAL parser produces code in IF1, an intermediate graph language used by all implementations. IF1 programs are formed into a monolithic program or module and then optimized and translated into a second intermediate form, IF2, by a machine-independent optimizer, which applies 13 optimizations, such as function inlining and dead code removal. IF2 code is next given to IF2PART for parallelization. On concurrent machines, IF2PART concurrentizes product-form loops automatically through a partitioning algorithm based on parallel nesting level and cost estimates. On vector machines, IF2PART vectorizes innermost product-form loops automatically. The output of IF2PART is given to CGEN for C and FORTRAN code generation, and the result code is compiled by a C or FORTRAN compiler.

In this paper, we present a compiling method based on dataflow analysis to translate the functional programming language *Valid* [8] into object code which is executable on a commercially available shared memory multiprocessor, Sequent Symmetry S2000. *Valid* is a high-level programming language designed for dataflow machines. A *Valid* program is constructed with function definitions and expressions. Sequent Symmetry, the target machine of the *Valid* compiler, incorporates 20 microprocessors (Intel 80486) and a common memory, all linked by a high-speed bus. Fine-grain parallelism is not useful for conventional computers. Since processors have to carry out both user programs and process management, the overhead of process management becomes excessive. Therefore, we employed coarse-grain parallelism at function application level. In function application level parallelism, function applications are implemented by a fork-join mechanism. A new child process is created in a function application, and the newly created child process executes its function instance concurrently with other processes. When a parent process encounters a synchronization point before its child process completes execution, the parent process suspends until the child process terminates. In our implementation, only arguments and addresses for return values of a function instance are required as parameters in the creation of a new child process.

The *Valid* compiler has two phases. In the first phase, the compiler constructs controlflow graphs from *Valid* source programs through dataflow analysis. In the graphs, a node corresponds to a source level instruction and an arc shows a controlflow, which reflects data dependency. In the next phase, the compiler partitions a graph into several parts, in which as many function application nodes as possible are included except for their descendants. All of the function applications in each part are executable in parallel, since they have no data dependency on each other. The compiler serializes each part according to arcs. The source program semantics is preserved because of the Church-Rosser property. Then the compiler translates the serialized code into target machine code.

In the next section, we describe the compiler. In section 3, we describe technical details of the implementation. Then, in section 4, we report the results of performance evaluation of the compiled *Valid* programs on Symmetry, and discuss the usefulness of our method by comparing it with the SISAL compiler.

2 Compiler

The compiling process consists of two phases. In the first phase, the compiler constructs controlflow graphs from *Valid* source programs with dataflow analysis. In the next phase, the compiler serializes the graphs and then translates into the target machine code.

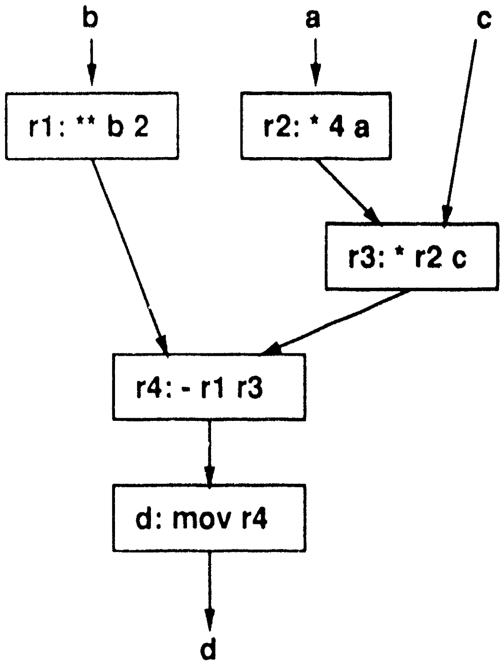


Figure 1: A controlflow graph

2.1 Graph – phase I

In phase I, the compiler constructs controlflow graphs, which are DAG and correspond to function definitions in *Valid* source programs. In the graphs, a node corresponds to a source level instruction and an arc shows a controlflow, which reflects data dependency. For example, Figure 1 shows a graph for the following expression.

```
d = b ** 2 - 4 * a * c
```

Figure 2 and 3 show a graph of a conditional expression and a graph of a recursive expression (tail recursive) respectively.

Conditional expression

Figure 2 shows the graph of a conditional expression:

```
if Ec then Et else Ef
```

If Ec is true, then the expression Et is evaluated. If Ec is false, then the expression Ef is evaluated. In Figure 2, Ec, Et and Ef are graphs, which correspond to the above expressions Ec, Et and Ef respectively. The compiler generates a pair of *sw* and *merge* nodes from a conditional expression. The *sw* node has one operand, of which the value is either true or false.

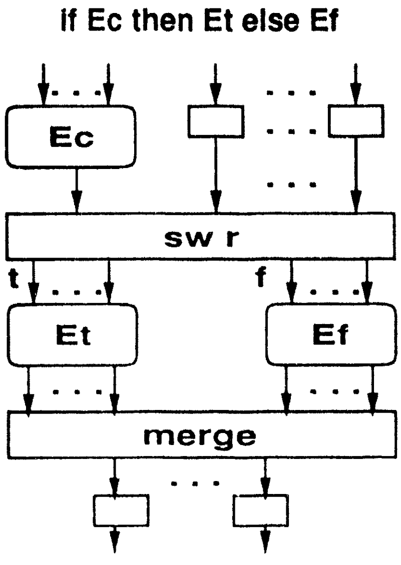


Figure 2: Graph of conditional expression

Only the *sw* node indicates nodes with special arcs, *t* arcs and *f* arcs. The *sw* node indicates nodes in the graph Et with *t* arcs and nodes in the graph Ef with *f* arcs. Nodes which are indicated with *t* or *f* arcs are top nodes in the graph Et or graph Ef.

Data dependencies between nodes in the graph Et or Ef and others are shown with arcs to the *sw* node or arcs from the *merge* node.

Recursive expression

Figure 3 shows the graph of a recursive expression:

```
for (v1,..,vn) init (E1,..,En) body
  if Ec then return Et
  else recur(R1,..,Rn)
```

Variables v1,..,vn are initialized to expressions E1,..,En, respectively. Then, the loop body, which is a conditional expression in the case above, is evaluated. In general, the loop body includes a conditional expression, which controls recursive evaluation. In the above, if the boolean expression Ec is true, the **return** expression is evaluated. It is the result of the recursive expression. If Ec is false, the **recur** expression is evaluated. It corresponds to a recursive call to the loop body considered to be a function which has no name. The above expression is tail recursive. In this case, the compiler constructs a graph, which stands for iteration. The **recur** expression is expressed as a jump node. The jump node has one operand, which is a label of a $loop_i$ node. The jump node shows that control transfers to the node named by the label which is

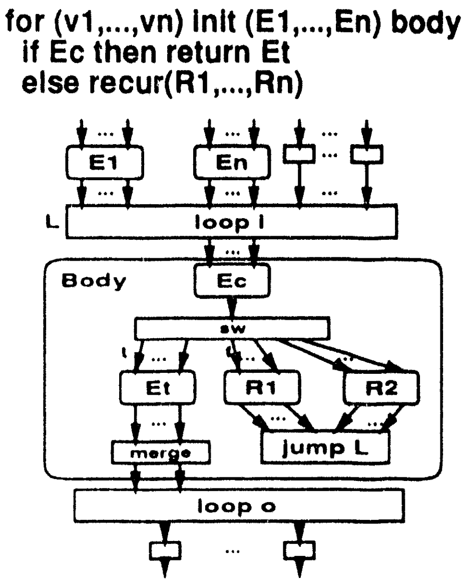


Figure 3: Graph of recursive expression

the value of its operand. Data dependencies between nodes in the graph body and others are shown with arcs to $loop_i$ nodes and arcs from $loop_o$ nodes.

Parallel expression

Parallel expression is used to write parallel executable units explicitly. For example, in the following parallel expression, the parallel body u*u preceded by the reserved word **body** is a parallel executable unit [8].

```
foreach u in [1..5] body u*u
```

The above parallel expression yields the array [1, 4, 9, 16, 25], which comprises the squares from 1 to 5. The semantics of parallel expression is based on the fork-join concept. In the above expression, parallel bodies u*u, for each of which argument u is bound to an integer value from 1 to 5 respectively, are forked, and executed in parallel. They are joined eventually, and the results are packaged as an array. Parallel expression corresponds to the **for** expression of SISAL carried out in a distributed manner, except that it is explicit. In general, the number of forked parallel bodies, which are specified with expressions preceded by the reserved word **in**, such as ranges or arrays or lists, is far larger than the number of available processors. Therefore, parallel expression is translated into code which considers parallel bodies as virtual processes, and distributes them equally to real processes created according to the number of processors. The compiler defines a new function of which the body is a recursive expression to iterate the execution of the parallel body n times, where n is:

$$n = \frac{N_{vp}}{N_{rp}}$$

,where N_{vp} is the number specified by the expression preceded by **in**, and N_{rp} is the number of available processors. The compiler translates the parallel expression into a recursive expression, which forks the application of the function N_{rp} times and joins them.

2.2 Code scheduling – phase II

In phase II, the compiler partitions a graph into several parts, serializes each part according to the arcs, and then translates the serialized code into target machine code.

Partitioning

The compiler partitions a graph at synchronization points, which correspond to function application nodes (*call* nodes). Synchronization points are extracted as follows:
Because of arcs based on data dependencies, child nodes of a *call* node are divided into three types.

type 1 A node which has the destination of its parent *call* node as its operand.

type 2 An *sw* node or $loop_i$ node.

type 3 A *merge* node or $loop_o$ node.

Because of a scope rule of *Valid*, nodes may have a number of type 1 and type 2 nodes but only one type 3 node as their child nodes. When a *call* node has a number of type 1 and type 2 nodes as its child nodes, a synchronization point is placed between the *call* node and its child nodes in order to preserve the semantics of the source program. When the *call* node has a type 3 node as its child node, a synchronization point is placed between that type 3 descendant of the *call* node and its own type 1 or type 2 child nodes, if any. At synchronization points, the compiler inserts *join* nodes, which have a label of the *call* node associated with them as operands.

Code scheduling

In the graphs, arcs reveal partial orders of operations. A parent node operation has to be completed before its child nodes operations start. Since function applications take more time than other operations, and they are processed by child processes of the current process, before execution of their child nodes operations, other executable operations, if any, should be executed.

The compiler traces the graph along arcs based on depth first search, and rearranges nodes in tracing order. When the compiler selects the next node, the compiler selects a node among child nodes of which parent nodes have been selected already. However, *join* nodes are selected when there are no nodes but *join* nodes. When the compiler schedules descendants of *sw* nodes or $loop_i$ nodes, to preserve the semantics of the source programs, the compiler applies the tracing method from *sw* nodes or $loop_i$ nodes to *merge* nodes or $loop_o$ nodes associated with them, recursively. Figure 4 shows an outline of tracing of a graph. The scheduling algorithm is the following:

Algorithm A – code scheduling

step A1 All nodes in a graph are initialized to NOT SELECTED. A set of selectable nodes S is initialized as follows.

S = {n|n is a node which has no parent nodes except for join nodes.}

step A2 The following steps are repeated until S becomes the empty set.

step A2.1 A set of *join* nodes J is initialized to the empty set.

step A2.2 Algorithm B is applied to S.

step A2.3 If more than one *call* node is scheduled in step A2.2, all of them except the last *call* node are changed to *fork* nodes. The *fork* operation implies creation and invocation of a child process, which executes a function application concurrently with the current process, while the *call* operation implies that the current process executes a function application.

step A2.4 If one or more than one *call* node is scheduled in step A2.2, S is set as follows:

S = {n|n is a child node of a join node, which is a child node of the last *call* node.}

If any *call* nodes are not scheduled in step A2.2, S is set as follows:

S = J

Algorithm B – graph tracing

The following steps are applied to each element n of S repeatedly, until S becomes the empty set.

step B1 If n is an *sw* node, n is set to SELECTED and then algorithm A is applied to both the **then** part graph and the **else** part graph. The **then** part graph is the graph which is originated from node n's child nodes indicated with *t* arcs and destined to a *merge* node associated with node n. The **else** part graph is the graph which is originated from node n's child nodes indicated with *f* arcs and destined to the *merge* node. After scheduling the **then** part graph and the **else** part graph, the *merge* node is added to S.

step B2 Else if n is a $loop_i$ node, n is set to SELECTED and then algorithm A is applied to the body graph, which is originated from node n's child nodes and destined to a $loop_o$ node associated with n. After scheduling the body graph, the $loop_o$ node is added to S.

step B3 Else if n is a *join* node, n is added to J.

step B4 Else n becomes SELECTED. n's child nodes of which parent nodes have been SELECTED already, are added to S.

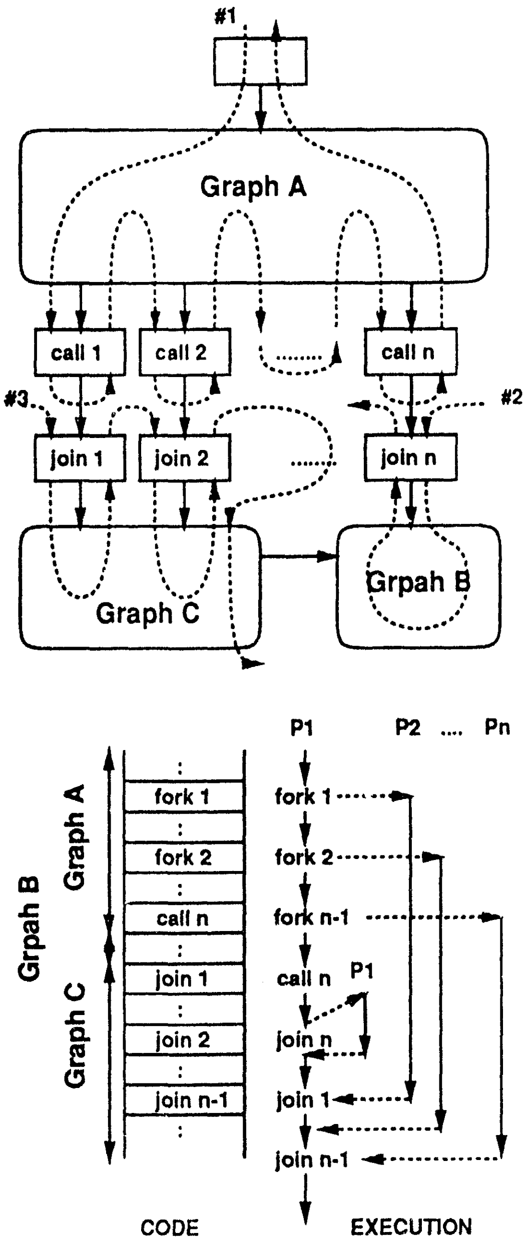


Figure 4: Code scheduling

3 Implementation

Since free variables are not allowed in *Valid* in principle, applying of a function requires only the entry address of the function code, values of arguments and return addresses for return values. We have implemented a multi-task monitor on DYNIX. The multi-task monitor has a light weight fork-join mechanism, and we evaluated the performance of object code generated by our compiler on this multi-task monitor.

Figure 5 shows the outline of the multi-task monitor. Frames are data structures in a shared memory, and consist of a header and work area. A code entry address to start execution, a pointer to a frame associated with the caller function and a pointer to a barrier variable are stored in the header of a frame associated with a called function. Arguments and local variables are stored in a work area. Frames are created by *fork* operations or *call* operations, and are kept available until completion of their function. To reduce the overhead of creation and releasing of a frame, each processor has own free-list of frames, which are the only ones it is allowed to access. A task pool is a list of frames

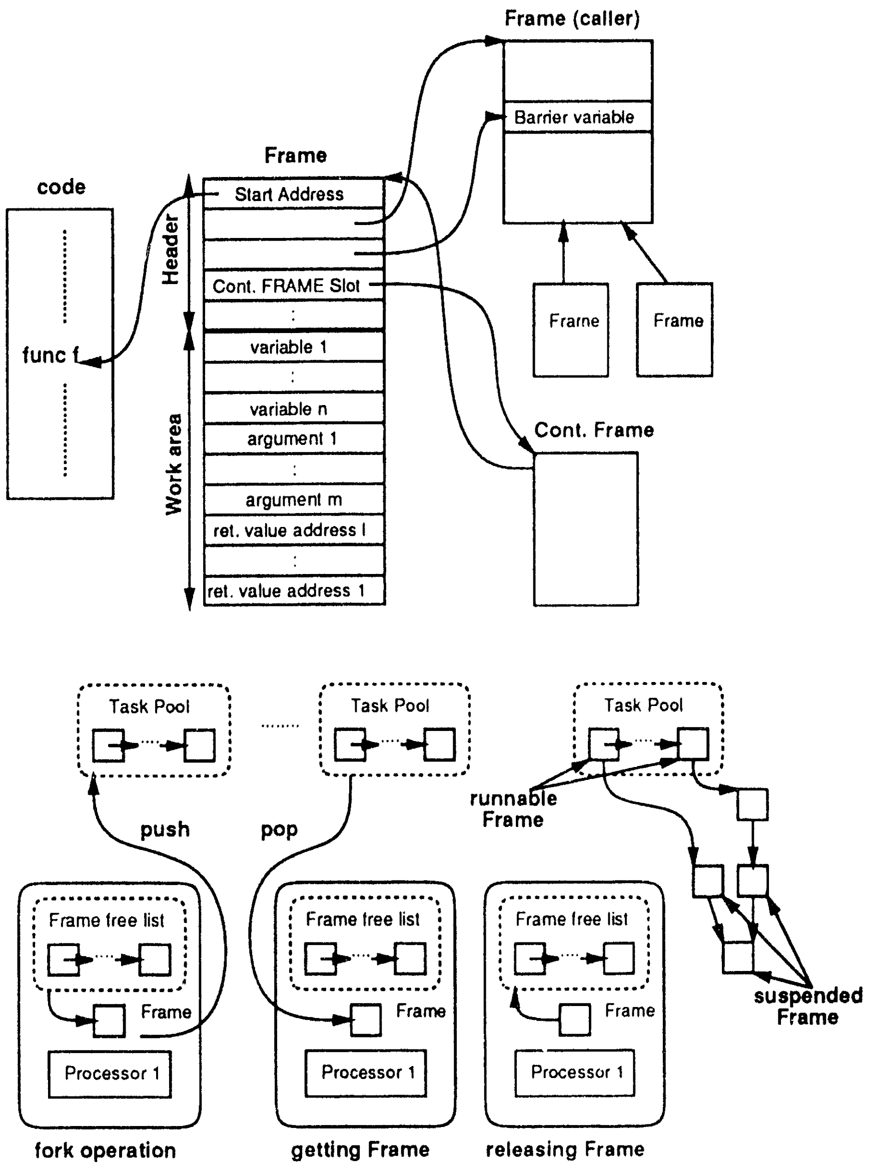


Figure 5: Multi-task monitor

ready to run. This list is implemented as a LIFO. Processors get a runnable frame from a task pool in a mutually exclusive way. The number of task pools is set to more than the number of processors to reduce the overhead caused by the access competition to task pools.

Processors get an runnable frame from a task pool and execute the frame repeatedly until runnable frames are exhausted, which means the program has completed. By scanning task pools, when processors succeed in locking a task pool having one or more frames, processors get the frame from it. Since processors execute a function with their local stack and registers, access to frames in shared memory is only access to arguments, local variables and barrier variables, so that the cache mechanism is exploited effectively. This implies that it is possible to avoid bus saturation which is a main cause of bottlenecks in shared memory multiprocessors. In a *fork* operation, the barrier variable, which is a local variable of the caller function and is initialized to 1, is incremented. Then, a new frame is gotten from the current processor's frame free-list, a called function code entry point and a pointer to the frame of a caller function, etc, are set to those of its header, and arguments are copied into its work area. Lastly, the frame is put into a task pool, which is selected in the same way that a runnable frame is obtained. A *call* operation is the same as a *fork* operation except that a new frame is put into the continuation frame slot of the current frame. In a *join* operation, the barrier variable is decremented. If the barrier variable becomes 0, the next operation is executed. Otherwise, the processor abandons the current frame and starts to execute the frame which is stored in the continuation frame slot of the current frame in a *call* operation. A *call* operation and switching context from the current frame to the continuation frame cost less than a *fork* operation and getting a runnable frame, since they do not require mutually exclusive access to task pools. When the current function is completed, the barrier variable in the frame of the caller function is decremented and the processor stores the current frame in its own frame free-list. If the barrier variable becomes 0, the processor resumes the execution of the frame of the caller function, otherwise the processor picks up a runnable frame from a task pool again.

4 Performance

We have evaluated the performance of compiled code of *Valid* programs on a Sequent S2000, using from 1 to 16 processors. We report results of comparisons of *Valid* programs with SISAL programs and C programs in elapsed time, and evaluations of the speedup of *Valid* programs. The SISAL programs and C programs are written using the same algorithms as the *Valid* programs. The SISAL programs are compiled with the optimizing SISAL compiler, OSC version 12.9, using default optimization mode (no options), and run on 16 processors. The C programs are sequential and compiled with the Sequent C compiler. The C programs are not optimized. Table 1 shows elapsed time in seconds for the *Valid*, SISAL and C programs, the speedup and the overhead caused by parallel control for *Valid* programs. The elapsed times of the *Valid* programs and C programs are measured with a *Sequent microsecond clock*, which is 32-bit up-counter updated every 1 micro second [9]. The elapsed times of the SISAL programs are measured with *speedups*, which is the SISAL parallel speedups data gatherer. In the C column and SISAL column, figures in brackets are the relative speed evaluated as follows:

$$\text{relative speed} = \frac{\text{time of C/SISAL program}}{\text{time of } \textit{Valid} \text{ program}}$$

The Overhead column shows the proportion of system time to total time based on results from the DYNIX profiler. System time includes time of *fork/call* operations, *join* operations and getting and releasing frames. Figure 6 shows the speedup of *Valid* programs relative to processor number for each benchmark program. In the graph, the horizontal axis shows the number of processors used, the vertical axis shows the speedup, and the linear proportion line shows the ideal speedup.

The program 'sum(l, h)' calculates the summation from l to h integers. In SISAL, this program is implemented with a product-form loop and a reduction operation. In *Valid*, this program is implemented with a parallel expression and a reduction operation. Table 1 shows that the performance of the *Valid* program is comparable to the performance of the SISAL program. This program is partitioned into relatively large portions and distributed to processors equally, so process creation overhead is less critical. This program also shows a very nice speedup in Figure 6.

The program 'matrix(n)' computes the product of two $n \times n$ matrices. Although the speedup is close to linear, the speed is about one-third that of the SISAL program. This is probably as caused by the difference in implementation of array structure. An array in SISAL (and C) is a one-dimensional or multidimensional collection of homogeneous values. On the other

Table 1: Time comparisons of *Valid*, SISAL and C, and speedups and overheads of *Valid*.

Program	Time (sec.)					Speedup	Overhead
	Valid 16cpus	C 1cpu		SISAL 16cpus			
sum(1, 10^6)	0.0241	0.241	(10.0)	0.0200	(0.830)	15.0	7.69%
matrix(128)	0.987	4.49	(4.55)	0.323	(0.327)	12.8	37.5%
matrix(256)	9.07	38.4	(4.23)	3.11	(0.343)	13.3	23.4%
nqueen(10)	4.21	25.3	(6.01)	65.2	(15.3)	7.48	15.5%
qsort(10^4)	3.21	12.1	(3.77)	8.09	(2.52)	5.07	65.8%

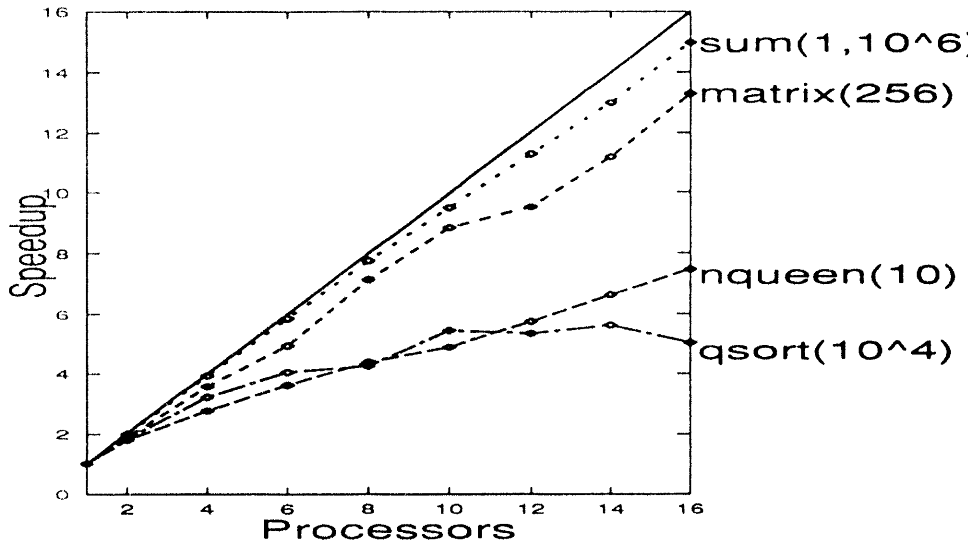


Figure 6: Speedup graphs for four benchmark *Valid* programs

hand, an array in *Valid* is one-dimensional and may have a collection of heterogeneous values. Therefore, the implementation of an array in Valid must be more complicated than the implementation of an array in SISAL (and C). We consider that it is possible to increase the speed of the *Valid* programs up to that of the SISAL ones by introducing the same array specification as SISAL into *Valid*.

The program 'nqueen(n)' searches all solutions of the n-Queen puzzle. The C program uses library functions from *Valid* to manipulate lists. The SISAL program is implemented with streams and is not parallelized. In the SISAL program, the size of a stream, which contains the solutions, is determined after the search of all solutions is completed, so that it is not invariant. OSC does not concurrentize loop forms which produce variant size streams (and arrays). On the other hand, the *Valid* program is parallelized. This is because our implementation involves the function application level parallelism, not loop levels, as we have mentioned before.

The program 'qsort(n)' rearranges the elements of a list, which has n integers, in order of size with a quick sort algorithm. The C program uses library functions from *Valid* to manipulate lists. The SISAL program is implemented with arrays. The SISAL program is not parallelized, because of the concurrentization strategy of OSC mentioned above. In Figure 6, the speedup curve of Valid saturates at about 5.5 after 10 processors. The reason is that list generations and function applications cannot overlap each other in this program, because non-strict evaluation has not been implemented yet.

In function application level parallelism, it is possible to extract more parallelism from programs than with iteration level parallelism. When there are low-cost recursive functions, such as the Fibonacci function, process creation and switching occur frequently. In a shared memory machine, frequent process creation and switching cause bus saturation. As a result, performance of the system degrades. Table 2 and Figure 7 show effect of the cost of the function sum' on performance.

The program 'sum'(l, h, i)' calculates the summa-

Table 2: Effect of the cost of the function **sum'** on performance.

Program	Time (sec.)	Speedup	Overhead
sum'(1,10^6,1)	6.75	6.23	63.9%
sum'(1,10^6,10)	0.935	6.85	61.9%
sum'(1,10^6,10^2)	0.155	9.79	44.6%
sum'(1,10^6,10^3)	0.0648	14.0	11.5%

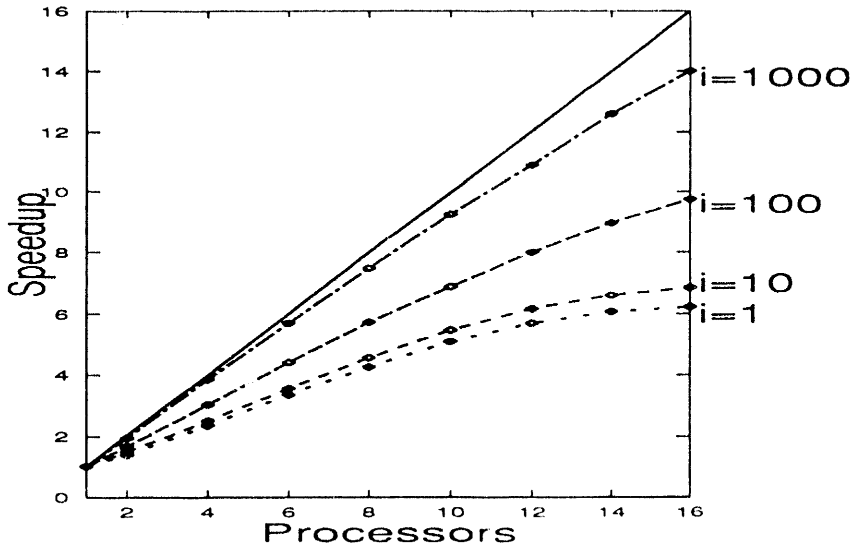


Figure 7: Speedup graph for sum(1,10^6, i)

tion from l to h integers.

```
function sum'(l,h,i:integer)
  return(integer)
  = if h-l < i then
      for (s,j:integer) init(0,l) body
        if j > h then return s
        else recur(s+j, j+1)
      else {let m = (h - l) / 2
            in sum'(l,m,i)+sum'(m+1,h,i)};
```

This program switches algorithms from a divide and concur version to a loop version, according to the third parameter i. If the range from l to h is equal to or larger than i, the divide and concur version is used. If the range size is smaller than i, the loop version is used. When $i = 1$, this program is thoroughly parallel, and when $i > h - l$, this program is thoroughly sequential. Thus, the grain size of parallelism can be controlled with i. Using this program, we evaluated the effect of grain size and the number of processes on the efficiency of our system. Note that this program is not meaningful for SISAL, because only product-form loops are parallelized in SISAL. When $i \leq 100$, the overhead caused by control for parallelism is high. The explanation of the speedup data is that, in running the program, almost all processors access task pools to write for *fork* operations and to read for getting a runnable frame all together, and, due to bus saturation, accesses to task pools slow down. As for speeds, the low cost of function body and the large number function applications lead to amplification of the differences in cost between *fork* operations and function body.

To solve the problem of *fork* operation overhead, the compiler estimates the cost of each function, and generates code in which light weight function applications are not forked or are inline expanded. However, the exact cost of recursive functions can't be estimated at compile time. In the current specification of *Valid*, improvement of the algorithm is the way to solve the fork overhead problem, as mentioned above in connection with the sum' program. This solution destroys the portability of programs. Functional programming languages have the merit that programmers can write programs without attention to the parallelism of the program, while they have the demerit that program optimization is difficult. For example, even if the most effective strategy, that is, determin-

Table 3: Effect of introducing annotation to control fork operation.

Program	Time (sec.)	Speedup	Overhead
fibo(30)	7.61	6.24	73.5%
fib8(30)	2.84	15.4	70.6%

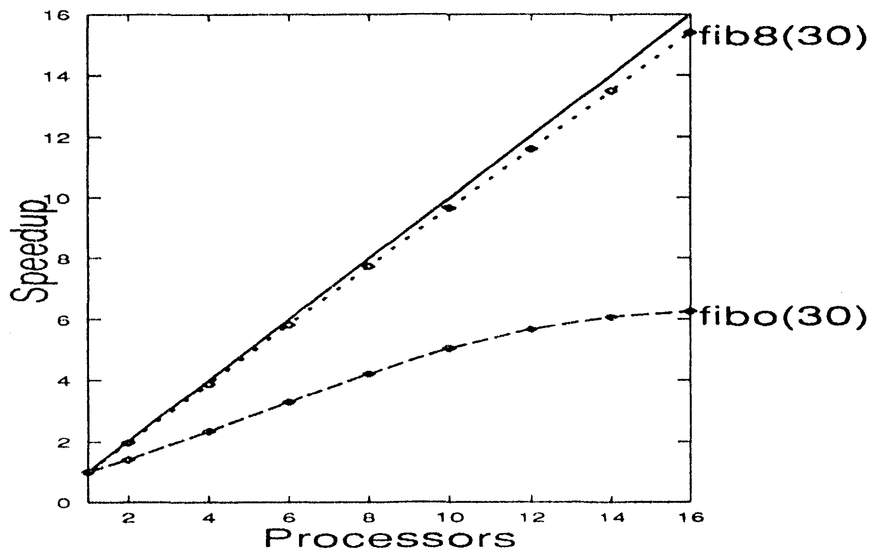


Figure 8: Speedup graph for fibo(30) and fib8(30)

ing whether to evaluate in parallel or sequentially, and the most effective mapping of functions and data to processors in multicomputers are obvious to programmers, it is difficult to express them in programs. As a paradigm to solve the above problem without losing the merit of functional programming languages, parafunctional programming, such as ParAlfl from Yale university, has been offered [10, 11]. This is a method to extend a functional programming language by introducing metalinguistic devices such as annotations in the language. Since ways of processing and mapping functions and data structure to processors are individual for the semantics of programs, it is expected that the problem mentioned in connection with sum' can be solved by extending *Valid* with the parafunctional method. We have attempted to extend *Valid* to optimize the strategy of processing functions, and evaluated its performance.

An extension of function application specification in *Valid* allows the programmer to express a strategy for deciding between sequential evaluation or parallel evaluation. The extension is as follows.

```
function_application
  ::= function_name(argument_list)
        $[boolean_expression]
```

The above specification means that if the boolean expression is true, the function is evaluated with *fork*, if the boolean expression is false, the function is evaluated with *call*. In Table 3 and Figure 8, the program 'fib8(n)' is an extended version of 'fibo(n)', in which *fork* is controlled with the argument n. The *Valid* program 'fib8(n)' is as follows.

```
function fib8(n:integer)return(integer)
  = if n<2 then 1
    else fib8(n-1)$[n>=8]
         + fib8(n-2)$[False];
```

In the above program, if n $\geq$ 8, the function application fib8(n-1) is evaluated with a *fork* operation in parallel, otherwise it is evaluated sequentially with *call* operation. Figure 8 shows that speedups are improved substantially. The improvement in speedups implies bus saturation relaxation. This is caused by decreasing the number of fork operations, which lowers the effect of cache. Overheads are improved a little. The reason is that the cost of setting up frames is still high.

To solve this problem, when a function evaluation is sequential, it is done by using a mechanism similar to C on a stack which each processor has locally, or which is in the work area of the expanded frame which has a stack. In the former case, the multi-task monitor has to be changed, because data required for a function evaluation are all fixed on the stack so that frames are not necessary. We have implemented and evaluated this version [12]. In this version, the *Valid* program 'fib8(n)' achieved about 5 times the speeds of C. In this version, however, lenient evaluation causes deadlock. In the latter case, memory efficiency is lowered, because evaluation of recursive functions require sufficiently large frames. We expect that the memory efficiency problem can be solved by a method which determines whether to the use a non-expanded frame or an expanded frame according to the type of evaluation (*fork* or *call*).

The *fork* control method mentioned above can be expanded easily when mapping functions and data to processors when the target machine is a multicomputer. By allowing an integer expression to be an expression in $[ ] and generating code, which regard the value of the expression in $[ ] as a processor ID and map the function application to a processor, the mechanism of mapped expression in ParAlfl can be implemented.

5 Related work

Several methods for implementing a functional programming language on a commercially available parallel machine have been proposed. Among them are the $\langle\nu, G\rangle$-machine proposed by Thomas Johnsson [4] and the Process-Oriented Dataflow System (PODS) proposed by Lubomir Bic [6].

The $\langle\nu, G\rangle$-machine is a graph reduction machine, which evaluates super combinators efficiently. Programs in Lazy ML are translated into a set of definitions of super combinators. The definitions of super combinators are then compiled to the object code of the $\langle\nu, G\rangle$-machine, which reflects the behavior of the graph reduction machine exactly. In the $\langle\nu, G\rangle$-machine, the processing order of instructions is determined dynamically at runtime. So all the data required for evaluating a program are in the shared memory as fragments of the program, which form a program graph dynamically. Frequent accesses to the shared memory cause bus saturation, so that the efficiency is lowered. Our implementation has the same problem. But in our implementation, it is possible to reduce access to the shared memory by using as many stacks and registers as possible, since access to frames in the shared memory is only to arguments and local variables of a function.

Therefore from the viewpoint of the effect of cache, our implementation has an advantage over the $\langle\nu, G\rangle$-machine. However, since in our implementation the processing order of instructions is determined statically at compile time except for function applications, it is difficult to implement lazy evaluation and higher-order functions, especially functions which are constructed efficiently at runtime. In our implementation, in order to implement such dynamically constructed functions, a special mechanism such as an interpreter is required.

PODS is based on a thread level dataflow model and assumes as the target machine a multicomputer, which has special mechanisms for parallel evaluation such as Matching Unit, Memory Manager etc. Dataflow graphs are constructed from Id programs. Sequentially processed blocks, called sequential code segments (SCSs), are extracted from the dataflow graph according to the arcs of the graph. The object code of PODS is generated by compiling the SCSs. When the cost of an SCS is smaller than the cost of control for parallel processing, it can be made larger by combining other SCSs according to data dependencies. When adopting PODS on a shared memory multiprocessor, a function becomes an SCS, because the cost of control for parallel processing is high in such a machine. Combination of SCSs corresponds to the code scheduling in our implementation. PODS code may cause suspension even when executable instructions exist, because code scheduling in PODS does not regard function application instructions as special instructions. Therefore, considering the above, when a target machine is a shared memory multiprocessor, our implementation has an advantage over the PODS.

6 Conclusions

In this paper, we presented a compiling method to translate the functional programming language *Valid* into object code executable on a Sequent Symmetry S2000. This method implements function application level parallelism by dataflow analysis of functional programs. We have evaluated the performance for speed and speedup of the compiled *Valid* code on a Symmetry. This evaluation made clear that the frequency of *fork* operations causes bus saturation, and the efficiency is lowered, and low-cost functions and data structure become bottlenecks. To solve the bus saturation problem, we attempted to add parafunctional features to *Valid* and evaluated the performance of extended *Valid* programs, showing that this problem can be solved.

The next step in this work is to implement a stream parallel processing mechanism and to adapt the method mentioned in this paper to multicomputers such as the FUJITSU AP-1000.

References

[1] M. Amamiya and R. Taniguchi, "Datarol:A Massively Parallel Architecture for Functional Languages", In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pp.726-735, (1990).

[2] Simon L.Peyton Jones, *The Implementation of Functional Programming Languages*, PRENTICE-HALL INTERNATIONAL (1987).

[3] Thomas Johnsson, *Compiling Lazy Functional Language*, Chalmers University of Technology DEPARTMENT OF COMPUTER SCIENCES (1987).

[4] Lennart Augustsson and Thomas Johnsson, "Parallel Graph Reduction with the $\langle\nu, G\rangle$-machine", ACM Proc.4th International Conference on Functional Programming Languages and Computer Architecture, 202 (1988).

[5] Paul Hudak, "Distributed Execution of Functional Programs Using Serial Combinators", IEEE TRANSACTIONS ON COMPUTERS, Vol.C-34, No.10, 881 (1985).

[6] Lubomir Bic, "A Process-Oriented Model for Efficient Execution of Dataflow Programs", JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 8, 42 (1990).

[7] J.McGraw et al., *SISAL: Streams and Iteration in a Single-Assignment Language: Reference Manual*, Ver.1.2, Manual M-146, Rev.1, Lawrence Livermore National Laboratory, Livermore, Calif.,(1985).

[8] R. Hasegawa and M. Amamiya, "The Design and Implementation of A High Level Functional Language for Data Flow Machines", IEICE Trans.Inf.& Syst., Vol.J71-D, No.8, pp.1532-1539, (1988).

[9] Sequent Computer Systems,Inc., *Symmetry System Summary*.

[10] Paul Hudak, "Para-Functional Programming", IEEE Computer 19.8,60 (1986).

[11] Paul Hudak, "Exploring Parafunctional Programming: Separating the What from the How", IEEE Software, Vol.5, No.1, pp.54-61, (1988).

[12] E. Takahashi, R. Taniguchi and M. Amamiya, "Compiling Technique based on Dataflow Analysis for Functional Programming Language Valid", TECHNICAL REPORT OF IEICE, COMP 92-96, SS92-43 (1993-03).

Copy Elimination for True Multidimensional Arrays in SISAL 2.0

Steven M. Fitzgerald
Dept. of Computer Science
University of Massachusetts Lowell
Lowell, MA 01854
sfitzger@cs.uml.edu

Abstract

Applicative languages have been proposed for defining algorithms for parallel architectures because they are implicitly parallel and lack side effects. However, a straightforward implementation can induce excessive copying which can limit performance. To address execution efficiency, optimization techniques, such as build-in-place [Ran87] and update-in-place [Can89], have been developed. These optimizations remove unnecessary array copy operations through compile-time analysis. Additionally, update-in-place eliminates unnecessary reference counting, reducing parallel bottlenecks that can occur at run-time [OC88].

Both build-in-place and update-in-place are based on hierarchical ragged arrays, i.e., the vector-of-vectors array model. Although this array model is convenient for certain applications, many optimizations are precluded, e.g., vectorization. In the design of SISAL 2.0, two array models have been included: the vector-of-vectors model and the flat model. In this paper we discuss the changes to reference inheritance, *which is part of update-in-place analysis. These changes are necessary for arrays that are stored in contiguous memory, i.e., under the flat model.*

1 Introduction

SISAL is an applicative programming language designed to facilitate the development of applications that can run efficiently on a wide range of architectural platforms [MSA+85, BOCF92] To achieve efficient execution under the applicative model of computation, sophisticated optimizations are necessary to prevent the copying of large arrays. The SISAL 1.2 compiler depends upon both *build-in-place* [Ran87] and *update-in-place* [Can89] analysis to remove unnecessary copy operations. Build-in-place attacks the incremental construction problem, while update-in-place attacks the array update problem. As a result, array-intensive applications written in SISAL 1.2 execute as fast as their FORTRAN equivalents [Can89, Feo90, Can92].

The design of SISAL 1.2 arrays is based on the *vector-of-vectors* array model. Under this model, multidimensional arrays are built hierarchically from one-dimensional arrays, i.e., from vectors [Gao90]. Although hierarchical arrays are convenient for many applications, it is expensive to manipulate array values represented in this model [Feo90]. Since the additional overhead is unnecessary for applications that do not utilize the flexibility of the vector-of-vectors model, SISAL 2.0 provides a second array model that is based on the *flat* array model. Under this model, multidimensional arrays are built by the concatenation of the subarrays of the innermost dimension to form a single one-dimensional array [Gao90]. The array's uniform structure allows many more optimizations to be performed, such as vectorization; however arrays must be stored in contiguous memory.

It is claimed that SISAL 2.0 applications that utilize the flat array model can achieve better performance than their FORTRAN counterparts [Feo90]. To realize this goal, the current optimizations that are based on the vector-of-vectors model must be both extended and generalized to operate on flat arrays. In this paper, we examine one of the optimizations that is part of update-in-place [Can89], *reference inheritance*, and show how it can be generalized to support both array models. We also introduce a new subphase of reference inheritance that significantly reduces the array copying imposed by the contiguity requirement under the flat model. This subphase can also be beneficial for vector arrays because it can reduces pointer copying.

2 Array Models Used in SISAL 2.0

The language design for SISAL 2.0 incorporates two models for arrays: the vector-of-vectors model and the flat model. An array's data type specifies both the operations that can be performed on the array and the array model used to represent the array. Arrays that follow the flat model must be declared explicitly by specifying both the number of dimensions and the base type. Additionally, the size of each dimension must be specified so that the necessary memory is allocated during array definition; however the size information is not part of the data type. Arrays that follow the vector-of-vectors model need not be declared, i.e., their type can be determined by context. Memory is allocated for vector arrays as required.

For example, consider the following type declarations:

```
type OneVector = array of double;
type TwoVector = array of array of double;
type OneDim = array [..]  of double;
type TwoDim = array [..,..]  of double;
```

Both the `OneVector` and the `TwoVector` declarations specify a one-dimensional array type that follows the vector-of-vectors model. An array of type `OneVector` is a one-dimensional array comprised of doubles, whereas an array of type `TwoVector` is a one-dimensional array comprised of arrays of doubles. Both of the `OneDim` and `TwoDim` declarations specify array types that follow the flat array model. The ".." in the declarations above are placeholders that indicate the number of dimensions. When an array is defined, the size of each dimension must be given. Notice that the types, `OneVector` and `OneDim`, are equivalent, i.e., their physical representation are identical.

2.1 Vector-of-vectors Array Model

Strictly speaking, all arrays under the vector-of-vectors model have a single dimension. Conceptually, multidimensional arrays are constructed hierarchically from other one-dimensional arrays. For example, a two-dimensional array of integers is logically equivalent to a one-dimensional array whose elements are one-dimensional arrays of integers [Gao90]. Since the size of an individual array is not part of its type, each subarray in a multidimensional array may have a different length. Additionally, the bounds of each subarray may not match. This allows the formation of ragged arrays, where the range of indices for each subarray may differ.

In SISAL 1.2, an n-dimensional array is physically represented by a series of $(n-1)$-dimensional arrays. As depicted in Figure 1b, these pointer arrays form a tree structure with the leaves containing the innermost dimension of the array. Under this representation, subarrays that have different or continually changing sizes can be implemented efficiently. When either an element or a dimension is added to an array only the affected dimension is modified. Additionally, individual components of an array may be shared. Other implementations are possible but are less efficient. For example, if the array is stored in contiguous memory, the entire array must be copied to a new location whenever an new element is added.

The vector-of-vectors model is convenient for expressing algorithms that operate on a dimension-by-dimension basis, e.g., row-ordered, but not for algorithms that operate on a region-by-region basis. Read operations are expensive since the entire array structure must be traversed linearly for each element accessed. Additionally, memory overhead is large because physical storage must be both allocated and deallocated for each subarray individually. Furthermore, maintaining and referencing an array descriptor, i.e., a dope vector (cf. [Old92]), to determine each subarray's bounds and location degrades performance.

2.2 Flat Array Model

Under the flat model, all arrays are monolithic. Multidimensional arrays are constructed by the concatenation of one-dimensional arrays to form a single flattened array [Gao90]. For example, FORTRAN specifies that multidimensional arrays are represented by the concatenation of the innermost dimensions, i.e., column-major order [Mac83]. Since the physical size of the array must be known prior to its construction, the bounds of each dimension must be specified. Although this prevents the dynamic growth of individual subarrays, the overhead for allocating and deallocating multidimensional arrays is decreased.

The flat model is efficient for algorithms that operate on an element-by-element basis. Accessing an array element can be performed in constant time since the address calculation is based on the size and bounds of each dimension. This address can be partially evaluated at compile-time since each element's location within the array is known *a priori*. Additionally, optimizations, such as vectorization, can be performed because the array is stored contiguously. Typically, for these optimizations to be performed, the array must be accessed in the same order in which it is stored, i.e., as specified by the language definition.

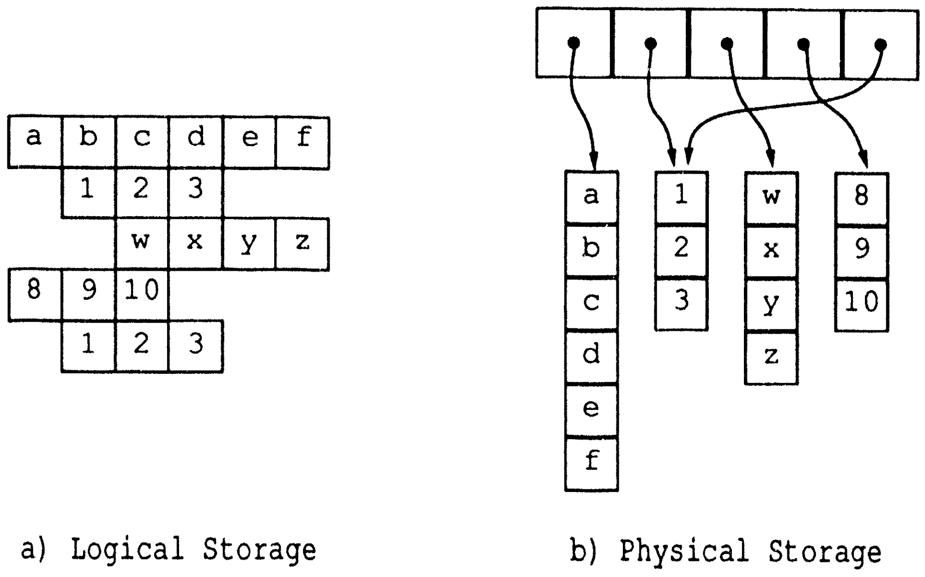


Figure 1: Logical and physical storage of a two-dimensional vector array

In a language such as Ada [GH83], where the bounds of an array are not part of its type, a dope vector is used to perform address calculations at runtime. Although this increases array overhead, additional flexibility is gained. By changing only an array's bounds, a subsequence of a one-dimensional array can be selected. This subsequence can be operated on as if it is a distinct one-dimensional array. Newer languages, such as FORTRAN 90 and SISAL 2.0, have extended this functionality to allow arbitrary subcomponents of a multidimensional array to be accessed, e.g., slices [Seb93].

In SISAL 2.0, a multidimensional array can be divided along any dimension or combination of dimensions. Additionally, the order in which elements appear within an array can be specified. This allows elements that are uniformly distributed within an array to be selected. For example, the SISAL 2.0 expression

"A[i in 4..1..-1, j in 4..1..-1 {i dot j}]"

selects the elements along the major diagonal of the 4×4 array **A** in reverse order, as depicted in Figure 2. Operations that select slices of an array can be performed by simply modifying an array's dope vector [Old92]; a copy operation is not required. Additionally, modifying a multidimensional array's dope vector can change the array's logical layout without changing its physical layout. Program analysis could be used to determine the most efficient layout for an array. Furthermore, the dope vector can give the illusion that the array is stored in contiguous memory. However a consistent spacing is maintained between array elements, thus allowing optimizations, such as vectorization, to be applied. We propose the term "dimensional" to refer to any array model that allows an arbitrary region of an array to be both selected and operated on as it is a true multidimensional array.

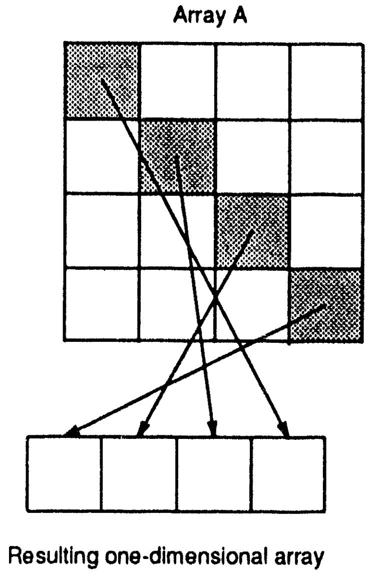


Figure 2: Array mapping for the SISAL 2.0 expression "A[i in 4..1..-1, j in 4..1..-1 {i dot j}]"

3 Copy Elimination and Reference Counting

An implementation that strictly adheres to the applicative model must copy data values that are updated. This copying ensures that other operations, which access the original data value, are not affected. For large data aggregates, such as arrays, the cost of copying is prohibitive. Array accesses can be ordered so that array read operations occur before any array write operations. The final write operation can update the array directly, preventing a copy operation, since all other accesses to the array have been performed. However, sequentializing array accesses decreases the amount of parallelism exploited.

Reference counting can be used to decrease the amount of array copying. A reference counter, which is adjusted at run-time, is used to record the number of potential users of an array. For each array operation, the reference count is modified. When a reference count is one, the associated array can be updated directly, i.e., in place. Although reference counting may reduce copying, it both increases run-time overhead and produces parallel bottlenecks [OC88, Feo90]. Additionally, operations that access multidimensional arrays can overestimate reference counts, preventing in-place operation[Can89]. Fortunately, program analysis can be used to reduce both copy operations and reference-counting operations [CO88, Hud87, Hud86, SS88].

Within the SISAL compilation environment, two intermediate forms, IF1 [SG85] and IF2 [WSYR86], are used to model program execution. Initially, an IF1 graph is produced from SISAL source code. This graph is then altered by a series of optimizations, producing an IF2 graph.[1] These optimization alter the operational semantics of the graph to allow for more efficient computation.

Consider the IF1 graph for the SISAL expression "A[i:0], A[j]" (see Figure 3a). This expression is comprised of an array-update operation, (A[i:0]), and an array-read operation, (A[j]). In the IF1 graph, the AReplace node performs the array-update operation. The AReplace node must first copy the array **A** to ensure the array-read operation is not affected (since the execution order is not defined). The i^{th} element of the new array is then updated with the value 0. To prevent the copy operation from always occurring, the IF1 graph is transformed into an IF2 graph that contains a NoOp node and reference count pragmas. The resulting graph is depicted in Figure 3b.

NoOp nodes are inserted into a graph to perform the copy operations associated with array modifiers, thus limiting the copy logic of a graph to single node type. Additionally, edges are annotated with marks to indicate the type of copying that is performed. These marks include **r**, **R**, and **O**; we summarize the meaning of these marks in Table 1. Additionally, some operations need to modify only an array's dope vector, e.g., setting the lower bound of an array. A **P** mark is used to indicate that the copying is performed on the dope vector and not on the array's physical data.

Type of Copying	Dope Vector	Array Data
None	**PR[O]**	**RO**
Conditional	**Pr[O]**	**r[O]**
Unconditional	**P[O]**	**[O]**

Table 1: NoOp node semantics based on input edge marks (Marks enclosed in brackets are optional.)

The incoming edge of the NoOp node in Figure 3b, is decorated with an **r** mark. The mark indicates that the NoOp node performs a conditional copy operation. At run-time, the reference count of the array **A** is examined; if the associated value is one then the array is not copied, but the array is passed directly, via its dope vector, to the AReplace node. The AReplace node performs a destructive update to the array presented to it (as indicated by the **RO** marks).

Reference count pragmas specify how an array's reference count is modified; these pragmas include: **sr**, **pm**, and **cm**. The **sr** pragmas are used to initialize reference counts. The **pm** and **cm** pragmas are used to increment and to decrement reference counts, respectively. Each node in an IF2 graph modifies the reference count of both imported and exported arrays based on the values associated with these pragmas. To ensure correct operation, the reference count of an exported array is adjusted prior to its export, and the reference count of an imported array is adjusted after the results are computed.

Consider the graph in Figure 3b. Initially, the reference count for array **A** is two. If the AElement node executes first then a copy operation is avoided. The execution sequence of the graph is as follows.

1. The AElement node executes and then decrements the reference count of array **A**, as indicated by the **cm=-1** pragma.

[1]For a more information on these optimizations refer to [Ran87] and [Can89]

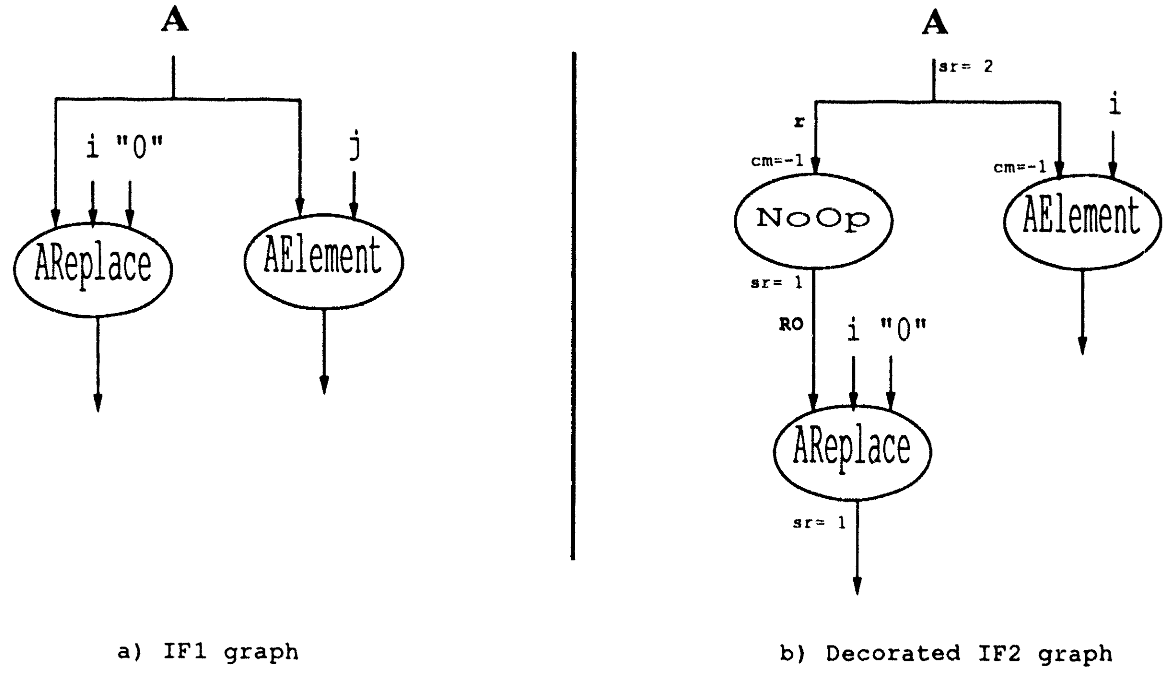


Figure 3: IF1 and Decorated IF2 graph for the SISAL expression "A[i:0], A[j]"

2. The NoOp node performs a conditional copy of the array. The value of the reference count is examined and since it one the NoOp does not perform a copy operation. Instead the dope vector for array **A** is passed to the AREPLACE node.
3. The AREPLACE performs a destructive update to array **A**.

However, if the NoOp node is executed first, a copy operation is performed.

Initially, reference count pragmas are associated with each graph edge that transmits an aggregate. Reference counts must be modified within critical regions of code because they are a shared resource and are modified independently. Since this can create parallel bottlenecks, a series of optimizations, known collectively as *update-in-place* [Can89], is applied. The optimizations are used both to reduce reference counting and to eliminate expensive copy operations.

These optimization are based on the vector-of-vectors array model. To achieve the desired performance in SISAL 2.0, copy elimination must be performed on both vector arrays and flat arrays. Additionally, new IF2 nodes, which manipulate dope vectors, may be required to prevent expensive copy operations. For example, an operation that reverses the specification of an array's dimensions could be incorporated into IF2 to allow the transpose function to be implemented without copying. As a starting point for this research, we examine the modifications required by reference inheritance to operate on arrays that are stored in contiguous memory, i.e., under the flat model. In this paper, we restrict ourselves to SISAL 1.2 syntax so that we may concentrate on the problem at hand. Additionally, this approach allows optimization to be implemented based on the current definition of IF2.

3.1 Update-in-place analysis

Update-in-place analysis is comprised of the following five optimizations [Can89]: reference inheritance, node reordering, reference count elimination, mark assignment, and ownership analysis. These optimizations are based on the vector-of-vectors model for array representation. If multidimensional arrays are stored in contiguous memory, these optimizations must be adjusted to ensure proper results. Additionally, the application and insertion of reference count pragmas must be adjusted. Under the vector-of-vector model, each subarray has its own reference count, whereas a multidimensional array under the flat model has a single reference count for the entire array.

For example, consider the IF2 graph for the SISAL expression "A[i,j:0]", where the two dimensional array **A** must be stored in contiguous memory. This graph,

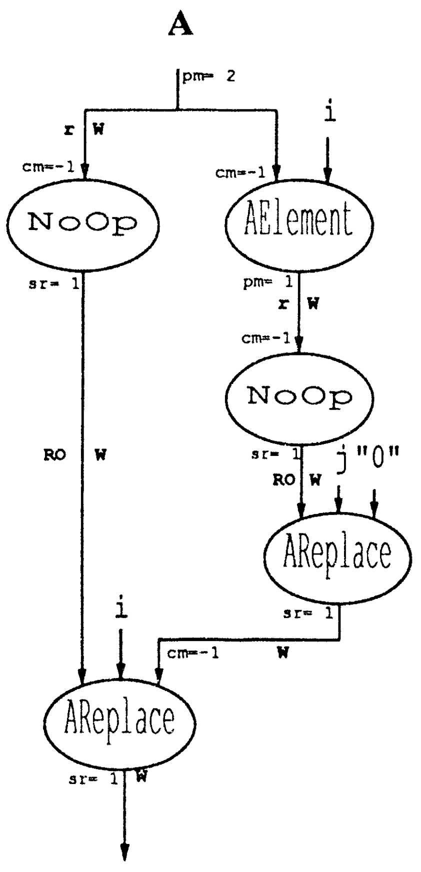


Figure 4: Decorated IF2 graph of the SISAL expression "A[i,j:0]"

depicted in Figure 4, has been augmented with NoOp nodes and reference count pragmas based on this requirement. The array update is performed by two AReplace nodes, one for each dimension. The ith row is first selected by the AElement node and then modified by the rightmost AReplace node. The bottommost AReplace node copies the modified subarray into the correct location within the array **A**. This extra copy operation is required to ensure that the array **A** remains contiguous. Additionally, the original modified subarray's reference count is decremented. Under the vector-of-vectors model the modified subarray is not copied, but the ith element of the array, which is a pointer, is updated; as such, its reference count is not modified.

In the graph's current form, up to three copy operations can occur, one for each NoOp node and one by the bottommost AReplace node. However, all three copy operations are unnecessary. The graph overestimates the reference counting needed to perform the single element update since each update operation contributes to the array's reference count. Fortunately, the graph can be transformed to allow the extra reference counting information to be removed, as described in Section 3.2.

3.2 Reference inheritance transformation

In this section we introduce a generalized version of reference inheritance that works for both the vector-of-vectors and flat array models. We also introduce an additional phase to the optimization that identifies when subarrays are updated-in-place. Based upon this identification, the copy operation, which is performed by a AReplace node, to ensure the array is stored contiguously is eliminated. This additional phase is also beneficial for vector arrays since the pointer to the modified row is not updated unnecessarily.

First, we describe the original transformation as developed by David Cann [OC88, Can89] for the vector-of-vectors array model. We also provide an example to demonstrate the inefficacy of reference inheritance under the flat array model. We then present the necessary modifications in Section 3.2.2.

3.2.1 The current transformation

Updating a value in a multidimensional array often causes unnecessary copying. To update a single value in an N-dimensional array, a series of write operations is required, one for each dimension. Since each of these write operations accesses the same array, each write operation contributes to the array's reference count. Due to the artificially high reference count, $N-1$ copy operations are performed at run-time.

To prevent these unnecessary copy operations, the optimization *reference inheritance*[OC88] restructures the graph. Each subgraph that performs a subarray update operation is made a direct descendant of the NoOp node that performs the conditional copy on the next outer subarray, sequentializing the execution of the NoOp nodes. Based on this new order, reference count information is then modified so that each subarray update operation does not introduce an additional reference count. As a result, many of the subsequent NoOp nodes do not perform a copy operation.

To perform *reference inheritance*, we need to consider only local information. This transformation, as illustrated in Figure 5, is applied for each AReplace and AElement pair that accesses the same subarray; the order of application is irrelevant [Can89]. This transformation is only applied when the subgraph is used to modify a subarray. In an earlier preparation phase, **W** marks are added to the graph to identify these edges. In the original transformation, a NO mark is attached to the AReplace node. This mark indicates that the AReplace node should not decrement the reference count of the original i^{th} row of **A**. The NO mark is not attached to the AReplace node when the transformation is used for flat arrays since a separate reference count is not maintained for each subarray.

Applying the transformation to the graph in Figure 4 produces the graph in Figure 6. The subgraph that updates the innermost subarray has been placed directly under the NoOp node associated with the bottommost AReplace node. The initial reference count for the array returned by the NoOp node is set to 1 by the sr pragma, even though there are two operations that access the array. In this example, the innermost NoOp node does not perform a copy operation since the row's reference count is 1. However, the bottommost AReplace node unnecessarily copies the modified i^{th} row on top of itself.

Notice that an artificial dependency edge (ADE), represented by a dashed line, has been added to the graph. This edge prevents the bottommost AReplace node from executing until after the innermost AElement node has executed. This ensures that the array read operation completes before the replacement operation begins.

Although this transformation is safe when arrays are represented under the vector-of-vectors model, problems arise when arrays are represented under the flat model. Consider the SISAL expression "A[i:B], A[i][j:0]", where the two-dimensional array **A** is stored in contiguous memory. This expression produces a two-dimensional array and a one-dimensional array. The resulting two-dimensional array is identical to the array **A** except the i^{th} row is replaced with the values contained in the one dimensional array **B**, and the resulting one-dimensional array is identical to the i^{th} row except the j^{th} element is replaced with the value 0. After reference inheritance has been applied, the resulting IF2 graph allows data values to be corrupted (see Figure 7). Since the rightmost NoOp node does not perform a copy operation,[2] the physical space of the produced arrays is overlapped. However, both arrays are modified independently; the leftmost AReplace node copies the array **B** onto the i^{th} row, and the rightmost AReplace inserts the value "0" into the j^{th} element of the same row.

3.2.2 Reference Inheritance for the flat array model

In many cases, the subarray being updated is placed back into the same location within the original array, as illustrated in Figure 8. Under this situation, reference inheritance always provides correct results under the flat model, but the modified subarray is unnecessarily copied back into the original location. To eliminate the unnecessary array copying under the flat model, two problems must be solved:

1. Reference inheritance must be generalized to work under the flat array model

2. The redundant copying of subarrays must be identified and eliminated.

To accomplish this, a two-phase optimization is employed. The first phase is a generalized version of reference inheritance which only restructures the graph. This forces the NoOp node for the outer dimension to dominate the graph for the subarray update operation. A second phase is then used to determine if the AReplace node for the outer dimension receives the modified subarray. In this case the graph is said

[2] The reference count is guaranteed to be 1.

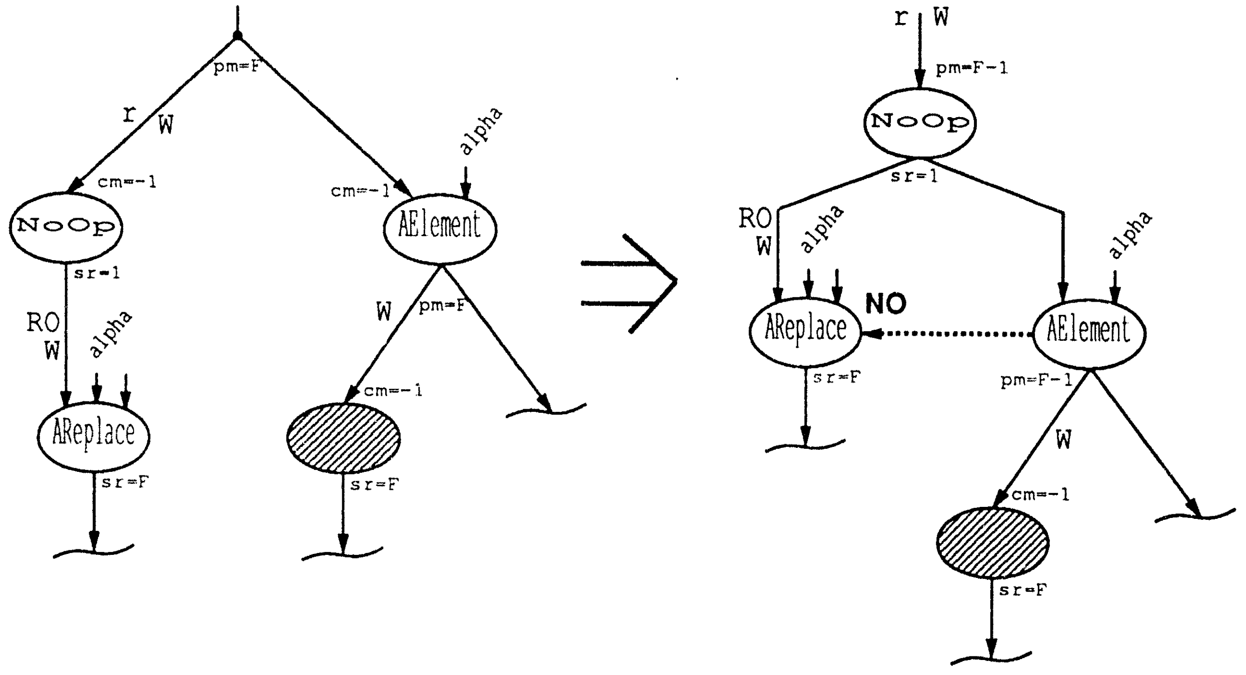


Figure 5: Original Reference Inheritance Transformation [Can:89] (F represents the number of output edges)

to be *mutually strong-dependent* (MSD).[3] If the graph is MSD then the unnecessary copy operation is eliminated.

To generalize reference inheritance two changes are needed. First, the interior NoOp node must perform a copy operation if the resulting array is not passed to the bottommost AReplace node. In the first phase of the optimization, we force the NoOp node to perform the copy operation by setting the initial reference count for the array returned by the exterior NoOp node to 2, the number of direct descendants. If the interior copy operation is unnecessary, the reference count information is adjusted by the second phase. Second, the bottommost AReplace node must be delayed until after the interior NoOp executes to ensure that the original subarray is copied, when necessary. This is accomplished by repositioning the ADE so that its source is attached to the interior NoOp.[4]

This transformation is illustrated in Figure 9. Notice that the source of ADE is not restricted to a particular node type (depicted as a hatched node in the figure); the node may be a compound node which contains the NoOp that performs the copy operation. Additionally, there may be several nodes that are descendents of the AElement node. An ADE must be inserted for each of these nodes since each node must perform any necessary operation prior to the modification of the original subarray.

In the second phase, a graph traversal is used to determine if the graph is MSD. For each AReplace node, the graph is walked in reverse dataflow order starting with the third input of the AReplace node. If a path exists to the exterior NoOp node, the graph is deemed MSD. This operation can be performed in constant time since there is only one backward path and the path length is bounded by four.

If the graph is MSD, the current array update operation can occur without inducing any copy operations, i.e., it can operate in-place. The graph is annotated to reflect this. First, the interior NoOp node is forced not to perform a copy operation. The NoOp node's **r** mark, which indicates conditional copying, is changed to an **R** mark. Additionally, an **O** mark is attached to the interior NoOp node. Together, the **R** and **O** marks indicate that no copying is to be performed by the NoOp node. If there are multiple interior NoOp nodes, then the one that lies along the path from the bottommost AReplace to the exterior NoOp node must be the NoOp node that does not perform a copy operation; we refer to this NoOp node as the major NoOp node. The edges to the other NoOp nodes must have their **r** mark removed, forcing them to copy

[3]We have borrowed the term, mutually strong-dependent, from [Kim88]. The term is used here in a similar sense.

[4]If the graph is MSD, this ADE is unnecessary and is removed by the second phase of the optimization.

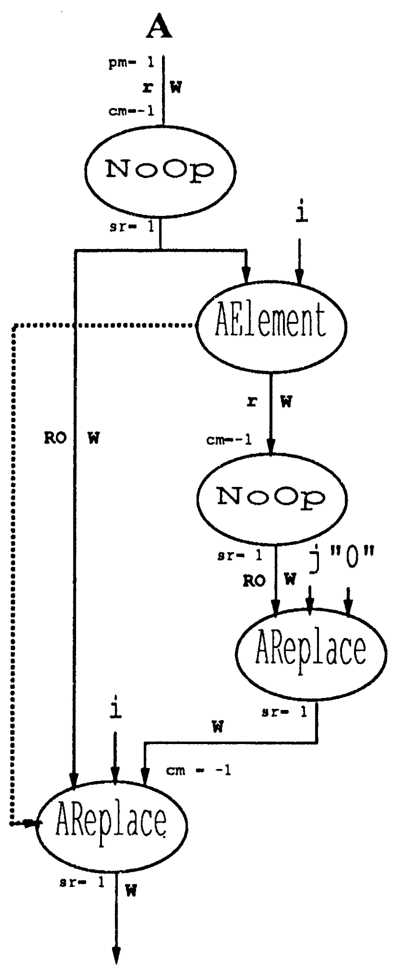


Figure 6: IF2 graph for the SISAL expression "A[i,j:0]", after reference inheritance

the subarray.

Recall that each interior NoOp node is a source for an ADE that extends to the bottommost AReplace node. These ADEs ensure that all necessary copying is completed before the outermost update operation is started. However, the ADEs must be adjusted to ensure that copying is performed before the innermost AReplace node executes. Since the major NoOp node does not perform a copy operation, the innermost AReplace node directly modifies the subarray. Additionally, other read operations performed on the subarray must be completed prior to the innermost update operation. This constraint can be assured by applying the node-reordering algorithm [Can89] to the subgraph rooted at the AElement node.

Second, the bottommost AReplace node must be informed that the subarray has been updated in-place. A **P** mark is attached with the AReplace node's third input edge. This mark indicates that the corresponding array has been constructed in the proper location. In this situation, the AReplace node is used only as a synchronization point.

As a final step, reference counting information is adjusted. Each reference count pragma on the path from the exterior NoOp node to the third input of the AReplace node is decremented by one. This ensures that the run-time reference counts correspond to the operation of the graph.

If the graph is not MSD, the interior NoOp node is forced to perform a copy operation, i.e., the **r** mark is removed. Under the vector-of-vectors model, this copy operation may be unnecessary. Type information can be used to determine how the array is stored. If the array is stored hierarchally then the reference counting information from the exterior NoOp node to the interior NoOp node can be adjusted as defined by the original reference inheritance transformation. However, the **r** mark associated with the interior NoOp

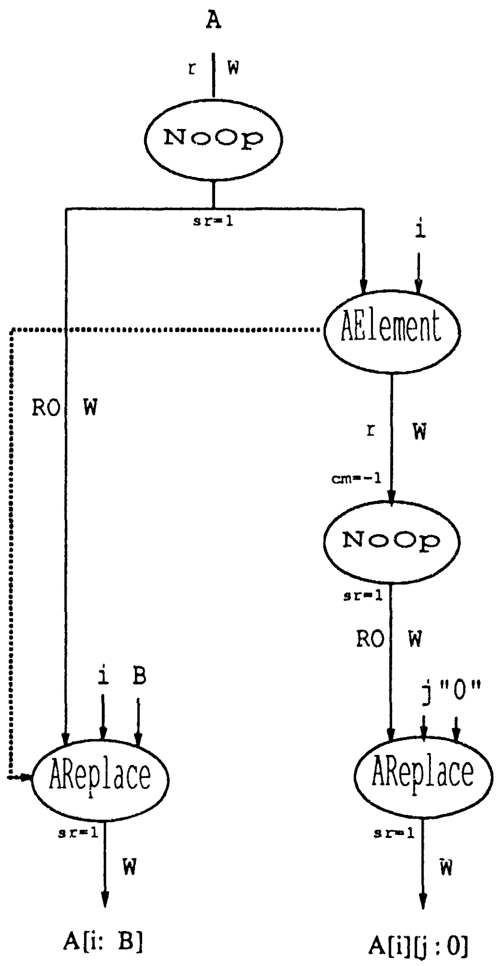


Figure 7: IF2 graph for the SISAL Expression "A[i:B], A[i][j:0]"

node cannot be changed into an **R** mark because of the possibility of row sharing (as in [FCO90]). Further optimizations that are part of update-in place may determine that the interior NoOp node does not perform a copy operation. If copying is not perform and the graph is MSD then a **P** mark can be associated with the 3^{rd} edge of the bottommost AReplace node of the graph, thus saving a pointer-update operation.

4 Preliminary Indications

A partial implementation of the reference-inheritance transformation for the flat array model was developed. Because SISAL 1.2 does not support flat arrays, only **P** marks were added to the MSD graphs. Reference counting information was adjusted as defined by the original reference-inheritance transformation. These marks have no effect on the execution of SISAL applications, but are used to identify MSD graphs. Additionally, the code-generation phase, which produces C code, of the SISAL 1.2 compiler was modified. A print statement is inserted for each AReplace node that is a member of a AReplace and AElement pair. The print statement indicates whether or not the AReplace node is part of a MSD graph and reports the number of elements passed to the AReplace node on its third input, i.e., the number of elements needed to be copied.

To determine the effectiveness of our technique, a number of SISAL applications that manipulate multidimensional arrays were compiled and executed. These programs included: two- and three-dimensional convolution (2d_conv.sis and 3d_conv.sis), a 40 time step gel-chromatography simulation (ricard.sis), matrix inverse (inverse.sis), and loop 13 of Livermore Loops benchmark (loop13.sis). The optimization technique is applied mostly for applications that use while

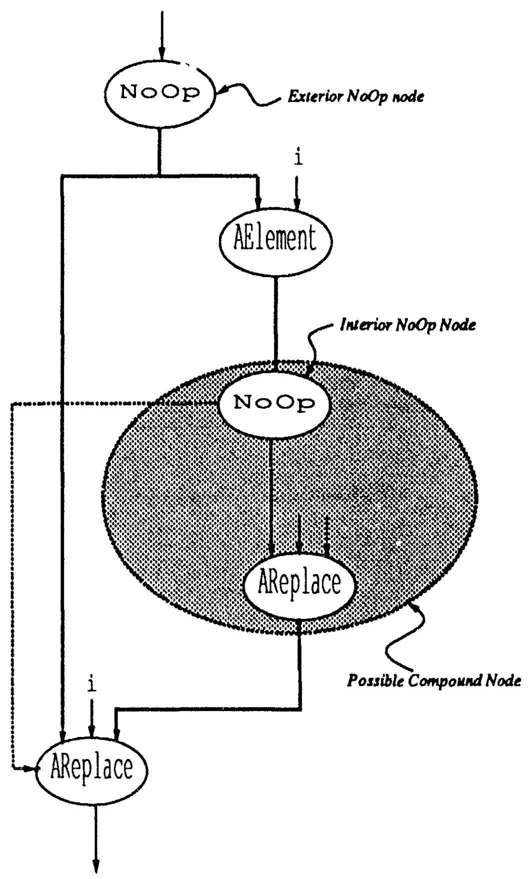


Figure 8: Graph template for an update operation

loops. As such, the convolution functions used in our test suite were implemented using while loops, the other applications are available with the SISAL distribution.

In Table 2, we report both the number of applications of reference inheritance and the number of MSD graphs produced during compilation. These numbers where obtained by examining the C code produced by the modified SISAL compiler, i.e., the inserted print statements were counted. The C code was then compiled with a native C compiler and executed. A shell script that examines each program output calculated the number of MSD graphs executed[5] and the total number of array elements that are not copied. We also indicate both the number of non-MSD graph and the number of elements that must be copied to preserve the contiguity requirement. We present these results in Table 3.

[5]This number also represents the number of pointer-copy operations saved under the vector-of-vector model.

5 Conclusions and Future Work

Update-in-place analysis has virtually eliminated the copy problem in SISAL 1.2 [Feo90]. These optimizations must be extended to handle both vector-of-vectors arrays and true multidimensional arrays. Additionally, there is a higher potential for copying with monolithic arrays than with vector arrays. Updating a single element in a shared array can induce a full array copy, whereas an array copy operation is only performed under the vector-of-vectors model if the lowest dimensional subarray is shared.

Reference inheritance is an optimization that was developed for the vector-of-vectors array model. As described in this paper, this optimization can be generalized and extended to reduce copying under both array models. In our approach, we stayed within the context and current definition of both IF1 and IF2. An alternative approach is to define an extended AREPLACE node. The extended node would receive an n-dimensional array, n indices, and the replacement value for a total of $n+2$ inputs (see Figure 10a). How-

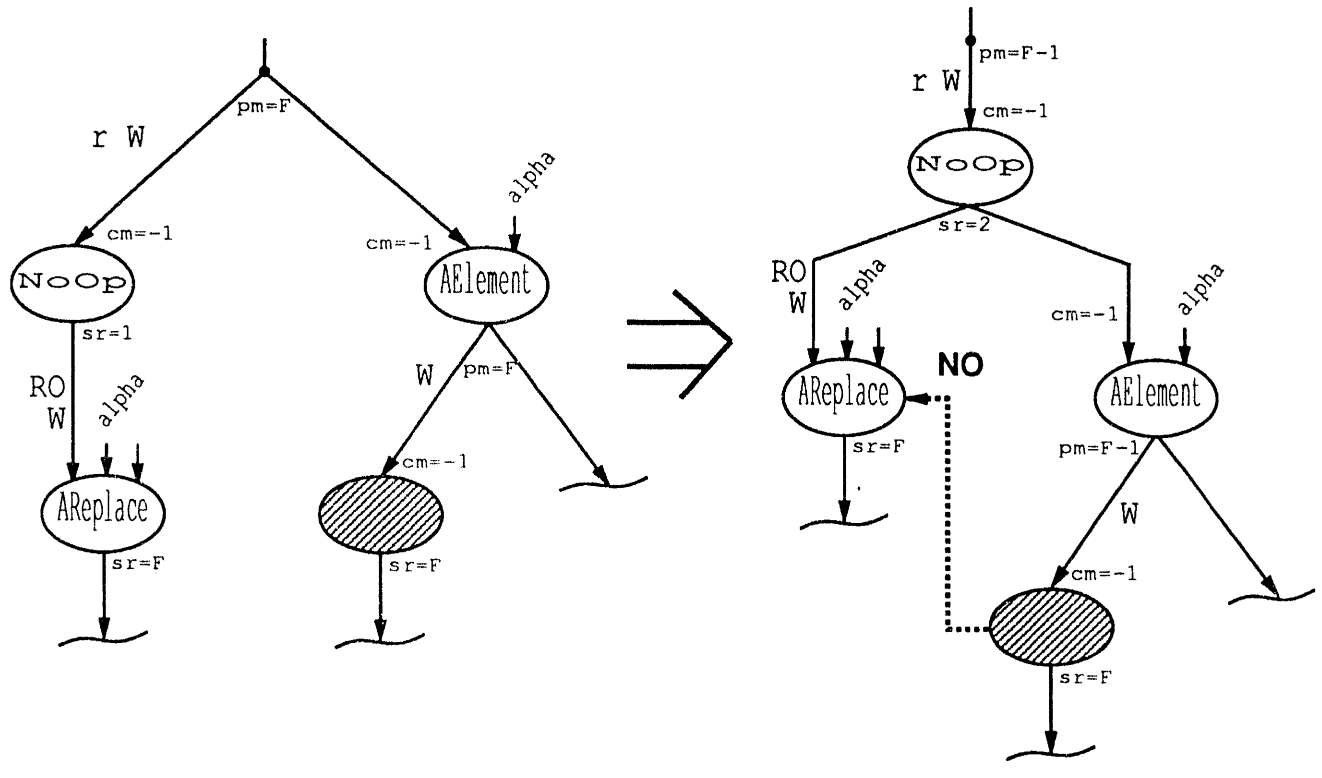


Figure 9: Modified reference inheritance graph transformation

ever, we see two main problems with this approach:

1. Complicates the definition of the AREPLACE node
   In the current definition of IF2, the AREPLACE node can update a number, say n, of sequential array elements. The AREPLACE node receives $n + 2$ inputs which consist of an array, an index, and n replacement values. For example, the array-update operation for the SISAL expression "A[i:5,4,3]" is performed by a AREPLACE node that has five inputs. The array elements at positions i, i+1, and i+2 are replaced with the values 5, 4, and 3, respectively. Although type information can differentiate between the two node definitions, the extended definition unduly complicates the semantics of the AREPLACE node.

2. Prevents the application of some optimizations, e.g., common subexpression elimination
   Consider the SISAL expression "A[i,j:0], A[i,j]". Using the extended AREPLACE node prevents an indexing operation from being shared by the array-read and the array-write operations, as depicted in Figure 10a. If the extended AReplace node is exploded, as in Figure 10b, then the indexing operation, performed by the topmost AELEMENT node, is identified as a common subexpression.

The extended reference inheritance transformation eliminates much of the copying used to preserve the contiguity requirement for arrays. However, our approach cannot remove the extra copying induced by non-MSD graphs. This extra copying can occur when a region of a multidimensional array, such as a row, is updated as a single operation.

For example, in the SISAL expression "A[i:B], A[i]", the one-dimensional array **B** needs to be copied into the two-dimensional array **A**. As shown in Figure 11, array **B** is copied onto the i^{th} row of **A**. Program analysis, such has build-in-place [Ran87], may be extended to determine that the array **B** can be built within the array **A**, eliminating the copy operation. However this can only occur if the construction of the array **B** is delayed until all operations that access the i^{th} row of **A** are completed. Since this reduces parallelism within the graph, any optimization needs to consider the tradeoff between copying and the loss of parallelism.

To eliminate the copy problem in SISAL 2.0, ad-

Program	Appl. of Ref. Inher.	Number of Graphs Types	
		MSD	non-MSD
2d_conv.sis	1	1	0
3d_conv.sis	2	2	0
ricard.sis	5	5	0
inverse.sis	6	1	5
loop13.sis	7	7	0

Table 2: Number of MSD graphs identified

Program	Data-set Size	Number of Graphs Types		Element Copies	
		MSD	non-MSD	Saved	Performed
2d_conv.sis	10^2	56	0	560	0
3d_conv.sis	10^3	686	0	37730	0
ricard.sis	5×1315	100	0	200000	0
inverse.sis	10^2	10	200	100	2000
loop13.sis	—	448	0	200704	0

Table 3: Number of MSD graphs executed, and number of array elements copied (The 2d and 3d convolution applications used 3^2 and 3^3 kernel, respectively. The input used for loop13.sis was provided by benchmark.)

ditional work needs to be performed. Currently IF2 [WSYR86] only defines the semantics of array nodes based on the vector-of-vectors model. Our current work is to determine the modifications to IF2 that are necessary to support monolithic arrays. Additionally, we are examining the semantics of other nodes that manipulate dope vectors. These nodes would support the functionality of SISAL 2.0 array operation without introducing unnecessary copying.

Acknowledgments

I thank John Sieg of the University of Mass Lowell and Rodney Oldehoeft of Colorado State University for their numerous comments and suggestions which have helped to improve this paper. Additionally, I thank Fred Lewis of the University of Mass Lowell for the many fruitful discussions.

References

[BOCF92] A. P. W. Böhm, R. R. Oldehoeft, D. C. Cann, and J. T. Feo. *SISAL Reference Manual* Language Version 2.0. Colorado State University — Lawrence Livermore National Laboratory, 1992.

[Can89] David C. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, 1989. CSU Technical Report CS-89-108.

[Can92] David C. Cann. Retire FORTRAN? A debate rekindled. *CACM*, 35(8):81–89, August 1992.

[CO88] D. C. Cann and R. R. Oldehoeft. Reference count and copy elimination for parallel applicative computing. Technical Report CS-88-129, Department of Computer Science, Colorado State University, November 1988.

[FCO90] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10(4), December 1990.

[Feo90] John T. Feo. Arrays in SISAL. Technical Report UCRL-JC-106081, Lawrence

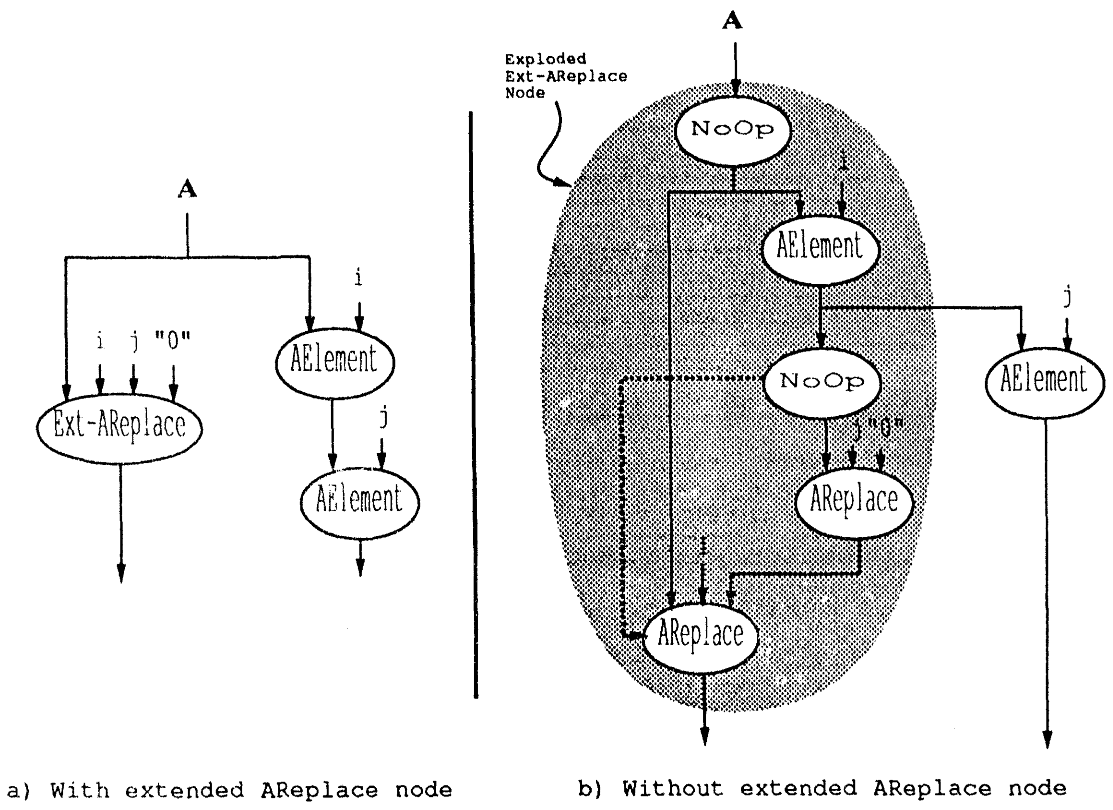

A
i
i j "0"
Ext-AReplace
AElement
j
AElement
Exploded Ext-AReplace Node
NoOp
i
AElement
j
NoOp
j "0"
AReplace
i
AReplace
AElement
a) With extended AReplace node
b) Without extended AReplace node

Figure 10: Undecorated IF2 graphs for the SISAL expression "A[i,j :0], A[i,j]", with and without the extended AREPLACE node

Livermore National Laboratory, September 1990. *Published in* The First International Workshop on Arrays, Functional Languages, and Parallel Systems.

[Gao90] Guang R. Gao. *A Code Mapping Scheme for Dataflow Software Pipelining.* Kluwer Academic Publishers, 1990.

[GH83] G. Goos and J. Harmanis. *The Programming Language Ada Reference Manual.* American National Standards Institute, 1983.

[Hud86] Paul Hudak. A semantic model of reference counting and its abstraction (Detailed summary). In *Proceedings 1986 ACM Conference on LISP and Functional Programming*, pages 351–363. ACM, August 1986.

[Hud87] P. Hudak. A semantic model of reference counting and its abstraction. In *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood, 1987.

[Kim88] Sung Jo Kim. *A General Approach to Multiprocessor Scheduling.* PhD thesis, The University of Texas at Austin, Austin, Texas 78712–1188, February 1988.

[Mac83] Bruce J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation.* CBS College Publishing, New York, New York, 1983.

[MSA+85] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, Chris Kirkham, Bill Noyce, and Robert Thomas. *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2.* Lawrence Livermore National Laboratory, M-146 edition, March 1985.

[OC88] Rodney R. Oldehoeft and David C. Cann. Applicative parallelism on a shared-memory multiprocessor. *IEEE Software*, 5(1):62–70, January 1988.

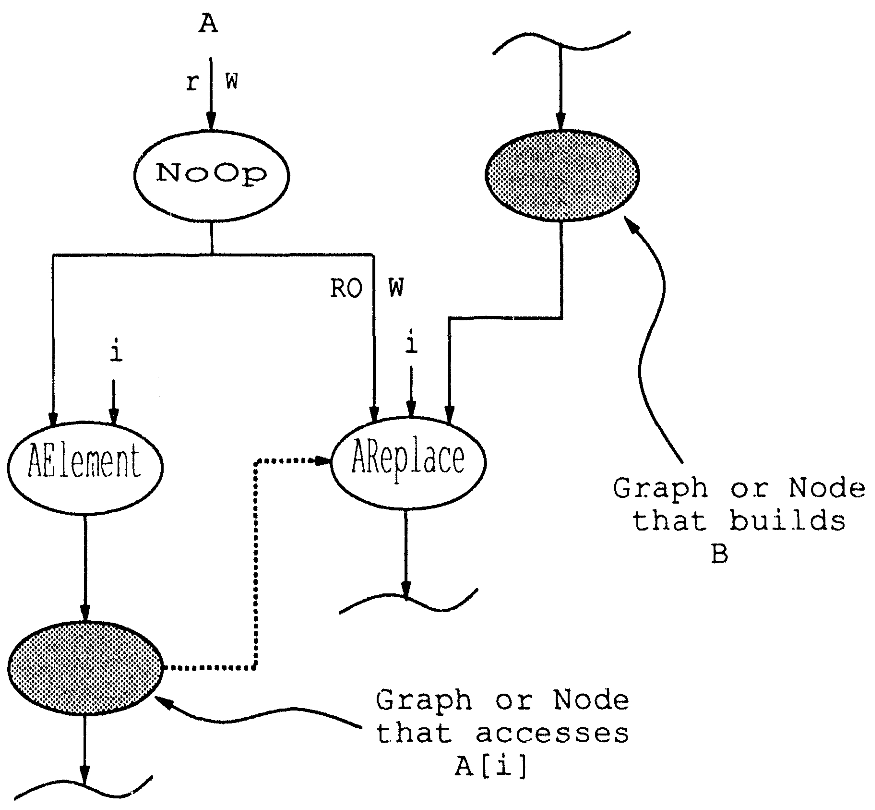


Figure 11: Unnecessary copying because of a non-MSD graph

[Old92] R. R. Oldehoeft. Implementing arrays in SISAL 2.0. *Proceedings of the Second SISAL Users' Conference*, pages 209–222, December 1992.

[Ran87] John E. Ranelletti. *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages.* PhD thesis, University of California Davis, 1987. UCD Technical Report UCRL-53832.

[Seb93] Robert W. Sebesta. *Concepts of Programming Languages.* Benjamin Cummings, Second edition, 1993.

[SG85] Stephen Skedzielewski and John Glauert. *IF1 — An Intermediate Form for Applicative Languages.* Lawrence Livermore National Laboratory, Livermore, CA, M-170 edition, July 1985.

[SS88] Stephen K. Skedzielewski and Rea J. Simpson. A simple method to remove reference counting in applicative programs. Technical Report UCRL-100156, University of California Davis — Lawrence Livermore National Laboratory, November 1988.

[WSYR86] Michael Welcome, Stephen Skedzielewski, Robert Kim Yates, and John Ranelletti. *IF2 — An Applicative Language Intermediate Form with Explicit Memory Management.* University of California — Lawrence Livermore National Laboratory, M-195 edition, November 1986.

Increasing Parallelism for an Optimization that Reduces Copying in IF2 Graphs

Steven M. Fitzgerald
Dept. Computer Science
University of Massachusetts Lowell
Lowell, MA 01854
sfitzger@cs.uml.edu

Abstract

To exploit the benefits of applicative languages, such as SISAL, optimizations are needed to eliminate inefficiencies that can result from a naive implementation. In particular, costs associated with copying large data aggregates can be significant, outweighing the benefits achieved through parallel execution. Within the SISAL compilation environment, an optimization known as node-reordering *is performed to reduce copying [Can89]. This optimization constrains the execution order of IF2 [WSYR86] graphs by introducing artificial dependency edges (ADEs). Although the resulting IF2 graph provides greater opportunities to eliminate expensive copy operations, parallelism is restricted. Additionally, the introduced ADEs can increase both memory usage and token traffic for programs that execute on fine-grain architectures.*

In this paper, we describe a new framework for the node-reordering *optimization. The resulting algorithm, which is based on the algorithm presented in [Can89], prevents unnecessary ADEs from being inserted into IF2 graphs. As a result of our algorithm, parallelism is constrained only when necessary to eliminate costly copy operations. Furthermore, removing the overhead associated with unnecessary ADEs results in better performance of SISAL programs that run on dataflow architectures [Fit93].*

1 Introduction

To define algorithms for parallel architectures, languages that follow the applicative paradigm have been proposed [Arv88, Bac78, Den80, AN90, FCO90]. In applicative languages, such as SISAL [BOCF92, MSA+85], computation is carried out via the evaluation of expressions. Since expressions are not influenced by side effects, the evaluation order of expressions is dependent only on the availability of values. As values are computed, separate copies can be provided to many independent operations that can execute simultaneously, thus exploiting parallel architectures.

An implementation that strictly adheres to the applicative model is required to copy all data values. However, the cost associated with copying large data aggregates, such as arrays, can become prohibitive, nullifying the benefits achieved through parallel execution. To reduce this cost, aggregates can be passed by reference, with copying performed only by operations that alter the value of an aggregate. A compiler can minimize these remaining copy operations by restricting evaluation order. For example, a write operation can be delayed until after all read operations on the same aggregate have been performed. Since the write operation is now the last operation to reference the aggregate, it can directly modify the aggregate without affecting program semantics.

Within the SISAL compilation environment, the execution order of a program is modeled by an IF2 [WSYR86] graph. Based on this internal program representation, a series of optimization techniques, known collectively as ***update-in-place***, is applied to reduce the number of aggregate copy operations [Can89]. One of these optimizations, ***node-reordering***, introduces artificial dependency edges (ADEs) into IF2 graphs to defer write operations until after all pending read operations have executed. Although this results in better performance because of the elimination of many costly copy operations, many "redundant" and "superfluous" ADEs are introduced. On fine-grain architectures, such as dataflow, these edges can decrease performance since they increase both token traffic and data memory usage. Additionally, superfluous ADEs unnecessarily restrict execution order, further decreasing parallelism.

In this report, we present a new approach to the node-reordering algorithm that was originally designed by David Cann [Can89]. First, we present an overview of dataflow computation and the intermediate language IF2. In Section 3, we define two types of ADEs, redundant and superfluous, and discuss the costs associated with these ADEs. The new node-reordering algorithm, which does not insert unnecessary ADEs, is presented in Section 4. In Section 5, results are provided. We then conclude this report with a brief overview and a description of future work.

2 Execution Model

2.1 Dataflow and IF2

Program execution in dataflow architectures is data driven [Den80]. Graphs that depict data dependencies serve as the basis for these architectures and can be used to define programs for these architectures [Den80]. Each node in a dataflow graph represents an operator that executes when all its operands are available. The node is said to *fire*, producing a value that flows along edges to other nodes.[1] Thus, program execution is based on dataflow order, which is constrained only by data dependencies represented by edges in the graph.

IF2 [WSYR86] is a graph-based language designed as an intermediate form for applicative languages such as SISAL. Although IF2 is based on the applicative model, memory management information can be incorporated into a dataflow graph via a set of primitive operations. Using IF2, a series of optimizations can be performed to improve the execution efficiency of algorithms. For example, update-in-place analysis [Can89] is performed in the SISAL compilation environment to eliminate unnecessary copy operations. ADEs are introduced to delay the execution of write operations until after read operations have completed. Although this limits parallelism because of the restricted execution order, many expensive copy operations are eliminated. In most situations, program execution is improved.

Consider the SISAL expression "A[i,j:0], A[j,i]" which returns an array and an integer. The returned array is identical to **A** except the value at location "[i,j]" is replaced with the value 0. Both expressions can execute in parallel, but depending on the order of evaluation a copy operation may or may not be required. If the array update operation (A[i,j:0]) executes first, the array **A** must be copied to ensure that the array read operation (A[j,i]) retrieves the correct value.[2] We can prevent the copy operation if the array write is delayed until the array read operation completes.

To restrict execution order, node-reordering introduces ADEs into an IF2 graph. For our example, the IF2 graph produced by update-in-place is depicted in Figure 1. Three ADEs, represented by dashed edges, have been inserted: two by the node-reordering optimization and one by another optimization, *reference-inheritance*. The ADEs introduced by node-reordering are placed between array read and array copy operations, e.g., from AELEMENT nodes to NOOP nodes. The NOOP nodes have been inserted into the graph to indicate the location of potential copy operations. Since these ADEs affect the amount of copying performed, we focus our attention on ADEs introduced by node-reordering.

In our example, these ADEs prevent the top-most NOOP node from firing until the two right-most AELEMENT nodes have fired, completing the array read operation. Once the read operation completes, the top-most NOOP node can execute, allowing the array update to proceed. At run-time, NOOP nodes conditionally copy a data aggregate based on reference count information [CO88]. Since all other operations, not dependent on the top-most NOOP node, have completed, the reference count for the array **A** is equal to 1. In this case, the NOOP node only serves as a synchronization point. The array **A** is not copied but directly modified, thus eliminating a copy operation.

2.2 Activity Frames and the Cost of ADEs

In many prototype dataflow computers, an activity frame or template is used to represent operators in programs [Den80, AN90]. Each of these frames specifies the operation code of an operator and the destinations of its results.[3] Additionally, places are provided to store each operand until the frame is ready for execution. The basic form of a frame is shown in Figure 2.

In some dataflow architectures, these activity frames reside in a data memory. Once all the operands of a frame have been supplied, the operation is executed, producing the appropriate number of results.

[1] A separate copy is made for each output edge.

[2] Consider the case where the values of i and j are the same.

[3] Other information may also be included in a frame, e.g., a tag used in dynamic dataflow [BG90].

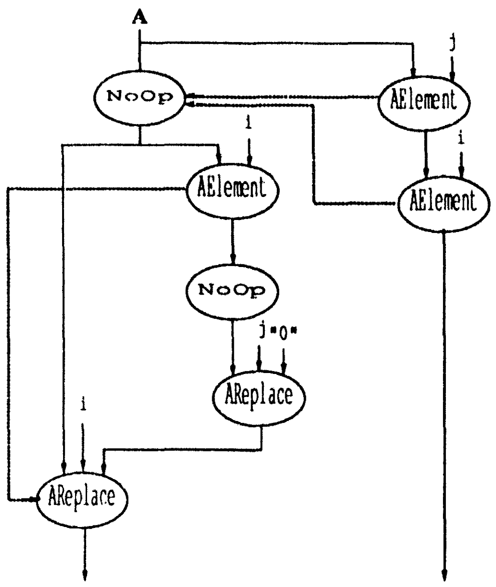


Figure 1: IF2 graph for the SISAL expression "A[i,j:0], A[j,i]"

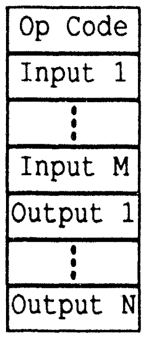


Figure 2: Activity template for a dataflow node

Each of these results is delivered to another activity frame within a packet consisting of a value-destination pair. The system delivers a separate packet for each reference to a result. Once the packet is received by the destination frame, the value is stored, where it resides until the frame is executed. In this manner, each data value requires two storage locations: one in the source frame that indicates its destination, and one in the destination frame that contains its value. Additionally, each transmitted data value introduces a packet that the system must process.

The IF2 graph in Figure 1 can be directly transformed into a dataflow graph that uses activity frames. The resulting graph is shown in Figure 3. In this type of graph, it is easy to see the cost associated with each edge in a dataflow graph. For example, consider the ADE (edge 4) connecting the top-most AElement node with the top-most NoOp node, depicted in Figure 3. Removing this ADE frees up two memory locations and eliminates one packet. Consequently, each ADE removed can increase the run-time performance of programs that run on fine-grain architectures.

In most dataflow computers, instructions are limited in the number of input and output tokens [AN90, BG90]. Removing redundant ADEs for these architectures can further improve performance. For each activity frame that transmits a multiple number, say N, of ADEs, additional activity frames must be inserted into the graph. On machines that do not have an iterative instruction, e.g., TUP instruction, a tree of $N-1$ duplication nodes (DUPs) must be inserted [BG90]. Similarly, a tree that collapses several ADEs into one is needed for each frame receiving multiple ADEs. These additional frames affect both the code size and the token traffic of a program.

In our discussion, we assumed that an ADE is treated like any other edge in the graph. However their are other possible implementations for ADEs that would require less memory. For example, a memory location contained in the activity frame can be used to count the number of ADEs received by the frame. To schedule a frame, this value is compared to the number of ADEs required which must also be stored in the frame. In this manner, only two memory locations per activity frame are needed to implement ADEs. However, under this approach an activity frame must be processed each time an ADE is received, decreasing performance. Furthermore the scheduling

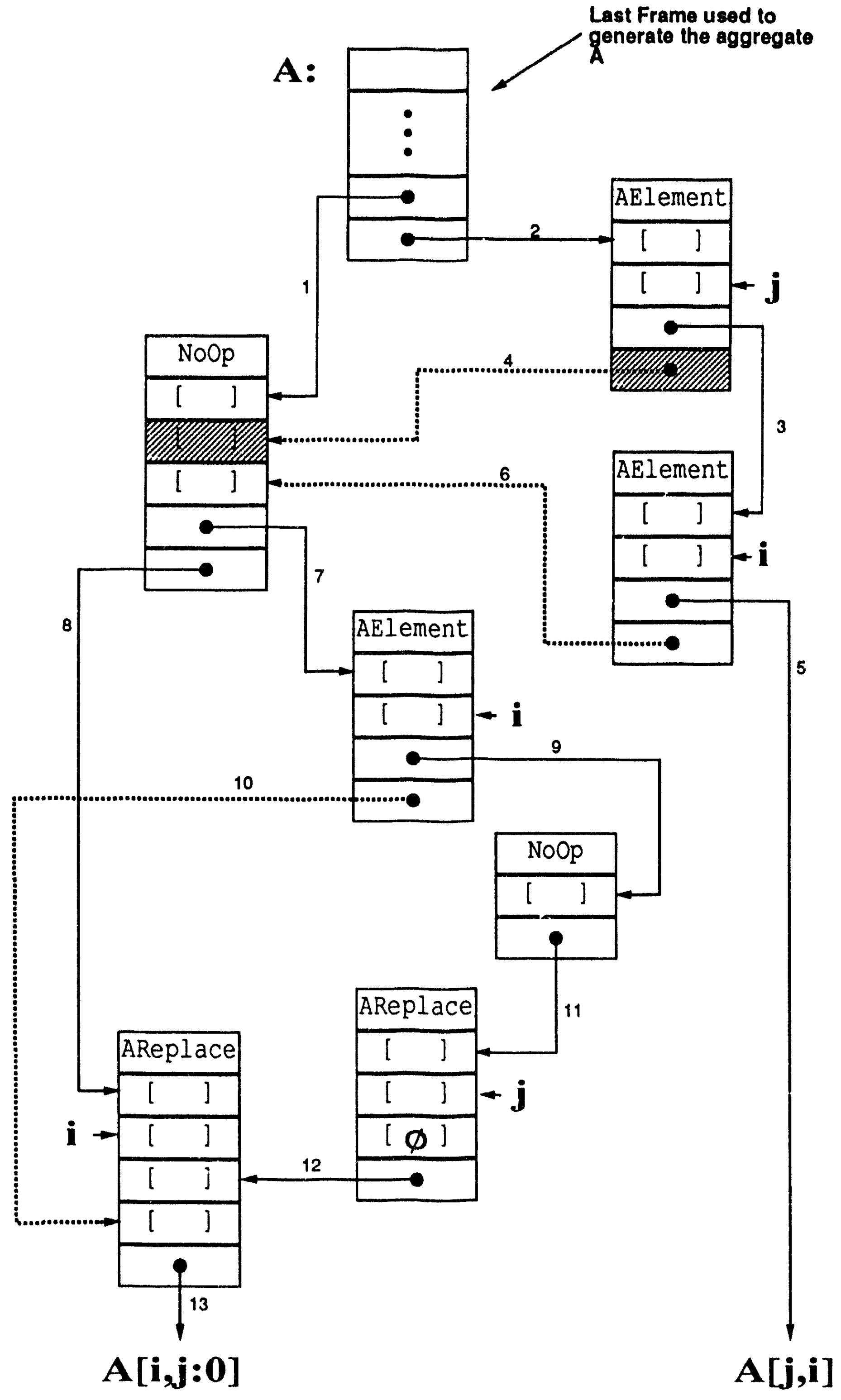


Figure 3: Dataflow graph for the SISAL expression "A[i,j:0], A[j,i]"

of activity frames, i.e., the basic firing rule, is further complicated. Therefore, eliminating as many ADEs as possible is beneficial for performance.

3 Redundant and Superfluous ADEs

To reduce expensive copy operations, node-reordering restricts the execution order of an IF2 graph. ADEs are introduced to delay array write operations until array read operations have occurred. Although the new execution order prevents many costly copy operations, parallelism is reduced. Additionally, execution overhead is increased for fine-grain architectures due to the cost associated with ADEs.

In the original node-reordering algorithm, unnecessary ADEs are inserted. For example, consider the partial IF2 graph for the SISAL expression "A[j,i:0], A[k,i], A[l,i:0]" presented in Figure 4. Four ADEs have been inserted into the graph to prevent both NoOp nodes from executing until both AElement nodes have executed. The ADEs emanating from the top-most AElement node (edges 1 and 2) are "redundant" since the bottom-most AElement node's ADEs also delay the firing of both NoOp nodes. The ADEs associated with the top-most AElement node can be safely removed without disturbing either the graph's execution order or its semantics [Fit93].

The other ADEs (edges 3 and 4) ensure that array read operation, which is performed by the AElement nodes, completes before the start of a potential copy operation. Although this ordering ensures that only one copy of the array **A** is made, the NoOp node that performs the copy is delayed unnecessarily. Deferring the execution of only one NoOp node is sufficient to ensure that an unaltered copy of the array **A** is available for the read operations. We can choose, at compile time, the NoOp node to be deferred. Since the ADEs associated with the other NoOp node are "superfluous," they can be safely removed from the graph, increasing parallelism.

As depicted in Figure 5, the resulting graph contains only one ADE (edge 3). Edges 1 and 2 have been identified as being redundant, and edges 2 and 4 have been identified as being superfluous.[4] Alternatively, we could have identified edges 1 and 3 as superfluous. Since reference counting determines whether a NoOp node performs a copy operation, we can arbitrarily choose either NoOp node to be the sink for the remaining ADE.

[4]Edge 2 can be classified as either redundant or superfluous; its final classification is based on the classification of edge 4.

In general, a tree of nodes is associated with an array aggregate. The interior of the tree is formed by a series of AElement nodes that de-reference the data aggregate. Each AElement node decomposes the array by one dimension. The leaves of the tree consists of other nodes that either read or update each subaggregate. For example, consider the graph fragment for the SISAL expression

```
let
    B := A[l]
in
    A[j,i:0], A[k,i], B[i:0]
end let
```

depicted in Figure 6. The two top-most AElement nodes form the interior of the tree, and decompose the array into subarrays. These subarrays are then accessed by the leaf nodes, which return either a modified copy of a subarray or a scalar value. Notice that the bottom-most AElement node is considered a leaf node because it returns a scalar element and not a subarray.

In the original node-reordering algorithm, ADEs are inserted connecting each interior node with all NoOp nodes that lie on the frontier of the de-reference tree. Since each leaf node that performs a read operation also maintains an ADE with each NoOp node, the ADEs associated with interior nodes are "redundant" and can be removed. The new algorithm does not insert these edges into the graph. Additionally, a single NoOp node is selected to be the sink for the final set of ADEs. Thus the ADEs associated with the other NoOp node are deemed "superfluous." In this manner, a maximum of one ADE per array read operation is required to prevent the unnecessary copy operation.

Removing unnecessary ADEs can increase performance. Since ADEs restrict the amount of parallelism in a graph, removing "superfluous" ADEs allows some of the parallelism to be recovered without increasing the amount of copying since one of the NoOp nodes is required to perform a copy operation regardless of the presence or absence of ADEs. Additionally, with the reduction of the number of edges in the graph, a mapping algorithm may be better able to partition a graph due to less node interaction. On fine-grain architectures, such as dataflow, removing unnecessary ADEs provides us with two further benefits: less memory used and less token traffic. The reduction of token traffic can increase the throughput of the system.

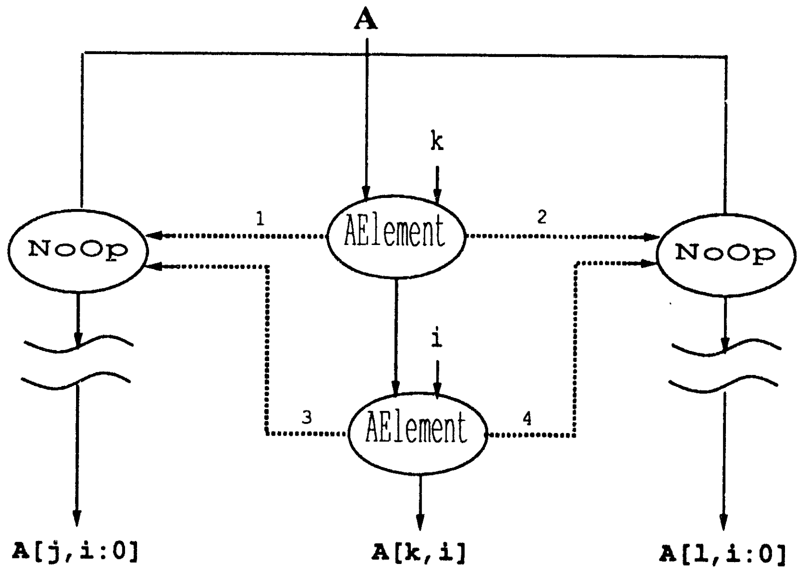


Figure 4: IF2 graph fragment of "A[j,i:0], A[k,i], A[l,i:0]"

4 The New Approach to Node-reordering Algorithm

As part of update-in-place, an IF2 graph is restructured [Can89]. The optimization *node-reordering* inserts artificial dependency edges (ADEs) to defer write operations until all read operations have been performed. As shown in Section 3, many of the ADEs inserted are unnecessary and can be safely removed.

In this section, we describe a modified version of the *node-reordering* algorithm that prevents the insertion of "redundant" and "superfluous" ADEs. The original algorithm developed by David Cann is composed of three phases: read-write set construction, ADE insertion, and graph reconstruction.[5] In the modified version, we retain this organization but have altered the framework of the read-write set construction phase.

4.1 Read-write Set Construction

Before read-write set construction, each edge in an IF2 graph is classified. An earlier phase of update-in-place decorates edges with a **W** mark if it carries data that is to be modified or copied. For example, if an edge's sink is attached to an AELEMENT node, no **W** mark is associated with it. A **W** mark is associated with an edge if its sink is attached to either a NOOP or another aggregate modifier.

[5]In [Can89], the last phase is called node-reordering. However, we have chosen to use the name "node-reordering" to refer to the complete set of three phases.

The read-write sets are constructed by examining all graph nodes in a top-down fashion. A read and a write set are created for each node that is the root of a de-reference tree. The de-reference tree is then traversed using a depth-first search strategy. When a leaf node is encountered, the node number is recorded in either the read set or the write set. The node is recorded in the write-set if the edge that carries the data aggregate to the node is decorated with a **W** mark, otherwise the node is recorded in the read-set. The resulting sets determine the placement of the non-redundant ADEs.

For example, consider the read-write sets constructed for the three dimensional array **A** in the following SISAL expression:

```
let
   B := A[j]
in
   A[i,i,i:0], B[i,i:0], foo(B[k]), B[k,i]
end let
```

The IF2 graph for the expression, depicted in Figure 7, is traversed in dataflow order. In our example, the traversal begins with the node that creates the array **A**. First, an empty read set and an empty write set are assigned to the data array **A**. The de-reference tree for the array is then traversed in depth-first order. The first leaf node encountered, the top-most NOOP, is recorded in the write set since it is connected to the

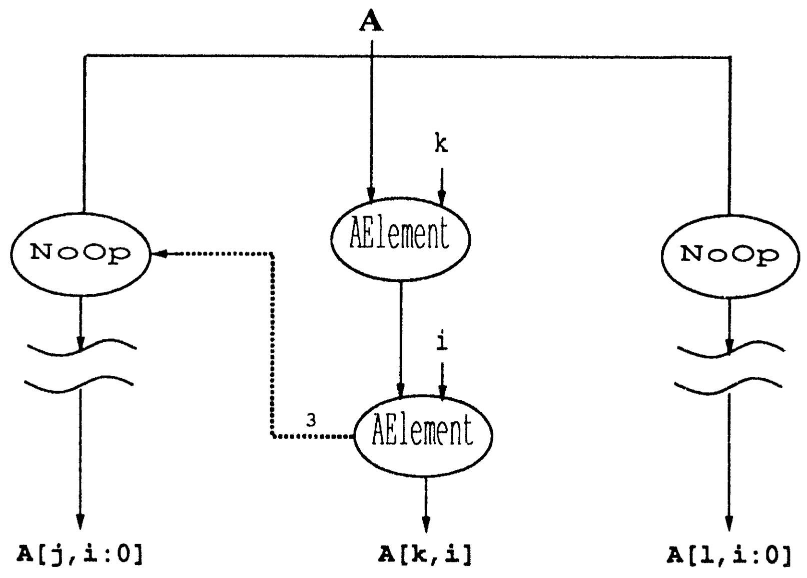


Figure 5: Modified IF2 graph fragment of "A[j,i:0], A[k,i], A[l,i:0]"

de-reference tree by a **W** marked edge. Continuing in this fashion, the other NoOp node is placed into the write set, and the other two leaf nodes are placed into the read set. Notice that the bottom-most AElement node is classified as a leaf node because it returns a scalar value and not a subaggregate of the array **A**. Thus, the final read and write sets for the array **A** are {3, 4} and {5, 10}, respectively.

The next phase of node-reordering, ADE insertion, determines the placement of ADEs. First, the non-redundant ADEs are determined by examining each read-write set pair. The locations of these non-redundant ADEs are defined by the Cartesian product of the read set with the write set. In our example, the read-write set for the array **A** defines four non-redundant ADEs: (3, 5), (3, 10), (4, 5), and (4, 10); here we represent an ADE by the pair (*source-node*, *sink-node*).

To define the final set of ADEs, one node is selected from each write set to be the sink node for the non-superfluous ADEs. For the purpose of copy elimination, this choice can be made arbitrarily. In practice, the selection may be based on a node's depth in the de-reference tree to retain the maximum amount of parallelism. For the IF2 graph in Figure 7, the bottom-most NoOp node has been chosen as the sink for the ADEs for the array **A**; thus the ADEs inserted are (3, 5) and (4, 5).

The final phase of node-reordering is graph reconstruction. Since the inserted ADEs have restricted the original dataflow ordering of the graph, the graph is internally reconstructed to reflect this new order. This phase of the optimization is the same in the modified version as in the original version developed by David Cann.

5 Some Empirical Results

A series of SISAL programs[6] were compiled to determine the number of unnecessary ADEs that are removed by the new *node-reordering* algorithm. Additionally, these programs were augmented with code to allow the number of ADEs encountered at run-time to be calculated. In this section, we present some of these results.

The results indicating the number of ADEs removed at compile time for the following SISAL programs are presented in Table 1: Gaussian elimination (gauss.sis), LaGrangian hydrodynamics (simple2a.sis), quicksort (quicksort.sis), mergesort (mergsort.sis), matrix inverse (inverse.sis), particle dynamics modeler (moldyn.sis), paraffins problem (para.sis), parallel simulated annealing (psa.sis) and three loops contained in the Livermore Loops test suite (loop13.sis, loop10.sis and loop23s.sis). In Table 2 we present the number of ADEs that are executed. Since the ADE counts at run-time are based on all ADEs inserted by update-in-place, we present the total number of ADEs removed at compile time.

[6]All of these programs are available with the SISAL distribution.

Program	Source Lines	Original ADEs	Removed ADEs Redundant	Removed ADEs Superfluous	Remaining ADEs
gauss.sis	227	3	1	—	2
simple2a.sis	1527	3	2	—	1
loop23s.sis	46	6	3	—	3
quicksort.sis	50	7	0	—	7
loop10.sis	54	13	10	—	3
loop13.sis	60	14	4	—	7
mergesort.sis	85	14	0	3	11
inverse.sis	158	17	3	—	14
moldyn.sis	878	27	17	—	10
para.sis	824	61	46	—	15
psa.sis	771	156	41	—	115

Table 1: Counts of ADEs removed at compile-time (only ADEs associated with node-reordering are reported)

Program	Source Lines	Static Original	Static Remaining	Static % Removed	Dynamic Original	Dynamic Remaining	Dynamic % Removed
loop10.sis	54	13	3	76.92	13	3	76.92
loop13.sis	60	18	14	22.22	1152	896	22.22
mergesort.sis	85	13	11	15.38	3197	2999	6.19
inverse.sis	158	29	26	10.34	3441	3351	2.62
moldyn.sis	878	46	29	36.96	—	—	—
para.sis	824	65	19	70.76	107	0	100.00
psa.sis	771	175	134	23.42	8162	5951	27.09

Table 2: Counts of ADEs executed at run-time (all ADEs are reported). Runtime counts are not provided for the "moldyn.sis" program because valid input data was not available at the time of our studies.

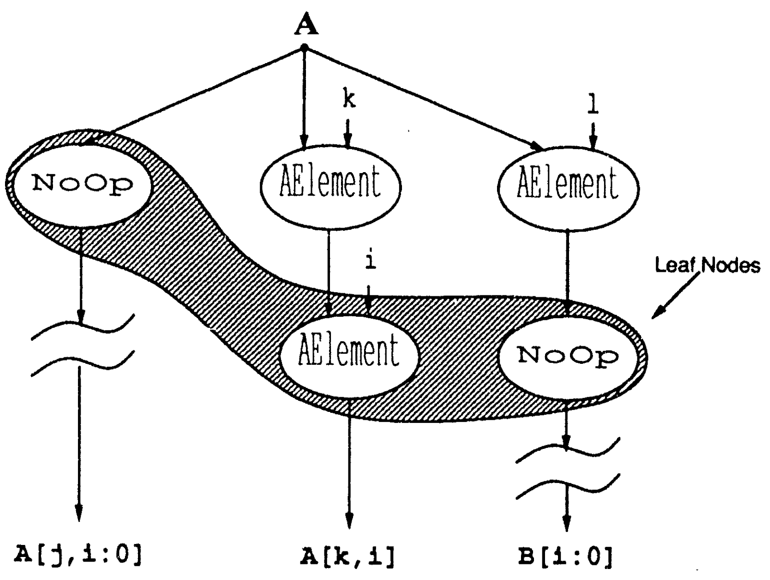


Figure 6: IF2 graph fragment of "A[j,i:0], A[k,i], B[i:0]"

Although the percent of ADEs removed range from 2.62% to 100.00%, a large number of programs can benefit from the modified algorithm, particularly those that make extensive use of multidimensional arrays. Additionally, programs that do not achieve any improvement through the modified algorithm do not incur any increase in compilation cost since each node is examined at most the same number of times as in the original algorithm. Notice that only one program exhibited superfluous ADEs, mergesort. We hope that further analysis will identify a class of algorithms that benefits from the removal of superfluous ADEs.

6 Conclusion

6.1 Overview

In this paper, we present the optimization node-reordering, originally developed by David Cann [Can89], in a new framework. This optimization inserts ADEs between read and write operations to defer write operations until pending read operations have been performed. This restructuring of an IF2 graph increases opportunities to eliminate many expensive copy operations. The algorithm that results from the modified framework is both clear and concise, allowing for a simple implementation. Additionally, ADEs are inserted only when necessary to prevent copy operations. The optimization is well suited for fine-grain architectures since the eliminated ADEs result in a decrease in both memory usage and token traffic.

6.2 Future Work

The modified algorithm prevents many ADEs from being inserted into the graph. Although this allows many operations to be performed in parallel, reference counting information is needed to determine when NoOp nodes must perform a copy operation. The overhead associated with reference counting can decrease performance. Fortunately, most of the reference counts can be eliminated through program analysis [Can89, CO88, Hud86, SS88].

Within update-in-place, the optimization *edge-neutralization* removes reference count pragmas based on the location of ADEs inserted by the node-reordering algorithm [Can89, FCO90]. Since we have eliminated many of these ADEs, we must re-examine this optimization.

For example consider the graph depicted in Figure 8. Using the existing edge-neutralization algorithm, the reference count pragmas associated with the interior nodes of the de-reference tree are not eliminated because the ADEs that would have been inserted by the original *node-reordering* algorithm are not present. Since these interior nodes have implicit ADEs, we should also be able to remove their reference counts. Furthermore, it may be possible to remove all reference counting information within the de-reference tree if we select, *a priori*, the NoOp node that performs the copy operation.

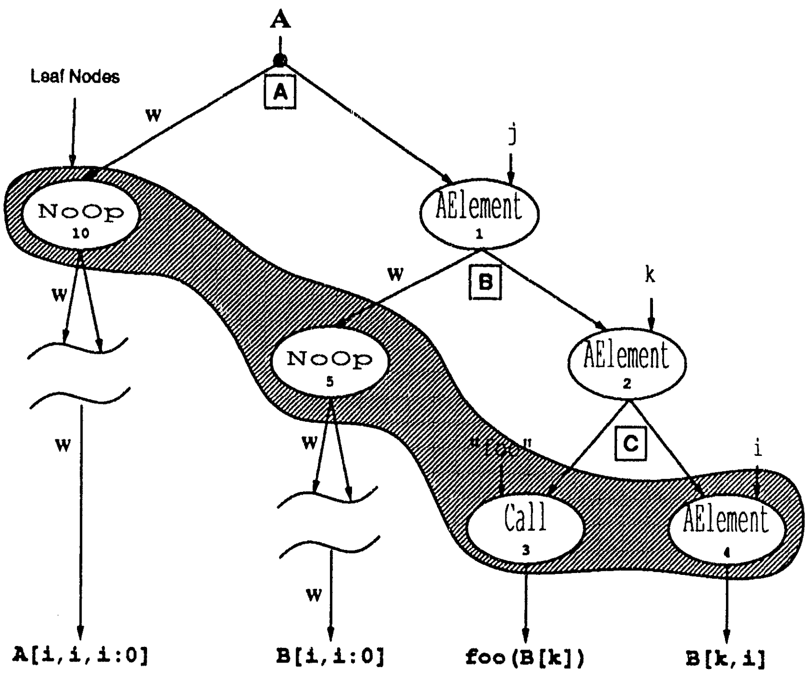


Figure 7: Partial IF2 graph for "A[i,i,i:0], B[i,i:0], foo(B[k]), B[k,i]"

In general, to select the NoOp nodes that perform the copy operations, a set of heuristics may be developed. These heuristics could examine the trade-off between copying and the loss of parallelism. Since only one algorithm in our test suite, mergesort, has "superfluous" ADEs, we must first identify a class of algorithms that might benefit from the proposed heuristics. Currently, we are examining the characteristics of mergesort to determine an appropriate class of algorithms.

Acknowledgments

I am grateful to Linda Wilkens of the University of Mass Lowell for her support and advise. I also thank Rich Wolski of LLNL and Rodney Oldehoeft of Colorado State University for their comments and suggestions on early drafts of this paper.

References

[AN90] Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[Arv88] Arvind. Dataflow approach to general-purpose parallel computing. Computer Science and Engineering, MIT Video Tape, October 1988.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8), August 1978.

[BG90] A. P. Wim Böhm and John R. Gurd. Iterative instructions in the Manchester Dataflow Computer. *IEEE Transactions on Parallel and Distributed systems*, 1(2):129–139, April 1990.

[BOCF92] A. P. W. Böhm, R. R. Oldehoeft, D. C. Cann, and J. T. Feo. *SISAL Reference Manual* Language Version 2.0. Colorado State University — Lawrence Livermore National Laboratory, 1992.

[Can89] David C. Cann. *Compilation Techniques for High Performance Applicative Computation*. PhD thesis, Colorado State University, 1989. CSU Technical Report CS-89-108.

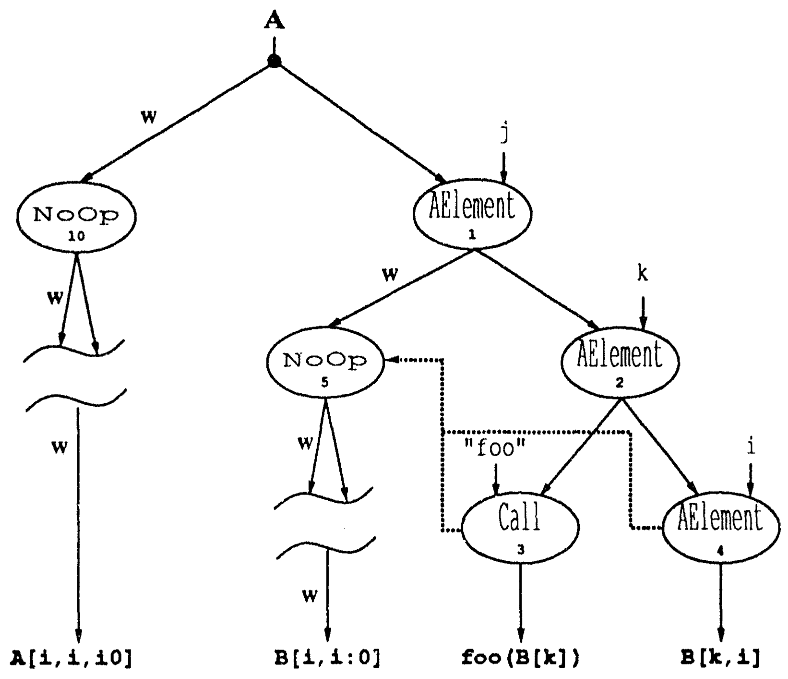


Figure 8: Partial IF2 graph for "A[i,i,i:0], B[i,i:0], foo(B[k]), B[k,i]", recall "B := A[j]"

[CO88] D. C. Cann and R. R. Oldehoeft. Reference count and copy elimination for parallel applicative computing. Technical Report CS-88-129, Department of Computer Science, Colorado State University, November 1988.

[Den80] Jack B. Dennis. Data flow supercomputers. *Computer*, pages 48–56, November 1980.

[FCO90] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10(4), December 1990.

[Fit93] Steven M. Fitzgerald. Removal of redundant artificial dependency edges. Technical Report R-93-001, University of MASS Lowell, 1993.

[Hud86] Paul Hudak. A semantic model of reference counting and its abstraction (Detailed summary). In *Proceedings 1986 ACM Conference on LISP and Functional Programming*, pages 351–363. ACM, August 1986.

[MSA+85] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, Chris Kirkham, Bill Noyce, and Robert Thomas. *SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2.* Lawrence Livermore National Laboratory, M-146 edition, March 1985.

[SS88] Stephen K. Skedzielewski and Rea J. Simpson. A simple method to remove reference counting in applicative programs. Technical Report UCRL-100156, University of California Davis — Lawrence Livermore National Laboratory, November 1988.

[WSYR86] Michael Welcome, Stephen Skedzielewski, Robert Kim Yates, and John Ranelletti. *IF2 — An Applicative Language Intermediate Form with Explicit Memory Management.* University of California — Lawrence Livermore National Laboratory, M-195 edition, November 1986.

Caching in on Sisal: Cache Performance of Sisal vs. Fortran

P. L. Nico

Department of Computer Science
University of California, Davis
Davis, CA 95616
nico@cs.ucdavis.edu

A. Park

Department of Computer Science
University of California, Davis
Davis, CA 95616
park@cs.ucdavis.edu

Abstract

In this paper we investigate the relative cache performance of Sisal *and* Fortran *through trace-driven simulation of representative scientific applications. The range of cache configurations considered corresponds to on-chip caches in current and next-generation microprocessors. We find that in unified caches the performance is equivalent. With split instruction and data caches, performance is still comparable, yet the two languages demonstrate somewhat different tendencies.*

1 Background

With gains in CPU speed far outstripping gains in memory access times, memory performance is rapidly becoming the limiting factor in computer performance. Nearly all current architectures attempt to alleviate this bottleneck by placing small fast cache memories near the processor to take advantage of locality in memory references.[6] Since the difference in memory access time between a cache hit and a cache miss can be in the tens of cycles on a uniprocessor and in the tens to hundreds of cycles on a multiprocessor, cache behavior is of vital interest to those seeking high-performance from their computer systems.

At the speed at which current generation microprocessors run, even the time required to go off-chip has become substantial relative to the clock speed. As a result of this and of advancing VLSI technology, most current-generation microprocessors and virtually all next-generation microprocessors will have on-chip memory caches. Since chip real-estate is a valuable commodity, these caches are relatively small(*e.g.* 8–32Kbytes for the MIPS R4000[3]).

The ready availability of fast, inexpensive microprocessors will make them the building blocks of the next generation of parallel computers[5]. This means, of course, that regardless of how they are interconnected, microprocessor performance in general and memory performance in particular will be of increasing importance in the future.

While FORTRAN is still the workhorse language of the scientific community, the SISAL programming language[4] offers an attractive alternative since it provides the programmer with well-defined, determinate functions while providing the compiler with data dependency information from which it can extract implicit parallelism. Although SISAL's performance competitiveness with FORTRAN has been established on supercomputers such as the CRAY which has no cache[1], questions have been raised as to the relative performance in the cache environment of microprocessor based systems.

We explore some aspects of the relative behavior of FORTRAN and SISAL in the cache environment of current and next-generation microprocessors through simulating their behavior in a variety of caches.

2 The Model

In this study we examine the relative behavior of representative scientific applications in both SISAL and FORTRAN in cache environments similar to those of current and next generation RISC microprocessors.

To properly model potential on-chip cache configurations, we simulate both unified caches and split instruction and data caches. Sizes range from 2Kbytes to 64Kbytes for the instruction caches and 2K to 256K for the data caches. These ranges, though too small for external caches, far exceed the limits of current on-chip cache technology.

Since, by its very nature, a cache must be fast, and an on-chip cache must also be small and simple, most on-chip caches are, and will probably remain, direct

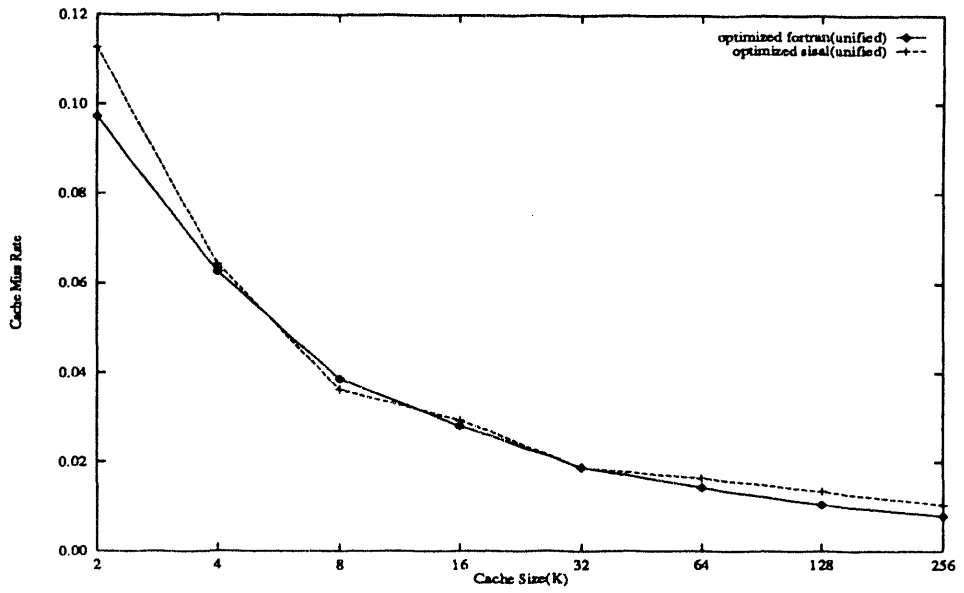


Figure 1: Average miss rate for unified Instruction/Data Caches

mapped.[2] For these reasons, we will concentrate on direct-mapped caches.

2.1 The Programs

The applications used here were taken from those used by David Cann for the performance comparison of SISAL and FORTAN on the CRAY Y-MP [1]. Missing are FFT and the Lawrence Livermore Loops. The FFT program was simply a call to the CRAY library which was clearly not portable for our purposes, and the Livermore Loops did not produce execution traces of sufficient length to properly exercise the caches.

The other applications and their subject areas can be seen in Table 2.1. We chose these applications as a reasonably representative scientific workload with which to exercise the compilers and caches. No attempt was made to re-tune them for the memory architecture of the microprocessor, and no special optimization instructions were given to the compilers.

Application	Subject Material
AMR	Hydrodynamics
BMK11A	Particle Transport
KIN16	Gel Electrophoresis
RICARD	Gel Chromatography
SIMPLE	Hydrodyramics
WEATHER	Weather Modeling

Table 1: The Applications

2.2 The Simulation

The cache performance data were generated by tracing the execution of each program and simulating the behavior of different cache configurations for that sequence of references. The reference streams were captured using the pixie profiling system on a Silicon Graphics MIPS R3000 based computer. All of the execution traces are from sequential execution.

The SISAL and FORTRAN programs were compiled using OSC version 12.9 and the MIPS distribution f77 compiler respectively. In each case, the standard "-O" level of optimization was used. The pixie profiler modifies the executable image so that, while running, it outputs its memory reference stream on an unused file descriptor. After being instrumented, each program was run and its reference stream fed into our cache simulator and analyzed.

Each reference stream was made up of 10 million references after skipping the initial 2 million references to account for transient start-up behavior. Since the largest caches simulated are only 256K, this is sufficiently long to demonstrate steady-state behavior.

The results of these simulations are presented below.

3 Results

The results presented here are those for direct mapped caches with line sizes of 16 bytes. Though

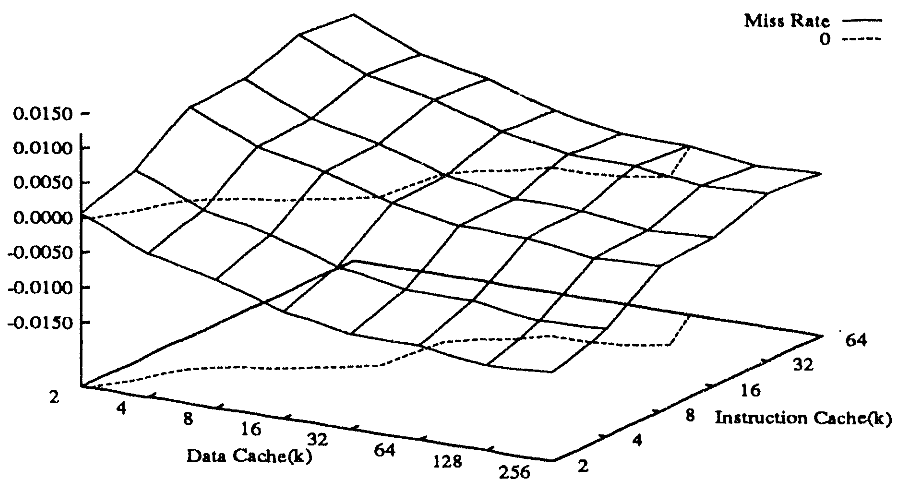


Figure 2: Average overall miss rate difference for split caches (FORTRAN–SISAL)

caches with different associativity and line sizes were simulated and gave different values, the relative behavior remained consistent.

First, Figure 1 gives the average miss rates for unified caches. These results were disappointing not because they were bad, but because nothing interesting happened. As can be seen from the figure, the SISAL and FORTRAN miss rates track each other closely with no clear advantage. Over the sampled space, the average difference is well less that 1%. Still, this shows the two languages to be competitive in the unified cache.

The case where the instruction and data caches are split is somewhat more intersting. In these figures, Fig. 2 through Fig. 4, we show the difference in the miss rates found by subtracting the SISAL value from the FORTRAN value rather than the miss rates themselves. . We find that not only does this reduce the clutter of the graphs, but it also shows what we are really looking for, the *relative* performance. Where the difference is above zero, SISAL is performing better, while when below zero, FORTRAN is better. The contour line shows the crossover point.

Figure 2 shows the overall miss rate differences for the whole range of caches averaged over the six applications. Here the differences begin to develop. For small instruction caches, FORTRAN consistently outperforms SISAL except when there is only 2k of data cache. When the instruction cache size in increased, however, SISAL comes into its own. For a current microprocessor, instruction and data caches of 8k are not unreasonable to consider. This puts it just barely above zero (0.2%). Note that in all cases the difference is not very great. The greatest differences at the top and bottom corners of the graph are 1.1% and −1.2% respectively.

Figures 3 and 4 decompose this graph into instruction and data miss rates respectively. Now it becomes evident why Figure 2 has the form it does. It can be seen from Figure 3 that SISAL requires a greater amount of instruction cache than FORTRAN before its miss rates come down. SISAL only catches up to FORTRAN when the instruction cache reaches 32k.

In the data cache, this tendency is reversed. Here, in Figure 4, it appears that SISAL makes more efficient use of the data cache until the data cache becomes as large as 128k and FORTRAN catches up. In each case, the two level off together when most of the working set is resident in the cache.

4 Analysis

Though we have not determined the cause of these differences, we have examined some possibilities.

Since we see that SISAL and FORTRAN behave so differently in their instruction and data references we considered the possibility that the two could have very different memory demands, as opposed to simply better or worse locality. To see the memory characteristics of the applications, each was run and the instruction and data references counted. Since some of these applications can run for a very long time, they

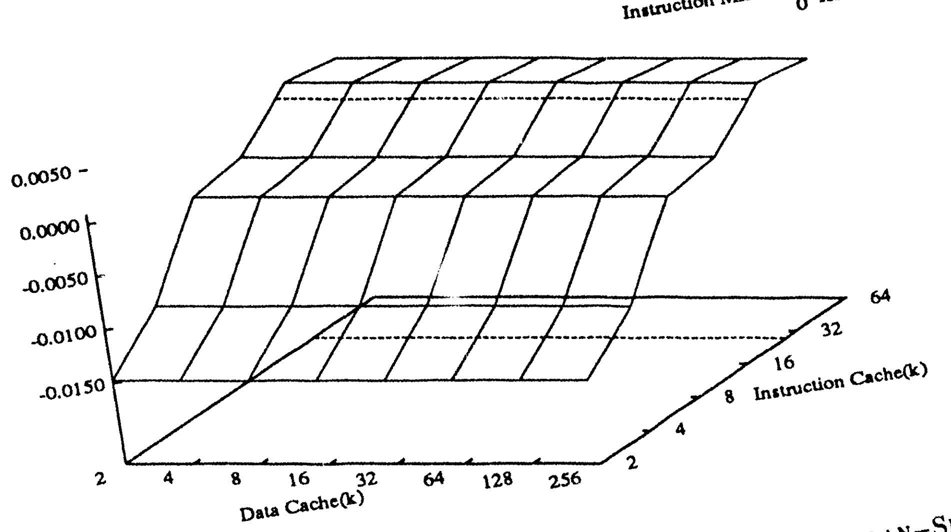


Figure 3: Average Instruction miss rate difference for split caches (FORTRAN–SISAL)

were limited to 10 minutes of processor time. Table 4 shows the memory characteristics for each application. There is variability among the applications, but no clear tendency emerges. It appears that the effect above is indeed based on locality rather than demand.

It is also reasonable to consider whether the behavior above is an artifact of the direct mapped caches with 16 byte lines. Since the compilers did not have any information about the cache architecture, it is possible that one could lay out the instructions or data in such a way that this particular configuration was disadvantageous. In order to counter this possibility we also ran simulations with line sizes of 4 bytes so there could be no prefetching effects, as well as two-way associative caches to limit destructive interference. In each case, the magnitude of the difference between the two languages changed, but the characteristic shape remained the same.

5 Conclusions and Future Work

In this paper, we have attempted to characterize the relative cache performance of FORTRAN and SISAL in environment of current and next-generation microprocessors. We have seen that in all cases simulated the performances have be .. comparable. Neither language dominates the other.

We have also seen that each language does have its areas of strength. SISAL behaves better in the data cache while requiring more instruction cache. FORTRAN behaves in a complementary fashion, running in a smaller instruction cache, but requiring more data space. This information could be useful either in selecting a language for a known cache configuration or in allocating the precious amount of cache real-estate if the application language is known at the time the cache is configured.

There are still many unanswered questions. We have only simulated one version of each of the compilers on one architecture and have only begun to investigate why the two languages behave as they do. In the future we would like to see how they behave on a wider variety of platforms and to further explain their behavior.

References

[1] D. Cann. Retire fortran? a debate rekindled. *Communications of the ACM*, 35(8):81–90, August 1992.

[2] M. D. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(12):25–40, December 1988.

[3] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[4] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkheim, W. Noyce, and

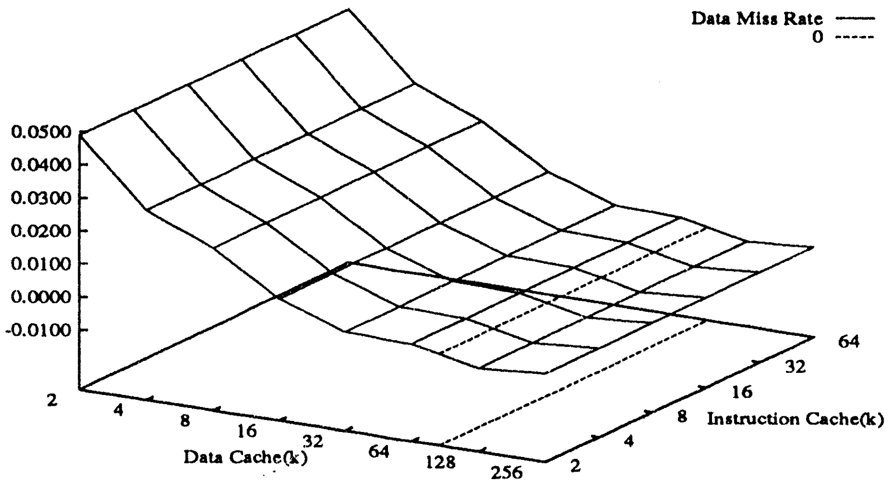


Figure 4: Average Data miss rate difference for split caches (FORTRAN–SISAL)

Program		Fetches	Loads	Stores	mem/inst
amr	SISAL	1,043,831,565	268,213,741	111,393,127	0.363
	FORTRAN	1,136,562,760	256,871,897	80,011,158	0.296
bmk11a	SISAL	854,104,288	212,210,754	127,706,750	0.398
	FORTRAN	941,454,917	244,412,172	141,005,308	0.409
kin16	SISAL	1,309,306,131	480,184,102	69,513,119	0.420
	FORTRAN	965,836,314	408,995,150	52,259,190	0.478
ricard	SISAL	805,401,526	298,707,707	143,946,230	0.550
	FORTRAN	803,445,241	238,072,848	147,489,178	0.480
simple	SISAL	844,093,836	272,734,276	90,036,300	0.430
	FORTRAN	867,464,552	278,596,699	51,846,183	0.381
weather[a]	SISAL	385,490,039	99,840,172	38,605,406	0.359
	FORTRAN	391,735,035	92,457,819	42,783,829	0.345

[a]Ran to completion

Table 2: Memory characteristics of programs (10 minutes running time)

R. Thomas. SISAL: *Streams and iteration in a single assignment language: Reference manual version 1.2.* Lawrence Livermore National Laboratory, Livermore, CA, manual M-146, rev. 1 edition, March 1985.

[5] J. Rattner. Supercomputing's destiny: the microprocessor. *Supercomputing Review*, June 1989.

[6] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

FFT Algorithms on a Shared-Memory Multiprocessor

A.L. Cricenti and G.K. Egan.*

Abstract

This paper deals with the coding of some FFT algorithms in the functional language SISAL, to exploit the available concurrency, on a shared memory multiprocessor (Encore Multimax). Run times and speed-up are presented for two conventional array based and two pipeline stream based FFT algorithms. The performance of the stream based algorithms is compared with that of the array based algorithms.

Introduction

The Discrete Fourier Transform (DFT), and other related transforms, are of key importance in the field of digital signal processing. Calculation of the DFT from its definition is computationally expensive requiring $O(N^2)$ multiplications and a similar number of additions. However much effort has been put into developing fast methods of calculating the DFT, since efficient calculation of the DFT makes much of discrete signal processing possible. Several good algorithms now exist for the efficient calculation of the DFT, these algorithms are generally known as Fast Fourier Transforms (FFT).

To speed up the calculation of the DFT further, either faster algorithms need to be developed, or any concurrency in the algorithm be exploited. Some researchers have been looking into the parallel implementation of the FFT. Pease [Pea68] pioneered work in this area by suggesting a parallel FFT algorithm. Norton and Silberger [NS87] have presented results for a FORTRAN implementation of the Cooley-Tuckey FFT on a shared memory architecture (IBM-RP3 machine) while Cvetanovic [Cve87] presents methods for performance analysis of two FFT algorithms on shared memory machines. Adams et. al.[ABC*91] present results for parallel FFT algorithms on a connection machine and a Cray 2. Recently the performance of some FFT algorithms coded in SISAL have been presented by Cann[Can91] and Bollman[BSS92].

This paper deals with the coding of some FFT algorithms; and is also concerned with the use of the SISAL data type stream to implement pipeline FFT algorithms. The performance of these pipeline algorithms is compared with that of the standard ones.

Algorithm derivation

Calculation of the Fourier Transform of a signal involves the evaluation of the following integral:

$$F(\omega) = \int_{-\infty}^{\infty} f(t) * e^{-j\omega t}\, dt \qquad (1)$$

In many cases the signal of interest is not a continuous time signal, but a discrete time signal. Discrete time signals are therefore sequences of numbers and therefore lend themselves for implementation and analysis on digital computers.

In the case of discrete signals the integral above becomes a summation:

$$F(n) = \sum_{k=0}^{N-1} f(k) * e^{-j2\pi kn/N} \qquad n = 0,1...N-1 \qquad (2)$$

The above summation is called the Discrete Fourier Transform (DFT).

Calculation of the DFT is computationally expensive requiring order N^2 multiplications and a similar number of additions.

Many approaches for improving the computational efficiency of the DFT, rely on the properties of the $e^{-j2\pi kn/N}$ term, which is periodic and symmetric. Exploitation of these properties has led to several fast algorithms for evaluating the DFT.

Although there are several FFT algorithms, most are based on the principle of decomposing the computation into successively smaller DFT computations, this method was re-discovered by Cooley and Tuckey [CT65].

One common FFT algorithm is called the Cooley-Tuckey Radix Two Fast Fourier Transform. This algorithm is based on the successive partitioning of the data sequence into even and odd indices (note N must be a power of two hence the name Radix two).

The Radix Two algorithm can be derived by either separating the input sequence f(k) into two N/2 point sequences, Decimation in Time (DIT), or by dividing the output sequence F(n) into two N/2 point sequences, Decimation in Frequency (DIF). The number of operations in each algorithm is the same, however the DIF algorithm accepts data in natural order, and outputs the data in scrambled order, while the DIT algorithm accepts data in scrambled order, and outputs it in natural order. These features can be advantageous in convolution or correlation, as unscrambling (bit reversing) can be avoided by using an DIF algorithm to transform to the discrete frequency domain and a DIT to transform back to the discrete time domain.

As previously stated the Radix 2 DIT algorithm, for $N=2^m$ can be obtained by splitting the input sequence into two N/2 sequences consisting of the even elements and odd elements of the input sequence. The Radix 2 Cooley-Tuckey algorithm can be conveniently expressed in tensor notation [BSS92] as:

$$F_{2^k} = \prod_{i=1}^{k} \{(I_{2^{k-i}} \otimes F_2 \otimes I_{2^{i-1}})(I_{2^{k-i}} \otimes T^{2^i})\} R(2^k) \qquad (3)$$

where: $\otimes$ denotes the tensor product.

$R(2^k)$ is the bit reversal permutation.

T^{2^i} represents the twiddle factors.

$F_2 \otimes I_{2^{i-1}}$ is a two point transform (butterfly).

Alternatively the FFT algorithm can be conveniently expressed as a signal flow graph as shown in figure 1. The signal flow graph has an advantage in that it shows up possible concurrency, and aids in the coding of the algorithm This algorithm is termed fast since it requires order $N\log_2 N$ multiplies rather than order N^2 multiplies to calculate the DFT of a sequence.

As evident from the signal flow graph this FFT algorithm has a high level of symmetry as well as potential concurrency, this was recognised early by Pease [Pea68] and Gold [GB73]. The concurrency can be observed by noting that the input data to each butterfly in a stage depend only on the previous butterfly or the input. Since there are no data dependencies for each butterfly in a particular stage, the calculation of all these butterflies could proceed concurrently. Although the signal flow diagram of the FFT is quite elegant, coding the algorithm to exploit the available concurrency is not as simple as it would seem. One must partition the FFT graph into segments and assign each of these to a processor. Proper synchronisation must be assured so that the data at the end of each stage is valid.

Sequential algorithm

A FORTRAN program to implement of the abov FFT, due to Cooley, Lewis and Welch, adapted fron [RG75], is shown in figure 2.

The program is divided into two sections; the firs part is devoted to performing the bit reversal on th input sequence, such that it is in the order required fo the FFT. Note bit reversal is not essential in some case such as computing a convolution, thus it will not b considered further. The second part of the program i concerned with the computation of the FFT. This pa consists of three nested loops, the most outer loop step through each stage of the signal flow graph, anothe loop performs the indexing on the powers of W a required by the butterflies, while the third loop keep track of which butterfly calculation is being performed.

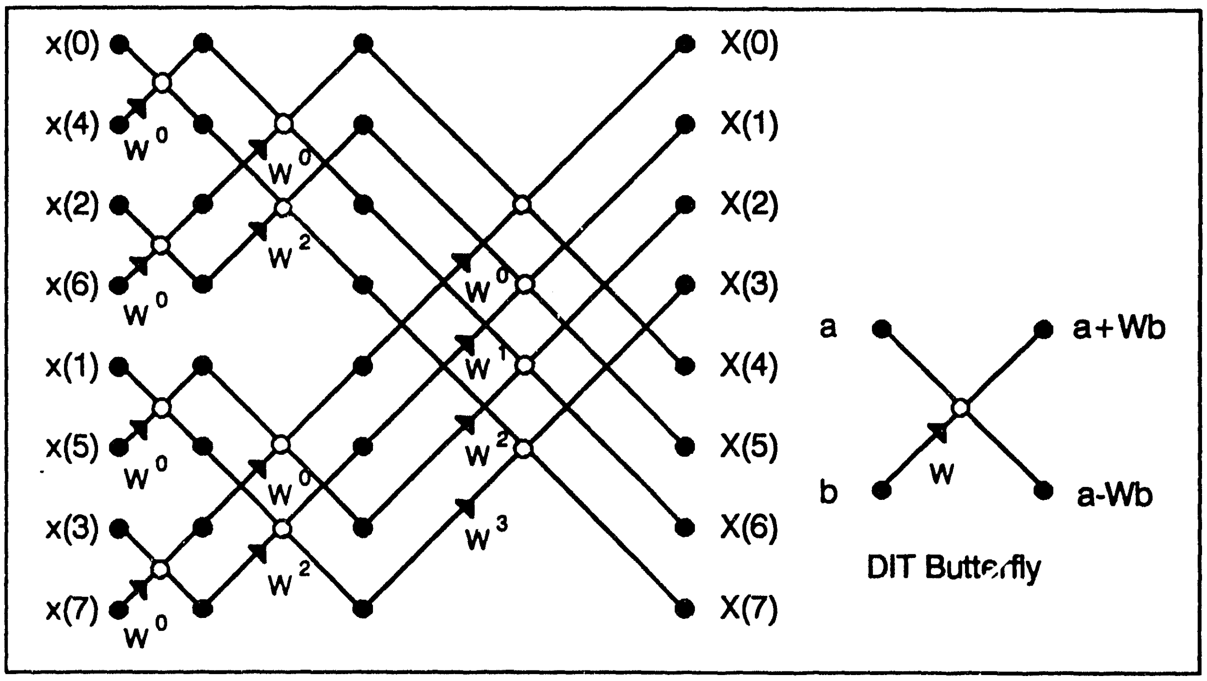


Figure 1 DIT signal flow graph for 8 point FFT.

```
      SUBROUTINE FFT(A,M,N)
      COMPLEX A(N),U,W,T
      N=2**M
      NV2=N/2
      NM1=N-1
      J=1
      DO 7 I=1,NM1                              Bit Reverse input
       IF(I.GE.J) GO TO 5
       T =A(J)
       A(J)=A(I)
       A(I)=T
5      K=NV2
6      IF(K.GE.J) GO TO 7
       J=J-K
       K=K/2
       GO TO 6
7     J=J+K
      PI=3.141592653589793
      DO 20 L=1,M                               For each stage
       LE=2**L
       LE1=LE/2
       U=(1.0,0.)
       W=CMPLX(COS(PI/LE1),SIN(PI/LE1))         Calculate the new e^-j2πkn/N term
       DO 20 J=1,LE1                            Do each butterfly
        DO 10 I=J,N,LE
         IP=I+LE1
         T=A(IP)*U                              Computation of the butterflies
         A(IP)=A(I)-T
10       A(I)=A(I)+T
20    U=U*W                                     Update the e^-j2πkn/N term
      RETURN
      END
```

Figure 2 FORTRAN Radix 2 FFT Decimation in Time Algorithm.

The program of figure 2 if compiled to run on a shared memory multiprocessor, such as the Encore, shows no speed-up because of the way the "Twiddle Factors", ($e^{-j2\pi kn/N}$) terms are calculated, that is an initial cosine and sine term is computed, then on each iteration of the outer loop (label 20) the twiddle (W term) is updated by a recursion relation, this method of calculation is economical in terms of machine instructions, but it makes the program sequential. Since the new W value depends on the its value on the previous iteration, a data dependency exists which is not evident in the signal flow graph. The translation from FORTRAN to SISAL is quite straight forward since most of the FORTRAN control structures map directly to SISAL. One major problem in the translation is that SISAL lacks a complex number type and complex operations. This deficiency was overcome by defining a set of functions for handling complex numbers, and declaring a "type complex" as a:
Record[Realp, Imaginaryp : Double_Real].

Although this record construct overcomes the problem it leads to clumsy programming, since all operations involving this data type must be performed explicitly.

A pre declared type complex is essential for signal processing as complex data is often manipulated. The need for the complex type has previously been noted by Chang [CED90] and will be implemented in SISAL 2 [COBGF].

As expected, the SISAL program shows no speed-up. The main outer loop is sequential as can be seen from the signal flow graph, however the computation of the butterflies is also sequential, while the signal flow graph suggests that this process could be parallel. There are two reasons that make this process sequential. The first is the way the W terms are calculated. To remove this loop iteration data dependency all the W terms can be precalculated and stored in an array for access by the program. The second reason for the sequential behaviour of the loop is in the way SISAL exploits parallelism from loops. SISAL does not allow one to express a sequential process in a parallel form, since it cannot be written as a *product for* form loop. One expects that the loop which would step through each butterfly calculation can be expressed in a parallel fashion, since there are no data dependencies between butterflies in the same stage.

When the data operated upon is stored as an array, then SISAL requires that the elements of that array are processed in order, if the product form for loop is to be used. Thus SISAL can only slice loops such as the following array build statement

```
A:=for i in 1,10
    returns array of i
    end for
```

This loop is considered concurrent by OSC since the elements of A are created in order, ie A[1],A[2]...A[10], thus the concurrency of this loop can be exploited by loop slicing. Note OSC achieves concurrency by loop slicing, on a shared memory multiprocessor.

The Cooley-Tuckey FFT algorithm is not easily expressed in SISAL in a way that achieves useful speed-up. The problem with this algorithm is that the elements of the array output from each stage are not produced in order. An alternative algorithm is required which can be expressed in SISAL in a parallel form.

Constant geometry algorithm

As seen from the signal flow graph the output vector of each stage of the FFT is not produced in order, but the elements of the vector come out in a different order for each stage. This implies that the sequential SISAL non product *for* loop must be used. To overcome the limitation of arrays in SISAL being built in a strict sense for parallel loop constructs, the FFT algorithm must be modified so that the elements of the output of each FFT stage, are produced in order. The reason for the elements being produced out of order is because the Cooley-Tuckey FFT program is based on a so called "In-place algorithm". This means that each butterfly output is put back into the index where it came from. This is desirable since it means that only one array is required to implement the program. In a SISAL implementation this is not an advantage since a copy of the old array may exist, (*old*) when using a non product *for* loop. If the restriction of in-place computation can be relaxed then the indexing can be kept constant from stage to stage and in order. This allows the inner loop, which performs each butterfly for a given stage, to be expressed in the product *for* form loop. This algorithm is termed "Constant Geometry" [RG175] and the signal flow diagram is shown in figure 3. Note the signal flow graph below represents a DIF algorithm, the DIT version would have the powers of W in the bottom wing of the input to the butterfly. Since each stage of the constant geometry algorithm is the same, refer to signal flow graph, the program is simple to express.

Pipeline algorithms

The FFT algorithms presented thus far rely on the data being fed into the algorithm in a parallel fashion, that is as an array. In practice the data would arise from sampling a signal at discrete time intervals. In this case the data would arrive in a sequential fashion.

Inspection of the signal flow graphs shows that the butterflies in a particular stage could be processed in any order. In fact it is not necessary to fully complete a stage before commencing the next stage, subject to the availability of the appropriate data for the next stage.

When using arrays, data cannot, in general, be output from each stage of the algorithm until each stage has been completed, that is all of the elements of each array must be computed. Arrays therefore restrict the amount of exploitable concurrency in the FFT algorithm.

In some situations, it is desirable to pass the data in a serial fashion to the FFT and have it output in a serial fashion. This scheme is attractive since it is possible have data output at the sampling rate, once the FFT has been primed with data. An algorithm which achieves this is a pipeline algorithm. Several pipeline Fast Fourier Transforms have been presented in the literature, some of these however are usually implemented with special purpose hardware [GB73][GW70][SJ90].

A pipeline algorithm [GB73] can be derived by considering the FFT signal flow graph of figure 4, this is the graph of an 8 point DIF algorithm, note input data are normally ordered, output data are bit reverse ordered. The DIF version is chosen as it is assumed that the input data are in natural sequential order.

It can be seen from the graph that to co ute the X(0) and X(4) output points only three butterflies need to be evaluated, as shown in figure 5. A similar situation exists for the other output points.

The input to the first butterfly stage is:

$$x(t) = x(k) \qquad k=0...N/2-1 \qquad (4)$$
$$x(b) = x(k+N/2) \qquad k=0...N/2-1$$

that is two points separated by N/2 where N is the FFT length. Therefore the first N/2 input samples to the pipeline are routed to the top arm of the butterfly while the next N/2 samples to the bottom. A delay of N/2 is used in the top arm of the butterfly to ensure that the data at the butterfly are synchronised. The appropriate W^p must be used at the butterfly output. The data leaves the butterfly in parallel pairs. The input to the second butterfly stage is:

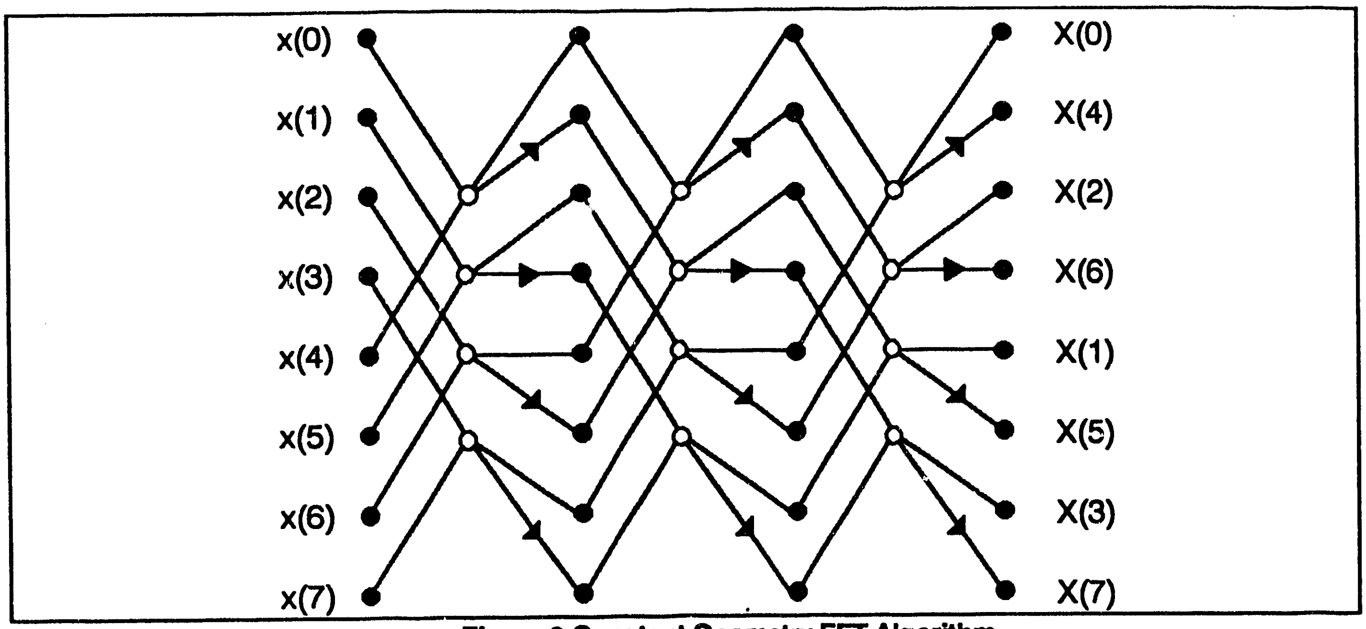


Figure 3 Constant Geometry FFT Algorithm.

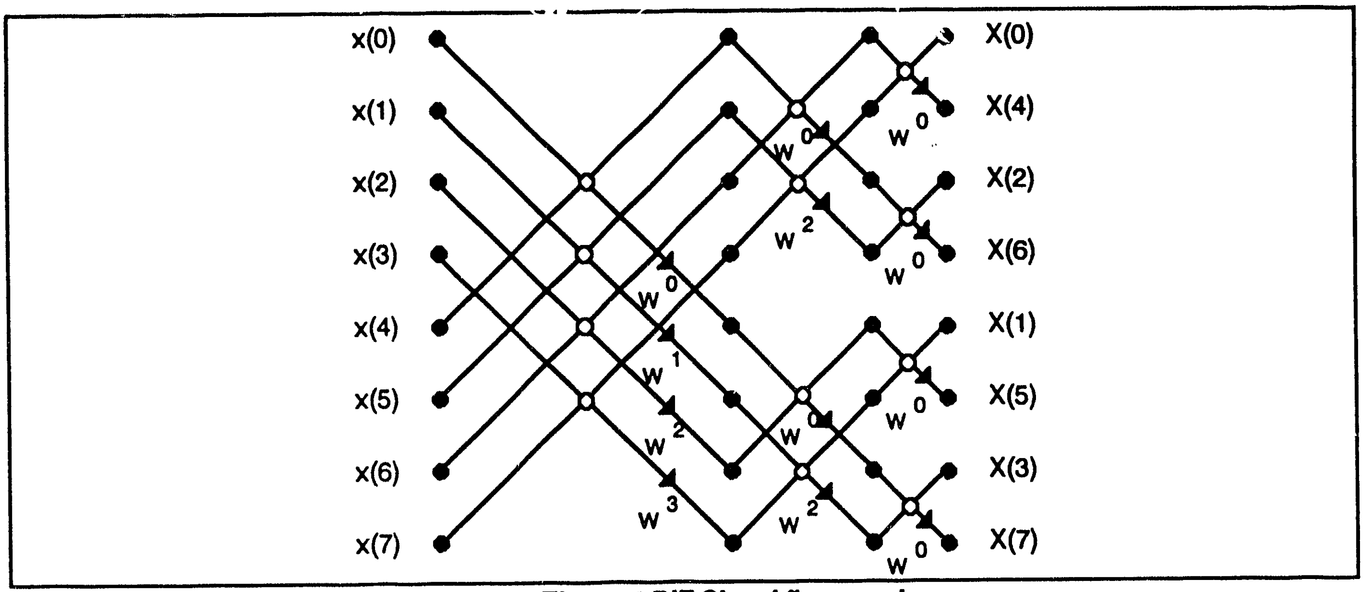


Figure 4 DIF Signal flow graph.

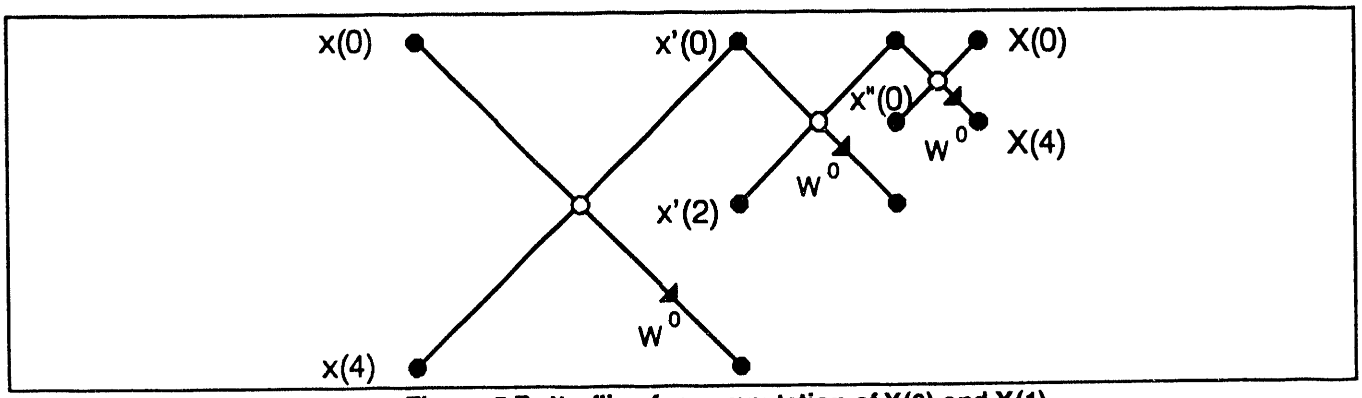


Figure 5 Butterflies for computation of X(0) and X(1).

x'(t) and x'(t+N/4)   t=0...N/4-1   (5)
x'(b) and x'(b+N/4)   b=0...N/4-1   (6)

where x'(t) is the output of the top arm of the first butterfly and x'(b) is the output of the bottom arm of the first butterfly. Again the appropriate W^p must be applied. The argument can be continued for the third and following stages of the pipeline and the results this shown in figure 6.

The elements of the data sequence must be routed to the appropriate arm of each butterfly, this is achieved by the switches and delay lines in the pipeline. Each switch in the pipeline switches at twice the frequency of its predecessor, and the delay lengths in a given stage are half that of the previous stage. The first switching block works as follows:

-The data samples are routed straight through for the first N/4 elements.

-The samples are crossed over for the next N/4 samples.

SISAL has an appropriate data type for pipeline algorithms, that is the type STREAM [Can89][MSA*85]. A stream is a sequence of values of uniform type that allows pipeline concurrency to be expressed directly by their use. Streams differ from arrays in that the elements of the stream can only be accessed in sequence, no subscripting or random access is possible. This form of access allows the use of elements from the stream without having to produce the complete stream as would be required with an array. By definition SISAL streams require a non-strict implementation. A problem occurs with the use of streams in OSC because OSC implements streams strictly [CO]. As a consequence of this implementation, pipeline concurrency is not exploited. In addition parallel loops are difficult to write with streams because only the first element of the stream can be accessed.

The switch blocks of the pipeline are implemented by a function *stream_switch*. This function is complicated since the switching period is stage dependent. Another problem arises because the two streams that enter the function *stream_switch* are not processed simultaneously, that is the part of top stream is processed before the bottom stream. The delays in the pipeline are meant to cope with this problem. The software implementation uses the SISAL ***when & unless*** masking clauses to filter out the unwanted values from the ***returns*** part of the ***for*** loop.

The pipeline algorithm could be made simpler to express if the switching block could be simplified. To achieve this aim the switching should be made independent of the stage of the pipeline. Again a constant geometry algorithm could be used to achieve this. A pipeline implementation of the constant geometry algorithm is shown in figure 7.

It can be seen that each stage of the pipeline is identical. This feature makes the expression of this pipeline algorithm straight forward.

The switch blocks for this algorithm are very simple since they are composed of two switches. The first switch switches the sample rate, while the second switch switches at a rate of N/2 samples.

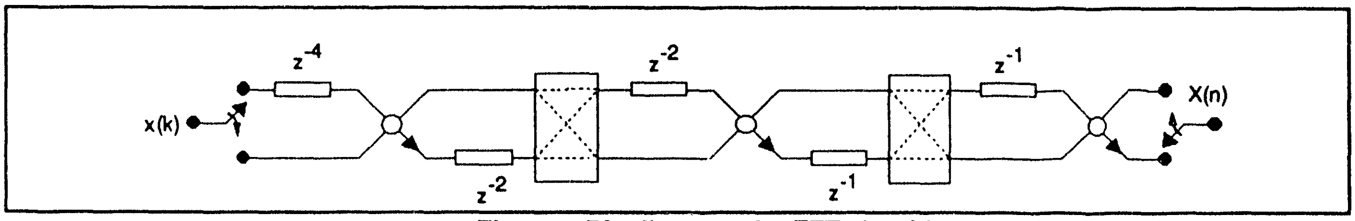


Figure 6 Pipeline 8 point FFT algorithm.

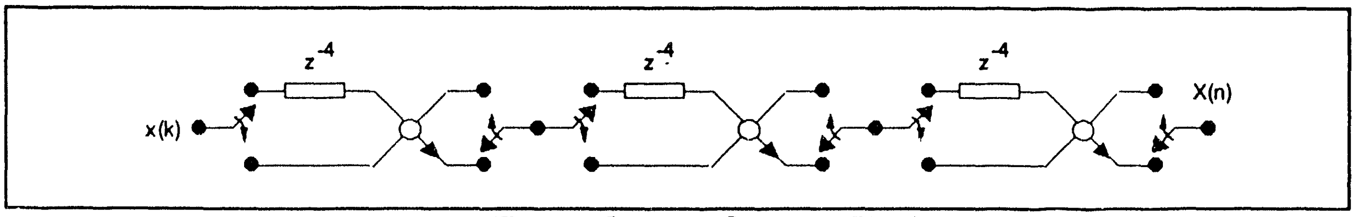


Figure 7 Constant Geometry Pipeline.

Results

The various algorithms presented in the previous section were all run on an Encore shared memory multiprocessor, using OSC as the compiler. The OSC -h500 switch was used to ensure that the compiler considered all concurrent loops for slicing. The value of 500 was arrived at by trial and error. Run-times were obtained using the SPEED-UPS routine from the OSC library. All times are in seconds and the longest wall time was chosen when multiple processors were used. Generally in cases where there was a significant difference between the wall time and the CPU time the results were discarded (CPU utilisation <95%). When multiple processor run-times showed a significant difference for each processor the results were also discarded.

All input data were generated internally by each program, therefore the run-times reflect this. Output data from the programs was suppressed by using the -z switch.

Speed-up is defined as T_1 proc / T_n procs.

Various FFT sizes were used in the study to check the performance of the algorithms against model size.

Sequential algorithm

Tables 1 and 2 present the results for run-time and speed up for the Sequential FFT algorithm.

As can be seen from table 2, the Sequential FFT algorithm shows no significant speed-up. This is to be expected since the algorithm is sequential.

Constant geometry algorithm

Tables 3 and 4 present the results for run-time and speed up for the Constant Geometry FFT algorithm.

The constant geometry algorithm achieves good speed-up, refer to table 4 and figure 8. The single processor run-times are longer than the sequential algorithm (table 1), as is to be expected because of the less efficient calculation of the indexes of the W factors used in the butterflies. However because of the speed-up obtained, this code becomes faster than the sequential algorithm.

The droop in the speed-up graph for 10 -16 processors is due to other processes competing for resources. The speed-up improves with the size of the FFT and high efficiencies are obtained for N > 4096.

Processors	1024	2048	4096	8192	16384	32768	65536
1	0.6	1.3	2.78	6.240	13.140	28.620	59.640
2	0.58	1.28	2.74	6.140	13.000	28.400	59.140
3	0.58	1.28	2.74	6.120	12.960	28.380	59.180
4	0.6	1.3	2.78	6.100	12.980	28.420	59.040
5	0.58	1.3	2.76	6.120	13.000	28.420	59.480
6	0.58	1.26	2.78	6.140	13.020	28.680	59.160

Table 1 Run-time vs FFT length for the Sequential FFT.

Processors	1024	2048	4096	8192	16384	32768	65536
1	1	1	1	1	1	1	1
2	1.03	1.02	1.01	1.02	1.01	1.01	1.01
3	1.03	1.02	1.01	1.02	1.01	1.01	1.01
4	1.00	1.00	1.00	1.02	1.01	1.01	1.01
5	1.03	1.00	1.01	1.02	1.01	1.01	1.00
6	1.03	1.03	1.00	1.02	1.01	1.00	1.01

Table 2 Speed-up vs FFT length for the Sequential FFT.

Processors	1024	2048	4096	8192	16384	32768	65536
1	0.82	1.8	3.9	8.700	18.000	38.100	80.860
2	0.44	0.96	2.02	4.480	9.280	19.680	41.540
3	0.32	0.64	1.36	2.940	6.200	13.140	27.820
4	0.24	0.5	1.06	2.300	4.680	9.900	20.940
5	0.2	0.4	0.86	1.800	3.780	8.020	16.860
6	0.18	0.36	0.74	1.520	3.180	6.720	14.160
7	0.16	0.32	0.64	1.340	2.780	5.800	12.560
8	0.14	0.28	0.600	1.200	2.440	5.160	11.140
16	0.1	0.2	0.440	0.740	1.440	2.920	6.200

Table 3 Run-time vs FFT length for the Constant Geometry FFT.

Processors	1024	2048	4096	8192	16384	32768	65536
1	1	1	1	1	1	1	1
2	1.86	1.88	1.93	1.94	1.94	1.94	1.95
3	2.56	2.81	2.87	2.96	2.90	2.90	2.91
4	3.42	3.60	3.68	3.78	3.85	3.85	3.86
5	4.10	4.50	4.53	4.83	4.76	4.75	4.80
6	4.56	5.00	5.27	5.72	5.66	5.67	5.71
7	5.13	5.63	6.09	6.49	6.47	6.57	6.44
8	5.86	6.43	6.50	7.25	7.38	7.38	7.26
16	8.20	9.00	8.86	11.76	12.50	13.05	13.04

Table 4 Speed-up vs FFT length for the Constant Geometry FFT.

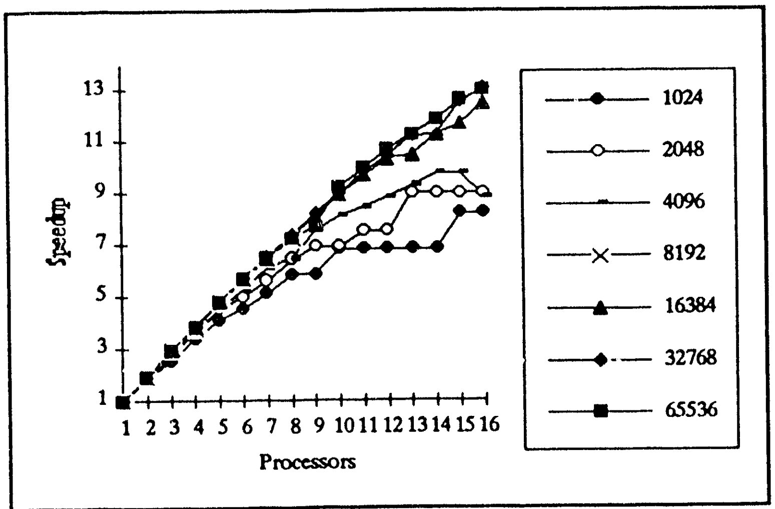


Figure 8 Speed-up vs FFT length for the Constant Geometry FFT.

Pipeline algorithms

Tables 5 and 6 present the results for run-time and speed up for the Pipeline FFT algorithm.

The pipeline algorithm shows poor speed-up, mainly because the implementation of this algorithm introduces a large amount of array copying, which severely affects its performance. The OSC compiler warns when array copying may occur.

Tables 7 and 8 present the results for run-time and speed up for the Constant Geometry Pipeline FFT algorithm.

The constant geometry pipeline code performs better than the previous pipeline algorithm, but still suffers from array copying.

Processors	1024	2048	4096	8192	16384	32768	65536
1	0.45	1.22	3.11	9.35	25.44	63.06	181.83
2	0.44	1.20	2.51	7.36	20.79	57.54	160.45
3	0.43	1.16	2.45	7.10	20.10	53.82	153.59
4	0.44	1.05	2.57	7.14	19.54	52.03	150.77
5	0.43	1.06	2.51	7.09	19.23	52.26	150.02
6	0.43	1.10	2.46	7.16	22.71	56.14	159.57

Table 5 Run-time vs FFT length for the Pipeline FFT.

Processors	1024	2048	4096	8192	16384	32768	65536
1	1	1	1	1	1	1	1
2	1.04	1.02	1.24	1.27	1.22	1.10	1.13
3	1.07	1.05	1.27	1.32	1.27	1.17	1.18
4	1.04	1.17	1.21	1.31	1.30	1.21	1.21
5	1.06	1.15	1.24	1.32	1.32	1.21	1.21
6	1.05	1.11	1.26	1.31	1.12	1.12	1.14

Table 6 Speed-up vs FFT length for the Pipeline FFT.

Processors	1024	2048	4096	8192	16384	32768	65536
1	1.16	2.28	6.1	14.600	35.800	87.940	214.400
2	0.76	1.88	3.96	10.060	25.720	63.540	170.820
3	0.6	1.56	3.28	8.580	22.420	56.500	156.300
4	0.54	1.44	2.96	7.820	20.780	53.340	149.640
5	0.5	1.34	2.74	7.420	19.820	52.040	145.820
6	0.46	1.26	2.6	7.140	19.440	50.800	144.900

Table 7 Run-time vs FFT length for the Constant Geometry Pipeline FFT.

Processors	1024	2048	4096	8192	16384	32768	65536
1	1	1	1	1	1	1	1
2	1.34	1.34	1.45	1.44	1.36	1.32	1.30
3	1.43	1.50	1.61	1.63	1.50	1.46	1.39
4	1.58	1.60	1.73	1.82	1.54	1.48	1.39
5	1.66	1.74	1.88	1.81	1.59	1.53	1.43
6	1.60	1.86	1.84	1.86	1.66	1.55	1.46

Table 8 Speed-up vs FFT length for the Constant Geometry Pipeline FFT.

Both pipeline algorithms require a function to implement the switch that switches with a period of N/2 samples. This function must separate the input stream into two streams, one containing the first N/2 samples the other the rest of the stream. An obvious way of expressing this switching function in SISAL is as:

```
        for aelement in input at i
        returns  stream of aelement when i<=length
                 stream of aelement when i>length
        end for
```

Figure 9 'Product form for loop' for switch.

This approach has two problems when OSC is used to compile it. The first problem is that OSC will not slice this loop as the compiler considers it sequential because of the *when* clause. Thus using the product for form loop is not an advantage.

The second problem with this approach is that this loop introduces copying. To overcome some of the copying, the loop can be expressed as the non product form for loop as shown in figure 10. This implementation of the loop is much faster as some array copying is eliminated. Note this was verified using the OSC *-time* switch.

```
    for initial
      i: = 1;
      bottom: = input
    until i > length repeat
      i: = old i + 1;
      bottom: = stream_rest(old bottom);
    returns  stream of stream_first(bottom) unless i > length
% This introduces copying
          value of bottom
    end for
```

Figure 10 'Non product form for loop' for switch

The array copying which results from this loop is still significant both in terms of run-time and speed-up performance. This is particularly noticeable for large FFT lengths. If the *unless* clause is removed from the above loop, the execution time is reduced considerably, and the speed-up performance is improved, refer to table 10. Removal of the *unless* clause, is not practical since the code produces incorrect results. The speed-up improvement implies that the array copying due to the unless clause is of a sequential nature.

Processors	1024	16384	65536
1	1.04	22.52	101.4
2	0.60	12.70	57.22
4	0.38	7.70	34.84
6	0.32	6.58	27.66

Table 9 Run-time vs FFT length for the Constant Geometry Pipeline FFT without *unless*.

Processors	1024	16384	65536
1	1	1	1
2	1.73	1.77	1.77
4	2.74	2.92	2.91
6	3.25	3.42	3.67

Table 10 Speed-up vs FFT length for the Constant Geometry Pipeline FFT without *unless*.

The shared memory machine and the OSC code do not exploit the available pipeline concurrency. Therefore to better compare the pipeline algorithm with the array based (non pipeline) algorithms, it was coded using arrays rather than streams.

Performance of the array based pipeline algorithm is much better than the stream version since the array copying is eliminated. By eliminating the array copying one can then express the function **switch** as a parallel for construct, thereby improving the speed-up performance.

Run-time and speed-up results, for the array based pipeline algorithm, are given in tables 11 and 12. The results show that the use of arrays in OSC gives better performance than the use of streams. The performance of this pipeline algorithm is similar to the constant geometry algorithm, refer to tables 3 and 4.

Note a dataflow machine would possibly achieve a better result for the stream based pipeline algorithms as pipeline concurrency could be exploited assuming a non strict implementation of the streams. However at the time of writing a dataflow machine that implements SISAL streams was not available. It is expected that the CSIRAC II simulator will eventually support the stream type.

Processors	1024	2048	4096	8192	16384	32768	65536
1	0.9	1.94	4.22	9.080	19.280	41.360	86.680
2	0.48	1	2.16	4.600	9.820	20.980	43.960
4	0.26	0.54	1.12	2.380	5.000	11.100	22.520
8	0.18	0.32	0.680	1.340	2.820	5.920	12.380
16	0.16	0.26	0.520	1.000	1.960	4.060	8.540

Table 11 Run-time vs FFT length for the Constant Geometry (Array) Pipeline FFT.

Processors	1024	2048	4096	8192	16384	32768	65536
1	1	1	1	1	[illegible]	1	1
2	1.88	1.94	1.95	1.97	1.96	1.97	1.97
4	3.46	3.59	3.77	3.82	3.86	3.73	3.85
8	5.00	6.06	6.21	6.78	6.84	6.99	7.00
16	5.63	7.46	8.12	9.08	9.84	10.19	10.15

Table 12 Speed-up vs FFT length for the Constant Geometry (Array) Pipeline FFT.

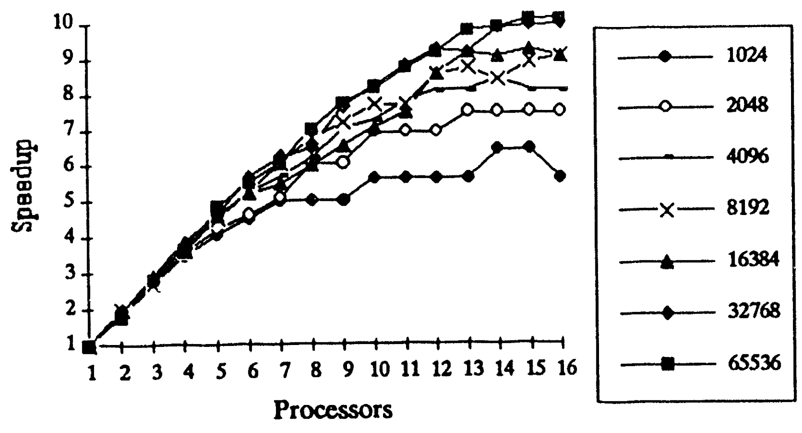


Figure 11 Speed-up vs FFT length for the Constant Geometry Pipeline (Array) FFT.

Conclusions

The paper has presented several FFT algorithms which have been coded in SISAL. The Cooley-Tuckey FFT algorithm is difficult to code in SISAL in a way that exploits the concurrency which seems to be present in the algorithm. This is due to the order in which the output elements of each butterfly stage are produced. Useful speed-up has been demonstrated, for the constant geometry FFT algorithm, without significant programming effort. However SISAL's lack of a complex number type and operations, leads to clumsy and long code.

For signal processing the data type stream is of key importance as the data in this field occurs naturally as a stream. Here SISAL could have an advantage over other languages. However the strict implementation of streams is inefficient making the writing of code difficult if copying is to be avoided. The use of strict streams limits the possibility of real time signal processing as the whole of the input data, must be in memory before processing can commence.

References

[ABC*91] Adams D.E., Bronson E.C., Casavant T.L., Jamieson L.H., Kamin R.A., "Experiments with Parallel Fast Fourier Transforms", Parallel Algorithms and Architectures for DSP Applications, pp 49-75 Kluwer Academic Publishers 1991.

[BSS92] Bollman D., Sanmiguel F., Seguel J, "Implementing FFT's in SISAL" Proceedings of the Second Sisal Users' Conference. pp 59-65, December 1992.

[Can89] Cann, D.C. "Compilation Techniques for High Performance Applicative Computation" Technical Report CS-89-108, Colorado State University, pp 12-13, May, 1989.

[Can91] Cann D.C. "Retire Fortran? A Debate Rekindled", Technical Report UCRL-JC-107018, Lawrence Livermore National Laboratory, April 1991.

[CED90] Chang P. and Egan G.K., "An Implementation of a Numerical Weather Prediction Model in SISAL" Technical Report 31-017, Laboratory for Concurrent Computing Systems, Swinburne Institute of Technology December. 1990.

[CO] Cann D., Oldehoeft R.R., "A guide to the Optimising SISAL Compiler" p32.

[COBGF] Cann D., Oldehoeft R.R., Bohm A.P.W., Grit D., Feo J., " SISAL Reference Manual, Language Version 2.0", Colorado State University and Lawrence Livermore National Laboratory, 1990.

[CT65] Cooley J.W., Tukey J.W., "An Algorithm for the Machine Calculation of Complex Fourier Series", Math. Comput., Vol. 19, pp 297-301 April 1965.

[Cve87] Cvetanovic Z., "Performance Analysis of the FFT Algorithm on a Shared-Memory Parallel Architecture", IBM J. Res. Develop. Vol. 31 No. 4, pp 435-451 July 1987.

[GB73] Gold B., Bially T., "Parallelism in Fast Fourier Transform Hardware", IEEE Trans. Audio Electroacoust., Vol. AU-21 No 1, pp 5-16 February 1973.

[GW70] Groginski H.L., Works G.A., "A Pipeline Fast Fourier Transform", IEEE Trans. Comput. Vol. C-19, pp 1015-1019 November 1970.

[MSA*85] McGraw J., Skedzielewski S., Allen S., Oldehoeft R., Glauert J., Kirkham C., Noyce B. and Thomas R., "SISAL: Streams and Iteration In a Single Assignment Language, Language Reference Manual, Version 1.2", Lawrence Livermore National Laboratory, March 1985.

[NS87] Norton V.A., Silberger A., "Parallelization and Performance Prediction of the Cooley-Tuckey Algorithm for Shared Memory Architectures", IEEE Trans. Comput. Vol. C-36, No 5, pp 581-591 May 1987.

[Pea68] Pease M.C., "An Adaptation of the Fast Fourier Transform for Parallel Processing", J. Ass. Comput. Mach., Vol 15, pp 252-264 April 1968.

[RG75] Rabiner L.R., Gold B., "Theory and Application of Digital Signal Processing", Prentice-Hall 1975 p 367.

[RG175] Ibid, pp 575-576.

[SJ90] Sapiecha K., Jaroki R., "Modular Architecture for High Performance Implementation of the FFT Algorithm", IEEE Trans. Comput. Vol. C-39, No 12, pp 1464-1468 December 1990.

Acknowledgements

The authors thank the members of the Laboratory for Concurrent Computing Systems at the Swinburne University of Technology, for their assistance in this research.

The authors thank the Royal Melbourne Institute of Technology (RMIT) for the use of the Encore facility.

The Laboratory for Concurrent Computing Systems is funded under a special research infrastructure grant for parallel processing research by the Australian Commonwealth Government.

Note all FFT codes are available from the principal author.

---

* · A.L. Cricenti is a member of Faculty in the School of Electrical Engineering and a Researcher in the Laboratory for Concurrent Computing Systems at the Swinburne University of Technology, John Street, Hawthorn 3122, Australia, Phone: +61 3 819 8516, E-mail: alc@stan.xx.swin.oz.au.

G.K. Egan is Professor of Computer Systems Engineering and Director of the Laboratory for Concurrent Computing Systems at the Swinburne University of Technology, John Street, Hawthorn 3122, Australia, Phone: +61 3 819 8516, E-mail: gke@stan.xx.swin.oz.au.

A Parallel Implementation of Nonnumeric Search Problems in SISAL

Andrew Sohn
Dept. of Computer and Information Science
New Jersey Institute of Technology
Newark, NJ 07102-1982, sohn@cis.njit.edu

Abstract - *Parallelization of nonnumeric problems is known to be difficult due to the unknown number of iterations and random memory usage. This report presents experiences parallelizing and implementing typical nonnumeric problems using Sisal. In particular, we select two search problems: the Eight-Puzzle and the Towers of Hanoi. We take a two-step parallelization approach to implement the nonnumeric problems: algorithm level and implementation level parallelization. Algorithm level parallelization uses various search strategies suitable for parallel search. Implementation level parallelization is done by the Optimizing Sisal Compiler (OSC). Parallelism profiles of the two search problem are plotted to help evaluate the effectiveness of parallelization process. Both the sequential and the parallel versions are implemented in Sisal and executed on a 26-processor Sequent Symmetry shared-memory multiprocessor. Experimental results demonstrate that (1) the combination of Sisal and OSC is effective for parallel implementation of the two nonnumeric search problems, and (2) the automatic parallelizing compiler OSC can give high programmability as it gives up to six-fold speedup on 26 processors, with little efforts on implementation level parallelization. However, we also find that the implementation level parallelization provided by OSC needs improvement as nonnumeric problems often require finer parallelization due to nondeterministic and sequential nature.*

1 Introduction

The issue of processing *nonnumeric* problems has been one of the major research foci of parallel processing. Those search problems typically found in artificial intelligence are such representative problems. They are however known to be difficult to parallelize due to (1) unknown number of iterations (or perhaps number of recursive function calls) at compile time, and (2) highly irregular memory accesses and large resource usage at runtime. Techniques developed to speedup processing of such nonnumeric problems can be classified into two different approaches: algorithmic heuristic approach and parallel processing approach [11].

The heuristic approach includes those various search strategies developed in artificial intelligence [9]. These heuristic approach is aimed at reducing the problem complexity by avoiding unnecessary paths in the state space. While brute-force search follows all the paths until a solution is found, heuristic search follows *selective* paths by using information which distinguishes promising states among others. A number of heuristic search has been developed, including hill-climbing, best-first search, A*, iterative-deepening A*, etc. [1,4,6,9]. Developing problem-dependent heuristics is central to this approach and is beyond the scope of this report.

Parallel processing approach emphasizes an actual implementation on parallel machines [5,7,10,11]. The basic assumption is the availability of multiprocessors. The approach can be divided into two steps: (1) algorithm level parallelization step, and (2) implementauon level parallelization step. The first step involves parallelizing the sequential algorithm of the given problem for algorithmic improvement. This step modifies the given algorithm to suit to multiprocessor environments, not necessarily a particular parallel machine environment. A study of the potential parallelism in the given algorithm is particularly helpful before and after the parallelization.

The second step is to implement the parallelized algorithms in parallel machine environments. Unless an automatic parallelizing compiler is used, various parallel constructs should be used to implement parallelized algorithms into parallel programs. This second step will have to take the machine architecture into consideration. Strategies of mapping programs onto processors, allocating resources to processors, and load balancing policies will become critical in this step if the parallel implementation is to be efficient on the target machine.

This paper presents a parallel processing approach to search, an important subset of nonnumeric problems. Specifically, we select two search problems: the Eight Puzzle and the Towers of Hanoi. Our target machine is a 26-processor Sequent Symmetry as because it is available to us. We attempt to parallelize the two problems both in algorithm level and implementation level. The two problems are all implemented in a functional language SISAL [2,3]. Algorithm parallelization is done through parallel A* search strategy [6,11] and parallel bidirectional A* search [10]. Much of the implementation level parallelization is done by the Optimizing SISAL Compiler (OSC) [2].

We start our discussion in section 2 by giving a brief introduction to search, followed by plotting potential parallelism in search problems. Section 3 presents parallelization methods on search problems. Section 4 lists execution results of the problems on the target multiprocessor. We discuss in section 5 the effectiveness of our approach to parallel implementation of search problems.

Last section concludes our experiences parallelizing and implementing nonnumeric problems in SISAL and Appendix lists an example SISAL program for the eight puzzle.

2 Parallelism in Search

Two search problems are briefly presented in this section. We list a generic and sequential search method, followed by the potential parallelism in the two search problems.

2.1 Two search problems

The Tower of Hanoi (ToH) and the Eight-Puzzle (8-P) are typical search problems found in AI. Given an initial state of each problem, the search process is to find a path which can lead to the goal state. The ToH with three disks can be stated as follows: There are three pegs, 1, 2, and 3, and three disks of different sizes A, B, and C. The disks can be stacked on the pegs. Initially the disks are all on peg 1; the largest, disk C, is on the bottom, while the smallest, disk A, is on top. It is desired to transfer all of the disks to peg 3 by moving one disk at a time. Only the top disk on a peg can be moved, but it can never be placed on top of a smaller disk.

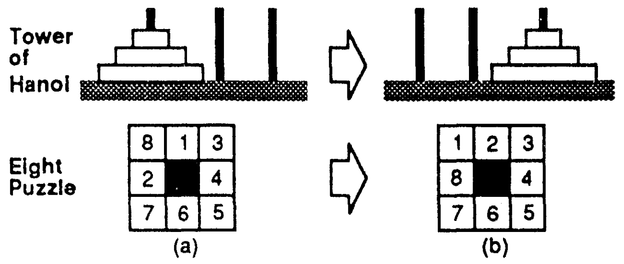


Figure 1: ToH and 8-P. (a) initial state, (b) goal state.

The Eight-Puzzle is another typical search problem. It consists of 8 numbered tiles set in a 3x3 frame, as shown in Figure 1. One cell of the frame is empty to allow an adjacent numbered tile to move into the empty cell. Given the initial state, the search process is to find a path which can lead to the goal state. While searching through the state space, a search strategy can be employed to guide the search process in an attempt to reduce the combinatorial explosion of the search effort. Figure 2 lists a generic and sequential search process.

```
OPEN={initial_state}, CLOSED=∅
Repeat
1.  TEMP ← select(OPEN)
2.  SUCC ← expand(TEMP)
3.  SUCC ← filter(SUCC,OPEN,CLOSED)
4.  OPEN ← merge(SUCC,OPEN – TEMP)
5.  CLOSED ← merge(TEMP,CLOSED)
Until (goal_state ∈ SUCC) or (OPEN = ∅)
```

Figure 2: A sequential search

It involves two lists: OPEN and CLOSED. OPEN contains nodes to be examined whereas CLOSED has those nodes that are already examined. Line 1 selects a *single* node from OPEN. Line 2 generates successors of the selected node. Line 3 removes from SUCC those nodes that are either on OPEN or CLOSED. Line 4 simply merges two lists to form a new OPEN. Line 5 also forms a new CLOSED. This algorithm is generic that it uses no particular search strategy. A particular search strategy can be embedded by modifying the merge step (line 4). For example, depth first search can be readily implemented by inserting SUCC in front of (OPEN – TEMP), provided that selection step takes a node from front of OPEN. Breadth First Search can be realized by inserting SUCC at the end of (OPEN – TEMP). A guided heuristic search, A* search strategy [4], can also be easily implemented by inserting nodes on SUCC into (OPEN – TEMP) in ascending (or descending) order of heuristic values.

The above generic search is sequential in three aspects: First, the entire loop is sequential due to loop-carried dependencies between iterations. The central data-structure is OPEN. Iteration i uses OPEN which was *modified* at iteration $i-1$. No two iterations can therefore be executed in parallel. The second sequentiality lies within the loop. Each iteration consists of five steps. Those five steps are again sequential due to their data dependencies. For example, lines 2 and 3 cannot be executed in parallel due to the true data dependencies. The third sequentiality stems from the fact that the selection step selects only *one* out of many nodes on OPEN. The expansion step therefore works only on one node at a time in each iteration.

2.2 Parallelism in search problems

Potential parallelism in the ToH is constructed by using a data-flow graph generated from the SISAL functional language. Parallelism refers to a number of instructions (such as +, –, *, /, etc.) which can be executed in an instruction cycle. Figure 3 shows a parallelism profile of the ToH with 3 disks. The x-axis indicates the execution time while the y-axis is a number of instructions that can be executed in parallel, namely, parallelism. Note that the y-axis is plotted to logarithmic scale.

Execution of the ToH begins with an insertion of the initial state on OPEN. Each iteration goes through the repeat loop of Figure 2. As we observe from the above figure, the amount of potential parallelism increases exponentially. Each iteration shows two typical peaks: the first peak is an expansion step (line 2 of Figure 2) while the second peak is a filter step (line 3 of Figure 2).

For example, consider iteration 4 which spans roughly 2000-2900. The first peak, spanning 2122-2600, is an expansion step which generates SUCC for all nodes on OPEN. Parallelism is high at the beginning of the expansion step but quickly diminishes towards the end of the expansion

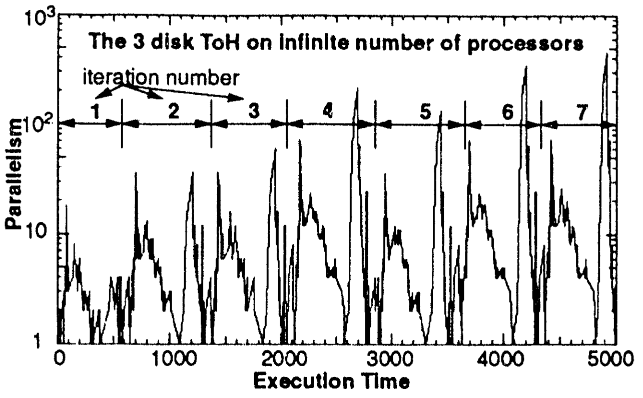


Figure 3: A parallelism profile of the ToH. Seven iteration is an optimal number of iterations for the 3 disk ToH. Note that the y-axis is plotted to logarithmic scale.

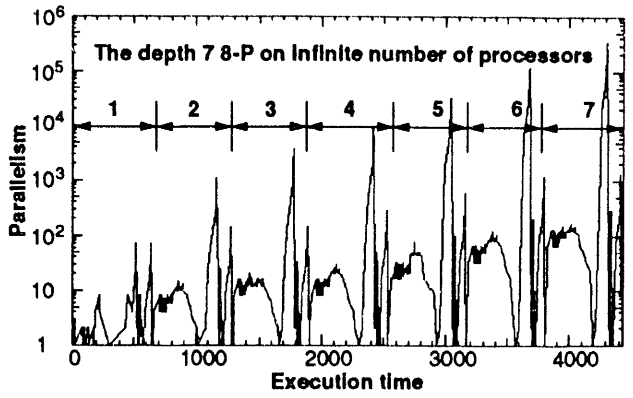


Figure 4: A parallelism profile of the Eight Puzzle. Note that the maximum parallelism for tree depth 7 of the 8-P is roughly 400,000!

step. One main reason exhibiting such exponential decay is because it involves many data copying operations and memory (common data structure) access operations. At the very beginning, *n* expansion functions will be called in parallel for *n* nodes on OPEN. However, instructions within each expansion function call are strictly sequential. In fact, most instructions engage in legality checking, i.e., check to see if which disk can move to what peg. This legality checking involves heavy array operations to prepare a new array for next possible legal movements.

The second peak, spanning 2601-2720, is a filter step. It checks OPEN and CLOSED to see if any node on SUCC has already been on one of them. It is not surprising that the filter step exhibits a thick and relatively high peak because the step consists of three nested loops (one for each list) and is *completely* parallelized. Our SISAL implementation uses an array of *p* elements to represent a node, where *p* is a number of disks. A simple algebra can easily show that checking *n* nodes on SUCC, *m* nodes on OPEN, and *l* nodes on CLOSED will execute *lmnp* comparison instructions.

Figure 4 shows potential parallelism for the Eight Puzzle. Its parallelism behavior is essentially the same as that of the ToH except now that the amount of parallelism is even higher. And the rate at which the parallelism increase in iterations is much greater than the ToH due to the larger branching factor. The ToH has a branching factor of two on the average while the 8-P has three.

One would argue that the search process presented in Figure 2 does have much *task-level* (or coarse grain) parallelism to explore. We shall find below if it is indeed the case. Consider a search tree for the Eight-Puzzle shown in Figure 5. Assume that nodes are generated in the following order: left, right, up, and down. A simple parallelization method assigns a processor to each path of the search tree.

Assuming that we have 10 processors, all paths can be searched by 10 processors as shown in the figure. Whichever processor finds the goal state will terminate the search process. This parallelization is simple and straightforward to implement. It would possibly give speedup close to linear as 10 paths are explored *independently* and *simultaneously* by 10 processors. However, we reject this type of task-level parallelization because (1) it lacks the global view of the search tree, (2) it requires a number of processors proportional to the number of paths in the search tree, and (3) it is unlikely to find an optimal solution in an optimal (or suboptimal) number of iterations unless an infinite number of processors is available. Further, this simple parallelization is not a *true* parallelization method as *no* processors cooperate to solve the problem. We present below how we parallelize search process.

3 Parallelization

We discuss below how we parallelize the problems we selected to suit the multiprocessor environments. Parallelization of search problems are described in detail.

3.1 Parallel A search*

A* search is a highly efficient search strategy [4]. It is a guided search based on the evaluation function $f = g + h$, where *g* is the cost of getting to the current state from the initial state, and *h* is the estimated cost to reach the goal from the current state. Given *n* nodes on OPEN, it selects the most promising node, i.e., the node with the lowest (or highest) *f* value. The performance of A* search depends on the quality of heuristic function which estimates the remaining distance to the goal state from the current state.

Parallel A* search (PA*S) is the same as A* search except now that *n* nodes can be examined simultaneously by *n* processors [7,11]. Figure 2 can be modified slightly to accommodate PA*S as follows:

```
1. TEMP ← select_n_best_nodes(OPEN)
2. SUCC ← parallel_expand(TEMP)
3. SUCC ← parallel_filter(SUCC,OPEN,CLOSED)
4. OPEN ← insert(SUCC,OPEN – TEMP)
```

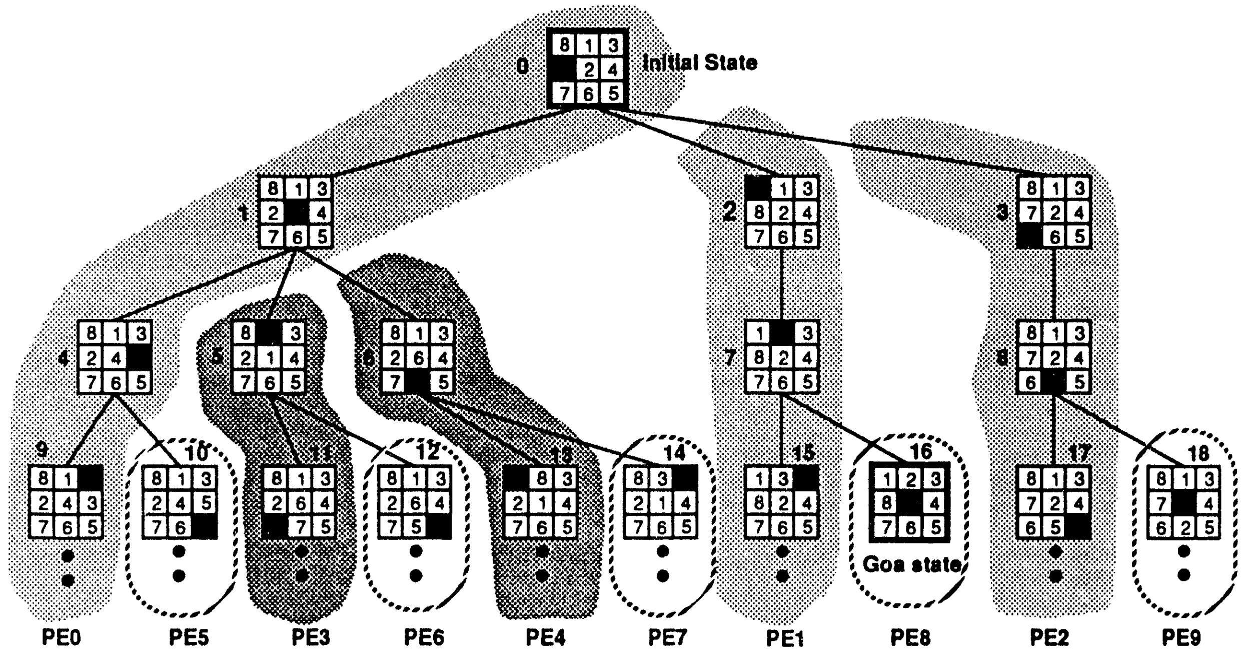


Figure 5: A simple parallelization method. As a new path is found, a new processor, if available, is assigned to the path. If every processor is working on a path, each processor will have to work on more than a path as the search tree grows. The above figure requires 10 PEs all of which are working *independently* and *simultaneously* to find the goal state in 3 iterations.

The main difference between the above and the sequential search of Figure 2 is the selection step and the insertion step. The selection step takes the first *n* nodes on OPEN, assuming that nodes on OPEN are now placed in ascending order of *f* value. Taking the first *n* nodes from OPEN is equivalent to selecting the best *n* nodes on OPEN. The merge step inserts SUCC into OPEN – TEMP in ascending order of *f* such that the node with the lowest *f* value is placed at the beginning of OPEN.

For our experiments, we set *g* to 1 per arc and *h* to *the number of misplaced tiles*. This heuristic function gives the goal state 0 for *h* value and therefore the most promising node is the one with the lowest *f* value. To briefly illustrate how A* works, let us use $N_{f,g,h}$ to denote a node with the three values. For example, $2_{4,1,3}$ denotes node 2 with f=4, g=1, and h=3. Suppose we have *two* processors. Given OPEN = (node 0), the PA*S will go through the following iterations (see Figure 6):

It. No.	Slct/Exp		SUCC		OPEN	CLOSED
	PE0	PE1	PE0	PE1	PE0	
0					0	
1	0		1,2,3		$1_{4,1,3}$, $2_{4,1,3}$, $3_{6,1,5}$	0
2	1	2	4, 5, 6	7	$7_{4,2,2}$, $3_{6,1,5}$, $5_{6,2,4}$, $4_{7,2,5}$, $6_{7,2,5}$	0,1,2
3	7	3	15,16	8	$16_{3,3,0}$, $15_{6,3,3}$, $8_{6,2,4}$, $5_{6,2,4}$, $4_{7,2,5}$, $6_{7,2,5}$	0, 1, 2, 7,3

For example, at the beginning of iteration 0, OPEN=(0) and CLOSED=(). Now, PE0 selects 0 (there is only one on OPEN anyway) and generates (1, 2, 3). Attaching the three values (*f,g,h*) to each successor node and inserting them into OPEN in ascending order of *f*, we now have OPEN = ($1_{4,1,3}$, $2_{4,1,3}$, $3_{6,1,5}$) and CLOSED = (0). Note that OPEN is kept at PE0 (master PE) while CLOSED is evenly distributed to two PEs. The PA*S takes three iterations to find the goal state (node 16) which is an optimal solution (or tree depth) for this particular initial state (node 0).

3.2 Parallel Bidirectional A Search*

Bidirectional search examines the search space from an initial state and a goal state, hoping to meet somewhere between them. Parallel Bidirectional A* search (PBiA*S) is a bidirectional version of PA*S [10]. It searches from both directions in parallel while search in each direction is *also* performed in parallel. To illustrate the PBiA*S, we now need four lists: TOPEN and TSUCC for top-down (forward) direction, and BOPEN and BSUCC for bottom-up (backward) direction. CLOSED does not change as we need only one CLOSED. The following implements the PBiA*S:

```
TOPEN=(initial_state), BOPEN=(goal_state),
CLOSED=∅, TSUCC=(), BSUCC=()
Repeat
  By Processor 0:
    1. TTEMP ← select_m_nodes(TOPEN) ;m=n/2, n PEs.
    2. TSUCC ← parallel_expand(TTEMP)
    3. TSUCC ← parallel_filter(TSUCC,TOPEN,CLOSED)
    4. TOPEN ← merge(TOPEN – TTEMP, TSUCC)
  By Processor 1:
    1. BTEMP ← select_m_nodes(BOPEN)
    2. BSUCC ← parallel_expand(BTEMP)
    3. BSUCC ← parallel_filter(BSUCC,BOPEN,CLOSED)
    4. BOPEN ← merge(BOPEN – BTEMP, BSUCC)
  5. CLOSED ← TTEMP ∪ BTEMP ∪ CLOSED
Until (TSUCC ∈ (BOPEN,CLOSED)) or (BSUCC ∈ (TOPEN,
CLOSED)) or (TOPEN = ∅) or (BOPEN = ∅)
```

Suppose we have *four* processors. Given TOPEN = (node 0), BOPEN = (node 16), and CLOSED=(), The PBiA*S will go through the following iterations (see Figure 6):

Fwd It. #	Slct/Expnd		TSUCC		TOPEN	CLOSED
	P0	P1	P0	P1	P0	
0					0	
1	0		1, 2, 3		$1_{4,1,3}$, $2_{4,1,3}$, $3_{6,1,5}$	0, 16
2	1	2	4, 5, 6	7	$7_{4,2,2}$, $3_{6,1,5}$, $5_{6,2,4}$, $4_{7,2,5}$, $6_{7,2,5}$	0, 16, 1, 2, 7, 19

Bwd It. #	Slct/Expnd		BSUCC		BOPEN
	P2	P3	P2	P3	P2
0					16
1	16		19, 20, 7, 21		$7_{4,1,3}$, $19_{5,1,4}$, $20_{6,1,5}$, $21_{6,1,5}$
2	7	19	2, 26	22,23	$2_{4,2,2}$, $22_{6,2,4}$, $26_{6,2,4}$, $20_{6,1,5}$, $21_{6,1,5}$, $23_{7,2,5}$

The first column indicates iteration numbers. Select/Expnd shows those nodes that are selected and expanded by the corresponding processor. For example, the following four activities simultaneously take place at iteration 2:

- Processor 0 selects 1 and generates (4, 5, 6).
- Processor 1 selects 2 and generates (7), resulting in TOPEN=(7,3,5,4,6).
- Processor 2 selects 7 and generates (2,26)
- Processor 3 selects 19 and generates (22,23), resulting in BOPEN=(2,22,26,20,21,23).

Iteration 2 then gives TCLOSED = (0,16,1,2,7,19). At this moment, the search process stops because (TSUCC,BSUCC)∩(TOPEN,BOPEN,CLOSED) = 7 ≠ ∅. The PBiA*S takes two iterations to meet in between the search space.

4 Experimental Results

We execute the two search problems on our target machine. Execution results are listed in this section. There are three types statistics collected including execution time, number of iterations and number of nodes searched.

4.1 Execution time

All forgoing parallelization techniques and problems are implemented in SISAL and executed on a Sequent Symmetry shared-memory multiprocessor. An example program is listed in Appendix. Various execution results are listed in this section. The statistics we collected are execution time, number of iterations, and number of nodes generated. The target machine we chose to run our implementation is a 26-processor Sequent Symmetry Model 81 shared-memory multiprocessor. All the programs are implemented in a functional language SISAL. Part of the parallelization is done by the Optimizing SISAL Compiler (OSC) [2]. Loops specified in parallel 'for' of SISAL are converted to parallel loops by OSC and executed in parallel by the Symmetry.

For the Tower of Hanoi, we have executed five different problem sizes: 3-7 disks. For the Eight-Puzzle, we have

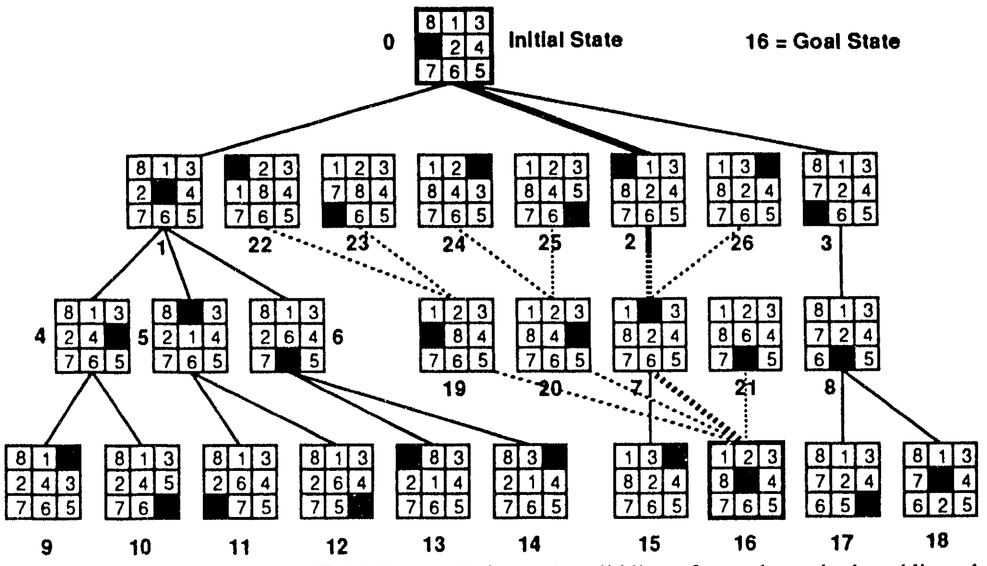


Figure 6: Parallel Bidirectional A* Search. Thick lines=solution path, solid lines=forward search, dotted lines=backward search.

also implemented 5 different problems as shown in Figure 7. Each problem is defined in terms of the optimal number of iterations (or tree depth). A particular initial state uniquely defines the problem size of the 8-P since the optimal number of iterations is fixed for the given initial and goal states. For example, the left most initial state of Figure 7 (d=10) indicates that the tree depth (or optimal number of iterations to reach the goal state) is 10.

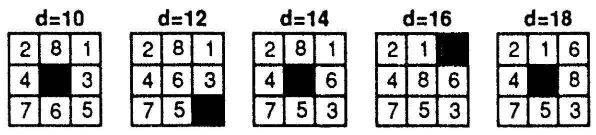


Figure 7: Five initial states for the 8-P. Goal state is in Figure 6.

Three types of execution results are collected: *execution time*, *number of iterations*, and *number of nodes searched.* Execution time is central to evaluate our approach since it eventually indicates the performance of our parallelization techniques. Tables 1 and 2 summarize execution time for the two problems.

No of Processors	Parallel A* (PA*S)				
	10	12	14	16	18
1	1.00	1.73	14.46	63.64	727.55
2	0.70	1.23	9.03	38.13	435.50
4	0.53	0.86	6.66	26.25	298.30
6	0.45	0.94	5.61	23.21	249.67
8	0.70	0.91	5.80	20.86	222.69
10	0.86	1.18	4.29	19.76	208.85
12	1.08	1.45	4.23	19.39	209.11
14	1.28	1.85	4.33	19.31	195.29
16	1.59	2.14	6.41	22.57	192.98
18	1.85	2.52	5.75	20.54	184.19
20	2.32	3.16	6.34	20.50	187.12
22	2.45	3.68	7.32	16.41	184.47
24	3.28	6.98	9.32	16.13	191.72
26	4.89	6.91	13.79	20.08	208.32

No of Processors	Parallel Bidirectional A* (PBIA*S)				
	10	12	14	16	18
1	0.83	2.69	8.18	11.73	94.53
2	0.57	1.51	4.81	7.85	52.46
4	0.68	1.62	2.79	5.57	34.93
6	0.35	1.40	2.84	4.41	25.21
8	0.46	1.07	2.67	4.02	21.96
10	0.53	1.29	2.27	4.64	18.28
12	0.61	1.15	2.24	3.58	19.53
14	0.66	0.91	1.97	3.65	17.45
16	0.73	1.12	2.33	2.90	17.26
18	0.74	1.21	2.57	3.19	16.59
20	0.86	1.45	3.16	3.94	17.49
22	0.86	1.46	3.35	4.27	17.27
24	1.03	1.87	4.09	5.29	17.40
26	1.03	1.83	4.44	5.58	17.04

Table 1: Execution time of the 8-P (in seconds)

No of Processors	Parallel A* Search (PA*S)				
	3 disks	4 disks	5 disks	6 disks	7 disks
1	0.08	0.33	3.99	22.58	326.53
2	0.06	0.27	2.79	15.69	313.73
4	0.04	0.17	2.32	4.16	194.00
6	0.05	0.21	1.10	10.73	88.63
8	0.05	0.21	1.18	8.19	80.53
10	0.05	0.21	1.24	8.46	95.56
12	0.05	0.22	1.42	8.95	81.56
14	0.05	0.22	1.32	9.58	76.25
16	0.05	0.22	1.39	9.41	77.99
18	0.05	0.23	1.40	9.85	81.07
20	0.05	0.23	1.41	10.46	84.85
22	0.05	0.23	1.40	10.93	88.13
24	0.06	0.23	1.43	11.63	90.47
26	0.06	0.25	1.44	11.74	95.99

No of Processors	Parallel Bidirectional A* Search (PBIA*S)				
	3 disks	4 disks	5 disks	6 disks	7 disks
1	0.06	0.33	2.26	19.68	164.22
2	0.05	0.27	1.69	13.14	125.39
4	0.06	0.19	1.37	10.54	94.76
6	0.05	0.20	1.04	10.53	92.54
8	0.06	0.19	0.96	9.00	88.50
10	0.05	0.19	0.99	7.42	97.61
12	0.06	0.19	0.99	6.85	74.64
14	0.06	0.20	0.95	6.93	58.51
16	0.06	0.20	0.99	6.70	64.06
18	0.07	0.20	0.98	6.83	64.78
20	0.06	0.20	0.99	6.68	62.33
22	0.07	0.20	0.97	6.82	59.11
24	0.07	0.21	1.00	6.75	58.26
26	0.06	0.22	0.99	7.15	58.35

Table 2: Execution time of the ToH (in seconds)

4.2 Number of iterations

The second type of runtime statistics is a number of iterations. Table 3 lists the total number of iterations. We consider the number of iterations is important not only to search problems but many other problems. It will enable us to identify the effectiveness of algorithm level parallelization and to characterize the efficiency of search strategies and heuristic functions. If we can estimate the total number of iterations for a given problem, we may be able to predict the runtime complexity and therefore can more effectively utilize precious resources such as memory. Further, the estimation of number of iterations will help execute the loop iterations in parallel to a certain extent. In fact, we have already undertaken an approach to partially overlap sequential loop iterations, called partial overlapping of loop iterations (POLI) [11]. This POLI is similar in principle to software pipelining but applied to a little higher level than instruction scheduling.

No of PEs	Eight Puzzle									
	Parallel A* (PA*S)					Parallel Bidirectional A* S				
	10	12	14	16	18	10	12	14	16	18
1	52	70	214	447	1440	23	45	77	97	271
2	27	37	109	224	722	12	22	39	53	137
4	14	19	57	114	366	9	15	19	29	72
6	10	15	39	78	244	5	11	15	20	47
8	10	12	31	60	182	5	8	12	16	36
10	10	12	23	49	147	5	8	10	15	28
12	10	12	20	41	125	5	7	9	12	26
14	10	12	18	37	106	5	6	8	11	22
16	10	12	19	34	94	5	6	8	9	20
18	10	12	17	31	84	5	6	8	9	18
20	10	12	16	28	77	5	6	8	9	17
22	10	12	15	24	70	5	6	8	9	16
24	10	12	14	22	66	5	6	8	9	15
26	10	12	14	22	62	5	6	8	9	14

No of PEs	Towers of Hanoi									
	Parallel A* (PA*S)					Parallel Bidirectional A* S				
	3	4	5	6	7	3	4	5	6	7
1	20	50	189	462	1622	9	28	87	264	803
2	10	29	107	267	1155	5	15	47	139	416
4	7	16	66	88	568	4	9	26	73	211
6	7	16	35	104	281	4	9	19	55	149
8	7	15	33	77	217	4	8	17	44	118
10	7	15	32	70	198	4	8	17	38	107
12	7	15	32	68	164	4	8	17	35	88
14	7	15	31	66	147	4	8	16	35	76
16	7	15	31	64	140	4	8	16	33	75
18	7	15	31	64	137	4	8	16	33	73
20	7	15	31	64	135	4	8	16	33	71
22	7	15	31	64	132	4	8	16	33	69
24	7	15	31	64	131	4	8	16	33	67
26	7	15	31	64	131	4	8	16	33	67

Table 3: Number of iterations for the two problems.

No of PEs	Eight Puzzle									
	Parallel A* (PA*S)					Parallel Bidirectional A*S				
	10	12	14	16	18	10	12	14	16	18
1	99	132	383	776	2462	89	164	292	339	951
2	101	137	388	777	2466	90	155	292	367	954
4	100	132	399	782	2490	124	203	277	391	995
6	95	149	398	793	2485	86	207	308	392	963
8	130	149	422	801	2463	110	187	317	399	975
10	146	180	369	804	2478	120	214	300	444	931
12	168	199	365	796	2515	130	196	302	396	1000
14	187	231	381	823	2476	134	170	282	404	961
16	206	247	461	861	2494	138	184	306	347	977
18	227	275	438	868	2472	138	192	327	373	964
20	246	300	446	857	2515	138	200	349	403	987
22	253	326	444	769	2504	138	204	369	432	998
24	268	348	438	747	2550	138	208	384	454	979
26	289	364	480	803	2587	138	212	408	475	957

No of PEs	Towers of Hanoi									
	Parallel A* (PA*S)					Parallel Bidirectional A* S				
	3	4	5	6	7	3	4	5	6	7
1	27	64	220	500	1704	26	67	188	545	1626
2	26	75	245	570	2308	25	71	198	570	1684
4	27	65	256	365	2260	22	61	206	586	1693
6	27	77	192	623	1713	22	62	176	621	1768
8	27	81	209	569	1703	22	58	169	573	1769
10	27	81	219	587	1887	22	58	170	526	1900
12	27	81	235	612	1769	22	58	170	505	1668
14	27	81	231	632	1730	22	58	165	506	1499
16	27	81	243	635	1757	22	58	166	493	1537
18	27	81	243	657	1802	22	58	166	494	1548
20	27	81	243	673	1849	22	58	166	494	1516
22	27	81	243	692	1883	22	58	166	494	1500
24	27	81	243	713	1912	22	58	166	494	1477
26	27	81	243	717	1951	22	58	166	494	1480

Table 4: Number of nodes generated for the Eight-Puzzle.

4.3 Number of nodes

The third type of runtime statistics is a number of nodes searched. Table 4 lists the total number of nodes generated for the two problems. This third information indicates (1) the total amount of work to be done to solve a certain problem and (2) the efficiency of the heuristic search algorithm used to solve the problem. If the total number of iterations indicates how much execution time can be reduced by using parallel processing techniques, the number of nodes searched indicates how efficient the algorithm is for the given problem.

Table 4 indicates that the amount of work to be done for the given problem size is almost the same, regardless of number of processors. We note however there is still a little fluctuation as the number of processors changes. The fluctuation is due mainly to the fact that the number of nodes searched is determined by the three values, f, g, and h.

5 Discussion

Execution results are analyzed in terms of two types of speedup curves: one from the number of iterations and the other from execution time. These two speedup curves are in turn compared to identify if the search problems can indeed be parallelized. Three search strategies are also compared based on our problem instances to identify the merit of each search strategy.

5.1 Effect of algorithm level parallelization

Number of iterations is one of the important parameters. It characterizes the efficiency of search strategies and heuristic functions. To further clarify the efficiency of various search strategies, let $I_{d,n}$ be the number of iterations for a problem size d on n processors. We define iteration speedup for a problem size d as $S_{iter}(d) = I_{d,1}/I_{d,n}$, where n is a number of processors. For example, consider d=14 of the

8-P using PA*S. We find from Table 3 that $I_{14,1}$=214 and $I_{14,26}$=14. Speedup for d=14 using PA*S on 26 processors is $S_{iter}(14) = I_{14,1}/I_{14,26} = 214/14 = 15.3$. Figure 8 plots some of the speedups drawn from the number of iterations for the 8-P.

It appears that our implementation can give a significant speedup for the given problem instances. Considering that the A* search strategy is a highly sequential algorithm, the speedup of 15 is significant. However, it should be noted that this speedup is an *ideal* algorithmic speedup, which can be achieved only in an ideal environment. The ideal environment here refers to a parallel machine with no communication and synchronization overhead.

From the above plots, we find that the PA*S performs slightly better than PBiA*S in general. The main reason is that one iteration of PBiA*S is equivalent to 2 iterations of PA*S. This also explains why the rate of change of speedup for PBiA*S is slightly smaller than that of PA*S. In any case, the above plots suggest that each strategy can give a substantial speedup in an ideal environment. We shall find out below it is indeed the case.

5.2 Effect of implementation level parallelization

Execution time shown in Tables 1 and 2 are now converted to speedup factors to identify the effectiveness of ***implementation*** level parallelization. Execution time speedup, S_{exe}, is defined the same as that for iteration, except now that real execution time is used. Figure 9 plots some of the speedups. We find that speedup is disappointing as the maximum speedup for the given instances is merely 6.5. In an ideal environment, we obtained over 15. There is a significant difference between the two speedups, S_{iter} and S_{exe}. This large difference indicates that the parallelization techniques are not the best choice. In any case, we observe from the curves that speedup increases as the problem size increases for both the PA*S and the PBiA*S.

5.3 Discrepancy of two speedup curves

We have seen from the previous two curves (iteration speedup S_{iter} and execution time speedup S_{exe}) that there is a large discrepancy between them. Discrepancy indicates how effective our parallel search strategies are in a multiprocessor environment. It also indicates whether our parallelization methods can effectively utilize potential

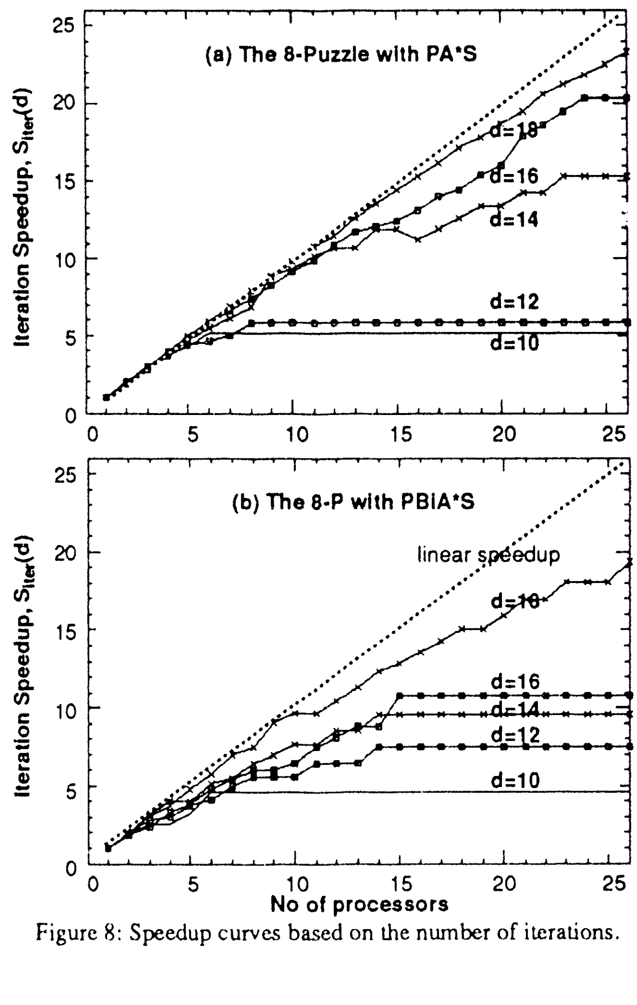


Figure 8: Speedup curves based on the number of iterations.

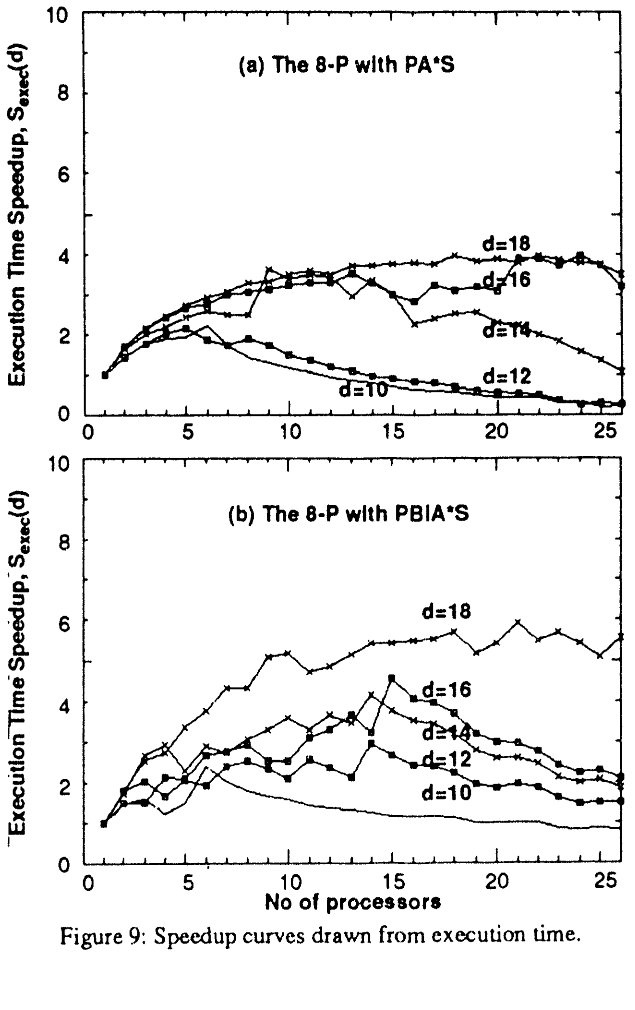


Figure 9: Speedup curves drawn from execution time.

parallelism present in the given algorithm. Large discrepancy indicates that the search strategy is ineffective in utilizing potential parallelism while small discrepancy indicates that it is effective. To be more precise, we define discrepancy as $D(d) = (S_{iter}(d) - S_{exec}(d))/S_{iter}(d)$, where d is a problem size.

Consider the 8-P with d=14 on 26 processors. From Tables 1 through 3, we obtain:

Search Strategy	$S_{iter}(14)$	$S_{exe}(14)$	$D(14)$
PA*S	15.3	1.0	(15.3−1.0)/15.3 = 93%
PBiA*S	9.6	1.8	(9.6−1.8)/9.6 = 81%
Comparison	$D_{PBiA*S}(14) < D_{PA*S}(14)$		

The above relation indicates that the PBiA*S is better than PA*S in terms of parallelism utilization. We find that this relation holds for most of the problem instances we implemented. Figure 10 shows discrepancies for the PA*S and the PBiA*S. In general, PA*S gives larger discrepancy than PBiA*S. The main reason is that the PA*S does more work than the PBiA*S, where work refers to the number of nodes generated. PA*S generates more than twice the number of nodes PBiA*S does for d=18 (Table 4). The inequality becomes prominent as the problem size increases.

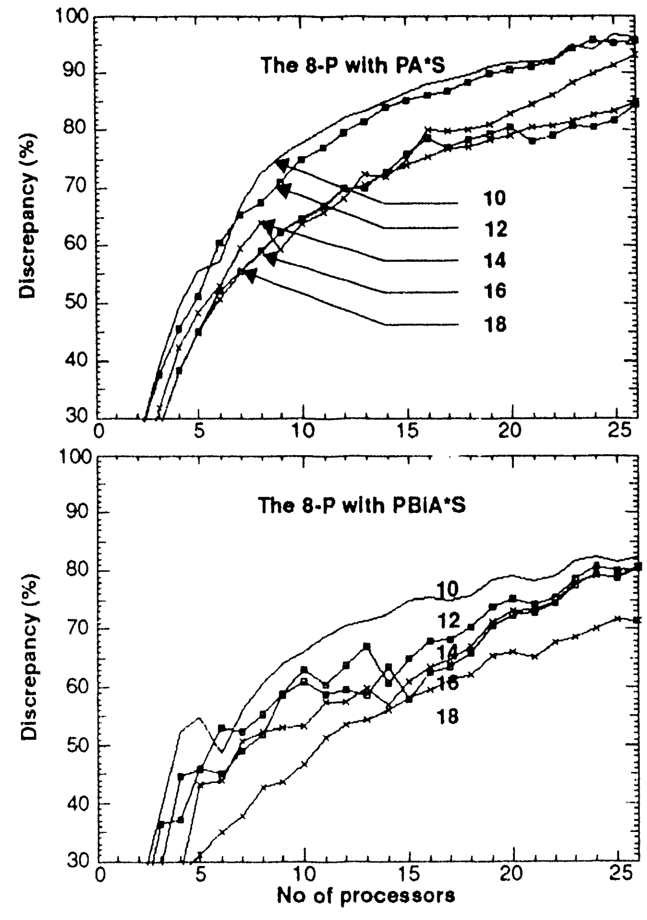


Figure 10: Discrepancy of iteration speedup and execution time speedup.

For each individual search strategy, we find that the discrepancy *decreases* as problem size increases. This is a promising result since the parallelization method starts showing its effectiveness. For small problem sizes, the parallelization methods are not effective but they become effective for large problem sizes. We have already discussed this effect in Figure 10.

To summarize, the large discrepancy indicates that the parallelization method is not the best one. However, at the same time it also indicates that there is *room* to explore. In other words, larger discrepancy implies that the search strategy can be further parallelized to improve parallelism utilization. This is a rather a promising observation. We take this large discrepancy as a challenge which motivates us to further investigate on parallelization methods for the search problems.

5.4 Discussion

We observe from the curves that (1) speedup increases in general as the problem size increases, and (2) speedup also increases as the number of processors increases for a particular problem size. This verifies that the approach we have taken to parallelize the given problems is effective for the selected problem instances.

However, when we closely examine the speedup curves and parallelism profiles we constructed earlier, we find that our implementation is not highly effective. Note in section 2 that we plotted potential parallelism. Figures 3 and 4 indicated that the three problems do have a substantial amount of instruction level parallelism. The speedup curves of Figure 9 show however that a maximum speedup is merely 6-fold. This is rather contradictory. Where are the tens of thousands of instruction-level parallelism gone? Did our implementation ever utilize the potential parallelism in the given problems? The speedup curves show that parallelism was apparently not utilize by our implementation. Although the three problems do have parallelism, the automatic parallelizing compiler did not effectively utilize them. We believe that the main reason this under-utilization of parallelism is due mostly to loop slicing which assigns an independent iteration of loops to a processor.

For the search problems, the speedup is obtained mostly from the parallel expansion step, where n nodes are expand by n processors. For the Ops5, the speedup is obtained mostly from the parallel pattern matching step. Note that these two steps are function calls, each of which contains a large number of instructions. They can be classified into medium (or perhaps coarse) grain parallelism, but certainly not instruction level fine-grain parallelism. As a matter of fact, to obtain this much speedup of up six, we had to dramatically change the original algorithm. If we did not change the algorithms, the OSC may have found difficulties extracting these parallelism.

This reminds us of an old issue, programmability versus performance. On one hand, we wish to have a high programmability by having an automatic parallelizing compiler take care of all the parallelism utilization. On the other hand, we like to obtain a good performance (or high speedup) by having manual parallelization which would give much control over the potential parallelism. As our experimental results demonstrate, obtaining both the high programmability and good performance is not an easy task for our problem instances.

In any case, our SISAL implementation gives a high programmability. Considering that an ultimate goal of parallel processing is to provide a tool which can automatically parallelize and execute programs written independent of the underlying machine architecture, the SISAL and OSC are certainly successful. Our target nonnumeric problems are different from scientific computation. As we have stated earlier, they are known to be difficult to parallelize due to irregular memory usage and unknown number of iterations. We spent little effort to parallelize the problems. After all, the SISAL and OSC gave a reasonable performance of up to six-fold speedup. We are currently working on how to effectively utilize potential parallelism in nonnumeric problems.

7 Conclusions

Parallel implementation of nonnumeric problems remains a challenging task. We had difficulties implementing the two problems. The main difficulty was the large resource usage of the problems. The two search problems are small in terms of code size, several pages of SISAL codes but large in terms of runtime and storage usage. One can implement the search problems in double recursion in Lisp which would perhaps take only a page or less of Lisp code. However, it was simply not the case for our implementation since we had to utilize parallelism by writing programs more complex.

The search problems indeed have shown very high runtime and space complexity. Their memory usage (space complexity) was very large that the 8-P with depth 18 needed 16MB of runtime storage. This is precisely the reason why we were unable to execute a tree depth of say 100, for example. After all, we have successfully implemented the selected problems using SISAL on the target machine, a Sequent Symmetry shared-memory multiprocessor. We have spent much time writing codes and parallelizing the three problems both in algorithm level and implementation levels. The search problems have been parallelized and implemented using two search strategies (parallel A* search and parallel bidirectional A* search). In the course of parallelization, we plotted potential parallelism in the problems. This construction of parallelism profiles was very helpful to determine where we should concentrate on our parallelization efforts. We included several parallelism profiles to demonstrate the potential parallelism in the two search problems.

Our execution results have indicated that the nonnumeric problems can also be effectively parallelized and implemented using the combination of SISAL and OSC for a shared memory machine. We were able to obtain up to a maximum of six-fold speedup on 26 processors. However, we have also found that the utilization of parallelism is poor for the two problems. Our parallelism profiles have indicated that the nonnumeric problems have large potential parallelism but the execution results were not as good as the profiles suggested. This under utilization of parallelism has to be improved for the SISAL and OSC to be successful for realistic nonnumeric problems. To conclude our experiences implementing nonnumeric problem, we find that the combination of SISAL and OSC can give a high programmability to effectively implement hard and sequential nonnumeric problems.

References

1. Bolch, L., and Cytowski, J., Search Methods for Artificial Intelligence, Academic Press, 1992.
2. Cann, D.C., and Oldehoeft, R.R., "A Guide to Optimizing SISAL Compiler," *Technical Report* UCRL-MA-108369, Lawrence Livermore Laboratory, Livermore, CA, 1991.
3. Feo, J.T., Cann, D.C., and Oldehoeft, R.R., "A Report on the SISAL Language Project," *Journal of Parallel and Distributed Computing* 10, December 1990, pp.349-365.
4. Hart, P.E., Nilson, N.J., and Raphael, B., "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on SMC* 4(2), pp.100-107, 1968.
5. Kale, L.V., and Saletore, V.A., "Parallel State-Space Search for a First Solution with Consistent Linear Speedups," in *Int'l Journal of Parallel Programming 19*, 1990, pp.251-293.
6. Kanal, L., and Kumar, V. (Eds.), Search in Artificial Intelligence, Springer-Verlag, 1989.
7. Kumar, V., Ramesh, K., and Rao, V.N., "Parallel Best-First Search of State-Space Graphs: A Summary of Results," in *Proc. AAAI-88*, pp.122-127.
8. McGraw, J.R., Skedzielewski, S.K., Allan, S.J., Oldehoeft, R.R., Glauert, J., Kirkham, C., Noyce, W., and Thomas, R., "SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual version 1.2," *Manual* M-146, Rev. 1, Lawrence Livermore Laboratory, Livermore, CA, 1985.

9. Pearl, J., Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, Reading, MA, 1984.

10. Sohn, A., "Parallel Bidirectional A* Search on a Symmetry Multiprocessor," in *Proc. IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, December 1993.

11. Sohn, A., Gaudiot, J-L., and Sato, M., "Performance Studies of the EM-4 Data-flow Multiprocessor on Combinatorial Search Problems," *Technical Report* 92-39, Electrotechnical Laboratory, Tsukuba, Japan, November 1992.

Appendix

The following SISAL code implements the eight puzzle with parallel bidirectional A* search strategy. Those function names in bold directly correspond to those names shown in section 3.

```
type node=array[array[integer]];
type state=array[node]; %A snap shot of search tree at time t.
function eight_puzzle(start,goal:node; n:integer
     returns integer,integer)
     let
          open, closed: state;
          open,iteration,nodes:=
          for initial
               open := array[1: start];
               closed := array state []; no_nodes:=1; i:=0;
               flag:=0; %1=goal found, 0 = goal not found.
          while (array_size(open)>0 & flag=0) repeat
               first,second:=select(old open,n);
               succ1 := expand(first);
               succ2 := filter(old open,old closed,succ1);
               succ := set_values(succ2,goal);
               open := insert(succ,second);
               closed := first || old closed; %merge
               flag:=goal_found(succ,goal,array_size(succ));
               i := old i + 1;
               no_nodes:= old no_nodes + array_size(succ);
          returns value of open value of i value of no_nodes
          end for
          in
               iteration,nodes
     end let
end function
```

Computer Vision Algorithms in Sisal

Srdjan Mitrovic
Computer Engineering and Networks Laboratory
ETH Zuerich, Switzerland

Marjan Trobina
Computer Vision Laboratory
ETH Zuerich, Switzerland

Abstract

Functional languages like Sisal 1.2 had the stigma that they either cannot produce efficient code or that they are not powerful enough to express all kinds of problems. While the reproach of inefficient code has been sufficiently disproved, the second reproach still remains. To either disprove the reproach or to make suggestions for changes in the language definition, Sisal user community is engaged in writing applications for many different computing domains. This work presents the experience in implementing computer vision algorithms in Sisal. The chosen problems are representatives of two different types of algorithms found in the computer vision domain. Furthermore, these algorithms already exist as C-programs that are used by the members of the Computer Vision Laboratory at ETH Zuerich. The paper concludes with a performance comparison between C- and Sisal-code on a sequential machine and measures the speedup obtained by Sisal programs on a multiprocessor.

1.0 Introduction

There are several elements of a language that can raise its acceptance in the programming community and we will name some. The implementation of a language, i.e., its availability across several programming platforms coupled with efficient compilers, comfortable programming environments and debuggers, is certainly important but will not be discussed in this paper[1]. Rather, we will look at the flexibility and the power of the programming constructs that support the programmers in the formulation of a problem.

The language designers have the dilemma between implementing very primitive but flexible constructs or adding complexity to a language by including constructs that support the abstract formulation of algorithms. On such a ruler of language constraints and complexity C and Sisal are situated on different ends. We should note that the measuring of constraints is not necessary the measuring of the flexibility because unnecessary[2] freedom must not add flexibility. It is logical that, to a certain degree, the presence of a "strait-jacket" for programmers in a language diminishes the number of errors in a program as well as it decreases the time needed to develop a program. However, it is important that the constraints should not limit the flexibility, in terms that most algorithms can be programmed in such language, and that the constraints provide the programmers with essential benefits. The structured programming (no GOTO's) and strong type-checking are two examples that demonstrate useful constraints in a programming language.

The "strait-jacket" in Sisal consists mainly of the array-constructs, reductions and the single-assignment rule which as a benefit offers the determinate execution behavior and automatic parallelization on varying parallel machine size. Ideally, it should be possible to write whole operating systems, compilers and other applications with the same language. We acknowledge that this is not the case for Sisal[3], but the vast amount of scientific applications can be formulated with it [Cann92] [CannFeo90] [FeoCaOl90]. This paper shows that Sisal is also suitable for implementing computer vision algorithms.

Apart from being able to formulate problems clearly, a language must be able to generate efficient machine-code[4]. This is the case when the semantic gap between the language or some of its intermediate representations is not too wide. Nonetheless, a language with a "strait-jacket" need not deliver worse code than a more "liberal" language. The "strait-jacket" introduces inefficiencies only when high-level constructs of a language cannot be mapped to efficient machine-code. It is a drawback of a specific construct and not of the principle. A language that has not efficiently

1 This exclusion does not marginalize the importance of excellent compilers and programming environments, rather they can sometime be the key to the success of a language.

2 The matter of what is necessary and what is not is rather a religous discussion so we will abstain from elaborating on this topic.

3 There are several ongoing research projects that investigate the expansion of functional languages by adding non-functional feattures. It is still not a certain thing how those two different paradigmas can coexist in reality.

4 For many scientific application the time to write, debug and maintain a program is much larger than the time that is needed to execute the program. This fact favours the usage of functional level languages anyhow.

implementable constructs, will suffer a failure in domains where not only the coding time but also the execution time is very important. This is the case for long and repetitive scientific simulations as well as in computer vision applications.

2.0 The Programming Languages

The languages C and Sisal will be often compared in this paper and we assume that the reader is acquainted with both languages.

C is a widely used imperative language with very primitive constructs and with efficient compilers on UNIX machines. It is a positive example how high-level programming can lead to efficient executable code. It is a negative example how unlashing freedom slows the coding of a problem and increases the debug time. Its main achievement is that of a portable and standardized high level assembler.

Sisal stands for Streams and Iterations in a Single Assignment Language [McGrSk85]. It is a functional language based on the single-assignment rule. It targets mainly the scientific computation on parallel machines. The Sisal compiler is available on many kinds of UNIX systems with efficient code-generators available for sequential and, shared-memory and vector parallel machines. There are also several code-generators written for non-conventional computer architectures [Mitrov93]. Major benefits of the Sisal language include its determinate behavior that allows developing and debugging parallel programs on sequential machines and the automatic parallelization of Sisal programs

3.0 The Algorithms

A computer vision algorithm receives a digitized image as its input and transforms it either to a new image or a different representation: the symbolic interpretation of the image. Because of the hard time-constraints in computer vision, the algorithms are often small and fast and have much less computation than data-access operations. Every image transformation, either to an image or to a symbolic form, is called an image operator. The evaluation of computer vision algorithms in this paper is based on three different image operators. The first two, Gaussian smoothing and Canny edge detector, transform images to images. The third algorithm, image compilation, transforms an image into an edge graph. We will shortly describe the foundations of these algorithms and compare their implementations. All three algorithms are implemented in a software tool used at the Computer Vision Laboratory at ETH Zuerich

The three distinct algorithms depend on each other's result and form a transformation pipeline as shown in figure 1. After an image is read from a camera or a file, it is passed to the three operators and transformed to a picture graph. Figure 1 does not depict some picture type conversion routines and thresholding operations although those routines are included in the transformation pipeline. The goal of the implemented algorithms is to detect object boundaries in the input image and to represent the extracted edges.

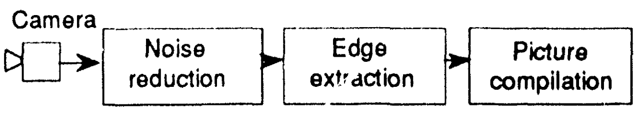


Figure 1: The relation between the three image operators

3.1 Noise reduction

The noise-reduction operator eliminates noise from an image. The implemented noise reduction is based on the Gaussian smoothing [Canny86]. The input to the Gaussian smoothing algorithm is an image containing gray values of pixels in single-precision floating-point format. This representation of an image is needed by noise reduction and edge extraction algorithms so that rounding losses remain small.

A half of Gaussian kernel is precalculated in the array *gauss_kern* as shown in the figure 2 where the value of *sigma* is a parameter given by the user. With a typical *sigma* of 1.0 the value of radius is 5.

```
radius := trunc(sigma*
                sqrt(-2.0*log(0.001))+0.5);
gauss_kern :=
     for x in 0, radius
     returns array of norm *
                etothe(-real((x)*(x))/sigq)
     end for;
```

Figure 2: Calculation of Gaussian transformation array

The array *gauss_kern* is used to process the input image in the horizontal direction first. The resulting picture is then smoothed in the vertical direction. The transformation in horizontal direction is described in figure 3 -- where only the body of the innermost loop is shown -- using the values calculated in figure 2.

```
newPoint :=
    pict[x,y]*gauss_kern[0] +
    for r in 1, radius
    returns value of sum
          (pict[x, y-r]+
          pict[x, y+r])*gauss_kern[r]
    end for
```

Figure 3: Gaussian smoothing in y-direction[5]

The result of the noise reduction is a picture, which is 2**radius* pixels less high and wide than the input picture.

3.2. Edge Detector

An edge detector is a mathematical operator that extracts edges from the intensity discontinuities in an image. It is easier to find object boundaries from the edge representation than from the gray-level values in an image. The chosen edge operator is described in [Canny86] and is also referred as Canny operator. Figure 4 depicts the computation structure of the algorithm that implements the Canny operator. It creates four derivation images from the smoothed image. Every pixel in every of the four derivation-images is compared with each other in the *NonMaxSuppression* algorithm. The resulting edge image has edges that are only one pixel wide and are of varying strength. The succeeding threshold operation removes the edges with weak strengths.

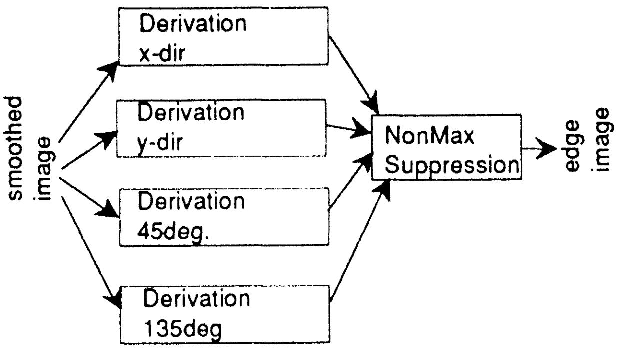


Figure 4: The structure of the Canny operator algorithm

3.3. Image Compilation

The construction of edges is primarily motivated by the need of the image compilation algorithms. These algorithms transform an image into an easier analyzable form, e.g., graphs or lists. The compilation may consist of grouping objects together and finding object boundaries, or even recognizing objects. We use a compilation algorithm that finds connected edges, their lengths and their point-coordinates. The result of this algorithm is a list of all nodes -- where every node points at one or several edges -- and a list of edges with all of theirs point-coordinates.

The implemented algorithm will be briefly sketched in the following. First, the edge image is transformed to a lambda-coded representation, which reduces the number of neighbors from a maximum of 8 to a maximum of 4 neighbors per pixel. This transformation makes the connection between two points unambiguous. Figure 5 shows on left the 8-neighborhood relation with the possible connections compared to the lambda-neighborhood on the right. We see that in 8-neighborhood there are two different ways to walk from pixel A to pixel C, one direct and one going over pixel B.

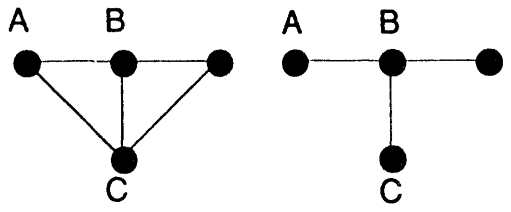


Figure 5: 8- neighborhood and lambda neighborhood

Every pixel in a lambda coded image contains a bit-pattern that defines the number of its neighbors and in what directions they are located.

The lambda-coded image is walked from the left to the right and from the top to the bottom. Together with the scanning of an image-row, a list of segments is being traversed and updated. Using this list of segments we can locate the start and the ends of edges as well as extract the edges that build circles, i.e., have no ends or starts. The detailed discussion is beyond the scope of this paper and can be read in [Klein87] and [KleKueb87].

Figures 6a and 6b show the input image -- a digitized view of the city of Zuerich -- and the result image that is reconstructed from the edge graph. The picture compilation algorithm passes the lines to a filter that redraws the picture with the edges that are longer than a specified minimal value.

[5]Note that in mathematical and Sisal notation, horizontal means y-direction and in computer vision terminology, x is the horizontal coordinate

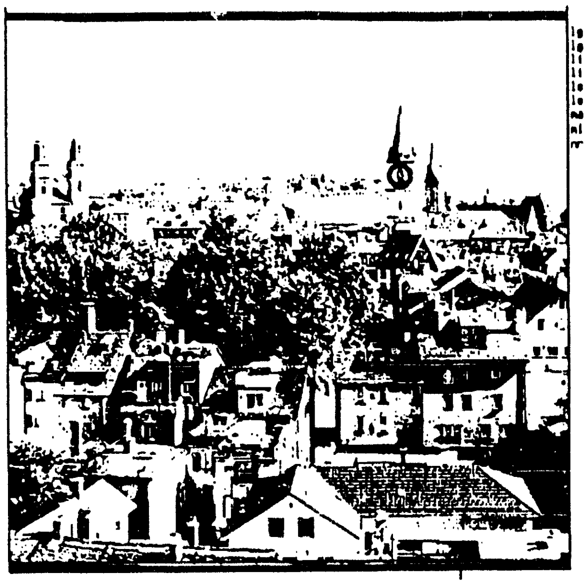


Figure 6.a: The input frame

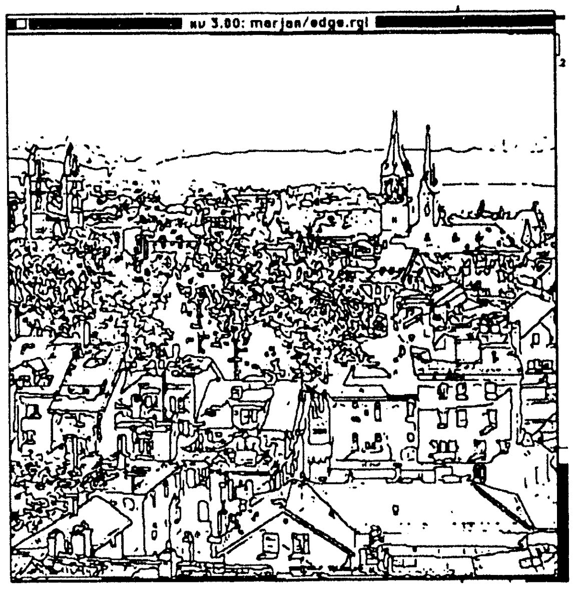


Figure 6.b: The result frame

4.0 The Implementations

The algorithms described in the previous paragraphs are implemented in C and Sisal. Although they deliver the same results and are based on the same algorithms, C and Sisal versions differ in some points that will be discussed in the following paragraphs. The reason for differences between Sisal code and the implementations in other imperative languages stems from the fact that SISAL programmer has to investigate the mathematical foundation of a problem instead of imitating the existing implementation that often takes into account how to map an algorithm on the machine. Therefore, we should rather talk about implementing than porting an algorithm to Sisal.

One difference is the representation of an image. In C-code an image is described by the image descriptor and the image data. The image descriptor holds information about the horizontal and the vertical size of the image as well as what part of the image contains valid data -- it is called the region of interest. It is the duty of the C programmer to correctly assign the values to the image descriptor. In Sisal the arrays are dynamic, i.e., the bounds and size can be changed at run-time. Therefore, there is no image descriptor needed in Sisal code[6]. The correct allocation and deallocation of data are also the task of the C programmer. In Sisal there are no memory handling operations because its intermediate form, the dataflow graphs, make automatic memory management feasible without use of a dedicated garbage collector. The two-dimensional access to a point of an image in C is written as a one-dimensional access (using the information about the total width of image-data) or just by increasing a pointer (when scanning sequentially through the image). Sisal code is written using two-dimensional access by specifying both indices.

The implementation of the smoothing and edge extraction operators is straightforward in Sisal because it offers powerful array handling operators. Figure 7 shows a code skeleton for a frame to frame image operator in Sisal.

```
for x in x1,x2 cross y in y1,y2
     newPoint := FrameOperator(pict[x,y])
     returns array of newPoint
     end for
```

Figure 7: Code skeleton for a frame to frame transformation

The graph compilation algorithm is more difficult to implement in Sisal as it is based on a dynamic list of segments that is being walked through. Lists do not exist in Sisal and therefore the segment-list has been implemented as a dynamic array[7]. This approach includes much copying because the insertion into a list is more efficient than insertion into an array.

The code that involves reading data from an image input file and writing it back as an edge image, uses the

[6] Actually, the image descriptor has a standardized Sun-rasterfile format and must be used for reading picture frames. It contains more information than only the bounds, however the algorithms use only the bounds information and the image descriptor can be reduced to carrying only that type of data.

[7] Dynamic arrays can be viewn as lists without the insertion operation. As long as we refrain from inserting elements in an array, dynamic arrays allow often more efficient implementations than lists do.

standard routines written in C. For Sisal this means that the main program, that includes I/O, is written in C. Sisal code is called from C using the mixed language interface provided by OSC.

5.0 Benchmarks

The goal of the presented benchmarks is to compare the best available sequential code generated by C compiler, and code generated by Sisal compiler. This includes the usage of best compiler options for both languages and using the best available compilers: GCC 2.4 for C and OSC 12.8 for Sisal. Generally, the programming of benchmark loops in Sisal may falsify the results by introducing unnecessary dependencies or even removing completely the loop code as part of the optimization pass. Therefore, the dataflow graphs generated by the Sisal compiler have been checked with the IFBrowser tool [MitMur91] before running the benchmarks.

We have used two kinds of data to run the benchmarks. On the one hand artificial images generated outside the benchmark loop have been given to the algorithm and on the other hand an image has been loaded from the disk as shown in figure 6. The latter has been used for comparing the performance on a sequential machine.

5.1 Comparing the Sources

The Gaussian smoothing and Canny edge detector are examples of mathematically formulated transformations. In Sisal the usage of arrays is supported by high-level constructs and range-checking options at run-time. There is little array support in C, either at write- or at run-time. As the constructs have to be written by the user and because there is almost no consistency check, the C-code tends to be longer. The C version of Gaussian smoothing and Canny operator has 600 lines of code and the time to write and debug it should be about a week. The Sisal version is about 300 lines long and it takes about 2 days to write and debug the code.

The image compilation algorithm has more than 600 lines in Sisal and less than 500 lines in C. The reason for larger Sisal code is, that most parts of the algorithm consist of operations on dynamic lists and because of many *if-then* constructs which are typically longer and more complex in Sisal than in C[8]. This algorithm took the most time to implement because its mathematical description is not clear. After several attempts with different approaches, the Sisal algorithm converged to the solution presented in [Klein87] which was the base of the used C-algorithm.

[8] Sisal *If-Then* constructs must always include an *Else* clause so that the same number of values is returned from the expression regardless which condition is fullfilled

The main conclusion from our experiences is that Sisal programmer should concentrate on the abstract mathematical form and use it as the base for implementing algorithms in Sisal. The existing algorithms in imperative languages can be used only to verify the results or to better understand the problem.

Another important advantage of Sisal programs is that Sisal programs do automatically allocate and deallocate data structures. The user is not bothered with the problems of disappearing memory or dangling pointers to structures that have been deallocated too early. This feature shortens much the coding and debugging cycle.

5.2 Comparing the Performance

Sisal is a language designed for programming parallel computers. Nonetheless we do measure and compare the performance on a single-processor computer, because not only the speedup of a program is important. The code generated by a language must not only deliver a good speedup when compared to itself but when compared to the best available sequential code. A good sequential performance shows also that the language is suited for being used on single-processor machines.

For the performance measuring on the single-processor machine SUN SPARC-Station, we read the image from figure 6 into the noise reduction and edge extraction algorithms and did 20 iterations with both algorithms. Figure 8 shows the measured execution times and the used memory.

Sisal	63 sec.	9620kB
C	70 sec	7160kB

Figure 8: Comparing execution time and memory requirements on a single-processor machine

Figure 8 shows that Sisal code is somewhat better than its C counterpart. This is insofar astonishing as the C-code, being heavily used at the Computer Vision Laboratory, has been already optimized manually and the code is very good. After looking at the generated IF2 graphs we conclude that this stems from the fact that the coding of mathematical transformation to C-programs leads to a loss of information, which the C-compiler is not able to recover and therefore cannot use for optimizing code. In our case this means that Sisal is able to prefetch much more from the innermost loops than best C-compiler are able to do.

We run the same two algorithms with an artificial image (requiring no I/O) on the 4 processor SGI machine. Figure 9 shows the execution times and the derived speedup.

	1 proc.	2 proc.	3 proc.	4 proc.
Time	31.7	16.86	11.98	8.26
MFlops	2.63	4.95	6.97	8.26
Speedup	1	1.88	2.65	3.14

Figure 9: Measuring performance on a multiprocessor

The third algorithm is a special one as it represents a domain of programs that use mostly dynamic list manipulations and have very little parallelism. The performance of Sisal code (using arrays to hold lists of segments) is typically slower by 50%. It depends on what kind of edge-image we are using, i.e., how much copying of segments from one image row to the next one occurs. However, the third algorithm needs less time than the first two, so that the performance of the whole program is similar to C program.

6.0 Summary

We do not want to overestimate the results of the benchmarks but we can certainly state that the generated Sisal code is competitive, although it is a functional language with much higher level of abstraction than C is. On contrary, by providing more information about the structure of the algorithm instead of mapping it on the hardware -- which is what C-programmers do -- the compiler is able to provide better code. It is also possible to formulate algorithms in Sisal, then look at the generated IF2 graphs and thus understand how they can be mapped on the machine efficiently: the structure of IF2 graphs can be used to improve the structure of a program in an imperative language.

The Sisal version of the noise reduction and edge extraction algorithms runs excellently on single processors and provides good speedup on parallel machines without any changes to the sources. Together with the short coding and debugging time, Sisal outperforms other imperative languages like C.

The lack of true dynamic lists, i.e., structures that allow efficient element insertion, leads to inefficient implementation of the image compilation algorithm. For any algorithm that is mainly based on manipulating complex data structures, Sisal does not perform sufficiently well.

Finally we can say that there are two reasons why we even on a single processor machine should use Sisal instead of an imperative language: shorter development time and better performance. Some algorithms may be still kept in C because they cannot be efficiently formulated in Sisal. For those algorithms, as well as for I/O, Sisal provides a mixed language interface.

7.0 Acknowledgments

We would like to thank Lawrence Livermore National Laboratories for running the code on their SGI parallel machine and especially John Feo, Pat Miller and Tom DeBoni for helping us with advice and actions. Also we would like to thank our Profs. A. Kuendig and O. Kuebler for their support.

8.0 References

[Cann92] David Cann. *Retire Fortran? A Debate Rekindled.* Communications of the ACM, August 1992/Vol. 35, No. 8.

[CannFeo90] David Cann, John Feo. *SISAL versus FORTRAN. A Comparision Using the Livermore Loops.* Lawrence Livermore National Laboratory Report, 1990.

[Canny86] J.Canny, *A Computational Approach to Edge Detection*, IEEE Trans. Pattern Anal. Machine Intell., 8(6), pp. 679-697,1986.

[FeoCaOl90] John T. Feo, David C. Cann, Rodney R. Oldehoeft. *A Report on the Sisal Language Project.* Lawrence Livermore National Laboratory, January 5, 1990.

[Klein87] F.Klein, *Vollstaendige Mittelachsenbeschreibung binaerer Objekte mit euklidischer Metrik und korrekter Geometrie.* PhD thesis, ETH Zurich, Switzerland, 1987.

[KleKueb87] F.Klein and O. Kuebler, *Euclidean distance transformations and model-guided image interpretation*, Pattern Recognition Letters 5, 1987, 19-29

[McGrSk85] J. R. McGraw, S. K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. *SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2.* Lawrence Livermore National Laboratories, Livermore CA, March 1985

[MitMur91] S. Mitrovic and St. Murer. *A Tool to Display Hierarchical Acyclic Dataflow Graphs.* Proceedings of the International Conference on Parallel Computing Technologies, September 1991, Nowosibirsk USSR, World Scientific Publishing, p. 304-315

[Mitrov93] S. Mitrovic. *Compiling Sisal for the ADAM Architecture.* TIK-Schriftenreihe Nr. 2, 1993 VDF, ISBN 3 7281 2051 0.

Compilation of Sisal for a High Performance Data Driven Vector Processor

W. Marcus Miller, Walid A. Najjar, and A. P. Wim Böhm

Department of Computer Science
Colorado State University
Fort Collins, CO 80523

Abstract

Although the dataflow model has been shown to allow the exploitation of parallelism at all levels, research of the past decade has revealed several problems: Synchronization at the instruction level has precluded the exploitation of locality, and lack of support for efficient aggregate data structure access results in poor vector and array performance. Many novel Hybrid von-Neumann Data Driven machines have been proposed to alleviate these problems. Current studies suggest sufficient locality is present in dataflow execution to merit its exploitation. In this paper we present a data structure for exploiting locality in a data driven environment: the Vector Cell. *A Vector Cell consists of a number of fixed length* chunks *of data elements. Each chunk is tagged with a presence bit, providing intra-chunk strictness and inter-chunk non-strictness to data structure access. We describe the semantics of the model, an instruction set and a processor architecture as well as a Sisal to dataflow vectorizing compiler back-end. The model is evaluated by comparing its performance to those of both a massively parallel fine-grain dataflow processor employing I-structures and a conventional pipelined vector processor. Results demonstrate the model is surprisingly resilient to long memory and communication latencies, and effectively exploits underlying parallelism across multiple processing elements.*

1 Introduction

The classical dataflow model provides instruction level support for the exploitation of all forms of program parallelism [1]. Classical dataflow architectures employed large name spaces coupled with instruction level synchronization, allowing the processor to tolerate and mask unpredictable latency due to memory access and inter-processor communication [32]. Although the dataflow model has been shown to be a viable contender in the arena of general purpose parallel architectures [15], research of the past decade has revealed several fundamental implementation problems. These include:

- A significant number of instructions executed by classical fine-grain architectures, including coloring and re-labeling, represent non-compute overhead [13, 28].
- Token matching represents a substantial bottleneck in the dataflow circular pipeline. Purely fine-grain execution has precluded the exploitation of program and data locality, and inherently sequential threads of code give rise to pipeline stalls due to matching latency [16, 24, 29, 20].
- Efficient handling of aggregate data structures has been hampered by costly synchronization at the data element level. Fine-grain execution and context switching has precluded the exploitation of pipelined vector hardware [12, 22].

The recent advent of *multi-threaded* [33, 30] or hybrid von Neumann-Data Driven architectures arose from a desire to combine the most salient features of both coarse grain von Neumann and fine-grain Data Drive models. Multithreaded architectures mask memory latency by taking advantage of fine-grain parallelism without the overhead of instruction-level synchronization inherent in traditional data driven processors. This is accomplished by increasing task granularity from one to multiple instructions. The major objective has been to reduce or eliminate unnecessary synchronization costs through simplified operand matching schemes and increased task granularity [16, 28, 14,

[0]This work is in part supported by NSF Grants CCR-9010240 and MIP-9113268

25, 7]. Recent studies indicate sufficient locality (both spatial and temporal) is present in dataflow execution to merit its exploitation [20]. Exploiting locality may increase latency as coarser grain instructions require longer times for multiple inputs to arrive, however results reported in [23] indicate that for the inner loops of many scientific codes, a coarse grain model of execution will not substantially impact latency.

In contrast to multi-threading, we propose to exploit available data structure locality through the pipelined execution of coarse grain instructions. In this paper we present a hybrid data structure, the *Vector Cell* or V-cell designed to exploit pipeline parallelism in the dataflow model. Task granularity is increased over classical dataflow machines by allowing more data elements per instruction. As a consequence overall program overhead is significantly reduced due to decreased matching operations and the exploitation of data structure locality. In this model, vectors are operated on in small fixed size segments or vector *chunks*. This segmentation allows for the non-strict production and consumption of vector elements, thereby decreasing load and store latencies. In the processor, data is pipelined into vector functional units, thereby exploiting data locality, and synchronization cost is reduced to the matching of vector handles (pointers). We describe the semantics of this model, an instruction set and a processor architecture as well as a dataflow vectorizing compiler back-end. The model is evaluated by comparing its performance to those of a massively parallel fine-grain dataflow multiprocessor employing I-structure memory and a conventional tightly coupled MIMD vector supercomputer (the Cray C90). The results indicate the V-cell model is capable of a significant reduction in runtime overhead when compared to a massively parallel fine-grain multiprocessor. We show that the model substantially reduces both synchronization (matching store) and non-compute instruction overhead and simulation results indicate the model achieves a factor of 2 to 5 reduction in execution time over a fine-grain dataflow model using 7 times fewer execution units. Overall, a 40% improvement in floating point rates when compared to the Cray C90 is indicated. Furthermore, the pipelining of vector chunks makes the model surprisingly resilient to long memory latencies.

The token is the basic computational mechanism in the classical dataflow model, however it is not well suited for the efficient support of large aggregate data structure accesses. The content of entire array could be placed on a single token, however this would be costly in terms of matching store bandwidth and at odds with goal of tolerating latency through efficient context switching. If large numbers of data tokens are injected into the network by the structure store as a result of an update operation, the matching store will become saturated. Several approaches have attacked the problem by reducing or eliminating the copying of large volumes of data during updates. To reduce matching store traffic several models have been proposed to increase instruction granularity.

The RMIT [9] CSIRAC II dataflow architecture supports generic functions with specific type coercion and strongly typed variable length tokens including vectors. Structure stores are integrated in the processing elements permitting extended structure functions such as block copying, accumulators and vector operations. The USC Decoupled Multilevel Data-Flow Execution Model [10] exploits macro actors and vector instructions. The SIGMA I dataflow machine [31] has iterative instructions both of the proliferate and fetch type. The DFC (single assignment C) compiler only generates the fetch type instructions because of serious pipeline bubbles caused by the proliferates.

The redesign of the original Manchester Dataflow Architecture using current supercomputer technology is discussed in [17]. The original processing ring is supplanted by a modified processing ring in which all subsystems operate synchronously in pipelined mode. Matching store and node store operations are performed in parallel, and a multi-channel instruction distribution unit feeds instruction packets to a six function pipelined ALU.

In [34] Multi-Threaded Vectorization is used to broaden the range of vectorizable code while retaining conventional vector machine efficiency. The proposed architecture may be viewed as a hybrid between a vector processor and a VLIW machine.

The remaining paper is organized along the following lines: The semantics of our hybrid model are presented in Section 2 along with the instruction set, processor architecture and compiler back-end. A comparative performance evaluation in which we compare the performance of our model against a massively parallel dataflow architecture employing I-structure memory is reported in Section 3. The performance of our model relative to that of a conventional pipelined vector supercomputer is reported in Section 4. Section 5 concludes the paper.

2 The Vector Cell Model

The I-structures [2] model is ideally suited for accesses where the production and consumption patterns of data structures are highly irregular and indeterministic. However, when the patterns are highly regular, as in most vectorizable scientific codes, the I-structure model suffers from an inordinate amount of overhead: its fine grain synchronization, at the element level, prevents the efficient exploitation of data structure locality and the pipelining of vectorizable operations. To enable the exploitation of locality and provide for non-strict pipelined execution semantics, we introduce a hybrid data structure consisting of a number of fixed length *chunks* of data elements. We refer to this structure as a *Vector Cell* or *V-cell*. As opposed to I-Structures, where each individual data element is tagged by a presence bit, only chunks are tagged by presence bits in V-cells. Access to data is therefore strict at the intra-chunk level and non-strict at the inter-chunk level. This allows for the exploitation of vector consuming and producing operations that take chunks as input and produce scalars or chunks as results. Vector cells retain the advantages of traditional I-structures: inexpensive context switching and synchronization at the instruction level allows long memory latencies to be masked. V-cells add the advantage of providing a mechanism by which data structure locality may be exploited using vector instructions on pipelined vector hardware. The affects of this mechanism are the reduction of matching store and network communication costs. This model may be viewed as a hybrid between traditional von Neumann vectors and the I-structure model. A presence bit associated with each chunk is set only when all the elements of that chunk are available in the structure store. Read access to a chunk is split-phase, as with I-Structures, it is deferred until the presence bit is set. If we let k denote the vector chunk length, an array of n elements can be viewed as a set of $\lceil n/k \rceil$ chunks.

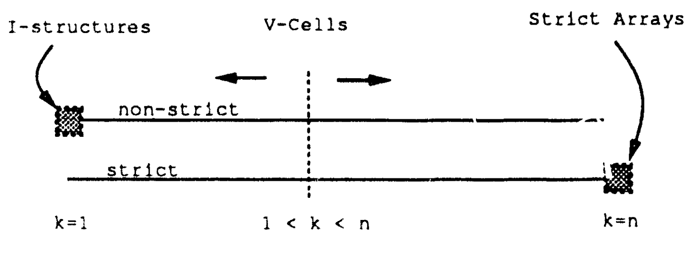


Figure 1: I-Structures, V-cells and strict arrays

By varying the chunk length, the V-cell model can span a continuum from I-structure cells on one extreme, where $k = 1$, to traditional strict arrays, where $k = n$, as shown in Figure 1. By measuring the effect of chunk length on access latency and throughput we determine optimal chunk lengths in terms of measured input latency, pipeline startup delay, and total program execution time.

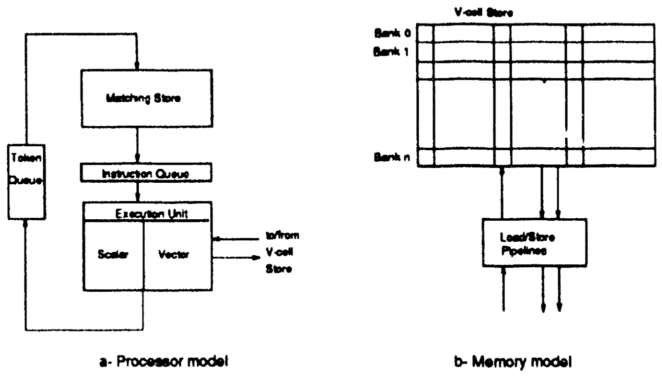


Figure 2: Processor and memory models

The vector cell storage model is coupled with a pipelined load/store architecture, where all instructions except load and store types operate on vector or scalar registers. The architecture of a processor supporting the V-cell model is depicted in Figure 2. The vector processor consists of two basic stages: a matching stage, containing a token matching store, and an execution stage. The execution stage is similar to a traditional vector processor containing a scalar and a vector execution unit. The vector execution unit consists of multiple pipelined functional units, load/store pipelines and a vector register set. The operations follow the basic dataflow execution: tokens destined to the same operation, vector or scalar, are matched in the matching stage. When all the input tokens are available, an instruction packet is formed and queued for execution. A token consist of a tag identifying its context and a data field. In scalar tokens the data field contains the actual data value while in vector tokens it contains the identifier of the vector register containing the vector operand.

2.1 Instruction Set Architecture

Vector elements in the V-cell model are stored in adjacent locations in the structure store and are accessed by a structure store *handle*. In the processor, a chunk element resides in a vector register. A chunk handle is either a structure store pointer consisting of a base address and a displacement or a vector register identifier. In the current implementation, vector cell memory consist of an allocation directory and a linearly addressable memory. Directory entries are indexed by chunk address and consist of a chunk length descriptor and a set of presence bits.

Vector Opcode	Operands	Result	Function
RSSV	offset,@a	handle of V_i	Read a single chunk from V-store into V_i, possibly deferred.
WSSV	offset,@a	handle of V_i	Write vector chunk in V_i to V-store at address @a, set presence bit.
ADRV	V_i,V_j	handle of V_i	Add vector V_i to V_j, result is stored in V_i.
SBRV	V_i,V_j	handle of V_i	Subtract vector V_j from V_i, result is stored in V_i.
MLRV	V_i,V_j	handle of V_i	Multiply vector V_i and V_j, result is stored in V_i.
DVRV	V_i,V_j	handle of V_i	Divide vector V_i by V_j, result is stored in V_i.
SUMRV	V_i,R_j	$R_j + \sum(V_i)$	Sums the elements of V_i, then adds result to scalar in register R_j.
PRDRV	V_i,R_j	$R_j * \prod(V_i)$	Forms product of elements of V_i, then multiplies result by R_j.

Table 1: V-cell vector instruction set

Vector instructions take a combination of vectors and scalars as input, producing a vector or scalar result. The vector instruction set is summarized in Table 1. Arithmetic vector instructions that do not return a scalar result return a vector chunk handle which is the identifier of the vector register containing the result. In addition to the arithmetic instruction types listed in Table 1, variants of the ADDV, SBRV, MLRV, and DVRV operations are supported in which one operand is a vector registei and the other a scalar value.

2.2 Compilation and Vectorization Strategy

The Manchester Dataflow Machine compiler [3] was modified to generate the appropriate data driven vector code for the V-cell model from Sisal programs [18]. We refer to this as Vector Dataflow Code or VDC. The approach taken by the vectorizing compiler is straight forward: the operations within innermost Sisal *ForAll* loops are vectorized with no vectorizing optimizations or loop unrolling.

To illustrate the production of vector dataflow code a simple example is presented. Code produced by our vector compiler for the computation of the inner loop of Livermore Loop 3 is illustrated. The code is for the computation of a scalar inner product $\sum_{i=1}^{n} V_1(i) * V_2(i)$. Figure 3.a shows the Sisal source for this kernel and Figure 3.b depicts the resulting vector dataflow code. V-cell execution is based on the matching of vector handles, not individual data elements. Vector elements are pipelined to and from vector registers from V-cell memory and all arithmetic vector instructions operate exclusively on vector register operands.

In Figure 3.b the PRLV vector instructions are used to produce a stream of $(\lceil n/k \rceil - 1)$ vector handles (double reals in the case of the RSSV targets and boolean for the BRR operator) indexed with $k, 2k, 3k, ...,$ where k is the vector chunk length. The read structure store vector instruction (RSSV) is used to read a single vector chunk from the structure store and return a vector register handle to the target instruction. The SUMRV instruction sums the elements in the vector chunk referenced by its right input handle, adds this to the scalar value at its left input and increments the index by k. The BRR instruction sends its left input to the right output if its right input value is true, otherwise the result is sent to the left output.

Translation to VDC is accomplished in two phases: lexical analysis, parsing, and translation to IF1 (Intermediate Form 1) are performed by the Optimizing Sisal Compiler front end. Vectorization analysis and code generation occurs in the final phase. During translation to VDC, the VDC compiler only attempts to vectorize inner most ForAll loop bodies.

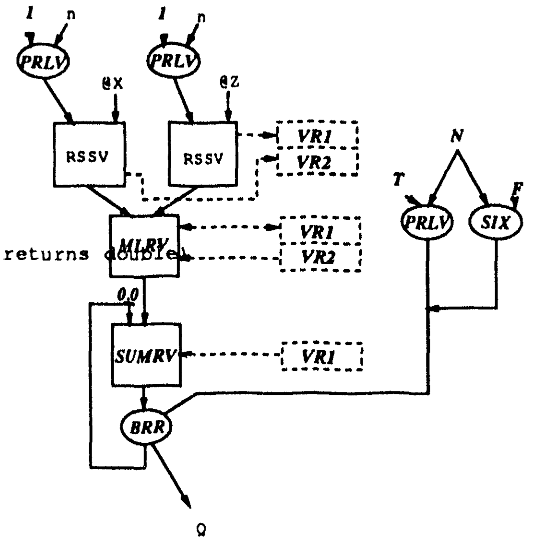
3 Performance Compared to a Massively Parallel Fine Grain Dataflow Processor

In this section we evaluate the performance of the V-cell model relative to a massively parallel fine grain dataflow processor. The evaluation is carried out in two stages: in the first we compare the execution of a data driven vector multiprocessor to that of a massively parallel fine-grain dataflow architecture employing I-structures. The objective is to evaluate the advantages derived from exploiting data structure locality, and pipelined execution in the V-cell model. In the second stage we evaluate the resilience of both models to increases in network and memory access latencies. We assume a worst-case allocation of tasks which is modeled by imposing a latency cost of d cycles to every token.

Objectives. In this section we evaluate the performance of data driven vector execution relative to a

```
type double = double_real;
type OneD   = array[double];
function loop3(n:integer; X,Z: OneD returns double)
    for i in 1,n
      Q:= X[i] * Z[i]
    returns value of sum Q
    end for
 end function
```

a.



b.

Figure 3: Data driven vector code to compute inner loop of L3

massively parallel fine-grain dataflow execution using I-structures. A vector processor is significantly more complex than a fine-grain dataflow processor, but because it can exploit data structure locality, fewer processors are needed. The first objective is to evaluate the tradeoff between processor complexity and the number of processors. Since the objective of both the V-cell and the I-structure model is to mask the effects of memory access and inter-processor communication, it is also important to evaluate the effects of additional network latency on the execution of both models.

Benchmarks. A total of twenty three benchmarks were selected for comparisons consisting of: ten from the *Lawrence Livermore Loops* [19, 11], six benchmarks from the *Purdue benchmarks* [26, 27], and four kernels from the *Linpack* routines [8]. Three additional programs, *Simple*,*Bmk11a*, and *Hilbert* codes were also benchmarked. Refer to Table 2 for benchmark statistics. Throughout the rest of the paper, Li and Pj are used as abbreviations for the Livermore and Purdue benchmarks i and j respectively.

All benchmark programs are written in Sisal. Data driven vector code was generated by the VDC backend to the Manchester Dataflow compiler. Fine grain codes were generated by the Manchester Dataflow Machine compiler [4]. Both the Manchester compiler and our vectorizing dataflow compiler were configured to generate code for non-strict structure store accesses.

Model	Instruction	Issue	Result
scalar (MC88110)	fadd/fsub	1	3
	fmult	1	3
	fdiv	13	13
vector (Cray C90)	faddv/fsubv	1	6
	fmultv	1	7
	fdivv (reciprocal approx.)	1	20

Table 3: Scalar and vector instruction latencies

Methodology. Both models were simulated on a cycle-level discrete-event simulator with a variable number of processors. The Cray C90 was used as a reference for all vector instruction latencies and the Motorola MC88110 [21] for all scalar instructions (Table 3). Both massively parallel and vector codes were run with the same matching store parameters and latencies: 1 cycle for a failing match and 2 cycles for a successful match (this corresponds to the matching times in the EM-4 [29]).

Minimal execution time and cost. The objective of this experiment is to evaluate the tradeoff between the complexity of a processor and the number of processors. We assume an ideally optimal allocation of tasks to processors. This is modelled in our simulation by setting a constant memory access time and the cost of inter-processor communication to zero.

In both models, the benchmarks were run with increasing numbers of processors until no speed-up was

Benchmark	Sisal Source Lines	Problem Size	Description
Livermore Loops	426	1000	Scientific/Numeric Kernels
Purdue Benchmarks	362	100-1000	Scientific Kernels
Linpack Routines	212	1000	Linear Algebra
Simple	1555	10x10	Hydrodynamics
Bmk11a	1007	64	Particle Transport
Hilbert	567	200x200	Hilbert Matrix/Linear Algebra

Table 2: Benchmarks

observed. This minimum execution time is denoted by T_{fmin} in the fine-grain model and T_{vmin} in our hybrid model. The results of these simulations are depicted in the histograms of Figures 4 and 5. All timing measures are in machine cycles, the subscripts f and v refer to fine-grain and vector executions respectively. T_{f1} (T_{v1}) is the execution time of each benchmark on a single processor. N_f (N_v) indicates the number of processors at which the minimum execution time T_{fmin} (T_{vmin}) was reached and FPR_f (FPR_v) is the floating-point rate in operations per cycle in the fine-grain (hybrid vector) model. Figure 4.a plots the ratio of fine-grain single processor execution times and V-cell single processors execution times (T_{f1}/T_{v1}) for each benchmark. Figure 4.b depicts the ratio of fine-grain minimum execution times and V-cell minimum execution times (T_{fmin}/T_{vmin}) for each benchmark. The relative number of processors required to achieve an execution time of T_{min} is shown in the histogram of Figure 5.a, and the floating point rates ($FLOPS/cycle$) attained by each model at T_{min} are reported in Figure 5.b.

A vector chunk length of $k = 32$ was used in V-cell model simulations. This chunk length does not necessarily coincide with the minimum execution time for all benchmarks, on the average the minimum occurs between 32 and 256 elements per chunk for a problem size of 1000.

The results from these experiments can be summarized as follows:

- *Reduction of run-time overhead.* The fine-grain model incurs a run-time overhead in synchronization (matching) and instruction scheduling for every scalar operation. By exploiting data structure locality the vector model eliminates a large fraction of this overhead. In fact, the vector model was capable of processing twice as many floating-point operations per cycle (FPR) as the fine-grain model: 0.38 floating point operations/cycle versus 0.17. The average processor utilization with unlimited resources was 10% in the fine-grain model and 28% in the vector, a threefold ratio. At minimum execution times, the vector model is, on the average, 2.63 (2.21 weighted by total cycles) times faster (T_{fmin}/T_{vmin}) than the fine-grain one. The total number of execution cycles (T_1) is 5.56 (2.25 weighted by total cycles) times larger in the fine-grain model than in the vector model.
- *Exploiting data structure locality.* On average, the fine-grain model required 72 processors to reach maximum speed-up while the vector execution required only 10. This means that without taking into account the increased cost of the interconnection network (which grows at least as $n \log n$) it takes about 7 times as many fine-grain scalar processors to outperform the vector multiprocessor.

Figure 6 shows the hybrid vector execution time profiles for some of the benchmarks. Note that the curves become flat between 4 and 32 processors. While these results are to be expected, our objective is to quantify the degree to which a chunked vector access strategy can enhance performance in a data driven model.

Tolerance of network latency. The objective of this experiment is to evaluate the resilience of both models to increases in network and memory access latencies. Since network latency increases as the number of processors increases, it is important to evaluate the effect of increased latency on both models. We model increased latency by imposing a fixed cost of d cycles to every token. This implies that all accesses to the structure store and all inter-processor communication would incur this delay.

The effects of the added delay on the average execution time of the benchmarks (assuming unlimited resources) is shown in Figure 7 with d increasing to 128 cycles: The effect is minimal on the vector model whereas the average total execution time of the fine-grain model increases linearly with increased latency.

The ratio of T_{min} for each model for a network access delay of $d = 128$ (worst-case) is depicted in the

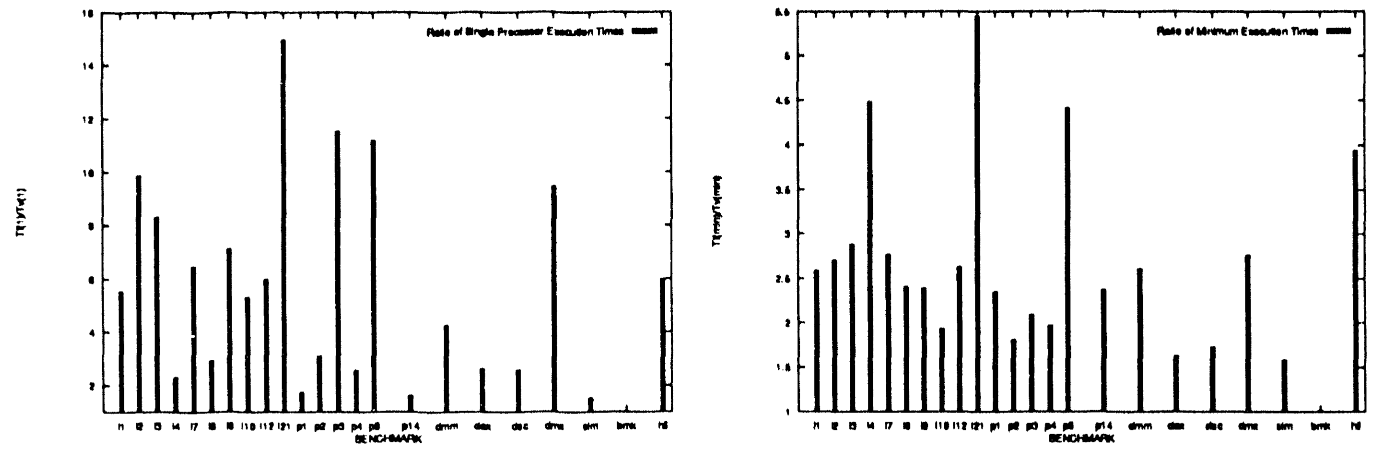


Figure 4: Ratio of T_1 (single processor) and T_{min} (minimum execution) times

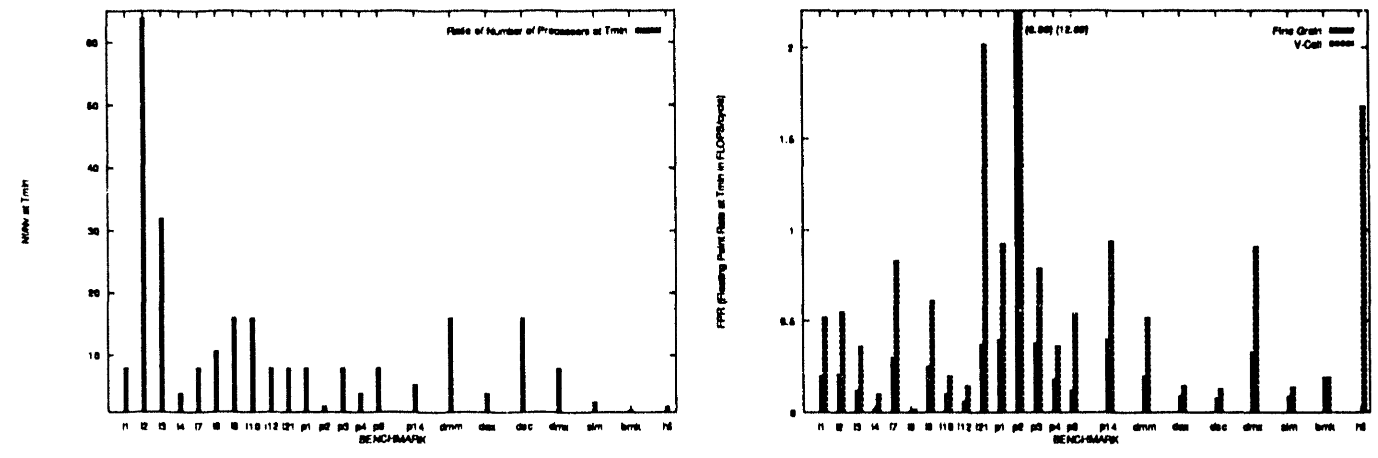


Figure 5: Ratio of number of processors at T_{min} and floating point rates in FLOPS/cycle

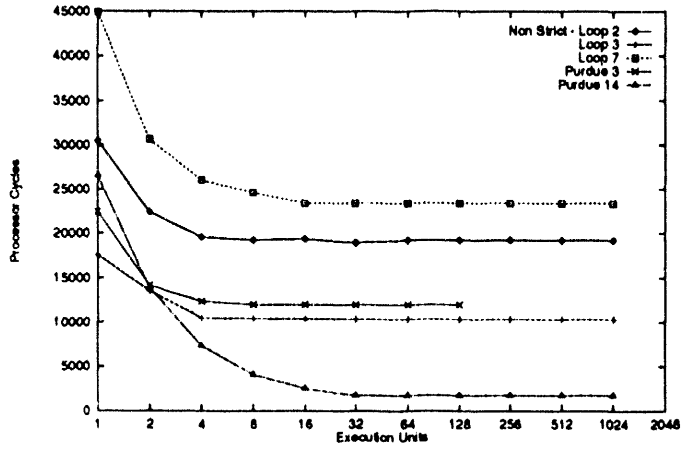


Figure 6: V-cell execution time versus number of processors

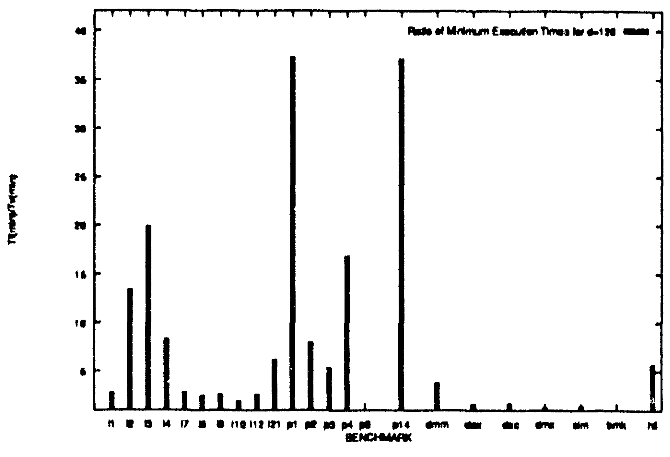


Figure 8: Ratio of fine-grain and V-cell minimum execution times for a network delay d=128

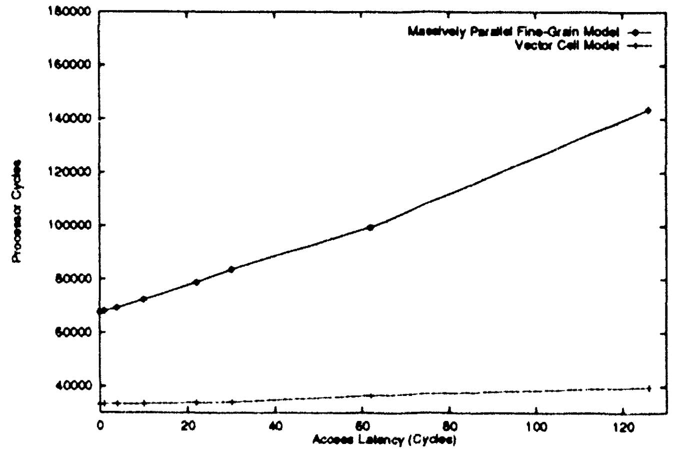


Figure 7: Average execution time for all benchmarks as a function of access latency d (unlimited resources)

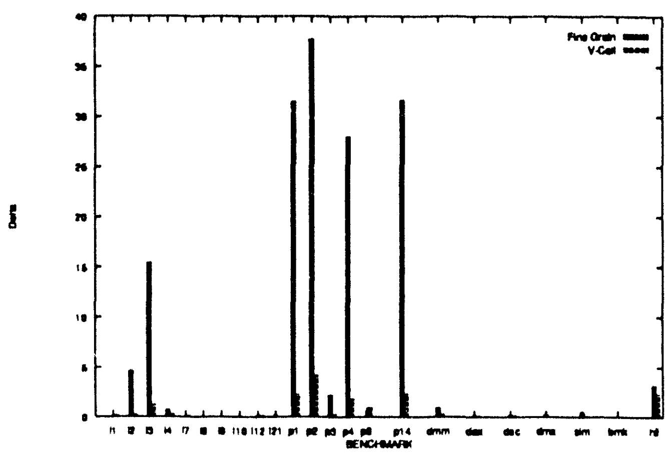


Figure 9: Access latency degradation factors Δ_f and Δ_v

histogram in Figure 8. The factors Δ_f and Δ_v measure the relative degradation in performance as a result of the added latency. They are defined by:

$$\Delta = \frac{T_{worst-case} - T_{best-case}}{T_{best-case}}$$

Figure 9 compares Δ_f and Δ_v for each of the benchmarks in the test suite.
To summarize, the results of latency tolerance experiments indicate:

- Because of the added latency, the average fine-grain execution time is degraded by 74% while the V-cell execution time is degraded by only 10%. This is due to the significantly lower number of tokens that are generated and communicated in the vector model relative to the fine-grain one. Further, the degradation in the vector model is much more consistent, i.e has a lower variance, than in the fine-grain model.
- For 10 of the benchmarks, there was no or very minimal ($\leq$ 10%) degradation in execution time due to longer latencies. Degradation in vector mode performance was observed to occur when the vectorized inner loop bounds were much shorter than the vector chunk length (e.g. L4, L21). Little reduction in matching overhead occurs in this case as the resulting vector chunk lengths were not long enough to substantially reduce the token traffic. Degradation in fine-grain performance was observed to be most significant in those programs containing reduction operations. Reduction operations sequentialize segments of the code, greatly reducing the available parallelism in the underlying program. In these cases program execution becomes limited by matching store bandwidth and the majority of execution units idle. Latency cannot be masked by context switching due to an insufficient number of parallel threads.

For programs in which the underlying parallelism was sufficient to sustain multiple execution threads, both the fine-grain and vector models were able to effectively mask latency. For inherently sequential code segments, the V-cell model was able to achieve greater performance than the fine-grain model. These results demonstrate that a vector based dataflow execution model suffers little from added communication and memory latency and is very resilient to increases in these factors. This is primarily due to a dramatic reduction in data memory token traffic and the exploitation of data structure locality.

4 Performance Compared to a Tightly Coupled MIMD Vector Processor

Objective. In this section we evaluate the performance of data driven vector execution relative to the Cray C90 architecture. The non-strictness of data structure access in the V-cell model can allow the masking of memory access latency: a chunk can be read before the whole array is written and therefore an efficient pipelining of producer/consumer codes is possible. In the Cray code this synchronization is most often done at the level of the whole array. On the other hand, the Cray C90 compiler is much more mature and sophisticated than our vectorizing back-end and is capable of vectorizing a higher fraction of operations. The objective of this section is to evaluate this tradeoff.

Benchmarks. We employ the same set of twenty three benchmarks in this comparative evaluation as was used in Section 3.

Methodology. The Cray C90 versions of these benchmarks were compiled using the Optimizing Sisal Compiler (OSC version 12.1) [6] with the following optimizations enabled: traditional scalar optimizations such as record fission, common subexpression elimination, constant folding, dead code removal, function inline expansion, loop fusion and optimization of the resulting dataflow graph through update-in-place analysis. Loop unrolling was disabled for these trials as our vector dataflow compiler does not perform this optimization. By employing highly aggressive optimizations at the intermediate code stages in conjunction with update-in-place analysis, OSC can produce code comparable in performance to FORTRAN on the Cray C90 and Cray Y-MP [5].

The vector dataflow compiler back-end we developed for V-cell code generation is based on an earlier version of Sisal compilation technology (Sisal $\rightarrow$ IF1 $\rightarrow$ VDC) than the Cray Sisal compiler (Sisal $\rightarrow$ IF1 $\rightarrow$ IF2 $\rightarrow$ C/FORTRAN), hence our compiler does not provide many of the scalar optimizations that the Cray compiler performs at the IF1/IF2 stages. Despite this, performance of our model compares favorably with that of the Cray C90.

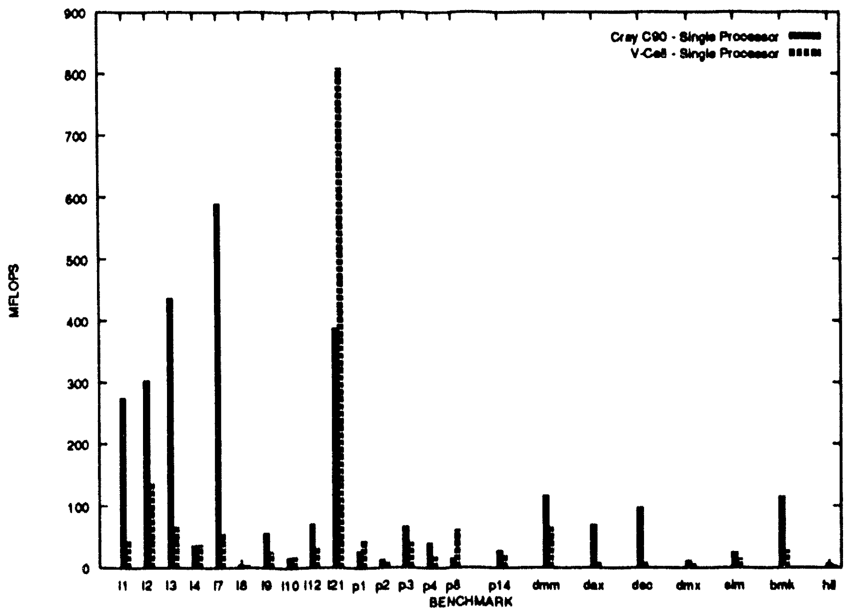


Figure 10: Single processor Cray C90 and V-cell performance

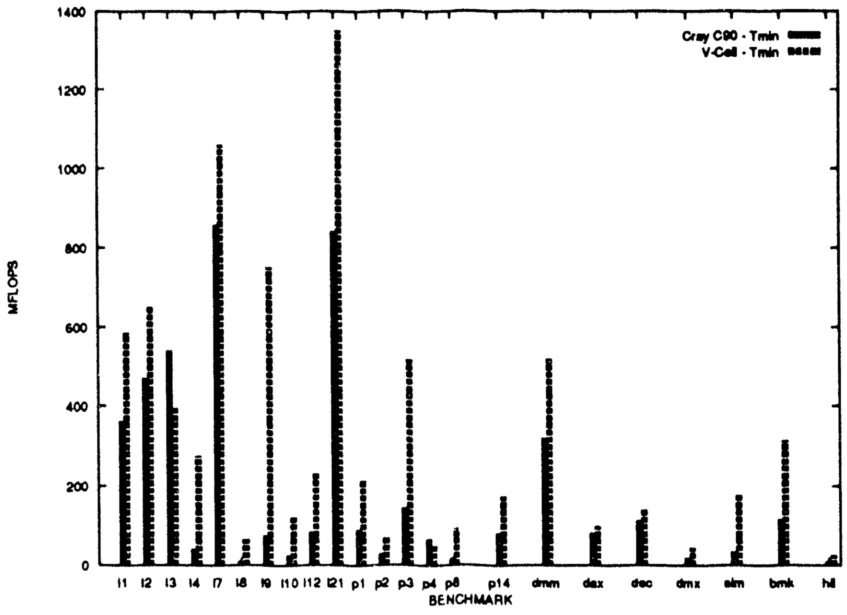


Figure 11: Multiprocessor Cray C90 and V-cell performance

The data driven vector code was executed on a cycle-level, discrete-event simulator that is configured with the same number of functional units and load/store pipelines as the Cray C90 and with the same issue and operation latencies. A chunk length of 128 elements, corresponding to the C90 vector register length was used.

Benchmark	Cray C90	V-Cell
L1	8	16
L2	12	16
L3	2	16
L4	2	8
L7	12	16
L8	16	16
L9	2	16
L10	16	16
L12	2	12
L21	16	16
P1	16	16
P2	16	16
P3	12	16
P4	8	8
P8	1	16
P14	16	16
DMM	16	16
DAXPY	2	12
DSCAL	2	16
DMXPY	12	16
SIMPLE	4	16
BMK11A	1	16
HILBERT	4	12
Ar. Mean	9	15

Table 4: Number of vector processors at maximum performance

Single Processor Execution. Results of experiments conducted to compare the performance of single processor executions are shown in the histogram of Figure 10. These plots compare the maximum MFLOPS rate attained by each architecture employing a single vector processor for each benchmark. For benchmarks restricted to execution on a single vector processor, the Cray C90 execution was superior to that of the V-cell model. The V-cell model only outperformed the Cray C90 on 3 of the 23 test codes, attaining an average performance of 67 MFLOPS. By contrast, the C90 averaged 122 MFLOPS over the test suite, or about 45% higher floating point rates in 48% fewer machine cycles the our data driven model. Note that the V-Cell performance of Livermore Loop 21 is significantly better than that of the C90. Loop 21 is a triple nested loop, in which the inner loop forms the sum reduction of the product of two arrays. The inner loop bounds are quite short (25 elements), making the resulting vector code very inefficient for the C90s relatively longer vector registers. V-Cell non-strict, inter-chunk semantics in tandem with run time (demand driven) scheduling allows for relatively high utilization of processor resources when short vectors are processed.

Multiple Processor Execution. Results of experiments conducted to compare the performance of multiple processor executions are shown in the histogram of Figure 11. These plots compare the maximum MFLOPS rate attained by each architecture employing multiple vector processor for each benchmark. For multiple vector execution units, these results indicate the V-cell model compares very favorably with the Cray C90 execution. The Cray C90 outperformed the data driven model in only 1 of the 23 benchmarks, attaining an average performance of 191 MFLOPS (53 MFLOPS weighted by the total number of machine cycles in the test suite) across the test suite. Although the Cray C90 supports 16 concurrent vector processing units, program partitioning and data allocation, coupled with communication overhead and barrier synchronization delays can actually cause performance to fall as the number of processing units is increased. The average number of processors required to obtain maximum performance on the Cray C90 for the test suite was 9. Refer to Table 4 for the number of vector processors at which minimum execution time occurred for each benchmark. Results of V-cell simulations indicate an overall performance of 361 MFLOPS (289 MFLOPS weighted by the total number of machine cycles in the test suite) or about a 47% advantage in floating point rates over the Cray C90 in 72% fewer machine cycles. The average number of processors required to obtain maximum performance in V-cell execution was 15.

The Cray C90 performed well on kernels containing low degrees of parallelism and those codes that are easily vectorized and chainable. For example, the C90 execution outperforms the V-cell model in Loop 3 which is an inner product of arrays of 10,000 elements. In this code the shortest execution time on the C90 is achieved with just two processors, this time is only 5% better than the single processor one. Our model was superior for codes involving reduction operations and kernels containing parallelism of degree greater than two. This may be attributed to the fact that in the V-cell model, load/store pipeline latency is effectively masked by the vector units ability to switch between a small pool of available vector chunks when elements from the current vector stream block. Moreover, in most vectorizable codes the ability to overlap floating point operations, memory latency and loop overhead results in substantial performance gains. The execution of both scalar and vector instructions may proceed in parallel in our model, allowing most of the non-compute loop overhead to be covered by actual floating point computations (for example, in the code of Figure 3.b, the execution of the scalar BRR instruction may be overlapped with the execution of the vector SUMRV instruction for all but the last chunk iteration).

Overall, V-cell model performance was superior to the Cray C90, Notably:

- Because of its non-strictness, the V-cell model can allow a more efficient pipelining between vector operations. This is especially true in producer/consumer type codes where the consumer code could start execution as soon as the first chunk is available. Further, when the production and consumption of data elements are out-of-order, the V-cell model can still provide the pipelining.

- The asynchronous nature of dataflow execution permits the dynamic exploitation of asynchronous instruction level parallelism over the set of vector and scalar functional units. Even though the Cray architecture can exploit parallelism among vector and scalar operations, a significant advantage was observed in loop bodies that exhibit even moderate degrees of instruction level parallelism.

- Although the granularity of instructions in our model is considerably larger than in classical dataflow models, the overhead associated with the run time scheduling of V-cell instructions on a single vector processing unit appreciably degrades floating point performance. In addition, the quality of the vectorized code in the Cray version of the benchmarks was always better or equal to that obtained from the VDC compiler back-end resulting in superior performance on the part of the Cray when executed on a single vector processor.

- The Cray C90 supports the *chaining* of vector instructions. The V-cell results presented in this section do not chain vector results.

5 Conclusion

In this paper we have presented and evaluated a data structure model designed for the exploitation of locality in the data driven paradigm. The Vector Cell or V-cell incorporates vectors in a hybrid dataflow/von Neumann model: vectors are stored in fixed length chunks across interleaved memory, and each chunk is tagged with a presence bit. The access to these vectors is non-strict at the chunk level but strict within a chunk. The advantage of this model over a conventional pipelined vector processor is that memory latencies are effectively masked by the split phase production and consumption of data structure elements. Compared to a massively parallel fine-grain dataflow multiprocessor, our model is able to effectively exploit data structure locality: In our model data is pipelined into vector functional units, thereby exploiting data locality, and synchronization cost is reduced to the matching of vector handles. This model can be seen as a hybrid strict/non-strict data structure and also as a hybrid between traditional vectors and I-structures. A set of vector instructions has been defined for this hybrid model, employing pipelined load, store and arithmetic units as well as a set of vector registers. A data driven architecture model for the processor and structure store supporting pipelined vector operations has been described. The performance of our hybrid model has been compared to both a massively parallel fine-grain dataflow architecture employing I-structure memory and a pipelined vector supercomputer through the use of 23 benchmark codes.

In comparing the V-cell model to a massively parallel fine-grain dataflow multiprocessor a three fold average speed-up is measured. The number of processors required to achieve minimum execution time is seven times smaller for our model than it is for the fine-grain model. Also, the total number of execution cycles is reduced by a factor of five due to the elimination of a large amount of synchronization (matching store) overhead. Both the fine-grain and vector hybrid models are able to effectively mask latency in programs in which the underlying parallelism is sufficiently large. The V-cell model exhibits extreme resilience to additional network and memory latencies. Overall the hybrid vector model suffers an average degradation of 10% on its execution time versus 74% for the fine-grain model, when a high access latency for every token crossing a processor or store boundary is assumed. A significant degradation in fine-grain performance was observed in programs where reduction operations determine the behavior, as these reductions limit the available parallelism. By contrast, in the vector model, vector operations effectively exploit data structure locality in the reduction operations through pipeline parallelism.

The results of benchmarks executed on the Cray C90 indicate that for equivalent codes, on the average 40% higher floating point rates are possible, with a 70% net reduction in execution cycles when V-cell operations are employed. This reduction is mostly attributable to the data driven models ability to more effectively schedule small tasks at run time in concert with non-strict data structure access semantics. This performance advantage was still evident even though the quality of the Cray vector code is superior to that produced by our vector dataflow compiler.

References

[1] Arvind and R.A. Iannucci. Two fundamentals issues in multiprocessors: the data-flow solutions. Technical Report LCS CSG 226-6, Laboratory for Computer Science, MIT, Cambrige Massachussett, May 1987.

[2] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. MIT Computation Structures Group Memo 269, Laboratory for Computer Science, MIT, 1987.

[3] A. P. W. Böhm and J. R. Gurd. Iterative instructions in the Manchester dataflow computer. *IEEE Trans. on Paralle and Distributed Systems*, 1(2):129–139, 1990.

[4] A. P. W. Böhm and J. Sargeant. Code optimization for tagged-token dataflow machine. *IEEE Trans. on Computers*, 38(1):4–14, 1989.

[5] D. C. Cann. Retire Fortran? A Debate Rekindled. *Communications of ACM*, 35(8):81–89, August 1992.

[6] D. C. Cann. The Optimizing Sisal Compiler: Version 12.0. Technical Report CS-92-114, Colorado State University, 1992.

[7] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1991.

[8] J. J. Dongarra. Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment. *Computer Architecture News*, 16:47–69, March 1988.

[9] G. K. Egan, J. J. Webb, and A. P. W. Böhm. *Some Architectural Features of the CSIRAC II Data-Flow Computer*. Prentice-Hall, 1991.

[10] P. Evripidou and J.-L. Gaudiot. *The USC Decoupled Multilevel Data-Flow Execution Model*. Prentice-Hall, 1991.

[11] J. T [illegible]o. The Livermore Loops in Sisal. Technical Report UCID-21159, Computing Research Group, Lawrence Livermore National Laboratory, Livermore, CA 94550, August 1987.

[12] J.-L. Gaudiot. Structure handling in data-flow systems. *IEEE Trans. on Computers*, C-35(6):489–502, June 1986.

[13] J.-L. Gaudiot and W. A. Najjar. Macro-actor execution on multilevel data-driven architectures. In *Proc. of the IFIP Working Group 10. 3 Working Conference on Parallel Processing*, Pisa, Italy, 1988.

[14] V. G. Grafe and J. E. Hoch. Implementation of the Epsilon dataflow processor. In *Proc. of the Twenty-third Ann. Hawaii Int. Conf. on System Sciences*, pages 19–29, 1990. volume 1.

[15] J. Hicks, D. Chiou, B. Seong Ang, and Arvind. Performance Studies of Id on the Monsoon Dataflow System. *Journal of Parallel and Distributed Computing*, 18:273–300, 1993.

[16] R. A. Iannucci. Toward A Dataflow/Von Neumann Hybrid Architecture. In *Int. Ann. Symp. on Computer Architecture*, pages 131–140, 1988.

[17] Y. Inagami and J. F. Foley. The Specification of a New Manchester Dataflow Machine. In *Proceedings International Conference on Supercomputing*, 1989.

[18] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and Iteration in a Single Assignment Language: reference manual version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[19] F. H. McMahon. Livermore FORTRAN kernels: A computer test of numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.

[20] W. M. Miller, W. A. Najjar, and A. P. W. Böhm. A quantitative analysis of locality in dataflow programs. In 24th *Int. Symp. on Microarchitecture (MICRO-24), Albuquerque, NM*, pages 46–53, Nov 1991.

[21] Motorola. *Second Generation RISC Microprocessor*. Motorola, INC., 1991. User's Manual, MC88110UM/AD.

[22] W. A. Najjar, A. P. W. Böhm, and W. M. Miller. A quantitative analysis of dataflow program execution – preliminaries to a hybrid design. *Journal of Parallel and Distributed Computing*, 18(3), July 1993.

[23] W. A. Najjar, W. M. Miller, and A. P. W. Böhm. An analysis of loop latency in dataflow execution. In *Int. Ann. Symp. on Computer Architecture*, Gold Coast, Australia, May 1992.

[24] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Int. Ann. Symp. on Computer Architecture*, June 1990.

[25] G. M. Papadopoulos and Kenneth R. Traub. Multithreading: A revisionist view of dataflow architectures. In *Int. Ann. Symp. on Computer Architecture*, June 1991.

[26] J. R. Rice. Problems to test parallel and vector languages. Technical Report CSD-TR 516, Purdue University, 1985.

[27] J. R. Rice. Problems to test parallel and vector languages - 2. Technical Report CSD-TR 1016, Purdue University, 1990.

[28] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a data-flow single chip processor. In *Int. Ann. Symp. on Computer Architecture*, pages 46–53, May 1989.

[29] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. Pipeline optimization of a dataflow machine. In J-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, pages 225–246. Prentice Hall, 1991.

[30] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-controlled multithreading for lenient parallel languages. In J. Hughes, editor, *Conf. on Functional Programming Languages and Computer Architecture*, 1991.

[31] S. Sekiguchi, K. Hiraki, and T. Shimada. Efficient vector processing on a Dataflow supercomputer SIGMA-1. In *Int. Conf. on Parallel Processing*, 1988.

[32] B. Smith. The end of architecture (keynote address). In *Int. Ann. Symp. on Computer Architecture*, May 1990.

[33] K. R. Traub. Multithread code generation for dataflow architetures from non-strict programs. In J. Hughes, editor, *Conf. on Functional Programming Languages and Computer Architecture*, 1991.

[34] Tzi-cker Chiueh. Multi-threaded vectorization. In *Int. Ann. Symp. on Computer Architecture*, pages 352–361, Gold Coast, Australia, May 1991.

Sisal on Distributed Memory Machines*

Santosh S. Pande

Computer Science Department
Ohio University
Athens, OH - 45701
sspande@monsoon.cs.ohiou.edu

Dharma P. Agrawal and Jon Mauney

Electrical and Computer Engineering Department
North Carolina State University
Raleigh, NC 27695-7911
dpa@ncsu.edu and mauney@csljon.csl.ncsu.edu

Abstract

Sisal, is targeted to Intel Touchstone i860 systems, by mapping the functional parallelism in its intermediate IF-1 representation. A new compile time scheduling method is developed that investigates a trade-off between the schedule length and the required number of processors. The compile time schedule is found using a new concept of threshold *of a task that quantifies a trade-off between the schedule-length and the degree of parallelism. At compile time one of the following scheduling goals is realized:*

- *Compiling for minimizing schedule length : Suitable for large systems.*
- *Compiling for processor requirements below a given maximum number of available processors in the system : Suitable for small systems.*

The run time system is designed to support call by value semantics on Intel Gamma, Delta, and Paragon, by minimizing the overheads. Each processor contains the inlined Sisal program's code and starts executing its own C main() corresponding to SisalMain(). The code that is specialized for a given processor is identified by a case statement that corresponds to the processor number, thereby making the start-up overhead as small as possible. The universally owned code is replicated on all processors. The processors exchange data values using asynchronous 'put' and 'pick' primitives to maximize the overlap of communication and computation.

1 Introduction

The distributed memory systems are becoming popular. However, programming distributed memory systems still remains very complex. The lack of proper support software like the compilers, debuggers, and the operating systems are the main concerns of many programmers.

One of the main reasons for the programmer's difficulties is that most of the compilers for this important class of multiprocessors demand that the users specify data partitioning and/or code mapping. Such data and code partitioning techniques largely dependent on user's judgement, are cumbersome, and may suffer from inaccuracy and poor execution speed.

Some of the semi-automatic compilers ask the user to partition either the data or the code of his/her program and automate the partitioning of the other aspect. The compilation methodology presented in this paper is fully automatic, and produces reliable and efficient code using data and code partitioning in an unified framework. This methodology has been incorporated in the Sisal (Streams and Iterations in a Single Assignment Language) compiler backend for producing code to run on Intel Gamma, Delta and Paragon family of multiprocessors.

Section 2 surveys the different program partitioning techniques proposed in the literature. Section 3 discusses and illustrates our compile time partitioning method through an example. Section 4 addresses the code generation and the run time system for distributed memory machines. Section 5 offers conclusions.

2 Program Partitioning Issues

The problem of program partitioning and scheduling on distributed memory multiprocessors has been attempted by many researchers, and the approaches can be mainly classified as:

1. Data Driven Code Partitioning, and,

[0]*This research was supported by the U. S. Army Research Office under grant no. DAAL03-91-G-0031

2. Code Based Data Allocation.

In data driven approaches, the program data is partitioned for different processors, and the code is produced so that the generated data references are locally available. This reduces the costly interprocessor communication on distributed memory machines. The code based approaches, on the other hand, carry out the partitioning so that each processor gets approximately an equal share of the program code. The resulting data references are then examined, and data is allocated to different processors to reduce the communication. The goals of locality of data and load balanced code can conflict, and for certain types of codes like the particle codes, are impossible to achieve simultaneously.

Kennedy et al. [6] follow the data driven scheme. They define language extensions to Fortran with functions for managing data distribution in non-shared address spaces. The new language is called Fortran D. They define compile time data domains to map the aggregate (mainly arrays) slices to local memory. The user is responsible for specifying the data layout. The compiler then supports a virtual address mechanism to correctly map the global references to the local ones. The code generation phase ascertains that the references in the computation are correctly mapped to the local memory.

Pingali et al. also use data driven code partitioning approach by using user specified data mapping at compile time. A compile time ownership analysis is carried out and the code is produced by employing the concept of evaluators and participators. The compile time data mapping mostly defines the ownership information required for the correct code generation. The compile time unresolved ownerships can be obtained using run time ownership resolution.

In compilation of loops, communication overhead can be reduced by using locality of reference and by sending and receiving data in blocks between different loop slices. In a recent work, a new loop transformation called *access normalization* is proposed that restructures the index sets of the loop to exploit both the locality and the block transfers of loop data [8].

The project C* [11] relies on user partitioning of data aggregates on SPMD machines like CM-5. For example, the data-parallel program can look like:

```
domain vector { real a, b, max; } x[100];
...
Select:
[domain vector].{
        if (a > b) then max = a;
        else max = b;
end Select;
}
```

The type "vector" defines a domain containing two real values named a and b. By declaring x[100], 100 instances of the variable pair are created one pair per processing element. The 'Select' statement activates every processing element whose instance has domain type "vector". Every processing element evaluates the statement a > b. The universal program counter enters 'then' clause and those PEs for which the statement is true, perform the assignment max = a. Then the universal program counter enters 'else' clause and those PEs for which the expression is false, perform max = b. This approach, thus, supports data parallel programming on SPMD machines.

Koelbel et al. [7] carry out a data mapping in their Blaze project and use many optimizations to reduce message passing overhead.

Amongst the code based data allocation approaches, the most notable is that of Mansour et al. [2]. It uses a load balanced code, and mapping data on the processors, to minimize the communication. The problem domain is decomposed into subdomains at compile time, and each partition is judged on the basis of an objective function that determines the locality of references.

Some researchers have also attempted combining both the above approaches. For special cases of DO loops with constant dependence distances, Ramanujan et al. [10] have devised a test that determines whether a loop can be split to achieve a communication free partition. Based on the test, an algorithm is developed that achieves communication free partitioning of a loop, if it exists. For example, consider the following 'for' loop:

```
for i = 2 to N
   for j = 2 to N
      A[i,j] = B[i-1,j]+B[i,j-1]
```

In the above loops, to compute each element of A[i,j], two elements of array B are needed. It can easily be seen that if both arrays A and B are divided along their anti-diagonals, a communication free partition of the loops is achieved. On the other hand, for certain types of loops, no such partition can be found.

Gajski et al. [3] address the loop partitioning problem on a distributed-shared memory system. A given loop partition is evaluated on the basis of the amount of parallelism, and the memory access and synchronization overheads. The memory access overheads are modeled on the basis of whether it is a local access (for read only variables), a local synchronized access (for read/write variables), or a network synchronized access (for non-local variables). A heuristic algorithm is used to reduce the number of loop partitions examined to determine the best one.

The above approaches save a lot of effort on the compiler's part. However, these approaches might not always yield a good program partition due to the following reasons:

- Data driven code partitioning approaches rely heavily on the user judgement in correctly partitioning the data.

- Some of the approaches treat the data distribution as static for the complete scope of the program and do not allow remapping of the data. Also, even if the user partitioning of the data is optimal at compile time, it may not be so at run time, due to compile time unknowns.

- Current data driven approaches tend to use the regularity of the computational structure to generate data and code partition. For example, the nearest neighbor communication in Jacobi, or communication in four dimensions in 2-D SOR used frequently in scientific computation, are used to drive the code generator by these compilers. These schemes are not sufficient to allow locality for more general irregular computational structures.

- Data driven code generation over-emphasizes the locality issue. The resulting code partition may be non-optimal. For example, an unbalanced code might result from a data driven code generation, if the preprocessing dependence analysis stage does not properly discover the distributed variables. Dependence analysis is extremely difficult and imprecise in an imperative framework leaving these approaches questionable.

- Strictly code driven approaches also suffer from the communication overhead, which could nullify all the benefits of parallelism.

- The approaches that combine both schemes, are too specific to constructs like DOALL, and also demand a special structure of the loop. In a general program, such approaches may not be viable, since many of the conditions that make use of the array indices may be unknown at compile time due to the presence variable index coefficients.

These limitations of the existing purely data or code based approaches prohibit fully automatic and efficient program partitioning and scheduling on distributed memory machines. The approach proposed in this work combines the data and code based approaches in a unified framework based on the functional parallelism present in a program. Functional parallelism is the parallelism amongst different operations carried out in a program, by following data and control dependences. Functional parallelism is also referred to as the DAG parallelism in the literature due to its most popular representation in the form of a Directed Acyclic Graph. In this work, the two terms - functional or DAG parallelism are used interchangeably, to suit the context.

2.1 DAG Parallelism And Functional Paradigm

One of the major limitations of the above approaches is that they deal primarily with the loop based parallelism. Loop parallelism is quite localized, and in general, there exists more general DAG parallelism in the full scope of the program, in an interprocedural framework. To exploit the high degree of parallelism available in newer distributed memory machines, the loop parallelism must be augmented with the general DAG parallelism, and hence the compiler must be capable of detecting and effectively mapping the interprocedural DAG parallelism onto processors.

A few research efforts have addressed the issue of DAG parallelism for imperative languages. Girkar [5] has specified HTG (Hierarchical Task Graph) as the intermediate representation for DAG parallelism. He has also developed a method to remove the redundant dataflow dependences and proved that in general the minimization problem for task dependences is NP-complete. Sarkar [12] uses a compile time cost model to analyze the trade-off between task overhead and task granularity.

Extracting DAG parallelism demands extensive interprocedural dependence analysis. Such analysis is very hard in the imperative framework due to the presence of aliases, side-effects, and common blocks. The functional paradigm offers a more clean and neat model of DAG parallelism in the form of dataflow graphs where, a node represents computation, and an edge carries the values from one node to another. One of the arguments against the functional programming is the lack of efficiency. However, the recent success in very efficient compilation of functional languages have led them to outperform conventional languages like Fortran in terms of many efficiency issues like execution speed and code-size.

This work, therefore, concentrates on the problem of scheduling functional parallelism in the form of a DAG, on distributed memory multiprocessors.

Amongst the different approaches proposed in the literature on DAG scheduling, most notable is due to Gerasoulis *et. al.* [4]. They have proposed Dominant Sequence Algorithm (DSC) that is $O((v + e) log\ v)$. DSC calculates a priority of each of the nodes at every step and schedules a highest priority node whose all predecessors have been scheduled.

However, the issue of mapping tasks on distributed memory machines as a trade-off between the schedule length and the number of required processors remains unaddressed. We believe that it is very important to address this issue in order to fully use the power of the distributed memory machines that could have small as well as large number of processors. This issue along with the necessary run time system form the subject of this paper.

To demonstrate this approach, Sisal (Streams And Iterations In A Single Assignment Language) has been selected, due to its clean semantics and an elegant functional representation in its intermediate form. A task model based on dataflow graph representation of Sisal programs is used. The dataflow graphs are mapped to different processors, and a partition is generated at compile time. The efficient code generation method along with the run time system keep the overheads low to effectively use the parallelism at run time.

3 Compile Time Partitioner

The partitioning problem for a general DAG (Directed Acyclic Graph) of task representation of a program, is known to be strong NP-hard thereby ruling out the possibility of a pseudo-polynomial algorithm. Several variants of a new heuristic algorithm have been developed for partitioning Sisal dataflow graphs on i860 based Intel Gamma, Delta and Paragon (Refer to [1] for details about Sisal).

3.1 Sisal Compiler Modifications

The first phase translates a program into IF-1. The second phase IF1Ld of the compiler combines the modules of a program into a monolith IF-1 program. IF1Opt is an optimizing phase of the compiler that carries out standard dataflow optimizations such as common subexpression removal, loop invariants removal, loop fusion, and constant folding. IF2Mem allocates abstract memory locations to IF-1 nodes, producing an augmented form called IF-2. IF2Up carries

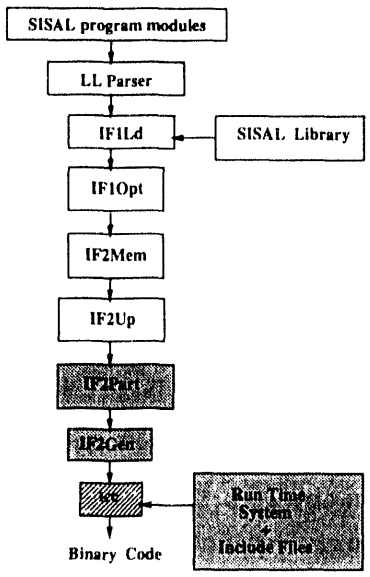


Figure 1: Osc - Optimizing Sisal Compiler

out update-in-place analysis to carry out some operations in place without additional memory allocation or copying. IF2Part is the phase that is responsible for parallelization of the Sisal programs. Finally, the last phase IF2Gen translates the optimized IF-2 graphs into C code and the C compiler on the target links the run time libraries and produces the binary executable code.

In the scope of this work, IF2Part has been rewritten to perform compile time scheduling of functional parallelism in IF-1 nodes. The IF2Gen has been modified to generate efficient code for the Intel Gamma, Delta, and Paragon family. The run time system has been modified to support message passing communication between different processors to send and receive the data values. These contributions have been indicated by the shaded areas in the figure 1.

3.2 Threshold Scheduling Algorithm

First, the preprocessor phase of the partitioner carries out a dependence analysis, identifies actual dependencies and scope imports of values using the Sisal intermediate form, and performs cost assignments based on iPSC/860 timings (Refer to Figure 2) [9]. This partitioner has been incorporated in the IF2Part phase of the Osc.

Let's first introduce some definitions and the assumptions about Sisal IF-1 graph execution before discussing the Threshold Scheduling Method and the results.

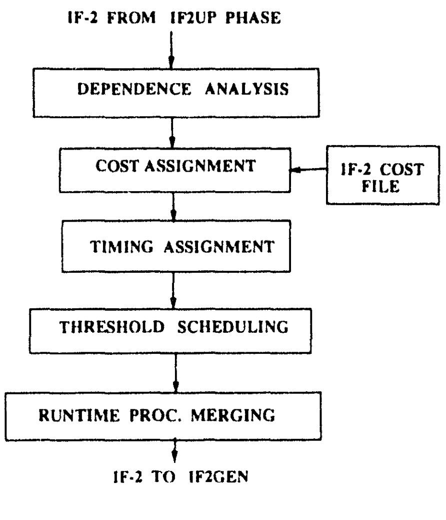


Figure 2: Threshold Partitioner

3.2.1 Assumptions

The following assumptions are made about the execution of a Sisal task graph on the target machine:

1. The task graph is a directed acyclic graph (DAG).
2. The tasks are strict (or, in other words, a task cannot start execution unless all of its inputs become available). This restriction is imposed by the Sisal semantics.
3. The tasks are non-preemptive and have a finite termination time.
4. The values are exchanged between two processors in the form of messages, by using asynchronous send() and receive() calls. This assumption is made specially for Intel Touchstone i860 systems.
5. The cost of a message is determined by the startup cost and the length of the message. For a message m bytes long, the cost is given by $\alpha+\beta m$, where α is the fixed start-up cost for the messages and β is the incremental cost per unit length of message.
6. The task creation overhead is assumed to be negligible compared to task execution time.

3.2.2 Definitions

A task graph G(V,E) of Sisal IF-1 nodes is a directed acyclic graph (DAG) such that each node $v_i \in V$ of G is a Simple IF-1 node representing a task and the directed edge $e(v_i, v_j) \in E$ represents the precedence constraint from v_i to v_j, that corresponds to the actual dependence edge. This dependence edge could be either a data or a control dependence edge. The scaled computation cost of v_i is denoted by $t(v_i)$ and the scaled communication cost of edge $e(v_i, v_j)$ is denoted by $c(v_i, v_j)$. The scaled computation and communication costs for each of the nodes and edges are found as described earlier.

Four kinds of timings associated with each node are defined:

Using the strictness condition, and the precedence constraints, the earliest start time of the node v_i is defined as,

$$est(v_i) = \min_{j,e(v_j,v_i)\in E} \max_{k,k\neq j,e(v_k,v_i)\in E} (ect(v_j), ect(v_k) + c(v_k, v_i)),$$

The earliest completion time of the node v_i is given by,

$$ect(v_i) = est(v_i) + t(v_i)$$

The latest start time of the node v_i is given by,

$$lst(v_i) = \max_{j,e(v_j,v_i)\in E} (lst(v_j) + c(v_j, v_i)),$$

The latest completion time of the node v_i is given by,

$$lct(v_i) = lst(v_i) + t(v_i)$$

The schedule margin represents the delay in the schedule time of the given task, had all the tasks in the graph been allocated to different processors (maximum parallelism case), over the best case in which a task starts execution at the earliest start time given by the critical path length. Thus, the schedule-margin is a measure of the tradeoff between the schedule length and the degree of parallelism. Let's define a schedule-margin value associated with node v_i as,

$$sm(v_i) = lst(v_i) - est(v_i)$$

When a task v_i is scheduled on a processor $p(v_j)$ on which a predecessor task v_j is scheduled, no communication is needed between v_i and v_j. The schedule time is given by,

$$stime(v_i, p(v_j)) = \max_{j \neq k}((stime(v_j, p(v_j)) + t(v_j)), \\ (stime(v_k, p(v_k)) + t(v_k) + c(v_k, v_i)))$$

The schedule delay of a node v_i on processor $p(v_j)$ defines the delay in scheduling v_i from its earliest start time and is given by,

$$sd(v_i, p(v_j)) = stime(v_i, p(v_j)) - est(v_i)$$

The schedule delay is upper-bounded by the schedule-margin. The completion time of a node v_i on processor $p(v_i)$ is given by,

$$ctime(v_i, p(v_i)) = stime(v_i, p(v_i)) + t(v_i)$$

The threshold assumes the values in the range,

$$\min_{v_i \in V} sm(v_i) \leq Threshold \leq \max_{v_i \in V} sm(v_i)$$

A merit function is used to break ties between different tasks competing for the same processors. A merit function decides the quality of matching between a task and a processor. Task A is better than task B on a given processor if task A delays the processor completion less than B. Similarly, task A gets a preference over task B on a given processor, if task A is delayed more from its earliest start time, than task B. The merit function is a composition of these two requirements. The merit function of a node v_i on processor $p(v_i)$ is given by,

$$merit(v_i, p(v_i)) = (stime(v_i, p(v_i)) - est(v_i)) \\ -(ctime(\tau + v_i, p(v_i)) - ctime(\tau, p(v_i))))$$

where, τ is the set of tasks already scheduled on $p(v_i)$ and $\tau + v_i$ is the new task set resulting by adding v_i , $ctime(\tau, p(v_i))$ gives the completion time of task set τ on $p(v_i)$. The merit function of a task set τ is given by,

$$merit(\tau, p(k)) = \max_{k \in \tau} merit(k, p(k))$$

The essence of the scheduling algorithm is that it tries to limit the schedule delay of each task below the threshold. Thus, a greater threshold value implies larger allowable delay in schedule time and it will be more likely that the task is scheduled on one of the predecessor task's processor and vice-versa. The threshold, thus, controls the tradeoff between the degree of parallelism and the schedule length.

Let's first illustrate the method by the following example.

Refer to the example in figure 3. It shows a task graph created from the following code fraction:

```
...
funct1(arg1,arg2,arg3 : in; arg4, arg5, arg6 : out);
funct2(arg4 : in; arg7 : out);
funct3(arg5 : in; arg8 : out);
funct4(arg6 : in; arg9 : out);
funct5(arg7,arg8,arg9 : in; arg10,arg11,arg12 : out);
funct6(arg10 : in; arg13 : out);
funct7(arg13,arg11 : in; arg14 : out);
funct8(arg12 :in; arg15 : out);
funct9(arg14,arg15 :in);
...
```

The computation and communication costs of each of the nodes and edges are shown next to them. The earliest possible schedule time of task 2 is 100 (right after the completion of task 1), and its latest possible schedule time is 150, if communication due to edge (1,2) is taken into account. Only the task precedence relations, and the *strictness* condition mentioned earlier are taken into account, to find out the earliest and latest schedule times of each task. For example, the earliest and latest possible schedule times for task 5 will be determined on the basis of timings of each of the tasks 2, 3, and 4, and the communication cost along the edges (2,5), (3,5) and (4,5). If a task is to be executed on one of its predecessor task's processor, the communication overhead along the corresponding edge is saved.

In this manner the earliest and the latest possible schedule times of each of the tasks are found. The threshold is varied between 0 and 157. Suppose a threshold value 50 is being used. First task 1 is scheduled on p(1). When task 1 completes, each of the tasks 2, 3, and 4 are ready to run at time t=100. The tasks 2, 3, and 4 compete for p(1), to avoid communication. This tie is broken using a merit function, that basically gives a measure of the task delay, and the processor completion delay. The merit functions of task t on processor p is found as follows:
merit(t,p) = Task delay of t on p - Processor completion delay of p due to t.
Since, the merit function of 4 is the highest, task 4 is allocated to p(1). Next, the tasks 2 and 3 are allowed

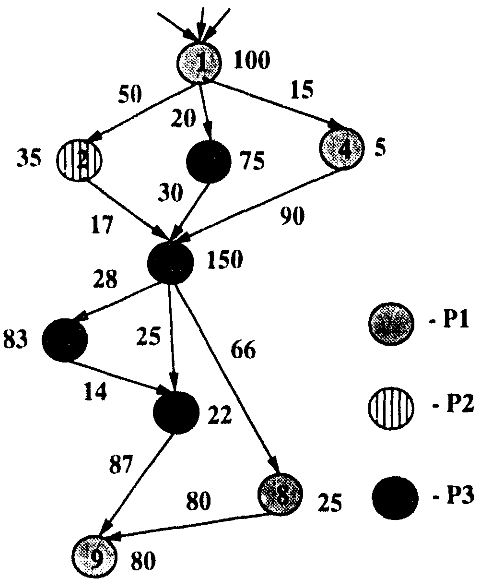


Figure 3: Example Task Graph

to be allocated to new processors, and the resulting task delays are examined. In this case, both the task delays of 50 and 20 are below the threshold, permitting such an allocation. The processor assignment is carried out in this manner for a set of the thresholds values chosen by the algorithm, and the best value is chosen to satisfy the given scheduling goal of either minimizing schedule length or reduce the number of required processors below the maximum available number of processors.

3.2.3 Algorithm

The Threshold Scheduler varies the value of the threshold, between its minimum and maximum bounds and determines a value that gives the minimum schedule length or reduce the number of required processors below that of the maximum number of available processors. The set of threshold values is found by using a difference between the actual schedule time and the earliest schedule time of each task on its predecessor tasks' processors. The scheduling is attempted for each threshold belonging to this set, and is thus called Discrete Threshold Scheduling. Appendix A gives the algorithm that traverses the IF-1 graph in depth-first manner to perform scheduling. It takes $O(k\ v)$ worst case time, where v is the number of nodes, and k is the number of threshold values used by the algorithm to find the best threshold.

Two options are offered for compiling programs:

- Scheme A: Reduce schedule length as much as possible which could be useful on large systems.
- Scheme B: Find a schedule with processor demands below a given maximum number of available processors in the system. Of the many schedules that could satisfy this condition, the one with the minimum processor demands is chosen. The maximum number of available processors is assumed to be 16.

3.3 Compile Time Results

Using the threshold scheduling algorithm described in Appendix A, the schedule lengths, required number of processors, and the speedups obtained for different numerical packages are given in figures 4, 5 and 6 using discrete threshold scheduling. Referring to figures 4, 5 and 6 the results of Scheme A are summarized as follows:

- This scheme finds the schedule length within a factor of 2.0 of the critical path lower bound. It is well known that the critical path lower bound itself might not be achievable for optimum solution.
- The speedups range from 1.89 upto 26 and are higher than scheme B.
- The utilization are low due to the high number of processors found by this scheme. IF-1 graphs are sparse and in many cases tree-like parallelism

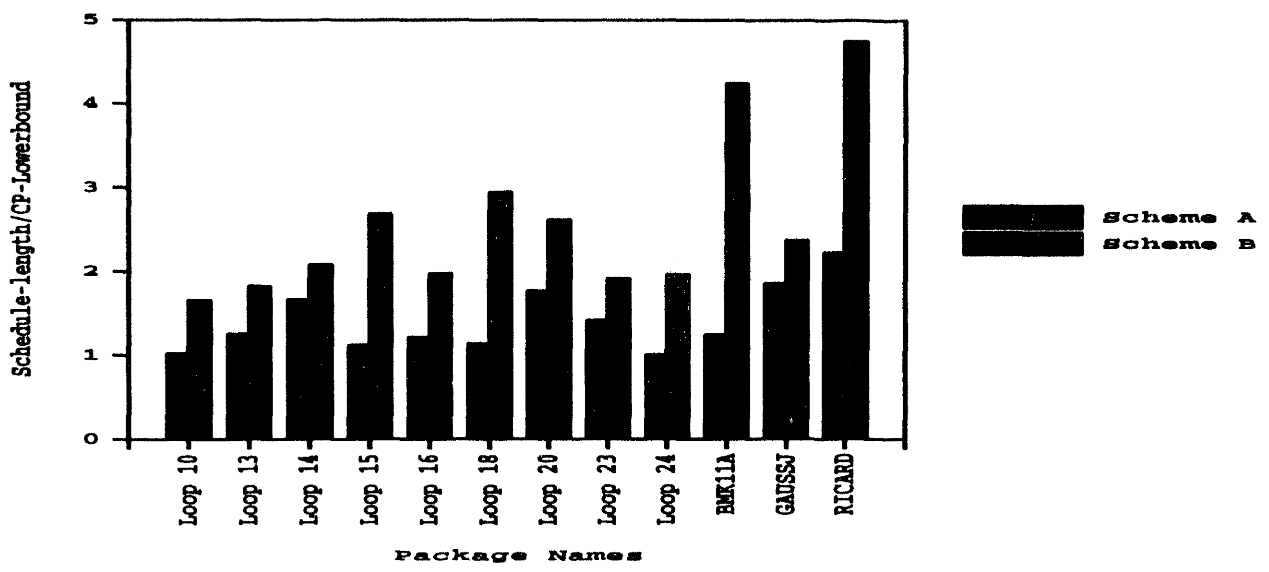


Figure 4: Discrete Threshold Scheduling: Schedule length

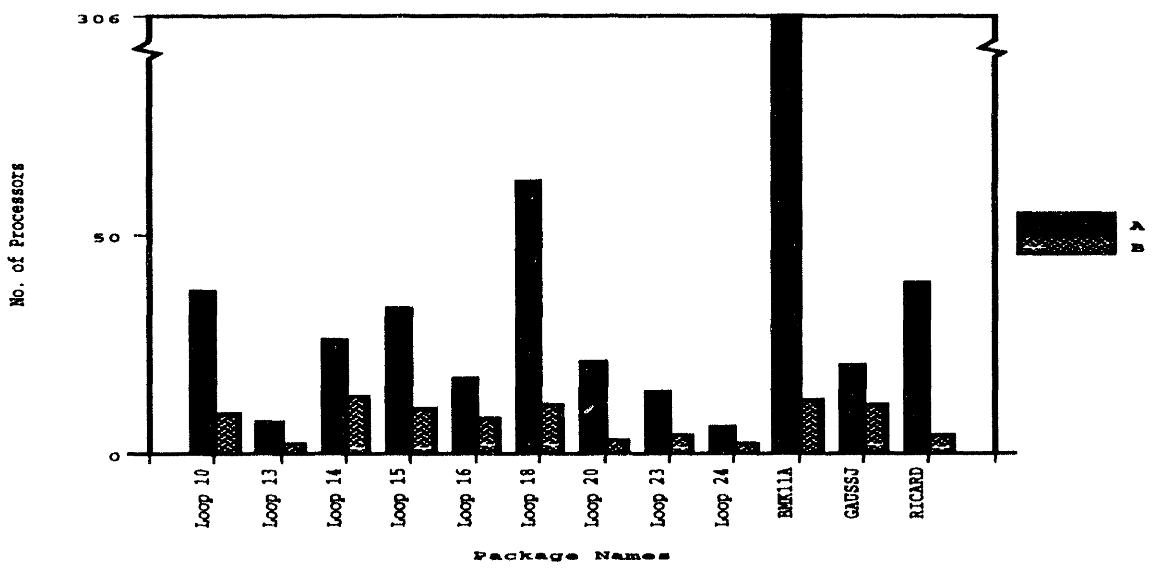


Figure 5: Discrete Threshold Scheduling: Processor requirements

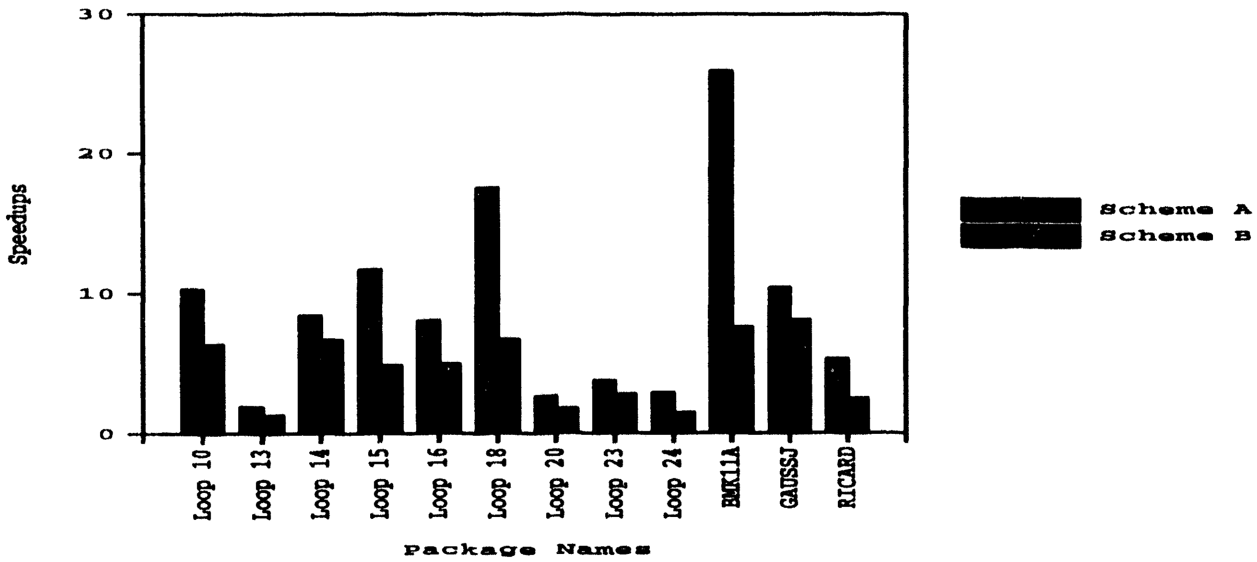


Figure 6: Discrete Threshold Scheduling: Speedup

results after the dependence analysis, requiring a large number of processors. The number of processors are quite high. But these number of processors could be allocated on a large system such as an Intel Touchstone i860 with 512 nodes.

Referring to figures, 4, 5 and 6 the results of Scheme B are summarized as follows:

- The schedule length increases to as much as upto 5 times the critical path lower bound. But in many cases, it is maintained within a factor of 3.
- The speedups range from 1.22 upto 8 and are much smaller than scheme A.
- The number of processors is low and the utilization is high in this scheme as compared to Scheme A. This is due to the fact that at high values of thresholds, every attempt is made to schedule a given task on the predecessor task's processor. This results in a fewer number of processors. The number of processors found by this scheme are below 16 (a typical size for 'small' Intel Touchstone i860)

4 Code Generation and Run Time System

Let's first present an overview of the code generator and run time system of Sisal compiler, Osc, that produces code for a shared memory system like Sequent Balance.

The Sisal compiler, Osc, translates the IF-2 to a C program in its final phase, IF2Gen. This C program is then linked to the run time system producing object code on the target machine using its native C compiler. In this manner, the Sisal project has attained its goal of easy portability across wide spectrum of platforms, as well as ease of modifying the run time system for different parallel architectures.

4.1 Code Generator

The Shared Memory Sisal compiler Osc, tries parallelization of FORALL loops, and stream tasks, on systems like Sequent Balance, and Cray YMP.

The code generation phase starts with the IF-2 graph in which FORALL and Stream graph nodes are decorated with parallelization/noparallelization pragmas in earlier partitioning phase, IF2Part (refer to figure 1). Figure 7 shows the structure of shared memory IF2Gen. The first part **If2Opt** tries optimization of AGatherAT node that gathers a scattered array. The aim of **IF2Opt** is to manage storage required by AGatherAT efficiently. The next phase is responsible for improving array indexing and referencing. The arrays are implemented through a dope vector that has a pointer to the physical space. The **If2Yank** phase first carries out a type merging operation based on structural type equivalence and casts other types into the ones representative of each equivalence class. The

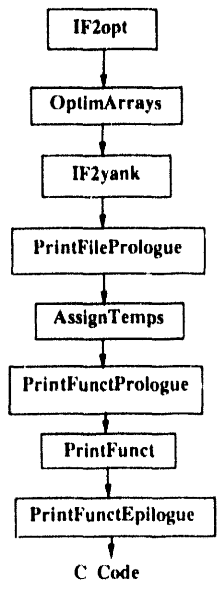


Figure 7: Osc code generation phase

above phases can be collectively called the preprocessor part of the code-generator.

The next part **PrintFilePrologue** puts the correct headers, globals, forward definitions, the type table, the union and record structures as a preamble in the C code file. The custom read/write and deallocation utility routines for records and unions are then printed and their addresses are entered in the type table. The index into the type table is used by the run time system to lock up for a particular allocation/deallocation routine. Next, prototypes for user defined functions and for libraries are printed. The **AssignTemps** phase assigns the temporaries to the different Simple and AT nodes. The **PrintFunctions** pass then traverses the IF-2 in a top down manner. For every function, first a function name and an argument list is printed using C syntax, followed by its local variable declarations. Then, the IF-2 nodes belonging to function's body are visited in the depth first order and their operations are printed in terms of macros that expands appropriately using Sisal run time system, and that take proper arguments in terms of the assigned temporaries. For Compound nodes, special semantic action is taken to print the conditionals, union tags, or the loop control structure as required. At the end of each of the functions, a function epilogue is printed. The shared memory implementation prints epilogues only for parallel tasks, which consist of a frame deallocation routine.

4.2 Run Time System

The shared memory Sisal run time system consists of a group of routines that co-operate to carry out three functions:

- To manage machine defined aggregates such as the arrays and streams.
- To carry out input/outputs for Sisal Fibre standard. Fibre is a Sisal input-output standard (format).
- To manage the processes and block allocations for tasking mechanism.

Array handling:
The array handling routines mainly perform: array allocation, deallocation, copying and referencing. The arrays are maintained through dope vectors, and indexed using base+offset addressing. Special bounds checking routines ascertain the legality of references. The arrays in Sisal are single dimensional while multi-dimensional arrays are implemented as arrays of pointers that point to each row.

Stream Handling:
The streams are implemented as buffers (of MaxStreamSize) keeping read and write pointers. The float and int streams are implemented separately to maintain better efficiency. The important operations on streams: read, write, dealloc, and error are performed by various routines.

Fibre I/O:
The Fibre standard is managed by a toolset written in lex that mostly performs a parsing and formatting job. **Process and block manager:**
It is the most important part of the Sisal run time system. It initially establishes a shared memory system by getting a chunk of shared memory and divides it into blocks for allocating them to different workers. Each of the workers obtains tasks for itself from a shared queue of ready to run tasks.

The code generator and the run time system is modified in this work to support functional parallelism on Intel Gamma, Delta, and Paragon family of distributed memory machines, efficiently. Functional parallelism is quite fine-grained as compared to the communication costs on these machines; hence care must be taken to minimize the overheads.

4.3 Distributed Memory Systems Modifications

The run time model has been modified for the distributed memory system implementation. In this implementation, functional parallelism between the Simple nodes of a program's IF-2 representation, is used. By in-lining the Sisal function calls in its SisalMain(), a corresponding C main() of the program is created. Every processor has the same copy of the generated code and individual code for each processor is separated in the C main() by using a case statement. The code for the execution of a Simple node is present only under the scope of that processor which is supposed to execute it.

The other code including control code of the program like 'if' statements, and loop control structures is repeated under the scope of each processor. Communication primitives are put at the beginning and end of each Simple node to receive and send values from and to other processors.

All the processors start execution through a start-up routine and set up their context that includes processor number, process id etc. A processor then takes the appropriate branch to the corresponding case statement in C _MAIN() corresponding to SisalMain(), and starts execution of the program under this case. Whenever it needs to receive or send values, it executes the embedded send() and receive() calls to communicate with other processors. On completion of the _MAIN(), each processor prints its timings and the output values of the program by using PutFibreOutput() as described earlier

4.3.1 Pre-processing

Before generating the intermediate C code, the 'in-lining' optimization in Osc is activated so that all the function calls are in-lined at their respective call sites. Since Sisal does not allow recursive definition of functions, this optimization can be always carried out. Thus, after in-lining, the functional parallelism is exposed in the interprocedural framework, and a single SisalMain() function is created that contains all the in-lined code.

In the IF-2 representation of Sisal programs, multiple edges might carry the same value(s) from one node to another. This results in sharing the same temporary amongst multiple edges. For a shared memory implementation, these extraneous edges do not matter, but in a distributed memory implementation, it might generate a lot of unwanted communication. Therefore, before the code generation starts, the following algorithm performs some pre-processing to remove such extraneous edges and marks other edges active.

```
1.For all the nodes in the IF-2 graph of
  SisalMain() do,
  1.a Mark all the edges and their corres-
      -ponding temporaries, INACTIVE.
2. For all the export edges of a node,
   such that their destination node
   is not Simple, do
       2.a If the temporary corresponding
           to an edge is INACTIVE, make both
           the edge and the temporary ACTIVE.
3. For all the export edges of a node, such
   that their destination node is Simple, do
       3.a If the temporary corresponding to
           the edge is INACTIVE, make both
           the edge and the temporary ACTIVE.
4. Remove all INACTIVE edges in the graph.
```

To make the objective of this algorithm clear, let's present an example. Figure 8 contains a small IF-1 graph. Nodes 1 and 2 are Simple nodes. The output value t generated by node 1 is fed to both node 2 and the surrounding graph boundary. Let's assume that node 1 is scheduled on processor 1 and node 2 is scheduled on processor 2. The graph boundary 0 is universally owned and hence belongs to both processors 1 and 2. In this case, no messages will be generated for the edges e0 and e1, since their source is a universally owned graph boundary and the values generated there are available locally on each processor. As far as the output values of nodes 1 and 2 are concerned, the code generator would take following actions. Send value of

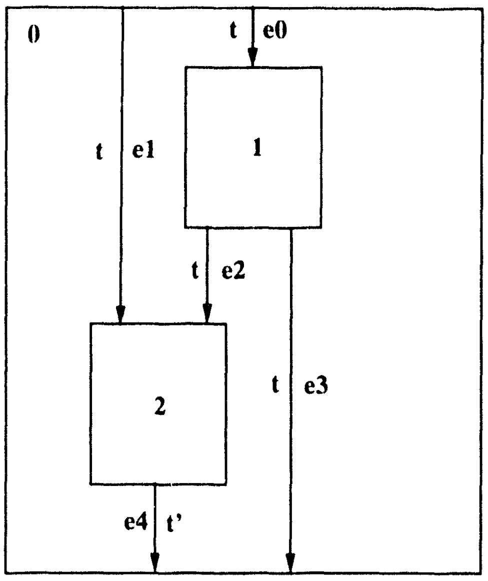


Figure 8: IF-1 Example Graph

t from node 1 to 2 for edge e2. Send value of t from node 1 to all the processors (since graph 0 boundary is universally owned), for edge e3. Send the value of t' from node 2 to all the other processors. As one can see, processor 2 who owns node 2, gets the value of t, twice: once, the universally broadcast value on edge e3, and once the value on edge e2. Thus, there is a redundant send and receive pair in the code. If the edge e2 is eliminated, the redundancy disappears. The above algorithm achieves this objective.

4.3.2 Code Generation

In the code generator, the phases before **PrintFunction** are unmodified. While printing the SisalMain()'s equivalent C main(), the code to be executed by each processor is separated by using a case statement, that uses the processor number. This scheme was preferred over the scheme of isolating each of the Simple nodes (that are parallelized) using an 'if' guard to check processor ownerships. A processor is said to own a given Simple node, if only it and no other processor is responsible for the node's execution[1]. The reason for the choice of using a 'case' statement instead of 'if' statement is that the overhead in using 'if' guards is tremendous. On the other hand, by collecting all the Simple nodes owned by a given processor and putting them under a single 'case' amortizes this overhead. Also, the C compiler would probably process 'switch' better than individual 'ifs' due to the fact that 'switch' may be implemented by a jump table where each 'case' (that corresponds to the processor number) can index in it to go to its branch.

Once a 'case' starts the scope of the statements to be executed by the corresponding processor, the complete IF-2 graph of the SisalMain() function is traversed and the following algorithm is used to generate the code for every processor:

```
For processor from 1 to Required_processors do,
1. If ((node.type = Simple) &&
   (node.owner != processor)) then
    generate communication primitives to
    receive values generated by this node
    that are universally owned. A value
    is said to be universally owned if
    it is available on every processor.
    Such a value is generated by an edge
    whose destination is a node other
    than a Simple node that is universally owned.
    Go to step 4.
2. If ((node.type = Simple) &&
   (node.owner = processor)) then
     For all the input edges to this node, do
       2.a. If the source of the edge is a
            Simple node, generate a communication
            primitive to get the values from
            owner of that node.
       2.b. If the source of the edge is not
```

[1] All the nodes other than Simple nodes, are executed by every processor and are thus, universally owned

```
            a Simple node, then the respective
            value is universally owned and is
            available locally. No action is
            needed by the code generator.
       end for
3. Generate the computation code
   corresponding to the given node.
4. If ((node.type = Simple) &&
   (node.owner = processor)) then
    For all the output edges to this node, do
      4.a. If the destination of the edge
           is not a Simple node,
           generate a 'broadcast' primitive
           to send the universally owned
           values to every processor.
       4.b. If the destination of the edge
            is a Simple node, put a
            communication primitive to send
            the values to the owner
            of the destination node.
    end for
5. Continue steps 1 to 4 above for all the
   nodes in the function graph of SisalMain().
end for
```

The communication primitives are generated for each of the values to be passed from one processor to another. Following algorithm generates communication primitive(s) to pass data objects:

```
1. If (temporary.type = Scalar),
   calculace its size on the target
   in bytes and generate appropriate
   communication primitive.
2. If (temporary.type = Array or Stream),
     2.a. First calculate the size of
          its dope vector and generate
          communicative primitive for
          passing and receiving it.
     2.b. Generate communicative primitive
          for passing and receiving it
          based on the physical size given
          in the dope vector as in 2.a.
3. If (temporary.type = Record or Union),
     For each of the the fields,
        3.a. Follow steps (1) to (3) above.
```

Thus, it can be seen that, due to the conformance of the arrays and streams to the data size in Sisal, one has to first pass its dope vector that gives the size of the physical space and then pass the array or stream itself. Therefore, two messages are needed for passing such a variable size data object.

4.3.3 Run Time System

The main job of the run time system is to properly support the communication primitives. Two basic communication primitives are defined as follows:

```
pick(message_type, recv_data_addr,
     no_bytes, source_proc, source_pid)
put(message_type, send_data_addr,
    no_bytes, dest_proc(s), dest_pid)
```

The put and pick primitives operate asynchronously so that the sending and the receiving processor do not have to *rendezvous*. This exposes more parallelism. These communication primitives are implemented as macros that expand to appropriate system call(s) on Intel machines.

Also, the different semaphore locks and cache blocks from which memory is allocated to a requesting processes, are not needed in the distributed memory implementation. They are, therefore, removed from run time system. Also, the start-up code is simplified since all the processors execute the C main and then branch off through a case statement. Therefore virtually no process management is needed unlike the shared memory implementation.

Most of the array and stream compute macros remain as they are in the shared memory implementation.

4.4 Benchmark Results

Some benchmarks have been carried out on Intel Gamma, Delta, and Paragon. Intel Gamma is a hypercube, whereas Delta is a mesh. Unlike both Gamma and Delta, Paragon has OSF/1 as its operating system with much simplified micro kernels.

The parallelized programs have been executed on these machines for both Scheme A and B and speedups have been measured. Refer to tables 1 and 2 for the results of the benchmarks. The timings have been gathered using a millisecond clock on these machines.

The predicted speedups are those found by compile time scheduler (as described before) by assuming the architectural cost model.

4.5 Discussion

Following observations can be made from the above results:

1. The speedups obtained for all the three machines are lesser than those predicted by the compile time method. This can be mainly attributed

Table 1: Scheme A Speedups

Package Name	No. Procs.	Predicted Speedup	Gamma Speedup	Delta Speedup	Paragon Speedup
Loop 10	37	10.3	4.67	6.1	7.22
Loop 13	7	1.89	1.1	1.38	1.66
Loop 14	26	8.12	3.25	4.55	6.12
Loop 15	33	11.68	8.21	10.22	10.45
Loop 16	17	8.04	4.46	5.33	6.8
Loop 20	21	2.64	1.59	1.89	1.94
Loop 23	14	3.75	1.88	2.11	2.33
Loop 24	6	2.88	1.22	1.67	1.75

Table 2: Scheme B Speedups

Package Name	No. Procs.	Predicted Speedup	Gamma Speedup	Delta Speedup	Paragon Speedup
Loop 10	9	6.37	3.29	3.89	4.5
Loop 13	2	1.29	.89	1.05	1.14
Loop 14	13	6.71	3.23	3.78	4.77
Loop 15	10	4.88	3.12	3.77	4.15
Loop 16	8	4.94	2.58	2.78	2.9
Loop 18	11	6.71	4.77	5.1	5.33
Loop 20	3	1.77	1.12	1.15	1.2
Loop 23	4	2.78	2.16	2.22	2.61
Loop 24	2	1.47	.96	1.09	1.16
RICARD	4	2.48	1.34	1.67	1.88

to the run time overheads and the higher communication latencies resulting from unpredictable network delays. The communication model of $\alpha + \beta * m$, where α denotes the start-up cost of a message, β denotes the incremental cost per byte, and m denotes the number of bytes, is based on the fact that there are no other messages present in the system at the time of communication. The presence of more than one messages in the network makes the communication latencies unpredictable, due to the unpredictable network delays. Thus, the latencies can not be analytically modeled, and are higher than predicted by the above model. This can adversely affect the fine grain functional parallelism. This is the main reason of the loss of parallelism (in some cases as high as 50%).

2. The speedup for Delta is better than Gamma, and that of Paragon is better than Delta. The main reason for this behavior is that Delta has better routing hardware and topology than Gamma, and Paragon has a better operating system software (OSF/1) as compared to Delta.

3. The speedups in some cases are low compared to required number of processors, because the fraction of the code that could be run in parallel is small (Amdahl's law). In fact, in most Sisal programs, an inverted tree type parallelism is present. In other words, there many operations that can be executed in parallel at the beginning of a graph or sub-graph boundary, and as the processing occurs in the IF-1 nodes, fewer and fewer values are produced after the reductions that are fed to the subsequent IF-1 nodes. In many cases, towards the end of a graph or sub-graph boundary, just one or two data values are operated upon leading to an almost serial code. Thus, there is a high processor demand at the beginning of a boundary that quickly diminishes towards its end. This phenomenon leads to inverted tree type parallelism that reflects in an overall high processor demand with relatively low speedups.

5 Conclusion

This work introduces a new compile time method to schedule functional parallelism in a program on distributed memory systems, to minimize either program completion time, or to generate a schedule with the processor requirements below the maximum number of available processors in a system.

The results are very promising and this strategy could be of immense use in compiling for varying sizes (in terms of number of processors) of distributed memory systems, and thus could be used for a variety of platforms.

The work, thus, provides for a general framework involving a compilation methodology to effectively map functional parallelism in any language on distributed memory systems, with both, small and large numbers of processors.

References

[1] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A Report on Sisal Language Project", *Journal of Parallel and Distributed Computing*, Vol. 10, No. 4, October 1990, pp. 349-366.

[2] N. Mansour, and G. C. Fox, "An Evolutionary Approach to Load Balancing Parallel Computation", *Proc. 6th Distributed Memory Computing Conf.*, April 1991, pp.200-203.

[3] D. Gajski, and J. Peir, "Camp: A Programming Aide for Multiprocessors", *Proc. Int'l Conf. On Parallel Processing*, August 1986, pp. 475-482.

[4] A. Gerasoulis, and T. Yang, "A Comparison Of Clustering Heuristics For Scheduling Directed Acyclic Graphs On Multiprocessors", *Journal Of Parallel And Distributed Computing*, Vol. 16, No. 4, December 1992, pp. 276-291.

[5] M. Girkar, and C. Polychronopoulos, "Formalizing Functional Parallelism", CSRD Report 1141, June 1991. University of Illinois at Urbana-Champaign, 1991.

[6] S. Hiranandani, K. Kennedy, and C. W. Tseng, "Compiling Fortran for MIMD Distributed-Memory Machines", CACM, August 1992, Vol. 35, No. 8, pp. 66-80.

[7] C. Koelbel, P. Mehrotra, and J. Van Rosendale, "Semi-automatic Domain Decomposition in Blaze", *Proc. Int'l Conf. On Parallel Processing*, August 1987, pp. 521-524.

[8] W. Li, and K. Pingali, "Access Normalization: Loop Restructuring for NUMA Compilers", Technical Report, TR 92-1278, Cornell University.

[9] S. S. Pande, D. P. Agrawal, and J. Mauney, "A New Threshold Scheduling Strategy for Sisal programs on Distributed Memory Machines", *Journal Of Parallel And Distributed Computing* (To appear).

[10] J. Ramanujan, and P. Sadayyapan, "Access Based Data Decomposition for Distributed Memory Machines", *Proc. 6th Distributed Memory Computing Conf.*, April 1991, pp. 196-199.

[11] J. Rose, and G. Steele, "C*: An Extended C Language for Data Parallel Programming", Technical Report PL87-5, Thinking Machines Inc.

[12] V. Sarkar, "Automatic partitioning of a program dependence graph into parallel tasks", IBM Journal Of Research And Development, Vol. 35, No. 5/6, Sept.-Nov. 1991.

Appendix A : Threshold Scheduling Algorithm

```
/* Following pseudo-code calculates the Best Threshold and finds schedule*/

      1. Input : Outermost IF-2 graph, G, and minimum, and maximum
         bounds on Threshold, and choice of either of compilation
         schemes A or B (Scheme A allows compilation for minimizing schedule
         length and scheme B allows compilation for minimizing number of
         processors).
      2. Output: Processor assignment for Simple and At nodes of G.

          procedure GetProcessor(n, Threshold, P)
          begin
                        1. Select a predecessor task m of n such that the
                           schedule time of n is minimum.
                        2. Let p(m) be the respective processor of the
                           predecessor task m. Find schedule time of n on
                           p(m).
                        3. Find the schedule delay sd(n, p(m))
                           If sd(n, p(m)) <= Threshold, allocate
                           n to p(m).
                        4. If sd(n, p(m)) > Threshold, find set of
                           clashing tasks, T on p.
                        5. Find merit(n, p(m)) and merit(T, p(m))
                        6. If merit(n, p(m)) < merit(T, p(m)), go back
                           to step 1 by choosing next best predecessor
                           task of n. If all the predecessor tasks are
                           visited go to step 8.
                        7. If merit(n, p(m)) >= merit(T, p(m)), allocate
                           n to p(m). Migrate all tasks g in set T.
                           a. for all the tasks g in set T do,
                              call Getprocessor(g, Threshold, P).
                        8. If in steps 1 to 7, n does not find its
                           processor, find any free processor p(k) such
                           that sd(n, p(k)) <= Threshold, and is minimum.
                        9. If no processor could be found in 8, add
                           another processor and allocate n to it.
                           P = P + 1
          end procedure
```

A Virtual Shared Addressing System for Distributed Memory Sisal

Matthew Haines*

ICASE
NASA Langley Research Center
Hampton, VA 23681

Wim Böhm†

Computer Science Department
Colorado State University
Fort Collins, CO 80523

Abstract

Efficient implementations of Sisal exist for shared memory and hierarchical memory multiprocessors, but distributed memory implementations have been hampered by the Sisal compiler assuming a single address space. We have designed and implemented a solution to this problem by developing the VISA (Virtual Shared Addressing) runtime system, which provides distributed task and data management. In this paper we discuss the VISA data management system, including the design and implementation of a single addressing space, data distribution functions, and address translation. We provide an example of the performance of Sisal with VISA on the nCUBE/2 distributed memory multiprocessor using a two-dimensional smoothing algorithm called Laplace. We show that the current Sisal compiler's implementation of multi-dimensional arrays is highly inefficient in a distributed system, but that an efficient implementation of rectangular multi-dimensional arrays is possible. We also show that multithreading can effectively increase the performance of this program even further.

1 Introduction

Large-scale distributed memory multiprocessors represent the current state of the art in high-performance computer architecture [14, 10, 18]. Programming these machines requires the management of both parallel tasks and distributed data, which is often done explicitly using language constructs for spawning and synchronizing tasks, and for message passing. The resulting programs are difficult and time-consuming to write, and contain a large amount of machine dependent housekeeping code not germane to the specification of the problem. An alternative approach is to use a high-level language that provides implicit parallelism and the resulting management of parallel tasks and distributed data.

Sisal [13] is a functional language that supports data types and operations for scientific computation, and provides implicit parallelism on a number of shared memory parallel architectures. The Sisal compiler (*OSC*) consists of three parts: a frontend, a backend, and a runtime system. The frontend translates the source program into intermediate dependence graph form. The backend optimizes the intermediate representation and generates native C code. The *runtime system* provides the Sisal compiler with two main abstractions: task management and shared memory management. We have modified the runtime system to provide support for both abstractions in a distributed memory environment, and in [7] we provide an overview of the initial system and its performance.

In this paper we discuss the design of the distributed memory data management portion of the runtime system, called *VISA*. The compiler (or programmer) is provided with a shared memory abstraction, which consists of a set of primitives for allocating and accessing shared data structures within a virtual address space (see Appendix A for a complete list of VISA primitives). Data distribution is accomplished by specifying a data decomposition, or mapping function, upon allocation of the structure, so that each data structure can be individually distributed (or replicated) independent from the other data structures. We provide an example of the performance of Sisal with VISA using the nCUBE/2 distributed memory multiprocessor and a two-dimensional smoothing algorithm called Laplace. We show that the current Sisal compiler's implementation of multi-dimensional arrays is highly inefficient in a distributed system, but that an efficient implementation of rectangular multi-

*This work supported in part by a grant from Sandia National Laboratories while at Colorado State University

†Supported in part by NSF grant MIP-9113268

dimensional arrays is possible. We also show that multithreading can effectively increase the performance of this program even further. This paper does not address the implementation issues VISA, such as how a Sisal-generated C program is augmented with the VISA primitives.

In Section 2 we provide an overview of VISA, and the design and implementation of the supporting system. Section 3 provides a description of the Laplace algorithm and its performance, along with an analysis of the results. Section 4 provides a brief description of related research projects, and we conclude in Section 5.

2 The Design and Implementation of VISA

Central to the current Sisal compiler is the assumption of shared memory, which is required for both system and user data structures. In [5] we outlined the design of a task management system that eliminates the need for global system data structures by employing a distributed, rather than centralized, task distribution approach. In this paper we describe the design of runtime support for a single addressing space and general data decompositions used to manage global user data structures.

2.1 Design Goals

Our goal in designing the VISA runtime system is to eliminate the burden of explicit data management from the programmer, while at the same time providing explicit control over the general distribution of global data structures. Towards this end, VISA provides the following services:

- *Single Addressing Space.* One of the primary difficulties in programming a distributed memory multiprocessor is the lack of a single addressing space for user data structures, such as arrays. This results in encumbering the compiler, or worse yet, the programmer, with the task of distributing data structures and inserting the proper code to fetch and store non-local references. Therefore VISA provides a single addressing space, and a set of associated functions that operate on that space, so that the programmer, or in our case the Sisal compiler, is given a familiar shared memory model of computation.

- *Mapping Functions.* In association with the single addressing space, VISA provides a method for the compiler or programmer to specify "how" the data is to be distributed across the memories for improved efficiency. The idea is to distribute the data structures in tight accordance with the distribution of parallel tasks, so that *local references are maximized.* New analysis techniques [15, 9, 4] can yield the optimal distribution in restricted cases, and in these cases compilers could either insert the communication code directly or pass the distribution information to a runtime system like VISA in the form of a mapping function directive.

- *Split-Phased Transactions.* In [8] we introduced the design of multithreaded task management, which relies on the ability to perform remote references as *split-phased transactions*, where the request and reply phase are decoupled to allow for thread switching between the two phases. VISA provides split-phased transactions in support of the multithreaded task execution model.

The Sisal compiler augments a parallel program with VISA primitives for allocating and accessing the data structures to be kept in the single addressing space[1]. Any variables not placed in the VISA space are unaffected by the system. The augmented program is then compiled using the native C compiler of choice, and linked with the VISA library to create the object program, which can then be executed on a distributed memory multiprocessor.

2.2 Message Passing Abstraction

All message passing required for accessing remote values is handled by the VISA system through the use of a *message passing abstraction*, supporting both synchronous (blocking) and asynchronous (non-blocking) operations. Since these operations are provided by most host operating systems for distributed memory multiprocessors, VISA can be easily ported to other distributed memory multiprocessors by modifying the message passing abstraction to make the proper native calls.

Specifically, the abstraction supports a *non-blocking send* for point-to-point communication, a *broadcast* for disseminating information to all processors, a *blocking receive* for synchronous communication, and an interrupt-based *asynchronous receive* to handle incoming requests. Asynchronous message reception requires polling at some level to determine

[1]Actually, the VISA primitives are currently inserted by hand into the Sisal-generated C code. We are awaiting changes to the compiler which would automatically insert these calls.

when a message arrives and take appropriate action. Most systems, including the nCUBE/2, provide hardware polling for incoming messages, resulting in a hardware trap that is caught by the operating system, and then passed into the user-level in the form of an interrupt. The interrupt causes a VISA message interrupt handler to deal with the message. If the interrupt handler is allowed to be invoked at any arbitrary time during the computation, it cannot modify the global state of the computation. Therefore, either the interrupt handler must be selectively disabled during the times when global data structures are accessed, or it must be prevented from modifying global data structures. The former option requires the placement of expensive system calls for enabling and disabling interrupts around all global data structure accesses, which can be costly and error-prone. Therefore, the VISA system employs the latter option: Any message requiring a global modification is enqueued onto a message list for handling outside of the scope of the interrupt handler.

2.3 Data Distribution

As depicted in Figure 2, the VISA address space is allocated in part of the local memory of each participating node. This creates two types of addressing space for each participating node in the system: a shared *virtual* addressing space that spans all of the nodes, and a *local* address space for data visible only to the local node. Each data structure allocated to the VISA space receives a contiguous set of *virtual* addresses that are shared among the nodes and mapped onto *physical* addresses from each node.

Data distribution (or *data decomposition*) determines how the physical storage for a global data structure is to be divided among the participating nodes. The goal is to divide the data structure among the nodes so as to *minimize the number of remote references* caused by the distribution. This implies that the distribution of data must be tied to the access pattern of the parallel computation, and therefore data distribution needs to be flexible to support a wide variety of access patterns. For VISA, data distribution is accomplished by dividing a data structure into a set of blocks, where each block contains *blocksize* elements. The blocks are then allocated to the physical memories of the nodes in round-robin fashion until all of the blocks have been distributed.

To facilitate a variety of distribution schemes, we assign a set of control parameters to each data structure that define the *blocksize* of each block, the *start node* to which the first block is assigned, and the processor *stride* at which the blocks are distributed. These *data* control parameters correspond to the *task* control parameters that are used to distribute parallel tasks, thus providing a unified method for tying task distribution to data distribution, which is necessary for avoiding unnecessary remote references that occur when tasks and data are not properly aligned. A fourth control parameter for data structures specifies whether or not a data structure is to be replicated. Any data structure, from a single variable to an entire array, can be replicated among the nodes in the VISA system. Replication is accomplished by allocating enough local storage from each node to accommodate the entire structure, and broadcasting all writes to the data structure. Rather than implementing an expensive coherence protocol, VISA assumes that the replicated data structures are controlled by the compiler, where explicit synchronization can be provided which minimizes the synchronization required while still maintaining a coherent system.

Table 1 details the parameter settings for several one-dimensional mapping functions, where the *map_arg* is passed in from the allocation routine, typically specifying the starting node. Most variables and structures are allocated using either the `scalar_map` or the `replicate_map`, depending on the nature of the variable. For example, a structure containing arguments for a parallel slice routine would be replicated to eliminate the remote references required by each of the nodes executing the parallel slice, whereas a global counter would be allocated using the `scalar_map` to ensure consistency. Data arrays are typically allocated using the `block_map`, which provides an even distribution of the data among the nodes in chunks that are often exploited by the contiguous loop structure of the Sisal tasks. Arrays can also be replicated, and as we will see with two dimensional data structures, the pointer array is replicated to eliminate the need for two remote references when accessing an array element.

The current Sisal compiler represents multi-dimensional data structures as structures containing sub-structures. For example, a two-dimensional array is represented as an array of *pointers*, where each element points to the location of a one-dimensional data structure (see Figure 3). This is done to conform to the way in which multi-dimensional arrays are represented in both Sisal and C[2]. Mapping functions for multi-dimensional arrays must therefore consider both the pointer arrays as well as the data arrays. Pointer

[2]True multi-dimensional arrays in C are possible only if the array bounds are given at compile time.

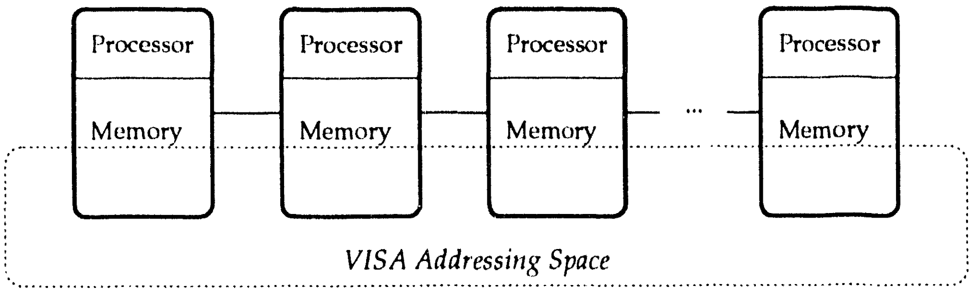


Figure 1: The VISA addressing space

Mapping Function	*blocksize*	*start node*	*stride*	*replicate*
`scalar_map`	n	map_arg	1	No
`replicate_map`	n	P_{id}	1	Yes
`block_map`	n/p	map_arg	1	No
`variable_block_map`	map_arg	P_0	1	No
`interleave_map`	1	map_arg	1	No

Table 1: Control parameter settings for various 1D mapping functions

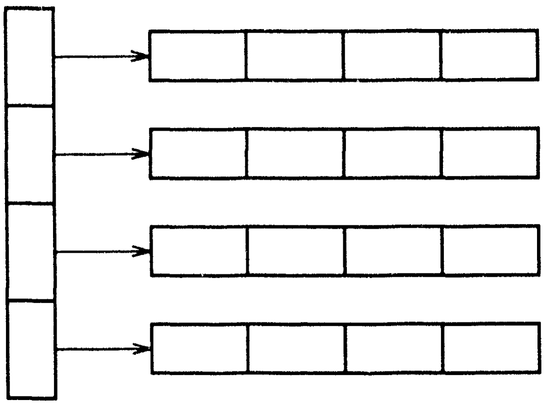


Figure 2: Two-dimensional arrays in Sisal

arrays are replicated to guarantee that accessing any element of a matrix will require at most one remote reference.

Assuming that all pointer arrays are allocated using the **replicate_map**, Table 2 details how the control parameters are established for each of the data arrays in a two-dimensional matrix. The *map_arg* for these mapping functions represents the starting node, and is typically some function of i, corresponding to the i^{th} row of the matrix. For **matrix_row_map**, the map_arg is typically set to $i/(n/p)$, where n is the number of rows and p is the number of processors. When the number of rows is equal to the number of processors ($n/p = 1$), the i^{th} row is placed on the i^{th} processor, and when the number of rows exceeds the number of processors, each processor gets a group of n/p contiguous rows. However, if an interleaved row allocation is desired, the map_arg can be set to $i \bmod p$ instead. For **matrix_block_map**, the map_arg is typically set to $i/(n/rbs)$, where rbs is a control parameter for the **matrix_block_map** function, and is fixed for a given number of processors to allow for the proper layout of the blocks. Figure 4 depicts the distributions for an 8x8 matrix on 4 processors using the ***matrix_row_map*** with contiguous rows and the ***matrix_block_map*** with $rbs = 2$.

It is possible to create many different mapping functions, given the ability to modify the data control parameters. This *general* approach to data distribution is necessary to accommodate the various access patterns that applications exhibit, and VISA allows the user to add to the set of available mapping functions so that customized decompositions are possible. Mapping functions are specified upon requesting memory from the single addressing space using the ***visa_malloc*** function. This allows a compiler that is generating the VISA primitives to invoke *visa_malloc* with the desired mapping function, either obtained from analysis or through user directives. Likewise, a programmer using the VISA primitives directly can select the desired mapping function for each data structure without having to specify the actual message passing details necessary for implementing such a distribution

Mapping Function	*blocksize*	*start node*	*stride*	*replicate*
`matrix_row_map`	n	map_arg	1	No
`matrix_col_map`	1	map_arg	1	Yes
`matrix_block_map`	$rbs * n/p$	map_arg	rbs	No

Table 2: Control parameter settings for various 2D mapping functions

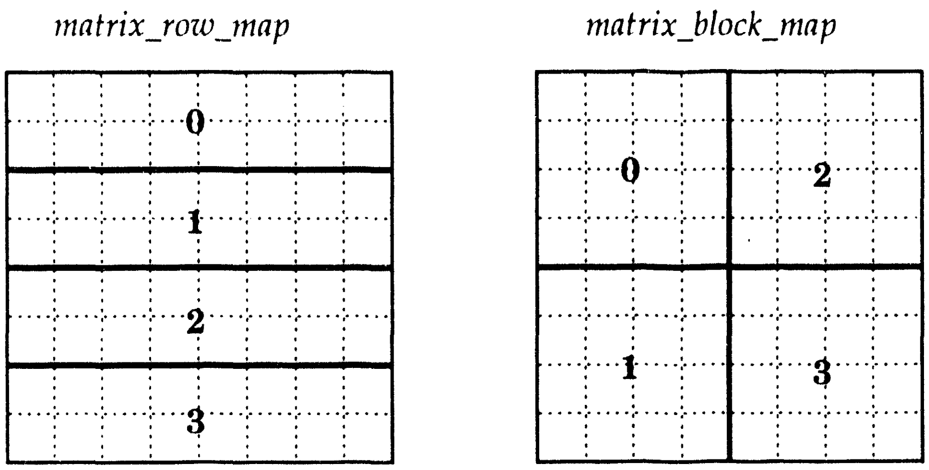


Figure 3: `matrix_row_map` and `matrix_block_map` functions

scheme.

2.4 General Address Translation

Address translation is the process of obtaining the physical address of a datum given its virtual address. For a distributed memory multiprocessor, a physical address consists of the tuple *(node, offset)*, where *node* is a node designator and *offset* is the physical address within that node. Since VISA employs a block-based addressing scheme, where the blocksize, starting node, and stride may all vary, it is necessary to store these control parameters, along with other information about each data structure, in a descriptor called a *range_map entry*. The entire VISA space is therefore described by the collection of these entries, called the *range_map table*. The term "range" refers to the fact that, since all data structures are assigned contiguous addresses in both virtual and physical spaces, the range (low, high) is sufficient to represent all of the addresses within a data structure. To ensure local access of the range_map entries, the range_map table is replicated. There is no coherence problem, since each range_map entry is written only once (upon creation by *visa_malloc*).

In addition to the data distribution control parameters, each range_map entry (depicted in Figure 5) contains three address ranges for each data structure:

- The *visa_base* represents the range of global virtual (VISA) addresses for this data structure.
- The *local_base* represents the range of local physical addresses of the blocks that are allocated for this data structure.
- The *optimized_base* represents the optimized range of global addresses, as explained in Section 2.5.

After a data structure has been distributed with the *visa_malloc* routine, access requires a translation from the virtual VISA address to the physical address tuple *(node, offset)*, which proceeds as follows:

- The range_map entry for the desired data structure is fetched by the *find_rm()* routine, which is exposed to the compiler so that the range_map entry for a data structure that is to be accessed many times need only be fetched once.
- From a *virtual address*, the relative element position within the data structure, the block which

Field	Function
visa_base	The range of global virtual addresses
local_base	The range of local physical addresses for locally-owned blocks
optimized_base	The range of optimized virtual addresses
nelems	The number of elements
size	The size of each element
blocksize	The blocksize (elements per block) used for distribution
start node	The node ID on which to begin distributing the blocks
stride	The stride at which to distribute the blocks
replicate	A boolean to determine if this data structure is replicated
table_index	The index into the range_map table for this entry
next	A utility pointer

Figure 4: Description of a **range_map** entry

contains the desired element, and the relative offset of the desired element within this block are computed:

```
element = address - low_range
block = element / blocksize
block_offset = element mod blocksize
```

- Next, the node which owns the desired block is computed. If the **replicate** flag is set, then the computed node always equals the local node designator, indicating that each node has a copy of the desired block. Otherwise, we compute:

  ```
  node = (start node + (block * stride))
  mod P
  ```

  where **P** is the number of participating nodes.

- If the number of blocks for this data structure exceeds the number of participating nodes, then some (or all) of the nodes will own multiple blocks. Next, the relative block number within the desired node and the relative offset within this node are computed:

  ```
  node_block = block / P
  rel_offset = node_block * blocksize +
  block_offset
  ```

- If the access is local (i.e. **node** is equal to the local node designator) the **rel_offset** is incremented by the local_base from the range_map entry to produce the actual offset in local physical memory, since local_base contains the address of the first byte for this structure:

  ```
  offset = rel_offset + local_base.
  ```

- If the access is remote, a message is sent to the computed node, requesting that the desired datum be fetched and returned. For multithreading support, this is implemented as a ***split-phased transaction***, where the first phase involves a sending a **request** to the desired node and the second phase involves waiting for a **reply**. When multithreading is enabled, a thread scheduler is invoked between these two operations to start another parallel thread while the request is being processed.

2.5 Optimized Address Translation

One of the first things we noticed about the VISA address translation scheme is that the overhead for translation is minimal when compared with the time required to perform a remote reference, but dominates the time required for a local reference. In addition to exposing the routine which finds a desired range map entry so that range_map entries can be stored locally to avoid repeated searching of the range_map_table, we have designed and implemented an optimization that *eliminates* the need for an address translation when the access is local. We introduce a new function, called *visa_opt*, which re-writes the virtual base address with the structure's ***optimized*** base address, and establishes a pair of "water mark" registers to hold the low and high values of the range corresponding to the local_base. The optimized_base is the local_base minus the offset necessary to generate a global address that will result in a local access. For example, suppose an array of 40 integers (4 bytes each) is allocated using *block_map* among 4 nodes, as depicted in Figure 6, where the local_base values can be differ-

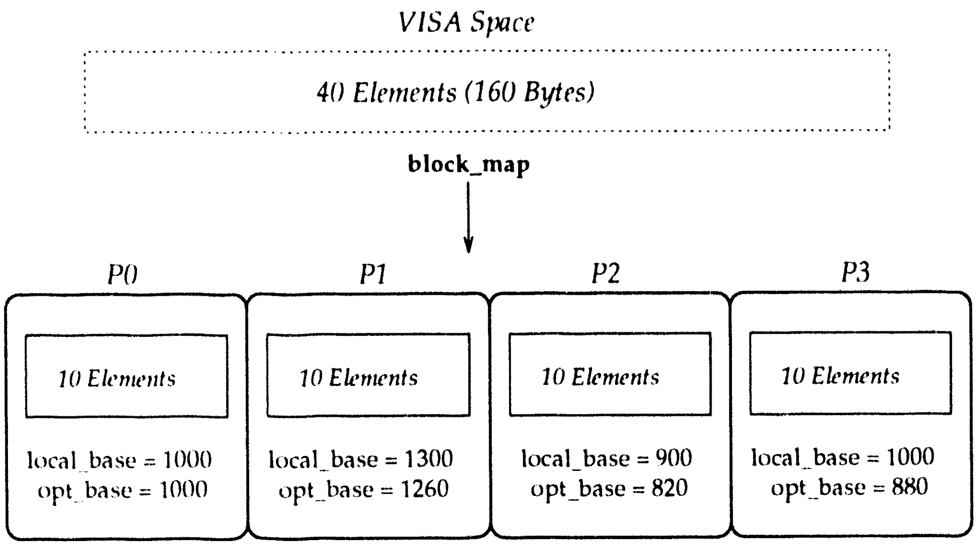


Figure 5: Sample VISA data structure with computed optimized_base values

ent for each node, which is possible since each node manages its local memory independently of the other nodes. Each processor would allocate local storage for *blocksize = 10* elements (40 bytes), and set the local_base accordingly. If, for example, the third node wishes to optimize the base address for this structure, then the optimized value is the local_base minus 20 elements (80 bytes), corresponding to the two blocks of 10 elements each that proceed it in the distribution. Once the base address for a structure has been optimized, any further access to this structure, represented as some offset from the base, will be checked against the low and high water marks. If the computed address falls within the water marks, then the access can proceed without translation, otherwise the address is passed along to the VISA access routines for general address translation and proper remote handling. Special macros are defined to perform the water mark checks, so that the total overhead for a local access has been reduced to the time required for three comparisons.

3 Performance

We now examine the performance of a two-dimensional smoothing algorithm called *Laplace*, which is a smoothing algorithm that uses a five-point stencil over a two-dimensional array. The Sisal code for Laplace is given in Figure 7.

This problem highlights two-dimensional Sisal arrays, mapping functions, and the ability to combine task distribution with data distribution at a multi-dimensional level. The mapping functions attempt to minimize the edges (or boundary elements) in the distribution, since boundary elements require remote references.

Since the Sisal compiler does not support true rectangular arrays, the matrix in this program is implemented as an array of rows, where each row is a one-dimensional array of values. Also, each Sisal array is represented using three data structures, two descriptor structures and the actual array, and each of these data structures is placed into VISA space, requiring an additional VISA descriptor for each. Thus, for an $n \times n$ matrix, the Sisal compiler generates $6n + 6$ data structures, $5n + 5$ of which are replicated. Finally, since Laplace is an iterative algorithm, the compiler generates two additional swap matrices, bringing the total data structure count for the program to $3(6n+6)$, of which $3(5n + 5)$ are replicated across all nodes. As we will see, handling multi-dimensional arrays in this manner has a profoundly detrimental effect on the performance of the program.

Laplace employs a five-point stencil computation, which implies that the computation of all boundary elements for a give distribution will require remote references. Thus our first intuition is to minimize the number of elements on the distribution boundaries. We examine two matrix mapping functions to see how effective they are at minimizing remote references. For our comparisons, let us assume that the matrix size is $256 \times 256 (n = 256)$ and we are using 16 processors $(p = 16)$.

```
% laplace.sis

define main

type OneD = array[double_real];
type TwoD = array[OneD];

function 2d_fill (N: integer returns TwoD)
    for I in 1,N cross J in 1,N
        el := if mod(I+J,2) = 0 then
            double_real(1.0)
        else
            double_real(N)
        end if
    returns array of el
    end for
end function % 2d_fill

function laplace (Init_M: TwoD; N,KMax: integer returns TwoD)
    for initial
        K := 1;
        M := Init_M
    repeat
        K := old K + 1;
        M := for I in 1,N cross J in 1,N
            nM := if I=1|I=N|J=1|J=N then
                old M[I,J]
            else
                old M[I,J] / double_real(2.0) +
                    (old M[I-1,J] + old M[I+1,J] + old M[I,J-1] +
                    old M[I,J+1]) / double_real(8.0)
            end if
        returns array of nM
        end for
    until K >= KMax
    returns value of M
    end for
end function % laplace

function main (N,K : integer returns TwoD)
    let
        A := 2d_fill (N)
    in
        laplace (A, N, K)
    end let
end function % main
```

Figure 6: Sisal code for laplace function

- The *matrix_row_map* mapping function allocates b contiguous rows to each processor, where $b = n/p$. This mapping scheme eliminates all of the interior remote references, leaving only those on every b-th row boundary. Thus we have a total of $(p-2)2(n-2)+2(n-2)$ remote references , which is 7,620 for our example of a 256 × 256 matrix.
- In the *matrix_block_map* mapping function the p processors are arranged in a $\sqrt{p} \times \sqrt{p}$ grid, each owning a $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of the matrix. In the example case this would lead to a 4x4 grid with 128x128 elements per processor, with 4 corner processors performing 254 remote references each, 8 side processors performing 382 remote references each, and 4 interior processors performing 512 remote references each, producing total of 6,120 remote references.

When evaluating the performance of Laplace we must select a problem size, which raises the problem that a problem size which fits into the memory of a single node is not large enough to saturate an order of magnitude larger machine configuration, and a problem size that saturates a large machine configuration does not fit into one memory. Therefore, we create three processor configuration groups: 1,2,4,& 8 nodes, 4,8,16, & 32 nodes, and 16, 32, 64, & 128 nodes. We use the same array size within each processor group, and increasingly larger array sizes between the groups. We measure the efficiency of Laplace relative to the smallest configuration in each group, and call this measure *relative efficiency (REff)*.

The results of running Laplace with the two matrix mapping functions are given in Table 3, where *REff* is defined as $REff=(T_\alpha/(T_{n\alpha} * n)) * 100$, where α is the number of processors for the base configuration in a group, T_α is the execution time for this base configuration, and $T_{n\alpha}$ is the execution time on $n\alpha$ processors, and Sp gives the speedup, in terms of execution time, of the block map over the row map. These results are clearly disappointing. The poor performance is caused by the need for replicating the administrative data structures of the two dimensional arrays, creating $3(3n+3)$ Sisal data structures and $3(3n+3)$ VISA data structures (range map entries), for a total of $3(6n+6)$ administrative data structures, $3(5n+5)$ of which must be replicated. In a one PE machine there is no broadcast, hence the much better sequential performance. Dealing with N-dimensional arrays in this fashion works for shared memory machines, but is clearly unacceptable in a distributed memory machine. The correct way to solve this problem is to have true N-dimensional arrays in Sisal, resulting in one descriptor for the whole structure. Sisal 2.0 [2] defines true N-dimensional arrays and gives a method of distributing regions of these arrays. A quick fix to this problem given the current version of Sisal is to represent the matrix by a one dimensional structure and rewrite the inner loop of Laplace as follows:

```
M := for k in 1,n*n
       i := (k-1)/n + 1; j := mod(k,n);
     nM := if i=1|i=n|j=1|j=0
           then old M[k]
           else old M[k] / 2.0 +
                (old M[k-n] + old M[k+n] +
                 old M[k-1] + old M[k+1]) /
                 8.0
           end if
    returns array of nM
   end for
```

Table 4 presents the results of the improved Laplace program, using a one dimensional array and a block mapping function that corresponds to the *matrix_row_map* function, where Sp gives the speedup of the true 2D array implementation over the Sisal 2D array implementation. The results are significantly better, and can be further improved by using *multithreading* to hide part of the cost from a remote reference.

Table 5 gives the performance results for Laplace with multithreading, where the number of threads is 16 for all multithreading cases, and Sp gives the speedup of the multithreading case over the non-multithreading case. For this experiment, we use the improved version of Laplace that utilizes a one-dimensional array rather than the Sisal two dimensional arrays. Going from no message passing (1 PE) to message passing (2 PEs) slows the program down in the non-multithreading case, but since the compute/communicate ratio of Laplace is high (the computation of $O(n^2)$ array elements requires $O(n)$ remote references), parallelizing this code pays off and performance is regained. At the same time, multithreading is effective for Laplace as there are enough remote reference to cover the multithreading overheads, which gives rise to speedups of between 1.26 and 1.65. Clearly, multithreading is effective at tolerating the remote references for this program.

Finally, this program could be further improved by employing compiler generate block moves allowing a whole row to be communicated between nodes. This is a case where making the compiler aware of the distributed memory architecture, and performing the appropriate analysis and optimization, will provide re-

PEs	Matrix Size	*matrix_row_map* Time (s)	*matrix_row_map* REff (%)	*matrix_block_map* Time (s)	*matrix_block_map* REff (%)	*matrix_block_map* Sp
1	256x256	17.1284	100.0	18.6020	100.0	0.92
2	256x256	27.8498	30.8	34.3491	27.1	0.81
4	256x256	19.4244	22.0	24.4763	19.0	0.79
8	256x256	20.7248	10.3	22.7965	10.2	0.90
4	512x512	52.2188	100.0	53.0218	100.0	0.98
8	512x512	47.8333	54.5	51.3777	51.6	0.93
16	512x512	58.5527	22.3	63.2369	21.0	0.92
32	512x512	89.9674	7.3	97.4665	6.8	0.92
16	1024x1024	129.7875	100.0	134.5676	100.0	0.96
32	1024x1024	185.1608	35.0	199.6552	33.7	0.93
64	1024x1024	318.1579	10.2	346.8237	9.7	0.92
128	1024x1024	529.3233	3.1	646.9596	2.6	0.82

Table 3: Performance of 2D Laplace, row versus block map, 10 iteration

PEs	Matrix Size	*Sisal 2D Arrays* Time (s)	*Sisal 2D Arrays* REff (%)	*True 2D Arrays* Time (s)	*True 2D Arrays* REff (%)	*True 2D Arrays* Sp
1	256x256	17.1284	100.0	16.6974	100.0	1.03
2	256x256	27.8498	30.8	18.5326	45.0	1.50
4	256x256	19.4244	22.0	12.4936	33.4	1.55
8	256x256	20.7248	10.3	8.7825	23.8	2.36
4	512x512	52.2188	100.0	41.3844	100.0	1.26
8	512x512	47.8333	54.5	25.4780	81.2	1.88
16	512x512	58.5527	22.3	17.4767	59.2	3.35
32	512x512	89.9674	7.3	14.5955	35.4	6.16
16	1024x1024	129.7875	100.0	50.6113	100.0	2.56
32	1024x1024	185.1608	35.0	36.0326	70.2	5.14
64	1024x1024	318.1579	10.2	29.1644	43.4	10.91
128	1024x1024	529.3233	3.1	26.8367	23.6	19.73

Table 4: Performance of improved Laplace (`matrix_row_map`)

		No Multithreading		*Multithreading*		
PEs	Matrix Size	Time (s)	REff (%)	Time (s)	REff (%)	Sp
1	256x256	16.6974	100.0			
2	256x256	18.5326	45.0	11.2740	74.1	1.65
4	256x256	12.4936	33.4	8.9311	46.7	1.39
8	256x256	8.7825	23.8	6.8160	30.6	1.27
4	512x512	41.3844	100.0	26.7381	100.0	1.54
8	512x512	25.4780	81.2	18.2013	73.5	1.39
16	512x512	17.4767	59.2	13.6886	48.8	1.27
32	512x512	14.5955	35.4	11.5710	28.9	1.26
16	1024x1024	50.6113	100.0	36.4239	100.0	1.39
32	1024x1024	36.0326	70.2	27.3604	66.6	1.31
64	1024x1024	29.1644	43.4	23.1235	39.4	1.26
128	1024x1024	26.8367	23.6	20.3319	22.4	1.32

Table 5: Performance of Laplace with multithreading, MT=16, 10 iterations

sults superior to the general runtime approach.

4 Related Research

In [3] shared memory implementations of Sisal and Fortran are compared. The shared memory implementation of Sisal compares favorably with Fortran on a wide variety of benchmarks. By providing a virtual shared memory runtime system, we have taken the shared memory implementation to distributed memory machines. In [6] we introduced the design of our task management system, in [8] we quantify the characteristics of software multithreading, and in [7] we present the entire runtime system and provide experiments that measure the relative effects of certain design decisions using a larger set of test programs.

Another area of research that offers a language-independent shared memory paradigm is Distributed Shared Memory [1, 12, 16]. However, the inability to couple parallel tasks tightly with the distribution of data, controlled implicitly by the operating system, can result in misalignment, causing excessive message passing. Also since the granularity of sharing data in these systems is often very large (typically a page), contention, or *false sharing* can occur, in which two unrelated data items exist on the same sharable unit, prohibiting simultaneous access. Since the sharable unit in VISA is an individual data structure, false sharing does not occur.

The most common alternatives to programming distributed memory multiprocessors using an explicit parallel language with message passing are distributed memory language compilers, such as FortranD [9], Kali [11], and Superb [19]. These systems offer the advantage of implicit management for both tasks and memory, and allow the programmer to use a familiar programming paradigm: sequential shared memory. Although these systems have had success in implementing some applications, there are several problems that have kept them from wide-spread use:

- Parallelizing a sequentially written program requires extensive dependence analysis that can be hampered with common imperative programming phenomena such as aliasing [17].
- Due to the complexity of these compilers and the difficulties in porting them to new machines, their availability is limited to only of few of the currently available distributed memory multiprocessor systems. As stated earlier, such a compiler is not commercially available for the nCUBE/2.
- Though parallelizing/vectorizing compilers have proven to be successful for some applications on shared memory multiprocessors and vector processors with shared memory, they are still largely unproven for distributed memory multiprocessors.

5 Conclusions

We have introduced the design and implementation of a runtime-based approach to providing a shared memory paradigm and implicit memory management for a distributed memory implementation of Sisal. We have also examined the performance of a two-dimensional smoothing algorithm using Sisal and augmented with VISA to run on the nCUBE/2. The results clearly demonstrate the need for true rectangular arrays, as the current implementation of two-dimensional arrays in Sisal creates an excessive number of supporting data structures, most of which need to be replicated. This not only wastes an enormous amount of local memory, but clogs the network with replication messages, resulting in dismal performance. However, if the two-dimensional arrays are "flattened" into one-dimensional arrays, then reasonable performance can be achieved, upon which multithreading can improve.

VISA represents only the first step in achieving an efficient and competitive distributed memory implementation of Sisal, and efforts should now be concentrated on the compiler to add explicit knowledge of a distributed memory system, as well as generating the appropriate primitives for the distributed memory runtime system that we have created. Only then can optimizations concerning task and data layout be implemented, which are necessary for performance that will compete with other distributed memory programming approaches.

References

[1] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Shared memory for distributed memory multiprocessors. Technical Report Rice COMP TR89-91, Rice University, April 1989.

[2] A. P. W. Böhm, D. C. Cann, J. T. Feo, and R. R. Oldehoeft. SISAL 2.0 reference manual. Technical Report CS-91-118, Computer Science Department, Colorado State University, Fort Collins, CO, November 1991.

[3] David Cann. Retire Fortran? A debate rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[4] Manish Gupta and Prithviraj Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[5] Matthew Haines and Wim Böhm. Thread management in a distributed memory implmentation of Sisal. In *International Symposium on Computer Architecture, Workshop on Dataflow Computing*, May 1992.

[6] Matthew Haines and Wim Böhm. Towards a distributed memory implementation of Sisal. In *Scalable High Performance Computing Conference*, pages 385–392. IEEE, April 1992.

[7] Matthew Haines and Wim Böhm. On the design of distributed memory Sisal. *Journal of Programming Languages*, 1:209–240, 1993.

[8] Matthew Haines and Wim Böhm. Task management, virtual shared memory, and multithreading in a distributed memory implementation of Sisal. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *Parallel Architectures and Languages Europe*, pages 12–23. Springer-Verlag Lecture Notes in Computer Science, June 1993.

[9] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[10] Intel iPSC/860 specifics. Brochure, 1991.

[11] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[12] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, September 1986.

[13] J. R. McGraw, S. K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[14] nCUBE, Beaverton, OR. *nCUBE/2 Technical Overview, SYSTEMS*, 1990.

[15] Michael O'Boyle and G.A. Hedayat. Data alignment: Transformations to reduce communication on distributed memory architectures, April 1992.

[16] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Unifying synchronization and data transfer in maintaining coherence of distributed shared memory. Technical Report GIT-CS-88/23, Georgia Institute of Technology, June 1988.

[17] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An emperical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[18] Thinking Machines Corporation, Cambridge, Massachusetts. *CM5 Technical Summary*, October 1991.

[19] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1986.

Appendix A: VISA Primitives

- *Allocation*
  - V_ADDRESS **visa_malloc** (int *nelems*, int *size*, map_function *map*, int *map_arg*)

    This function allocates a block of VISA space (*nelems* * *size* bytes), which will be distributed according to *map*, and returns a pointer to the start of the allocated space. A range_map entry is also created and distributed among the nodes, and local space is allocated, according to the map, to store the data structure.
- *Deallocation*
  - void **visa_free** (V_ADDRESS *address*)

    This function returns the given portion of VISA space to the free pool, removes the corresponding range_map entry from each of the range_map tables, and deallocates the local storage used for storing the structure.
- *Access*
  - range_map_type * **find_rm** (V_ADDRESS *address*)

    Return a pointer to the range_map entry corresponding to the given VISA address. This pointer is then passed into each of the access routines as an argument so that the fetch does not have to be done for each access.
  - char **visa_get_c** (V_ADDRESS *address*, range_map_type **rm*)
    int **visa_get_i** (V_ADDRESS *address*, range_map_type **rm*)
    float **visa_get_f** (V_ADDRESS *address*, range_map_type **rm*)
    double **visa_get_d** (V_ADDRESS *address*, range_map_type **rm*)

    These functions return the desired value from the given VISA address. If the range_map entry *rm* is not defined, then the corresponding range_map entry for this structure will be fetched, which is true for all of the access functions.
  - void **visa_get_m** (POINTER *data*, int *size*, V_ADDRESS *address*, range_map_type **rm*)

    This function copies the block of data starting at the given VISA address and for a length of *size* into the local address pointed to by *data*.
  - void **visa_put_c** (char *value*, V_ADDRESS *address*, range_map_type **rm*)
    void **visa_put_i** (int *value*, V_ADDRESS *address*, range_map_type **rm*)
    void **visa_put_f** (float *value*, V_ADDRESS *address*, range_map_type **rm*)
    void **visa_put_d** (double *value*, V_ADDRESS *address*, range_map_type **rm*)

    These functions place *value* into the given VISA address location.
  - void **visa_put_m** (POINTER *data*, int *size*, V_ADDRESS *address*, range_map_type **rm*)

    This function copies the local data block of size *size* and pointed to by *data* into the given VISA address location.
  - void **visa_update_c** (uchar *red*, char *value*, V_ADDRESS *address*, range_map_type **rm*)
    void **visa_update_i** (uchar *red*, int *value*, V_ADDRESS *address*, range_map_type **rm*)
    void **visa_update_f** (uchar *red*, float *value*, V_ADDRESS *address*, range_map_type **rm*)
    void **visa_update_d** (uchar *red*, double *value*, V_ADDRESS *address*, range_map_type **rm*)

    These functions update the value stored in the given VISA address with *value*, according to the reduction *red*. Currently supported reductions include *V_SUM* and *V_PRODUCT*.

Developing a high-performance FFT algorithm in Sisal for a vector supercomputer

John Feo and David Cann
Lawrence Livermore National Laboratory
Livermore, CA 94551

Abstract

Functional languages provide a level of program abstraction devoid of most details regarding implementation and architecture. Memory management, task scheduling, communication, synchronization, and resource management are implied by the language's semantics, and are not programmed explicitly by the programmer. While these properties of functional languages are attractive, they may be detrimental if the programmer's objective is to realize high performance. In this paper, we discuss these programming issues and study the difficulty of expressing a machine-specific algorithm in a functional language. We chose to study the Fast Fourier Transform, the Cray C90, and Sisal. We present an implementation of the Fast Fourier Transform designed specifically for the Cray hardware, and explain how to express the computation in Sisal. Despite its complexity, the Sisal code runs in constant memory, exhibits good speedup, and executes close to the achievable performance limits of the machine.

1 Introduction

Functional programming languages provide a level of program abstraction devoid of most implemental and architectural details. The functional programmer works at or near the level of mathematics. Allocation and deallocation of memory, identification of concurrent tasks, communication, synchronization, and resource management are implied by the semantics of the language, and are not programmed explicitly by the user. While these properties reduce the cost of developing correct, determinate parallel programs, they also reduce the user's control over how programs execute. Fine control over program execution is one way to achieve high performance.

Since functional languages do not provide execution control, the functional programmer must develop algorithms that naturally exploit the target architecture, and then rely on the compiler and runtime system to compile and execute the code efficiently. In this paper, we study the difficulty of expressing machine-specific algorithms in a functional language. For the algorithm, we chose the Fast Fourier Transform (FFT). This algorithm is the kernel of many large scientific applications, and has been studied extensively. Since speed is paramount in kernel routines, FFT algorithms are usually written to machine specifications. Recently, several papers have appeared in the literature discussing the implementation of FFT algorithms in functional languages [1,2,9]. For the machine, we chose the Cray C90, a sixteen processor vector supercomputer. Each processor is capable of executing one billion floating point operations per second. Writing high-performance code for the Cray is a formidable challenge. For the language, we chose Sisal [6], a functional language developed by Lawrence Livermore National Laboratory and Colorado State University. Sisal programs have achieved good performance on a variety of multiprocessor computers [4].

In section two, we present the Fast Fourier Transform and identify important optimizations. In section three, we introduce the Cray C90 architecture and give requirements necessary to achieve high performance. In section four, we present an implementation of the FFT algorithm designed specifically for the Cray hardware, and show how to express the algorithm in Sisal. In section five, we explain how the code is compiled and give performance numbers. In section six, we present our conclusions and observations.

2 Fast Fourier Transform

We can model the state of many physical processes as a function of time or frequency. The two models are related by the Fourier Transform,

$$\mathbf{H}(f) = \int_{-\infty}^{\infty} \mathbf{h}(t)e^{2\pi i f t}\,\partial t \qquad (1)$$

and

$$\mathbf{h}(t) = \int_{-\infty}^{\infty} \mathbf{H}(f)e^{2\pi i f t}\,\partial f \qquad (2)$$

The two forms are referred to as the decimation in frequency and the decimation in time, respectively. The discrete form of Equation (1) for n samples is

$$\mathbf{H}_f = \sum_{k=0}^{n-1} \mathbf{h}_k e^{2\pi i f\left(\frac{k}{n}\right)}, \quad 0 \le f \le n-1 \qquad (3)$$

As formulated the Fourier Transform is $O(n^2)$. But an $O(n \log n)$ algorithm exists based on the principal of divide-and-conquer. The algorithm is known as the Fast Fourier Transform (FFT) [5]. An elegant formulation of the algorithm ascribed to Danielson and Lanczos is presented in [10]. Observe that the sum in Equation (3) may be divided into two sums: one over the even components of the data and the other over the odd components of the data,

$$\mathbf{H}_f = \sum_{k=0}^{\frac{n}{2}-1} \mathbf{h}_{2k} e^{2\pi i f\left(\frac{2k}{n}\right)} + \sum_{k=0}^{\frac{n}{2}-1} \mathbf{h}_{2k+1} e^{2\pi i f\left(\frac{2k+1}{n}\right)} \tag{4}$$

where $0 \le f \le n-1$. We may now rewrite Equation (4) as the sum of two Fourier Transforms of size $n/2$

$$\mathbf{H}_f = \sum_{k=0}^{\frac{n}{2}-1} \mathbf{h}_{2k} e^{2\pi i f\left(\frac{2k}{n}\right)} + e^{\left(\frac{2\pi i f}{n}\right)} \sum_{k=0}^{\frac{n}{2}-1} \mathbf{h}_{2k+1} e^{2\pi i f\left(\frac{2k}{n}\right)} \tag{5}$$

$$= \mathbf{H}_f^{\text{even}} + e^{\left(\frac{2\pi i f}{n}\right)} \mathbf{H}_f^{\text{odd}}$$

The complexity of Equation (5) is $O(n \log n)$. Note that f ranges from 0 to $n - 1$ in the last statement, but since H^{even} and H^{odd} are periodic in f with length $n/2$, all values are present. Figure 1 illustrates the recursive character of the Cooley–Tukey FFT algorithm. The repeating pattern is known as a butterfly. The heavy lines on the left- and right-hand sides highlight a one- and a two-stage butterfly, respectively. The pattern is simple and regular, but there is a problem—the results are out of order. A post-processing step to put data in the correct order is required. To find the correct position for the datum at index i, we reverse the bits of the index. For example, if n is 16, then 0 goes to 0, 1 goes to 8, 2 goes to 4, 3 goes to 12, etc. The permutation is known as bit reversal.

There are several ways to improve the performance of the FFT:

1. Compute two butterfly stages at once (radix-4). A one-stage, two-input butterfly requires one complex multiplication and two complex additions, or a total of 10 floating point operations. Let a and b be the inputs of a one-stage butterfly, then the outputs are

$$\begin{pmatrix} a + e^{\frac{\alpha\pi}{2}} c \end{pmatrix} \qquad \begin{pmatrix} a - e^{\frac{\alpha\pi}{2}} c \end{pmatrix} \tag{6.1}$$

A two-stage, four-input butterfly requires three complex multiplications and eight complex additions, or a total of 34 floating point operations. Let a, b, c, and d be the inputs of a two-stage butterfly, then the outputs are

$$\begin{gathered} \left(a + e^{\frac{\alpha\pi}{2}} c\right) + e^{\frac{\alpha\pi}{4}} \left(b + e^{\frac{\alpha\pi}{2}} d\right) \\ \left(a + e^{\frac{\alpha\pi}{2}} c\right) + e^{\frac{5\alpha\pi}{4}} \left(b + e^{\frac{\alpha\pi}{2}} d\right) \\ \left(a + e^{\frac{3\alpha\pi}{2}} c\right) + e^{\frac{3\alpha\pi}{4}} \left(b + e^{\frac{3\alpha\pi}{2}} d\right) \\ \left(a + e^{\frac{3\alpha\pi}{2}} c\right) + e^{\frac{7\alpha\pi}{4}} \left(b + e^{\frac{3\alpha\pi}{2}} d\right) \end{gathered} \tag{6.2}$$

But, since $e^{\pi} = -1$ and $e^{\pi/2} = i$, we may rewrite Equation (6.2) as

$$\begin{gathered} \left(a + e^{\frac{\alpha\pi}{2}} c\right) + \left(e^{\frac{\alpha\pi}{4}} b + e^{\frac{3\alpha\pi}{4}} d\right) \\ \left(a + e^{\frac{\alpha\pi}{2}} c\right) - \left(e^{\frac{\alpha\pi}{4}} b + e^{\frac{3\alpha\pi}{4}} d\right) \\ \left(a - e^{\frac{\alpha\pi}{2}} c\right) + i \left(e^{\frac{\alpha\pi}{4}} b - e^{\frac{3\alpha\pi}{4}} d\right) \\ \left(a - e^{\frac{\alpha\pi}{2}} c\right) - i \left(e^{\frac{\alpha\pi}{4}} b - e^{\frac{3\alpha\pi}{4}} d\right) \end{gathered} \tag{7}$$

Since four one-stage butterflies comprise a two-stage butterfly, a radix-4 algorithm executes 15% fewer instructions.

2. Compute the first two butterfly stages separately. Use constants, and not names, for the exponential terms. Since the exponential values are either 1, –1, i, and –i, we can eliminate the multiplication operations.

3. Compute all the exponential terms required for the computation before initiating the algorithm, and store them in a manner that is convenient for retrieval by each stage.

4. Permute the results of each butterfly stage such that the results of the final butterfly stage are in the correct order. This optimization eliminates the post-processing step to do the bit-reversal. It is important, however, that the permutations at each stage preserve the computation's regularity; otherwise, the savings will be spent on complex array index calculations.

3 The Cray C90

The National Energy Research Supercomputer Center's Cray C90 (a.nersc.gov) is a sixteen processor, vector supercomputer. The machine's clock speed is 4.167 nanoseconds. It has 258M words of main memory organized in 1024 banks. The memory speed is 23 clocks.

Each processor has two vector multiply–add pipelines fed by sets of 128-element vector registers. The peak execution speed of each processor is 1 gflop per second. The geometric mean of the Livermore Loops [7] is 86.3 mflops with a peak speed of 826 mflops [8].

In addition to the vector hardware, each processor has a scalar add and multiply unit. The execution time of scalar operations is much less than the execution time of vector operations. The geometric mean of the Livermore Loops executed as all scalar operations is 22.25 mflops with a peak speed of 53 mflops [8]. Only highly vectorized codes achieve peak performance on the Cray C90. Moreover, the lengths of the vectors are important—longer the vectors, better the performance.

Peak performance depends on keeping the vector registers full. The Cray hardware can issue a memory read per cycle; but if consecutive reads address the same memory bank (a bank conflict), then the second read will be delayed until the memory is refreshed. Since memory is interleaved by words, vector strides of one are optimal.

In summary, to achieve high-performance on the Cray C90, a programmer must:

1. implement a vector algorithm,
2. maintain long vector lengths throughout the computation, and
3. avoid vector strides of two or multiples of two.

4 The Sisal program

To realize a high-performance FFT algorithm in Sisal for the Cray C90, we have to address each of the seven points raised in the previous two sections. Since we can not explicitly encode implemental details in a Sisal program, we have to address the seven requirements through algorithm design.

For the purposes of this discussion, we assume the size of the input data is n, a power of two. Since Sisal does not have a complex data type, we store the data as two vectors of real values, `xre` and `xim`. `xre[i]` and `xim[i]` are the real and imaginary components, respectively, of the i-th datum. Using two vectors, instead of a vector of records, simplifies the code and aids vectorization.

4.1 Two-stage butterflies

Since a Sisal function can return multiple values, encoding a two-stage butterfly in Sisal is easy. We define the function `fft_4_2` of fourteen inputs (the real and imaginary components of the four butterfly inputs and three exponential terms) and eight output values (the real and imaginary components of the four butterfly outputs). Its body consists of the three complex multiplications and eight complex additions implied by Equation (7). The function is the algorithm's central kernel.

4.2 The initial iterations

We compute the initial butterfly stages in a function named `level_1`. Succeeding stages are computed in an iterative loop in the body of the main function. If n is an even power of two, `level_1` computes the first two-stages; else, it computes the first three stages. The function `level_1` calls `fft_4_2` to compute each butterfly. We pass the exponential terms as constants and rely on the Sisal compiler to eliminate the unnecessary multiplication operations.

4.3 Storing the exponential terms

To reduce index calculations and improve vectorization, we store the exponential terms in three pairs of two-dimensional arrays. Each pair of arrays provides the real and imaginary components of one of the three exponential terms in Equation (7). Each array has $\log_4 n$ rows, numbered 0 to $\log_4 n - 1$. If n is an even power of two, row i has 4^i values; otherwise, it has $2 * 4^i$ values. The total memory requirement is $2n$ words. We do store some terms more than once (the minimum storage requirement is $1.5\,n$), but using the extra space significantly reduces execution time.

4.4 Vectors, long vectors, and bit reversal

We now describe logically several possible vector implementations of the FFT algorithm. Say that at level i, we divide the data into 4^i packs, and we divide each pack into four sections. Figure 2 depicts such a data decomposition for $n = 64$. The labels a, b, ..., z, 1, 2, ..., 38 represent the 64 data values. The implied algorithm is ideal for a concurrent, vector computer such as the Cray C90. It consists of a set of parallel vector tasks. There is one task per pack, and the lengths of the vectors are $n/4^{i+1}$. Additionally, we may execute each vector task on a single processor or on multiple processors as necessary to optimize performance. Figure 3 illustrates one possible permutation of the outputs of each level to effect bit reversal in place. Notice that only the relative positions of the packs change. The positions of the data within a pack do not change. Thus, the computation's regularity is maintained. The algorithm's one drawback is that the lengths of the vectors shrink from $n/4$ to 1 as the computation proceeds (we refer to the algorithm as *long-to-short*). The short vector lengths at the end cause the final iterations to execute slowly [Table I].

Figure 4 depicts an alternative vector computation (short-to-long) similar in nature to the algorithm described in the previous paragraph. In this algorithm, the vector lengths grow from 1 to $n/4$ as the computation proceeds. Its drawback is that the initial iterations execute slowly. We can maintain long vector lengths throughout the computation by executing the first algorithm for the first half of the computation and then switching to the second algo-

# tasks	vector length	seconds
2^2	2^{16}	.0080
2^4	2^{14}	.0080
2^6	2^{12}	.0083
2^8	2^{10}	.0094
2^{10}	2^8	.0155
2^{12}	2^6	.0471
2^{14}	2^4	.1755
2^{16}	2^2	.6911

Table 1 – The effect of vector length on performance

rithm for the second half of the computation. Unfortunately, the second algorithm expects the data in a different order than it is written by the first algorithm (compare the order of the data at the end of the first level in Figures 3 and 4). So, we need for a third algorithm (switch) that reads the data as written by the first algorithm and writes the data as read by the second algorithm.

In an imperative language, the programmer can read and write data in any order. However, the Sisal programmer is limited to the ways in which the different array expressions in Sisal construct arrays. For example, the `for` expression gathers array elements in a determinate, prescribed manner. The *i*-th iteration of a `for` expression defines the *i*-th elements of each resultant array. Permutation of array elements during construction is not permitted. One might think that this constraint would make it impossible to express the algorithms described in the previous paragraphs. But careful study of Figures 3 and 4 reveals that we can built the levels in sections such that the constraint holds for each section. After the sections are built, we can paste them together in the correct order.

For example, consider the second level butterflies in Figure 3. The first four inputs to the first butterfly are a, e, i, and m. The four outputs are the first values in the four sections shown at the third level

$\underline{a}$ b c d . . .

$\underline{i}$ j k l . . .

$\underline{e}$ f g h . . .

$\underline{m}$ n o p . . .

Thus, there exists an *i*-to-*i* correspondence between loop iteration and result value. We have the vector loop return four arrays, and catenate the arrays to build the four sections. We then catenate the sections together, interchanging the second and third sections, to build the full data array at the third level.

As a second example, consider the first section of the data array at level three in Figure 4

a 7 q 23 i 15 y 31 e 11 u 27 m 19 3 35

Now divide the section into four parts

a 7 q 23

i 15 y 31

e 11 u 27

m 19 3 35

Notice that the first four output values of the first butterfly, a, e, i, and m, are the first values of the four parts. Again, there exists an *i*-to-*i* correspondence between loop iteration and result value. We have the vector loop return four arrays, and catenate the arrays together, interchanging the second and third arrays, to build the sections. We then catenate the sections together to build the full data array at the third level.

Sisal pseudo-code for long-to-short, short-to-long, and switch algorithms is

```
function long_to_short(...
        returns array[real], array[real])

  let

    AAre, AAim, BBre, BBim,
    CCre, CCim, DDre, DDim :=
    for j in 0, number_of_packs - 1

      % start address of the
      % four sections of pack j
      p0, p1, p2, p3 := ... ;

      Are, Aim, Bre, Bim,
      Cre, Cim, Dre, Dim :=
      for k in 0, (size_of_pack / 4) - 1
        are, aim, bre, bim,
        cre, cim, dre, dim := fft_4_2(...)
       returns array of are   array of aim
               array of bre   array of bim
               array of cre   array of cim
               array of dre   array of dim
       end for

   returns value of catenate Are
           value of catenate Aim
           value of catenate Bre
           value of catenate Bim
           value of catenate Cre
           value of catenate Cim
           value of catenate Dre
           value of catenate Dim
   end for
  in
    AAre || CCre || BBre || DDre,
    AAim || CCim || BBim || DDim
  end let

end function % long_to_short


function short_to_long(...
        returns array[real], array[real])

  for j in 0, (number_of_packs / 4) - 1

    % start address of packs
    % (4 * j) + 1, + 2, + 3, and + 4
    p0, p1, p3, p4 := ... ;

    Are, Aim, Bre, Bim,
    Cre, Cim, Dre, Dim :=
    for k in 0, size_of_pack - 1
      are, aim, bre, bim,
      cre, cim, dre, dim := fft_4_2(...)
    returns array of are   array of aim
            array of bre   array of bim
            array of cre   array of cim
            array of dre   array of dim
    end for

   returns
    value of catenate  (Are||Cre||Bre||Dre)
    value of catenate  (Aim||Cim||Bim||Dim)
   end for

end function % short_to_long


function switch(...
        returns array[real], array[real])

  for j in 0, (size_of_pack / 4) - 1

    Are, Aim, Bre, Bim,
    Cre, Cim, Dre, Dim :=
    for k in 0, number_of_packs - 1

      % address of value j in
      % the four sections of pack k
      p0, p1, p2, p3 := ... ;

      are, aim, bre, bim,
      cre, cim, dre, dim := fft_4_2(...)

    returns array of are   array of aim
            array of bre   array of bim
            array of cre   array of cim
            array of dre   array of dim
    end for

   returns
    value of catenate  (Are||Cre||Bre||Dre)
    value of catenate  (Aim||Cim||Bim||Dim)
   end for

end function % switch
```

4.5 Strides of two or multiples of two

The inner **for** expression of function `long_to_short` traverses the four sections of a pack value-by-value. The inner **for** expression of function `short_to_long` traverses four packs value-by-value. In both cases, the **for** expression is a vector loop with stride one. But, the inner **for** expression of function `switch` jumps from pack to pack. Its stride is the size of a pack, i.e., a power of two. The resulting bank conflicts degrade performance. We can eliminate the offending stride by inserting a one word gap between packs. We define a fourth function `gap` that executes the long-to-short algorithm, but inserts a 0 between packs. It is identical to `long_to_short` except for the returns clause of the outer **for** expression,

```
returns
   value of catenate array_addh(are,  0.0d0)
   value of catenate array_addh(aim,  0.0d0)
   value of catenate array_addh(bre,  0.0d0)
   value of catenate array_addh(bim,  0.0d0)
   value of catenate array_addh(cre,  0.0d0)
   value of catenate array_addh(cim,  0.0d0)
   value of catenate array_addh(dre,  0.0d0)
   value of catenate array_addh(dim,  0.0d0)
```

The "padding" of vectors to improve vector or cache performance is a well known optimization. It is used often by scientific programmers.

To summarize, we have presented an FFT algorithm designed specifically for the Cray C90. It addresses the seven requirements for high-performance listed in Sections 2 and 3. We have shown how to write the algorithm in Sisal. The Sisal code consists of five major routines: `level_1`, `long_to_short`, `gap`, `switch`, and `short_to_long`. The functions are called in the order listed and compute, respectively, levels 1, 2 through *middle* – 2, *middle* – 1, *middle*, and *middle* + 1 through $\log_4 n$, where *middle* = $\log_4 n / 2$.

5.0 Performance

Performance of the Sisal code depends on building the data aray at each level in place, parallelizing the nested **for** expressions to balance the work load, and vectorizing the inner **for** expression. A naive implementation of the Sisal code would build each array defined by the inner **for** expressions separately. If insufficient space was allocated for an array, copying would occur whenever more space was required. Each concatenation operation would execute in two steps: 1) space for the composite array would be allocated, and 2) the component arrays would be copied into the space. Such an implementation would copy an enormous number of values and would be useless.

To eliminate needless copying the Optimizing Sisal Compiler [3] includes build-in-place analysis [11]. The goals of the analysis are to calculate the sizes of arrays and to determine the start addresses of the components of ar-

P	time	mflops	Sp
1	0.041	489	1.0
2	0.023	872	1.8
3	0.016	1,253	2.6
4	0.012	1,672	3.4

Table II – Performance of the Sisal code, n = 2 ** 18

P	time	mflops	Sp
1	0.083	510	1.0
2	0.044	962	1.9
3	0.030	1,411	2.8
4	0.024	1,764	3.5

Table III – Performance of the Sisal code, n = 2 ** 19

P	time	mflops	Sp
1	0.171	521	1.0
2	0.087	1,024	1.9
3	0.070	1,273	2.4
4	0.054	1,651	3.2
5	0.043	2,073	4.0
6	0.035	2,597	4.9
7	0.030	2,971	5.7
8	0.027	3,301	6.3

Table IV – Performance of the Sisal code, n = 2 ** 20

rays built by catenation, `array_addh`, or `array_adjust` operations.

Consider the first result returned by the function `long_to_short`. The size of the array is

array_size(*AAre*) + array_size(*CCre*) + array_size(*BBre*) + array_size(*DDre*)

and the size of each component array, *AAre*, *BBre*, *CCre*, and *DDre*, is

number_of_packs * *size_of_pack* / 4

The start addresses of the four component arrays are

0

array_size(*AAre*) – 1

array_size(*AAre*) + array_size(*CCre*) – 2

array_size(*AAre*) + array_size(*CCre*) + array_size(*BBre*) – 3

Each component array is a composite array. For example, `AAre` consists of m arrays of size

size_of_pack / 4

where m = *number_of_packs*. The address of the k-th component array of *AAre* is

k * *size_of_pack* / 4

The OSC compiler inserts code into the Sisal program to compute the above equations at runtime. Prior to execution of the function `long_to_short`, the program calculates the sizes of the results and calls the memory management system to allocate sufficient space. Then, the program calculates the start address of each component array within the space allocated, and passes the address to the appropriate instance of the inner `for` expression. Three types of savings accrue:

1) the memory management system is called only once,

2) since the arrays are built in place, the many catenation operations in the original code are no longer needed and may be deleted, and

3) no copying of values or intermediate arrays occurs.

OSC slices all outer `for` expressions and vectorizes all inner `for` expressions. The Sisal runtime system divides the range of the outer expressions by the number of workers, and allocates one task per worker. This strategy works well whenever the extent of the range is greater than the number of workers and the amount of work per slice is the same. In our code, the latter is always true. However, since the number of outer iterations grows from 1 to $\log_4 n - 1$ and then shrinks again to 1, the number of outer iterations is not always greater than the number workers. To insure that we used all processors throughout the computation, we forced the Sisal compiler to slice the inner `for` expressions. The compiler still automatically vectorized each slice.

Tables II, III, and IV list the execution speeds, mflops, and speedups of the Sisal code for $n = 2^{18}$, 2^{19}, and 2^{20}. Performance is uniform and close to the achievable limits of the hardware. The Sisal code runs in $12n$ words of memory. When an array is no longer needed, its space is deallocated automatically and reused—there is no memory leakage.

6.0 Conclusions

We have successfully written a machine-specific FFT algorithm in Sisal. Sisal's functional semantics have not prevented us from addressing key implemental and architectural issues. Moreover, the algorithm's performance is uniform, parallel, and close to achievable limits of the machine. A key result is that we completed the study in only six-man weeks. In that time, we developed five different algorithms, and wrote, debugged, and evaluated approximately 1500 lines of parallel code.

We did discover several shortcomings of Sisal. First, Sisal lacks subarray operations and monolithic array constructors. This deficiency made the code more complex and the programming effort more difficult. We will include subarray operations and may include a monolithic array constructor in the next version of the language. Second, the simplicity of the OSC partitioner introduced needless overhead. If we accepted the default partitioning, then we under-utilized the machine whenever the number of outer iterations was less than the number of workers. By forcing the compiler to slice both the outer and inner `for` expressions, we achieved good load balancing, but increased runtime overhead. We hope to support dynamic partitioning of nested expressions in the next version of the runtime system.

Acknowledgments

Cray time was provided by the Sisal Scientific Computing Initiative. This work was supported by Lawrence Livermore National Laboratory under DOE contract W-7405-Eng-48.

References

1. Bohm, A.P. and R.E. Hiromoto. The dataflow time and space complexity of FFTs. *Journal of Parallel and Distributed Computing*, **18**, 3 (July 1993).
2. Bollman, D., F. Sanmiguel and J. Seguel. Implementing FFT's in Sisal. *Proc. of the Second Sisal User's Conference*, Lawrence Livermore National Laboratory, San Diego, CA, October 1992.
3. Cann, D. C. *The Optimizing Sisal Compiler: Version 12.0*. Lawrence Livermore National Laboratory Manual UCRL-MA-110080, Lawrence Livermore National Laboratory, Livermore, CA, April 1992.
4. Cann, D. C. Retire FORTRAN? A Debate Rekindled. *CACM*, **35**, 8 (August 1992).
5. Cooley, J. and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computing*, **19**, (1965).
6. McGraw, J. R. et. al. *Sisal: Streams and iterations in a single-assignment language, Language Reference Manual, Version 1.2*. Lawrence Livermore National Laboratory Manual M-146 (Rev. 1), Lawrence Livermore National Laboratory, Livermore, CA, March 1985.
7. McMahon, F. H. *Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range*. Lawrence Livermore National Laboratory Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.
8. McMahon, F. H. private communications, October, 1993.
9. Novoa, J., D. Bollman and J. Seguel. A Sisal code for computing the Fourier Transform on Sn. *Proc. of the Second Sisal User's Conference*, Lawrence Livermore National Laboratory, San Diego, CA, October 1992.
10. Press, W. H., et. al. *Numerical Recipes, The Art of Scientific Computing*. Cambridge University Press, Cambridge, England, 1986.
11. Ranelletti, J. E. *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages*. Ph.D. thesis, Department of Computer Science, University of California at Davis/Livermore, 1987.

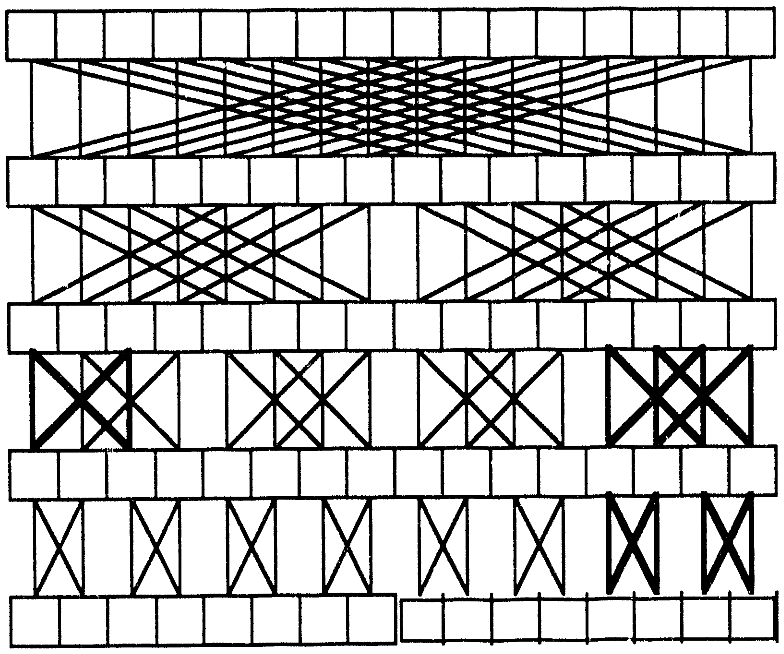


Figure 1 – A logical view of the Cooley-Tukey FFT algorithm

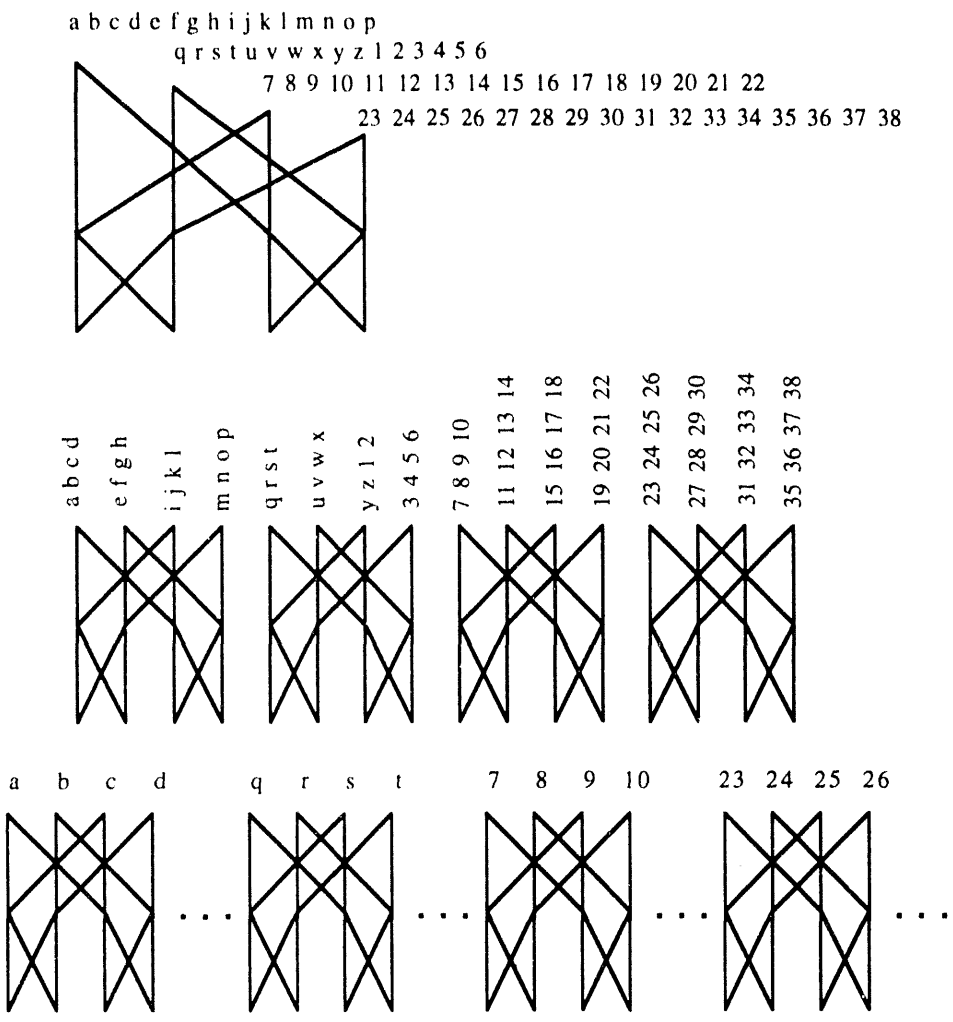


Figure 2 – Long-to-short algorithm

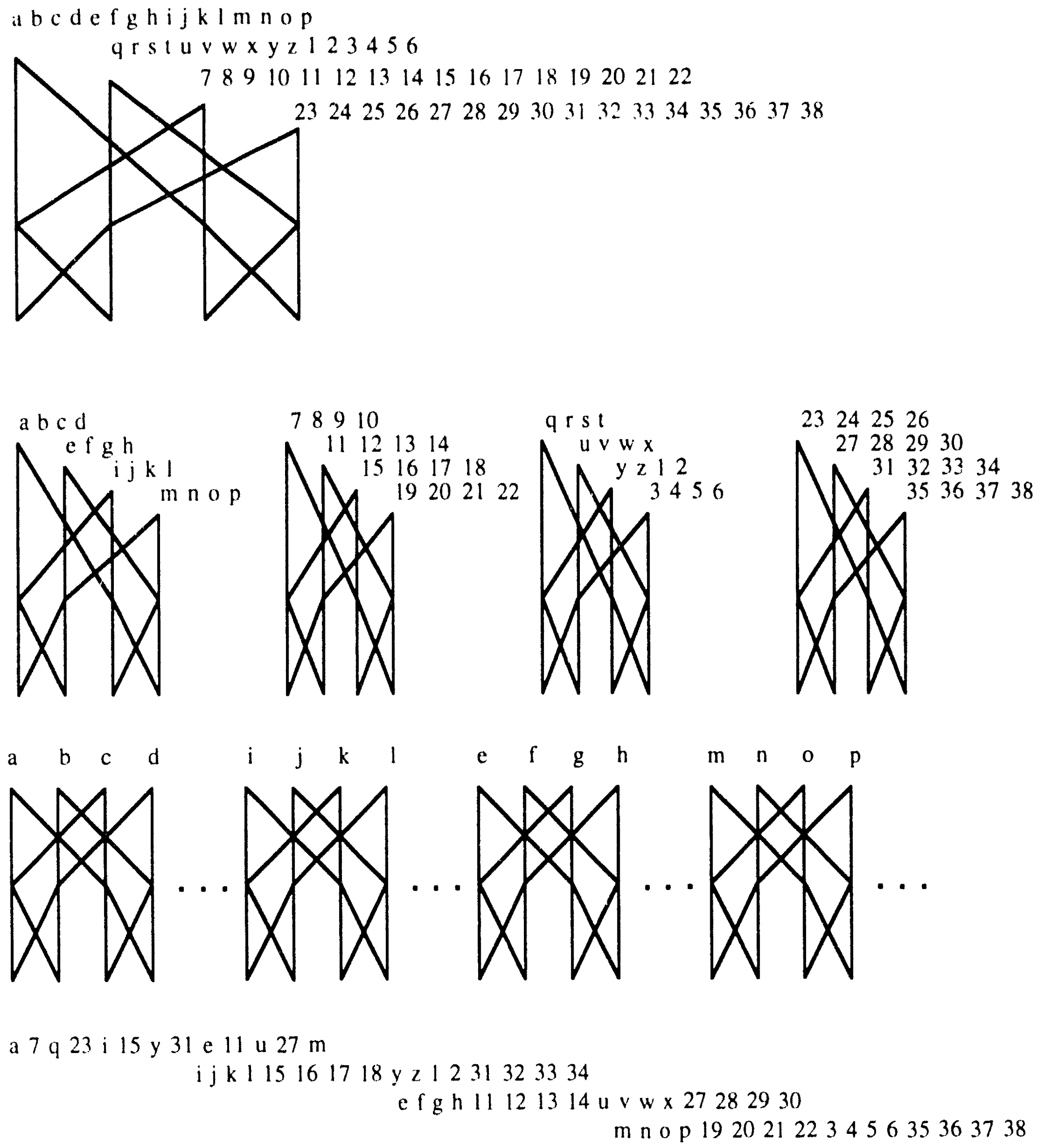


Figure 3 – Long-to-short algorithm with bit reversal

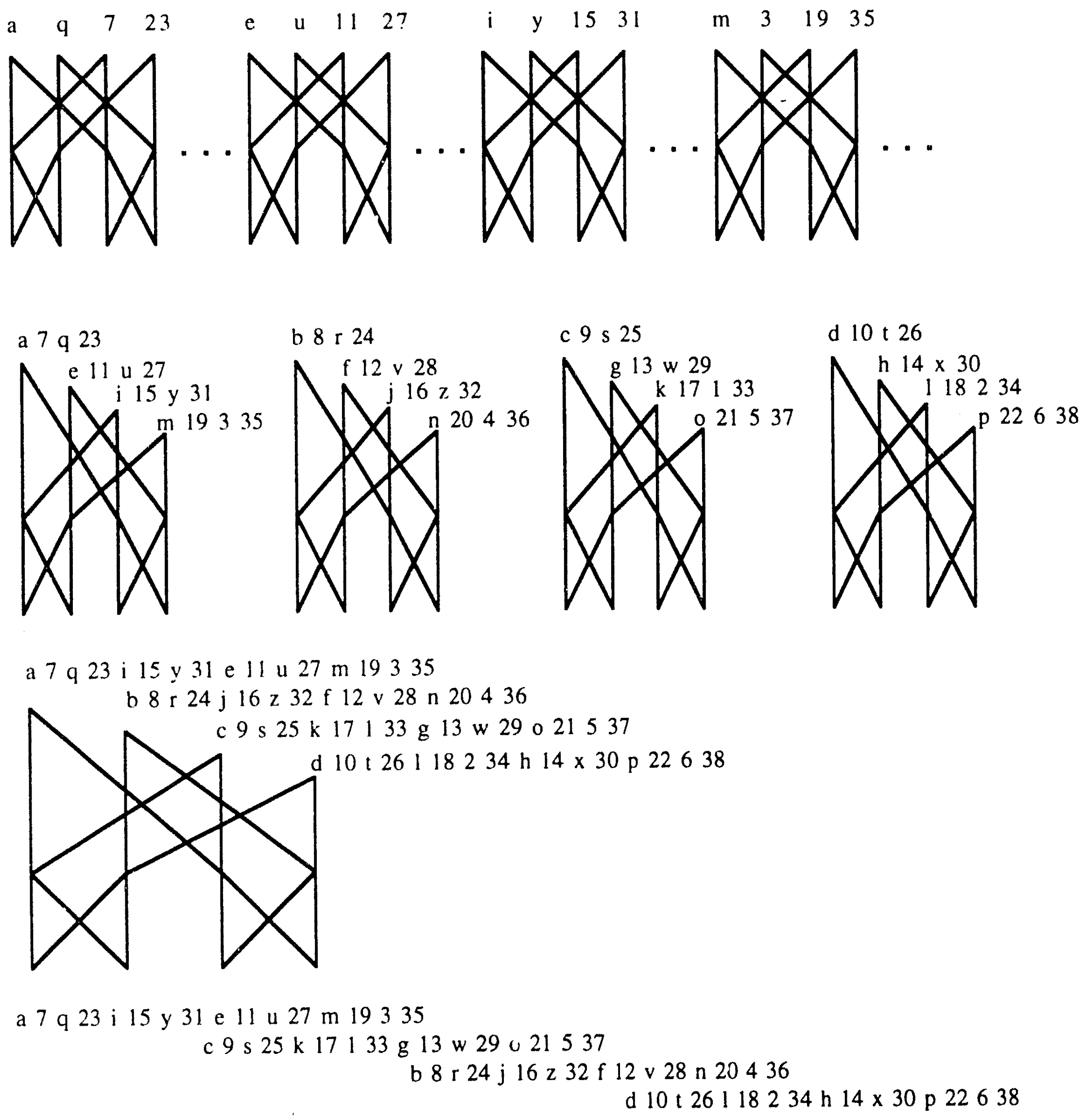

a q 7 23
e u 11 27
i y 15 31
m 3 19 35
a 7 q 23
e 11 u 27
i 15 y 31
m 19 3 35
b 8 r 24
f 12 v 28
j 16 z 32
n 20 4 36
c 9 s 25
g 13 w 29
k 17 1 33
o 21 5 37
d 10 t 26
h 14 x 30
l 18 2 34
p 22 6 38
a 7 q 23 i 15 y 31 e 11 u 27 m 19 3 35
b 8 r 24 j 16 z 32 f 12 v 28 n 20 4 36
c 9 s 25 k 17 1 33 g 13 w 29 o 21 5 37
d 10 t 26 l 18 2 34 h 14 x 30 p 22 6 38
a 7 q 23 i 15 y 31 e 11 u 27 m 19 3 35
c 9 s 25 k 17 1 33 g 13 w 29 o 21 5 37
b 8 r 24 j 16 z 32 f 12 v 28 n 20 4 36
d 10 t 26 l 18 2 34 h 14 x 30 p 22 6 38

Figure 4 – Short-to-long algorithm with bit reversal

Implementation Issues for IF2 on a Static Data Flow Architecture

Linda M. Wilkens

Department of Computer Science
University of Massachusetts Lowell
Lowell, MA 01854

Department of Mathematics and
Computer Science
Bridgewater State College
Bridgewater, MA 02325

Aaron Garth Enright

Department of Computer Science
University of Massachusetts Lowell
Lowell, MA 01854

Abstract

Data flow architectures offer a natural hardware environment for functional programming languages. Unfortunately, very few data flow systems are commercially available. One data flow environment currently available is the ULowell Data Flow Imaging Coprocessor based on the NEC μPD7281 data flow processor. This environment is well-suited for repetitive, highly data-parallel applications, such as low-level image processing; however, programming the μPD7281 is often difficult due to the lack of available high-level languages such as SISAL. Because IF2 graphs are similar to the graphs that represent μPD7281 assembly language, a straightforward translation from IF2 to μPD7281 assembly language should exist. In this paper, we discuss the problems and possibilities of implementing IF2 for this environment.

1 Introduction

There are two main problems associated with parallel programming: *latency* and *synchronization* [AI91]. Latency is the time delay between when a processor makes a request for a resource and when that request is satisfied. Most processors spend this time idling. Synchronization is the time spent structuring concurrent activities to avoid both safety problems and nondeterminism.

Data flow computing addresses these problems by using the availability of data, rather than explicit control flow, to drive computation. This paradigm both alleviates problems caused by memory latency [Den91] and provides run-time scheduling of instruction execution. Rather than waiting for data to arrive, a data flow machine schedules a computation when its operands are available. Thus, a minimal amount of time is spent idling, waiting for data.

The two main classes of data flow architectures are *static* and *dynamic*. Static data flow machines lack the sophisticated token-matching mechanisms that characterize dynamic data flow architectures [GKW85], and are suitable for a class of algorithms that exhibit a great deal of regularity. The ULowell Data Flow Imaging Coprocessor [CNMW90] is based on the Nippon Equipment Corporation μPD7281 data flow processor [Nip85, Jef85], which is a static architecture. The theme of our research is to effectively utilize this static data flow architecture by relying on the compiler to ensure correct token matching occurs even though the hardware does not provide this functionality. Interesting compilation techniques for data flow architectures can be seen in [Dav79, Gao86].

Applicative programming languages such as SISAL [BOCF] may be appropriate high level languages for data flow architectures because of interesting parallelisms between the functional and the data flow paradigms. The task of implementing SISAL on a data flow machine is simplified by the similarity between both IF1 [SG85] and IF2 [WSYR86] graphs and data flow graphs.

This paper is organized as follows: Section 2 presents background information on data flow architectures. Section 3 describes our target architecture in detail. Section 4 explores the issues involved in porting IF2 to the architecture of Section 3. Section 5 contains concluding remarks and directions for future research.

2 Data Flow Architectures

Parallel architectures based on traditional system designs often do not perform as well as expected because von Neumann architectures are inherently sequential. They function by fetching an instruction, fetching its operands, executing the instruction and then fetching the next instruction. Highly parallel architectures based on this paradigm suffer from two main problems:

1. Fetching operands from memory is time consuming, and processors often remain idle during this period.

2. It is difficult to determine which instructions may be executed in parallel, because data dependencies are not explicit.

Data flow architectures address these two problems by dismissing the fetch and execute paradigm. Instead of loading an instruction and then fetching its operands, data flow systems wait for all of an instruction's operands to become available, and then execute the instruction. Thus data, not the operations performed on data, drive the machine.

Since data is the driving force in a data flow machine, operands for instructions must contain more information than just a data value. Packages of data, called *tokens*, must minimally contain both a reference to the instruction that uses them and a destination address for the result. When a token arrives at a data flow processing unit, its instruction reference is checked to see how many operands are required. If all of the operand tokens for the instruction are available, the instruction is scheduled for execution; otherwise the input token is held until the other operands arrive. The result token produced by executing the instruction is sent to the destination address contained in the input token.

The two problems of traditional computer architectures are addressed by the data flow paradigm, because:

1. Processing elements operate on available data while waiting for other data to arrive.

2. Since all tokens must contain a destination address, data dependencies can be detected be searching for all nodes with the same destination address. By representing this flow of data as a directed graph and then reversing the direction of the arcs, a data-dependency graph is created.

2.1 Data Flow Graphs

A data flow program may be represented by a directed graph in which nodes represent instructions and edges represent the flow of tokens, as can be seen in Figure 1. In this figure, data tokens conceptually travel on the arcs joining the nodes. For example, token "a" travels down the left-side incoming arc of the "+" node; token "b" travels down the right-side incoming arc of the "+" node. Both tokens carry the information that the resulting sum token is destined for the "*" node.

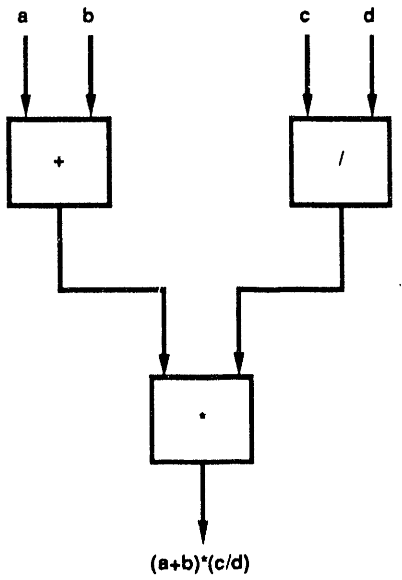


Figure 1: A typical data flow graph

One of the principle problems which face data flow designs is what to do if two tokens arrive at the left-side input arc before any tokens arrive at the right side input arc. One possibility would be to queue tokens, or simply store them, but then the question arises "if two tokens are waiting on the left-side arc and a token arrives on the right-side arc, which token from the left side does it match with?" This question has led to the development of the two classes of data flow architectures: static and dynamic. Strictly static architectures provide no hardware assist for matching up tokens that arrive out of order. Dynamic architectures use a tag field in the tokens so that an instruction executes only on tokens with matching tags. These two types of architectures are discussed in detail in the following sections.

2.2 Static Data Flow

Static data flow has no special purpose hardware for matching tokens. In these systems, it is the responsibility of the software to ensure that instructions operate on matching tokens. These systems have the advantage of being cheaper and faster, but are inherently more difficult to program and debug.

Tokens in a static data flow environment must contain (minimally) the following fields:

1. A data value field.
2. The operation to perform on the data value field.
3. The destination of the operation's result.

Tokens are matched with the first available token at an arc. This can have different meanings depending on the support hardware on a particular system. For example, some data flow systems allow for queuing along an arc; this entails using a finite length queue to store arriving tokens. As long as the queue does not overflow, the first token arriving on the other arc will match with the first token in the queue. However, because of the finiteness of these queues, values can be lost or overwritten, resulting in mis-matches among the remaining tokens. μPD7281 queuing is illustrated in Figure 2.

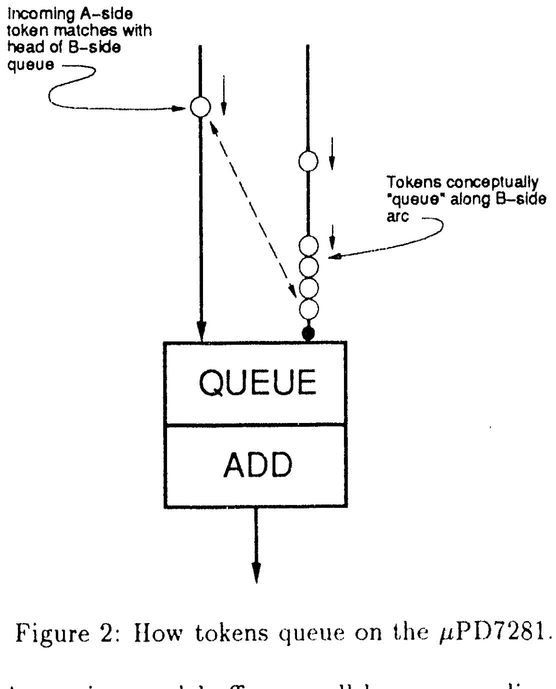

Incoming A-side token matches with head of B-side queue
Tokens conceptually "queue" along B-side arc
QUEUE
ADD

Figure 2: How tokens queue on the μPD7281.

A queuing model offers a well-known paradigm for the analysis of data flow program execution. Even if a machine offers no hardware support for queues , the architecture can be characterized as having queues of length one (imagine a queue where one value can be enqueued, but if it is not read before the next value is enqueued, the old value is overwritten). Thus, one can generalize static data flow as a "first-in-first-out" matching paradigm over finite queues, allowing analysis by means of queuing models. The problem with this approach is that finite queuing models are more difficult to work with than are the more familiar infinite models.

The strictest firing rule for static data flow is that an instruction can be executed when all of its input arcs are occupied, and none of its output arcs are occupied. This rule avoids the out-of-order matching problem, but introduces other difficulties. The requirement that the output arcs be unoccupied reduces the amount of parallelism that can be exploited and also requires some means of enforcement. In early versions of static data flow, this rule was enforced by means of acknowledgement tokens sent from a destination to a source indicating that the corresponding data arc was available. This solution had the unfortunate consequence of doubling the token traffic. More modern static architectures avoid acknowledgement arcs but are still more restrictive than the dynamic model. Unless the one-token-per-arc rule is enforced in hardware, the compiler must very carefully structure loops and other control constructs to ensure correct program execution.

2.3 Dynamic Data Flow

Dynamic data flow alleviates the problem of matching tokens by implementing explicit token-matching hardware. This makes dynamic data flow hardware more complex and more expensive than static data flow hardware, but removes a considerable burden from the programmer.

Tokens in a dynamic data flow environment must contain (minimally) the following fields:

1. A data value field.
2. The operation to perform on the data value field.
3. The destination of the operation's result
4. A field containing information for matching other input tokens to the instruction.

The destination and matching information are combined into a tag field. Tokens are matched only with tokens that contain the same tag. The tag field may combine information not only pertaining to the token's destination operation, but also pertaining to the specific loop iteration, the given function invocation, or other context identification. Token matching can be quite time consuming, so several strategies have been developed for minimizing the cost, such as Explicit Token Store (ETS) [ABU91] and associative caches [AE91]. Unlike the FIFO model that we observed for static data flow, dynamic data flow matching follows no simple paradigm.

The execution rule for dynamic data flow is that an instruction is executed only when tokens with matching tag fields are available at all of its input arcs. The execution rule is typically enforced by hardware.

2.4 Advantages of Static Data Flow

The reason we chose static data flow for our research is that static data flow components are cheaper and faster than their dynamic counterparts, and the commercial availability of one such processor enabled the production of an inexpensive testing platform. The FIFO model of matching also makes execution simulation and analysis easier. The problem we wish to address is token mis-matching; we believe that this problem can be handled in most cases at compile time.

This is essentially the same approach as RISC on traditional computer architectures: off-load as much as possible to the software, keeping the hardware as simple as possible. If difficult operations such as token matching and synchronization could be done in software, especially at compile time, then no matching hardware would be needed.

It is a non-trivial problem to guarantee token synchronization through software, and we only begin to consider the problem in this paper. It is our continuing research to find out if token synchronization can be done exclusively in software and what tradeoffs are involved.

2.5 Similarities between Data Flow Hardware and Functional Programming Languages

Data flow systems are neither fully applicative nor von Neumann in style [Bac78], yet the model of computation aspires to many of the same goals as purely functional languages. The concept of clean semantics, free from side-effects, expressed in a strongly mathematical style are the common denominators that link these two forms. Data flow is often criticized for not fitting into the applicative model; however, we submit that this criticism applies to particular implementations, not to the design philosophy of data flow.

Many of the digressions away from the purely functional implementation of data flow have been accommodations made to the von Neumann community. Many of these implementation constructs are similar to the various ways in which LISP has been altered to placate programmers versed in the imperative style. Yet, inherently, data flow hardware and functional languages strive toward the same goals: understandability and clean-semantics.

2.6 Graphical Models

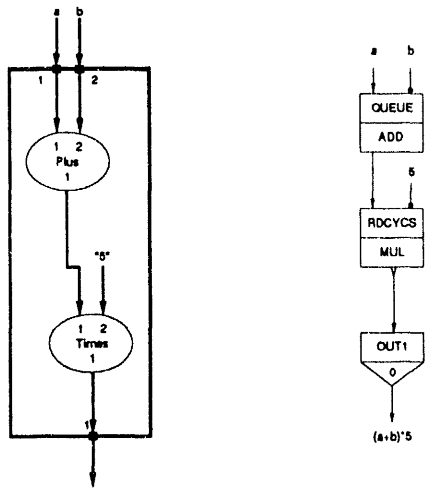


Figure 3: Data Flow and μPD7281 graphs of the same function

Both data flow programs and the intermediate forms IF1 and IF2 are conveniently represented as directed graphs. Furthermore, these two sorts of program graphs have similar formats, as can be seen in Figure 3. These graphs both represent the expression $a + b * 5$, and even though the μPD7281 style data flow graph on the right has an additional node, they are similar in many respects. The similarity in graphical formats indicates that a translation from one form to the other should exist.

3 The ULowell Data Flow Imaging Coprocessor

3.1 Overview

The ULowell Data Flow Imaging Coprocessor was developed at the University of Lowell (now the University of Massachusetts Lowell) as a high-performance

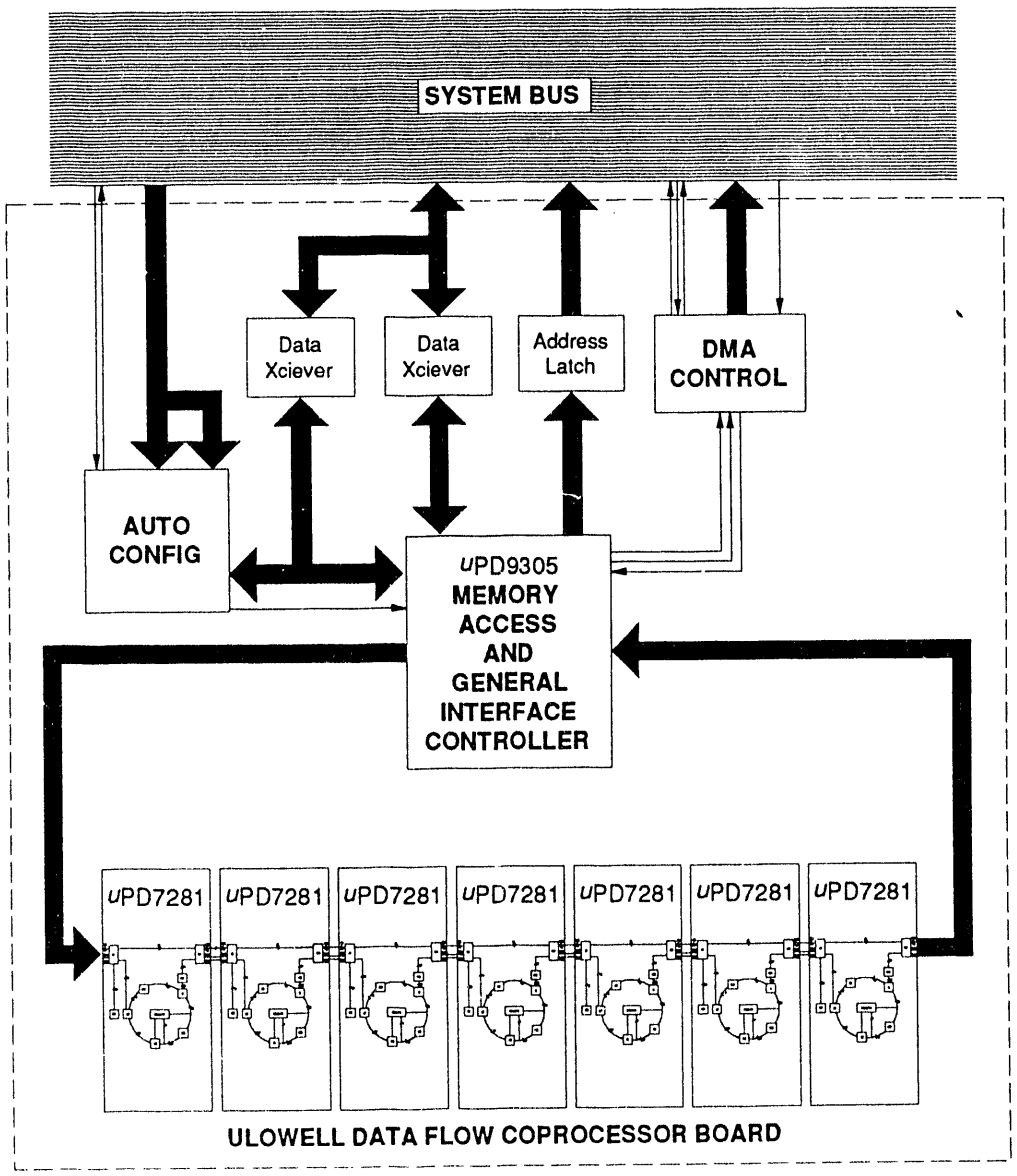


Figure 4: The Ulowell Data Flow Co-Processor Board

imaging cor· :essor hosted by an Amiga microcomputer. The Amiga computer historically has offered support for both graphics and imaging. In addition, the Amiga architecture facilitates the design and implementation of coprocessor boards. The ULowell Data Flow board consists of seven NEC μPD7281 data flow processors connected in a one-direction ring. Incorporated in the ring is a NEC μPD9305 Memory Access and General Interface Controller (MAGIC) chip, which is the interface to the Amiga. The coprocessor board is able to directly access the Amiga system memory, and can use it to store large images which will be manipulated by code running on the board. The board is illustrated in Figure 4 (derived from [Sim88]).

3.2 The NEC μPD7281

The Nippon Equipment Corporation (NEC) μPD7281 is a commercially available static data flow processor, designed to support imaging operations. It consists of seven functional units such as the link table, the function table, and the data memory; plus input, output and refresh controllers. The functional units are arranging in a circular pipeline as depicted in Figure 5. Because the chip provides hardware support for multiple tokens per arc, the chip is not strictly static. Furthermore, because the function table holds state information between successive function node executions, the architecture is not strictly functional. The functional units are discussed below.

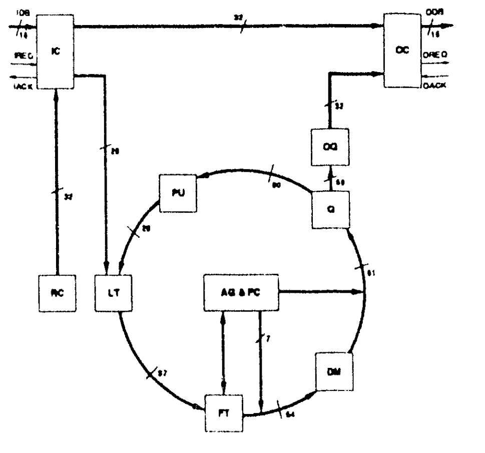


Figure 5: The μPD7281 Internal Pipeline Architecture

IC – Input Controller. Routes incoming tokens either to this processor, or forwards them to the next processor in the ring.

RC – Refresh Controller. Sends refresh tokens through the processor for memory refreshes.

OC – Output Controller. Routes tokens to other processors.

LT – Link Table. Stores the link (edge) information of the data flow graph.

FT – Function Table. Stores the function (node) information of the data flow graph.

AG&FC – Address Generator and Flow Controller. Manages queues, data memory references, and flow control constructs.

DM – Data Memory. Contains 512 words of general purpose memory, which is used to store queues, to hold constants, and acts as a small local memory.

Q – Queue. Used as a buffer between the data memory and the processor unit.

PU – Processing Unit. Performs arithmetic and logic instructions. Can also duplicate tokens.

OQ – Output Queue. Queues tokens waiting to be output to other processors.

3.3 NEC μPD7281 Data Flow Graphs

Processing Unit instructions can be associated with either queues or memory references, both of which are stored in Data Memory. To sufficiently represent this association, NEC has devised an alternative to traditional data flow graphs. μPD7281 data flow graphs use a compound node which contains the Processing Unit (PU) instruction, and may also contain an associated queue or memory reference (AG&FC) instruction. Examples of these compound node types as well as other types of nodes for output and conditional executions can be seen in Figure 6.

An NEC μPD7281 data flow graph node's left-side input is referred to as the "A" side input and the right-side input is referred to as "B" side input. All instructions are either unary or binary. Output arcs are referred to as "X" and "Y" in the general case, but this unique instruction set allows for duplicating the "X" output. The type of a PU instruction determines the number of inputs and the number of outputs as well as the type of AG&FC instructions that it can be associated with. AG&FC instructions can also appear as nodes by themselves. Another type of instruction, generator (GE) instructions can also be used for the

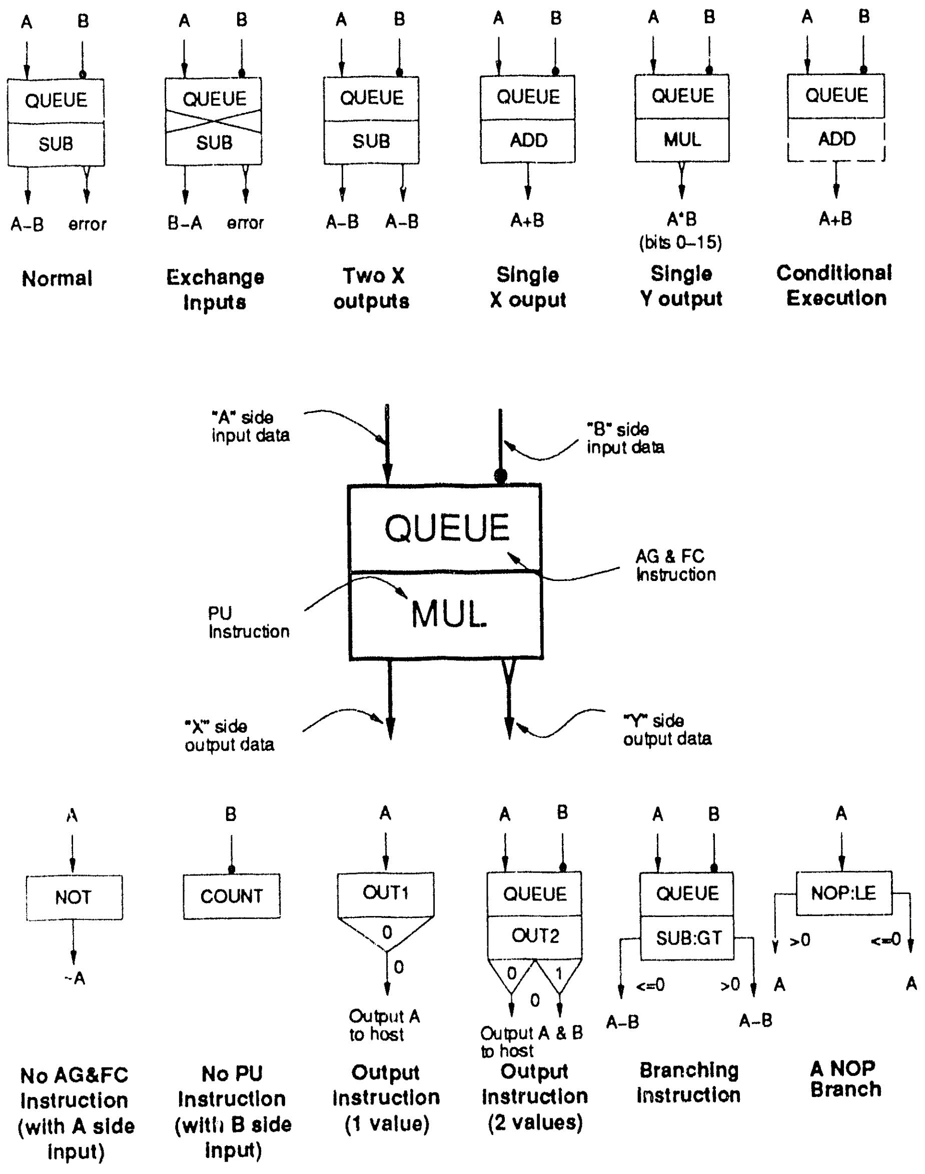


Figure 6: Examples of μPD7281 Data flow Graph Elements

duplication and generation of tokens. GE instructions can have more than two outputs.

μPD7281 data flow graphs lack an explicit notion of hierarchical composition. When programs are large, graphs can become quite large and cumbersome to read. We suggest an alternative approach to standard μPD7281 graphical design by creating *overview* graphs. Overview graphs are standard data flow graphs such as Figure 1. In these graphs, more complex functions are represented as a single node (a similar concept to graph nodes [SG85] in IF2). When the final graph is drawn, graphs representing the complex functions are spliced in where only a node with the functions' names appeared in the overview graph. This process is illustrated in Figure 7.

3.4 Programming the NEC μPD7281

The programming paradigm for the NEC μPD7281 is quite straightforward, and is illustrated in Figure 8.

1. Devise the algorithm.
2. Draw the NEC μPD7281 data flow graph.
3. Label all nodes and arcs on the graph with unique names.
4. Write the μPD7281 assembly language program by converting the labelled nodes and arcs into FUNCTION and LINK statements.

The last step is actually fairly easy if the graph was drawn and labelled correctly. The PU, AG&FC and GE instructions should all appear on the graph with all necessary information to write the code. This information would include:

1. The number and side(s) of the inputs (indicated by the incoming arcs).
2. The number and side(s) of the outputs (indicated by the outgoing arcs).
3. The PU instruction.
4. Associated functions (for example, when a QUEUE (AG&FC) instruction is associated with an ADD (PU) instruction).
5. Special processing such as conditional execution, input operand swap, test conditions, or flow control.
6. Destination processor addresses for output functions.

Once this information is obtained, it can be placed in the assembly language program in a "fill-in-the-blank" manner.

4 Implementing IF2 on the NEC μPD7281

4.1 Common Functions

There are very few one-to-one correspondences between the instructions of IF2 and the NEC μPD7281. First of all, the μPD7281 is an integer processor, so all floating point operations must be performed in software. Secondly, the μPD7281 carries condition and sign information on two extra bits, implying that boolean type data must be extracted from the condition bit and converted to integer form. Finally, due to the limited amount of on-processor memory, there is no inherent support for records, arrays, or streams. (This problem is discussed in detail in Sections 4.2 and 4.3).

The few instructions that are common between IF2 and the μPD7281 are shown in Table 1.

IF2 Type	IF2 Instruction	μPD7281 Instruction
Integer	Plus	ADD
Boolean	Plus	OR
Integer	Times	MUL
Boolean	Times	AND
Integer	Minus	SUB
Boolean	Not	NOT
	NoOp	NOP

Table 1: Equivalent instructions for IF2 and the μPD7281

Despite the fact that many IF2 instructions do not correspond in a one-to-one fashion to μPD7281 instructions, several can be developed easily. In Figure 8, we showed how to develop an absolute value function on the μPD7281. The code in this example can be easily spliced into a μPD7281 data flow graph anywhere an **Abs** node appears in an equivalent IF2 graph. The same is true for **Min**, **Max**, **Div**, **Mod**, **Less**, **LessEqual**, **Equal**, **NotEqual**, and other similar nodes.

4.2 Memory Management

Because there are only 512 words of data memory available on-chip, memory management on the

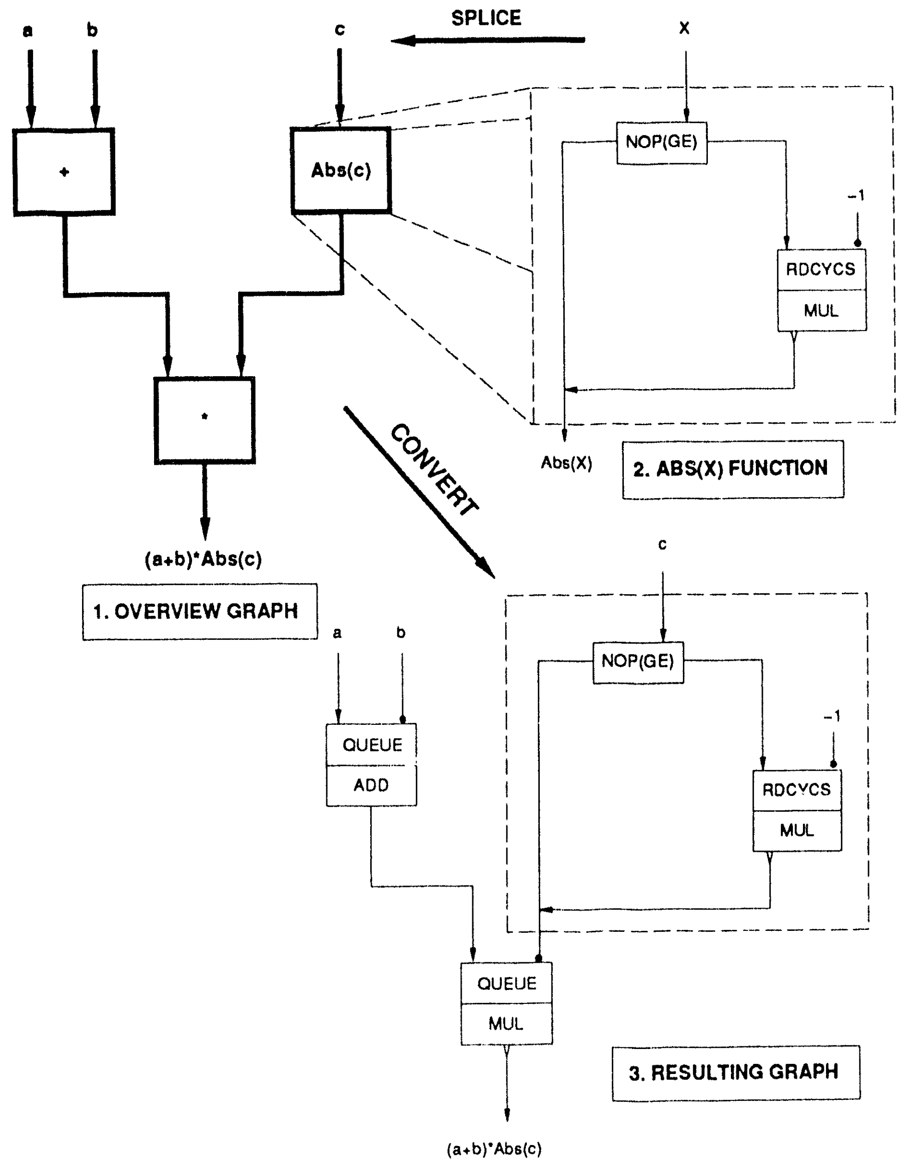
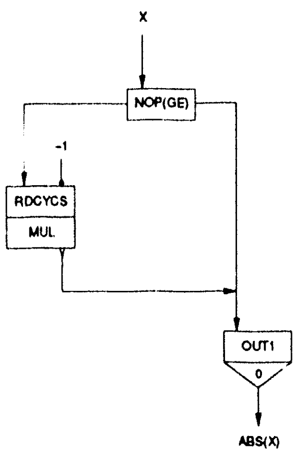


Figure 7: Splicing an Abs(x) function into an overview graph.

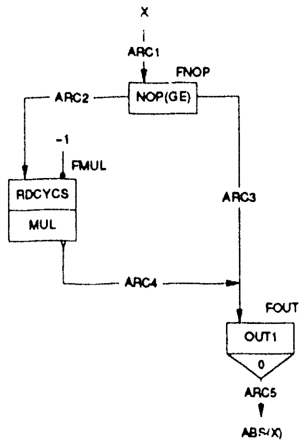
```
ABS(X) := IF X >= 0 THEN
              X
            ELSE
             -X
            END IF
```

1. Conceptualize



2. Draw Graph

3. Add Labels



4. Write Code

```
INPUT     ARC1 AT 1;
OUTPUT    ARC5;

LINK      ARC2,ARC3  = FNOP(ARC1);
LINK      ARC4       = FMUL(ARC2);
LINK      ARC5       = FOUT(ARC3);
LINK      ARC5       = FOUT(ARC4);

FUNCTION    FNOP   = NOP(GE);
FUNCTION    FMUL   = MUL(Y),RDCYCS(NO,1);
FUNCTION    FOUT   = OUT1(0,0);

MEMORY      NO     = -1;

END;
```

Figure 8: Writing an Absolute value function for the μPD7281

μPD7281 becomes difficult, and good memory management techniques are crucial. Reference counts, and the ability to swap words in-and-out of data memory all need to be done in the software loaded onto the μPD7281. Fortunately, the features of IF2 such as operating on data by reference, memory preallocation and AT nodes [WSYR86] all can relieve the memory management burden from the processor.

There are two possible strategies for dealing with the limited amount of processor memory:

1. Use the processor memory as a cache. This strategy involves including cache management routines in all μPD7281 programs.

2. Store only temporaries and constants in processor memory, leaving all variables in host system memory and accessing them through the μPD9305. This approach has the advantage of simplicity, with the greater cost of more memory fetches; however, this approach might be feasible in a data flow architecture, because the processor can (theoretically) be kept busy with other operations while waiting for the memory fetches to complete.

An optimal strategy might be a hybrid of these two approaches. We propose to borrow another idea from RISC computing and use part of the data memory for constants and queues and the rest as a *register file* [Hwa93] rather than a cache. Using a register file rather than a cache allows data memory allocation to be done at compile time using register allocation techniques such as *interference graph coloring* [HP90]. We refer to the application of RISC techniques such as this to static data flow architectures as *Data flow-RISC*.

4.3 Arrays and Streams

Arrays and streams must be accessed as individual data words from image memory, due to the architecture of the μPD9305. In a control flow system, the penalty for this von Neumann bottleneck [Bac78] would be intolerable, but because this is a data flow system, the processor should not see significant idle time. Techniques such as *load balancing* [Gao86] are needed to keep the processor pipeline busy.

4.4 Artificial Dependency Edges (ADEs)

One of the major contributions of IF2 is the introduction of Artificial Dependency Edges (ADEs) [WSYR86]. These edges in the IF2 graph model are used for synchronizing nodes which would normally execute independent of each other. Surprisingly, these are also one of the easiest features to implement on the μPD7281.

To construct an ADE on the μPD7281:

1. Place a new combination **QUEUE/NOP** node on the graph.

2. Remove the arc to the "A" side input of the ADE destination node and connects it instead to the "A" side input of the **QUEUE/NOP** node.

3. Draw an arc from the "X" output of the **QUEUE/NOP** node to the "A" side input of the ADE destination node.

4. Construct an arc from the source ADE node output to the "B" side input **QUEUE/NOP** node. Since the value of the token is ignored, it is unimportant whether it is an "X" or "Y" side output. If, however, there is no free output arc on the source ADE node, one of the output tokens must be replicated and sent to the **QUEUE/NOP** node. The μPD7281 provides the instructions **COPYM** or **COPYBK** for token replication, and one of these node types can be spliced in between the source ADE node and the **QUEUE/NOP** node.

This entire procedure is demonstrated in Figure 9.

The definition of the **QUEUE/NOP** node specifies that it will not output its token until the source node has output its token. Note that special care must be taken in order to prevent too many tokens from being queued at the **QUEUE/NOP** node. This involves using a longer queue, switching the "A" and "B" side inputs to the **QUEUE/NOP** node, and making the node exchange the "A" and "B" side inputs.

4.5 Other Functions on the μPD7281

The μPD7281 has many more instructions which might be useful for an efficient implementation of IF2. It also contains instructions for self-modifying code, which would not be used for implementing IF2, and object loading, which may be useful for distributing programs across multiple processors. A complete reference to the μPD7281 instruction set is given in [Nip85].

4.6 Load Balancing and Mapping

The ULowell Data Flow Imaging Co-processor contains seven μPD7281 data flow processors as well as

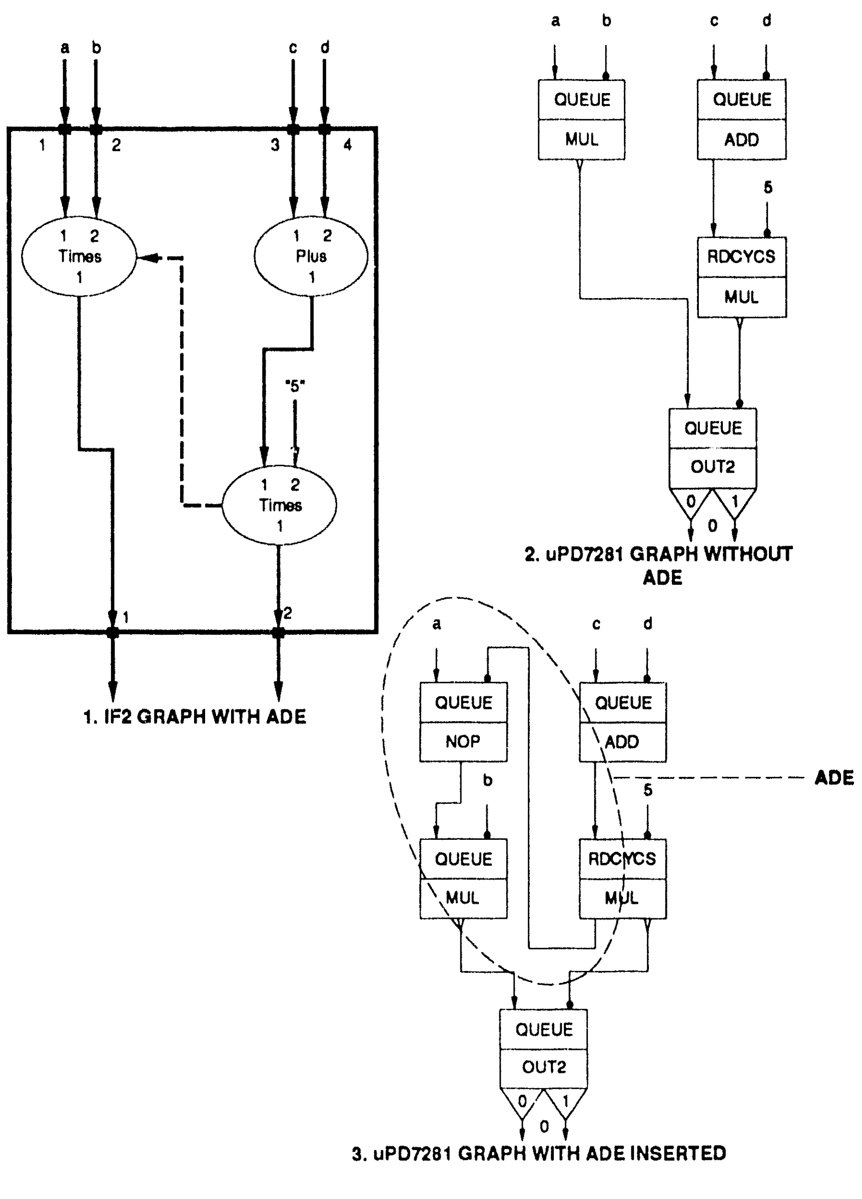


Figure 9: Implementing ADE's on the μPD7281

one μPD9305 Memory Access and General Interface Controller. To obtain optimal throughput on the co-processor, it is necessary to properly map programs onto processors. This problem is discussed thoroughly in [Wil92].

4.7 An Initial Implementation Suite

Initially, we propose the following implementation suite for IF2 on the ULowell Data Flow Imaging Co-Processor:

1. Integer arithmetic and boolean functions.
2. Single dimensional arrays.
3. Non-recursive function calls.
4. Streams.
5. One μPD7281 processor.

The primary goal of the initial implementation suite is to work out as much of the token synchronization as possible. Features such as floating-point, recursion, etc. can be added incrementally. Using multiple μPD7281 processors is an implementation of [Wil92].

5 Summary

We have discussed several ideas for an implementation of IF2 on a static data flow architecture and shown the potential advantages of such an implementation. We have, of course, left the actual implementation to future work, showing several concrete examples which can be used as a foundation. Efficient compilation techniques would allow static data flow to out-perform later dynamic data flow machines, and eliminate (or at least minimize) the need for expensive and slow matching hardware. All-in-all, the marriage of IF2 to data flow is a natural one, and should yield key results in the future.

References

[ABU91] Arvind, L. Bic, and T. Ungerer. Evolution of data-flow computers. In *Advanced Topics in Data-flow Computing*, pages 3 - 34, 1991.

[AE91] D. Abramson and G. Egan. Design of a high performance dataflow multiprocessor. In *Advanced Topics in Data-flow Computing*, pages 121 - 142, 1991.

[AI91] Arvind and R. Iannucci. Two fundamental issues in multiprocessing. In *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesburg*, June 1991.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21:613 - 641, August 1978.

[BOCF] A. Böhm, R. Oldehoeft, D. Cann, and J. Feo. *SISAL Reference Manual, Language Version 2.0.* Lawrence Livermore National Laboratory, P.O. Box 808, L-306, Livermore, CA 94550.

[CNMW90] J. Canning, I. Nwokogba, R. Miner, and L. Wilkens. A software development environment for data flow computation. In *Proceedings of the ISMM International Conference on Parallel and Distributed Computing and Systems*, pages 86 - 90, October 1990.

[Dav79] A. Davis. A data flow evaluation system based on the concept of recursive locality. In *Proceedings of the National Computer Conference*, pages 1079 - 1086, 1979.

[Den91] J. Dennis. A modern static data-flow architecture. In *Advanced Topics in Dataflow Computing*, pages 121 - 142, 1991.

[Gao86] G. Gao. *A Code Mapping Scheme for Dataflow Software Pipelining.* PhD thesis, M.I.T., Cambridge, MA, 1986.

[GKW85] J. Gurd, C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28:34 - 52, January 1985.

[HP90] J. Hennesy and D. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[Hwa93] K. Hwang. *Advanced Computer Architecture.* McGraw-Hill, New York, 1993.

[Jef85] T. Jeffery. The μpd7281 processor. *Byte Magazine*, pages 237 - 246, November 1985.

[Nip85] Nippon Equimpent Corporation. *The μPD7281 Reference Manual*, 1985.

[SG85] S. Skedzielewski and J. Glauert. *IF1: An Intermediate Form for Applicative Languages.* Lawrence Livermore National Laboratory, P.O. Box 808, L-306, Livermore, CA 94550, July 1985.

[Sim88] D. Simoes. On multiprocessor dataflow parallel pipelined processors in image processing. Master's thesis, The University of Lowell, 1 University Avenue, Lowell, MA 01854, February 1988.

[Wil92] L. M. Wilkens. *Modeling Parallel Computation via the Fusion of Timed Petri Nets with an Application to the Mapping Problem.* PhD thesis, The University of Massachusetts Lowell, 1 University Avenue, Lowell MA 01854, April 1992.

[WSYR86] M. Welcome, S. Skedzielewski, R. Yates, and J. Ranelletti. *IF2: An Applicative Language Intermediate Form with Explicit Memory Management.* Lawrence Livermore National Laboratory, P.O. Box 808, L-306, Livermore, CA 94550, December 1986.

Systematic Control of Parallelism in Array-Based Dataflow Computation

Kanad Roy and Carl McCrosky
Department of Computational Science
University of Saskatchewan, Saskatoon
CANADA S7N 0W0

Abstract

Dataflow systems must be concerned with the degree of parallelism generated by array computations. It would be preferable if the program largely avoided specification of parallelism, and the dataflow compiler later generated parallelism appropriate to the intended dataflow machine.

The present research addresses this objective. Our approach is based on maintaining a high degree of referential transparency in both the source program and the dataflow graph. Choices of two representations for arrays (one sequential and the other parallel), and a set of identities in the source language permit great flexibility in tailoring the degree of parallelism and the size of the dataflow graph presented to the machine.

Several patterns of systematically factoring computations into sequential and parallel sub-components are used to gain a high degree of control over the execution characteristics of array computations. The paper reports an implementation of this approach and includes a number of examples.

1. Introduction

Designers of dataflow systems must concern themselves with the degree of parallelism generated by dataflow computations [1][2]. The degree of parallelism is not simply maximized, as machines can be saturated with waiting tokens. Rather, the goal is to generate parallelism appropriate to the capabilities of the underlying dataflow machine. This paper studies the systematic control of the degree of parallelism for array computations in static dataflow systems.

The size of the dataflow graph is a related concern. Ideally the graph should be as small as possible to conserve memory and to minimize program loading time. However, the amount of available parallelism is loosely correlated with the size of the dataflow graph, particularly in static dataflow systems. Within the constraint of the parallelism available in the problem being solved, there is a trade-off between parallelism and graph size, which we explore.

Arrays are a primary source of parallelism in dataflow systems. Many of the scientific problems for which dataflow computation is intended have arrays as their principle data structure. Often operations on arrays can be performed in parallel. Unfortunately, the resultant run-time parallelism may be excessive. For instance, scaling all of the pixels in a large image may generate much too much parallelism. The alternative of sequential processing is equally unfortunate.

Some throttle on parallelism is required that supplies approximately the appropriate amount of work (executable dataflow actors) to the underlying machine. Often this throttle is clever programming by the user. Unfortunately this approach generally fixes the problem's parallel structure. The only executable form of the algorithm – the program – is thus committed to one particular view of parallelism, which is likely appropriate for only one particular machine. The present research takes another approach to the control of parallelism: we explore the use of compile-time techniques to compile declarative array languages to executable dataflow graphs which present appropriate degrees of parallelism at run time.

There are three key points of departure for this work. All three are chosen to maximize our ability to reason about and manipulate both programs and dataflow graphs. 1) We adopt a purely functional source language, as the relatively simple semantics of these languages supports our ability to reason about programs. 2) Our source language incorporates a well-founded mathematical model of arrays and an integrated set of array operations, thereby maximizing ease of reasoning about arrays. This formal basis for arrays provides a rich source of program equivalence laws which allow us to quickly find equivalent forms for array expressions. 3) We adopt unconventional representations of arrays, which permit a nearly transparent correspondence between our functional array programs and our dataflow graphs.

Our system – based on these three ideas – permits us two degrees of freedom in controlling parallelism and the related dataflow graph size. 1) We can selectively choose between two array representations to tailor parallelism and graph size. 2) We can use our algebraic equivalence laws to manipulate both the source program and the resultant dataflow graph. Both of these degrees of freedom can be controlled by automated systems to obtain appropriate degrees of parallelism and graph size.

Some of the ideas used in this research have been explored or discussed elsewhere. The principal originality of our approach is the systematic integration and exploitation of these ideas. The present work is a preliminary exploration of this approach; it is limited in several ways which are discussed in Section 10.

Our solution to the degree of parallelism problem does not apply to all reasonable array computations. There remain important problems for which more standard array representations and more procedural implementations are best. The techniques developed in this paper are applicable to a "pure" subset of array dataflow computations in much the same way that some of the more powerful compiler optimization techniques apply only to the pure expression subsets of procedural languages. Nevertheless, the techniques developed here can be usefully embedded in the larger context of practical dataflow systems.

This paper draws on a variety of sources of material; the first three sections establish necessary background. Section 2 defines our first-class nested array language, Λ. Section 3 discusses how separately reported research allows us to avoid the initial pitfall of functional array languages – the excessive generation of intermediate containers. Section 4 specifies our array representation schemes. Section 5 specifies our abstract dataflow machine. Section 6 defines a set of functions on arrays, which are used in later examples. Section 7 describes the compilation process from our functional language to static dataflow graphs.

A simple strategy which provides some control over the degree of parallelism is discussed in Section 8. Section 9 introduces some equivalence laws of our theory of arrays and describes their usefulness in improving our degree of control over parallelism. Section 10 concludes the paper by discussing the achievements of this line of research, the limitations that have been encountered, and future directions for research.

2. Λ, the language and the domain of arrays

The array-based functional language used in this paper, Λ, is a subset of the experimental language, Falafel, being developed at the University of Saskatchewan [7][8]. These languages are sugarings of the typed, higher-order lambda calculus. They differ from most other functional languages in that arrays are included as first-class objects in the value domain, in the type system, and in the selection of primitive operations. The arrays in these languages may be nested, thereby permitting more flexible manipulation of array computations. Falafel and Λ draw ideas from numerous other programming languages: Nial [6], Miranda [9], Id [11] and Sisal [3]. The greatest strength of Λ is that its simple semantics makes array-based computation relatively easy to reason about. (Λ and even Falafel are intended to serve only as experimental testbeds for declarative array-based computation. Useful results will be applied to larger languages.)

A program in Λ is defined below, where terminal symbols are in **bold** font and $[\alpha]_\beta$ means a list of zero or more α's separated by β's. const denotes constant values and functions; id denotes the identifiers.

$$\text{pgm} ::= \text{exp} \mid \textbf{let}\ [\text{id} = \text{exp}]_; \ \textbf{in}\ \text{pgm} \qquad (1.1)$$
$$\text{exp} ::= \text{const} \mid \text{id} \mid \text{array} \mid (\text{exp exp}) \mid \lambda\, \text{id} \,.\, \text{exp} \qquad (1.2)$$
$$\text{array} ::= \textbf{[}\ [\text{exp}]_,\ \textbf{;}\ [\text{exp}]_,\ \textbf{]} \qquad (1.3)$$

The square brackets in bold fonts denote the terminal symbols to represent arrays.

Formally, arrays are 3-tuples, <valence, shape, content>. valence is the dimensionality of the array. shape is a tuple of natural numbers specifying the number of elements in each dimension of the array. content is a tuple containing the items of the array in row major order. Empty arrays are permissible; they have at least one zero in their shape tuple. Arrays can be nested.

$$\text{array (baseSet)} = \{ \text{<valence, shape, content>} \mid \qquad (2.1)$$
$$\text{valence} \in \text{Nat}; \qquad (2.2)$$
$$\text{shape} = \text{<extent}_0, \ldots, \text{extent}_{\text{valence-1}}\text{>}; \qquad (2.3)$$
$$\text{extent}_i \in \text{Nat}; \qquad (2.4)$$
$$\text{content} = \text{<item}_0, \ldots, \text{item}_{\text{tally-1}}\text{>}; \qquad (2.5)$$
$$\text{item}_j \in \text{baseSet}; \qquad (2.6)$$
$$\text{tally} = \text{extent}_0 * \ldots * \text{extent}_{\text{valence-1}} \} \qquad (2.7)$$

valence is included in the representation of an array but not in the concrete syntax; it can be deduced from shape. Falafel does not permit *heterogeneous* arrays – all array items must be of the same type. A two dimensional array with shape (2, 3) whose items are first six natural numbers is represented as "[2, 3; 0, 1, 2, 3, 4, 5]", and corresponds to the nested tuple, <2, <2, 3>, <0, 1, 2, 3, 4, 5>>.

The semantic domain for Λ is given below. The value space, V, contains data and functions. Other constructors are not prohibited, but add nothing to our development.

$$V = \text{Int} + \text{Real} + \text{Bool} + \text{Array}(V) + (V \rightarrow V)$$

The denotational equations for Λ are given below:

$$\rho :: \text{Ident} \rightarrow V \qquad (3.1)$$
$$K :: \text{Const} \rightarrow V \qquad (3.2)$$

$$\mathbb{P} :: \text{pgm} \rightarrow \rho \rightarrow V \qquad (4.1)$$
$$\mathbb{P}\ [\![\ \textbf{let}\ [i = e]_;\ \textbf{in}\ e'\ ]\!]\ \rho = \mathbb{D}\ [\![\ e'\ ]\!]\ \rho[(\mathbb{D}\ [\![\ e\ ]\!]\ \rho)/i]_, \qquad (4.2)$$

$$\mathbb{D} :: \text{exp} \rightarrow \rho \rightarrow V \qquad (5.1)$$

$\mathbf{D} [i] \rho = \rho [i]$ (5.2)

$\mathbf{D} [c] \rho = K [c]$ (5.3)

$\mathbf{D} [(e_0 \ e_1)] \rho = (\mathbf{D} [e_0] \rho) (\mathbf{D} [e_1] \rho)$ (5.4)

$\mathbf{D} [\lambda i \ . \ e] \rho = \lambda k . \mathbf{D} [e] \rho[k/i]$ (5.5)

$\mathbf{D} [[s_0, ..., s_{v-1}; i_0, ..., i_{t-1}]] \rho =$ (5.6)

$\quad <v, <\mathbf{D} [s_0] \rho, ... , \mathbf{D} [s_{v-1}] \rho>,$

$\quad <\mathbf{D} [i_0] \rho, ... , \mathbf{D} [i_{t-1}] \rho>>$

Λ is restricted in two important ways: the shapes of all array values must be manifest at compile time; and user functions cannot be recursive. Shapes are omitted in the array notation when they are obvious (e.g., [1, 2, 3] = [3; 1, 2, 3]).

3. Intermediate container removal

Straightforward implementation of array-based functional programs leads to the generation of large numbers of *intermediate containers*. Consider f(g A) where f and g are functions from arrays to arrays (f, g : Array -> Array) and A is an array. The application of g to A produces an array – an intermediate container – which is passed to f. The systematic allocation, filling, reading, and deallocation of these intermediate containers has traditionally condemned first-class functional array operations to toy status. We would be ill-advised to adopt a purely functional array language if we had no remedy for this problem.

However, we have shown systematic means of removing these intermediate containers by finding the function, h, which is extensionally equal to the composition of f and g (h A = f(g A) for all A), but h does not generate the intermediate container [7]. *Intermediate container removal* (ICR) is a compiler optimization phase which systematically avoids the generation of intermediate containers; because Λ and its dataflow graphs are mathematically tractable, intermediate container removal can be applied to either form. (Related work was done in [13][14].) Section 7 of this paper demonstrates how the ideas of ICR are applied in this setting.

4. Representation of arrays

A *naive* approach to handle arrays in dataflow represents each array as a token, and passes these tokens on dataflow links. This approach implies arbitrary-width tokens and communication paths; it is unrealistic. The most widely accepted solutions for arrays in dataflow systems involve storage of the array in special memory locations; reading and writing of elements are carried out by special tokens and actors. In these *storage* solutions attention must be paid to the possibility of updating array elements and thereby violating referential transparency (numerous solutions have been proposed [1][12]). Other useful solutions to the representation of arrays have been proposed, such as [4].

In order to maintain ease of manipulation of dataflow graphs and array representations, it is necessary to avoid the complexities of the *storage* representations for arrays and adopt a modified *naive* representation [14]. As we shall see, this choice will not carry us all the way to our ultimate goal of practical array computations, but for a large and meaningful subset of array computations, it will be an advantageous choice. We choose two simple representations for arrays. The ***stream*** representation consists of time-ordered sequences of the items of arrays on single atomic dataflow links. The ***bundle*** representation consists of parallel collection of atomic dataflow links, each link carrying an item of the array. Operators exist to convert between streams and bundles.

R is the domain of permitted array representations. Nested arrays are represented as nested streams or bundles:

R = stream-of(Shape, Link)	(6.1)
+ stream-of (Shape, R)	(6.2)
+ bundle-of(Shape, Links)	(6.3)
+ bundle-of(Shape, R)	(6.4)

A *stream* is a shape and a time-ordered sequence of atoms on a dataflow link, (6.1), or a time-ordered sequence nested arrays on nested representations, (6.2). A *bundle* is a shape and a collection of atoms on parallel dataflow links, (6.3), or a collection of nested arrays on parallel nested representations, (6.4). In both streams and bundles the shape must be manifest at compile-time, but is not directly represented in the run-time dataflow graphs. In streams, known shape translates to known length of the sequence; in bundles, known shape translates to a known number of constituent *fibres* – each fibre has a unique identification corresponding to the addresses in the manifest shape. Figure 1 gives graphic representations for the array [0, 1, 2] as a stream, (a), and as a bundle, (b). Ovals with the long axis up and down collect streams and bundles, respectively.

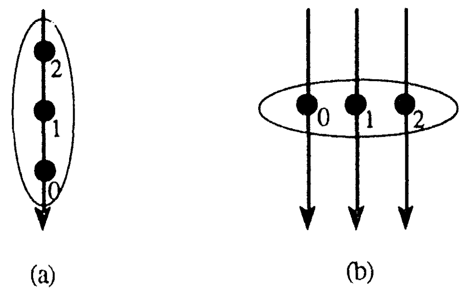


Figure 1: Stream and bundle representations of the array [0, 1, 2]

For any homogenous array of d levels of nesting, there are 2^d choices of representation. The four representations of the array [[1, 2], [3, 4]] are given in Figure 2.

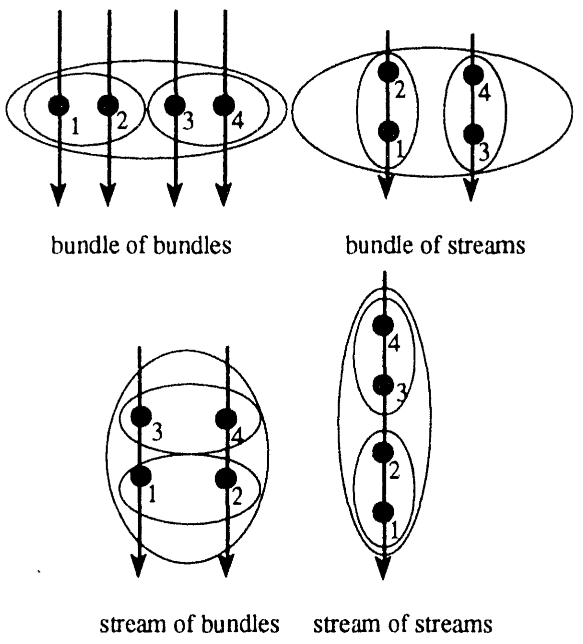


Figure 2: Four representations of the array [[1, 2], [3, 4]]

5. The dataflow model

Our abstract machine is a conventional parallel static dataflow machine [14]. No special array storage is assumed. Dataflow actors have a total of no more than four links (including inputs and outputs). The following dataflow actors are supported: dyadic operators (+, -, *, /, ^, mod, <, >, =, <=, >=, and, or), monadic operators (not, id), and six other special purpose operators.

Nodes in the dataflow graph are described as tuples, <actor; state; input; output>. actor is a tag which identifies the dataflow actor, state consists of state used by some actors, input (output) is a list consisting of the inputs (outputs) to (from) the node. The operational semantics of dataflow actors are described in terms of an operator, Fr, which maps from tuples to tuples. The treatment of input and output requires some clarification. The inputs and outputs are conceptually connected to dataflow links; the streams of values on these links are represented as queues (lists). The use of a head-tail pattern (head:tail) in the input position on the left-hand-side of a firing equation implies that the next input must be available before the node may fire. The use of a front-last pattern (front#last) in the output position on the right-hand-side implies that the last value is output. Underscore in the state position means the state is not used. An '@' sign implies an application of its first argument, which is an operator, to its latter arguments; the result of the application is an atomic value.

Dyadic, dyad, and monadic, monad, actors are defined by the patterns:

```
Fr <dyad; _; a:at, b:bt; pf> =                    (7.1)
        < dyad; _; at, bt; pf#@(dyad a b)>
Fr <monad; _; a:at; pf> =                         (7.2)
        < monad; _; at; pf#@(monad a)>
```

The six special purpose actors are: duplicate, which consumes one input and produces copies on its two outputs; switch, which consumes a value and a boolean selector and reproduces the input on the selected of two output edges; select, which consumes a control value and then selectively consumes one of two inputs and reproduces that input on its single output; replicate-n which consumes a single input and produces n copies of that value on its single output; split-m-n, which cyclically copies its first m inputs to its left output and then copies the next n-m inputs to its right output; and alternate-m-n, which cyclically copies m values from its left input and then n-m values from its right input, all to its single output. Fr rules for these actors are given below. Notice that the state is sometimes used for counting and some rules change the actor itself (e.g. from replicate-n to replicate-n').

```
Fr <duplicate; _; a:at; pf, qf> =                 (7.3)
        <duplicate; _; at; pf#a, qf#a>

Fr <switch; _; False:ct, a:at; pf, qf> =          (7.4.1)
        <switch; _; ct, at; pf#a, qf>
Fr <switch; _; True:ct, a:at; pf, qf> =           (7.4.2)
        <switch; _; ct, at; pf, qf#a>

Fr <select; _; False:ct, a:at, bt; pf> =          (7.5.1)
        <select; _; ct, at, bt; pf#a>
Fr <select; _; True:ct, at, b:bt; pf> =           (7.5.2)
        <select; _; ct, at, bt; pf#b>

Fr <replicate-n; -1, _; a:at; pf> =               (7.6.1)
        <replicate-n'; n-1, a; at; pf#a>
Fr <replicate-n'; 0, a; at; pf> =                 (7.6.2)
        <replicate-n; -1, _; at; pf>
Fr <replicate-n'; c, a; at; pf> =                 (7.6.3)
        <replicate-n'; c-1, a; at; pf#a>

Fr <split-m-n; -1; a:at; pf, qf> =                (7.7.1)
        <split-m-n'; m-1; at; pf#a, qf>
Fr <split-m-n'; 0; a:at; pf, qf> =                (7.7.2)
        <split-m-n''; n-m-1; at; pf, qf#a>
Fr <split-m-n'; c; a:at; pf, qf> =                (7.7.3)
        <split-m-n'; c-1; at; pf#a, qf>
Fr <split-m-n''; 0; at; pf, qf> =                 (7.7.4)
        <split-m-n; -1; at; pf, qf>
Fr <split-m-n''; c; a:at; pf, qf> =               (7.7.5)
```

<split-m-n''; c-1; at; pf, qf#a>

Fr <alternate-m-n; -1; a:at, bt; pf> = (7.8.1)
<alternate-m-n'; m-1; at, bt; pf#a>
Fr <alternate-m-n'; 0; at, b:bt; pf> = (7.8.2)
<alternate-m-n''; n-m-1; at, bt; pf#b>
Fr <alternate-m-n'; c; a:at, bt; pf> = (7.8.3)
<alternate-m-n'; c-1; at, bt; pf#a>
Fr <alternate-m-n''; 0; at, bt; pf> = (7.8.4)
<alternate-m-n; -1; at, bt; pf>
Fr <alternate-m-n''; c; at, b:bt; pf> = (7.8.5)
<alternate-m-n''; c-1; at, bt; pf#b>

Schematic representations for these six special actors are given in Figure 3.

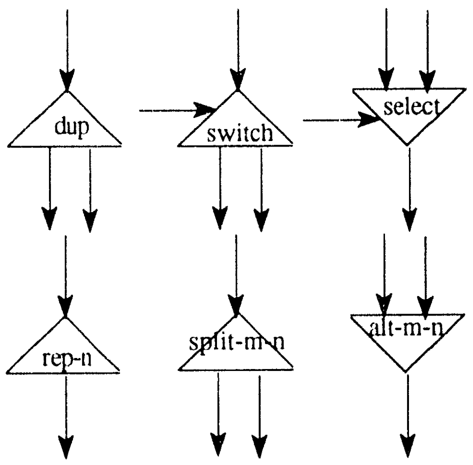


Figure 3: Schematic representations of special actors

The operations which have state (or modify their actor codes) all have simple cycles back to their initial state (and code). All uses of these actors in the graphs generated by our compiler cause complete circuits back to the initial state. In this sense, the graphs are self-cleaning.

The above six actors have corresponding macro versions which accept bundles and act on the individual elements separately. In the case of duplicate the macro actor DUPLICATE accepts a bundle and – with a set of duplicate actors – accomplishes a DUPLICATE on the entire bundle. Likewise, multiple input SELECT's and ALTERNATE's are constructed of trees of select's and alternate's. In text, we freely use these macro-actors in place of their corresponding graphs, if necessary.

Any nested representation of an array can be converted to any other representation. We illustrate the recovery of parallelism from sequentiality in nested arrays. Figure 4 shows how to use split-m-n operators to recover a bundle-bundle representation from a stream-stream representation of [[1, 2], [3, 4]].

Recent dataflow research indicates that it is advantageous to gather atomic actors into sequential threads which communicate through conventional registers [15]. We do not carry out this transformation on our graphs, although we could. We avoid this as an unnecessary complication to our purposes.

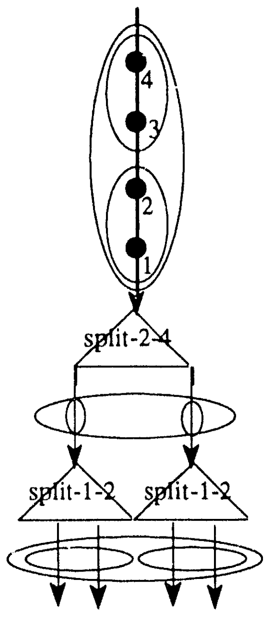


Figure 4: Recovering nested bundles from nested streams

6. Primitive operations on arrays

Research in array-based computation has produced numerous useful operations on arrays (first-order and higher-order). These operations constitute our basic tool set for constructing array-based functional computations. This section defines some of these operators (in a much abbreviated form, see [7], [8], and [10] for full details). All of these operations have dataflow graphs for both the stream and the bundle representations of arrays; we give several representative examples.

addresses are one-dimensional arrays that contain only natural numbers. If an address has the same valence as the shape of an array, and all the items of the address are less than the corresponding items of the shape, then the address uniquely addresses an item of the array.

grid generates an array with the shape of its argument, but with all items replaced by their own addresses. For example:

grid [2, 2; 0, 1, 2, 3] = [2, 2; [2; 0, 0], [2; 0, 1], [2; 1, 0], [2; 1, 1]] pick takes a (valid) address and an array and returns the specified item. It can be represented in infix form too. For example:

```
pick [2; 1, 0] [2, 2; 0, 1, 2, 3] = 2
```

shape returns the shape of an array as an array. For example:

```
shape [2, 2; 0, 1, 2, 3] = [2; 2, 2]
```

The semantics of Falafel, as described in [7], permit the definition of arrays by parts. The semantics of definition-by-parts is based on a complete partial order of partially defined arrays; the meaning of a definition-by-parts is the least upper bound of all the partial definitions. In Falafel, syntactic sugar is provided for this mechanism. An array, p, is defined by parts, p_i, by the form {p **suchthat** p_0, p_1, ... }, where the p_i are partial definitions. p is the least upper bound of all the p_i. All such forms can be compiled down to Λ, but we make use of this convenient syntax in defining further fundamental array operations.

each is the array equivalent of the familiar map operation:

```
each f a =     {z suchthat shape z = shape a;   (8)
                   forall x in grid a, (x pick z) =
                       (f (x pick a))}
```

each can be implemented as a dataflow graph for either stream or bundle arrays. Figure 5 gives both versions.

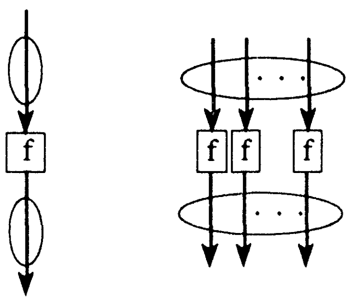


Figure 5: Stream and bundle dataflow graphs for 'each'

There is a family of operations related to each (eachleft, eachright, and eachboth). To conserve space, we present only eachleft; its stream-stream and bundle-bundle graphs are given in Figure 6.

```
eachleft f t r = {z suchthat shape z = shape t;   (9)
                    forall x in grid t, (x pick z) =
                        (f (x pick t) r)}
```

reach takes a list of addresses and a nested array. It uses pick to select an item using each address in the list of addresses in turn. [head | tail] is a head-tail decomposition notation for one-dimensional arrays.

```
reach [] a = a                                    (10)
reach [h|t] a = reach t (h pick a)
```

rows and cols return the list of rows and the list of columns, respectively, from a two-dimensional array. mix is the inverse of rows.

```
rows tab = {z suchthat                            (11)
             (shape z) = [(first (shape tab))];
             forall fi in grid z, (shape (fi pick z)) =
                        [(second (shape tab))];
             forall fi in grid z,
                 forall si in grid [(first z)],
                     ([fi, si] reach z) =
                           ([fi', si'] pick tab)}
             where [fi'] = fi; [si'] = si
```

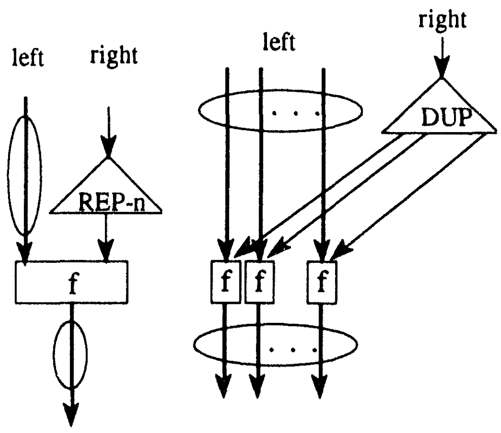


Figure 6: Stream-stream and bundle-bundle dataflow graphs for 'eachleft'

```
cols tab = {z suchthat                            (12)
             (shape z) = [(second (shape tab))];
             forall fi in grid z, (shape (fi pick z)) =
                        [(first (shape tab))];
             forall fi in grid z,
                 forall si in grid [(first z)],
                     ([si, fi] reach z) =
                           ([fi', si'] pick tab)}
             where [fi'] = fi; [si'] = si

mix lst = {z suchthat                             (13)
             (shape z) = [first (shape lst),
                     first (shape (first lst))];
             forall fi in grid z,
                 (fi pick z) =
                 ([[first fi], [second fi]] reach lst)}
```

reshape applies a new shape to an array, for example:

reshape [2; 1, 4] [2, 2; 0, 1, 2, 3] =
[1, 4; 0, 1, 2, 3].

link collapses one level of structure in nested arrays, for example:

link [4; [2; 0, 0], [2; 0, 1], [2; 1, 0], [2; 1, 1]] =
[8; 0, 0, 0, 1, 1, 0, 1, 1].

All of the operations presented in this section have dataflow graphs for both stream and bundle representations of all array arguments. Space constraints prohibit their presentation. They are used in the compilation process reported in the next section.

7. Compilation

This section describes – at a very high level – the process of compilation from Λ to static dataflow graphs comprised of the primitives discussed in Section 6. Compilation begins with a conventional translation to a nested abstract syntax tree in which branches are function application and leaves are arrays, atoms, or functions.

Consider the type of every function, for example, each, which is of type $[\alpha] \text{->} [\beta]$. The number of array constructors, [ ], in a type specification corresponds to the number of arrays (independent or nested) taken as arguments or returned as results by the function. For each array there is a choice of whether to use a stream or bundle representation. Where a function has d array constructors, there are 2^d choices for the representations of arrays. In theory, there are static dataflow graphs for each such choice (although some of the graphs are clearly never useful). Each primitive in Λ has this diversity of corresponding dataflow graphs.

Static dataflow graph code generation for Λ consists of four phases:

1) Shape information is propagated throughout the abstract syntax tree. After this process, the shape of every intermediate and final result is known.

2) Choices are made regarding which versions of primitives are to be used. As these choices are made, the primitives are instantiated as graphs. The shapes of arrays are used in this process to determine the width of bundle-represented arrays.

3) Normally, the representation choice of the output of one function will be the same as the representation choice for the input of the next function. In this case, the two dataflow graphs can be directly connected. On occasion, however, it may be useful to make differing representation choices in composed primitives. In this case, stream-to-bundle or bundle-to-stream conversion graphs must be interposed between the primitives. In this manner, the entire graph is constructed.

4) Having constructed the complete graph, optimizations are applied directly to the graph. These optimizations correspond to intermediate container removal, discussed above. In static dataflow graphs, where the output array of one function is just a (collection of) dataflow link(s), intermediate container removal is essentially automatic. Output dataflow links are simply connected to the appropriate inputs – the intermediate container is only the conceptual bundle (or stream), which has no physical reality.

It should be pointed out that the resultant static dataflow graphs have no run-time overheads for arrays. Only the dataflow actors and their connection links are involved in array processing. The entire cost of supporting arrays is paid in the size of the graph.

8. Multiple representations give control over parallelism and graph size

In this section we examine the nature of the dataflow graphs that can be generated using the approach established so far. We begin by examining some of the dataflow graphs that can be generated for a simple example, shuffle, which performs the perfect shuffle operation.

shuffle a = link (cols (reshape (14)
[2, (first (shape a)) div 2] a))

shuffle is illustrated by example. It is applied to a list of ten integers. Arrays are diagrammed as boxes to show spatial arrangement and nesting [10].

1	2	3	4	5	6	7	8	9	10

reshape arranges the argument into two rows:

1	2	3	4	5
6	7	8	9	10

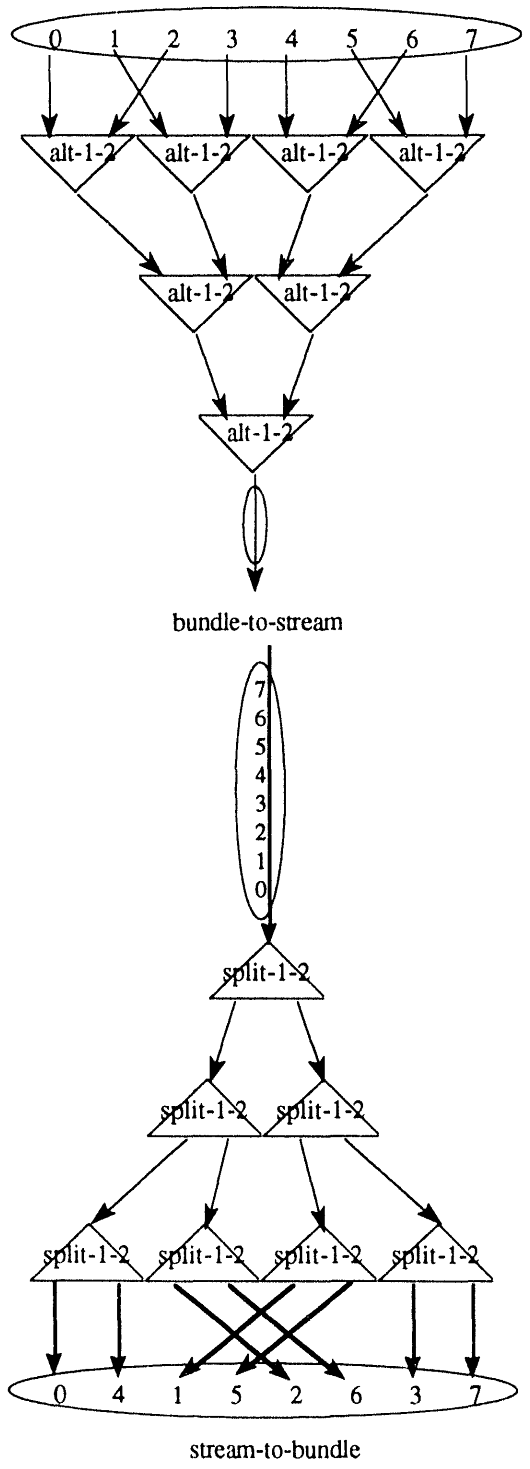
cols interleaves the items in the rows by "picking up" the columns, making a list of pairs:

1	6	2	7	3	8	4	9	5	10

link removes the extra structure added by cols, resulting in the shuffled array:

1	6	2	7	3	8	4	9	5	10

Four dataflow graphs can be generated for shuffle, corresponding to all four combinations of input and output representations: bundle-to-bundle, bundle-to-stream, stream-to-bundle, and stream-to-stream. These are drawn in Figure 7. The optimizations of intermediate container removal have been applied to these graphs.



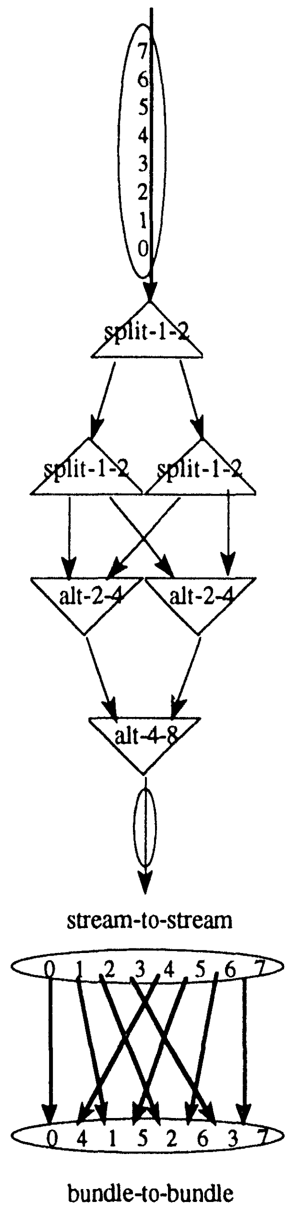


Figure 7: The four possible dataflow graphs for 'shuffle'

Notice that the bundle-to-bundle form of shuffle has no run-time cost; it is simply a wiring plan. The component operations of shuffle (link, cols, and reshape) naturally "wire up" without any intermediate containers during our compilation process. Given our array representation choice, it is impossible to generate intermediate containers. The composition of bundle-to-bundle "purely restructuring" operations such as link, cols, and reshape always results in a simple wiring pattern such as the bundle-to-bundle shuffle.

Unlike the each family of operators – which have equally good stream-to-stream and bundle-to-bundle graphs – shuffle is most naturally bundle-to-bundle. The bundle-to-stream and stream-to-bundle forms do have an important role: they permit simultaneous re-arrangement and fan-in or fan-out of parallelism. If the operations before (after) shuffle are time consuming and the operations after (before) shuffle are simple, then the bundle-to-stream (stream-to-bundle) forms of shuffle are advantageous.

The following example expression illustrates our ability to tailor parallelism using different choices of array representations and corresponding choices of primitives. The expression to be evaluated is each (each incr) [[0, 1], [2, 3]].

In Figure 8 we see a useful control of parallelism. Where i is the tally of the outer array and j is the tally of each inner array, we can perform the computation in O(1), O(i*j), O(i), or O(j) (not shown).

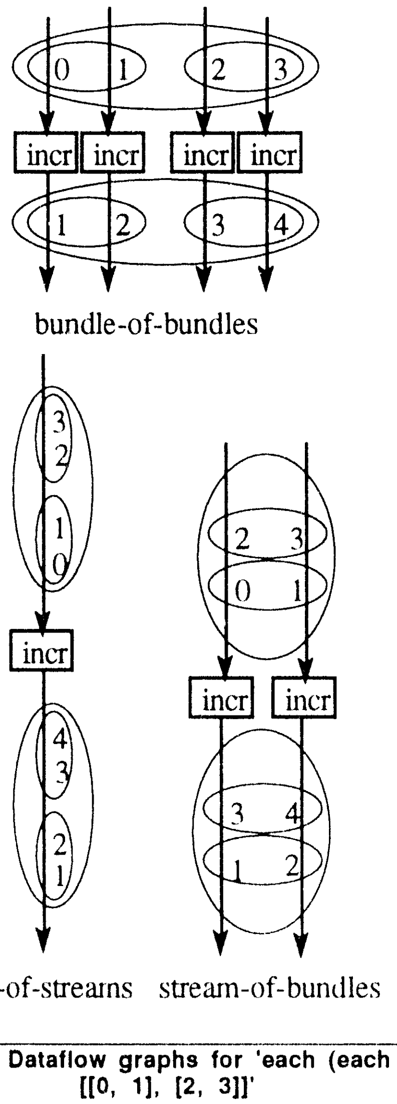


Figure 8: Dataflow graphs for 'each (each incr) [[0, 1], [2, 3]]'

Given any non-recursive, known-shape expression in A (or Falafel), it is possible to compile it to a dataflow graph. During this process, a decision must be made for every array (arguments, results, and intermediate containers) whether the array's representation is to be stream or bundle. Where there are c such choices in a program, there are 2^c possible dataflow graphs for the program (shuffle has 2^2 possible graphs). These graphs can be expected to exhibit a wide range of parallelism and graph size characteristics. If we generate all (or some) such graphs and plot parallelism versus graph size, we have a set of semantically equivalent implementations for the original program. The parallelism/graph-size measures can be used to select the appropriate trade-off for a given dataflow machine.

Several measures of the static and dynamic properties of dataflow graphs are made. *Static graph size* is the count of nodes in the graph. *Static graph depth* is the maximum latency through the graph. *Static average parallelism* is the ratio *static-graph-size / static-graph-depth*. *Run time* is the number of cycles required to complete execution of the graph, where an infinite number of processors are available. *Dynamic minimum (maximum, average) parallelism* is the least (greatest, average) number of actors firing on any given cycle.

The first program example that we shall investigate generates all cross-products of two nested lists of integers:

cp a b = eachleft (eachright (eachboth *)) a b (15)

We generate five different graphs for cp. The cases considered are described in Table 1. The argument used in the first two examples are of the form [2; [3; ...] ...]. In the next two examples the arguments are [3; [4; ...] ...]. The last example uses arguments of the form [7; [4; ...] ...].

It is apparent from the results in Table 1 that choices of stream versus bundle representations, and the corresponding choices of operators, give a degree of control over the parallelism and graph size of a dataflow program.

Table 2 gives results for a larger program fragment: matrix multiply. In all the examples in Table 2, the decision has been made to present the rows of the original arguments to matrix multiply, rather than the unnested two-dimensional arrays. The rows (list of lists) are presented in bundle-of-streams form. Consequently the parallelism and graph sizes for these runs are intermediate between the fully sequential and fully parallel versions.

Table 1: Parallelism and graph size for cross product

	cp-1	cp-2	cp-3	cp-4	cp-5
outer shape	2	2	3	3	7
inner shape	3	3	4	4	4
outer repr.	bun-dle	bun-dle	bun-dle	bun-dle	bun-dle
inner repr.	stre-am	bun-dle	stre-am	bun-dle	stre-am
static graph size	8	24	21	84	133
static graph depth	2	2	3	3	4
static avg. par.	4	12	7	28	33.25
oper. cnt.	24	24	84	84	532
run time	6	2	10	3	11
dyn. min. par.	4	12	4	24	14
dyn. max. par.	4	12	11	32	67
dyn. avg. par.	4	12	8.4	28	48.4

The results of this section offer considerable promise. We have been able to specify algorithms in a completely declarative form, without making any commitment to any particular form of parallelism. From these specifications, we have been able to generate a variety of static dataflow graphs, each presenting different options on the parallelism/graph-size tradeoff. The most appropriate tradeoff can be selected by the user (or – potentially – automatically).

In the next section we exploit more subtle means to generate a finer range of tradeoffs between parallelism and graph-size.

9. Array equivalence laws enhance control over parallelism and graph size

Referentially transparent functional array languages, with appropriate primitive operations, have large numbers of "universal laws" (identities) which describe equivalences between expressions [5]. For instance, the composition of **each** applied to two functions is equivalent to the each of the two composed functions:

(each f) o (each g) A = each (f o g) A (16)
for all f, g, and A

Table 2: Parallelism and graph size for matrix multiplication

	mm-1	mm-2	mm-3
first matrix shape	[5, 3]	[7, 3]	[7, 4]
second matrix shape	[3, 5]	[3, 7]	[4, 7]
outer repr.	bundle	bundle	bundle
inner repr.	stream	stream	stream
static graph size	440	868	868
static graph depth	11	11	11
static avg. par.	40	78.9	78.9
operation count	1070	2114	2933
run time	16	16	22
dynamic min. par.	25	49	49
dynamic max. par.	137	261	299
dynamic avg. par.	66.9	132.1	133.3

These laws greatly enhance our power to refine the trade-off between parallelism and graph size. Consider the application of (each incr) to a two-dimensional table of integers. The techniques developed in Section 6 give us only two results – the completely sequential graph and the completely parallel graph described in the first two columns of Table 3.

Three laws are available to improve our control over the parallelism in this simple example. The first law is based on the inverse relationship of mix and rows; it says that we can break a table into its rows, apply **each** f to each row, and then reassemble the table with mix.

each f a = mix (each (each f) (rows a)), (17)
when valence a = 2

Once we have obtained the **each-each** form on the right-hand-side of (17), we are free to choose stream or bundle representations for either level. As a consequence, we can process each row sequentially and all columns in parallel, or vis-a-versa. Thus where the table is m-by-n, we have O(m) and O(n) algorithms as well as the O(m*n) and O(1) algorithms. Statistics for these new versions are given on columns three and four of Table 3.

A dual law breaks the table in to a list of columns using the operations cols and transpose (which is not defined in this paper).

```
each f a  =  transpose (mix                          (18)
                (each (each f) (cols a))),
                when valence a  =  2
```

Another strategy for factoring tables into bundles of streams (or vis-a-versa) is to divide the table in half (thirds, forths, ...). The following identify breaks tables in half (take and drop are not defined in this paper).

```
each f a  =  (shape a) reshape (link                 (19)
                [each f firsthalf, each f secondhalf])
                where firsthalf =
                          (tally a div 2) take a;
                      secondhalf =
                          (tally a div 2) drop a
```

Table 3: Parallelism and graph size for '(each incr)'

	seq.	par.	row	col	half	half
sha-pe	5, 4	5, 4	5, 4	5, 4	5, 4	5, 4
repr	stre-am	bun-dle	bun-dle_ stre-am	stre-am_ bun-dle	bun-dle_ stre-am	stre-am_ bun-dle
stat. grp. sz.	1	20	5	4	2	10
stat. grp. dep.	1	1	1	1	1	1
stat. avg. par.	1	20	5	4	2	10
run ti-me	20	1	4	5	10	2
dyn. min par.	1	20	5	4	2	10
dyn. max par.	1	20	5	4	2	10
dyn. avg. par.	1	20	5	4	2	10

The results for this strategy with bundle-of-streams and stream-of-bundles representations are given in the last two columns of Table 3.

The above strategies factor two-dimensional arrays. They all generalize to higher dimensional arrays. An operation could be mapped on an array with shape [i, j, k] in O(1), O(i*j), O(i*k), O(j*k), O(i), O(j), O(k), or O(i*j*k) time. The graph sizes for these computations would be the reverse order of the time complexities, respectively.

Reductive applications of functions (e.g., reduce f i [a, b, c] = a f (b f (c f i))) can be performed sequentially, or – for associative functions – in a binary tree. Thus we can reduce arrays of shape [n] in either space O(1) and time O(n), or in space O(n) and time O(log n). When reducing large arrays, the reduction can be broken into two levels of reduction using the various strategies of rectangular factoring discussed above. For instance:

```
reduce f i a  =  reduce f i (each (reduce f i) (rows a))    (20)
```

when i is the identity of f, and f is associative. One reduce can be done sequentially and the other in parallel. This approach can be generalized to more than two levels of reduction if the architecture makes this a useful strategy.

Four patterns of factoring of expressions are exposed by our referentially transparent model of array computations: 1) re-arrangement operations, such as our shuffle example, 2) cross product, as illustrated in the eachleft-eachright-eachboth example, 3) operation mapping, as illustrated in the each-incr and matrix multiplication examples, and 4) the reductive patterns just discussed. These four patterns can be combined in any form in more complex examples.

10. Conclusions

We have systematically exploited identities and representation choices in a referentially transparent system of array expressions and dataflow graphs to control the degree of parallelism. This work has demonstrated, in principle, that array-based computations can be written in a very general form and can then be mapped to particular dataflow machine. Thus dataflow users can avoid early-binding of control of parallelism strategies.

A few fundamental patterns of factoring of array computations have been exposed by our examples. While the set of patterns is small, they are powerful strategies, and can be applied in combination to produce more complex possibilities for more complex problems.

While these factoring ideas are not new, our systematic exploitation of them appears to be both new and promising.

The present work is preliminary; there are many limitations. Presently, only structural recursions (recursions where the inductive principle is the structure of the array) are handled; our ideas do not yet extend to course-of-values recursions.

Only application to static dataflow has been studied. While there does not appear to be any serious difficulty in extending these ideas to dynamic dataflow, this work remains.

Our methods require that the shape of all arrays be known. This restriction is equivalent to the restriction to static dataflow – it is necessary that the structure of the final dataflow graph be known, and that structure is dictated by the shape of containers. This restriction can be relaxed for stream representations (which are not reconverted to bundles), but we have not done so.

There are important array computations which our techniques do not address. Computations which perform dynamic look-ups in arrays do not map well to our approach. For these computations, the storage solutions are superior. In our view, the present techniques should be applied within the larger context of an array dataflow system that uses storage solutions where necessary or preferable, but uses our techniques where appropriate to avoid the overheads of storage solutions, or to better control parallelism. The relationship between storage solutions and our approach has not been worked out.

The primary future effort suggested by this work is the automation of the exploration of the domains of possible graphs for array dataflow expressions. The choices of stream and bundle representation, and the applications of identities exploited in this paper were all human decisions. If this approach is to be practical, automated systems must find the better graphs. As the number of graphs is exponential in the number of arrays and identities, blind search will not be useful for large expressions. Heuristic approaches must be established.

The second future effort suggested by this work is to embed our approach within the larger context of a storage solution, as discussed above.

The success of this approach is due to the referential transparency of the source expressions and the dataflow graphs. While the approach cannot be expected to solve all dataflow programming problems, it is a useful approach for a large and interesting subset of array computations.

References

[1] Arvind and Ianucci R., "Two Fundamental Issues in Multiprocessing: The Data Flow Solution", Tech Report 226-2, Laboratory for Computer Science, MIT, July 1983.

[2] Dennis J.B., "Data Flow Supercomputers", IEEE Computer, November 1980, pp 48-56.

[3] Feo, J. T., Cann, D. C. and Oldehoeft, R. R., "A Report on the Sisal Language Project", Journal of Parallel and Distributed Computing, Vol. 10, 1990, pp. 349-366.

[4] Gaudiot, J. L. and Wei, Y., "Token Relabeling in a Tagged Token Dataflow Architecture", IEEE Transactions on Computers, Sept. 1989.

[5] Jenkins M.A., "The Role of Equations in Nial", Tech Report# 84-161, Department of Computing and Information Science, Queen's University, Canada, 1984.

[6] Jenkins M. A. and Jenkins W. H., "Q'Nial Reference Manual", Nial Systems Limited, Canada 1985.

[7] McCrosky C., "The Elimination of Intermediate Containers in the Evaluation of First-Class Array Expressions", Computer Languages, Vol 16, No. 2, pp 179-195.

[8] McCrosky C., Sailor K., and ven der Buhs, B., "The Semantics of Falafel", report in progress, University of Saskatchewan.

[9] "Miranda System Manual", Research Software Limited, 1987.

[10] More T., "Notes on the Diagrams, Logic and Operations of Array Theory", Tech Report G320-2137, IBM Cambridge Scientific Center, 1981.

[11] Nikhil R. S., "Id (Version 90.0) Reference Manual", Tech Report CSG Memo 284-1, MIT Laboratory for Computer Science, Cambridge, USA, July 1990.

[12] Pingali, K. and Arvind, "Efficient Demand-Driven Evaluation. Part I", ACM TOPLAS, April 1985, pp 311-333.

[13] Ranelletti, J. E., "Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages", Ph.D. Thesis, University of California at Davis, 1987.

[14] Roy K., "Static Dataflow Implementation of First-Class Arrays", M. Sc. Thesis, Department of Computational Science, University of Saskatchewan, Canada, 1990.

[15] Traub, K., "Multi-thread Code Generation for Dataflow Architectures from Non-Strict Programs", FPCA'91, Cambridge, pp. 73-101.

END

DATE
FILMED

3/7/94