Proceedings of the Second Sisal Users' Conference

Editors:
John T. Feo
Christopher Frerking
Patrick J. Miller

December 1992

DISCLAIMER

This report has been reproduced
directly from the best available copy.

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62, Oak Ridge, TN 37831
Prices available from (615) 576-8401, FTS 626-8401

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd.,
Springfield, VA 22161

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

CONF-9210270--

DE93 008019

Proceedings

of the

Second Sisal Users' Conference

Editors
John T. Feo
Christopher Frerking
Patrick J. Miller
Lawrence Livermore National Laboratory
Livermore, California

Sponsored by

Computer Research Group
Lawrence Livermore National Laboratory
Livermore, CA

October 4-5, 1992
San Diego, California

Contents

A SISAL CODE FOR COMPUTING THE FOURIER TRANSFORM ON S_N*

Jesus Novoa, Flor Sanmiguel and Jaime Seguel
Department of Mathematics, University of Puerto Rico at Mayaguez, PR 00681

Abstract

Non-abelian FFT's appear in connection with the statistical analysis of ranked data. Fast algorithms for computing non-abelian FFTs that are based upon a generalization of common FFT techniques have been proposed. However, the presence in non-abelian FFTs of group representations makes the coding of a non-abelian FFT much more complex, in particular, no parallel or vector implementations can be naturally derived by manipulating the non-abelian FFT mathematical formulas. In this paper, a SISAL code based on an almost direct translation of the mathematical expression of an FFT on the symmetric group S_N is presented. Especially useful features in SISAL in expressing the algorithm are ragged arrays and dynamic sized arrays.

1. Introduction. Throughout this work some standard concepts in algebra and linear algebra as well as some of their properties are used. Among them, perhaps the most important ones are the concepts of finite group and linear group representation. By a finite group we understand a finite set endowed with an operation under which the properties of clousure, associativity, existence of a unique identity and existence of a unique inverse for each element in the group are satisfied. A linear group representation of G (or simply, a representation) is a map ρ defined on G and with values in a space of square matrices that satisfies $\rho(g_1g_2) = \rho(g_1)\rho(g_2)$ for all g_1, g_2 in G. A representation is called reducible if there exist two representations ρ_1 and ρ_2 such that $\rho(g) = \rho_1(g) \oplus \rho_2(g)$ for all g in G. Here $\oplus$ denotes the direct sum of matrices. If such a pair of representations does not exist, then ρ is said to be irreducible. A good account of the finite group representation theory can be found in Serre [12].

Let f be a complex-valued function defined on a finite group G and let ρ be any complex irreducible representation of G. The *discrete Fourier transform* (DFT) of f over G with repect to ρ is the matrix

$$\hat{f}(\rho) \quad = \quad \sum_{s \in G} f(s)\rho(s), \tag{1}$$

Any abelian group of order N is isomorphic to Z_N, the set of integers modulo N endowed with addition modulo N. Furthermore, every irreducible representation of Z_N can be uniquely identified with an N-th root of unity; that is, an expression of the form ω_N^t, where $\omega_N = e^{2\pi i/N}$, $i = \sqrt{-1}$ and $t \in Z_N$. Therefore, formula (1) is a generalization of the better known

$$\hat{f}(t) \quad = \quad \sum_{s=0}^{N-1} f(s)\omega_N^{ts},\ t = 0, ..., N-1, \tag{2}$$

the so-called N-point discrete Fourier transform.

Throughout this work G denotes a finite non-abelian group. Consequently, the operator defined by formula (1) is called non-abelian DFT of f on G, while the one defined in (2) is called simply the abelian DFT of f.

Motivated by its numerous applications a large class of fast algorithms, collectively known as fast Fourier transforms (FFT's), have been designed for computing the abelian DFT of a complex function f. Complete treatments of abelian FFT's can be found in Aho *et al.* [1].

* This work was supported in part by NSF grant RII-8905080 and the Epscor II grant

The non-abelian DFT has only recently begun to attract attention. In works by Diaconis and Rockmore [5] the need for efficient methods for spectral analysis in non-abelian groups appears in connection with the statistical analysis of ranked data. Spectral analysis in non-abelian groups is also utilized in the study of random walks on groups. As a consequence of these applications, the search for fast non-abelian Fourier transform algorithms (non-abelian FFT's) has become an important research area for algorithm designers.

Recently, Rockmore [11] and Clausen [3] have proposed non-abelian FFT's involving $O(|G|\log|G|)$ arithmetic operations. These complexity bound estimates are similar to those obtained in the abelian case by Cooley and Tukey [4], Good [6], Winograd [13], Rader [10] and Bluestein [2]. These great savings in arithmetic operations are achieved by expressing the non-abelian DFT on G in terms of a sequence of non-abelian DFTs on a selected subgroup H of G. This expression constitutes a basic splitting formula for a recurrence relation. The algorithm is obtained by iterating on this recurrence relation down a tower of subgroups of G.

Indeed, the mathematical principles behind abelian and non-abelian fast Fourier transform's design are essentially the same. In order to illustrate this assertion we show below some of the mathematical facts involved in obtaining the Cooley-Tukey algorithm for $n = 12$ and the non-abelian FFT on S_3, the group of permutations on 3 elements.

Let f be a complex map defined on $Z_{12} = \{0, ..., 11\}$. Let us select the subgroup $H = 3Z_4 = \{0, 3, 6, 9\}$. Then, the classes of Z_{12}/H are the sets H, $1+H = \{1, 4, 7, 10\}$ and $2+H = \{2, 5, 8, 11\}$. The set $R = \{0, 1, 2\}$, which is formed by taking one and only one element from each of the sets H, $1 + H$ and $2 + H$, is called a set of representatives. The restriction of f to $a + H$ is denoted f_a. Now, the 12-point abelian DFT can be written as

$$\hat{f}(t) = \hat{f}(\omega_{12}^t) = \sum_{a \in R} \omega_{12}^{ta} \sum_{s \in H} f_a(s)\omega_{12}^{ts}, \tag{3}$$

which is a basic splitting equation for the 12-point abelian FFT.

On the other hand, if f is a complex map defined on $G = S_3$, and if S_2, the group of permutations of two elements is selected as a subgroup, then the classes in G/H are the sets S_2, $(13)S_2$ and $(23)S_2$. Here (ab) represents the permutation that interchanges the elements a and b and leaves the others alone. A set of representatives is $R = \{1, (13), (23)\}$, where 1 stands for the identity map.

The group S_3 possess three irreducible representations, two of them, the identity and the "sign" representation, are of order one. The other is a representation of order two. The values of this representation in seminormal Young form on the set of representatives are

$$\begin{aligned} \rho(1) &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \\ \rho(13) &= \begin{pmatrix} 1/2 & -3/4 \\ -1 & -1/2 \end{pmatrix}, \\ \rho(23) &= \begin{pmatrix} 1/2 & 3/4 \\ 1 & -1/2 \end{pmatrix}. \end{aligned} \tag{4}$$

The non-abelian FFT is obtained by means of the equations

$$\hat{f}(\rho) = \sum_{a \in R} \rho(a) \sum_{s \in H} f_a(s)\rho(s) \tag{5}$$

where f_a denotes the restriction of f to the class aS_2.

The iterated use of these formulas on a tower of subgroups leads to the tree structure that characterizes FFT's. However, when an irreducible representation of S_n is restricted to S_{n-1}, the restriction can be expressed as the direct sum of irreducible representations of S_{n-1}. Theorem 1 of section 2 establishes the

relationship between the irreducible representation of S_{n-1} and the restriction of a representation of S_n to S_{n-1}. Furthermore, Diaconis and Rockmore [5] have shown that a represention given in Young seminormal form is obtained without changes in basis. Unfortunately, the mathematical similarities between abelian and non-abelian FFT's are no longer valid when it comes to implementing non-abelian FFT's on general purpose machines. Another difference between the abelian and non-abelian cases is the fact that while each n-th root of unity is just a complex number, inexpensive to generate and store, representations are square matrices of different sizes, very expensive to generate and store. Furthermore, in non-abelian FFT's, some intermediate transforms might be stored in order to save arithmetic operations.

Generating the matrix representations of a non-abelian group is a highly complex task since only a few algorithms for computing the representations of the group generators are known and therefore, the matrix representations of the remaining elements must be computed by matrix conjugation. In this work, an algorithm for computing the seminormal Young representations is developed and used as a part of the non-abelian FFT code.

Unlike the abelian case, no standard parallel or vector versions of non-abelian FFT's are known. Again, the mathematical analogies between abelian and non-abelian FFT's cannot help in vectorizing or parallelizing non-abelian FFT's. As can be easily seen from its tree, the abelian FFT possesses a very regular data flow. In particular, the amount of operations involved at each level of the abelian tree is always the same. This fact allows very natural modifications on the data flow for adapting the algorithm to vector pipeline processing (Korn *et al.*) [8] or to distributed memory multiprocessors (Pease) [9] . In the non-abelian FFT tree, however, the number of operations involved varies from one level to the other as the matrix representations change in size. Even on a fixed level, the number of operations vary from one branch to another according to the different degrees of sparcity of the matrix representations involved. Therefore, the non-abelian FFT tree expresses neither a true potential parallelism nor a potential vectorization. These facts have motivated the functional approach to the computation of the DFT on S_N that will be presented in the next section.

2. Programming an FFT on S_N. The main steps involved in computing the discrete Fourier transform on S_N are

STEP 1. Compute the seminormal Young representations of S_N,

STEP 2. Compute the DFT by the nested sequence of sums:

$$\sum_{i_N=1}^{N} \rho(s_{i_N})[\sum_{i_{N-1}=1}^{N-1} \cdots [\sum_{i_k=1}^{k} \rho(a_{i_k})[\sum_{s \in S_{k-1}} f_{i_N \ldots i_k}(s)\rho(s)]] \ldots] \tag{6}$$

If ρ decomposes over S_k in the form $(\rho \downarrow S_k) = \rho_1 \oplus \ldots \oplus \rho_l$ the equation (6) becomes

$$\sum_{i_N=1}^{N} \rho(s_{i_N})[\sum_{i_{N-1}=1}^{N-1} \cdots [\sum_{i_k=1}^{k} \rho(a_{i_k}) \begin{pmatrix} \widehat{f}_{i_N \ldots i_k}(\rho_1) & & \\ & \ddots & \\ & & \widehat{f}_{i_N \ldots i_k}(\rho_l) \end{pmatrix}]]. \tag{7}$$

As pointed out in the introduction, step 1 is by far more complex than step 2. The algorithm for generating all the matrices of the seminormal Young representation uses the fact that there exist a one to one correspondence between these representations and all the integer partitions of N. A partition of N is an r-tuple $\mu = (\mu_1, \ldots, \mu_r)$ such that $N = \sum_{j=1}^{r} \mu_j$. We denote by ρ_μ the representation associated with the partition μ. The precise form of ρ_μ can be obtained only after associating with μ a collection of tables called standard tables. In order to compute these tables, a diagram called the Ferre diagram, is associated with μ. This diagram consists of N nodes distributed over r rows. The i-th row contains μ_i nodes. The standard tables are obtained by filling the nodes in this diagram with numbers from 1 to N, with no repetitions and

increasing from left to right in each row and from top to bottom in each column. For instance, the standard tables for the partition $\mu = (2,2,1)$ of 5 are

$$T_1 = \begin{matrix} 1 & 4 \\ 2 & 5 \\ 3 \end{matrix} \qquad T_2 = \begin{matrix} 1 & 3 \\ 2 & 5 \\ 4 \end{matrix} \qquad T_3 = \begin{matrix} 1 & 2 \\ 3 & 5 \\ 4 \end{matrix} \qquad T_4 = \begin{matrix} 1 & 3 \\ 2 & 4 \\ 5 \end{matrix} \qquad T_5 = \begin{matrix} 1 & 2 \\ 3 & 4 \\ 5 \end{matrix}.$$

The standard tables associated with a partition of N can be ordered by the following rule: If N appears in T in the i-th row and N appears in T' in the j-th row and $i < j$, then $T < T'$. If N is in the same row in both tables, then the same rule is applied but with $N-1$ instead of N.

The axial distance between x and y with respect to the table T_i is defined to be

$$d^i(x,y) = (c(x) - c(y)) + (r(y) - r(x)) \tag{8}$$

For example, the axial distance between 2 and 3 in T_1 is 1.

If $(k(k+1))$ is the permutation in S_N that interchanges the positions k and $k+1$ and if T is a standard table associated with a partition of N, then $(k(k+1))T$ denotes the table resulting from interchanging k and $k+1$ in T.

Definition 1. The matrix representation of (k(k+1)) in the seminormal Young form associated with the partition μ is indexed by the standard tables and the ij-th entry, which we denote by $\alpha_{ij}[k,k+1]$, is given by

a) $\alpha_{ii}[k,k+1] = \begin{cases} 1 & \text{If } k \text{ and } k+1 \text{ are in the same row of } T_i; \\ -1 & \text{if } k \text{ and } k+1 \text{ are in the same column of } T_i. \end{cases}$

b) If k and $k+1$ are neither in the same column nor in the same row of T_i, then

$$\begin{pmatrix} \alpha_{ii}[k,k+1] & \dots & \alpha_{ij}[k,k+1] \\ \vdots & & \vdots \\ \alpha_{ji}[k,k+1] & \dots & \alpha_{jj}[k,k+1] \end{pmatrix} = \begin{pmatrix} -d^i(k,k+1)^{-1} & \dots & 1 - d^i(k,k+1)^{-2} \\ \vdots & & \vdots \\ 1 & \dots & d^i(k,k+1) \end{pmatrix}$$

where $j > i$ is such that $(k(k+1))T_j = T_i$.

c) 0 in any other case.

In order to illustrate the above definitions and procedures let us compute $[\alpha_{ij}(23)]$. In T_1, 2 and 3 are in the same column. Hence $\alpha_{11}(2,3) = -1$. In T_2, 2 and 3 are neither in the same column nor the same row, and so $d^2(2,3) = -2$ and $(23)T_3 = T_2$. The calculations for T_4 and T_5 are the same. Hence,

$$\alpha_\mu(23) = \begin{pmatrix} -1 & 0 & 0 & 0 & 0 \\ 0 & 1/2 & 3/4 & 0 & 0 \\ 0 & 1 & -1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 3/4 \\ 0 & 0 & 0 & 1 & -1/2 \end{pmatrix}$$

The following result, known as branching theorem, is useful in computing representations.

Theorem 1. Let ρ be an irreductible representation of S_n associated with the partition μ of n. Then ρ restricted to S_{n-1} splits into the direct sum of irreducible representations associated with the partitions μ' of $n-1$ which can be obtained by deleting a single node from the diagram of μ in all permissible ways.

More information concerning representations of symmetric groups can be found in James [7].

In what follows, we give a more precise description of the proposed algortihm. We consider the computation of an FFT on S_4 starting from the representations of S_3.

Initial data:

1. Partitions of 3: (3), (2,1) and (1,1,1).
2. Standard tables associated to these partitions:

$$(3) \longrightarrow \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline \end{array}$$

$$(2,1) \longrightarrow \begin{array}{|c|c|}\hline 1 & 3 \\ \hline 2 \\ \cline{1-1} \end{array} \quad \text{and} \quad \begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 \\ \cline{1-1} \end{array}$$

$$(1,1,1) \longrightarrow \begin{array}{|c|}\hline 1 \\ \hline 2 \\ \hline 3 \\ \hline \end{array}$$

3. Matrix representations of S_3:

$$\begin{array}{c} \\ (3) \\ (2,1) \\ \\ (1,1,1) \end{array} \begin{array}{c} \begin{array}{ccc} (23) & (12) & 1 \end{array} \\ \begin{pmatrix} 1 & & \\ & \begin{pmatrix} 1/2 & 3/4 \\ 1 & -1/2 \end{pmatrix} & \\ & & -1 \end{pmatrix} \quad \begin{pmatrix} 1 & & \\ & \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix} & \\ & & -1 \end{pmatrix} \quad \begin{pmatrix} 1 & & \\ & \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \\ & & 1 \end{pmatrix} \end{array}$$

4. Pre-computed transforms on S_3: Since the index of S_3 in S_4 is 4, there are four transforms, namely $\hat{f}_1$, $\hat{f}_2$, $\hat{f}_3$ and $\hat{f}_4$. Their values are stored in the array:

$$\begin{array}{c} \\ \rho_3 \\ \rho_{111} \end{array} \begin{array}{c} \begin{array}{cccc} \hat{f}_1 & \hat{f}_2 & \hat{f}_3 & \hat{f}_4 \end{array} \\ \begin{pmatrix} \beta_{(3,1)} & \beta_{(3,2)} & \beta_{(3,3)} & \beta_{(3,4)} \\ \beta_{(111,1)} & \beta_{(111,2)} & \beta_{(111,3)} & \beta_{(111,4)} \end{pmatrix} \end{array}$$

where $\beta_{(\mu,i)} = \sum_{s \in S_3} f_i(s)\rho_\mu(s)$.

Procedure:

We first compute the partitions of 4. Then, for each partition, we compute the standard tables and matrix representation through the branching theorem and definition 1. For example, if $\mu = (31)$ then, because of the branching theorem

$$\rho_{(31)} \downarrow S_3 = \rho_{(21)} \oplus \rho_{(3)}. \tag{10}$$

The standard tables associted to (31) are obtained from the standard tables of (21) and (3) by adding a node with value 4 in the appropriated position.

Computing the matrix representation associted to (31) involves two steps:

Step 1. Compute $\rho_\mu(34)$ through definition 1.

Step 2. Compute $\rho_\mu(k(k+1))$, $k = 1, 2$ through equation (10).

The matrix representations of the elements $(i, 4) \in S_4$, $i = 1, 2$ is obtained from the above computed matrices through conjugation. In particular, $(24) = (23)(34)(23)$ and $(14) = (12)(24)(12)$.

Now, we use equation (7) to compute $\widehat{f}(\rho_\mu)$, this amounts to computing

$$\widehat{f}(\rho_\mu) = \sum_{i=1}^{4} \rho_\mu(s_i) \begin{pmatrix} \hat{f}_i(\rho_{(21)} & \\ & \hat{f}_i(\rho_{(3)}) \end{pmatrix}$$

with $s_1 = 1$, $s_2 = (14)$, $s_3 = (24)$ and $s_4 = (34)$.

3. A SISAL code. The procedure for computing the Fourier transform of a real valued function f over all irreducible representations of S_N for $N > 3$ can be summarized as follows:

(1) Calculate the transforms of f restricted to S_3 by means of the functions *res1*(f),*res2*(f) and *res3*(f). The result of each of these functions is an array of real matrices.

(2) Associate a record with each partition of S_3. The fields of each record contain the partition associated with the representation, the standard tables, the representation matrices for the elements (2 3), (1 2) and 1, and the array of restrictions obtained in (1).

(3) Using the function *sninfo*(N, f), make use of Definition 1, Theorem 1, and Equations (5) and (6) along with the information concerning S_{m-1}, to calculate for each m, $3 < m \leq N$ and each partition of m, the transform of f restricted to S_m. When execution of the loop with control variable m terminates (i.e., when $m = N$), the *restric* field of each partition record will contain the Fourier transform of f on the representation associated with the partition.

The above algorithm differs from those of Diaconis and Rockmore [5] in that it does not assume that the representations of S_N have already been computed, but rather computes them iteratively beginning with representations of S_3.

All of the functions of the Sisal program, too lengthy to give here, have been constructed in a very natural way from the given definitions and theorems. Especially useful features of Sisal in expressing the algorithm are ragged arrays, to represent the standard tables, and dynamic sized arrays, to represent collections of tables, partitions, and representations.

Although the use of aggregate structures has greatly enhanced the clarity of and ease of coding a very complex algorithm (the Sisal code consists of 525 lines), it no doubt undermines efficiency. Work is currently under way to optimize the code for implementation on a Cray.

Acknowledgements

The authors are grateful to Professor Dorothy Bollman for her helpful comments and to the Lawrence Livermore Computing Research group.

References

[1] Aho, A., Hopcroft, J. and Ullman, J. (1976), *The design and analysis of computer algorithm.* Addison-Wesley, Readings, Mass.

[2] Bluenstein, L. (1970), *A linear filtering approach to the computation of discrete Fourier transform.* IEEE Transactions on Audio and Electroacoustics, Vol. AU-18,4, 451-455.

[3] Clausen, M. (1989), *Fast Fourier transforms for metabelian groups.* Siam J. Comput. 18, 584-593.

[4] Cooley, J. and Tukey, J. (1965), *An algorithm for the machine calculation of complex Fourier series.* Math. Comp. 19, 297-301.

[5] Diaconis, P. and Rockmore, D. (1990), *Efficient computation of the Fourier transform on finite groups.* Journal of the American Mathematical Society. 3, 297-332.

[6] Good, I. (1958), *The interaction algorithm and practical Fourier analysis.* J.Royal Statistic Soc. 20, 361-372.

[7] James, G.D. ((1978),*The representation theory of the symmetric groups.* Lecture Notes in Math., vol. 682, Springer-Verlag, Berlin.

[8] Korn, D. (1979), *Computing the fast Fourier transform on a vector computer.* Math. Comp. 33, 977-992.

[9] Pease, M. (1968), *An adaptation of the fast Fourier transform for parallel processing.* J.ACM. 15, 252-265.

[10] Rader, C.M. (1968),*Discrete Fourier transform when the number of data points is prime.* Proc. IEEE 56, 1107-1108.

[11] Rockmore, D. (1990), *Fast Fourier analysis for abelian groups extentions.* Advances in Applied Mathematics. 11, 164-204.

[12] Serre, J.P. (1977), *Linear representations of finite groups.* Springer-Verlag, New York.

[13] Winograd, S. (1978), *On computing the discrete Fourier transform.* Math. Comp. 32, 175-199.

FIVE WAYS TO FILL YOUR KNAPSACK

Wim Böhm, Colorado State University, USA
Greg Egan, Swinburne University of Technology, Australia

October 25, 1992

ABSTRACT

We compare five solutions to the zero-one knapsack problem: *Divide and Conquer, Depth First with Bound, Dynamic Programming, Memo Functions, and Branch and Bound.* Our programs are written in Sisal and run on the CSIRAC II dataflow machine. Two of the algorithms, Memo Functions and Branch and Bound, benefit from non deterministic extensions of Sisal, *put* and *get.* We introduce these extensions and compare the performance of the five algorithms using knapsacks of 20 and 40 objects. We measure the performance of our programs in S_1: the number of instructions executed, S_∞: the critical path length, and $\pi = \lfloor S_1/S_\infty \rfloor$: the average parallelism. It turns out that the Branch and Bound algorithm performs best in terms of S_1, especially for the harder test cases.

Just blow up the stack Jack
Make a bad call Paul
Hit the wrong key Lee
Set your pointers free

Just mess up the bus Gus
Don't need to recurse much
Just listen to me

....

Kind of by Paul Simon
Courtesy of the net

1 INTRODUCTION

The zero-one knapsack problem is defined as follows. Given n objects with positive weights W_i and positive profits P_i, and a knapsack capacity M, determine a subset of the objects represented by a bit vector X with elements x_1 to x_n, such that

$$\sum_{i=1}^{n} x_i W_i \leq M \quad and \quad \sum_{i=1}^{n} x_i P_i \quad maximal$$

We assume the objects to be sorted by profit weight ratio, as solutions are often close to the *greedy approximation*: grab objects with a maximal profit weight ratio until the knapsack cannot be filled any further.

The knapsack problem gives rise to a search space of 2^n combinations of objects, which can be depicted as a binary tree, where the root represents an empty knapsack, and going from $level_i$ in the tree to $level_{i+1}$ represents either picking $object_i$ (going left down) or not picking $object_i$ (going right down). Given a partial solution (a choice for objects 1 .. i), a lower bound for the best total solution can be computed in linear time by adding objects with maximal profit weight ratio (i.e. objects i+1, i+2, ...) until an object exceeds the knapsack capacity, while an upper bound can be computed by adding part of the object that exceeded the knapsack capacity, such that the knapsack is filled to capacity.

In this paper we compare five solutions to the zero-one knapsack problem: *Divide and Conquer, Depth First with Bound, Dynamic Programming, Memo Functions,* and *Branch and Bound,* written in Sisal and run on the CSIRAC II dataflow machine. Two of these algorithms, Memo Functions and Branch and Bound, need non deterministic extensions of Sisal, *put* and *get.* We introduce these extensions. We compare the performance of the five algorithms using knapsacks of up to 40 objects. We measure the performance of our programs in S_1: the number of instructions executed, S_∞: the critical path length, and $\pi = \lfloor S_1/S_\infty \rfloor$: the average parallelism.

1.1 THE CSIRAC II DATAFLOW MACHINE

The CSIRAC II dataflow computer [1], used in this study, is characterised by random allocation of workload at the node level as distinct from a code block or procedure level; generic node functions; strongly typed, variable length tokens; loop unravelling as well as re-entrant code support using a single undifferentiated colour tag combined with the ability to preserve temporal ordering of tokens without tag manipulation overheads, tokens on any given arc with the same colour being maintained in strict FIFO order; imbedded storage functions for local state information; heterogeneous streams; integrated input/output and error mechanisms. More recent refinements to the architecture have included the addition of vector and compound token types and extensions to matching functions for streams.

2 THE ALGORITHMS

In the following programs W denotes the array of weights, P denotes the array of profits, M the knapsack capacity, n the number of objects, i the level in the search tree, and cp the profit gathered at a particular point in the search tree.

2.1 DIVIDE AND CONQUER

The Divide and Conquer solution to the knapsack problem is better seen as an executable specification. Apart from checking whether the weight of an object exceeds the remaining capacity of the knapsack, the Divide and Conquer algorithm does not prune the search space. As the left-down and right-down searches are independent, this algorithm is highly parallel. But, as is often the case when there is abundant parallelism, a lot of unnecessary work is performed.

```
function knapdc(W,P: array[integer]; i,M,n: integer  returns integer)
 if M < W[i] then
  if i<n then knap(W,P,i+1,M,n) else 0 end if
 else if i<n then
       let l := knapdc(W,P,i+1,M-W[i],n)+P[i];
           r := knapdc(W,P,i+1,M,n)
        in if l > r then l else r end if
       end let
      else P[i] end if
 end if
end function
```

The main function initializes W, P, M and n and calls $knapdc(W, P, 1, M, n)$.

2.2 DEPTH FIRST WITH BOUND

The Depth First with Bound solution computes, in a certain point of the search space, the upper bound given the partial solution, and if this upper bound is less than the best solution found so far, the sub-tree under the partial solution is not further explored. This avoids large amounts of work, but causes the search to proceed depth first left to right, and consequently looses almost all parallelism in the algorithm. It also forces the search to go down the "greedy" path, which may not always be favourable. The Branch and Bound algorithm in section 2.6 deals with these problems. Note that, when going left down (taking $object_i$), the upperbound does not need to be recomputed as it does not change.

```
forward function knapb(W,P: array[integer]; i,cp,M,n,best: integer; returns integer)

function knap(W,P: array[integer]; i,cp,M,n: integer; returns integer)
if (M<W[i]) then
     if i<n then knapb(W,P,i+1,cp,M,n,cp) else cp end if
else if i<n then
      let l := knap(W,P,i+1,cp+P[i],M-W[i],n);
          r := knapb(W,P,i+1,cp,M,n,l)
       in max(l,r)
      end let
     else cp+P[i]
     end if
end if
end function

function knapb(W,P: array[integer]; i,cp,M,n,best: integer; returns integer)
 let bound :=
  for initial
   b := cp; cm := M; j := i
  repeat
  b,cm,j :=
   if (old cm >= W[old j])
   then old b + P[old j], old cm - W[old j], old j + 1
   else old b + (old cm * P[old j])/W[old j],0,n+1
   end if
  until j > n
  returns value of b
  end for
  in if bound <= best then best else knap(W,P,i,cp,M,n) end if
 end let
end function
```

The main function initializes W, P, M and n and calls $knap(W, P, 1, 0, M, n)$.

2.3 DYNAMIC PROGRAMMING

The dynamic programming solution to the knapsack problem combines solutions of sub-problems bottom-up, saving answers to sub-problems in a vector V_i. At $stage_i$ in the computation, V_i contains solutions to problems with knapsack capacity 0 to M using objects 1 to i only. An element of V_i can be expressed in terms of elements of vector V_{i-1}:

$$V_i[j] = max(V_{i-1}[j], P[i] + V_{i-1}[j - W[i]])$$

The term $V_{i-1}[j]$ represents the choice of not taking $object_i$, the term $P[i]+V_{i-1}[j-W[i]]$ represents the choice of picking $object_i$.

```
function knapdp (W,P: array[integer]; M,n: Integer returns integer)
let FinalV :=
 for initial
  i := 0;  V := array_fill (0,M,0);
 repeat
  i := old i + 1; Pi := P[i]; Wi := W[i];
  V := for v1 in old V at j
          nv := if j >= Wi
                then let v2 := old V[j-Wi] + Pi in max(v1,v2) end let
                else v1
                end if
          returns array of nv
         end for
 until i = n
 returns value of V
 end for
in FinalV[M]
end let
end function
```

The main function initializes W, P, M and n and calls $knapdp(W, P, M, n)$. The algorithm computes $M * n$ values, each value takes constant time to compute, so knapdp has an S_1 complexity of $O(M * n)$. Also, this is the only algorithm with potential for vectorization.

2.4 TAGGED MEMORY, LOCKS, and NON DETERMINISM

In a dataflow machine, asynchronous structure accessing is implemented using split-phase read and write operations and storage cells augmented with two *tag* bits: a *presence bit* P and a *defer bit* D.

```
READ ( cell: storage-cell returns number):
   if cell.P
   then return cell.VAL
   else cell.D := True;
        enqueue the READ request using cell.VAL as a pointer
   end if

WRITE ( cell: storage-cell, val: number):
   if cell.P then ERROR
   else if CELL.D
        then honour ALL requests in the defer queue
        end if;
   cell.P := True;
   cell.VAL := val
   end if;
```

Until now we have been able to express our algorithms in standard Sisal. The implementations of the Memo Functions and Branch and Bound algorithms require non determinism and can therefore not be expressed in pure Sisal. We will use the *side effecting* operations *put* and *get* for this. The combination of *put* and *get* provides for light weight locks, using the presence and defer bits of tagged memory. *Get* reads a value from a storage cell and resets the presence bit, in other words, it reads and wipes out a value from storage. *Put* writes a value in a storage cell, in such a way that only one *get* can "get" it. If there are no deferred accesses, put just performs a write. If there are deferred accesses, put honours *one* request, leaves the cell empty and the rest of the accesses deferred.

```
GET ( cell: storage-cell returns number):
   if cell.P
   then cell.P := False; return cell.VAL
   else cell.D := True;
        enqueue the GET request using cell.VAL as a pointer
   end if;

PUT ( cell: storage-cell, val: number ):
   if cell.P then ERROR
   else if cell.D
        then honour ONE request in the defer queue
        else cell.P := True; cell.VAL := val
   end if;
```

The *get* and *put* functions are supported directly by the structure-read-and-reset (srr) and the structure store-write-read-once (srw) instructions of the CSIRAC II. These instructions have been used for some time in the runtime resource management library for CSIRAC II [1]. The following is the implementation of *put* and *get* in the intermediate code i2 [2] used in the Sisal to CSIRAC II compiler.

```
define __get(index)->value;
begin
  srr(index) -> value;
end;

define __put(index, value)->acknowledge;
begin
  srw(value index) -> pip_gate;
  pip(value pip_gate) -> acknowledge;
end;
```

where pip stands for "pass if present".

2.5 MEMO FUNCTIONS

The memo function solution to knapsack combines the divide-and-conquer and dynamic programming methods. A table is maintained containing all sub-solutions. The table elements are initialized to -1 to indicate that computation of the solution to the particular sub-problem has not been started.

```
function knapm(Pad,W,P: array[integer]; i,M,n: integer returns integer)
let Pos:=M*n+i; PP := get(Pad,Pos);
    BP := if OldP ~= -1 then PP
          else if M < W[i] then
                if i<n then knapm(Pad,W,P,i+1,M,n) else 0 end if
               else if i<n then
                     let l := knapm(Pad,W,P,i+1,M-W[i],n)+P[i];
                         r := knapm(Pad,W,P,i+1,M,n)
                     in max(l,r)
                     end let
                    else P[i]
                    end if
               end if
          end if
 in put(Pad,Pos,BP)
end let
end function
```

The main function creates a table *Pad* containing $(M+1)*n$ elements initialized to -1. The semantics of *put* and *get* ensure that only one process at the time will get the value of a certain table element. If it is -1 the process will compute the solution to the particular sub-problem and put the solution back. If the element is not -1, it has been computed, so the process puts it back in the table. Other processes needing this solution will be deferred until the solution is put back. As in the dynamic programming algorithm, the total amount of work is $O(M*n)$. $O(M*n)$ table

elements are computed. As the number of non deferred processes going down from $level_i$ to $level_{i+1}$ is at most M, at most $O(M * n)$ processes can get deferred.

2.6 BRANCH AND BOUND

The Branch and Bound algorithm exploits parallelism to implement branching, which means that the state space is searched breadth first. This avoids the drawbacks of the depth first with bound algorithm. Notice the absence of explicit queueing in the algorithm. Sub-trees are cut by estimating the upperbound of a partial solution and comparing it to a *shared variable GLow* containing the current best lower bound, maintained with *puts* and *gets*, ensuring that only one process can get GLow, use it and write an updated value back.

```
function knapbb (GLow, W, P: Vector; i, cp, M, n: integer returns integer)
 if i > n | M=0  then cp
 else
   let L, U :=
         for initial % compute lower and upper bound
            cl := cp;  cu := cp;  cm := M; j  := i
         repeat
             cl, cu, cm, j :=
                 if old cm >= W[old j]
                 then old cl + P[old j],  old cu + P[old j],
                      old cm - W[old j],  old j + 1
                 else old cl, old cu + ((old cm * P[old j]) / W[old j]),
                      old cm, n+1
                 end if
         until j > n
         returns value of cl value of cu
         end for;
     GL := get(GLow,1);
     GB := put(GLow,1,max(GL,L));
    in
     if U < GB
     then 0
     else if M >= W[i]
         then let l := knapbb(GLow, W, P, i+1, cp+P[i], M-W[i], n);
                  r := knapbb(GLow, W, P, i+1, cp, M, n)
              in if l > r then l
                          else r end if
              end let
         else knap (GLow, W, P, i+1, cp, M, n)
         end if
     end if
   end let
 end if
end function
```

2.7 Make that six: FUNCTIONAL BRANCH AND BOUND

We can make the above Branch and Bound algorithm functional by going down the tree level by level using a forall construct, creating a set of "viable tasks" for the next level down, using the same lower and upperbound computation, but comparing this not to a global shared variable, but to the best solution found in the previous level. A task is represented by two integers: a *current profit* and a *capacity left*, and a next level in the tree is therefore represented by two arrays of integers.

```
type Vector = array[integer];
function bstep( W, P, profits, capacities: Vector; i,n, best: integer
                returns integer,vector,vector)
  for pr in profits dot m in capacities
   lwb, prfs, caps :=
      if i > n   then  best, array vector [], array vector []
      else let L, U :=
                for initial              % Greedy algorithm
                 cl := pr;  cu := pr;  cm := m; j  := i
                repeat  cl, cu, cm, j :=
                 if old cm >= W[old j]
                 then old cl + P[old j], old cu + P[old j], old cm - W[old j], old j + 1
                 else old cl, old cu + ((old cm * P[old j]) / W[old j]), old cm, n+1
                 end if
                until j > n
                returns value of cl value of cu
                end for
           in if U < best then  best, array vector [], array vector []
                else   if m >= W[i]
                       then L, array vector[1: pr+P[i],pr], array[1: m-W[i],m]
                       else L, array vector[1: pr], array[1: m]
                end if end if
           end let
      end if
   returns value of greatest lwb    value of catenate prfs   value of catenate caps
  end for
end function

function main (returns integer)
let n := ... ; M := ... ;   W := array[1: ... ];   P := array[1: ... ];
    InitProfit := Array vector[1:0];    InitCap := Array vector[1:M];
in for initial   profits := InitProfit;  capacities := InitCap; i := 1 ; best := 0
   while i <= n repeat
    best, profits, capacities :=  bstep(W,P,old profits,old capacities,old i,n, old best);
    i := old i + 1
   returns value of best
   end for
end let
end function  % main
```

$n = 20$			S_1				
Off	Var	M	BB	FBB	DP	MF	DF
20	20	499	36325	47703	188329	485926	*15089*
20	16	432	212014	278080	163284	417389	*38286*
20	12	407	178552	225455	153944	378371	*21781*
20	8	384	142188	177388	145349	358291	*103834*
20	4	344	113578	141805	130394	194516	*48186*
40	20	819	87822	107098	307929	651997	*11731*
40	16	752	28089	37673	273544	584245	*10027*
40	12	727	28161	38007	273544	493647	*9553*
40	8	704	47822	64345	264949	384940	*10358*
40	4	64	55459	72243	264949	212993	*10557*

Table 1: S_1 for $n = 20$

$n = 20$			S_∞					π				
Off	Var	M	BB	FBB	DP	MF	DF	BB	FBB	DP	MF	DF
20	20	499	3412	4805	4053	*823*	1585	10	10	46	*590*	9
20	16	432	5458	15194	3517	*819*	5006	38	18	46	*509*	7
20	12	407	4123	10577	3328	*819*	2683	43	21	46	*462*	8
20	8	384	4132	9675	3154	*819*	11884	34	18	46	*437*	8
20	4	344	3900	8294	2845	*821*	5715	29	17	45	*236*	8
40	20	819	3521	5636	6528	*815*	992	25	19	47	*800*	12
40	16	752	3192	4117	6001	*811*	695	9	9	47	*720*	14
40	12	727	3320	4117	5800	*811*	470	8	9	47	*609*	20
40	8	704	3473	5317	5619	*807*	655	14	12	47	*477*	16
40	4	64	3444	5377	5305	*821*	661	16	13	47	*259*	16

Table 2: S_∞ and π for $n = 20$

3 EVALUATION

The knapsack problem instances are created by a C program with the following input parameters:

```
N   - number of candidate items
P   - capacity of knapsack as percentage of total weights
Off - minimal weight and profit of an object
Var - variance in weight and profit of an object
S   - random number seed
```

Given these parameters, the program creates N objects, with weights and profits randomly varying between *Off* and *Off+Var*. The arrays are sorted accordingh to highest profit to weight ratio. The *Off* and *Par* parameters allow to vary the discrepancy between the objects with the

$n = 40$			S_1		S_∞		π	
Off	Var	M	BB	DF	BB	DF	BB	DF
40	20	1585	85658	*35859*	10076	*2009*	9	*18*
40	16	1517	*1086312*	15705956	*22142*	2505189	*49*	6
40	12	1440	*987012*	1221636	*22561*	223835	*44*	5
40	8	1391	1223943	*435916*	*21814*	80978	*56*	5
40	4	1328	*5528756*	25377336	*74800*	3663355	*74*	7

Table 3: S_1, S_∞ and π for $n = 40$

highest and lowest profit weight ratio. With a small *Off* parameter and a large *Var* parameter it is possible to have a knapsack with "diamonds" and "bricks" at the same time.

We have run our programs for several knapsacks with 20 and 40 objects. The capacity of the knapsacks is always 80% of the total weight of the objects. Even for n = 20, the divide and conquer algorithm is unbearably inefficient, so we will not include its results in our tables. In the tables BB stands for Branch and Bound, FBB for functional Branch and Bound, DP for Dynamic Programming, MF for Memo Functions, and DF for Depth First with Bound. The winning value in a certain category is emphasized. For n = 20 we have used five knapsacks with an *Off* parameter of 20, and five with an *Off* parameter of 40 in table 1 and table 2, varying *Var* from 20 down to 4 with steps of 4. FBB is less efficient than BB for two reasons: the bound in FBB is not as good as in BB because it only takes points in the search space on a previous level into account, and FBB uses an expensive reduction operator: *value of catenate*. The case of *Off* = 40 shows the dependence on capacity in the case of the Dynamic Programming and Memo Functions algorithms.

Notice that, in the case of n = 20, the Depth First with Bound algorithm performs well in terms of total work, and that the Memo Functions algorithm exposes the most parallelism. For n = 40, the Functional Branch and Bound, Dynamic Programming and Memo Functions algorithms execute too many instructions to finish in a reasonable amount of time. The only algorithms that are efficient enough in terms of S_1 are Branch and Bound and Depth First with Bound. Table 3 shows the results for n = 40.

The trend seems to be that the harder the problem becomes, the better the Branch and Bound performs in terms of S_1, even though it does not show too much parallelism.

4 CONCLUSION

We have studied a number of algorithms solving the zero-one knapsack problem. These algorithms are written in Sisal and run on the CSIRAC II dataflow machine. Two of these algorithms use non deterministic extensions *put* and *get*, which allow for locking and updating of storage cells. One of these, the Branch and Bound algorithm, performs the best in terms of S_1, for a number of our test

cases. Also, the most parallel algorithm, the divide and conquer algorithm performs the worst in terms of S_1.

References

[1] Egan, G.K., N.J. Webb and A.P.W. Böhm , 'Some Features of the CSIRAC II Dataflow Machine Architecture', in Advanced Topics in Data-Flow Computing, Prentice-Hall 1991, pp143-173.

[2] Egan, G.K., Rawling, M.J. and Webb, N.J., 'i2: An Intermediate Language for the CSIRAC II Data Flow Computer', Technical Report 31-002, Laboratory for Concurrent Computing Systems, School of Electrical Engineering, Swinburne Institute of Technology,1990

Simulating Material Dislocation Motion in Sisal

Michael T. Strailey, Jr and Patrick Tibbits
Indiana Institute of Technoloey

Thomas DeBoni
Lawrence Livermore National Laboratory

1. Introduction

The goal of this project is to develop a computer model in Sisal to simulate the movement of dislocations in metal-class materials undergoing high temperature deformation. The project expands previous work in simulating the movement of dislocations [1]. The new model will be able to handle more dislocations and eventually will model curved dislocation lines in three-dimensional space. It is hoped that this new model will be able to elucidate the mechanism by which Persistent Slip Bands (PSB's) form. The decision to implement the model in Sisal was influenced by the Sisal Scientific Computing Initiative's offer of free supercomputer time.

The model tracks the motion of the dislocations by solving a system of coupled ordinary differential equations. The heart of our simulation then is the Runge-Kutta-Fehlberg integrator used to solve this system of equations. This process is expected to parallelize well because identical computations are used for each of the dislocations.

With the integrator recently completed and only preliminary results available, the main focus of this paper will be on our experiences with using Sisal to implement a preliminary project, the simulation of dislocation pileup formation. This simulation will be used to verify our results against previous simulations and the analytical results.

2. Theory

Figure 1 shows our simulation of pileup formation, in which dislocations emitted by a Frank-Reed source on the left are obstructed by a sessile dislocation on the right.

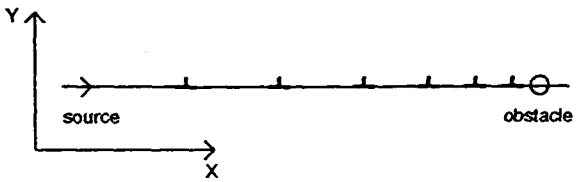


Figure 1. The pileup problem.

As outlined in [2], the glide velocity of some dislocation j in a pileup of N dislocations is obtained by summing the applied stress, the stresses exerted at j by the obstacle and by all the other dislocations in the pileup, then multiplying the resultant stress by a glide mobility Mg. Defining the x-axis parallel to the slip plane of the pileup the equations of motion of the dislocations are

$$\dot{x}_j = M_g\left[\sum_{\substack{i=1 \\ i\neq j}}^{N}\frac{(x_j-x_i)\left[(x_j-x_i)^2-(y_j-y_i)^2\right]}{\left[(x_j-x_i)^2+(y_j-y_i)^2\right]^2}+(\tau_{app}+\tau_{obs})\right]$$

$$\dot{y}_j = M_c\left[\sum_{\substack{i=1 \\ i\neq j}}^{N}\frac{-(y_j-y_i)\left[3(x_j-x_i)^2+(y_j-y_i)^2\right]}{\left[(x_j-x_i)^2+(y_j-y_i)^2\right]^2}+\tau_{obs}\right]$$

This simulation disallows climb, obviating equations of motion for the y-coordinates. In the above equation, Mg and Mc represent the glide and climb mobilities, respectively. The summation in the equation of motion gives stress in units $\mu/2\pi(1-\nu)$ where μ is the material's shear modulus and ν is Poisson's ratio. The dislocation coordinates are in terms of b, the Burgers vector.

Below is an outline of the basic algorithm used in the preliminary pileup simulation:

```
Loop

   Check Source Operation

   Call RKF to calculate new
   positions

      Call RHS to solve right-hand
      side of the ODEs

   Check Sink Operation

Until (#_DNs_Sunk > Scuttle) or
      (count > stop)
```

In the main loop we check to see if the stress on the source is positive, causing a new dislocation to be emitted. Then we solve the differential equations, by integrating the velocities, to calculate the new positions. Next, we remove any dislocations which have reached the sink. This entire process is repeated until a certain number of dislocations have reached the sink or a predetermined number of iterations has been completed. It should be noted that for the pileup problem no dislocations will be able to reach the sink because $Mc = 0$.

3. Analysis of Sisal

3.1 How Sisal differs from traditional languages

While members of our team had extensive experience with other programming languages, Sisal was our first encounter with an *applicative* programming language. The central idea is function application rather than the assignment statement [3]. The applicative nature of Sisal require a slight shift in the thinking process from that used for traditional imperative programming languages. Some of the traditional practices and concepts no longer apply in Sisal. For instance, functions may not access *global* variables, everything must be passed in through the function's parameter list. Since the main operation is the application of a function on its parameter list all variables must go through the parameter list. Of course, this is considered to be a good programming practice anyway, because it eliminates the chance for *side-effects*. This correlates with the mathematical definition of a function since a mathematical function does not rely on any hidden values and everything the function needs is passed in to it. In Sisal one must also abandon the traditional way of thinking about variables as *memory locations* and start thinking of them as *values*. Again, we see that this takes us back to the true mathematical sense of the term variable, only with the advent of computers that we have equated the term variable with a storage location. As we now see this shift in thinking is only required because our experience with traditional languages has caused us to stray away from true mathematical conventions.

3.2 Learning Sisal

Obviously, it was necessary for members of our team to learn Sisal. Except for the differences mentioned above, we found that our previous programming experience helped to speed up the learning process. Sisal has a very Pascal-like syntax, so those familiar with Pascal will have an even easier time learning Sisal. Our experience was that a person could master the basics in about a week or two. After a month a person could write programs using the full power of Sisal. Recent efforts by the developers at LLNL to expand the introductory documentation should further speed the learning process. In short, we found Sisal to be no more difficult to learn than the typical imperative language. When one considers that one is now writing programs which are inherently parallel, the ease of learning is quite impressive.

3.3 Sisal and Parallelism

As mentioned above, Sisal programs are capable of running in parallel. In fact, the compiler assumes that everything will be done in parallel, unless otherwise specified. Because our final model will require extensive computations, we are, of course, interested in exploiting the inherently parallel nature of the problem. Sisal allows our team which has no previous experience with parallel computing to easily write parallel code. In a sense the parallelism is free because it would require just as much or more effort to write our application in a traditional language.

What about other automatic parallelizing compilers? Sisal's applicative (almost functional) nature guarantees that there will be no side-effects. Non-functional automatic parallelizing compilers must be more conservative when searching for operations to be done in parallel.

With Sisal we do not have to concern ourselves with any of the details necessary to implement the parallelism. We do not have to worry about enforcing mutual exclusion through the use of critical sections, monitors, or semaphores. Also, we do not have to concern ourselves with synchronization since all of the details are taken care of by the compiler. By freeing us from these implementation concerns, Sisal allows us to concentrate on the details of our application.

3.4 Sisal is Portable

While many languages make this claim, it has been our experience that Sisal really lives up to this claim. We had a chance to test this when in the middle of our project the SGI machine we were using developed operating system problems, forcing us to move to a Sun for awhile. All the Sisal code required to run was recompilation. Unfortunately, the same could not be said about our post processors written in C and FORTRAN. Even though our target platform is the Cray, Sisal's portability allows us to do all of our development and debugging on local workstations.

3.5 Performance

There is a popular misconception that all functional (and applicative) programming languages have poor performance. Statements like this are quite common, "However, the execution speed of functional programs on sequential hardware is typically several orders of magnitude slower than conventional programs . . . " [4]. Studies have shown that Sisal is capable of producing code with similar performance to that of code written in FORTRAN [5].

3.6 Problems we encountered with Sisal

Along the way we have encountered a few minor problems or limitations with Sisal. First, the lack of high order functions in the current language specification made for very a cluttered parameter list for our Runge-Kutta-Fehlberg solver. The solver must call another routine to solve the right-hand side (RHS) of the differential equations. Since this routine is called from within the solver, any constants or other information needed by the RHS routine must be passed through the solver. With high order functions one could just pass the function to solve the RHS into the solver, making for a cleaner parameter list and more general purpose solver because the name of the RHS function could be changed.

The other minor criticism is Sisal's use of the FIBRE format for *all* data input and output. Since the data generated by our model was to be post-processed for graphical display, we had to first process the data out of FIBRE format and into a more suitable format for input into a graphics post-processor program. For many applications FIBRE is fine, but we would like the ability alter the format if necessary.

3.7 TWINE Debugger

For the debugging process we found the TWINE debugger to be very useful. With Sisal the debugger plays a bigger role because one cannot just insert print statements to output intermediate results. The interface for TWINE is a little crude by today's standards, but it has enough functionality to make it a valuable tool for debugging.

3.8 Code Examples

We found some of Sisal's unique loop reductions to be very useful. The following section of code, from our RHS function, demonstrates an instance where we found the *unless* filter to be useful.

```
dy := for i in 1, Ne
      TSigma_xy:= for j in 1, Ne
                  returns value of sum
                  sigma_xy(X[i] - X[j],
                           Y[i] - Y[j])
                  unless (i = j)
      end for;
  TSigma_xy1 := TSigma_xy + sigma_xy(
                X[i] - Xo, Y[i] - Yo);
  Ydot := TSigma_xy1 * Mc
  returns array of Ydot
end for;
```

Here we do not want to calculate the stress that a dislocation exerts on itself because it would lead to a division by zero error. This could have been done without the use of the *unless* filter, but with the filter it is very clear that we are omitting the case where i = j. It should be noted that the use of the filter sequentializes the inner loop; therefore, one would not want to use it for a loop that one wishes to execute in parallel. So, this clarity is gained at the expense of speed.

3.9 Tips for programming in Sisal

We agree with the language's developers that the easiest way to write a Sisal program is to start from the mathematical foundations of the problem. As outlined above in Section 3.1, Sisal has close correspondence to mathematical equations. If one wants to rewrite an existing application into Sisal we recommend going back to first principles,

rather than trying to translate from the original language.

4 Conclusions

On the whole our experience with Sisal has been a positive one. Based on our experience, we would recommend to other researchers to consider writing the computational kernel of future projects in Sisal. Especially if the group has little experience writing parallel programs or wishes to be freed from implementation details. Its combination of parallelism, portability, and performance is hard to ignore.

References

1. P. Tibbits, Dissertation, University Microfilms (1988).
2. Hirth and Lothe, *Theory of Dislocations*, 2nd ed. Wiley, New York (1982).
3. B. J. MacLennan, *Principles of Programming Languages*, 2nd ed., p. 335. Saunders College Pub., Fort Worth, TX (1987).
4. I. Sommerville, *Software Engineering*, 4th ed. Addison-Wesley Pub. Co., Wokingham, Eng.
5. D. Cann, *Communications of the A.C.M.* **35 : 8**, pp. 81 - 89 (1992).

Candis as an Interface for SISAL

David J. Raymond
Physics Department and Geophysical Research Center
New Mexico Tech
Socorro, NM 87801
raymond@kestrel.nmt.edu

October 1, 1992

1 Introduction

Candis stands for "*C* language *an*alysis and *dis*play", and is a system developed by the author for analyzing and displaying gridded numerical data. It has been in regular use at New Mexico Tech for a number of years, and is being used at a number of other institutions as well. The basic idea of Candis is to apply the UNIX filter paradigm to non-ASCII, gridded numerical data. In order to make this possible, a standardized, self-describing data format has been adopted. Details can be found in Raymond (1988). The advantage of this system is that complex applications can be constructed by piping together a number of standard Candis utilities. Its main applications have been the analysis and display of data from meteorological observation systems and the analysis of numerical model output. Both of these applications typically have data in the form of rectangular grids of varying dimensions.

In this paper I describe an interface between Candis and SISAL (McGraw, et. al, 1985). For SISAL applications that produce gridded numerical data, this provides a simple way to take advantage of the strengths of both systems – computations are done using SISAL and the results are analyzed using Candis.

2 The Cdf Record in SISAL

The key to the interface between Candis and SISAL is the *cdf record* construct in SISAL. This is a SISAL record with the following structure:

```
type cdf_cline = array[character];

type cdf_clist = array[cdf_cline];

type cdf_param = record[pname: cdf_cline; pval: real];

type cdf_plist = array[cdf_param];

type cdf_dim = record[dname: cdf_cline; dsize: integer;
          ddata: array[real]];

type cdf_dlist = array[cdf_dim];

type cdf_field = record[fname: cdf_cline; num_dims: integer;
          cdf_dhandles: array[integer]; tsize: integer;
          fdata: array[real]];

type cdf_flist = array[cdf_field];

type cdf = record[name: cdf_cline; instance: integer;
          comments: cdf_clist; params: cdf_plist;
          dims: cdf_dlist; fields: cdf_flist];
```

The cdf record is isomorphic to a subset of a Candis file. Thus, it is possible to translate a cdf record into a Candis file with no loss of information.

The cdf record consists of six parts, each of which will now be described. The *name* is simply an ASCII identifying string which can be used to name the Candis file into which the cdf record is translated. Since multiple record instances with the same structure are sometimes useful (e. g., records of the same variables at different times), the *instance* entry can be used as a suffix of the *name* entry for file name generation.

The *comments* entry is a list of strings in which a history of the processing of the cdf record can be kept. This proves invaluable in data processing.

The *params* entry provides a way to store parameters, i. e., name-number pairs that are used to characterize the included data in some way.

The *dims* entry stores information about the vector space over which the gridded data are defined. Information included are the names of each dimension, the number of points defined for that dimension, and the positions of each point. Points need not be equally spaced, but they should be monotonic. Candis supports up to four dimensions.

The *fields* entry stores the gridded data itself. Each *field* is defined over zero or more of the defined dimensions. Information included is the name of each field, the dimensions over which the field is defined, and the data.

Fields are internally represented as one-dimensional arrays of reals, even though a field may be interpreted as having multiple dimensions. Thus, the multidimensional features of SISAL are not used, and dimensional indexing must be done by the user. Beginning arrays at zero as in the C language makes this a relatively simple chore. The best strategy for simulating multidimensional fields seems to be to loop over the entire one-dimensional array, calculating the indices for each simulated dimension at each iteration step.

3 Input and Output Methods

The cleanest way to use the cdf record is to input and output records through the argument list of the highest level SISAL function. The cdf record is thus stored externally in Fibre format (Skedzielewski and Yates, 1985). The hardest part of this process is in constructing and interpreting the cdf record. In order to make this simpler, I have written ten SISAL routines:

```
global cdfcreate(file_name: cdf_cline;
         instance: integer; returns cdf)

global cdfcomment(cdf_record: cdf; comment: cdf_cline;
         returns cdf)

global cdfparam(cdf_record: cdf; pname: cdf_cline;
```

```
          pval: real; returns cdf)

global cdfdim(cdf_record: cdf; dim_name: cdf_cline;
          dim_size: integer; start_val, increment: real;
          returns cdf, integer)

global cdfidim(cdf_record: cdf; dim_name: cdf_cline;
          dim_size: integer; data: array[real];
          returns cdf, integer)

global cdffld(cdf_record: cdf; field_name: cdf_cline;
          num_dims: integer; dh1, dh2, dh3, dh4: integer;
          fdata: array[real]; returns cdf)

global cdfgetpar(cdf_record: cdf; pname: cdf_cline;
          returns real)

global cdfgetdim(cdf_record: cdf; dname: cdf_cline;
          returns integer, cdf_dim)

global cdfgetfld(cdf_record: cdf; fname: cdf_cline;
          returns cdf_field)

global cdfwrite(cdf_record: cdf; returns integer)
```

Cdfcreate creates an empty cdf record with a specified name and instance. *Cdfcomment* and *cdfparam* respectively add comments and parameters to an old cdf record, returning a new record. *Cdfdim* and *cdfidim* return a new cdf record with information added about a new dimension. The first form assumes that dimension points are equally spaced, and therefore only the starting point and increment are needed. The second form accommodates irregular spacing by allowing the calling routine to specify each point. Both functions also return an integer handle which is used by the *cdffld* function. *Cdffld* adds data for a particular field to a cdf record. The number of dimensions and the particular dimensions used are calling arguments, as well as the field name and the data itself. *Cdfgetpar*, *cdfgetdim*, and *cdfgetfld*

respectively extract information about parameters, dimensions, and fields of existing cdf records. This aids interpretation of input files. Finally, *cdfwrite* converts a cdf record to a Candis file using calls to the C language library for Candis. It is the only routine that isn't pure SISAL, and provides a way of bypassing Fibre.

I have also written a C program, *cdffibre*, to convert a Candis file into a cdf record in Fibre format (to the extent that this is possible), and a SISAL program, *fibrecdf*, that calls the C-based Candis library to convert a cdf record in Fibre format to a Candis file. *Fibrecdf* is just a wrapper for the cdfwrite function.

4 An Example

As an example of the use of the above tools, I illustrate the solution of the two dimensional Poisson equation, $\nabla^2\psi = S$, using multigrid methods in a SISAL program. The computational algorithm isn't of interest here, only the interface part. The SISAL program accepts as its input a cdf record in Fibre format containing the source function S, information about the grid over which S is defined, and two parameters, *iters* and *cycles*, which are needed by the multigrid method. The program produces another cdf record in Fibre format containing the initial information plus the solution to the problem. The input file is produced by a series of Candis filters used to define the source function and parameters. The resulting Candis file is converted to cdf record format by cdffibre. The results are then converted to Candis format with fibrecdf, and plotted using the Candis program *cdfplot*.

The details of the process are best illustrated by examining the output of cdfplot. Contour maps of the source term S and the result ψ are shown in figures 1 and 2. On the right side of each figure is a complete history of the computational process. Examining figure 1, each word terminated by a colon represents an invoked program. *Cdfnull* defines the space over which the computation is defined — 17 points each in x and z, centered on (0, 0), with a grid spacing of 1 in each direction. The first call to *cdfmath* defines the field *rsq*, the square of the distance from the center of the domain. The second call defines the source function as $S = \exp(-0.5 * rsq)$, while the third and fourth define the parameters *iters* and *cycles*. (The notation is reverse Polish.) *Ptest* represents the call to the SISAL program. The rest

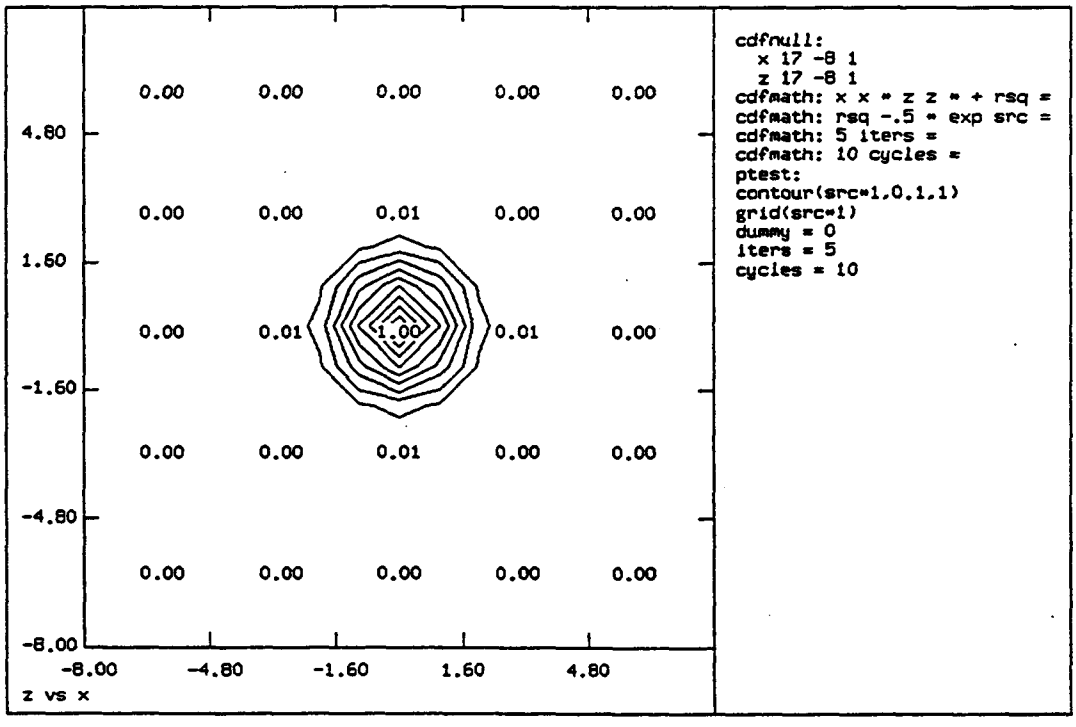


Figure 1: Contour plot of the source term, S, of the Poisson equation. On the right is a history list that provides a complete record of the computation.

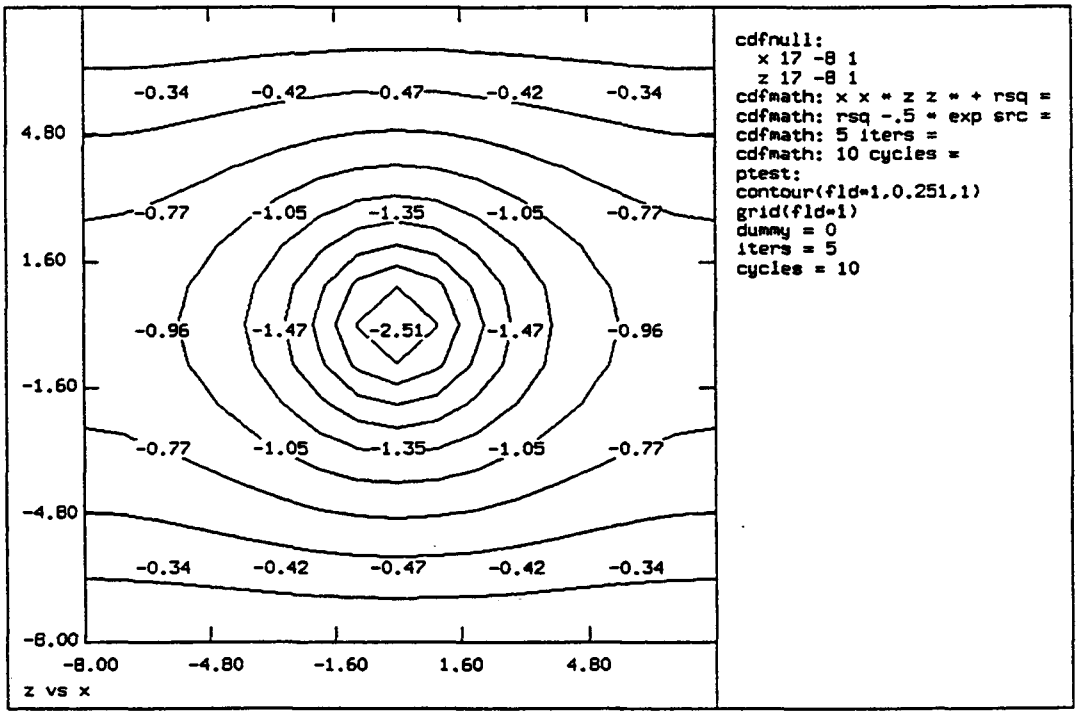


Figure 2: Contour plot of the solution, ψ, of the Poisson equation.

of the history list specifies the details of the contouring and values of scalar parameters.

Note that all initialization and plotting was done by canned programs. Only the computational program, *ptest*, had to be written especially for this problem. At worst, initialization and analysis requires the creation of simple shell scripts.

The working part of the ptest program (exclusive of type definitions and global statements) is shown below:

```
function ptest(cdf_in: cdf; returns cdf)
  let

% extract info from input cdf record
    dh1, dim1 := cdfgetdim(cdf_in, "x");
    dh2, dim2 := cdfgetdim(cdf_in, "z");
    nx := dim1.dsize;
    x0 := dim1.ddata[0];
    dx := dim1.ddata[1] - x0;
    nz := dim2.dsize;
    z0 := dim2.ddata[0];
    dz := dim2.ddata[1] - x0;
    pl := record[xstart: x0; zstart: z0; dx: dx; dz: dz;
              nx: nx; nz: nz];
    src := cdfgetfld(cdf_in, "src");
    cycles := integer(cdfgetfld(cdf_in, "cycles").fdata[0]);
    iters := integer(cdfgetfld(cdf_in, "iters").fdata[0]);

% call poisson solver
    fld := poisson(src.fdata, pl, cycles, iters);

% incorporate the output into the cdf record
    cdf1 := cdfcomment(cdf_in, "ptest:");
    cdf2 := cdffld(cdf1, "fld", 2, dh1, dh2, 0, 0, fld);
  in
    cdf2
  end let
end function % ptest
```

The actual solution is performed by the call to *poisson*, while the rest of the code deals with the interface. Note that the interface code is not very complex or lengthy.

5 Conclusion

A simple interface has been developed between SISAL and the Candis data analysis package. Experience with the interface shows that it is easy to use. SISAL can therefore be applied where it is strongest, i. e., in numerical computation. Input preparation and analysis of the output can be done using the Candis filter paradigm. Fibre is used as an intermediate form for input and output from SISAL programs. If desired (i. e., when the output file becomes very large, and hence inefficient to create and translate), output can be generated directly to Candis format from the SISAL program via a one line call to *cdfwrite*. This option is not available on input, but numerical simulations usually have much larger output than input.

The Candis package and the SISAL interface are available via anonymous ftp from the Unidata project, unidata.ucar.edu. A number of the filters in Candis are specific to the atmospheric sciences, but many are applicable to any gridded numerical data. Comments, bug reports, and suggestions should be sent to the author at raymond@kestrel.nmt.edu.

Acknowledgments. This work was supported by National Science Foundation Grant No. ATM-8914116.

6 References

McGraw, J., S. Allan, R. Oldehoeft, J. Glauert, C. Kirtham, B. Noyce, R. Thomas, 1985: *SISAL: Streams and iteration in a single assignment language: Language reference manual, version 1.2.* M-146, Univ. of California – Lawrence Livermore National Laboratory, Livermore, California.

Raymond, D. J., 1988: A C language-based modular system for analyzing and displaying gridded numerical data. *J. Atmos. Oceanic Tech.*, 5, 501-511.

Skedzielewski, S., and R. K. Yates, 1985: *Fibre: An external format for SISAL and IF1 data objects.* M-154, Univ. of California – Lawrence Livermore National Laboratory, Livermore, California.

PARALLELISATION AND PERFORMANCE OF THE BURG ALGORITHM ON A SHARED MEMORY MULTIPROCESSOR

A.L. Cricenti and G.K. Egan·

Abstract

This paper describes the implementation of a signal processing algorithm, specifically the Burg Algorithm, using both a high level al parallel language SISAL and Encore Parallel FORTRAN. The Burg Algorithm is an estimation technique for fitting an autoregressive model to a time series data set. This algorithm contains a time shift/inner product operation which is used in a number of other important signal processing algorithms such as convolution. The paper describes the results obtained using both the high level parallel language SISAL and EPF, on both an ENCORE Multimax multiprocessor machine, and a single processor IBM RS6000/530 machine.

1. Introduction

Signal Processing Algorithms are widely used and of vital importance in areas such as biomedical engineering, seismic data analysis, speech analysis and spectral estimation. The demand that signal processing algorithms place on computing system performance is increasing as more complicated algorithms are made to function in real time. As the limitations of current uniprocessor systems are being reached, many computer manufacturers are turning to multiprocessor configurations to obtain increased performance. In addition to hardware limitations, current computer languages must be evaluated for performance and ease of use with reference to their suitability for parallel machines.

The purpose of this paper is to describe the implementation of a signal processing algorithm, in this case the Burg Algorithm[1], using both a high level parallel language SISAL (Stream and Iteration in a Single Assignment Language)[2] and EPF (Encore Parallel FORTRAN). The Burg Algorithm is an estimation technique for fitting an autoregressive model to a time series data set. This algorithm contains a time shift / inner product operation which is used in a number of other important signal processing algorithms such as Convolution.

It is claimed that the optimising SISAL compiler (OSC) from Colorado State University yields performance competitive with FORTRAN [5][8]. Also the maximum concurrency of a SISAL program is theoretically only limited by the data dependencies. However, the OSC compiler on a shared memory machine, only exploits parallelism from the parallel loop construct.

The results obtained using an optimising SISAL compiler are compared to those obtained using the EPF (parallel FORTRAN) annotator. The comparison is made both on a 6 XPC processor (4 Mips per processor) based Encore Multimax Multiprocessor machine and a

---

·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 Institute of Technology, John Street, Hawthorn 3122, Australia, Phone: +61 3 819 8322, 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 Institute of Technology, John Street, Hawthorn 3122, Australia, Phone: +61 3 819 8516, E-mail: gke@stan.xx.swin.oz.au.

single processor IBM RS6000/530 (30 Mips) system. The performance of the IBM processor is representative of processors in next generation medium cost multiprocessors.

2. The Burg Algorithm

The Burg Algorithm is a method of generating an autoregressive model from a set of data samples, that is it gives estimates for A(z) in:

$$H(z) = \frac{1}{A(z)}$$

There are several ways of obtaining an AR model, the Burg algorithm is based on minimising the forward and backward prediction errors, assuming a lattice filter structure as shown in figure 1.

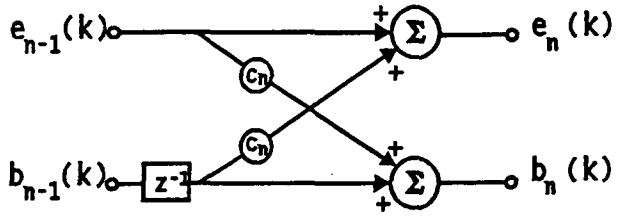


Figure 1 Lattice filter structure

where:

$$e_n(k) = e_{n-1}(k) + c_n\, b_{n-1}(k-1) \quad \text{forward prediction error} \qquad (1)$$

$$b_n(k) = c_n\, e_{n-1}(k) + b_{n-1}(k-1) \quad \text{backward prediction error} \qquad (2)$$

and $\mathbf{c_n}$ are called the reflection coefficients.

The Burg Algorithm involves the choice of reflection coefficients such that the error energy is minimised, when only a finite number of data samples is available.

The optimum value of the reflection coefficients can be easily derived[7] and is given by:

$$c_n = -2\frac{\sum_{k=n}^{M} e_{n-1}(k)*b_{n-1}(k-1)}{\sum_{k=n}^{M} e^2_{n-1}(k)*b^2_{n-1}(k-1)} \qquad (3)$$

where M is the number of data samples.

The autoregressive coefficients can then be estimated from:

$$a_n = c_n \qquad (4)$$

$$a_n(j) = a_{n-1}(j) + c_n a_{n-1}(n-j) \quad \text{for } j=1..n-1 \qquad (5)$$

A sequential implementation of the Burg algorithm is outlined in [1] and is reproduced in figure 2a with individual tasks labelled $T_{in}(1)$, $T_n(1)$, $T_{nn}(2)$, $T_{in}(2)$, $T_{in}(3)$. Tasks $T_{in}(1)$ are computations of the inner products in both the numerator and denominator of (3) above. $T_n(1)$ is the calculation of the division needed to compute c_n. This task cannot proceed in parallel, since it depends on the results of task $T_{in}(i)$. This data dependency can be also be seen from the maximally parallel graph for m=5 and max=3 reproduced in figure 2b. $T_{in}(2)$ updates the autoregressive coefficients and task $T_{in}(3)$ updates the forward and backward prediction errors (equations 1 and 2), the graph of figure 2b shows that each of the loops corresponding to tasks $T_{in}(1)$, $T_{in}(2)$, $T_{in}(3)$ can be computed in parallel.

```
1. INITIALIZATION

   FOR i=1 TO m DO
      e(i)=x(i)
      b(i)=x(i)

2. THE MAIN LOOP

   FOR n=1 TO max DO
      s1=s1+e(i)*b(i-n)                    Tin(1)
      s2=s2+e(i)**2+b(i-n)**2
   c(n)=-2.0*s1/s2                         Tn(1)
   IF n>1 THEN DO
      FOR i=1 TO n-1 DO
      a1(i)=a(i)+c(n)*a(n-i)               Tin(2)
      FOR i=1 TO n-1 DO
         a(i)=a1(i)
   a(n)=c(n)                               Tnn(2)
   FOR i=n+1 TO m DO
      temp=e(i)+c(n)*b(i-n)                Tin(3)
      b(i-n)=b(i-n)+c(n)*e(i)
      e(i)=temp
```

Figure 2a Sequential Burg Algorithm for m data points and max reflection coefficients from [1]

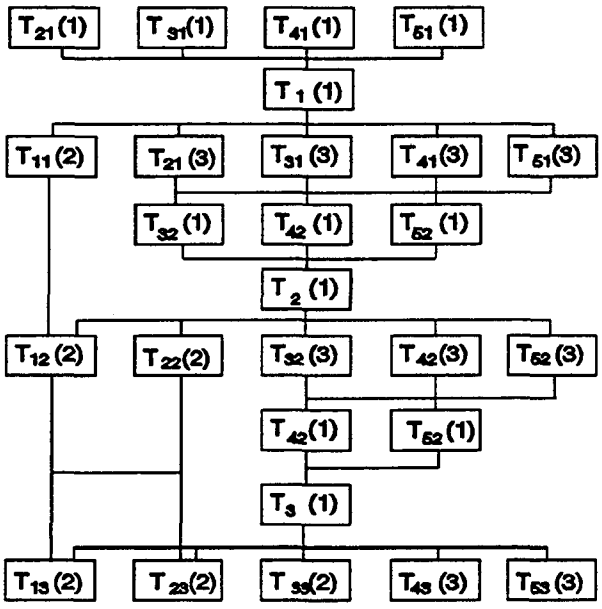


Figure 2b Maximally parallel graph for m=5 and max=3 from [1]

3. Language Considerations

3.1 Encore Parallel FORTRAN

The Encore Parallel FORTRAN compiler (EPF) is the UMAX f77 implementation of FORTRAN with enhancements which allow parts of a program to be executed in parallel. These statements are PARALLEL, DO ALL, CRITICAL SECTION, BARRIER, LOCK WAIT, LOCK SEND, and EVENT.

The EPF compiler consists of analysis and transformation tools, a parallelising compiler, parallel runtime libraries, and a code generator. Whilst programs can be written directly in EPF, EPF can also be used to convert a standard FORTRAN program into a source which is annotated with the parallel statements outlined above. During compilation, EPF first detects possible concurrent parts of the source programs, these are shown in a .LST file. The annotator then generates the EPF program , .E file, by inserting the appropriate EPF statements.

The EPF annotator may require user intervention to produce the most efficient code for a particular program; useful speedup can be achieved by fine tuning output of the annotator. However, for the simple code presented here, it is sufficient to rely on the annotator alone.

3.2 SISAL

SISAL is a functional language which has been targeted at a wide variety of systems including current generation multiprocessors such as the Encore Multimax and research dataflow machines[2][3][4]. The textual form of SISAL, in terms of control structures and array representations, provides a relatively easy transition for those familiar with imperative languages, since it has a PASCAL like syntax. The advantage of SISAL is that codes written in SISAL are portable to a variety of parallel architectures. SISAL also insulates programmers from the underlying machine architecture, and allows concurrency to expressed implicitly, thus removing the burden of processor synchronization and job scheduling.

4. Parallel Implementation

4.1 EPF

The simplest parallel implementation of the Burg algorithm is obtained by coding the sequential algorithm in FORTRAN and then using the Encore Parallel FORTRAN compiler (EPF) to produce the parallel code suitable for the Encore Multimax Multiprocessor. This process requires that the programmer knows very little about the underlying architecture of the machine, thus code may be generated very easily. This method is also attractive since it allows existing software, written in FORTRAN, to be implemented on parallel machines without any translation. There are two main disadvantages of this method. Firstly the optimum speedup is usually not obtained, since the original program may be sequential in nature. Secondly the annotated code produced by the EPF compiler is machine dependent.

The annotator has identified that all loops, in figure 2a, except the outer loop can be parallelised. Thus the annotator can successfully identify the parallel loops. The maximally parallel graph suggests that tasks $T_{in}(2)$, $T_{nn}(2)$ and $T_{in}(3)$ could be performed at the same time; unfortunately EPF can only slice loops, and since the outer loop is sequential, due to task $T_n(1)$, EPF cannot make these tasks parallel.

4.2 SISAL

The second implementation of the algorithm is in SISAL. The "disadvantage" here is that existing codes need to be re-expressed in the SISAL language, to expose the possible parallelism. The SISAL implementation of the Burg Algorithm as presented in [1], was directly transliterated from the FORTRAN version, excepting the loop which updates the autoregressive coefficients shown in figure 3a, which was transformed into a parallel format to ease the coding.

```
%calculate auto regressive coefficients

a:=  for k in 1,old n
  returns array of
       if k = old n then c
       else old a[k]+c*old a[old n-k]
       end if
  end for;
```

Figure 3a Implementation of the calculation of the autoregressive coefficients in SISAL.

The loop which updates the forward and backward errors was also changed, figure 3b. The original FORTRAN loop has been split into two loops. This was done so that the indexes of b change in manner which is suitable for the parallel *for* loop.

```
e:=
 for l in old n+1,m
 returns array of
      old e[l]+c* old b[l-old n]
 end for;
b:=
 for j in 1,m - old n
 returns array of
      old b[j]+c*old e[j+old n]
 end for;
```

Figure 3b Implementation of the updating of the forward and backward errors, in SISAL.

SISAL expresses concurrency naturally, therefore it is not possible to write a sequential loops in the parallel form. In order to achieve useful speedup, it is necessary to reorganise the computation, and rethink the algorithm in a parallel manner.

5. Results

The SISAL and FORTRAN versions of the program were run on both an Encore Multimax and IBM RS6000/530 system using the standard f77 FORTRAN compiler, the EPF compiler, where appropriate, and the optimising SISAL compiler (OSC v12.7).

For comparison purposes the number of data points was set to **m**=10000 and the model size to **max**=100. This model size was chosen so as to obtain run times which could be measured accurately.
The run times obtained for both the FORTRAN and SISAL implementations of the algorithm on the Encore Multimax multiprocessor with six XPC processors are summarised in table 1.

Note that $\text{Speedup} = \frac{T_{1\ proc}}{T_{n\ procs}}$ and $\text{Efficiency} = \frac{\text{Speedup(n)}}{n}$.

Processors	Time (s)	Speedup	Efficiency
1	29.6	1.00	1.00
2	17.2	1.72	0.86
3	12.7	2.33	0.78
4	9.7	3.05	0.76
5	8.2	3.61	0.72
6	7.8	3.79	0.63

Encore Parallel FORTRAN

Processors	Time (s)	Speedup	Efficiency
1	31.02	1.00	1.00
2	15.49	2.00	1.00
3	13.46	2.30	0.77
4	11.03	2.81	0.70
5	9.43	3.29	0.66
6	8.32	3.73	0.62

SISAL

Table 1 Experimental results on the Encore Multimax

As can be seen from the run times, speedup is achieved with the EPF compiler without significant programming effort. The EPF compiler converts DO LOOPS to parallel code. However, in some cases the annotator is fairly conservative, and further speedup may be obtained, in some instances, by manually annotating programs. In this study no manual annotation was performed as the speedup obtained for this simple code was satisfactory.

As can be seen, from the results, the FORTRAN and SISAL implementations achieve similar speedup, but the FORTRAN implementation has a lower run time for the single processor run, refer to table 1 and figure 4a.

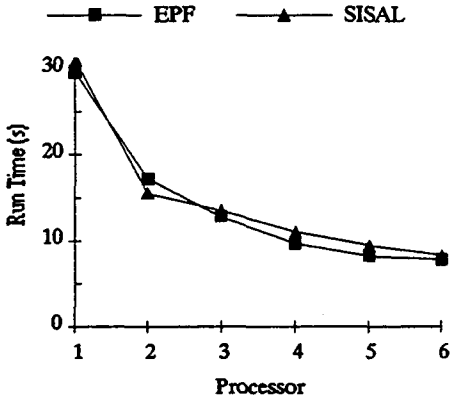


Figure 4a Run time vs processors

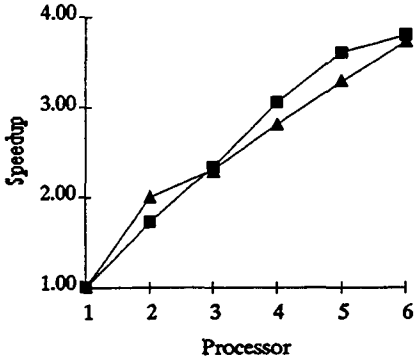


Figure 4b Speedup vs processors

It should be noted that initially no speedup was achieved with the SISAL implementation of the Burg Algorithm. Speedup was achieved by forcing the SISAL compiler to slice all for loops, by setting the -h pragma to 500; the cost estimator in SISAL had deemed the low complexity loops not worth slicing. The value of 500 was arrived at by trial and error. As this pragma is applied globally in the current version of the SISAL compiler there may be a risk of over parallelisation of some loops [6].

Speedup for the SISAL implementation is dependant on the size of the model, as can be seen from the graph shown in figure 5. Speedup increases as the number of data points increases, this is because for larger amounts of data, the processors spend relatively more time on useful computation, than on overheads computation.

The droop in the speedup curve, of figure 5, for six processors is due to other processes competing for the limited machine resources.

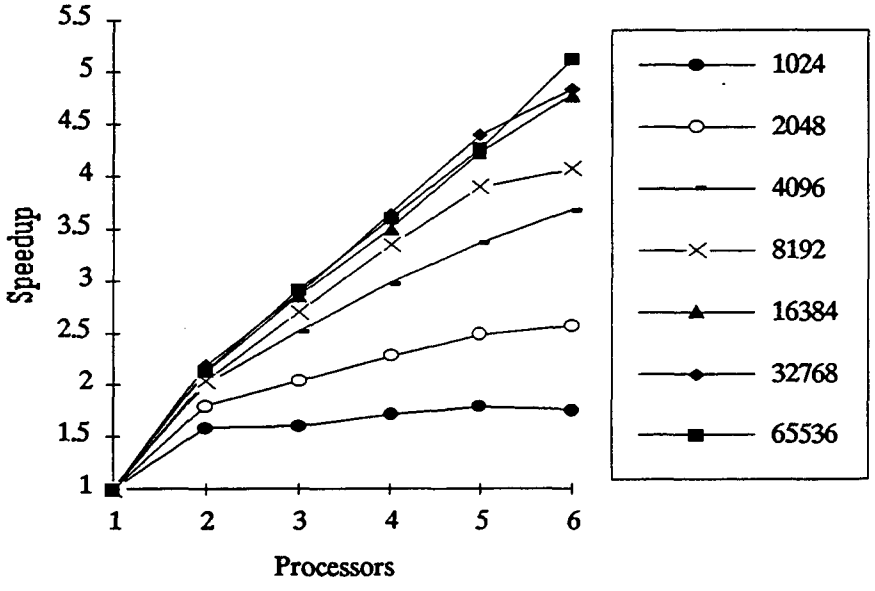


Figure 5 Speedup vs number of data points.

Speedup for the SISAL implementation increases beyond six processors as can be seen from the speedup curve shown in figure 6. These results were obtained on a slower 20 APC processor Encore Multimax. Only 16 processors were used so that interference from other users was minimised.

The graph shows the effect of Amdahl's law that forces the tail of the speedup curve to flatten. This limitation is due to the sequential part of the algorithm but good speedup is still achieved. The parallel FORTRAN (EPF) compiler was not available on this system.

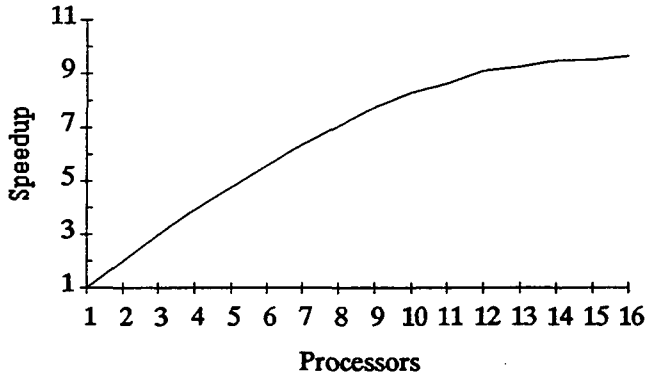


Figure 6 Speedup vs processors

The run times for a single processor (RS6000) machine are summarised in table 2. These times are for a model size of **m**=10000 and **max**=100. The time for the SISAL implementation is comparable with the FORTRAN[1] implementation.

	User +System (s)
SISAL	1.05
FORTRAN	4.96
FORTRAN -O	0.97

Table 2 Times for the IBM RS6000/530 (m=10000, max=100)

The results for the IBM RS6000/530 and Cray Y-MP are shown in table 3 for comparison with the results from [1], these results were obtained by tailoring the algorithm to the architecture of the target machine. The parameters for this study were **m**=16384 data points and model size **max**=10.

Machine	*HEP*	*iPSC/2*	*MPP*	*X-MP/48*	RS6000	Y-MP
Execution Time (s)	*1.679*	*0.24*	*0.5522*	*0.016887*	0.19	0.009 (1p)

Times from [1]

Table 3 Comparison of Burg Algorithm execution time (m=16384, max=10). Note the times for the Y-MP and RS6000 are for SISAL implementations.

6. Conclusions

The Burg filter was implemented both in FORTRAN and SISAL. Significant speedup is achieved with the SISAL implementation, suggesting that SISAL may be a useful language for signal processing algorithms. SISAL is useful since it allows parallelism to be expressed without considering processor synchronization and the underlying machine architecture. FORTRAN annotators such as the EPF annotator are useful in that speedup is obtained for little effort, and existing FORTRAN implementations of some simple algorithms need not be recoded, this is desirable since several digital signal processing FORTRAN programs already exist. Run times on modern single processor machines such as the IBM RS6000/530, are comparable to some existing parallel architecture machines, and give an indication of possible computation speeds of future generation multiprocessors.

---

1 XLF RS6000 FORTRAN Compiler

Acknowledgements

The authors thank the members of the Laboratory for Concurrent Computing Systems at the Swinburne Institute of Technology, in particular P.S. Chang, for their assistance in this research.

The authors thank Cray Research Australia for the use of the Cray Y-MP 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 both the EPF and SISAL codes for the Burg algorithm are available from the principal author.

References

[1] N.M. Sammur and M.T. Hagan. "Mapping Signal Processing Algorithms on Parallel Architectures." Journal of Parallel and Distributed Computing, Issue no.8 1990 pp180-185.

[2] McGraw et al., "SISAL: Streams and Iteration in a Single Assignment Language." Language Reference Manual, M146, Lawrence Livermore National Laboratories.

[3] A.P.W. (Wim) Bohm and J. Sargeant, "Efficient Dataflow Code Generation of SISAL", Technical Report UMCS-85-10-2, Department of Computer Science, University of Manchester, 1985.

[4] G.K. Egan, N.J. Webb and A.P.W. (Wim) Bohm, "Some Features of the CSIRAC II Dataflow Machine Architecture", in Advanced Topics in Data-Flow Computing, Prentice-Hall 1990,

[5] D.C. Cann, "High Performance Parallel Applicative Computation", Technical Report CS-89-104, Colorado State University, Feb.1989.

[6] P.S. Chang, and G.K. Egan "Analysis of a Parallel Implementation of a Numerical Weather Model in the Functional Language SISAL" Technical Report 31-012, Laboratory for Concurrent Computing Systems, School of Electrical Engineering, Swinburne Institute of Technology, March 1990.

[7] R.A. Roberts and C.T. Mullis, "Digital Signal Processing" Addison - Wesley 1987

[8] D.C. Cann, "Retire Fortran? A Debate Rekindled", Technical Report UCRL-JC-107018, Lawrence Livermore National Laboratory, Apr.1991.

Use of Genetic Algorithms in SISAL to solve the File Design Problem

Walter Cedeño
Computing Research Group/CMRD
Lawrence Livermore National Laboratory
Livermore, CA 94550 (wcedeno@llnl.gov)

Abstract

The File Design Problem is a NP-complete problem normally encountered in the design of databases. The goal is to find an assignment of database records to files that minimizes the average number of files examined over all single attribute queries. This paper describes a solution to the File Design Problem using Genetic Algorithms written in SISAL. Using the portability and architecture independence inherent in SISAL a Genetic Algorithm model is defined that provides increased performance without the sacrifice of convergence. A heuristic based mating operator essential to the solution to the problem is described. Selection and replacement operators useful for multimodal function optimization are used to search for multiple solutions. Results with various test cases are shown. Performance of the algorithm is shown for different platforms.

1. Introduction

The File Design Problem is a NP-complete problem, the number of possible solutions increases exponentially as the problem size increases linearly. Its applicability to database design in a homogeneous distributed system is known. Better solutions to the problem will enhance the overall performance of such systems.

The problem is defined as follows: Given a set of N records, characterized by a single attribute A which takes h different values { a_1, a_2, ..., a_h}. There are n_i records corresponding to attribute a_i, i.e., $n_1 + n_2 + ... + n_h = N$. Find an assignment of the records to K files of size b such that the average number of files (ANF) examined over all possible single attribute

queries is minimized. In other words, a configuration must be found such that in the average, requests for the records with the same attribute can be satisfied by reading from as few files as possible. The assumption is made that queries for any attribute are equally likely. The constants K, b, N, and h are all positive integers, moreover $K * b = N$. An example with four solutions is shown in table 1.

Table 1: Possible solutions for 12 records, 2 files of size 6, and $A = \{ a, b, c, d\}$, $n_a = 7$, $n_b = 2$, $n_c = 2$, and $n_d = 1$.

File 1	File 2	fex(a)	fex(b)	fex(c)	fex(d)	*ANF*
a a a a a a	a b b c c d	2	1	1	1	1.25
a a a b c c	a a a a b d	2	2	1	1	1.50
a a a a b c	a a a b c d	2	2	2	1	1.75
a a a b b d	a a a a c c	2	1	1	1	1.25

The *ANF* for a solution is given by the formula $\frac{\sum_{i=1}^{h} \texttt{fex}(a_i)}{h}$, where $\texttt{fex}(a_i)$ returns the number of files containing at least one record with attribute a_i. From table 1, the second solution has a value of 2 for fex(b) since both files contain a record with attribute b. The first and last solutions in the table are examples of optimal solutions for this problem. Even though the *ANF* values are the same, the last solution is better because it has a more balanced configuration. If requests for the attributes are distributed uniformly, file 1 and file 2 will be accessed 25% and 100% respectively in the first solution, and 50% and 75% respectively with the last solution. This idea is incorporated when evaluating solutions generated by a Genetic Algorithm.

Genetic Algorithms (GAs) are general purpose search procedures introduced by Holland [9] in the 70's. They are based on the principle of natural selection and genetic recombination. Like in nature, GAs use the mechanics of evolution to improve a set of initial solutions called a *population* using recombination and mutation of the "genetic material". This work follows loosely the steady-state GA model. In this model, new solutions called *offspring* are created using solutions from the current

population. Each offspring is then inserted sequentially in the population by selecting another solution to die. This model allows the best solutions to survive between generations.

GAs have been successfully applied to the Traveling Salesman Problem [16], Scheduling [14], and the Bin Packing Problem [4] to mention a few. In some cases better results were obtained when the mating operator was designed to capture the essential information in the problem. The mating operator for the File Design Problem is based on "first fit" and "best fit" heuristics, such heuristics group records with the same attribute together. The multimodal search space in the problem is explored in many directions by using selection and replacement operators [1] that encourages mating and replacement between solutions from the same extrema.

There are basically three parallel GA models [6] exhibiting different degrees of parallelism; fine grain, distributed, and direct. In a fine grain model [2,7,13], each solution in the population is mapped to a processor with genetic operators applied between nearest neighbors. In a distributed model [12,15], processors are assigned subpopulations, which converge locally and exchange genetic material among them. Direct models [8] exploit the parallelism inherent in the GA operators and the GA structure while having the same properties of a sequential GA. Our SISAL GA follows the direct model while having the localize convergence exhibited in the other models.

SISAL [11] is a functional language that provides the tools to implement a parallel GA application portable to different multi-processor platforms. The parallelism at the GA top level and in the operators is easily exploited. Performance is increased while maintaining the necessary computation for solving the problem. The best solution was found in all test cases with a speedup of at least 2.2 with four workers.

The paper is organized as follows. Section 2 gives an overview of the GA for this problem. Section 3 describes the genetic operators specific for the File Design Problem. Section 4 defines the experimental setup. Section 5 contain the results and conclusions.

2. The Genetic Algorithm Model

The SISAL GA was designed to capture the parallelism in the model while maintaining the search to multiple solutions. In this model multimodality is exploited by encouraging mating and replacement between solutions from the same extrema. Performance is improved by creating the offspring in parallel. The offspring are then inserted into the population sequentially to preserve replacement between members of the same extrema.

The solutions in the initial population are created in parallel by assigning records to files at random. Each file is divided into b slots corresponding to its size. The slots are uniquely numbered with a value between 1 and N. Each record is assigned a slot number corresponding to a position in a file. The constraints of the problem are easily maintained without the need for counters for each of the files. The *fitness*, a measure of how good is a solution, is then calculated for each member of the population.

The algorithm is executed for a fixed number of generations. Each generation consists of creating all the offspring and inserting them into the population. Three steps are involved to create two offspring: select the parents, apply the mating operator to the parents, and calculate the fitness to the offspring. Mutation is applied by the mating operator as part of the mating process. Each offspring is inserted sequentially in the population by selecting an existing solution to die.

Solution from the population	Group of *CF* candidates selected randomly	Most similar solution from group to Parent 1
	001101010011	
001110000111	100001111001	001101010011
(Parent 1)	000011111001	(Parent 2)

Figure 2. Selecting the mate from a set of solutions.

To create the offspring each solution in the population is allow to mate at least once in every generation. It's mating partner is selected in the following manner: form a group of *CF* (crowding factor constant) solutions

chosen at random from the population, and pick the most similar for mating. Similarity between two solutions is measured as the number of records assigned to the same file. An example is shown in figure 2 using the data from table 1. Assuming that each digit indicates the file where a record is located, the first solution is selected to mate with parent 1 because it has 8 records assign to the same file. This selection operator encourages mating between solutions within the same extrema. After selection, mating produces two offspring and their fitness are computed. The number of offspring created can be up to two times the number of solutions in the population. All of them are created in parallel with a given mating and mutation probability.

Step 1: Form *CF* groups with *CS* candidates selected at random from the population.

Group 1	Group 2	Group 3
001101010011	000011111001	001111000101
001110001011	100001111001	101001010011

Step 2: Select most similar solution to the offspring from each group and compare their fitness values.

	001110001011	000011111001	001111000101
fitness:	0.46	0.75	0.67

Step 3: Replace the solution with lower fitness with the offspring.

001110001011 is replaced by 001110000111

Figure 3: Inserting offspring 001110000111 in the population using a value of $CF = 3$ and $CS = 2$.

The offspring are inserted one at a time in the population by deleting the worst solution from a set of most similar solutions. For each offspring *CF* groups containing *CS* (crowding sub-population constant) randomly selected solutions from the population are formed. From each of these groups the most similar solution to the offspring is selected, forming a set of *CF* solutions. From this set, the solution with lowest fitness is replaced by the offspring in the population. After an offspring is inserted in the population it becomes a candidate for replacement. Some offspring will be replaced during the same generation. An example of this operator is shown

in figure 3 using the configuration from table 1. As in selection the replacement operator is bias toward solutions within the same extrema. Convergence is improved by eliminating solutions with lower fitness.

3. Genetic Operators

In this section the encoding and genetic operators for the File Design Problem are described. They were designed to preserve the constraints of the problem and make them easier to implement using SISAL arrays.

3.1 Gene Encoding

A *gene* represents a valid solution to the problem. It consist of an array of N *alleles* corresponding to each of the records in the problem. Each allele may assume a value between 0 and K - 1 inclusive, indicating the file containing the record. A valid encoding is a N digit number base K where all digits appear exactly b times. An example is shown in figure 4.

record number:	1	2	3	4	5	6	7	8	9	10	11	12
record attribute:	a	a	a	a	a	a	a	b	b	c	c	d
gene 1:	0	0	0	0	1	1	1	1	2	2	2	2
gene 2:	0	0	1	1	2	2	2	0	0	1	1	2

Figure 4. GA encoding for 12 records in 3 files of size 4.

3.2 Mating Operator

The mating operator for the File Design Problem creates two offspring and was designed with two ideas in mind. First, the characteristics expressed in both parents will be expressed in the offspring, thus preserving the schemata in both solutions. Second, fitness should be improved when combining two similar solutions. "Best fit" and "first fit" heuristics are used for this. Incorporating these features in the mating operator improves convergence between solutions from the same extrema. When two solutions from different extrema mate, offspring from other extrema can be created. This way the operator is not restricted to small areas in the search space.

The first step in the mating operator is to transfer similar characteristics from the parents to the offspring. This is done by transferring the records assigned to the same file in both parents to the same file in the offspring. Those records not assigned are counted for each attribute and sorted in decreasing order. One offspring is created using a best fit method based on the contents of files. Unassigned records will be located into files where records with the same attribute reside. The second offspring is created using a first fit method based on the empty space in the files. Unassigned records will be located where file space is available for records with the same attribute. Using the configuration in figure 4 an example is shown in figure 5.

Offspring inherits similar alleles from parents:

```
  Record Attribute: a a a a a a a b b c c d
           Parent 1: 0 0 1 2 2 2 0 1 0 1 2 1
           Parent 2: 0 1 0 1 2 2 1 2 0 0 1 2
          Offspring: 0 - - - 2 2 - - 0 - - -
Unassigned records by attribute: a:4, b:1, c:2, d:1
```

Assignment of records in sorted order to both offspring:

	Offspring 1 (Best Fit Method)	Offspring 2 (First Fit Method)
a:4	0 2 2 0 2 2 0 - 0 - - -	0 1 1 1 2 2 1 - 0 - - -
c:2	0 2 2 0 2 2 0 - 0 1 1 -	0 1 1 1 2 2 1 - 0 0 0 -
b:1	0 2 2 0 2 2 0 1 0 1 1 -	0 1 1 1 2 2 1 2 0 0 0 -
d:1	0 2 2 0 2 2 0 1 0 1 1 1	0 1 1 1 2 2 1 2 0 0 0 2

Figure 5. Mating operator for the File Design Problem

Given the parents in figure 5, the offspring inherits only four alleles; 3 records with attribute a and 1 record with attribute b. Using the best fit method the other 4 records with attribute a are assigned to file 2 and file 0 because those files contain records with the same attribute. Using the first fit method the 4 records are assigned to file 1 because that file is the most empty. The other records are assigned in a similar manner.

Mutation is applied with a fixed probability for each allele. When an allele is selected for mutation another position in the gene is selected at random and the two values are interchange. Such mutations may introduce a new configuration in succeeding generations.

3.3 Fitness Function

The fitness function captures three important characteristics of an optimal solution: low *ANF*, records with the same attribute are grouped together, and records with the same attribute are spread equally among the minimum number of files needed to store them. The last two points are captured in a grouping term (*GT*) and balancing term (*BT*) respectively. The two terms are contradictory in the sense that *BT* wants to group records together, while *GT* wants to spread records equally across files. Because it induces lower *ANF* values the *BT* value is given higher weight when calculating fitness.

The *ANF* value is given by the formula:

$$ANF = \sum_{i=1}^{h} fex(a_i) \Big/ h,$$

where $fex(a_i)$ returns the number of files with attribute a_i. From this formula, we compute an upper and lower bound to the problem. The lower bound represents a configuration where the records for all attributes are assigned to the least number of files needed to contain them. An upper bound is given when the records for all attributes are spread in as many files as possible. The lower and upper bound are called *min_anf* and *max_anf* respectively and are given below:

$$min\_anf = \sum_{i=1}^{h} \lceil n_i/b \rceil \Big/ h \leq ANF \leq max\_anf = \sum_{i=1}^{h} \min(n_i, K) \Big/ h.$$

The *GT* value is computed by adding for all files and all attributes the squared value of the fraction of records for each attribute. The higher the number of records of the same attribute in a file the higher the *GT* value. The formula for the *GT* value is given below:

$$GT = \sum_{i=1}^{h} \sum_{j=1}^{K} (attr(a_i, j)/n_i)^2,$$

where $attr(a_i,j)$ returns the number of records of attribute a_i in file j.

The *BT* value is computed by adding for all files the absolute value of the difference between the number of records for each attribute and a balance configuration for the attribute. Only files containing records for the given attribute are included in the summation. The formula for this term is given below:

$$BT = \sum_{i=1}^{h} \sum_{j=1}^{K} \lfloor |attr(a_i, j) - n_i/\lceil n_i/b \rceil| \rfloor, \text{ when } attr(a_i, j) \neq 0.$$

Here $\lceil n_i/b \rceil$ returns the number of files needed to store the records of attribute a_i. Values of *BT* closest to zero represent more balanced configurations.

The three terms *ANF*, *BT*, and *GT* are used to define the fitness value for a solution. Since higher positive values are used to indicate a better solution the terms are normalized to return values between 0.0 and 1.0. A percentage of each term is then added to form the final fitness value as indicated by the following formula:

$$\text{fitness} = 0.70 * \frac{GT}{h} + 0.25 * \frac{max\_anf - ANF}{max\_anf - min\_anf} + 0.05 * \frac{1.0}{1.0 + BT}.$$

The fitness value for any solution is a number between 0.0 and 1.0. Solutions where the fitness value is 1.0 represent configurations where the *min_anf* value is achievable and the records for any attribute is less than the file size. Having the property of fitting records with the same attribute in one file eliminates the conflict between *BT* and *GT* while obtaining a maximum value of *h* for *GT*.

4. Experimental Data

To evaluate the behavior of the algorithm six test cases where created having different properties. Test cases with solutions achieving the

min_anf lower bound, other cases having the number of records for some attributes exceeding the file size, and multiple attributes per file were mixed to create the different configurations. For all cases 100 records were used. Table 6 summarizes all configurations created.

Table 6: Configuration for all test cases

Case num	number of files	file size	number of attributes	number of records per attribute n1 n2 n3 n4 n5 ... nh	*min_anf* exist
1	5	20	10	7, 2, 3, 1, 5, 17, 18, 13, 15, 19	Yes
2	10	10	10	7, 2, 3, 1, 5, 17, 18, 13, 15, 19	Yes
3	5	20	10	7, 4, 3, 8, 6, 11, 18, 15, 10, 18	No
4	10	10	10	7, 4, 3, 8, 6, 11, 18, 15, 10, 18	No
5	5	20	21	7, 4, 3, 8, 6, 1, 8, 5, 10, 8, 1, 2, 4, 9, 5, 1, 6, 2, 3, 3, 4	Yes
6	5	20	15	7, 4, 9, 7, 7, 4, 9, 5, 9, 7, 9, 5, 6, 7, 5	No

To evaluate the performance of the implementation 3 different platforms were used: the SGI Iris 4D, Cray Y-MP, and Cray C90. The execution time from one to four workers was collected for the algorithm using case 1 in table 6. The GA parameters used for each run are:

Population size - 100

Number of generations - 50

Mating probability - 0.95

Mutation probability -0.01

CF for selection - 4

CF for replacement - 3

CS for replacement - 5

This parameters were chosen after a trial and error period. They represent a good set of choices for the test data shown in table 6.

5. Results and Conclusions

The results obtained for the test data in table 6 were very good. In all cases multiple optimal solutions were found to the problem. In four of the six test cases at least one optimal solution was found prior to generation 6. More generations were needed for the cases where *min_anf* did not exist

and the number of records for some attributes were high. In those cases, the solutions were competing between themselves for a very low improvement in fitness. Table 7 shows solutions for all test cases and the generation number on which they were obtained. Each solution is represented by the assignment to all records with the same attribute separated by commas.

Table 7: GA solution to test cases with number of generations needed

Case number	Generations needed	Solution found
1	3	444444, 11, 333, 0, 22222, 33333333333333333, 111111111111111111, 4444444444444, 222222222222222, 0000000000000000000
2	4	999999, 77, 111, 6, 88888, 33333333113311111, 2222222222777777777, 5555555555999, 444444444488888, 0000000000666666666
3	25	222222, 0000, 222, 44444444, 333333, 44444444444, 000000000100001000, 333333433333333, 2222222222, 111111111111111111
4	15	888888, 9999, 888, 55555555, 999999, 22222222227, 000000000066666666, 111111111155667, 3333333333, 4444444444777777777
5	5	000000, 3333, 222, 33333333, 222222, 0, 00000000, 11111, 2222222222, 44444444, 2, 44, 4444, 111111111, 33333, 4, 111111, 44, 333, 444, 0000
6	5	000000, 2222, 222222222, 4444444, 2222222, 3333, 333333333, 44444, 000040000, 3333333, 111111111, 11111, 111111, 4444444, 00000

A speedup between 2.2 and 2.9 was achieved with four workers in the three different platforms. Figure 8 summarizes the performance from one to four workers in the different platforms.

Platform	1 Worker	2 Workers	3 Workers	4 Workers
Y-MP C90	12.9799	8.6748	6.4425	5.4930
Y-MP	18.6106	10.4642	8.9153	6.3648
SGI Iris	25.8900	16.5300	13.4200	11.9000

Figure 8: GA execution time in seconds for 50 generations.

The results obtained with this GA model are encouraging. Exploiting the multimodality inherent in the File Design Problem resulted in a more balanced search over the entire space. Creating genetic operators that

enhance the search globally as well as locally were very important for this problem. Developing the GA from the problem's point of view provided positive results for this problem. The Convergence to optimal solutions was achieve in all cases while improving the performance using SISAL. Improvement in performance can be achieve by parallelizing the replacement operator and retain its property of replacing solutions within the same extrema.

Acknowledgments

This work was supported (in part) by the Applied Mathematics Program of the Office of Energy Research (U.S. Department of Energy) under contract number W-7405-Eng-48 to Lawrence Livermore National Laboratory.

References

[1] W. Cedeño and V. Vemuri, "Dynamic Multimodal Function Optimization using Genetic Algorithms", Proceedings XVIII Latinamerican Conference in Computer Science, August 1992.

[2] Y. Davidor, "A naturally Occurring Niche & Species Phenomenom: The model and First Results", Proceedings Fourth International Conference on Genetic Algorithms, Morgan Kaufman, July 1991.

[3] L. Davis, ed. "Handbook of Genetic Algorithms", Van Nostrand Reinhold New York, NY, 1991.

[4] E. Falkenauer and A. Delchambre, "A Genetic Algorithm for Bin Packing and Line Balancing", Proceedings of the 1992 IEEE International Conference on Robotics and Automation, May 1992.

[5] D. E. Goldberg, "Genetic Algorithms in Search, Optimization, and Machine Learning", Addison-Wesley, Reading MA, 1989.

[6] V.S. Gordon, D. Whitley, and A.P.W. Böhm, "Dataflow Parallelism in Genetic Algorithms", Parallel Problem Solving from Nature 2, Elsevier Science Publishers, September 1992.

[7] M. Gorges-Schleuter, "ASPARAGOS An Asynchronous Parallel Optimization Strategy", Proceedings Third International Conference on Genetic Algorithms, Morgan Kaufman, June 1989.

[8] Grefenstette, J.J., "Parallel Adaptive Algorithms for Function Optimization", Technical Report No. CS-81-19, Vanderbilt University, CS Department, 1981.

[9] J. H. Holland, "Adaptation in Natural and Artificial Systems", University of Michigan Press, Ann Arbor, 1975.

[10] J. Liang, C. C. Chang, R. C. T. Lee, and R. C. T. Wang J. S., "Solving the File Design Problem with Neural Networks", Tenth Annual International Phoenix Conference on Computers and Communications, March 1991.

[11] J. McGraw et al, "SISAL - Streams and Iteration in a Single-Assignment Language, Lawrence Livermore National Laboratory M-146, January 1985.

[12] H. Mühlenbein, M Schomish, J. Born, "The Parallel Genetic Algorithms as Function Optimizer", Proceedings Fourth International Conference on Genetic Algorithms, Morgan Kaufman, July 1991.

[13] P. Spiessens and B. Manderick, "A Massively Parallel Genetic Algorithm - Implementation and First Analysis", Proceedings Fourth International Conference on Genetic Algorithms, Morgan Kaufman, July 1991.

[14] G. Syswerda and J. Palmucci, "The Application of Genetic Algorithms to Resource Scheduling", Proceedings Fourth International Conference on Genetic Algorithms, Morgan Kaufman, July 1991.

[15] R. Tanese, "Distributed Genetic Algorithms", Proceedings Third International Conference on Genetic Algorithms, Morgan Kaufman, June 1989.

[16] D. Whitley, T. Starkweather, and D. Fuquay, "Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator", Proceedings Third International Conference on Genetic Algorithms, Morgan Kaufman, June 1989.

IMPLEMENTING FFT'S IN SISAL*

Dorothy Bollman, Flor Sanmiguel, and Jaime Seguel
Department of Mathematics
University of Puerto Rico at Mayaguez, PR 00681-5000

Abstract. Tensor notation is a powerful tool for improvising fast Fourier transform (FFT) algorithms in a functional language. In this work we apply this idea to the development of FFT's in the functional language Sisal. As an example, we develop a high performance Sisal implementation of a variant of Pease's FFT on a Cray Y-MP.

1. Introduction. Although the utility of tensor notation for FFT algorithms has long been recognized [3], it has been only recently [4], [8], [9], that this notation has gained wide spread acceptance by FFT practitioners. Tensor notation aids not only in algorithm expression and comprehension, but also in algorithm development and is an ideal tool for developing parallel-vector algorithms for functional programs. In this work we describe work in progress that exploits this idea to develop a family of high performance FFT's in the functional language Sisal. As an example, we present a high performance Sisal implementation of a variant of the radix 4 Pease [6] algorithm.

In this section, we establish definitions and notation. In Section 2, we review a well-known family of FFT's that are variants of the original Cooley-Tukey [1] algorithm and discuss their implementations in Sisal. In Sections 3 and 4 we compare Sisal implementations of variants of the radix 4 Korn-Lambiote and Pease algorithms. The Pease variant outperforms Korn-Lambiote as well as Cann's [1] implementation of Stockham's algorithm.

The n-point DFT of a complex sequence $x = x(k)$ of period n is the complex sequence of period n determined by the set of equations

$$y(j) = \sum_{k=0}^{n-1} \omega_n^{jk} x(j), \quad 0 \le j \le n-1, \tag{1}$$

where $\omega_n = e^{-2\pi i/n}$ and $i = \sqrt{-1}$. In matrix notation, the DFT is given by

$$y = F_n x, \quad F_n = (\omega_n^{ij}), \quad 0 \le i,j \le n-1. \tag{2}$$

Thus the DFT of a vector of length n can be computed in time $O(n^2)$. However in 1965, Cooley and Tukey [3] showed that the DFT can be computed in time $O(n\ logn)$. Subsequently, variations of the Cooley-Tukey algorithm, which vary in vector size and data flow, have appeared in the literature. These algorithms together with the original Cooley-Tukey algorithm are generically known as fast Fourier transforms (FFT's). In the next section we shall describe a family of FFT's in terms of their tensor formulations.

The tensor product of an $m \times m$ matrix $A = (a_{ij})$, $i,j = 0,...,m-1$, by an $n \times n$ matrix B is the $mn \times mn$ matrix defined by

$$A \otimes B = (a_{ij}B). \tag{3}$$

If x is a vector of length mn and if we partition x into m blocks of length n, $X_0, X_1, \ldots, X_{m-1}$, then

$$(I_m \otimes B)x = (BX_0, BX_1, \ldots, BX_{m-1}). \tag{4}$$

Thus, the computation of $(I_m \otimes B)x$ can be thought of as the *parallel* application of B to the m vectors $X_0, X_1, \ldots, X_{m-1}$. On the other hand, it is easily shown that

$$(B \otimes I_m)x = (b_{0,0}X_0 + \ldots + b_{0,n-1}X_{n-1}, \ldots, b_{n-1,0}X_0 + \ldots + b_{n-1,n-1}X_{n-1}), \tag{5}$$

where $B = (B_{ij})$, $0 \le i,j \le n-1$, and $X_0, X_1, \ldots, X_{n-1}$ represents a partitioning of x into n blocks of length m. In view of (5), $B \otimes I_m$ has been termed a *vector* operation. Although these parallel/vector

* This work was supported by NSF grant RII-8905080, the Computational Mathematics Group of Puerto Rico EPSCoR II grant, and the NSF Cornell NSI Army Research Office grant.

interpretations of the operations $(I_m \otimes B)x$ and $(B \otimes I_m)x$ are convenient for conceptional purposes, the actual machine vectorization/parallelizations depend on their use in programs and decisions made by the compiler.

An important aspect of FFT implementations is data communication, which is manifested in the tensor formulations by permutation matrices, or stride permutations. A *stride* permutation $P(mn, n)$ acting on a vector of length mn is defined by

$$P(mn,n)x = ((x_0, x_n, \ldots, x_{(m-1)n}), (x_1, x_{n+1}, \ldots, x_{(m-1)n+1}), \ldots (x_{n-1}, x_{2n-1}, \ldots, x_{mn-1})), \tag{6}$$

where $x = (x_0, x_1, \ldots, x_{mn-1})$.

A useful property of stride permutations is the so called Commutation Theorem:

$$P(mn,n)(A \otimes B) = (B \otimes A)P(mn,n), \tag{7}$$

where A and B are $m \times m$ and $n \times n$, respectively.

2. A family of FFT's. We define

$$D_s^{rs} = diag(1, \omega_{rs}, \omega_{rs}^2, \ldots, \omega_{rs}^{s-1}) \tag{8}$$

and

$$T_s^{rs} = \bigoplus_{i=0}^{r-1} (D_s^{rs})^i, \tag{9}$$

where

$$(D_s^{rs})^i = diag(1, \omega_{rs}^i, \ldots, {\omega_{rs}}^{i(s-1)}) \tag{10}$$

and where $\oplus$ respresents direct sum, i.e., T_s^{rs} is a matrix whose nonzero elements consist of $(D_s^{rs})^i$, $i = 0, 1, \ldots, r-1$, along the main diagonal. In what follows we denote $T_{r^{i-1}}^{r^i}$ simply by T^{r^i}.

An FFT for computing $F_n x$ can be described in terms of a factorization of the matrix F_n. The FFT's discussed here, the proofs of which can be found in, e.g., [8], all follow from the following identity:

$$F_{rs} = (F_r \otimes I_s) T_s^{rs} (I_r \otimes F_s) P(sr, r). \tag{11}$$

The original Cooley-Tukey algorithm in tensor notation is:

$$F_{r^k} = \{\prod_{i=1}^{k} (I_{r^{k-i}} \otimes F_r \otimes I_{r^{i-1}})(I_{r^{k-i}} \otimes T^{r^i})\} R(r^k), \tag{12}$$

where $R(r^k)$ represents the digit-reversal permutation. Note the succint way in which the tensor notation bares the idea of the algorithm. In view of the above definitions, formula (12) says that in order to compute the FFT of a vector x of length r^k, we perform a digit-reversal permuation on x. Then for each stage $i = 1, \ldots, k$, apply the "twiddle" factor T^{r^i} to each of r^{k-i} blocks of length r^i, followed by r^{k-i} applications of the "butterfly" operations $F_r \otimes I_{r^{i-1}}$ to each resulting block.

One of the problems with implementing the Cooley-Tukey algorithm on parallel/vector machines is that vector lengths become small and/or that overparallelzation takes place for small values of i. More suitable algorithms for parallel/vector machines are the "parallel" algorithm

$$F_{r^k} = \{\prod_{i=1}^{k} P(r^k, r)(I_{r^{k-1}} \otimes F_r) P(r^k, r^{k-1})(T^{r^i} \otimes I_{r^{k-i}}) P(r^k, r)\} R(r^k), \tag{14}$$

originally developed by Pease [6] and its "vector" variant

$$F_{r^k} = \{\prod_{i=1}^{k} (F_r \otimes I_{r^{k-1}})(T^{r^i} \otimes I_{r^{k-i}}) P(r^k, r)\} R(r^k), \tag{13}$$

first described by Korn-Lambiote [5].

Each of the above algorithms is given in its "decimation in time" form, i.e., the "combine phase" is applied to the result of digit-reversal permuting the input vector. In the "decimation in frequency" (DF) versions digit-reversal is applied to the output of the combine phase. The DF forms are easily obtained from the DT forms by taking the transpose of each side of the given formula and making use of following properties:

$$(A \otimes B)^t = A^t \otimes B^t, \tag{15}$$

where A^t denotes the transpose of A and

$$P^t(mn, n) = P^{-1}(mn, n) = P(mn, m). \tag{16}$$

Cooley-Tukey DF (Gentleman-Sande):

$$F_{r^k} = R(r^k)\{\prod_{i=1}^{k}(I_{r^{i-1}} \otimes T^{r^{k-i+1}})(I_{r^{i-1}} \otimes F_r \otimes I_{r^{k-i}})\}. \tag{17}$$

Pease DF:

$$F_{r^k} = R(r^k)\{\prod_{i=1}^{k} P(r^k, r^{k-1})(T^{r^{k-i+1}} \otimes I_{r^{i-1}})P(r^k, r)(I_{r^{k-1}} \otimes F_r)P(r^k, r^{k-1})\}. \tag{18}$$

Korn-Lambiote DF:

$$F_{r^k} = R(r^k)\{\prod_{i=1}^{k} P(r^k, r^{k-1})(T^{r^{k-i+1}} \otimes I_{r^{i-1}})(F_r \otimes I_{r^{k-1}})\}. \tag{19}$$

The above tensor formulas completely functionalize the given algorithms. One only has to program the functions, $R(r^k)x$, $(I_s \otimes B)x$, $(B \otimes I_s)x$, $(T^{n/s} \otimes I_s)x$, $(I_s \otimes T^{n/s})x$, $F_r x$, $P(n, n/s)x$, and Dx, where D is a diagonal matrix, to obtain the entire family of algorithms. However, the resulting algorithms are not necessarily optimal. In the first place, replacing B by F_r in $B \otimes I_s$ or $I_s \otimes B$ does not take advantage of the symmetries in F_r. Also, the compiler might not fuse loops across functions optimally. In particular, for implementations on a Cray, special care must be taken in fusing a power of 2 stride permutation in order to avoid memory bank conflicts. Nevertheless, the tensor notation provides a powerful tool for deriving the necessary functions for programming optimal FFT's.

3. A Sisal program for the radix 4 DF Korn-Lambiote algorithm. We now turn to the problem of developing Sisal programs for FFT algorithms defined by their tensor representations. We shall present only key functions and we shall do this in terms of a psuedo code rather than detailed, less readable, Sisal code. In particular, we shall represent complex vectors by the data type **array**[complex]. "Complex" is not actually a data type in Sisal 1.8. In our programs we implement complex arrays by pairs of real arrays.

All of the FFT's of the previous section involve a separate ordering phase, i.e., application of digit-reversal either before or after the combine phase. The Stockham algorithm [2], which has received considerable attention by implementers of FFT's on Crays, avoids the ordering phase by distributing digit-reversal throughout the combine phase. Cann [1] has developed, in an imperative style, a Sisal implementation of the radix 4 Stockham algorithm which he claims outperforms, on a Cray Y-MP, the FFT of the signal processing library provided by Cray Research, Inc..

The virtue of the Stockham algorithm is that it avoids the cost of digit- reversal. However, in some applications of the FFT, such as convolution, it is desirable to separate the combine and ordering phases. In this section and the next we compare functional versions of the DF Korn-Lambiote and the DF Pease algorithms and their implementations on a Cray Y-MP. In order to reduce the number of serial steps and the number of flops, as well as the number of memory stores and fetches, which is important for a Cray, we consider the case $r = 4$.

Let us first examine the DF version of the Korn-Lambiote algorithm which is traditionally regarded as a vector algorithm. For $r = 4$, (19) becomes

$$F_{4^k} = R(4^k)\{\prod_{i=1}^{k} P(4^k, 4^{k-1})(T^{4^{k-i+1}} \otimes I_{4^{i-1}})(F_4 \otimes I_{4^{k-1}})\} \tag{20}$$

The implementation of this algorithm clearly requires a sequential loop of length k followed by an application of digit-reversal. Our goal is to simplify the structure of each serial step by combining each application of $P(4^k, 4^{k-1})$ with the twiddle factor $T^{4^{k-i+1}} \otimes I_{4^{i-1}}$ and the butterfly operation $F_4 \otimes I_{4^{k-1}}$ in such a way as to eliminate power of 2 strides. To this end, let V_i be the diagonal of $T^{4^{k-i+1}} \otimes I_{4^{i-1}}$. Then for any vector y of length n and any i,

$$P(4^k, 4^{k-1})(T^{4^{k-i+1}} \otimes I_{4^{i-1}})(F_4 \otimes I_{4^{k-1}})y = P(4^k, 4^{k-1})(V_i \circ (F_4 \otimes I_{4^{k-1}}))y \tag{21}$$

where $\circ$ denotes component-wise vector multiplication. Now we would like to fuse $P(4^k, 4^{k-1})$ with the loop for $(V_i \circ (F_4 \otimes I_{4^{k-1}}))y$. However, this necessitates a scatter operation, which is not readily implementable in Sisal 1.8. Consequently, we take advantage of the ability of Sisal's forall expression to return multiple values. We compute, not $(V_i \circ (F_4 \otimes I_{4^{k-1}}))$ as a single vector, but rather the four block components $Y_0, E_{i,1} \circ Y_1, E_{i,2} \circ Y_2$, and $E_{i,3} \circ Y_3$, where $V_i = (I, E_{i,1}, E_{i,2}, E_{i,3})$ and $(F_4 \otimes I_{4^{k-1}})y = (Y_0, Y_1, Y_2, Y_3)$. Now $P(4^k, 4^{k-1})$ can be applied by interleaving the four blocks $Y_0, E_{i,1} \circ Y_1, E_{i,2} \circ Y_2$, and $E_{i,3} \circ Y_3$). If we precompute all values of $E_{i,1}, E_{i,2}, E_{i,3}$, then each serial step of our Korn-Lambiote variant will consist of an invocation of the function B followed by an *interleave*. The function B is defined by the following pseudo Sisal code:

```
function B(k:integer;W1,W2,W3:array[complex];x:array[complex]
          returns array[complex],array[complex],array[complex],array[complex])
let
 ek1 := exp(4, k - 1);
 vector1,
 vector2,
 vector3,
 vector4 := for j in 0,ek1-1
             comp1,comp2,comp3,comp4 :=
              f4(x[j], x[j + ek1, x[j + 2 * ek1], x[j + 3 * ek1], W1[j], W2[j], W3[j])
            returns array of comp1
                    array of comp2
                    array of comp3
                    array of comp4
            end for
in
 array_setl(vector1,0),
 array_setl(vector2,0),
 array_setl(vector3,0),
 array_setl(vector4,0)
end let
end function
```

Here $f4$ is an optimized function which computes the component-wise product of a vector of the form $(1, w_1, w_2, w_3)$ by the Fourier transform of a vector x of length 4. The function *interleave* can be implemented by a loop of length 4^{k-1} and stride 1 as follows:

```
function interleave(length:integer;x1,x2,x3,x4:array[complex] returns array[complex])
 array_setl(for i in 0,length-1
             returns value of catenate array[0:x1[i],x2[i],x3[i],x4[i]]
            end for,0)
end function
```

Now fft can be implemented as follows

```
function fft(k:integer;index:array[integer];E1,E2,E3:array[array[complex]];x:array[complex]
          returns array[complex])
```

```
let
 ek1 := exp(4, k − 1);
 n := 4 * ek1;
 z :=
          for initial
             i := 1;
             y := x
             while i <= k repeat
               y := interleave(ek1, B(k, E1[oldi], E2[oldi], E3[oldi], oldy));
               i := oldi + 1 ;
             returns value of y
          end for
in
 permute(z, n, index)
end let
end function
```

3. A Sisal program for the radix 4 DF Pease algorithm. Although the implementation of $P(4^k, 4^{k-1})$ in terms of *interleave* eliminates power of 2 strides, it does not come without cost, namely, it must be invoked at each serial step. In the case of the Pease algorithm, the cost of performing the first two stride permutations in

$$F_{4^k} = R(4^k)\{\prod_{i=1}^{k} P(4^k, 4^{k-1})(T^{4^{k-i+1}} \otimes I_{4^{i-1}})P(4^k, 4)(I_{4^{k-1}} \otimes F_4)P(4^k, 4^{k-1})\} \tag{22}$$

can be transferred to the precomputation of the twiddle factors. To see this, define V_i as before and observe that for any vector y

$$\begin{aligned} &P(4^k, 4^{k-1})(T^{4^{k-i+1}} \otimes I_{4^{i-1}})P(4^k, 4)(I_{4^{k-1}} \otimes F_4)P(4^k, 4^{k-1})y \\ &= P(4^k, 4^{k-1})V_i \circ P(4^k, 4^{k-1})P(4^k, 4)(I_{4^{k-1}} \otimes F_r)P(4^k, 4^{k-1})y \\ &= P(4^k, 4^{k-1})V_i \circ (I_{4^{k-1}} \otimes F_4)P(4^k, 4^{k-1})y \end{aligned} \tag{23}$$

In this case, we define $P(4^k, 4^{k-1})V_i = (I, E_{i,1}, E_{i,2}, E_{i,3})$ and we precompute all values of $E_{i,1}, E_{i,2}, E_{i,3}$. Now each serial step can be performed by a simple invocation of the following function

```
function B(k:integer;W1,W2,W3:array[complex];x:array[complex] returns array[complex])
let
 ek1 := exp(4, k − 1);
in
array_setl(
 for j in 0,ek1-1
  quad := f4(x[j], x[j + ek1], x[j + 2 * ek1], x[j + 3 * ek1], W1[j], W2[j], W3[j])
 returns value of catenate quad
 end for,0)
end let
end function
```

The *fft* function in this case is the same as the one given for the Korn- Lambiote variant except that *interleave* is omitted. The savings obtained by the elimination of the interleave operation is considerable. For example, execution time on the Cray Y-MP for $n = 4^9$ using all 8 processors is only 55% that of the Korn-Lambiote variant.

Each of the above algorithms is valid only for vector lengths that are powers of 4, i.e., even powers of 2. When $n = 2^k$ where k is odd, $F_n(x)$ can be computed making use of the following formula which is obtained from (11) by transposing each side and taking $n = rs$ where $r = 2$ and $s = 2^{k-1}$:

$$F_n = P(2^k, 2^{k-1})(I_2 \otimes F_{2^{k-1}})T^{2^k}(F_2 \otimes I_{2^{k-1}}) \tag{24}$$

The Sisal code for implementing (24) is straightforward. We compute $F_n(x)$ by computing $T^{2^k}(F_2 \otimes I_{2^{k-1}})x$ in two halves in a way similar to the computation of the B function for the Korn-Lambiote variant. Then an "even" FFT is applied to the two halves and the two results are interleaved.

Tables 1 and 2 compare running times on the NERSC CRAY Y-MP for our Sisal implementation of the Pease variant (PFFT) with those of Cann's (CFFT) after having changed the single precision real arithmetic of CFFT to double precision as used in PFFT. PFFT was compiled with the option *-maxconcur* while CFFT was compiled with the options recommended by Cann, i.e., *-hybrid -concur -vector.*

Table 1. Execution times in seconds for PFFT, $n = 2^k$.

k	1 CPU	4 CPU's	8 CPU's
16	0.042011	0.012496	0.006037
17	0.087329	0.024462	0.013726
18	0.175957	0.046390	0.024890
19	0.377403	0.099038	0.058168
20	0.825157	0.201309	0.107798

Table 2. Execution times in seconds for CFFT, $n = 2^k$.

k	1 CPU	4 CPU's	8 CPU's
16	0.034853	0.014313	0.013358
17	0.061205	0.023366	0.018410
18	0.123053	0.046848	0.038326
19	0.259286	0.094445	0.071659
20	0.523968	0.204599	0.159223

The above times do not include initialization, which for CFFT includes the computation of a table of sines and cosines and which for PFFT includes the computation of a table of roots of unity as well as the computation of a table of indices used for digit-reversal. Times for PFFT do, however, include the digit-reversal permutation, which constitute, no matter the number of processors, about 11% of the above running times.

The next two tables campare MFLOP rates for PFFT and CFFT.

Table 3. MFLOP rates for PFFT, $n = 2^k$.

k	1 CPU	4 CPU's	8 CPU's
16	106	356	738
17	109	391	697
18	114	432	805
19	114	431	735
20	108	443	961

Table 4. MFLOP rates for CFFT, $n = 2^k$.

k	1 CPU	4 CPU's	8 CPU's
16	112	272	292
17	127	334	424
18	145	381	465
19	137	377	498
20	153	392	504

In general, the performance of PFFT in comparison to CFFT will be best for $n = 2^k$ when k is even, as can be expected from (24).

The simple loop structure of PFFT is obtained in large part by trading code for the memory space needed to store, in a redundant way, various roots of unity. For example, for $n = 2^k$, k even, the precomputed tables in each of the programs requires the computation and storage of $\frac{3}{4}kn$ complex numbers. Computation of these tables is from three to ten times slower than computation of the table in CFFT. On the other hand, using a highly efficient algorithm [7], the computation of the table of digit-reversal indices is very fast, e.g., .001 seconds for $n = 4^9$ and eight processors. But in any case, the time for initialization is insignificant if the program is used to compute a large number of FFT's of the same size.

The same ideas used in PFFT could be applied to obtain fast programs for F_n where $n = r^k$ and r is any positive integer. This would require some slight changes in the precomputation of the twiddle factors as well as an optimal function for the computation of F_r for a given r. PFFT could also be modified, at the cost of additional loop structure, to reduce the amount of memory needed to store the precomputed powers of ω.

Conclusions. We have presented a mathematical framework for functionalizing and optimizing FFT algorithms. Starting with an FFT algorithm expressed in tensor notation, one can use the laws of tensor algebra to obtain tensor formulas that take into account a given architecture and are easily translated into Sisal. Work is in progress to develop Sisal implementations for other FFT algorithms using this same methodology. An interesting problem is to investigate the possibility of developing a high performance "universal" FFT Sisal program, in the sense that given an FFT algorithm A in terms of a tensor formula, the program optimally computes $F_n(x)$ according to A.

Acknowledgement

The authors are grateful to Lawrence Livermore National Laboratory for the Cray computer time made available through the Sisal Scientific Computing Initiative (SSCI).

References

[1] D. Cann, "Retire Fortran? A Debate Rekindled," Comm. ACM, vol. 30(8) (1992), 81-89.

[2] W. T. Cochran, et al. "What is the Fast Fourier Transform?," IEEE Trans. Audio Electroacoust., AU-15(2) (1967), 45-55.

[3] J.W. Cooley and J.W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," Math. Comp., vol. 19 (1965), 297-301.

[4] J. Johnson, R. Johnson, D. Rodriguez, and R. Tolimieri, "A Methodology for Designing, Modifying and Implementing Fourier Transform Algorithms on Various Architectures," Circuits, Systems and Signal Processing vol. 9 (1990), 449-500.

[5] D.G. Korn and J.J. Lambiotte, "Computing the Fast Fourier Transform on a Vector Computer," Math. Comp., vol. 33 (1979), 977-992.

[6] M.C. Pease, "An Adaptation of the Fast Fourier Transform for Parallel Processing," J. ACM, vol. 15 (1968), 252-265.

[7] J. Seguel, D. Bollman, and V. Celis, "A family of parallel algorithms for digit-reversal," submitted.

[8] R. Tolimieri, M. An, and C. Lu, Algorithms for Discrete Fourier Transform and Convolution, Springer-Verlag, New York, 1989.

[9] C. Van Loan, Computational Frameworks for the Fast Fourier Transform, S.I.A.M., 1992.

Programming and Evaluating the Performance of Signal Processing Applications in the SISAL Programming Environment *

Dae-Kyun Yoon and Jean-Luc Gaudiot
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089-2563

Abstract

This paper presents a signal processing application and its implementation in the SISAL programming environment. It shows that a signal processing application can be effectively coded in SISAL and can be analyzed to investigate the feasibility of its parallel implementation using the SISAL tools. We also present the implementation of Fast Fourier Transform (FFT) which is a kernel of many signal processing applications on the Sequent Symmetry systems. It is shown that exploiting function call parallelism can be more effective than the parallel *loop slicing* for some applications.

1 Introduction

Most signal processing applications are highly computation-intensive, and in many cases, real-time performance is needed. While the algorithms for solving such applications are well known and understood, their execution is still one or two orders of magnitude slower than what is needed for real-time processing.

A significant speed-up in the evaluation of such algorithms can be obtained by their implementation in a multiprocessor system. However, the implementation in a multiprocessor system is much more complex than in a single-processor system. The complexity of programming in multiprocessor systems is mainly due to the fact that a programmer has to specify parallelism *explicitly* to exploit the parallelism of the given algorithm. Moreover, even if it is possible for the compiler to extract parallelism out of the sequential program and to generate an appropriate code for a parallel machine, the extent of parallelization is limited because of the complexity in analyzing the sequential program.

In a pure functional programming language, parallelism is implicitly expressed. Since a program written in a pure functional language is side-effect-free, the order of each statement in the program is not as significant as in an imperative language. Each statement can be executed in any order as long as its inputs are already defined. Therefore, for a compiler, it is very easy to extract parallelism out of the program written in a functional programming language.

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

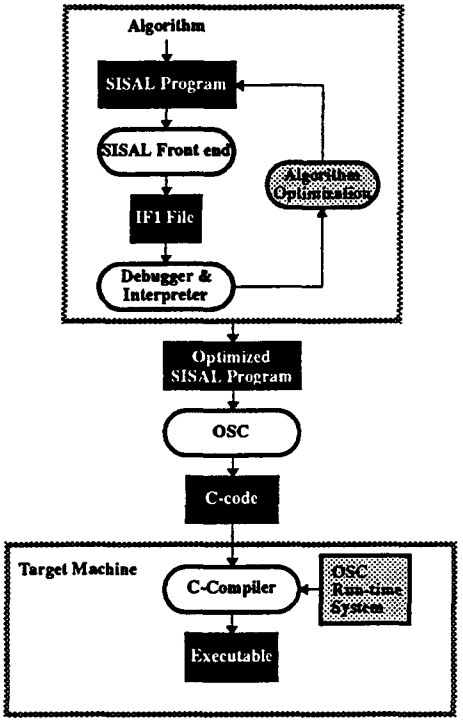


Figure 1: The overall programming process

SISAL (Streams and Iterations in a Single Assignment Language)[7] is the functional language which we chose for parallel implementation of our application. SISAL has been proved very effective in developing applications for implementation in multiprocessor systems. An optimizing compiler for SISAL has been developed for various shared memory multiprocessor systems[3, 4]. It has also been shown that the performance of the SISAL programs is comparable to (or better than) that of Fortran programs[2].

In this paper, we present a signal processing application and its parallel implementation in the SISAL programming environment. The overall programming process is shown in Figure 1. In the first phase (the upper block), we analyze the selected algorithm to verify that the algorithm is suitable for parallel implementation. In this phase, we mainly observe the potential parallelism of the algorithm. Its main purpose is to see whether the application (or the algorithm) has enough parallelism to exploit when implemented in a multiprocessor system, and determine the origin of the parallelism. If the potential parallelism is not enough, the algorithm is redesigned (or optimized). In the next phase (the lower block in Figure 1), we implement the algorithm on a target machine. In our case, we implemented the FFT, which is the kernel of our signal processing application, on the Sequent Symmetry shared memory multiprocessor system.

The FFT algorithm we selected is based on double recursion. This recursion is the main source of parallelism of the algorithm. The *loop-slicing* technique of most parallelizing compilers including the one we use(OSC) cannot speedup this type of algorithm. Therefore,

we implemented the parallel function call mechanism in the run-time system of the target machine, and the recursive function call was replaced by parallel function call in the translated C-code. We observed that, for the recursive FFT algorithm, the parallel function call outperformed loop-slicing. This result convinced us that, for a certain type of applications, especially with recursive function calls, parallel function call is a very effective way of exploiting parallelism.

In the subsequent sections, we describe the application and the relevant algorithms, and then we present the analysis of the application using SISAL tools. In Section 4, we describe the implementation details and the performance measures of the recursive FFT on the Sequent Symmetry using the parallel function call.

2 Description of the Application

The real-time analysis of sonar signals for the mapping of the ocean floor or the detection of submerged targets has been found to be similar to the modeling of wave propagation in the troposphere using the parabolic equation[5]. The *split-step* algorithm which is based on the following two equations is used to compute $u(x,z)$, the power gains (or losses) at (x,z), where x is the horizontal range and z is the vertical range from the source (*e.g.*, an antenna). Forward and backward Fourier transforms are denoted F and F^{-1} respectively.

$$U(x,p) = F[u(x,z)]$$

$$u(x+\delta x, z) = e^{i(k/2)(n^2-1)\delta x}\ F^{-1}[U(x,p)e^{-i(p^2\delta x/2k)}]$$

where:

$$\begin{aligned} p &= k\sin\theta \\ \theta &= \text{The angle from the horizontal} \\ k &= \omega\sqrt{\mu\epsilon(a,0)} \\ \epsilon(a,0) &= \text{The permittivity just above the earth's surface} \\ a &= \text{Radius of the earth} \end{aligned}$$

The split-step algorithm is applied along the horizontal range. Therefore, the dependence between each step of computation is as follows: the label (F or F^{-1}) on the arrow shows the Fourier transform involved at each step.

$$u(0,z) \xleftarrow{F^{-1}} U(0,p) \xrightarrow{F^{-1}} u(\delta x, z) \xrightarrow{F} U(\delta x, p) \xrightarrow{F^{-1}} u(2\delta x, z) \xrightarrow{F} \cdots$$

To begin calculations, we have to find the initial value(s), $U(0,p)$. $U(0,p)$ is obtained by the following equations:

$$\begin{aligned} U(0,p) &= U_e(0,p) + U_o(0,p) \\ U_e(0,p) &= 2F_c[f(z)\cos p_e z]\cos pz_A - 2iF_s[f(z)\sin p_e z]\sin pz_A \\ U_o(0,p) &= 2F_s[f(z)\sin p_e z]\cos pz_A - 2iF_c[f(z)\cos p_e z]\sin pz_A \\ f(z) &= F_c^{-1}[F_d(p)] \end{aligned}$$

Where $F_d(p)$ is the antenna pattern and F_c and F_s are cosine and sine transforms.

The application kernel shown above can be described in algorithmic form as follows:

Algorithm *split_step*

Input:

$F_d(p)$: Sampled values which represent the antenna pattern.

n : Number of steps over the horizontal range.

δx : Incremental value for each range step.

k : $\omega\sqrt{\mu\epsilon(a,0)}$ as described above.

n' : Refractive index.

θ_{max} : Maximum angle for which the antenna pattern is sampled.

n_p : Number of sampled points.

Output:

$u(x,z)$: The power gains (or losses) at (x,z)

Begin

1. Find the initial condition $U(0,p)$ from $F_d(p)$.
2. $u(0,z) \leftarrow$ *inverse_fft* $(U(0,p))$.
3. $c_1 \leftarrow e^{i(k/2)(n'^2-1)\delta x}$
4. $x \leftarrow 0, j \leftarrow 1$
5. **While** $j \leq n$ **Repeat**
   - 5–1. $c_2(p) \leftarrow e^{-i(p^2\delta x/2k)}$
   - 5–2. $u(x+\delta x,z) \leftarrow c_1$ *inverse_fft* $(U(x,p)c_2(p))$
   - 5–3. $U(x+\delta x,p) \leftarrow$ *fft* $(u(x+\delta x,z)$
   - 5–4. $x \leftarrow x+\delta x, j \leftarrow j+1$

End

The initial function $U(0,p)$ is obtained by proper modeling of the source, *i.e.* the antenna. The equations which are needed to compute $U(0,p)$ are shown in the previous section, and a detailed description of source modeling can be found in [5].

The FFT algorithm is recursively expressed as follows[6, 9]:

Algorithm *fft*

Input:

f : array of N complex numbers, where N is the power of 2.

Output:

F : array of N complex numbers which is the fourier transform of f.

Begin

1. Let f^e be the array of even components of f.
   Let f^o be the array of odd components of f.
2. Find $F^e = fft(f^e)$ and $F^o = fft(f^o)$.
3. Let W be $e^{2\pi i/N}$.
   For $k = 0,\ldots,N/2$, let $L_k = F^e_k + W^k F^o_k$.
   For $k = 0,\ldots,N/2$, let $U_k = F^e_k - W^k F^o_k$.
4. For $k = 0,\ldots,2/N$, let $F_k = L_k$ and $F_{k+N/2} = U_k$. In other words, F is the concatenation of L and U.

End

The *split_step* algorithm performs $O(n)$ Fourier transforms. The *FFT* itself has a time complexity $O(n \log n)$ when it is executed sequentially. Therefore the *split_step* algorithm takes $O(mn \log n)$ to complete its computation on a sequential machine, where m is the number of steps and n is the number of sampled points. The *Split_step* algorithm is based on iterating over each horizontal range step. In other words, each step is dependent on the result of the previous step, Therefore, the parallelism of the *split_step* algorithm can be achieved only by the potential parallelism of the Fourier transform at each step. Thus even if we increase the problem size with the number of range steps, parallelism does not increase. In other words, speed-up with a large number of processors can be obtained only if we have enough sampled points for the Fourier transform. Therefore, as per *Amdahl's law*[1], the execution time of the algorithm is dominated by the iteration over each range step which is the sequential part of the algorithm. Assuming an infinite number of processors, the *split_step* algorithm takes $O(m \log n)$ time.

3 Analysis of the Application using SISAL Tools

As we have already discussed in Section 1, several steps are involved in developing an application in a parallel programming environment. First of all, we have to select (or design) an algorithm which is suitable for parallel implementation. The selected algorithm is then coded in a target language. The next step is to debug and optimize the program using parallel programming tools. These steps are, indeed, the same needed when developing an application on a sequential machine. However, the appropriateness of the algorithm for parallel execution is mainly determined by the potential parallelism of the algorithm. If the algorithm itself does not have any parallelism, we cannot expect any speedup even with an infinite number of processing elements. In this section, we present the performance measures of the algorithm which are obtained using the SISAL tools.

We use the following tools to run the SISAL program of the application and observe the performance measures of the program (see also Figure 1).

1. OSC (Optimizing SISAL Compiler) : OSC generates a C-code out of the SISAL program with intensive optimization[4, 3]. We apply OSC to produce optimized IF1[11] in analyzing our signal processing application.
2. DI (Debugger and Interpreter) : DI is used to verify the functionality of the program,and to observe the potential parallelism and its simulated execution time.

The general procedure of developing an application using these tools can be summarized as follows.

1. Design an algorithm.
2. Implement the algorithm in SISAL.
3. Compile the SISAL program into an IF1 graph.
4. Run the IF1 graph by using DI and observe the potential parallelism.

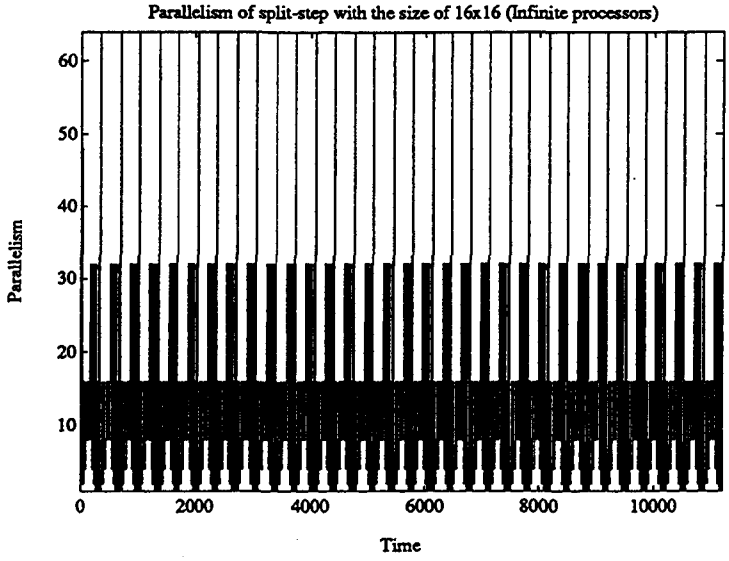


Figure 2: Potential parallelism of *split_step*

The potential parallelism obtained by the simulation of 16 sampled points over 16 range steps is shown in Figure 2. As we discussed in the earlier section, the *split_step* executes the loop body sequentially over each range step. Thus, as can be seen in Figure 2, there is little parallelism during the execution of the whole program. However, if we carefully look at the graph there are $2n$ repeated patterns, where n is the number of range steps over which we apply the *split_step* algorithm. The detailed view of the parallelism is shown in Figure 3. Parallel execution of the program, indeed, can reduce the interval between steps, since there is potential parallelism at each step, depending on the number of sampled points for which the Fourier transform is performed. The potential parallelism of FFT is shown in Figure 4. The first graph shows the maximum potential parallelism with an infinite number of processors and the second part shows the clipped parallelism with 32 processors.

We ran the *split_step* program for various data sizes, 64×32, 64×64, 128×32 and 128×64, where $m \times n$ means m sample points over n split-steps. The number of processors for which we simulated ranges from 1 to 256. The speedup is shown in Table 1 and Figure 5.

This simulation results confirm that a speedup can be obtained by employing a large number of processors only if we have enough sampled points. The fact that the number of steps over the horizontal range cannot increase the potential parallelism is rather disappointing. However, we can still benefit by employing multiple processors, if there are a large number of sampled points compared to the number of available processors. For example, as can be seen in Figure 5, we can achieve a nearly linear speedup up to 32 processors when we have 128 sampled points in the ideal case.

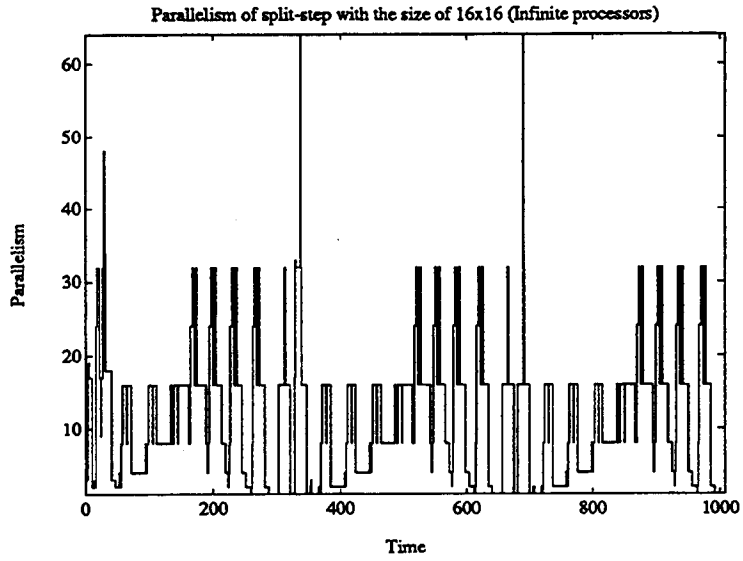


Figure 3: Potential parallelism of *split_step* (magnified view of the first three steps)

	64 × 32	64 × 64	128 × 32	128 × 64
4	3.7	3.7	3.8	3.8
8	7.0	7.0	7.2	7.2
16	12.3	12.3	13.3	13.3
32	20.2	20.2	22.8	22.8
64	31.6	31.6	36.6	36.6
128	36.6	36.6	53.3	53.2
256	36.7	36.7	61.3	61.2

Table 1: Speedup of *split_step*

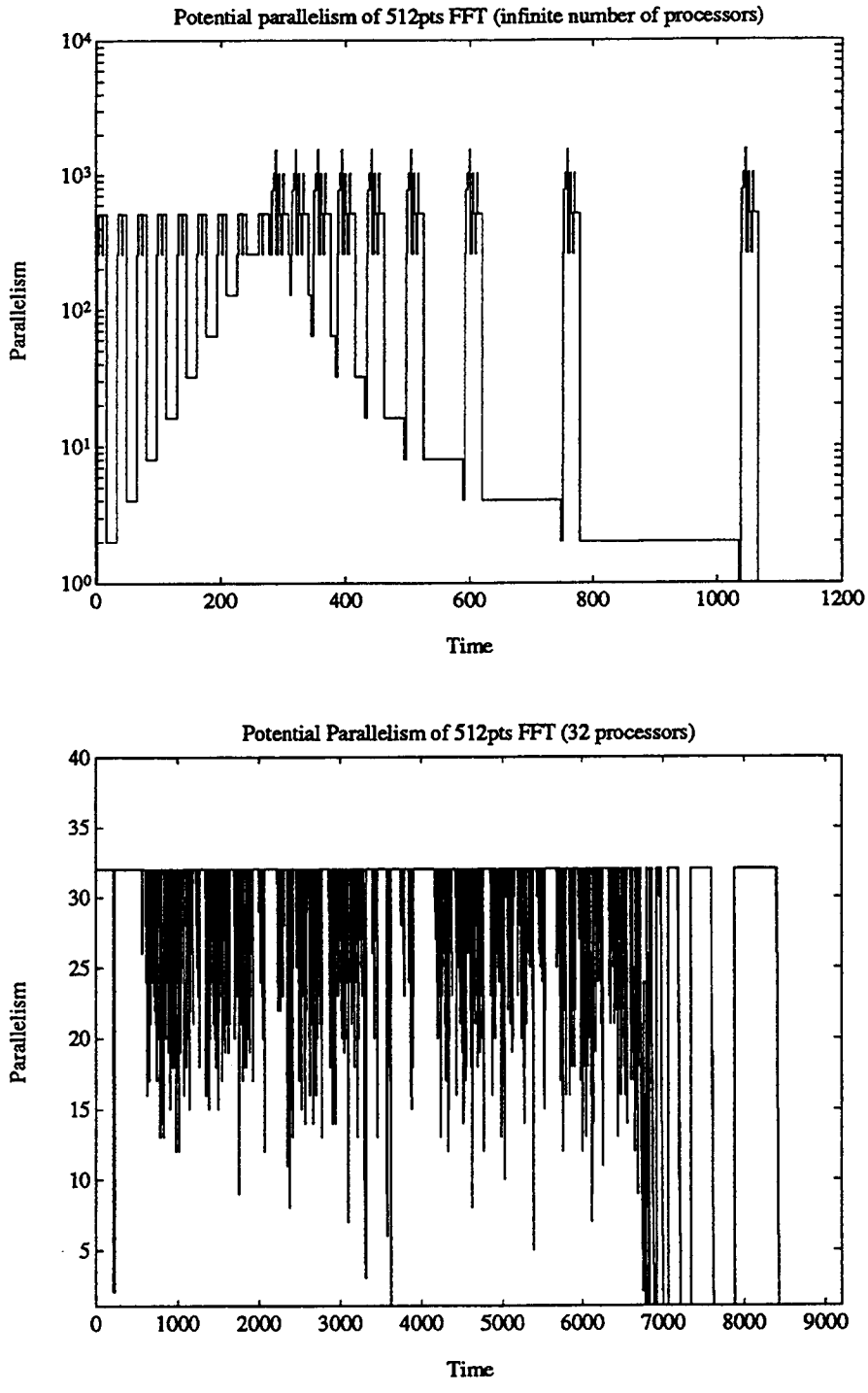


Figure 4: Potential parallelism of FFT

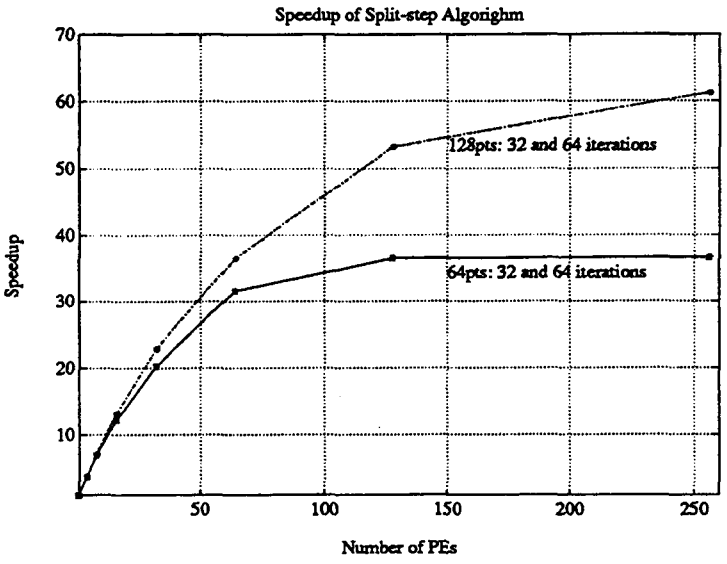


Figure 5: Speedup of *split_step*

4 Implementation of Recursive FFT on the Sequent Symmetry

In the above sections, we have discovered that the potential parallelism of the *split_step* algorithm mainly comes from the recursive-FFT. We also observed that the speedup of the *split_step* algorithm can be achieved solely by the speedup achieved by the FFT. Based on this observation, in this section, we explore the parallel implementation of the recursive FFT on a real parallel machine.

The target machine we chose to run our SISAL implementation of the recursive FFT is the Sequent Symmetry Model 81 shared-memory multiprocessor. This multiprocessor consists of 26 16Mhz Intel 80386 processors with Intel 80387 and Weitek WTL 3167 floating-point co-processors. Each microprocessor has a 64KB cache and is connected to other processors by a bus. The multiprocessor runs the Dynix V3.12 operating system, a multiprocessor version of UNIX.[8]

For our SISAL implementation, we use OSC (Optimizing SISAL Compiler) to generate a portable C-code [4, 10]. OSC mainly targets shared-memory systems, and it performs a certain level of parallelization. However, for our application, where the potential parallelism is mainly obtained by the double recursion, the parallelization of the OSC does not give us a reasonable speedup, since OSC performs loop slicing for parallel execution of the user program. Therefore, we introduced a *parallel function call* mechanism to the original OSC run-time system.

4.1 Overview of the OSC Run-time System for Parallel Execution

The run-time system which supports the parallel execution of a SISAL program provides the following major functions:

1. General-purpose dynamic storage management.
2. Interfaces with the operating systems for process management.
3. Input/output and command line processing.

In this section, we mainly describe how OSC supports the parallel execution of SISAL programs.

The OSC run-time model defines its own *activation record* for each concurrent task(similar to a *process control block* in a traditional operating system) and each task is executed by one of the worker processes. On each processor, the *worker process* is running concurrently waiting for the job to do. There can be a global *ready queue* for all the worker processes, or each worker process can have its own *ready queue* from which it dispatches the next task to do. Upon encountering the code segment[1] which is to be executed in parallel, an activation record is created for the task corresponding to the code segment, and then placed in the global ready queue.[2] After placing a child task into the ready queue, the parent task pursues its execution. When the results are needed, the parent task should wait for the child task. While waiting for its child task, the parent task is suspended, and the worker which was executing the parent task looks for if there is any other task to execute. In this way, the utilization of each worker process is increased and, in some cases, we can prevent a deadlock.

Note that there is no single centralized scheduler in the OSC run-time model. Each idling worker process keeps attempting to dispatch a task, and also any worker process can activate a new concurrent task during its execution of a task.

The general scheme of the parallel run-time model is briefly described as follows:

- In the beginning, create n worker processes on n processors. Only worker 0 continues the execution of the SISAL program while others are waiting for a task to execute.
- Upon encountering the code(function) that should be executed in parallel, the current task builds the activation record for a new subtask, and puts it in the ready queue, and then continues its execution.
- When the result of the function (which was spawned as a subtask) is needed repeat the following:
  1. If the child task has been completed, return to the parent task.
  2. If the child task is not completed, yet, suspends itself and check if there is any other tasks to execute.
- When the execution of the program is completed, shut down the run time system and produce results(if requested). Note that worker 0 is the only process which can start the shutdown procedure for the normal exit of the program.

A loop body is transformed into a C function by the C-code generator of the OSC. Further the m slices of the same loop body are created with different index arguments (Figure 6). A worker, after the initiation of m slices, immediately enters the waiting loop and suspends the current task to execute other tasks if any.

[1] Indeed, the code segment is converted into a C function by the OSC when it is to be executed in parallel.

[2] Alternatively, if we use a separate queue for each worker process, the job is placed into the proper queue according to the allocation policy. Hereafter, for simplicity, we assume there is a global ready queue.

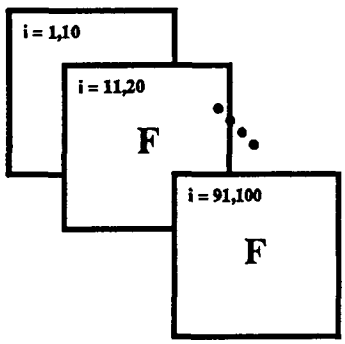


Figure 6: An example of loop slicing

4.2 Exploiting Function Call Parallelism

The Primitives

As mentioned earlier, OSC does not provide a general partitioning scheme in various levels of resolution. Therefore, our application, in which the parallelism is contained in the double recursion, cannot benefit from the parallelization of the current OSC implementation. Hence, we implemented a more general way of exploiting coarse grain parallelism by introducing *parallel function call*.

The following two primitives are used to invoke a function in parallel.

1. **ParallelCall** : Build a new task for a function invocation and returns a pointer to a new task.
2. **ParallelJoin** : Wait until the invoked function is completed.

Here is an example of parallel execution of two functions in the user program.

```
...
t1 = ParallelCall (f1, arg1, arg2, ...);
t2 = ParallelCall (f2, arg1, arg2, ...);
...
ParallelJoin(t1);
ParallelJoin(t2);
```

In the above example, functions `f1` and `f2` are called in parallel. The caller continues its execution until it reaches `ParallelJoin()`. The `ParallelJoin()` should be called before the result of the corresponding function call is needed. The caller itself, while waiting for the called function to be done, tries to execute another task if any.

Modification to output of the OSC

We still use OSC to generate a C-code from a SISAL program. After generating a C-code, we have to reorder the program statements to exploit the parallel function call. Consider the following code segment:

```
...
r1 = f1 (...)
x1 =  ... r1 ...
..
r2 = f2 (...)
x2 = .... r2 ...
...
```

If function `f1` and `f2` are independent and can be called in parallel, the above code should be reordered as follows:

```
...
r1 = f1 (...)
r2 = f2 (...)
...
x1 =  ... r1 ...
x2 = .... r2 ...
...
```

Further, using `ParallelCall` and `ParallelJoin`, the final code would look like:

```
...
t1 = ParallelCall (f1,...,r1)
t2 = ParallelCall (f2,...,r2)
...

ParallelJoin (t1)
x1 =  ... r1 ...
ParallelJoin (t2)
x2 = .... r2 ...
...
```

As can be seen in the above example, to fully exploit the function call parallelism, we have to provide a mechanism to re-order the statements of the target language. However in a real implementation, the re-ordering is implicitly done in the partitioning step. The code generator will produce the correct parallel code according to the partitioned data-flow graph. In the example above, if the partitioning is done properly, the lines calling `f1` and `f2` should be in different partitions. The code generator can therefore produce the correct code to execute these two partitions in parallel.

4.3 Performance Measures on the Sequent Symmetry

speedup

The execution time of the recursive FFT is shown in Table 2. In this table, we also showed the result obtained by the parallelization of the original OSC, *i.e.*, loop slicing. As shown in the table, for our recursive FFT, parallel function call is much more effective than

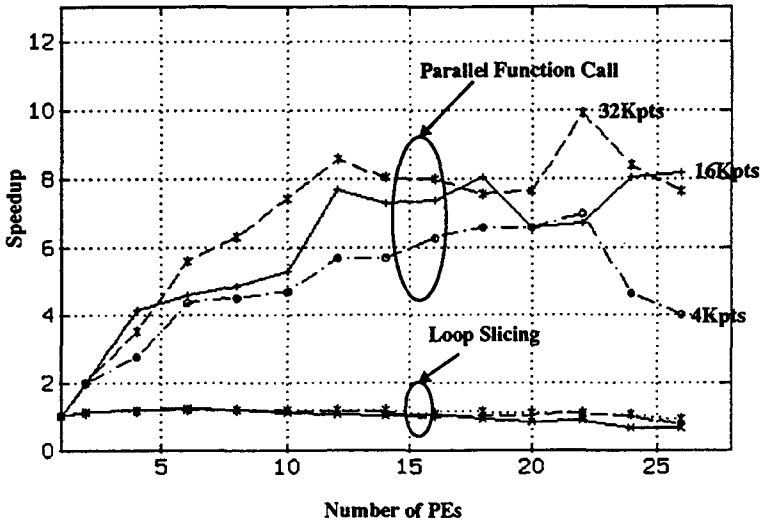


Figure 7: Speedup of the recursive FFT on the Sequent Symmetry

loop slicing. The corresponding speedup is also shown in Table 3 and in Figure 7.

Comparison with other sequential machines

We have run the same program on the following platforms:

1. HP : HP9000-425t with 25Mhz 68040 and a proprietary floating-point co-processor running HPUX.
2. SUN4 : Sun Sparc system 4000 running SUN OS.
3. SUN3 : Sun3/100 with M68881 floating point processor.

The execution time is shown in Table 4.

To compare the general performance of each machine, we also presented the execution results of the *Whetstone* and the *Dhrystone* in Table 5 and Table 6. The *Whetstone* is used to test the floating-point performance while the *Dhrystone* is used to test mainly the performance of general system calls.

Comments on the results

As shown in the results, parallel function call is very effective way of exploiting parallelism incurred by recursion. However, there is a big gap between the speedup we obtained on the Sequent Symmetry and the ideal parallelism observed by the simulation. We can summarize the possible causes of the gap as follows:

1. The simulated execution exploits the fine-grained parallelism at the IF1 instruction level while we exploit the coarser grain parallelism at the user-defined function level.
2. Synchronization bottleneck : With a large number of processors, we encounter a noticeable synchronization overhead. In our case, with the centralized task

	Loop Slicing			Parallel Function Call		
#PE	4096pts	16384pts	32768pts	4096pts	16384pts	32768pts
1	7.16	39.58	96.69	6.81	38.07	93.11
2	6.30	35.31	88.91	3.44	19.04	46.35
4	5.96	33.56	82.93	2.50	9.26	26.74
6	**5.85**	**32.73**	81.15	1.56	8.36	16.63
8	6.02	33.21	81.15	1.52	7.90	14.77
10	6.44	33.05	**80.81**	1.46	7.24	12.56
12	6.68	34.12	81.55	1.20	4.97	10.84
14	7.00	34.52	82.84	1.20	5.23	11.57
16	6.81	41.78	87.08	1.09	5.18	11.65
18	7.81	38.46	85.61	1.04	4.74	12.34
20	8.62	39.80	89.44	1.04	5.80	12.20
22	8.33	36.62	87.95	**0.98**	5.69	**9.39**
24	10.75	38.46	91.99	1.47	4.75	11.12
26	10.91	50.16	102.79	1.71	**4.66**	12.23

Table 2: Execution time of recursive FFT on the Sequent Symmetry

	Loop Slicing			Parallel Function Call		
#PE	4096pts	16384pts	32768pts	4096pts	16384pts	32768pts
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.14	1.12	1.09	1.98	2.00	2.01
4	1.20	1.18	1.17	2.72	4.11	3.48
6	**1.22**	**1.21**	1.19	4.37	4.55	5.60
8	1.19	1.19	1.19	4.48	4.82	6.30
10	1.11	1.20	**1.20**	4.66	5.26	7.41
12	1.07	1.16	1.19	5.68	7.66	8.59
14	1.02	1.15	1.17	5.68	7.28	8.05
16	1.05	0.95	1.11	6.25	7.35	7.99
18	0.92	1.03	1.13	6.55	8.03	7.55
20	0.83	0.99	1.08	6.55	6.56	7.63
22	0.86	1.08	1.10	**6.95**	6.69	**9.92**
24	0.67	1.03	1.05	4.63	8.01	8.37
26	0.66	0.79	0.94	3.98	**8.17**	7.61

Table 3: Speedup of recursive FFT on the Sequent Symmetry

#Pts	Symmetry(1PE)	HP	SUN4	SUN 3
4096	6.81	1.92	0.90	19.48
16384	38.07	10.27	4.56	92.38
32768	93.11	17.87	7.38	185.04

Table 4: Execution time of FFT on the various sequential machines

Symmetry(1PE)	HP	SUN4	SUN 3
5.35	2.05	0.91	23.88

Table 5: Time to spend to perform 10 Million Whetstone instructions.

Symmetry(1PE)	HP	SUN4	SUN 3
5263	25000	33333	3448

Table 6: Number of Dhrystone instructions executed per second

queue, the mutual exclusion on the task queue causes even much more overhead.

3. Storage Management Overhead : Our FFT uses a fairly large amount of storage space and each instance of FFT call has its own copy of input. Therefore, the dynamic creation and release of such storage space costs a lot.

We can reduce the overhead by several ways:

1. Decentralized task scheduling - especially for distributed-memory multiprocessors.
2. Optimizing *ParallelCall* : Many of the run-time optimizations can be embedded in the implementation of *ParallelCall*. For example, an efficient allocation scheme including load balancing and cache coherency can greatly reduce the overhead incurred by the parallel function calls.
3. To employ a scheme to share the same storage space for each instance of the parallel function call - especially for shared-memory multiprocessors.

5 Concluding Remarks

In this paper, we showed how the SISAL programming environment can be utilized to implement a signal processing application in a multiprocessor system. We also presented that the parallel function call is a very efficient way of exploiting coarse grain parallelism. Especially, for our recursive FFT implementation, we could achieve much better speedup with parallel function call than with loop-slicing (the current parallelization scheme of OSC). Indeed, for some applications, loop-slicing does an adequate job of parallelizing a program. Therefore, it would be important to have some elegant partitioning scheme to determine which kind of parallelism should eventually be exploited. Our next step will be to explore automatic partitioning schemes.

For more general and useful parallel programming models, we will consider the following areas for future research:

1. Implementation on a distributed memory system.
2. Implementation of an efficient run-time system.

3. Implementation of an automatic partitioning scheme for various grain size.
4. Implementation of a comprehensive parallel programming environment which includes compiler, debugger and profiler in a graphical user interface.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS Spring Joint Computer Conf.*, pages 483–485, Atlantic City, N.J., April 1967.

[2] D. Cann and J. Feo. Sisal versus FORTRAN: a comparison using the livermore loops. Technical Report (Unpublished), Lawrence Livermore National Laboratory, 1990.

[3] David Cann and R. R. Oldehoeft. A guide to the optimizing Sisal compiler. Technical Report UCRL-MA-108369, Lawrence Livermore National Laboratory, Sep. 1991.

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

[5] G. D. Dockery. Modeling electromagnetic wave propagation in the troposphere using the parabolic equation. *IEEE Transactions on Antennas and Propagation*, 36(10):1464–1470, October 1988.

[6] Douglas F. Elliott and K. Ramamohan Rao. *Fast transforms: Algorithms, Analyses, Applications*. Academic Press, 1982.

[7] J. R. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. R. W. Glauert, I. Dobes, and P. Hohensee. Sisal : Streams and iteration in a single assignment language – language reference manual version 1.2. Technical Report TR M-146, Lawrence Livermore Laborartory, March 1985.

[8] Mathematics and Computer Science Division of Argonne. Using the Sequent Symmetry S81. Technical Memorandum 139, Argonne National Laboratory, 1990.

[9] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes*. Cambridge University Press, 1986.

[10] Vivek Sarkar and David Cann. POSC — a partitioning and optimizing Sisal compiler. Technical report, Lawrence Livermore National Lab., 1990.

[11] 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.

SISAL and Von Neumann-Based Languages: Translation and Intercommunication+

C. Yoshikawa , U. Ghia* and G.A. Osswald

Department of Aerospace Engineering and Engineering Mechanics
* Department of Mechanical, Industrial and Nuclear Engineering
Computational Fluid Dynamics Research Laboratory
University of Cincinnati, Cincinnati, OH 45221, USA

1. MOTIVATION

SISAL is a functional programming language useful for creating programs that can be easily exploited for parallelism, run on a wide range of platforms (Cray, Iris, IBM RS/6000, etc.), and run at serial speeds competitive with FORTRAN (Ref.1). In order to take advantage of SISAL's speed and parallelism, however, it is necessary to extensively rewrite existing FORTRAN code or create new code in SISAL. Translating from a language such as FORTRAN to SISAL is not particularly easy, since SISAL's data is comprised of semantically immutable values. Therefore, it was decided to test the performance of a combination of a FORTRAN driver and a SISAL subroutine. Consequently, extensive rewriting of the original FORTRAN code is not necessary and the program performance can be expected to increase due to SISAL's decided advantages (Ref.2). Since documentation of a FORTRAN/SISAL interface is not widespread and the corresponding performance is not well understood, what data should be examined to evaluate the resulting performance is itself an area of investigation. However, the experience and information acquired concerning the functional and Von Neumann language interface should prove to be didactic, at worst. The present paper attempts to identify the approach needed to translate code and create the interface, while detailing the communication between a FORTRAN driver and a SISAL subroutine. Also, the performance gain or loss will be reviewed.

2. APPROACH

2.1 Introduction to FORTRAN Code

To develop the desired procedures and code interface, the FORTRAN program chosen is a 2-D flow code, called CN4DFC, that computes several variables, such as

+ This research is supported, in part, by an NSF REU Internship and, in part, by CRAY Research, Inc. Supercomputer time was provided by the Ohio Supercomputer Center and Lawrence Livermore National Laboratory.

vorticity, stream function and pressure, for flow around an airfoil. The subroutine OPG, which solves for the stream function, was chosen to be converted to SISAL, because it solves an elliptical partial differential equation using a direct method and, hence, accounts for a majority of the program 'work'. According to the Cray performance utility Flowtrace, the subroutine OPG accounts for 63% of the total workload of the program. Using Amdahl's law, this should realize a speed-up factor of at most 2.2 for 8-processor execution. While it is true that the subroutine OPG manipulates a large array of (461x129) data points, this is not the reason for its large workload; the large number of times the subroutine is called is the reason that it does the majority of the work. This was recognized to be a possible burden to the performance since mixed-language interfaces are deemed 'expensive' (Refs.3 and 4).

The program, CN4DFC, computes stream function (variable name *psi*), vorticity (*omega*), and surface pressure coefficient (f_{cp}) for a flow around a cambered airfoil. The subroutine, OPG, solves for *psi*. Basically, it takes the source term of a matrix equation and does the following operations on it:

(1) conversion to block Gaussian elimination plane(BGE),
(2) forward elimination,
(3) back substitution,
(4) insertion of boundary conditions,
(5) conversion back to computational plane.

The computational plane is converted to the BGE plane in order to create a tightly banded tri-diagonal matrix. This aids in enhancing the performance of the subroutine.

The OPG subroutine also updates the vorticity array, OMEGA, near the airfoil surface.

2.2 Specifics of Translation to SISAL

Attempt was made to retain the original program specifics, such as using the same variable names, array indexing techniques, and program structure. However, certain Von Neumann intrinsic techniques had to be removed or altered. The step of passing temporary work arrays to the subroutine had to be removed; SISAL deals with semantically constant variables; use of temporary arrays would have required sequential FOR LOOPS in order to reference 'old' values - this would have degraded performance. Also, the technique of updating array index variables after each section of code had to be changed. Again, variables in SISAL are kept semantically intact; the indices had to be computed based upon FOR LOOP variables and hand calculations.

A large amount of data altering is present in this routine; separate variables are necessary to store the current contents of the variable 'psi', after each of the above five operations. To make the OPG subroutine more flexible and to maximize concurrency and contiguous array concatenation, steps (4) and (5) above were combined. This

eliminated the need for a separate array for boundary condition results and the resulting sequential FOR LOOP to update non-sequential values in the *psi* array.

The most difficult part of the program was working around SISAL's basic notion of semantically intact arrays. Thus, some program alteration had to be done to accommodate this fact and build arrays contiguously. The modifications are listed below.

(1) The back substitution results were flipped both vertically and horizontally to work around the fact that SISAL does not yet support 'reverse concatenation' type of packaging for the results.

The back-substitution results were placed directly into arrays after computation of each value. Because of the need for back-substitution, it was necessary to place the new values in the following new order: each column of data was flipped upside down, and also the order of the columns was reversed. It should be noted that only the inversion of the column order was warranted, since the back-substitution propagates from the end to the beginning. The flipping of each individual column was optional, but was in fact implemented so as to ease the translation process from FORTRAN to SISAL; the original FORTRAN program computed the data in the same 'upside-down' manner.

(2) Creation of the BGE-results array in a new order to allow for proper concatenation.

Because of the changes described in (1) above, the required transformation back to the computational plane had to be modified in the SISAL source code since the BGE array now held data in a manner different from that originally used in the FORTRAN code.

(3) Filling in points at 'infinity' explicitly with zero to allow for a contiguous array.

In FORTAN, the points that are recognized to be infinitely far away from the airfoil are left unaffected and filled in later with the prescribed boundary values. In SISAL, however, these 'holes' could not be ignored. They had to be set to zero (arbitrarily) and later filled with the proper boundary values.

(4) Extended use of the parameter NC1EVN to remove any possibility of creating an array with redundant data.

When the data was flipped back to the computational plane in the original FORTRAN code, the program would overwrite one single column at the middle of the plane. This is due to the fact that the BGE transformation required two equal sized portions of data to fill in the upper and lower half of the BGE plane. Only if the number of horizontal mesh cells of data was odd (NC1EVN=TRUE) in the computational plane, the FORTRAN program would make a copy of the middle column and give it to both the upper and lower BGE halves, making them have an equal number of columns. When

this data was written back to the computational plane, this redundant column was written twice - once for each half of the BGE plane. Concatenation of the data in the SISAL subroutine caused an extra column to be placed in the center of the plane, rather than that column being overwritten, as was the case in FORTRAN. Therefore, the parameter NC1EVN was tested within the SISAL subroutine to prevent this error.

These changes reflect the amount of effort put forth to eliminate any data copying and to preserve array building.

2.3 FORTRAN/SISAL Interface

The translation from FORTRAN to SISAL involved some minor changes within the calling program, CN4DFC, to accommodate the interface. These included:

(1) Configuring, starting and stopping of the SISAL run-time system,
(2) Alteration of the argument calling list to include array and result descriptors,
(3) Creating an integer variable NC1EVEN to overcome the inability of SISAL to accept Boolean values from an interface.

Since Boolean values cannot be passed to SISAL through a mixed language interface, a new variable entitled NC1EVEN was created in the calling routine, CN4DFC, to replace the Boolean value NC1EVN.

Array descriptors were obviously necessary for all non-scalar arguments and results. To conserve memory, the *psi* and *omega* results were returned back into their original array memory locations. Identical data descriptors were used to describe the argument and result arrays of *psi* and *omega*. If non-matching data descriptors were used for array and result descriptors, then errors resulted because it is not possible to send and return different parts of the same array. The SISAL subroutines SSTART and SSTOP were called only once, rather than for each call of OPG, to eliminate overhead. This change also corrected another error. If SSTART and SSTOP were repeatedly called, then the S.INFO file, which holds diagnostic and performance data, would become very large and result in an error because of its size.

The initialization process for the SISAL run-time-system requires that a certain amount of memory be allocated for the SISAL program. This number of bytes turned out to be difficult to estimate. It was not obvious as to how much 'work-space' would be needed and whether the PSI, OMEGA and FCP arrays would be 'owned' by FORTRAN and not needed to be copied by SISAL. Hence, a 'binomial-estimate' was generally used, starting at a maximum of 20 megabytes, followed by trial and error, to arrive at the final memory size. It was also observed that the larger the number of processors requested, the more the memory used. For example, if 8 processors were called for, it was impossible to allocate 20mB on the Cray for the SISAL RTS. However, if the number of processors was reduced to 6 or even 7, the code would run with no error regarding memory. This leads the authors to believe that the amount of

private memory for each processor multiplied by the total number of processors yielded an excessive allocation request for memory.

2.4 Debugging the SISAL Code

During the debugging process, aside from concerns of syntax, the maximum number of errors were located using the -bounds option with the Ohio Supercomputer (OSC) compiler. This option informs the user where and what type of out-of-bounds array access occurred. Another useful tool was the DEBUG function callable from within any SISAL code. This subroutine prints out its argument's value, name, and type. With the -glue option, OSC will leave the debug commands in place rather than moving them for better code efficiency. Because of the limitations of FIBRE, a separate FORTRAN subroutine would have had to be written and interfaced to output data for error checking if DEBUG were not available.

3. RESULTS

3.1 Validation of SISAL Code

The SISAL subroutine proved to yield the same results as the original FORTRAN subroutine. Tests were run on a smaller grid (63x21) for three cases:

(a) FORTRAN program on one processor,
(b) SISAL/FORTRAN program on one processor, and
(c) SISAL/FORTRAN program on multi-processors.

The raw data files were compared for accuracy. The SISAL multi-processor results and the SISAL uni-processor results were identical, leading to the conclusion that there were no data dependencies that were not expressed explicitly via the sequential FOR LOOP in the SISAL subroutine. The SISAL runs and the FORTRAN runs had some differences, as shown in Tables 1-3. These differences were not a consequence of programmer error, but rather small precision errors resulting from different run-time sequences of arithmetic operations between the SISAL and FORTRAN programs. After significant checking, the error was traced to the variable *dgamma*. This variable was computed to be 0.0e0 from a sum of numbers of equal magnitude but opposite sign in the FORTRAN program, while the SISAL program, resulted in a value of 1e-11 for this variable, because of a different sequence of additions. This error is minimal, considering that the original values were of the order of 10e3, and the program was executed in single precision on the Cray. Because of the structure of the program, the error was transported to each time step and sometimes magnified slightly. Even though the error is unacceptable for aerospace engineers, the program was found to correctly compute the solution to the problem being solved. The error could be reduced or eliminated by using double precision to compute *dgamma*, but that was not considered necessary for the present purpose.

3.2 Performance

The performance of the SISAL/FORTRAN program was compared with that of previous runs of the FORTRAN program for 100 time steps, for Re=45000, and Δt=5e-4. SISAL can be executed with any number of processors (limited by the hardware) as specified by the user. After some initial test runs, six processors were chosen, to optimize parallelization overhead and unnecessary gathering of processors. Because of the degradation in speed caused by the insertion of the interface, the results were not as good as expected (see Table 4). Runs on the small (63x21) grid reveal a general decrease in performance for both the FORTRAN and SISAL/FORTRAN programs, due to the higher ratio of overhead/workload, since there were not significantly many floating-point operations being performed. The difference in performance is viewed as a result of a specific feature absent from SISAL thus far that does not allow it to perform in a busy environment.

Autotasking, available on the Cray supercomputers, is able to use a variable number of processors for a section of code based upon both the workload of the system and the amount of work in the loop. User-given hints about the loop specifics, such as iteration counts, can aid the compiler in determining this number. The limited experience of the authors suggests that SISAL cannot be coaxed into using anything but the number of processors indicated on the command line for SISAL programs, or the SSTART parameter for mixed programs with FORTRAN or C drivers. Thus, there is unnecessary overhead incurred when the code attempts to use 8 processors on a loop which can effectively use only four processors. It would be desirable if the SISAL language would incorporate a more effective run-time system or have compile time options, whereas the user could specify the number of processors to use for different loops.

Also deemed necessary for the novice user are performance tools. It was difficult for the authors to obtain the wall-clock timings and flopcounts of the SISAL code. The -fflopinfo option of OSC proved to yield contrary results when different numbers of processors were requested. This option was little documented, and the reason for the results was unknown. The only timing function of SISAL is the -time option of OSC. This, however, only returns CPU time and not wall-clock time.

4. CONCLUSION

The present SISAL/FORTRAN interfacing effort did not yield the performance results expected, but it did provide an opportunity for gaining experience and knowledge. The present paper can be used as a reference for information or help on interfacing the two languages. A performance increase is expected by 'inlining' all surrounding code which is included in the time-marching step iteration process. This will remove the cost of the interface by making at most 2 calls to a non-FORTRAN subroutine (an initialization call and all other time-stepping calls).

5. REFERENCES

1. Cann, David C., "SISAL 1.2: A Brief Introduction and Tutorial," Computing Research Group, L-306, Livermore, CA 94550.

2. Cann, D., "Retire FORTRAN: A Debate Rekindled," submitted to Journal of Communications of the ACM, March 1992.

3. Cann, David C., "The Optimizing SISAL Compiler: Version 12.0.," Computing Research Group, L306, Livermore, CA 94550.

4. McGraw, J., Allan, S., Oldehoeft, R., Glauert, J., Kirkham, C., Noyce, B., and Thomas, R. "SISAL: Language Reference Manual Version 1.2.," Livermore, CA, March 1, 1985.

5. Osswald, G. A., Ghia, K. N. and Ghia, U., " A Direct Implicit Methodology for the Simulation of the Rapid Pitch-up of an Airfoil with and without Leading-Edge Flow Control," presented at 13th International Conference on Numerical Methods in Fluid Dyn

Table 1: Percentage Difference in Solution Function PSI

Time Step	Maximum Difference	Average Difference
1	0.00	0.00
2	0.00	0.00
3	1.955e-2	6.068e-5
4	1.047e-3	2.006e-4
5	4.026e-4	5.63e-5

Table 2: Percentage Difference in Source Function OMEGA

Time Step	Maximum Difference	Average Difference
1	0.00	0.00
2	5.717e-14	4.32e-17
3	2.24e-2	3.235e-5
4	2.98e-2	1.35e-4
5	2.426e-2	2.62e-4

Table 3: Percentage Difference in Surface Pressure Coefficient FCP

Time Step	Maximum Difference	Average Difference
1	1.3380	0.49049
2	0.00	0.00
3	1.014e-2	1.61e-4
4	2.78e-2	4.421e-4
5	2.15e-13	5.295e-15

Table 4: Performance of CN4DFC Code with SISAL and FORTRAN Versions of OPG Routine.

	FORTRAN code	SISAL code
CPU Time (seconds)	709.835	2439.745
Wall-Clock Time (seconds)	239.441	422.433
MFLOP/CPU Time	99.85	29.01
MFLOP/Wall-Clock Time	295.00	167.22

An IF2 Code-Generator for ADAM Architecture

Srdjan Mitrovic
Institut fuer Technische Informatik und Kommunikationsnetze, ETH Zuerich
ETH-Zentrum, CH-8092 Zuerich, Switzerland

1. Introduction

Project ADAM[1] is based on research work in the area of parallel computers. ADAM architecture can be described as von Neumann/dataflow hybrid or coarse-grain dataflow architecture. The project encompasses development and analysis of a novel parallel architecture and corresponding tools. A register-level simulator allows quick changes of design parameters and a detailed monitoring of machine behavior. Programmability of architecture can be proved only by running existing programs written in high-level languages. Therefore, research results are obtained with an architecture simulator and code-generators for high-level languages.

The detailed simulator called Metamachine was implemented and a processing element was realized in hardware. Software environments, new languages for parallel processors and a code-generator for applicative language Sisal have been programmed by members of ADAM project. The project involved seven people over a period of five years. The research work resulted in various software tools and a number of experiments that verified the performance and illuminated the problems of ADAM architecture.

This paper presents one part of ADAM project. It describes the code-generator that uses IF2 intermediate form to generate assembler code for Metamachine. IF2 is generated by optimizing SISAL compiler system (OSC) [FeoCannOldeh90]. The author assumes that the reader has a knowledge of Sisal and its syntax [McGrSkedz85]. IF1 [SkedzGlau84] and IF2 [WelSkeYaRan86] graphs are hierarchic acyclic dataflow graphs that are stored in text files.

---

[1] ADAM is the abbrevation of Advanced DAtaflow Machine

2. Architecture and Simulator

Main goal of ADAM architecture is to achieve a programmable and scalable parallel machine. A programmable parallel machine need to remove the burden of knowledge about distribution of work and data from the programmer and the code-generator. The scalability means that high access latency caused by increased distance between data and accessing instructions needs to be tolerated. These requirements lead to following characteristics of ADAM architecture:

- dynamic load distribution
- distributed shared memory or global address space
- split-transactions and synchronization with present- and wait-bits.
- fast context-switch

ADAM simulator is called Metamachine[Maquelin93]. It is used as target machine for code-generators. Until now, code can be generated from high-level languages MFL [Murer92] and Sisal [Mitrov90] [Mitrov92]. Metamachine is a detailed simulator that models all aspects of ADAM machine. The simulation detail is at the level of register-transfer operations. The duration of a single instruction is not predictable because of the complexity of simulation. Any best cycle-time for an operation can be delayed by the locked resource. The delays can be generated by busy internal buses or cache misses. Figure 1 shows internal structure of one processing element of Metamachine.

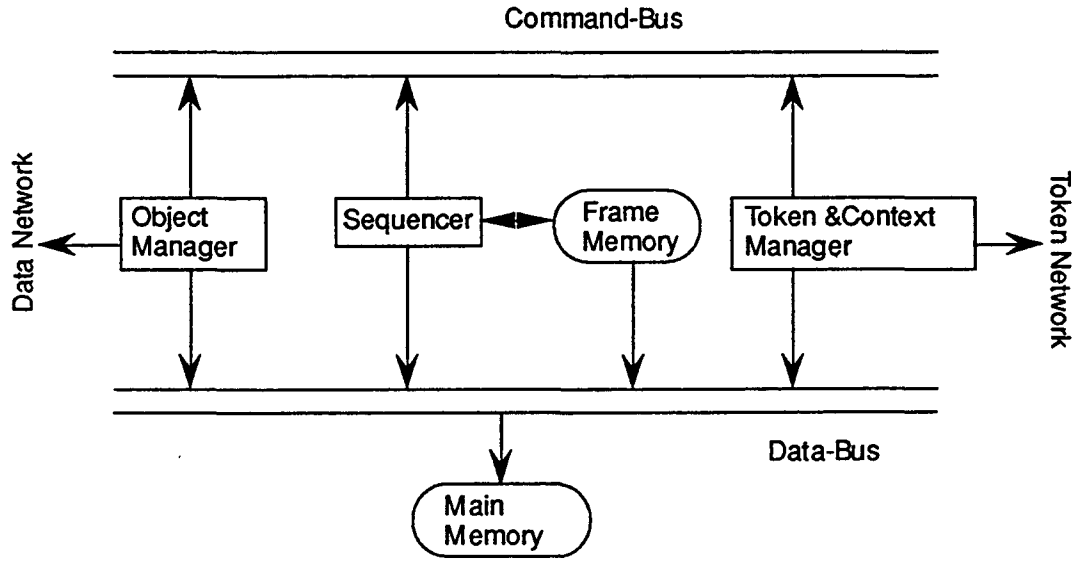


Figure 1: Internal structure of one processing element

One processing element consists of several asynchronous units. However, they may influence each other because they share buses internally. An overloaded unit may hinder the work of the others. Processing elements are connected with two configurable network topologies. Data network exchanges computation data and token network exchanges work between processing elements.

Figure 1 defines two types of memories inside a processing element. Frame memory is similar to a cache and is used for holding frames. A frame is attached to every initiated task. In ADAM machine, the tasks are called codeblocks. A codeblock has the size of a function or the body of a loop. It consists of a sequence of instructions. The instructions reference frame-content directly. The frame has a constant size of 32 words. Every frame word has a present- and a valid-bit. An instruction suspends its execution when it reads from a frame-register with cleared present-bit. The suspension of an instruction leads to the suspension of running codeblock. Frames hold only scalar values and build the synchronization space of Metamachine. Main memory holds structured data or more precisely the objects. Their contents are accessed with split-transactions using object-reference and an index. Figure 2 displays the accesses to frame and main memory.

Reading and writing to frame:

[4] := ADD([3], [2]) % reads scalar values from frame registers (FR) three and two and writes into FR four

Reading from the main memory:

[10] := LD([12], [1]) % reads object reference from FR 12 and the index from FR one; the request is send and the result will be written in FR ten.

Figure 2: Accessing main and frame memory

The same instruction accesses local and remote main-memory. The object-reference specifies the exact location of the indexed element. The location is defined by the processing element where the object has been allocated or by the type of object. The objects can be either local, replicated or interleaved. Figure 3 shows the differences between objects.

	Location of content is ...
local	...on same processor where the allocation instruction is issued
replicated	...on every processor
interleaved	for element with index *i* on a machine with *p* processors: ...on the processor *(i mod p)*

	Advantage	Disadvantage
local	enhances data locality when reader/writer on same processor	produces request bottleneck when accessed by many active readers
replicated	high locality for read operations	generation in parallel too slow, uses much memory
interleaved	removes request bottlenecks	low locality, not optimal implementation

Figure 3: Differences between various object classes

High data locality means that network traffic decreases. With low data locality, the network bandwidth decreases the computation rate of the parallel machine. Data locality is enhanced when using local type while writing into objects. It can lead to severe request bottlenecks if the reading parallel tasks are not processed on the same processor. Every store- and load-access to objects is synchronized. The processor that hosts the local object and the reading codeblocks will be blocked until requests are processed.

Interleaved objects were introduced because of the request bottlenecks. With interleaved objects, the requests are distributed in the same way in which the content is distributed. Every request is sent to the processor where the content element is located. Interleaved objects cause high network-traffic and require a well-balanced network and sequencer performance but remove request bottleneck.

Replicated objects are slower to generate and consume a lot of memory. Their advantage is excellent data locality for read accesses. Their usage generates request bottlenecks if several parallel tasks create replicated objects. In such case, object managers of the parallel machine will be overloaded, and will block other parts of the machine. Replicated objects can be used for all objects that are created in sequential part and that are read in the parallel part of the program. Execution code of the program is loaded in Metamachine as replicated object. An evaluation of various object classes is given in [MurerFaerb92].

The instructions of Metamachine are either of synchronous or of asynchronous type. Synchronous instructions read from and write to frame-registers. Asynchronous instructions read from frame-registers and clear the present-bit of the result frame-register. They initiate a request that will write back into the frame-register with the cleared present-bit. The instruction is terminated without waiting for the request to complete. This means that the instructions that follow an asynchronous instruction can be executed as long as they do not need results from that instruction. The split-transactions and call instructions are of the asynchronous type.

There are two types of call instructions and both are non-blocking. Every call must be synchronized, otherwise there is no guarantee that called codeblocks will be executed at all. CALL instruction is used for initiating codeblocks that represent Sisal functions. PCALL instruction starts a number of parallel codeblocks. Parallel codeblocks have identical code and vary only in a counter parameter. A parallel codeblock represents a slice of a parallel loop.

An adequate dynamic load-distribution (DLD) mechanism for a scalable parallel machine must fulfill the following requirements:

a) have a distributed mechanism

b) distribute the work evenly and quickly

c) throttle the parallelism.

d) have high execution locality

A scalable parallel machine requires a distributed DLD mechanism. Any central monitoring of workload would decrease the scalability. The even distribution of workload is important for irregular loop and recursive parallelism. This can be done only with dynamic load-distribution because workload is known only at run-time. Throttling of parallelism is necessary so that resources of the machine are used efficiently. Execution locality means that the caller and the called codeblock execute on the same processor. Thus, they exchange arguments and results without network access. The effect of ADAM DLD is that execution locality increases only after the work has been distributed. The workload consists of tokens. A token represents a codeblock that has been called but that was not yet assigned to a processor. A processor that has too little work fetches a token from token network and expands it locally. The token expansion assigns a frame to codeblock and sets it to ready state. A processing element should have at least two ready codeblocks so that it can do context-switch when the running codeblock suspends its execution. A detailed explanation of DLD mechanism is given in [Maquelin90].

The realistic simulation is important for code-generation research because many problems occur just because of the resource bottlenecks. Most problems and solutions of code-generation for Metamachine will be valid for hardware implementation of ADAM architecture. The details of the simulator are explained in Ph.D. thesis of O. Maquelin [Maquelin93].

3. Code Generator

The implemented code generator is called ITH, an IF to HLA translator. It reads hierarchical acyclic dataflow graphs and generates a list of codeblocks. The main goal of ITH is to generate efficient code from programs written in Sisal. Good programmability of a machine means that the programmer is not punished for using high level abstractions and that he need not know the structure of the machine. The existing OSC code-generators for conventional parallel machines already achieve this goal because of memory and update-in-place analysis of OSC [Cann92]. Therefore, ITH uses most optimizations from OSC backend. Another goal is to generate adequate code regardless the scale of the machine. This comprises that parameters of OSC and ITH should not be varied by the programmer and that the knowledge about machine size should not be used in code-generation. Certainly, the knowledge about the number of processors would produce a better code for single-user parallel machines running simple programs. The losses when this knowledge is dismissed will be discussed and measured. Figure 5 displays the way ITH is attached to OSC.

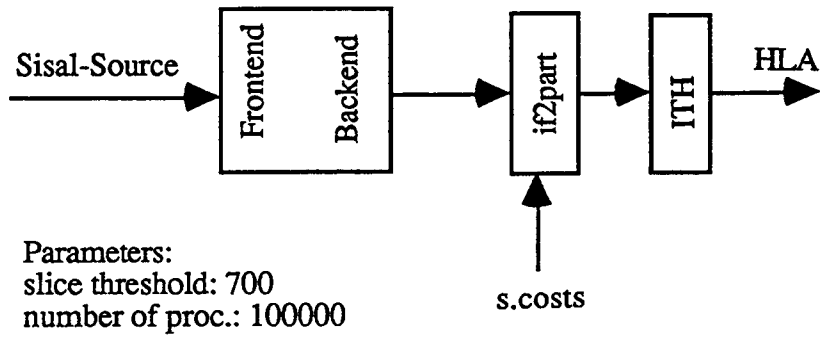


Figure 5: OSC & ITH

OSC parameters that are important for ITH are the slice threshold and the number of processors. Both are used by the partitioner pass "if2part" to decide which product-form loop should be run in parallel. Because the execution models of OSC and ITH are

different, the number of processors must be assumed to be large. Metamachine needs several ready tasks so that it can efficiently hide latency. The parameters are set to values given in figure 5. Another option of OSC specifies what functions or graphs should be inline expanded. OSC cannot do inter-graph analysis. Therefore, best code is generated when all graphs or functions are inlined. Only recursive functions cannot be inlined.

ADAM code-generator reads IF2 files. An earlier version was based on IF1 as input. IF1 is clear and well documented. This approach has been dismissed because of the optimizations that lead to IF2 graphs. The update-in-place analysis and memory preallocation were main reasons why we changed from IF1 to IF2. The lack of IF2 documentation was compensated by the availability of ADAM graph browser [MitroMurer91]. This graphical tool is integrated in ITH. Figure 6 displays a graph generated by the graph browser. Manually added descriptions explain naming conventions used for IF2 graphs.

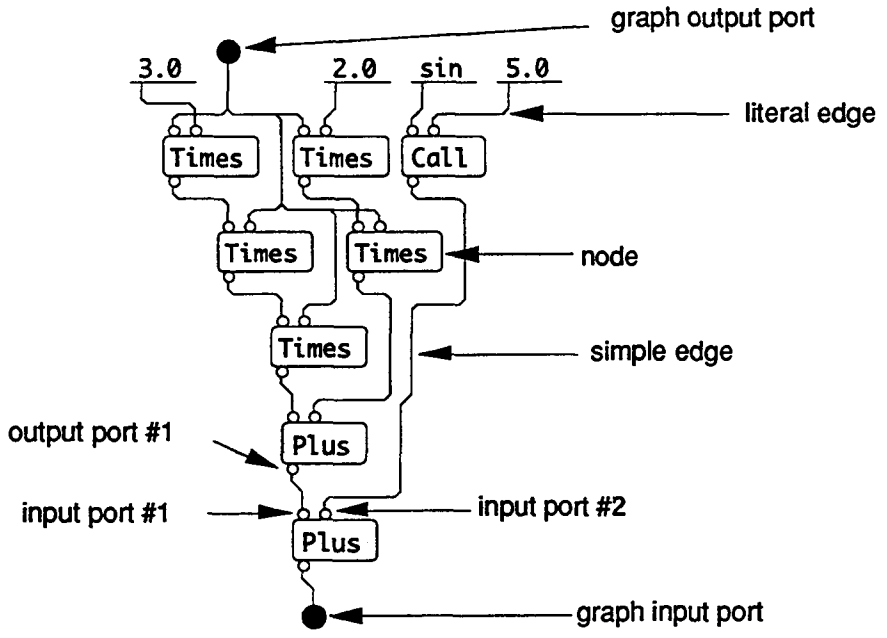


Figure 6: The naming conventions in an IF2 graph

IF2 parser is the same for ITH and the graph browser. It is written in object-oriented way using P1 Object Modula-2. The class hierarchies for hierarchical acyclic dataflow graphs are given in figure 7.

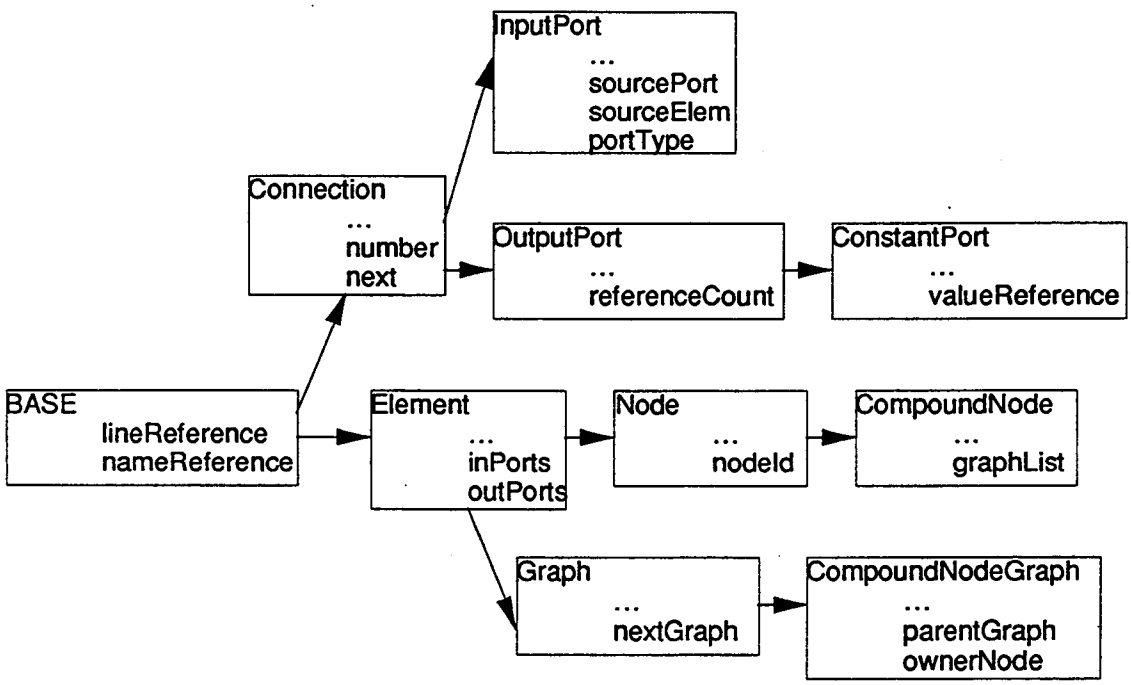


Figure 7: Class hierarchy for IF2 and IF1 graphs

3.1 Sequentialization

Dataflow graph defines partial order of its nodes. The set of correct total orderings of nodes is defined by data dependencies between nodes. Sequentialization of a dataflow graph selects one of several possible total orderings. The simplest ordering of dataflow nodes is depth-first. Figure 8 shows the appropriate algorithm when using definitions from figure 7.

```
PROCEDURE Walk (nd: Element);
VAR
        p: InputPort;
BEGIN
        p := nd.inPorts;
        WHILE p<>NIL DO
                IF p.sourceElem<>NIL THEN
                        Walk(p.sourceElem);
                        AppendInSequence (p.sourceElem); p := p.next
                END;
        END;
END Walk

PROCEDURE WalkGraph (gr: Graph);
BEGIN Walk(gr);
END WalkGraph
```

Figure 8: Depth-first ordering assuming sorted input-ports of a node

Main advantage of depth-first ordering is that it uses the resources efficiently. The life of temporaries that hold edge values is short. This is very important for machines with small number of registers like ADAM architecture. The performance of depth-first ordered graphs may be inferior because of missing parallelism. Therefore, we define three prerequisites for an adequate ordering of nodes, or more precisely of the instructions:

a) group call instructions and execute them together

b) delay instructions that depend on results from split-transactions

c) keep depth-first ordering if possible

Most important condition is that all data-independent call instruction execute without any synchronizing instructions between them. A call instruction represents either a function call or starts the computation of parallel loops. The reordering of nodes achieves speedup mostly through additional parallelism obtained with clustering of call-instructions.

A split-transaction sends a request and clears present-bit of result frame-register. The following instruction suspends its execution when it reads from frame-register with cleared present-bit. This leads to a context-switch and latency of the request can be hidden by executing another ready codeblock. Although a context-switch is fast, it still decreases the program performance. It is better when consuming instruction executes late so that the request can return results before they are needed. Such strategy also groups split-transactions so that a context-switch allows several requests to complete. The number of context-switches decreases when nodes are reordered. The clustering of calls has a higher priority than elimination of context switches.

3.2 Defining Object Types

ADAM architecture supports three different types of objects: local, interleaved and replicated. The code-generator decides which object is appropriate to allocate. The access to three different object classes is transparent. The access instruction is the same regardless the type of objects. Chapter 2 and figure 3 gave a brief description of the differences between object types. Various strategies presented in this chapter have the goal to remove the request bottlenecks by either increasing the data-locality or spreading the requests on the machine.

Objects are used for representing structured data. All records and arrays of Sisal programs are implemented with objects. The arrays of Sisal are dynamic in bounds and size. The ITH implementation allocates two objects for one dynamic array. The first object is array descriptor and the second is array content. The descriptor is a record that holds the reference to array content and the information about the size and bounds of the array. This is a mechanism similar to the one that is used in OSC. The problem of this model is that it is not reflected in IF2 graphs. Therefore, code-generators must accomplish optimizations like loop invariant removal and constant folding of descriptor records. This optimizations are already realized in first pass of OSC backend.

Replicated objects produce excellent data locality. The creation or modification of replicated objects is fast when object managers are idle. This is the case in sequential parts of programs. If replicated objects are modified in the parallel execution parts than their usage will collapse the machine. With analysis of the scope of a node, it is possible to determine if a node is executed in a sequential or a parallel path of the program. Of course, only those objects that will be read in parallel are allocated as replicated objects. A node is processed in a parallel part of program if it is nested either in a splittable parallel loop or in a recursive function. Depending on the code structure, parameter objects of PCALL instructions and the descriptors of arrays can be allocated as replicated or local objects. Most other allocation instructions generate local objects. Parameters and arguments between codeblocks are allocated as local objects too.

The implementation of interleaved object is simple. In a machine with p processors, the element with index i is on the processor $(i \bmod p)$. The interleaved objects decrease the data locality but distribute the requests. To be effective, interleaved objects need to have a homogeneous access structure and an appropriate size. This means that all elements of an interleaved structure should be accessed equally often and that all processors should

receive requests. Only in such case, the requests will be distributed across the machine. An array content that is accessed in parallel is allocated as interleaved object. The main problem with parallel tasks is the request bottleneck. In ADAM machine, data cannot be placed on the same processors where the consuming codeblock is executed. The only way to remove the bottleneck is to distribute data on all processors.

The request bottleneck does not only reduce the scalar performance but also hinders the load distribution. If more codeblocks must wait on their data, than more codeblocks will be expanded on every processor. This diminishes the amount of parallel work and increases the resource usage.

Figure 9 shows results when varying object types of array contents. The used program is a two-dimensional cellular automaton computing a grid of 256x256 cells. Sisal source is available from the ftp server of Lawrence Livermore National Laboratories. The suitable size of grid leads to an optimal distribution of requests when allocating array content as interleaved objects. The allocation of array descriptors as replicated objects improved the best results by 10% for 256 processors.

inner array	outer array	128 processors (cycles)	256 processors (cycles)
local	local	482360	268978
local	interleaved	449982	305108
interleaved	interleaved	351093	206178
local	replicated	479297	319811
interleaved	replicated	344350	206583

Figure 9: Varying object type for a two-dimensional problem

The measurings in figure 10 evaluate various allocation strategies for matrix multiplication of two 64x64 rectangular matrices. The main difference from previous example is that the interleaved objects are not spread evenly on a 128 processor machine but only on the lower 64 processing elements. This the defficiency of interleaved objects.

inner array	outer array	32 processors (cycles)	64 processors (cycles)	128 processors (cycles)
local	interleaved	1697095	871374	619834
interleaved	interleaved	1699264	899466	650533

Figure 10: Varying object type for the matrix multiplication program

3.3 Partitioning

As shown in figure 5, the partitioning pass of OSC reads a costs-file and assigns the execution costs to IF2 nodes and graphs. The cost analysis is used to decide whether a product-form loop should be executed in parallel or not. ITH reads IF2 file and evaluates the costs as given by OSC partitioner. This is a very simple method as the costs of IF2 graph do not correspond directly to the costs of the generated codeblock. For example, some nodes vary their execution costs depending on the source and destination nodes. Additionally, losses made by swapping the variables between the frame- and main-memory are not presented in costs of IF2 graphs.

Execution costs, as defined in IF2 file, are used to calculate the number and the size of parallel codeblocks. A sliceable forall loop is transformed into a parallel call and a parallel codeblock. The parallel call instruction issues a specific number of parallel codeblocks. Every parallel codeblock works on at least one loop-iteration. The number of iterations in the parallel codeblock is the slice size or the granularity of parallel work.

ITH code-generator uses two parameters to define the granularity of parallel codeblocks. One parameter defines the constant communication-costs attached to a parallel codeblock. The other parameter defines the wanted proportion between initialization- and execution-costs. The initialization costs are losses that are generated by a parallel codeblocks. Figure 11 shows an abstraction of the cost formula. The losses are the sum of the constant costs and the costs of input-parameter access. A constant proportion factor defines the minimum ratio between execution and communication costs in a parallel codeblock. Default ITH proportion factor is set to 90%, i.e., the losses should be at most 10% of execution costs.

```
losses := nrOfInputs*buildCosts + constCosts;
oneIterExecCosts := Costs(bodyGraph) + Costs(resultGraph)
sliceSize := (constantRatio*losses)/oneIterExecCosts;
```

Figure 11: Formulas for calculating granularity of parallel codeblocks

Three-dimensional cellular automaton is used to measure the effect of varying granularity. The call structure of this program is given in figure 12. We vary the granularity by changing the partitioning parameters of ITH. As it has been written before, these parameters cannot be changed by the programmer. They are changed here only to demonstrate the effects of variable granularity. Partitioning two uses default parameters.

```
Partitioning 1:   PCALL(40) -> PCALL(40) -> PCALL( 5)
                  PCALL(21);
Partitioning 2:   PCALL(40) -> PCALL(40) -> PCALL(8)
                  PCALL(42);
Partitioning 3:   PCALL(40) -> PCALL(40) -> PCALL(14)
                  PCALL(42);
Partitioning 4:   PCALL( 40) -> PCALL( 40) -> PCALL(20)
                  PCALL(42);
```

Figure 12: Call structure for the 3-dimensional cellular automaton on (40x40x40)

Partitioning two generates 14482 parallel codeblocks. Minimum number of codeblocks is 9661 and the finest partitioning generates 33682 codeblocks. Note that all nested loops are parallelized by ITH if OSC makes them sliceable. Figure 13 shows the total execution time with variable granularity and number of processors. Granularity two is the best compromise if we decide to deny the knowledge of the machine scale to ITH. The given execution time is the number of processors multiplied by the effective execution time in cycles. The graph for 256 processors has highest total execution time because of the decrease in speedup.

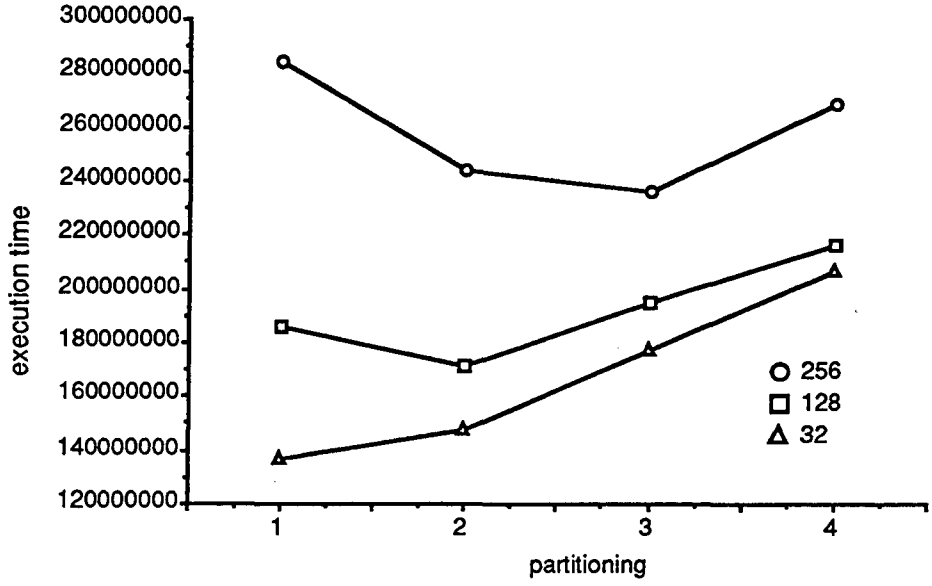


Figure 13: The effects of variable granularity

3.3 Banishment of Variables

Every codeblock owns a frame where it keeps scalar values. The frame has a fixed size of 32 registers. The situation in which the amount of active temporaries exceeds the number of available frame-registers requires a swapping mechanism between main-memory and frame. The swapping of frame-registers decreases the performance of a codeblock because of additional instructions. A more severe performance degradation occurs through additional synchronization through swap operations. A variable that must be saved into the main-memory can be the result of a split-transaction. The swap of such variable can cause an additional context-switch. An adequate banishment algorithm must minimize the number of swapping and synchronizing instructions.

Register and identifier assignment of ITH assumes an unbounded frame size when assigning registers to edges and nodes. The variables that are used for synchronization are marked with a weight. The variables that synchronize call instructions have a higher weight than those that represent results of split-transactions.

First, all variables are compacted into the legal register space. The variables with highest weight and access-frequency are reallocated first. If the number of needed registers exceeds the available number, then some variables need to be banished to the main-

memory. The variables with smallest weight and smallest access-frequency will be banished first. The access-frequency is calculated as follows:

access_frequency := number_of_accesses / life_time

The life-time of a variable is the distance between the first and the last instructions that reference the variable. Every access to a banished variable is preceded by a load or is followed by a write operation to the banishment object. Banishment pass removes all unnecessary reads and writes. In addition, the remaining write-instructions are delayed. Figure 14 shows banishment characteristics for two Sisal programs. Data is given for the largest codeblocks in the generated object-code.

	# statements of the codeblock	# banished variables	# deleted read&writes	# delayed writes
FitCubic	771	35	66	30
RICARD	305	40	37	43

Figure 14: Banishment specifications for two Sisal programs

4. Conclusion

The implemented code-generator compiles complex Sisal-programs and generates efficient code. Dynamic arrays pose additional problems as they increase the network traffic. The additional access to descriptor-object is prefetched outside the loop body but need to be done at the start of parallel codeblocks. The parallel access to array descriptors degrades the performance of dynamic arrays when compared to code that uses static arrays. Because some problems can be only modeled by dynamic arrays they cannot be replaced by static arrays. Other optimizations that will reduce the access on array descriptors need to be investigated. The dataflow graphs are still an important intermediate form although the execution model is based rather on blocks of instructions than on single parallel instructions as in pure dataflow. The dataflow graphs enable better scalar optimization and instruction ordering.

5. Acknowledgment

This work would have not been possible without the help of other members of project ADAM and the work done by students in their diploma thesis. Part of this work has been supported by grant No. 2.309-0.86 of the "Schweizerische Nationalfonds"

6. Bibliography

[FeoCannOldeh90] John T. Feo, David C. Cann, Rodney R. Oldehoeft. A Report on the Sisal Language Project. Lawrence Livermore National Laboratory, January 5, 1990.

[MitroMurer91] 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

[SkedzGlau84] S. Skedzielewski and J. Glauert, An Intermediate Form for Applicative Languages (Lawrence Livermore National Laboratory Report, 1984).

[WelSkeYaRan86] M. Welcome, S. Skedzielewski, R.K. Yates, J. Raneletti. IF2 - An Applicative Language Intermediate Form with Explicit Memory Management. Lawrence Livermore National Laboratory Report, December 2, 1986.

[McGrSkedz85] 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

[Maquelin93] O. Maquelin. ADAM architecture and its Simulation. Institut TIK, PhD Thesis, 1993.

[Maquelin90] O. Maquelin. ADAM: A Coarse-Grain Dataflow Architecture that Adresses the Load Balancing and Throttling Problems. Proceedings of CONPAR90, September 1990.

[Cann92] David Cann. Retire Fortran? A Debate Rekindled. Communications of the ACM, August 1992/Vol. 35, No. 8.

[Murer92] S. Murer. Translation of Functional Languages for ADAM Parallel Computer. Institut TIK, PhD Thesis 1992.

[Mitrov90] S. Mitrovic. et al. A Distributed Memory Multiprocessor Based on the Dataflow Synchronization in Proceedings of International Phoenix Conference on Computers and Communication, (1990).

[Mitrov92] S. Mitrovic. Programming ADAM Architecture with Sisal. Presented at the Second Workshop on Dataflow 1992, Hamilton Islands, Australia

[MurerFaerb92] St. Murer and Philipp Färber. A Scalable Distributed Shared Memory. Proceedings of CONPAR92, page 453-466, September 1992.

Program Partitioning for NUMA Multiprocessor Computer Systems

Richard Wolski
John Feo

Computer Research Group
Lawrence Livermore National Laboratory
Livermore, CA 94550

Abstract: An important part of parallel programming is program partitioning and scheduling. Partitioning is the separation of program operations into sequential tasks, and scheduling is the assignment of tasks to the processors of a computer system. To be effective, automatic methods require an accurate representation of the model of computation and the target architecture. Current partitioning methods assume the macro-dataflow model of computation and the homogeneous/two-level architectural model. The former is typically represented as a directed, acyclic graph of computation nodes and communication edges. The edges map directly to communication channels, but not read/write memories. Consequently, current methods optimize assuming the presence of communication channels, and not the complex memory systems of NUMA architectures—they fail to optimize for a critical component of these architectures. In this paper, we extend the conventional graph representation of the macro-dataflow model to enable mapping heuristics to work with a NUMA architectural model. We describe two such heuristics. Simulated execution times of programs show that our model and heuristics generate higher quality program mappings than current methods.

1.0 Introduction

An important part of parallel programming is program partitioning and scheduling. Partitioning is the separation of program operations into sequential tasks, and scheduling is the assignment of tasks to the processors of a target architecture. Together, partitioning and scheduling constitute the *mapping problem*. Program mapping can either be done manually by the programmer, or automatically by the language compiler and runtime system. If the compiler and runtime system are responsible for mapping, they require an accurate representation of both the program to be mapped (a model of computation), and the desired target architecture (an architectural model). Current partitioning methods assume the *macro-dataflow model* of computation and the *homogeneous/two-level* architectural model. The former is typically represented as a directed, acyclic graph of computation nodes and communication edges. The edges map directly to communication channels, but not read/write memories. Consequently, current methods optimize assuming the presence of communication channels, and not the complex memory systems of NUMA architectures—they fail to optimize for a critical component of these architectures [1].

Much of the research in the area of program partitioning and scheduling has centered around the macro-dataflow model of computation [21, 20, 11, 13, 6]. This model is best described in terms of its conventional representation, a directed acyclic graph. The DAG's nodes correspond to computations and the edges to communication between computations. Each node in the graph has one or more input edges representing the operands necessary for the computation to execute, and one or more output edges representing the computed results. The data dependencies enforce operation precedence. The order of execution is constrained only by the program's data dependencies, and computations and communications may be overlapped. Independent nodes may execute in parallel, and there is no notion of control flow. The model can support recursion and program looping by implied dependencies [22]. Nodes are assumed to be strict; that is, a node cannot begin execution until all of its inputs have arrived, and no output is available until all of its outputs are available. The model supports fine-, medium-, and coarse-grain parallelism. Overall, the utility of the macro-data flow computational model is high; however, since the edges do not map directly to read/write memories, it is difficult to design mapping heuristics that optimize for today's hierarchical memory systems.

The macro-dataflow model specifies what to execute, but not how it is to be executed. The latter requires the graph to be mapped to the resources of a target architecture. To estimate the cost of different mappings requires a description of this architecture. Due the limitations of the directed, acyclic graph representation of the macro-dataflow model, most of the previous research in partitioning and scheduling assumes a *homogeneous/two-level model* in which

- the target architecture is composed of a homogeneous set of processors,
- there are two levels of interprocess communication (local and remote), and
- communication between processors can occur in parallel.

If the computational model is medium- or coarse-grained, the cost of each node is the sum of the cost of its constituent instructions. Instructions within the same operation, as well as operations executing on the same processor, are assumed to communicate without delay (i.e., zero local communications). Remote communication (communication between operations executing on separate processors) is typically represented by a single cost function,

$$(volume\ of\ data\ /\ communication\ rate) + start\text{-}up\ cost$$

where *communication rate* and *start-up cost* are constants. Additionally, the model assumes that the communication of results from a producer to multiple consumers can occur in parallel.

The homogeneous/two-level model does not accurately describe existing parallel computer systems. In particular, it does not capture the hierarchy of extant memories. Today parallel computer systems consist of "conventional" sequential processing units linked by a communication system (shared bus, memory switch, or communication network) Few of these systems are true homogeneous/two-level machines. Most cache-coherent machines, such as the Sequent 81000, SGI Iris 430, or Multimax Encore, support at least three levels of storage: registers, cache, and central memory. Further, since communication between computations on the same processor may use any one of the three levels of storage, local communication costs cannot be normalized to zero. Other systems support an even richer diversity of memories and communication facilities. For example, on the BBN TC2000, two operations executing on different processors may communicate using one of four mechanisms: remote-write/local-read, local-write/remote-read, interleaved-write/interleaved-read, and MACH messages. Two computations executing on the same processor may communicate using three additional mechanisms: registers, cache, and private memory. We anticipate future architectures will be even more complex. For example, CEDAR [12] is organized as a tree of processing elements giving rise to (log P) communication levels (where P is the number of processing elements). The DASH system [7] imposes 5 different read latencies and 4 different write latencies depending on the level of the memory hierarchy used. Consequently, the two/level model does not adequately depict current or future systems.

The assumption that data can be "sent" in parallel to multiple consumers is also not realistic. Consumers can not simultaneously read the same memory location. A producer can not transmit

two different results simultaneously, nor can it send a single result to multiple consumers unless the hardware supports broadcast.

Previous research [21, 20, 14, 13, 11, 9, 17, 16, 24] uses the DAG program representation and the homogeneous/two-level architecture model. Sarkar [21, 20], Kim and Browne [11], and Yang and Gerasoulis [24] statically partition programs into a set of non-strict sequential partitions assuming an infinite number of processors, and then schedule the resulting partitions on a given target machine according to some scheduling heuristic. While these systems share a common approach (static partitioning followed by static scheduling), they use different heuristics to construct partitions and schedule those partitions to processors. All three systems assume zero-delay local communication costs, concurrent communication of results, and a single remote communication cost function. Sarkar, however, decomposes remote communications into three sequential phases: write, communicate, and read.

Kruatrachue [14, 13] considers scheduling before partitioning. He builds a static schedule directly from macro-dataflow nodes. He then considers nodes that are scheduled on the same processor and are contiguous in time as forming the partitions of the program. Nodes are first scheduled on the processors, and then partitioned into sequential tasks according to the schedule. Hwang and Xu [9] formulate the mapping problem as a mathematical optimization problem. They use simulated annealing to statically determine a near optimal schedule for hypercube architectures. They do not propose a partitioning scheme. Adopting a more graph-theoretic approach, McCreary and Gill [17, 16] propose graph-parsing as a way of separating nodes into partitions. They statically parse the macro-dataflow DAG into fragments, some of which may be executed in parallel. Again, while these approaches vary widely, they all use the macro-dataflow computational model and the homogeneous/two-level architecture model to estimate the cost of execution.

In this paper, we extend the conventional graph representation of the macro-dataflow model to enable mapping heuristics to work with a NUMA architectural model. We describe two such heuristics. Simulated execution times of programs show that our model and heuristics generate higher quality program mappings than current methods. Section two presents the new architectural model and illustrates its potential. Section three describes two simple partitioning algorithms based on this model. In section four, we compare program mappings generated using our new techniques with those generated by previously developed systems. In section five, we present our conclusions and discuss future directions.

2.0 Memory Node Model

In this section, we extend the directed, acyclic graph representation of the macro-dataflow model. To capture the behavior of read/write memories, we divide each communication into three phases. First, nodes write their outputs into some form of storage. Second, an intermediate communication facility may transfer the data to storage accessible by the receiver. Finally, the receiver reads its inputs from where they have been stored. To represent the write-communicate-read sequence, we include two memory nodes along each program edge, thereby separating the three phases [Figure 1a]. If the sender and receiver communicate via shared memory, then the communicate phase is non-existent, and we merge the two nodes [Figure 1b]. The inclusion of memory nodes permits mapping heuristics to consider the different types of memories and communication options supported by NUMA architectures, thereby generating improved program mappings.

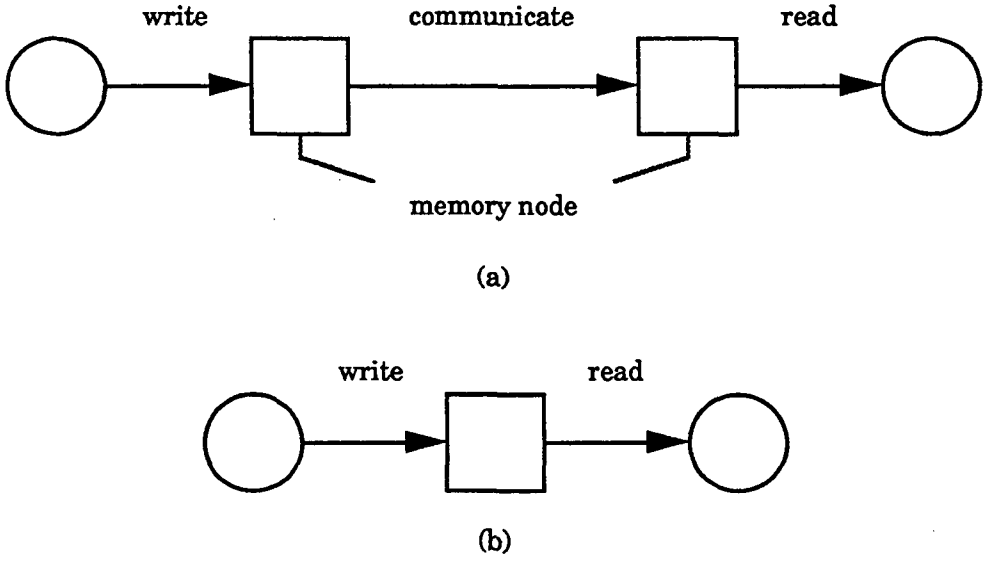


Figure 1 – Write, Communicate, Read

The architectural model specifies a write, communicate, and read cost for each type of memory and communication channel. The delay associated with a program edge depends on the memory and communication channel chosen, and the amount of data communicated. For example, consider two operations scheduled on the same processor of a BBN TC2000. Assume that the data communicated between these operations is small enough to fit within a single register. If a register is used, the communication costs two cycles [Figure 2a]. If cache is used instead of a register, the cost is 6 cycles [Figure 2b]. Lastly, if we use local private memory, the delay is approximately 16 cycles [Figure 2c]. Observe that Figure 2 shows three different costs for *the same*

local communication. The actual communication could vary by a factor of 8. The assumption made by current systems that local communications is zero or some fixed constant cost is not accurate.

Memory nodes also parameterize communication between processors if multiple communication facilities are supported. For example, the TC2000 supports three shared memory mechanisms: local, remote, and interleaved. The respective access times of the three memories are: 8, 37, and 37. Thus, a local write/remote read or remote write/local read takes 45 cycles, and a

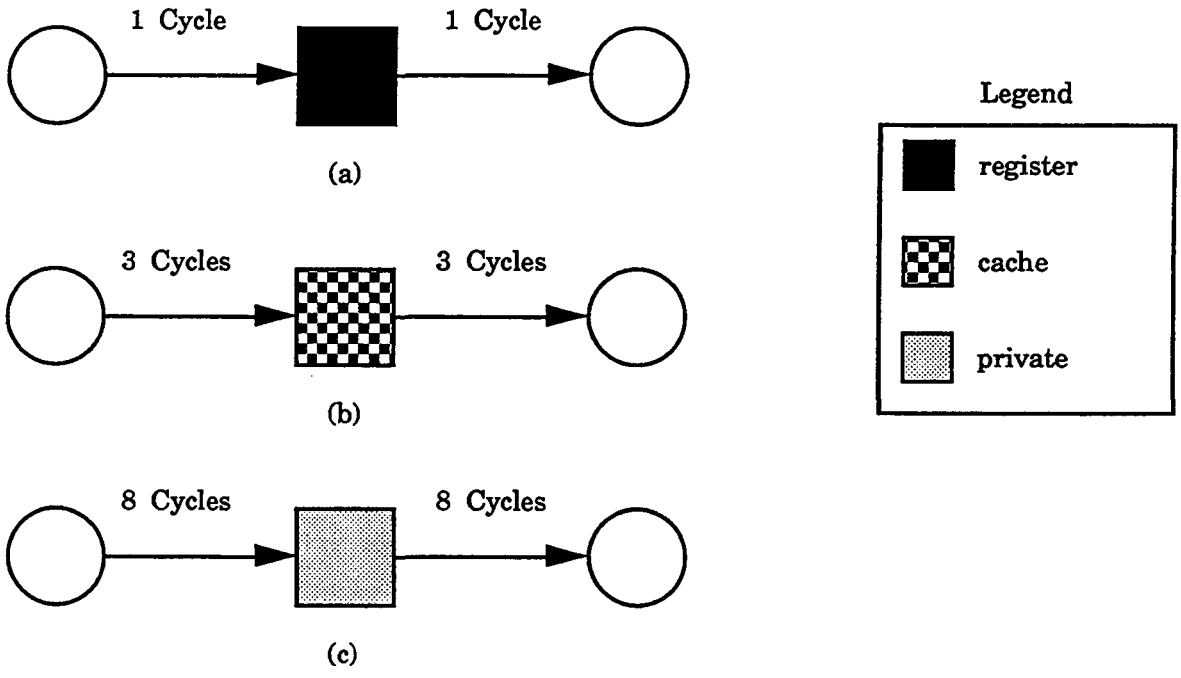


Figure 2 – Three local communication costs

write/read of interleaved memory takes 74 cycles [Figure 3].

Consider the three examples shown in figure 3. Both the local-write/remote-read and remote-write/local-read mechanisms imply an overall communication cost of 45 clock cycles. The overall cost for the interleaved case in 74 cycles. To understand the utility of the memory node model, assume that the two nodes shown in the examples are part of a larger program, and that the sender, but not the receiver, is on the critical path for the program. That is, the critical-path includes the sender and its *left-hand* output edge. Since computation nodes are strict with respect to their outputs, the overall delay through the sender includes the sum of its write costs. Thus part of

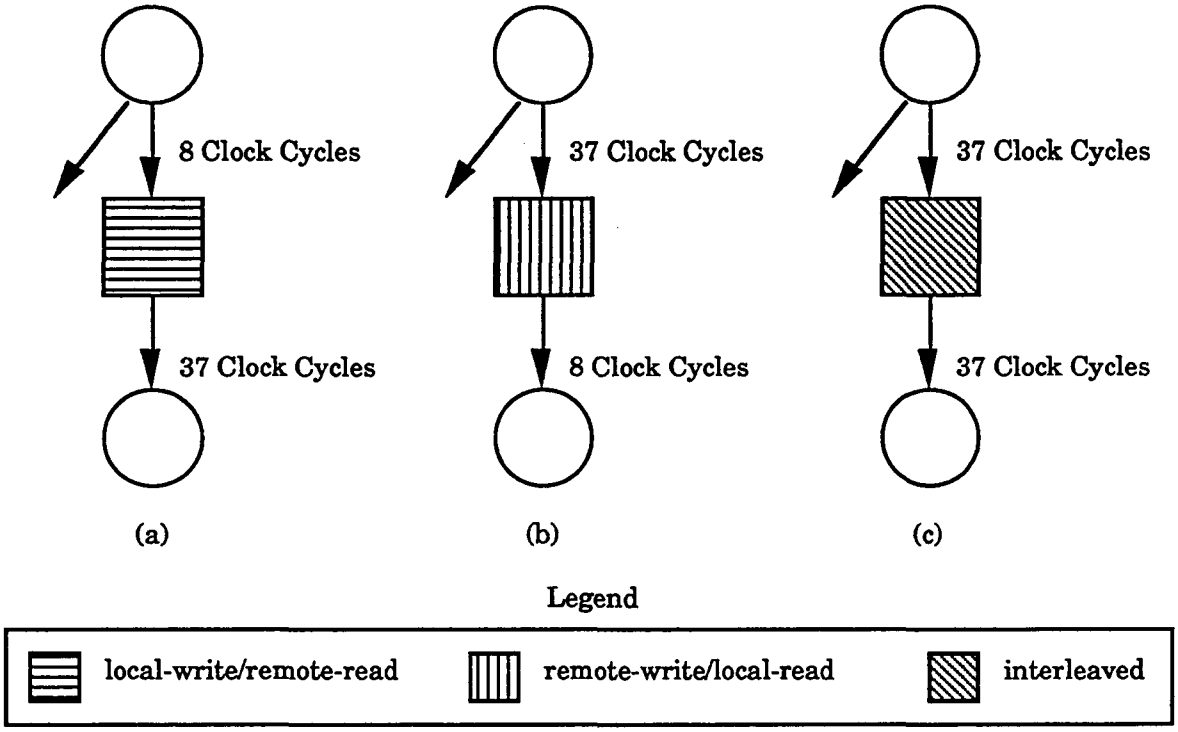


Figure 3 – Three remote communication costs

the delay along the critical path includes the cost associated with writing all of the sender's outputs. Since the receiver is not on the critical path, it is best to give the sender the smaller access time (thereby minimizing the path through the sender) and have the receiver incur a longer, non-critical delay. Thus, the best choice of communications is illustrated by the first graph—local write/remote read. Current systems that use a only a single communication cost function fail to consider all the available choices.

Another important feature of the memory node representation is that it correctly captures data sharing relationships. Quite often, the results of a computation must be sent to multiple remote receivers. In the models used by current systems, each communication is represented by a separate edge [Figure 4a]. If the same value is being sent to multiple consumers, the producer will most likely write it to memory only once. We depict the edge-sharing relationship as fan-out from the memory node [Figure 4b], and not fan-out from the computation node. Current systems may charge the producer three writes, while the true delay is only one write. Moreover, if it is advantageous for the producing computation to write its results into two separate memories, the different delays are captured by the memory node model, but not by the conventional DAG.

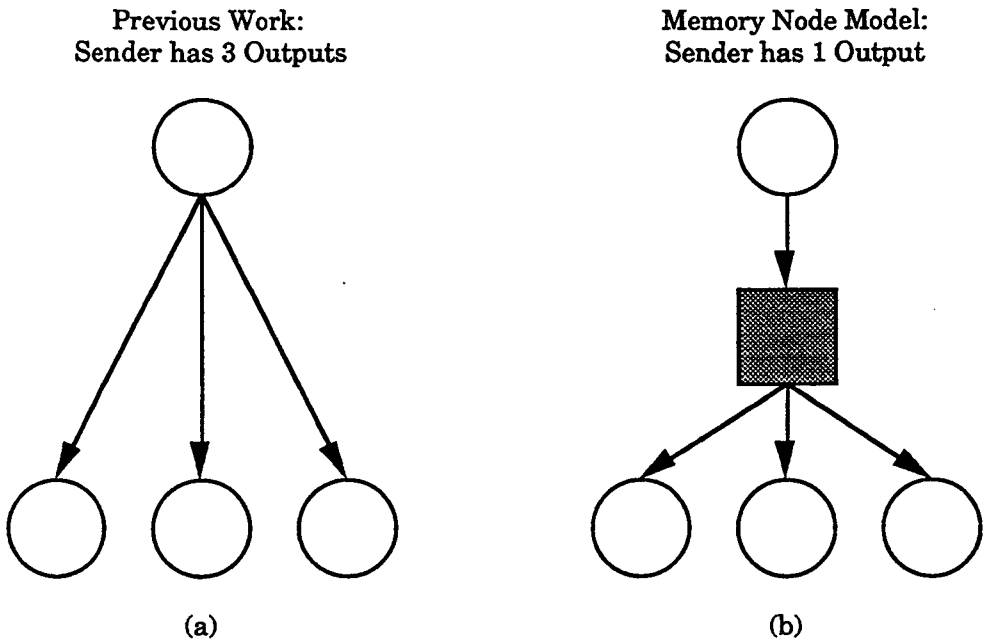


Figure 4 – Single producer and multiple consumers

Memory and communication systems have characteristics other than access latency which can affect program execution cost. For example, some types of memory may not support parallel accesses due to contention (block shared memory), while another memory type might perform well in the face of contention (interleaved memory), but at a higher cost. Since current systems do not consider memory, they neither model nor evaluate the different options. Our model identifies the true point of contention, and permits mapping heuristics to schedule memory accesses.

3.0 Partitioning Heuristics Based on the Memory Node Model

To test the utility of memory node graphs, we implemented a prototype partitioning system, and developed several simple partitioning algorithms based on the model. In this section, we outline the structure of our prototype, and describe the memory node based partitioning algorithms.

3.1 A Prototype

Our prototype partitions IF1 [22] program graphs. IF1 is the intermediate form of the parallel programming language Sisal [18, 5], is a high-performance functional language for numerical and scientific computation developed at Lawrence Livermore National Laboratory and Colorado State University. An IF1 program consists of one or more directed acyclic graphs made up of

simple nodes, compound nodes, graph nodes, edges, and types. Nodes denote operations, edges transmit data between nodes, and types describe the transmitted data. Simple nodes represent operations such as addition, division, and array and stream manipulation. Compound nodes encapsulate one or more subgraphs to define structured expressions such as conditionals and loop expressions. IF1 is well suited to parallel program representation due to its graphical nature and applicative semantics.

3.2 Architecture Description

The parameters associated with a particular target architecture are specified using the architecture description facility. To estimate the execution time of a program on a particular architecture, each computation node and communication edge in the program must be assigned an individual execution cost. The architecture description file lists the execution time of each IF1 node type on the given architecture, and, for each possible memory type, lists a read cost, a write cost, a flag indicating if the memory may be shared, and the degree to which the memory may be accessed in parallel. In addition, the file defines the communication cost between every possible pair of memory types, and the default, or preferred, communication mode.

Prior to partitioning, we assume that the IF1 program is annotated with communication volume estimates. Estimates may be derived by trace techniques [20], user annotations, and compiler analysis [3, 19, 2]. Initially, the partitioning system calculates the delay (number of cycles) through each computation node, calculates the size (number of bytes) of each memory node, and assigns a memory type (bytes per cycles) to each memory node.

3.3 Partitioning Algorithms

The prototype system includes two simple partitioning algorithms: a heavy-edge-first algorithm (HEF) inspired by Sarkar's work [21, 20], and a critical-path algorithm (CP) similar to DSC proposed by Gerasoulis and Yang [24, 6]. However, the prototype algorithms use the information provided by the memory node model when making partitioning decisions. As a result, they differ quite substantially from previous algorithms even though they have the same basic orientations.

General Characteristics

Both algorithms use the estimated critical path of the overall program as a performance metric. That is, each partitioning decision is evaluated based upon its effect on overall graph critical path. Both algorithms attempt to improve program critical path by assigning the best

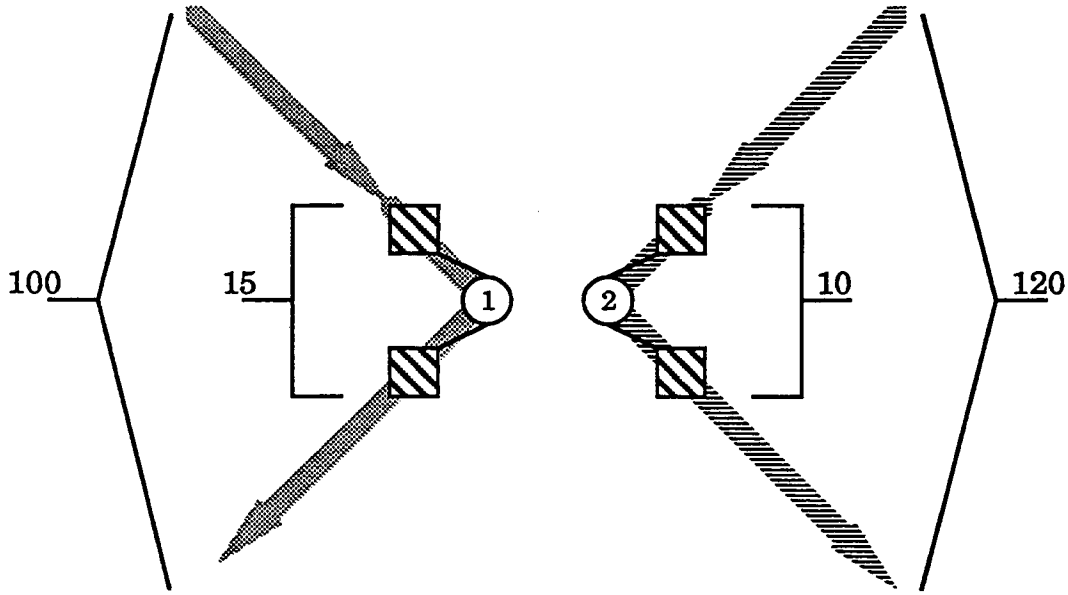


Figure 5a – Independent nodes

memory type to each memory node in the program. We term the assignment of a memory type to a particular memory node the "coloring" of that node with a given type. For example, a memory node might be "colored private" to indicate that it should be assigned to memory that is private to a processor. Also, both HEF and CP are iterative improvement algorithms. That is, the memory nodes in the graph are each assigned an initial coloring and then the algorithms attempt to alter the node colorings during each iteration. HEF and CP avoid backtracking in that once the coloring of a particular memory node is accepted, that node will not be recolored.

Linearization

Using HEF or CP, two computation nodes that communicate via a fused memory node which is colored with a private memory type are assigned to the same partition. Otherwise, the computations are assumed to be part of separate partitions. When a fused memory node between members of separate partitions is colored private, the partitions are merged. The ordering of independent computations that have been assigned to the same partition can effect the overall program's critical path.

Consider the graph fragment shown in figure 5a. In the figure, computation node 1 and computation node 2 are independent. The sum of the read delay, execution time, and write delay for computation node 1 is shown as 15 cycles. Similarly, the sum of the read, write and execution delays for computation node 2 is depicted as 10 cycles. Assume that the longest path from graph

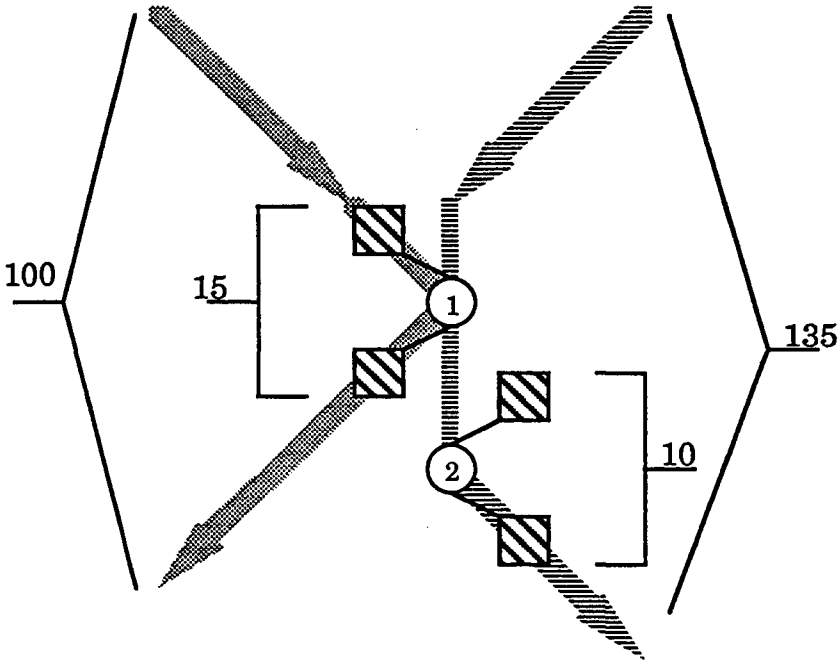


Figure 5b – Node 1 before node 2

source to sink through node 1 is 100 cycles, and that the similar path length through node 2 is 120 cycles (depicted by the heavy arrows in the figure). If node 1 and node 2 are assigned to the same partition, the partitioner is free to choose their relative ordering. That is, either node 1 may be scheduled to execute before node 2, or vice versa, and the overall program will execute correctly. Figure 5b shows the configuration that results when node 1 is chosen to execute before node 2. Notice that the distance through node 1 does not change, but node 2 is delayed by node 1's read, execute and write delays (the path length through node 2 is extended by 15 cycles to 135). Alternatively, figure 5c shows the configuration with node 2 scheduled before node 1. If node 2 comes first, then node 1 is artificially delayed by node 2, while node 2's path length remains unchanged. Notice, however, that the resulting path length through node 1 is shorter than that through node 2. If the program's overall critical path runs through node 2, then the configuration shown in figure 5b increases the critical path, but the shown in 5c does not.

The problem of determining the optimal sequence of nodes within a partition is NP-complete [20]. Since the ordering of computations within partitions is important, the partitioner needs to employ some heuristic to linearize potentially parallel computations. Yang and Gerasoulis refer to this process of linearization as scheduling, [6] whereas Sarkar terms it node sequence mapping [20], and the literature is rife with different techniques [8, 4, 13, 15, 24, 20] None of these heuristics seem to make the ordering decisions based on the overall graph critical path (which is the performance metric used by HEF and CP). We, therefore, propose an alternative heuristic that

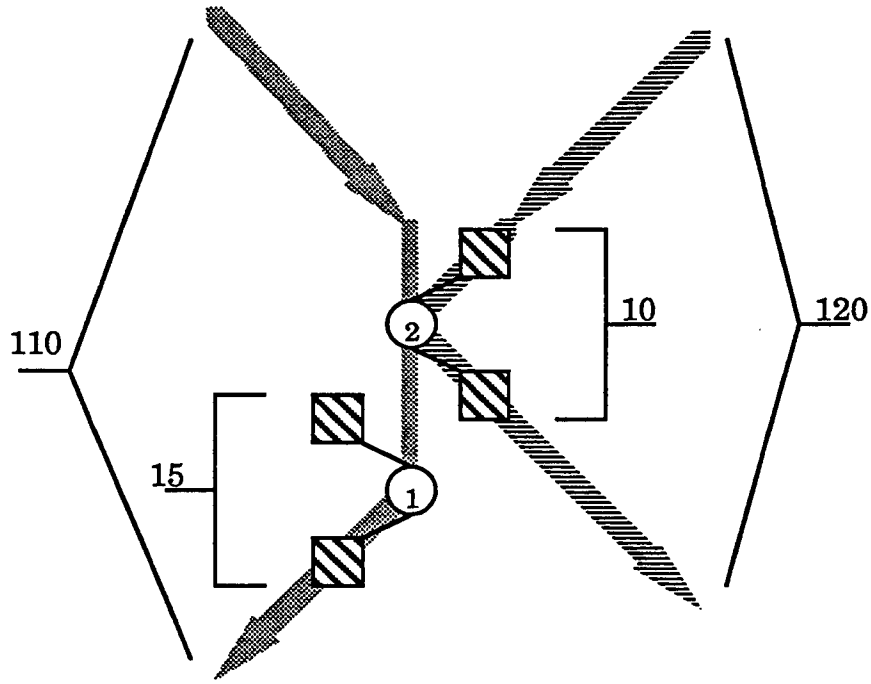


Figure 5c – Node 2 before node 1

uses the path length estimate through each node to form the orderings within partitions. When two partitions are to be merged, each partition is treated as a pre-sorted list of computations. The goal of the heuristic is to form a single sorted list of computations that does not violate program precedence, but which minimizes overall graph critical path. Essentially, the algorithm constitutes a single iteration of merge sort augmented to recognize program precedence. The heads of each of the two partitions is are compared. If program precedence dictates an ordering between the two, the partition head that must go first is removed from its partition, and added to the end of the new partition. Otherwise (the two partition heads are independent), the algorithm calculates the distance through each head for both possible relative orderings. The configuration that yields the shortest maximum path length through either node is chosen. For example, if the two nodes shown in figure 5a are the head nodes of the partitions being merged, the algorithm calculates the distances shown in figures 5b and 5c. Since the configuration shown in figure 5c yields the shortest maximum length (120 cycles), that configuration would be chosen. When all nodes in both partitions have been moved into the new partition, the algorithm terminates. The complexity of our algorithm is O(N) where N is the number of computation nodes in both input partitions.

HEF

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. Each node (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 nodes have been examined and assigned a memory color.

CP

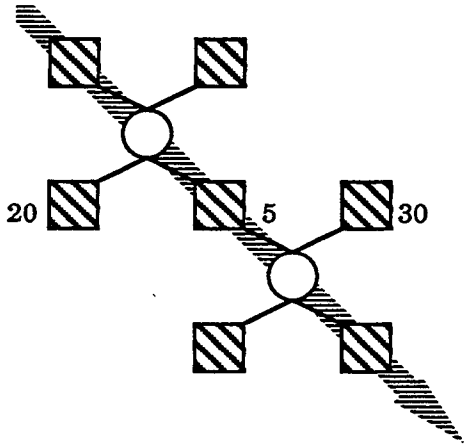
The other partitioning algorithm included in the prototype system attempts to reduce execution time by focusing on those memory nodes along a graph's critical path. All memory nodes in a program graph are first colored with a default memory type. Each iteration of the algorithm chooses an edge on the graph's critical path and attempts to reduce the path length through the memory node(s) associated with that edge. As in HEF, the memory nodes are speculatively colored with each of the k possible memory types, and the coloring that yields the shortest resulting critical path is accepted. Once a coloring is accepted, however, the graph's critical path is recomputed since it may have changed as a result of the coloring. Therefore, each iteration chooses memory nodes from the critical path that results from the coloring performed on the previous iteration. The algorithm terminates when no colorings along a graph's critical path result in a path length reduction.

Neighborhoods

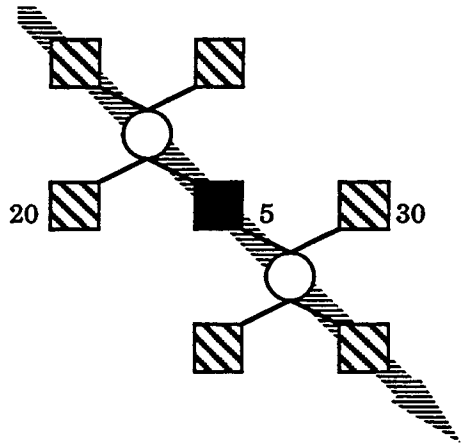
The naive implementation of CP only attempts to color those memory nodes that are directly on the critical path. For example, consider the graph fragments shown in figure 6a. The heavy arrow indicates that the program's critical path traverses both computation nodes in each fragment, and the numbers denote the communication volumes associated with their associated memory nodes. A naive implementation of CP might choose the memory node between the two computation nodes (as it is directly on the critical path) and assign it a faster memory type (figure 6b). By assigning the critical memory node to a faster memory, the overall path length is reduced by 90 cycles.

In figure 6c, the critical path length is reduced by 270 cycles. Notice that it is only the cost of the read for the 30 byte memory node which is included in the critical path length calculation. That is, in the graph fragment on the right, the memory node with size 30 causes a read delay of 300 cycles. The corresponding 300 cycle write delay is not included in the critical path length since the

producing computation is not part of the critical path. When the memory node is colored, the read cost becomes 30 yielding a net reduction of 270 cycles.

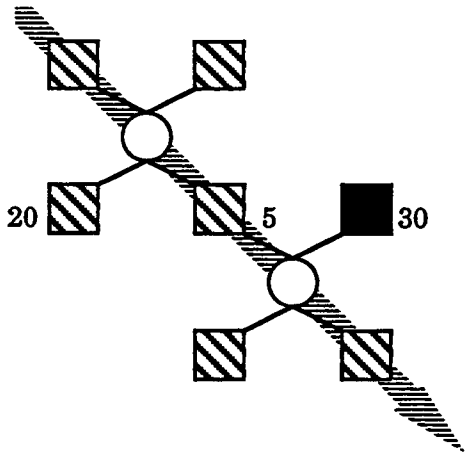


(a)



Critical path reduced by 90 cycles

(b)



Critical path reduced by 270 cycles

(c)

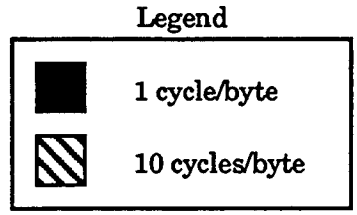


Figure 6 – The effect of neighborhood search in CP

In general, then, the path length through any memory node may be reduced by a coloring of the node itself, and/or some set of its neighboring memory nodes. We define the *neighborhood* of a memory node to be the region of the graph in which memory colorings are considered when the path length through that memory node is to be reduced. CP is parameterized by a neighborhood size which defines the extent to which non-critical nodes are examined. We plan to study ways in which the appropriate neighborhood size can be determined for each node, as opposed to having a single size for the duration of the algorithm.

CP/HEF

As we mentioned in our discussion of the complexity associated with CP, not all memory nodes in a program graph may be considered during the algorithm. If the neighborhood is small, for example, good candidates for coloring may be missed. One possible strategy to improve upon CP, then, is to apply HEF to those memory nodes which remain uncolored after CP completes. The complexity of the combined algorithms remains O(N^2) since we run them in sequence. Notice, however, that the reverse ordering (namely HEF followed by CP) will not yield an improvement. The chief difference is that HEF considers all nodes in the graph before terminating, while CP concentrates on the "important" ones at the expense of others. Therefore, we can only use HEF to clean-up what CP has missed and not vice versa.

4.0 Results

To test the validity of the memory node model, we conducted a series of studies using the prototype system. We first wanted to investigate the improvement, if any, that the memory node model by itself provides over previous architecture models. To do so, we implemented DSC [24, 6], both in its original form and with memory nodes, and compare the partitions generated by each. Secondly, we wanted to investigate the performance of HEF and CP, as they are algorithms that make partitioning decisions using the information presented by the memory node model. We compare the performance of HEF, CP and DSC (both with and without memory nodes) on the same set of graphs to gauge their relative effectiveness. Next, we wanted to indeed verify that our algorithms could exploit diverse memory systems. Since previous work exclusively assumes a two-level architecture model, we are unable to compare CP and HEF to a previous system for an architecture with more than a two-level communication hierarchy. We, therefore, simply present our results. Finally, we wanted to investigate the effect of our linearization algorithm, the modified merge sort. We have implemented a version of HEF that uses Sarkar's node sequencing algorithm [20], and we compare its performance to HEF using our technique.

Test Cases

We generated all of our results from a set of 100 randomly generated IF1 program graphs. Each graph has a single source and sink node but is otherwise composed of 100 primitive IF1 nodes. Although all graphs are directed and acyclic according IF1 semantics, their interconnections are random. In our tests, every computation node has a random fan-in of between 1 and 3 inputs, but an arbitrary number of outputs. Each graph edge is annotated with a random communication volume of between 1 and 100 units. Similarly, each computation node is

assigned a random execution time of between 1 and 100 cycles. We assume a homogeneous processor topology so execution times may be assigned before partitioning. Also, all tests were conducted assuming a NUMA machine architecture in which communication is facilitated via shared address spaces. As shared addresses are used for communication in our studies, all memory nodes are initially fused. We also do not consider the effect of memory node sharing. That is, each memory node has a fan-out of exactly 1.

For each partitioning algorithm, we use, as a measure of performance, the mean percentage improvement in critical path length resulting from partitioning over the 100 test graphs. We initially assume each graph is unpartitioned (or equivalently, each computation has been assigned to its own partition) and that all communication uses a default shared memory type. As in previous work, we assume an infinite number of processors during partitioning, so computations are not sequentialized in the unpartitioned "base case." We calculate the critical path length of the base case for each graph and use that as a measure of the graph's unpartitioned execution time. We then partition each graph using a particular partitioning algorithm, and calculate the critical path length of the resulting partitioned graph. The percentage improvement is defined as

$$\%Improvement = \frac{base\ case\ critical\ path\ length - partitioned\ critical\ path\ length}{base\ case\ critical\ path\ length} * 100$$

We use the mean percentage improvement over all 100 test graphs as an overall performance measure for each partitioning algorithm.

DSC and Memory Nodes

In the experiment using DSC, we wanted to isolate and investigate the utility of the memory node model by itself. To do so, we compare DSC as it is described by its authors, with a version than has been modified to use the memory node graph representation. Since DSC assumes a two-level architecture model, we specify only two types of memory in the architecture description for this experiment: local private memory, and global shared memory. All computations within a given partition are assumed to communicate via local private memory and all communication between partitions use global shared memory. To reflect program behavior on an actual machine, we assign the read and write costs for both memory types according to those specified for the BBN TC2000 [10]. Each read or write of local memory causes a delay of 8 clock cycles per unit of data (assuming the data is not cached). While the TC2000 supports several shared memory facilities, we emulate the behavior of interleaved shared memory since it most nearly supports parallel

access. The delay associated with each interleaved memory read or write is 37 clock cycles per unit of data.

Since DSC assumes a local communication cost of zero, we were concerned that it might not perform well for an architecture where the local communication cost is not negligible. Furthermore, the DSC algorithm does not adjust internal cost estimates during partitioning to keep its overall complexity low. Since we were interested in the effect of memory nodes (and not simply the effect of assuming a non-zero local cost) we needed a way to incorporate non-zero local communication costs once at the beginning of the algorithm. To do so, we decided to include the cost of all local accesses in the execution cost of each computation node, and then use the difference between local and shared communication cost as the cost of non-local communication. That way, when an edge is zeroed, the correct local communication delay is considered as part of the overall execution cost. For example, consider the graph fragments shown in figure 7a. In the fragment on the left, the overall communication delay between the computation nodes is 74 cycles per unit, and the overall path length is 99 cycles. If the partitioning algorithm assigns the computations to the same partition, then the communication can use local private memory (as depicted in the graph fragment on the right). The communication delay between computations scheduled within the same partition is 16 cycles per unit, and the path length is 41 cycles. If the local memory access cost is folded into the execution cost of each computation, the difference between local and remote communication cost may be used to represent the cost of remote communication. For example, in the right-hand graph fragment of figure 7b, the cost of each computation node includes a local access cost. The cost of the remote communication is 74 - 16 = 58 cycles. Notice that the overall path length is still 99 cycles. When the communication edge is subsequently zeroed by DSC, the path length becomes 41 cycles. Therefore, even though the edge is zeroed, the path length includes the cost of local communication.

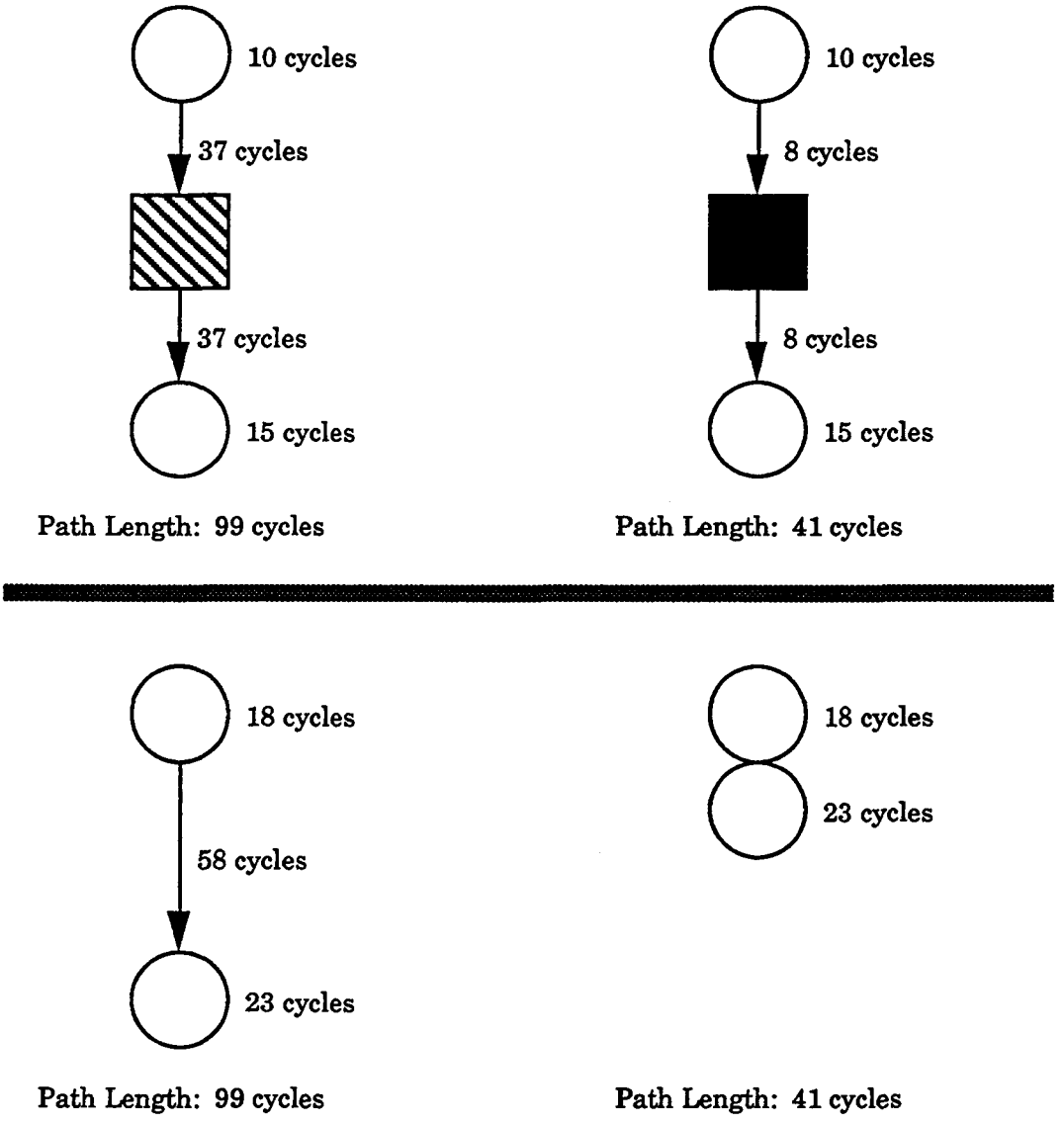


Figure 7 – DSC augmented with non-zero cost

To implement the memory node version of DSC, we changed the algorithm to consider the read and write delays associated with each computation. That is, we factored the true read and write delays into the path length estimates. For example, if a computation has a private output and a shared output, the overall delay through the computation would include the cost of writing the shared and private memory nodes corresponding to the outputs. We did not increase the complexity of the algorithm, however, by traversing the graph to adjust all cost estimates after every memory node coloring. The estimates, therefore, become more and more inaccurate as partitioning progresses.

Algorithm	% Improvement	
	Actual	Predicted
DSC	0.0 %	76.2 %
DSC with Non-zero Costs	18.9 %	62.2 %
DSC with Memory Nodes	21.5 %	28.4 %

Table 1 – Performance of DSC

Table 1 shows the mean percentage improvement for all three versions of DSC.

The tables shows two performance figures for each algorithm. The actual percentage improvement reflects the reduction in critical path that resulted from partitioning. That is, it is taken from the critical path length of each partitioned graph. However, DSC maintains its own estimate of critical path (called the graph's "dominant sequence") during partitioning. The predicted percentage improvement reflects DSC's internal estimate of critical path length.

We refer to the ratio between remote communication cost and local communication cost as the *granularity* of the communication system. In our example, we assume that the granularity of the target communication hardware is 37:8 or approximately 4.6:1. As we have mentioned, DSC over-estimates the granularity by assuming a zero local communication cost. For architectures where the granularity is coarser, however, we expect DSC to perform much better. Further, the performance of DSC with non-zero cost and the performance of true DSC should converge as communication granularity increases. Figure 8 plots the percentage improvement for DSC, DSC with non-zero local cost (denoted DSC-NZ in the figure), and DSC with memory nodes, over successively coarser granularities. We fix the local cost at 5 clock cycles, and vary the remote cost from 5 to 1000 cycles along the abscissa.

As predicted, the performance of DSC with non-zero cost converges to that of true DSC as granularity increases. Notice that DSC with memory nodes is able to achieve better improvement across the spectrum of granularities. As granularity increases, the penalty for misjudging local communication cost decreases. The degree to which a node's starting time is under-estimated increases, however. Since DSC with memory nodes captures both the local communication cost and true delays due to computation strictness, it performs better across the spectrum of granularities.

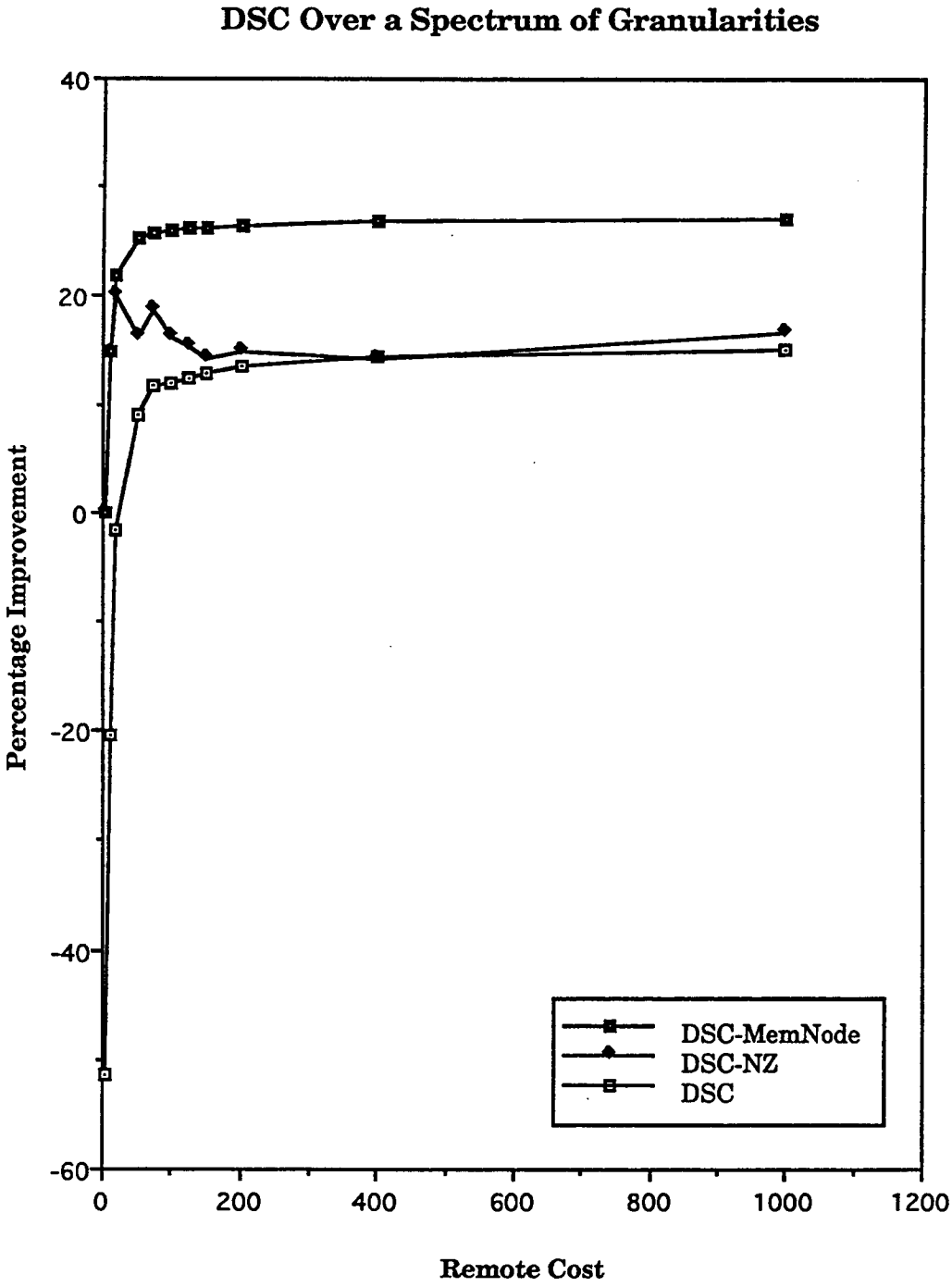

DSC Over a Spectrum of Granularities
40
20
0
-20
-40
-60
0
200
400
600
800
1000
1200
Percentage Improvement
Remote Cost
DSC-MemNode
DSC-NZ
DSC

Figure 8

Algorithm	% Improvement
HEF	30.0%
CP	32.1%
CP/HEF	33.8%

Table 2 – Performance of HEF, CP, and CP/HEF

HEF, CP, CP/HEF

Table 2 shows the mean percentage improvements for HEF, CP (with a neighborhood of six nodes), and CP/HEF.

Since CP and HEF both work with the true graph critical path lengths, there is no difference between the predicted and actual improvements. The cost of maintaining the correct critical path estimates during partitioning, however, is added algorithmic complexity. Compared to DSC which has a low complexity of O(N * log N), these algorithms are able to achieve approximately 10% better improvements in O(N^2) time. Figure 9 compares the performance of the DSC algorithms with that of HEF, CP, and CP/HEF over a spectrum of granularities. Again, we fix the local cost at 5 cycles and vary the remote cost between 5 and 1000 cycles.

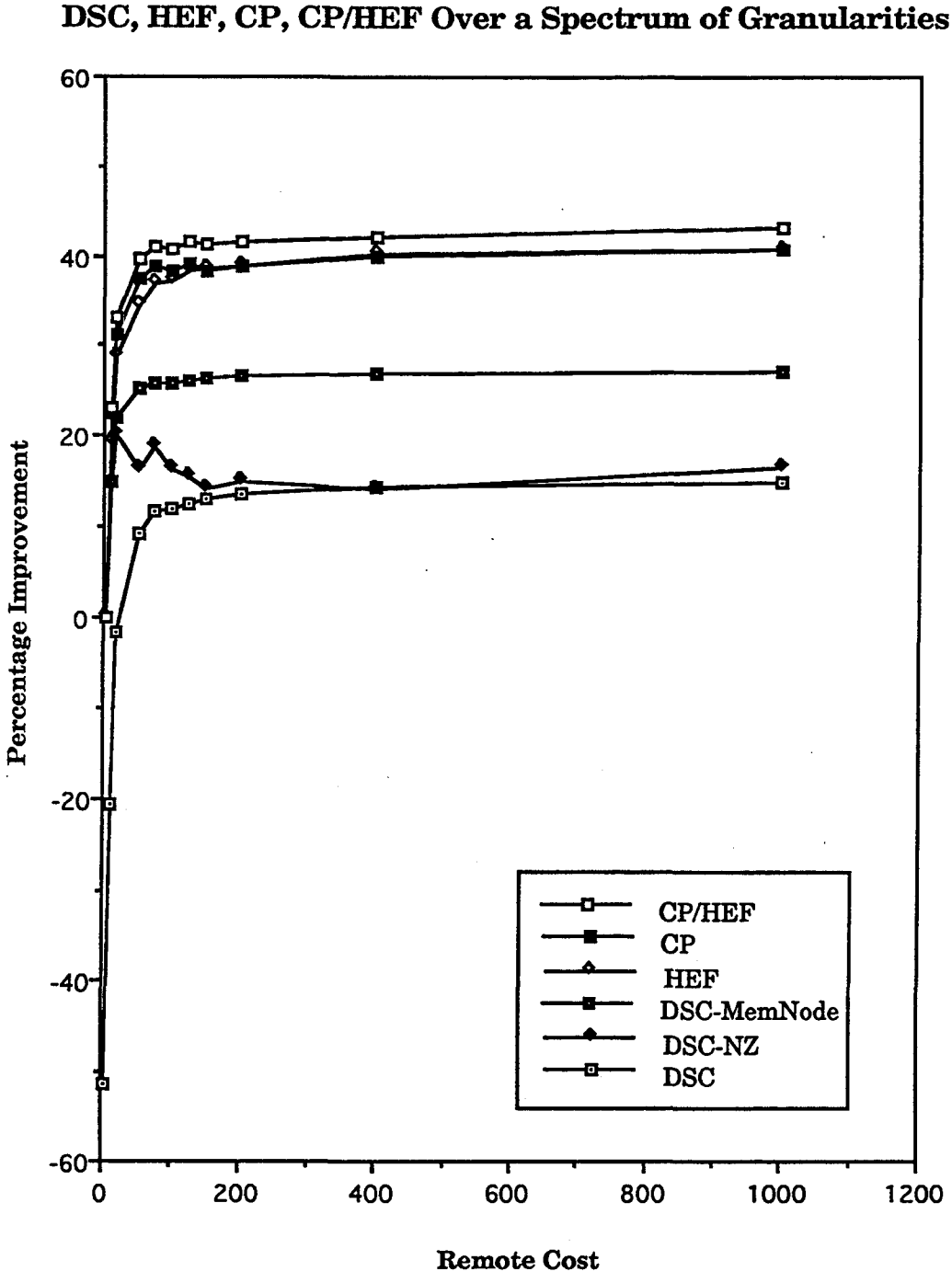


Figure 9

Conclusions for the Two-level Architecture Model

From our experiments with DSC, we conclude that the memory node model is appropriate for systems that support simple two-level communication hierarchies. The memory node version of

Algorithm HEF

Default Memory Type	Improvement
Interleaved Shared (Type 1)	57.9 %
Local Write/Remote Read (Type 2)	32.8 %
Remote Write/Local Read (Type 3)	43.2 %

Table 3 – Performance of HEF with different memory types

DSC demonstrates better performance even for coarse granularity systems where the cost of local communication effectively can be considered zero. Still better improvements are possible at the expense of increased algorithmic complexity as demonstrated by the performance of HEF and CP. As O(N^2) algorithms, however, both CP and HEF may be applied to large program graphs as their complexity is not overwhelmingly high. We therefore conclude that the memory node model, and partitioning algorithms that take advantage of the model, constitute an advance over previous work.

Multi-level Communication Architectures

Having verified that the memory node model and our partitioning algorithms work well for two-level architectures, we next wanted to investigate their utility for multi-level systems. As we were unable to find previous work which addresses multi-level communication, we present our results without comparison.

Table 3 details the mean percentage improvement of HEF for three different default memory types. Recall that HEF is an iterative improvement algorithm so it assumes all memory nodes are initially colored with a default memory type. As the amount of improvement depends on the initial critical path length, we show how HEF improves path length for each possible default type.

Merge Sort versus Sarkar's Sequencing Algorithm

We also wanted to evaluate the performance of the algorithm we propose (based on merge sort) to sequence computations within each partition. To do so, we implemented a version of HEF that uses Sarkar's `BuildSequence` algorithm [20] in place of our merging algorithm. Table 4 shows the performance of HEF using Sarkar's algorithm compared to HEF using the merge sort technique for the two-level architecture case.

Improvements

HEF with Sarkar's Algorithm	HEF with Merge Sort
30.2 %	30.0 %

Table 4 – Performnace of BuildSequence vs. MergeSort

While Sarkar's algorithm improves the performance of HEF slightly, the resulting algorithmic complexity is O(N^2 * log N) (as opposed to O(N^2) for HEF with Merge Sort). Since the overall improvement is less than 1%, we conclude that the merge sort based algorithm is viable, especially in light its lower complexity.

5.0 Conclusions and Future Work

Partitioning and scheduling techniques must improve so that parallel programs can take advantage of increasingly complex computer architectures. Previous work has relied almost exclusively on a two-level communication architecture model in which local communication implies no program execution delay. While such simplifying assumptions work well for specific architecture types, they do not adequately reflect the rich communication hierarchies present in many of today's machines. As a more general alternative, we propose the memory node paradigm which can capture the characteristics of complex systems as well as the simpler two-level architectures. To test the utility of the memory node model, we have developed two simple partitioning algorithms: HEF and CP. Although inspired by previous work, HEF and CP attempt to use the information presented by the memory node representation at the cost of some additional algorithmic complexity. While more computationally expensive than DSC [24] with its O(N * Log N) complexity, HEF and CP (each with O(N^2) complexity) are less complex than Sarkar's O(N^2 * log N) algorithm [20]. As part of HEF and CP, we propose an O(N) scheduling algorithm, based on merge sort, to order computations within each partition.

We tested the validity of the memory node model in three ways. First, we compared two versions of DSC that use the standard two-level model with a version of DSC that uses memory nodes. The memory node version yielded partitioned programs with shorter critical paths over a spectrum of architecture granularities. Secondly, we compared all three DSC versions to HEF and

CP. Again, the algorithms that used memory nodes performed best. Lastly, we examined the performance of the memory node model for a multi-level architecture. Since previous work cannot represent the various levels in the communication hierarchy, we presented our results without comparison. The results for the multi-level architecture show that our algorithms (using the memory node model) can realize good partitioning performance by considering the various levels in the communication hierarchy. We also wanted to consider the performance of the merge-sort-based scheduling algorithm used in HEF and CP. To do so, we compared a version of HEF that uses Sarkar's O(N * log N) sequencing algorithm with regular HEF and noticed no appreciable difference. Since the merge-sort-based algorithm is less complex, we conclude that it is viable.

As part of our future work, we plan to investigate how memory and communication characteristics other than simple access delay may be considered during partitioning. Resource contention, for example, is a major source of execution delay that does not seem to be adequately accounted for in the current models. We also plan to study different partitioning algorithms that use the memory node model. In particular, we want to consider how the memory node model might be useful to a task duplication heuristic such as the one proposed by Kruatrachue in [14].

Acknowledgments

The authors wish to thank Tao Yang and Apostolos Gerasoulis of Rutgers University for their guidance as we implemented DSC. We also thank Tom DeBoni for his many suggestions and improvements. This work was supported (in part) by the Office of Energy Research (U.S. Department of Energy) under contract No. W-7405-Eng-48 to Lawrence Livermore National Laboratory.

References

1. Bell, G. "Ultracomputers: A Teraflop Before its Time," *Communications of the ACM*, Vol. 35, No. 8, August 1992, pp. 26-47.

2. Cann, D., "Retire Fortran? A Debate Rekindled," *Communications of the ACM*, Vol 35, Number 8, August, 1992.

3. Cann, D.C. *Compilation Techniques for High Performance Applicative Computation*, Ph.D. Thesis, Technical Report CS-89-108, Colorado State University , Fort Collins, CO, May 1989.

4. Coffman, E.G., Jr. (Ed.), *Computer and Job Shop Scheduling Theory*, John Wiley and Son, N.Y., 1976.

5. Feo, J.T. D.C. Cann and R.R. Oldehoeft, "A Report on the SISAL Language Project," *Journal of Parallel and Distributed Computing*, vol. 12, 10 (December 1990), pp. 349-366.

6. Gerasoulis, A. and Yang, T., *A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors*, Technical Report LCSR-TR-169, Rutgers University, New Brunswick, N.J., September 1991.

7. Goopta, A. et al, "Comparative Evaluation of Latency Reducing and Tolerating Techniques," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, pp. 254-263.

8. Hu, T., "Parallel Sequencing and Assembly Line Problems," *Operations Research*, 1961, pp. 841-848.

9. Hwang, K., Xu, J., "Mapping Patitioned Program Modules on Multicomputer Nodes Using Simulated Annealing," *Proceedings of the 1990 IEEE International Conference on Parallel Processing*, 1990, pp. II-292 - II-293.

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

11. Kim, S. J., *A General Approach to Multiprocessor Scheduling*, Ph.D. Thesis, Technical Report No. TR-88-04, University of Texas at Austin, TX, February 1988.

12. Konicek, J. " The Organization of the CEDAR System," *Proceedings of the 1991 IEEE International Conference on Parallel Processing*, 1991, pp. I-49 - I-56.

13. Kruatrachue, B., Lewis, T., "Grain Size Determination for Parallel Processing," *IEEE Software*, January 1988, pp. 23-32.

14. Kruatrachue, B., *Static Task Scheduling and Grain Packing in Parallel Processing Sytems*, Ph.D. Thesis, Oregon State University, Corvalis, 1987.

15. Lewis, T. "Task Grapher: a Tool for Scheduling Parallel Program Tasks" *Proceedings of the 5th Distributed Memory Conference*, April 1990, pp. 1171-1178.

16. McCreary, C., Gill, H., "Efficient Exploitation of Concurrency Using Graph Decomposition," *Proceedings of the 1990 IEEE International Conference on Parallel Processing*, 1990.

17. McCreary, C., Gill, H., "Automatic Determination of Grain Size for Efficient Parallel Processing," *Communications of the ACM*, Sept. 1989, pp. 1073-1078.

18. 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.

19. Ranelletti, J. E., *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages*, Lawrence Livermore National Laboratory Technical Report UCRL-53832, Lawrence Livermore National Laboratory, Livermore, CA, November 1987.

20. Sarkar, V., *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*, Ph.D. Thesis, Technical Report No. CSL-TR-87-328, Stanford University, 1987.

21. Sarkar, V. and Hennessy, J., "Compile-time Partitioning and Scheduling of Parallel Programs," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, July 1986, pp. 17-26.

22. 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.

23. Warren, K. and Brooks, E. "Gauss Elimination: A Case Study,"*Proc.,COMPCON 91,* San Francisco, CA, February 1991, pp. 57-61.

24. Yang, T. and Gerasoulis, A., "A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors," *Proceedings from Supercomputing "91*, November 1991, pp.

Mapping Functional Parallelism On Distributed Memory Machines*

Santosh S. Pande, Dharma P. Agrawal and Jon Mauney
North Carolina State University
Raleigh, NC 27695-7911

Abstract

In order to effectively use the power of newer massively parallel multicomputers, exploitation of functional parallelism in a program with automatic data and code distribution, is a necessity. Functional paradigm offers a relative ease of generating an intermediate form that abstracts the DAG parallelism in the program, in an interprocedural framework.

Using the Sisal intermediate form IF-2, a new task-based methodology has been outlined to effectively map functional parallelism on distributed memory multicomputers. A new compile time method that investigates the trade-off between the schedule length and the number of required processors, partitions the IF-2. The partition is appropriately scaled at run time to match the available processors, to avoid recompilation.

The run time system is designed around an arbitrator processor and several worker processors. The arbitrator processor manages the control thread of the Sisal program and is also responsible for the housekeeping jobs like task assignment to the workers and ownership resolution of the task variables. Each of the workers await tasks to be assigned by the arbitrator and use the ownership information maintained at the arbitrator to obtain the live definition of a variable.

Many implementation issues related to supporting call by value and virtual address mechanism are addressed. Benchmark results are presented to demonstrate the viability of the approach.

*This research was supported by the U. S. Army Research Office under grant no. DAAL03-91-G-0031

1 Introduction

1.1 Distributed Memory Machines

Some of the reasons for the popularity of distributed memory machines are:

- They are highly scalable. These machines currently come in a variety of architectures like mesh, hypercube and ring, in which more processors could be easily added if the computing demands increase.
- With larger number of processors, there is virtually no performance degradation. One of the problems with the shared memory systems is that the memory bandwidths may not match the increase in number of processors. In fact, the overall system performance might degrade, due to the increased memory contention.
- For large parallel applications like Fluid Flow, Weather Modeling, and Image Processing in which the problem domains are perfectly decomposable, the achievable speedups are almost linear in the distributed memory systems. This is primarily due to fast accesses to the local data maintained in private memory of each processor.
- Distributed memory machines are relatively cheap as they do not require any specialized hardware for memory arbitration, and processor-memory interconnection. They offer more Mflops per dollar value.
- With the advent of new techniques like wormhole routing, and resulting very high interprocessor communication speeds, the newer Mflop ratings offered by distributed memory machines are comparable to the shared memory systems. e.g. Intel Touchstone iPSC/860.
- Fine grained parallelism could be profitable mapped on the newer systems like the Intel Paragon due to a very low communication/computation ratio.

However, programming distributed memory systems still remains very complex. The lack of proper software support is the main concern of many programmers.

Tn this paper, we present a methodology to compile Sisal on distributed memory machines. Section 2 surveys the different program partitioning techniques proposed in the literature. Section 3 illustrates our compile time partitioning method through an example. Section 4 addresses the code generation and the temporary management issues for distributed memory machines. Section 5 discusses the run time system. Section 6 discusses the mode of execution of Sisal using our methodology, and section 7 concludes the paper.

2 Program Partitioning Issues

Compiling programs on a distributed memory machine could be visualized as a resource allocation problem [10]. In particular, the problem of program partitioning on distributed

memory multiprocessors, has been attempted by many researchers, and the approaches could be mainly classified as:

1. Data Driven Code Partitioning, and,
2. Code Based Data Allocation.

In data driven approaches, the program data is partitioned for different processors, and the code is generated so that the generated data references are locally available. This results in lesser communication on distributed memory machines, which is a costly operation. The code based approaches, on the other hand, carry out the partitioning so that each processor approximately gets an equal share of the program code. The resulting data references, are then examined, and data is allocated to different processors to reduce communication. The goals of locality of data and load balanced code could be conflicting, and for certain types of codes impossible to achieve simultaneously [12].

The approach adopted by Kennedy et al. [6] falls under 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 – D standing for data distribution. They define compile time data domains to map the aggregate (mainly array) slices to local memory. The user is responsible for specifying such a data layout. The compiler then supports a virtual address mechanism to correctly map the global references to local ones. The code generation phase ascertains that the references in the computation are correctly mapped to the local memory. For example, by using the decomposition functions, the user could write following matrix multiplication program:

```
      PROGRAM MULT
      REAL A(256,256), B(256,256), C(256,256)
      READ *, N
      DECOMPOSITION V(256,256)
      DISTRIBUTE V(BLOCK_CYCLIC)
      ALIGN A,B,C with V
DO 10 I=1,N
 10       READ *,(A(I,J), J = 1,N)
      DO 20 J=1,N
 20       READ *,(B(I,J), J = 1,N)
      DO 30 I=1,N
            DO 30 J=1,N
                  C(I,J) = 0.0
                  DO 30 K=1,N
 30                     C(I,J) = C(I,J) + A(I,J)*B(K,J)
      DO 40 I=1,N
 40       PRINT *,(C(I,J), J=1,N)
      STOP
      END
```

In the above example, the function, DECOMPOSITION defines a virtual array V. Another function DISTRIBUTE defines the virtual mapping of elements of V on the processors. In this example, the above distribution implies that, processor 1 owns all the elements in rows 1 to (N/P) and columns 1 to (N/P), processor 2 owns the all the elements in rows 1 to (N/P) and columns (N/P+1) to (2N/P) and so on. The ALIGN function defines a mapping between the indexing of the real array and its corresponding virtual array. In this case, the mapping is idempotent i.e. V(I,J) means A(I,J), B(I,J) and C(I,J). The compiler uses these functions to examine the references and their ownerships and generates correct code to get the values from other processors, if needed. The code generation is driven by the *owner computes* rule with a careful application of determining the ownerships using distributed variables and reductions resulting from the preceding dependence analysis phase.

Pingali et al. [12] 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 could be obtained using run time ownership resolution. Consider the following piece of code in which the user specifies variable 'a' to reside on processor P1, and variables 'b', and 'c' to be on P2 and P3 respectively:

```
a:P1, b:P2, c:P3;
S1: a := 5;
S2: b := 7;
S3: c := a+b;
```

In compiling S1, it is known that P1 is the sole participator and evaluator, and for S2, P2 is the sole participator and evaluator. However, for compiling S3, the definitions of both a and b are needed, that make P1, P2, and P3 participators (P3 supplies the definition of '+' operator). P3 is the sole evaluator for S3. This results in the following code to be generated:

```
P1:                     P2:                     P3:
a := 5;                 b := 7;                 t1 := receive(a,P1);
send(a, P3);            send(b, P3);            t2 := receive(b,P2);
                                                t3 := t1 + t2;
```

In a more 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[9].

The ongoing project C* [14] rely on user partitioning of data aggregates on SPMD machines like CM-5, and fall under scheme one. For example, the data-parallel program can look like:

```
domain vector { real a, b, max; } x[100];
...
[domain vector].{
```

```
        if (a > b) then max = a;
        else max = b;
}
```

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 selection 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.

Koelbel et al. [8] carry out a semi-automatic 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 due to Mansour et al. [3] that stresses, obtaining 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 the partition is judged on the basis of an objective function that determines the locality of references. Other approaches have come from the efforts in porting the shared memory partitioning techniques to the distributed memory machines.

Some researchers have also attempted combining both the above approaches. For special cases of DO loops with constant dependence distances, Ramanujan et al. [13] have devised a novel scheme that attempts to achieve a communication free partition of a loop, if it exists. For example, consider the following DO 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, for defining each element of A[i,j], two elements of array B are needed. It can be easily seen that if both array A, and B are divided, along their anti-diagonals, communication free partition of the loops is achieved. On the other hand, for the loop below, no such partition can be found.

```
for i = 2 to N
   for j = 2 to N
     A[i,j] = B[i-2,j] + B[i-1,j-1] + B[i-1,j-2]
```

Gajski et al. [4] 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 semi-automatic approaches save a lot of effort on the compiler's part. However, these approaches might not always yield a good program partition due to following reasons:

- Data driven code partitioning approaches rely heavily on the user judgement in correctly partitioning the data.
- Many 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.
- Almost all of the 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 are used to drive the code generator by many compilers. The type of data distribution supported is thus, *block*, *cyclic*, and *block cyclic*. These distributions 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, due to an unbalanced code if the preprocessing dependence analysis stage does not properly discover the distributed variables. Dependence analysis is extremely tough and imprecise in an imperative framework leaving these approaches questionable.
- Strictly code driven approaches also suffer from the communication overhead, that would nullify all the benefits of parallelism.
- Approaches that tend to combine both the 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 are not satisfied at compile time.

2.1 DAG Parallelism And Functional Paradigm

One of the major limitations of all the above approaches is that they deal primarily with the loop based parallelism. To exploit the high degree of parallelism available in newer massively parallel distributed memory machines, the compiler must be capable of using more general DAG parallelism present in the program. However, extracting DAG parallelism demands extensive interprocedural dependence analysis. Such an 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. 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 speed and code-size [1].

A few research efforts have attempted 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 [15] uses a compile time cost model to analyze the trade-off between task overhead and task granularity. 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 have large number of processors. This issue along with the efficient task management mechanism on a distributed memory machine 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, which is then suitably scaled to the available number of processors 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 iPSC/860 (Refer to [2] for details about Sisal).

3.1 Task Representation Of Sisal

Sisal uses IF-2, a dataflow intermediate form for its internal representation of programs. IF-2 mainly consists of nodes that represent computation and edges that carry the data values. Each of the nodes has a set of input edges that carry input arguments, and a set of output edges to carry output arguments. IF-2 consists of three types of nodes:

- Simple Nodes, that define a variety of dataflow operations,
- Graph Nodes, that define scoping rules for the dataflow values passed from one node to another, and
- Compound Nodes, that consist of subgraphs. The semantics of a given compound node defines the manner in which its different subgraphs interact.

Our scheme uses the dataflow parallelism present in IF-2 graphs. We illustrate it by quoting an example given in IF-1 reference manual [16]. Figure 2 gives its IF-2 representation, of the following Sisal program:

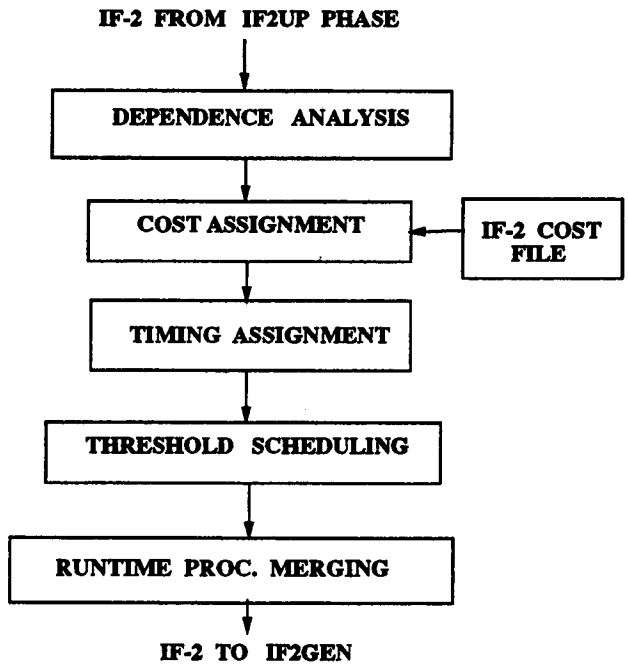


Figure 1: Threshold Partitioner

```
import g(I, J : integer returns integer)
function f(A, B, C, D : integer returns integer)
       let X, Y :=
              if A+B < C+D then
                 A, B
              else
                 C, D
              end if
       in g(X, Y)
       end let
end function % f
```

The Select is a Compound Node of IF-2, that consists of three subgraphs for implementing if...then...else condition. Depending on the result at the output port of the subgraph 0, either of the subgraphs 1 or 2 are evaluated. The Simple Nodes 1, 2, 3, and 4 in Select Subgraph 0 and Call Node 1 represent tasks and the edges connecting them represent task precedences. The outermost graph represents a scope boundary of function f in the example. Our scheme first schedules the Select Subgraph 0 and computes the two 'Plus' nodes in parallel if such partitioning is found beneficial by the partitioner at the compile time and then evaluates the 'Less' and 'Int' on one of the processors that computes the preceding 'Plus'. When the results of the Subgraph 0 are available, one of the Subgraphs 1 or 2 is scheduled.

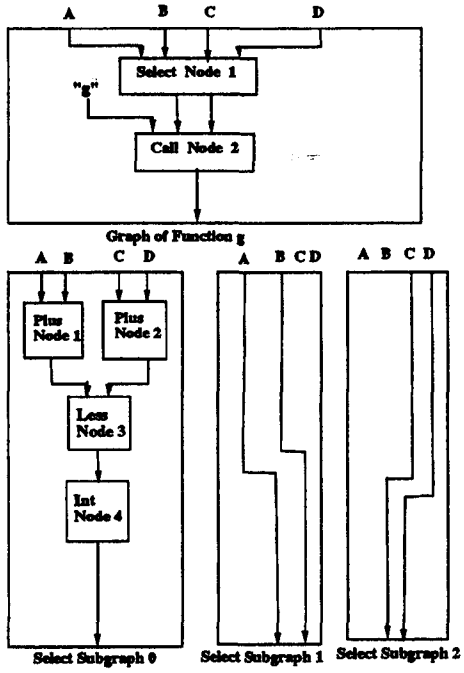


Figure 2: IF-2 Representation Of Example Program

The decision whether it is beneficial to parallelize the two 'Plus' nodes and which processor is to execute 'Less' and 'Int' is taken by using the computation costs of these operations and the communication costs of their operands by performing a compile time analysis.

3.2 Partitioning Strategy

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 assignment based on iPSC/860 timings (Refer to Figure 1) [11]. These cost assignment correctly identify the relative tradeoff between the communication and the computation that mainly dictate data vs. code partitioning issues [7].

The partitioning phase works partially at compile time and partially at run time. At compile time, the partitioner computes **partition margin** as the difference between the earliest and the latest possible schedule times of each task. The tasks are assumed to be strict. i.e. a task becomes *ready to run* only when **all** of its predecessor tasks have completed. A **threshold** value is then globally selected within the minimum and the maximum partition margins of all the tasks, and an attempt is made to map a task on a predecessor task's processor. A merit function is used to carry out the allocation if more than one tasks compete for the same processor. One of the two minimization schemes can be selected by the user:

- Scheme A: Compiling for minimum program completion time– Suitable for large sys-

tems.

- Scheme B: Compiling for minimum number of processors– Suitable for small systems.

The threshold is varied between the minimum and the maximum values of the partition margins of all the tasks, and the best threshold value that satisfies given option is found, and is used to generate the partition. In addition, four options are offered to keep compilation time under user control by changing the manner in which the threshold is varied :

- Adaptive Partitioning : Number of thresholds chosen to keep the compilation time almost constant,
- Fixed Step Partitioning : Fixed number of thresholds chosen specified by the user,
- Fixed Delta Partitioning : Fixed increment of a threshold is chosen specified by the user, and
- Discrete Threshold Partitioning : Threshold values are chosen as found on each of the input edges of a task.

Refer to the example in Figure 3. It shows a task graph created from the following code fraction:

```
...
funct1(arg1, arg2, arg3, arg4, arg5, arg6);
funct2(arg4, arg7);
funct3(arg5, arg8);
funct4(arg6, arg9);
funct5(arg7, arg8, arg9, arg10, arg11, arg12);
funct6(arg10, arg13);
funct7(arg13, arg11, arg14);
funct8(arg12, arg15);
funct9(arg14, arg15);
...
```

Let the execution costs of each of the nodes of the task graph in Figure 3 be as follows:

```
c(1) = 100, c(2) = 35,  c(3) = 75,
c(4) = 5,   c(5) = 150, c(6) = 83,
c(7) = 22,  c(8) = 25,  c(9) = 80.
```

Let the communication costs of each of the edges be:

```
c'(1,2) = 50, c'(1,3) = 20, c'(1,4) = 15,
c'(2,5) = 17, c'(3,5) = 30, c'(4,5) = 90,
c'(5,6) = 28, c'(5,7) = 25, c'(5,8) = 66,
c'(6,7) = 14, c'(7,9) = 87, c'(8,9) = 80.
```

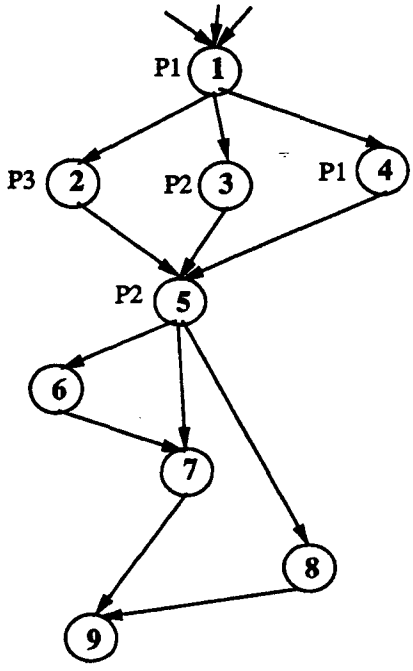


Figure 3: Example Task Graph

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 *ready to run* 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 corresponding partition-margins(pm) are found as follows:

```
pm(1) = 0,  pm(2) = 50, pm(3) = 20,
pm(4) = 15, pm(5) = 30, pm(6) = 58, and,
pm(7) = 72, pm(8) = 30, pm(9) = 157.
```

Thus, the threshold is varied between 0 and 157, in the manner described in each of the above schemes.

Suppose a threshold value 50 is being used. First task 1 is scheduled on P1. 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 P1, 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 (mf) of task t on processor p is found as follows:

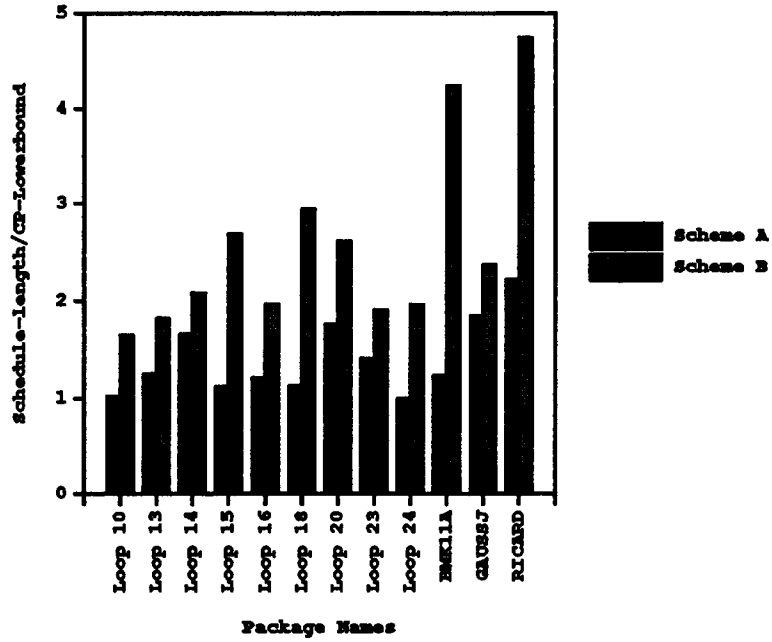


Figure 4: Discrete Threshold Scheme: Schedule Length Results

mf(t,p) = Task delay of t on p - Processor completion delay of p due to t.
The merit functions for 2, 3, and 4 are:

```
mf(2,P1) = 0 - 35 = -35,
mf(3,P1) = 0 - 75 = -75, and
mf(4,P1) = 0 - 5  = -5.
```

Since, the merit function of 4 is the highest, task 4 is allocated to P1. Next, the tasks 2 and 3 are allowed 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. Next, the task 5 could be allocated to either of the P1, P2, or P3. However, allocating task 5 on processor P2, results in the least schedule delay (the difference between the actual schedule time and the earliest possible schedule time). For task 5, the schedule time is 202, the earliest schedule time is 195, giving a schedule delay of 7, which is below the threshold. The processor assignment is carried out in this manner for all the thresholds, and the best value is chosen to satisfy the given goal in Schemes A or B.

Figures 4 and 5 give the results of the partitioning schemes for many numerical packages in terms of the schedule length and the average processor utilization found using a compile time cost analysis.

The second phase of the partitioner is carried out at run time. It remaps the tasks, producing a partition for a smaller number of processors than used at compile time. This phase merges the task lists of the earliest completing processors (determined according to compile time partition) allowing the program to scale to the processors actually available at

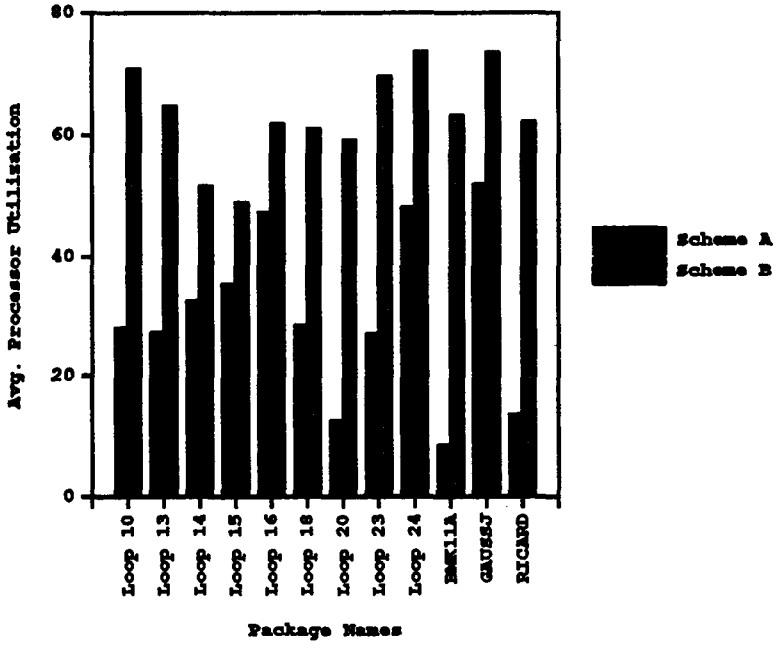


Figure 5: Discrete Threshold Scheme: Processor Utilization Results

run time. Figure 6 shows the variation of program completion time (schedule length) with the number of processors found for fewer available processors. It can be easily seen that this scheme produces a partition, that is almost linearly scalable with the number of processors.

4 Code Generation Technique

The code generation phase provides the support for call by value semantics on distributed memory machines, and a virtual addressing scheme for mapping global references to local addresses. The initialized global variables, and literals are replicated on all the processors. A processor called arbitrator manages control thread of the Sisal program, the task allocation and housekeeping jobs. Each of the other processors called worker is responsible for executing the assigned tasks.

4.1 Temporary Management

The run time management supports the call by value semantics through matching names. On distributed memory machines, this approach could improve temporary management, since the temporary physical space is now local than shared global. One of the possibilities is, that the temporaries are allocated to each of the function arguments through a pool of local names maintained for each of the workers individually, rather than through a global name space.

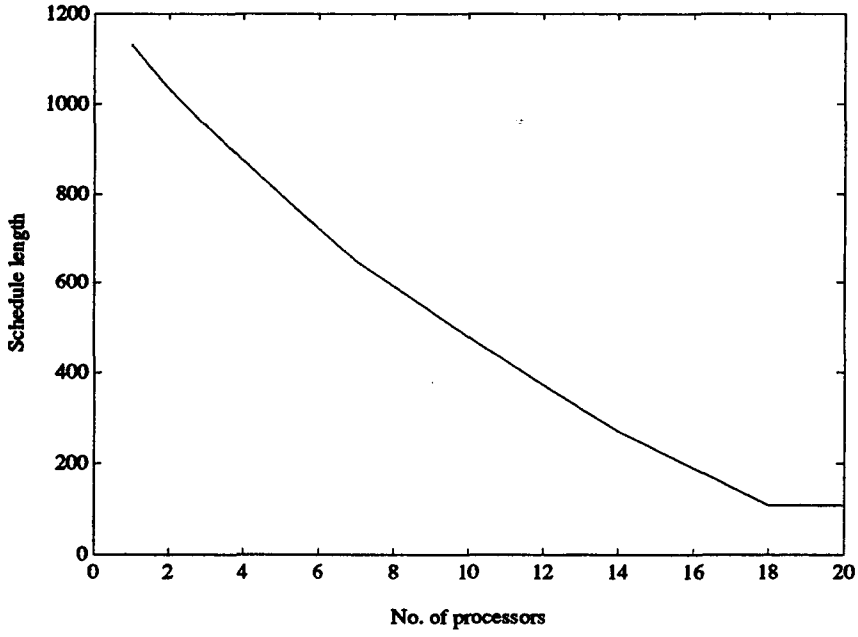


Figure 6: Discrete Threshold Scheme: Variation of Completion Time With Processors for Gauss-Jordan Method

Each of the global reference, then uniquely maps to a local name. This translation could be done at compile time itself, since the worker binding of a task and thus the ownership of each of its arguments is known. However, there are two specific drawbacks of the above scheme:

- Run time task merging is allowed to scale the code to fewer available number of workers. This implies that the worker bindings of tasks, and their owned variables could change at run time and compile time translations may not work.
- Task migrations are allowed at run time, again leading to the above problem.

Thus, to alleviate these two problems, a mechanism is required at run time that maintains a global-local translation (a local symbol table). The following two issues could visualized with a strictly local symbol table scheme:

- Modifications in the ownership of a given variable have to be reflected in all the non-local copies, e.g. redefinition of a variable on another worker.
- Due to the task migrations, the ownerships could change. The above coherency problem again arises.

Thus, the ownership analysis based on a local symbol table cannot be carried out, unless consistency is maintained among the copies with all the workers. This problem is similar to

that of the cache coherency. Thus, with every change in any of the local symbol tables, a global update has to be carried out. This solution is very inefficient. A simple solution is proposed, by carrying out a virtual address translation at each of the workers. The global temporary reference by name is the same as the local one. An address mapping of this name is maintained in the local symbol table. Thus, the local symbol table gives us a global name–local address translation.

The code generator is modified to support call by value semantics for non-local definitions. The ownership information for each of the function's input arguments is maintained in a central symbol table at an arbitrator, and is passed to the worker through an appropriate input AR. If this variable is owned by a worker, it is looked up in the local symbol table to get its appropriate physical address. If not, then the ownership information is found using input AR, and the proper value is obtained from its owner. Once such a mapping is done by the utility routine, the function then executes, and creates its set of output arguments with proper reference counts. The output argument names and their physical addresses are then entered in the local symbol table. The function after completing the execution, returns an appropriate output AR that describes the ownership information of its output arguments, as needed by the arbitrator to update the central symbol table. Thus, in an abstract sense, a generic function can be designed, that executes like this on each of the workers:

```
doexecute(f, inargs, outargs)
   begin
     get_my_inargs(inargs);
     exec(f, inargs, outargs);
     send_my_outargs(outargs);
   end
```

The only problem with the above scheme is that compile time space allocation cannot be performed, for any temporary, since at run time, it would not be known *a priori*, as to which temporary is physically going to reside where. Thus, the physical space is allocated, for each of the task's input/output arguments at the run time. This gives us some run time overhead coupled with the local address translation. Alternatively, the space could be allocated, at compile time to the input/output arguments of the assigned tasks, on the respective workers. For the migrated tasks, a run time space allocation could be carried out. Another simple solution is to keep global name space the same as the local, and allocate space at compile time itself for all the globals on all the workers. This avoids the local address translation, too. This simplistic solution is inefficient, since a lot of space is wasted on each of the workers. However, in space-time trade-off we opt for time and allocate the space at compile time.

5 Run Time Support

A central symbol and task table is supported, to be maintained by the arbitrator, to perform the ownership assignments of the task input arguments, to decide about the assignment of the ready to run tasks by using dependencies, and to perform the task migrations to balance the

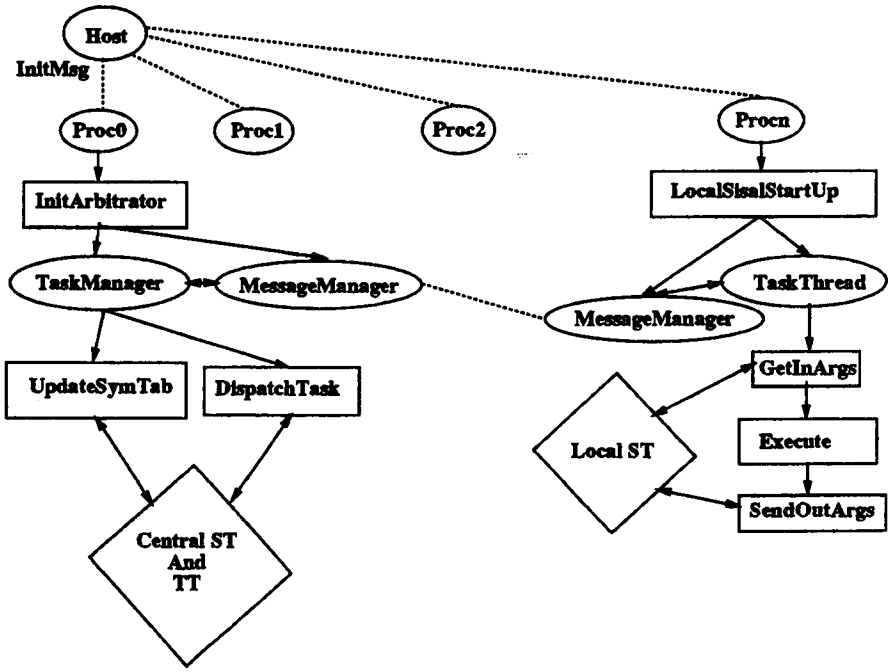


Figure 7: Run Time Support

compile time discrepancies (Refer to Figure 7). On startup, the host loads all the processors with a copy of the program and sends an 'init' message to all the processors. The processor zero acting as arbitrator, initializes the central symbol table, carries out the merge of the task lists to match the available workers, creates appropriate worker bindings for the respective tasks, and initializes the task dependency list. It then, spawns two processes:

- Task Manager: This is responsible for various housekeeping jobs like updating symbol table, task dependences, assigning and migrating tasks.
- Message Manager: This is responsible for managing the interface to the workers. Its job is to send input ARs, receive output ARs, and inform the Task Manager of various activities.

The Task Manager is divided into two logical modules. The UpdateSymTab manages the ownerships, and the DispatchTask manages task lists. These two modules interact using an interrupt driven handler invoked by the Message Manager.

Each of the processors other than processor zero acting worker also spawns two processes:

- Task Thread : This is responsible for executing tasks.
- Message Manager : This is responsible for receiving input ARs, sending the output ARs, and getting and sending values of nonlocal and local variables.

6.1.2 Array And Stream Manipulating Nodes

The stream referencing is non-strict, to allow pipeline parallelism. In other words, both the producer and the consumer tasks of the streams could be scheduled together. The streams are segmented using an optimization [7] to obtain maximum computation/communication overlap. Similarly, within a loop non-strict array production and consumption is allowed so that the generator subgraph of the loop could feed the values to the loop body in a non-strict manner. Also the array values generated in the loop body could be used by the result subgraph in a non-strict manner. This allows the vertical parallelism in a loop to be used.

6.1.3 Record And Union Manipulating Nodes

The three operations: create, replace, and extract result in different actions. The create operation creates a record or a union and sends an output descriptor to the arbitrator to indicate that the corresponding worker holds the live definition. Similarly replace operation updates a given field of the record or union and notifies the arbitrator of a changed ownership of the given data object. Extract operation simply retrieves a given field of the record or union and saves it in a temporary notifying its ownership to the arbitrator. The consumer then obtains the ownership information of this temporary through the arbitrator and gets the correct definition.

6.1.4 Gather/Scatter Nodes

These nodes occur in the generator or in the result subgraph of the loop. Similar to streams, we allow a non-strict execution of these nodes to allow a partial overlap of the values needed and generated by the loop body. The amount of segmentation is found using the optimization described in [7].

6.1.5 At Nodes

The At nodes denote operations like allocating array space, filling it with a given element, managing the underlying buffer for different data objects and many different memory related operations. They are assigned to the worker where the data object's live definition is to be maintained. For example, at the time of creation of a given array the AddLAT or AddHAT will be assigned to a worker that is the first owner of the created array. At a subsequent point in program execution, if another AT node wants to modify the data object, it will first get the the data object's live definition and after performing the modifications, proclaim the new ownership to the arbitrator.

6.2 Compound Graphs

Compound nodes of the IF-2 demand a special attention since they decide the control flow of the Sisal program. As noted earlier, the control decisions are made at the arbitrator. Appropriate sub-graphs of the compound nodes return the necessary variable values to the

The arbitrator after carrying out above initializations, dispatches appropriate tasks to the workers through its Message Manager. The assignment of tasks is decided by the compile time worker bindings in conjunction with the merge performed to match the available number of workers. In addition to this, the Task Manager runs an algorithm that tries to migrate the tasks at run time to achieve the load balance. The Task Manager, then waits for the output ARs to arrive from the completed tasks, and updates the symbol table ownerships and ready to run task list. Finally, when all the tasks are completed, the Task Manger sends a kill message to all the workers and then kills itself.

On the worker side, following events occur. Each Task Thread waits for an input AR to arrive through its Message Manager. It then uses the ownership information in the input AR to find out the definitions of its input arguments, and sends messages to the respective workers to get non-local values. The local symbol table is used to get the addresses of locally available variables. The task is then executed, and the ownerships of the output values are reported in the output AR to the arbitrator, and also entered in local symbol table. When the worker receives a 'kill' message from the arbitrator, the thread terminates.

6 IF-2 graph execution

The arbitrator after initializing the task and symbol tables starts the execution of SisalMain() that starts the Sisal control thread. As it proceeds the execution of the different types of IF-2 nodes takes place as under:

6.1 Simple And At Nodes

Whenever the control thread executing at the arbitrator encounters a Simple or At node, it finds the worker binding of that node and dispatches an appropriate descriptor to that worker. The control thread then waits for the completion of the execution of the node before attempting to schedule its successors [1]. Once a node is scheduled on a worker, the worker is responsible to get the input arguments of the node using the ownerships in the input descriptor. The ownerships of the output arguments are sent in the output descriptor.

6.1.1 Arithmetic and Boolean Nodes

Input arguments are obtained from that worker where the live definitions reside and the results are saved in local variables after performing the operation. The output values are sent in the output descriptor only if the control thread at the arbitrator has demanded it; otherwise just an ownership descriptor is returned for ownership update.

[1] A node is ready for execution only if all of its predecessors have completed their execution.

arbitrator which then evaluates a condition to determine the flow of control. We examine different compound nodes one by one:

- **Select:** First the arbitrator control thread schedules the 'Selector' subgraph of the 'Select' compound node. It dispatches the Simple and the AT nodes contained within this subgraph and once the result of 'Int' node is available at the arbitrator, it determines the appropriate 'Alternative' subgraph to be evaluated and schedules it.

- **Tagcase:** The arbitrator examines the tagcase variable for a match with a tag of one of the subgraphs and schedules that subgraph.

- **Forall:** The Forall compound node is evaluated in a non-strict manner to allow partial overlap between the 'Generator', 'Body' and the 'Result' subgraphs. The 'Generator's supplies a chunk of values to the nodes in 'Body' subgraph that consume them and produce a chunk of results to be supplied to the 'Result' subgraph. The chunk size is decided by the rate of the production and consumption of the values in respective subgraphs and the message passing delay. For details of this technique, please refer to [7].

- **LoopA:** The arbitrator first schedules the 'Initialization' subgraph and distributes its Simple and AT nodes. After the evaluation is complete, the arbitrator schedules 'Body' and 'Result' subgraphs together in a non-strict manner to evaluate the first instance of the loop. When the results from the first instance are available it schedules the 'Test' subgraph for checking the loop exit condition. If the exit condition is false, the arbitrator schedules the 'Body' subgraph for the next instance and so on. Whenever the exit condition is true, the arbitrator completes the execution of the loop.

- **LoopB:** It executes in a manner similar to LoopA, except that the 'Test' subgraph is scheduled earlier than 'Body' and 'Result' subgraphs.

7 Conclusion

We have presented a methodology for compiling Sisal on a distributed memory machine using the functional parallelism. The efficiency of this type of approach will be largely determined by the communication costs on the target machine and the task overheads resulting due to call by value semantics. However, with the advent of the newer systems like the Intel Paragon that have a very low communication/computation ratio, we expect that this approach will become viable.

Acknowledgement

The authors gratefully acknowledge the support of Lawrence Livermore National Laboratory for allowing the use of Sisal compiler source code.

References

[1] D. C. Cann, "Retire Fortran? A Debate Rekindled", CACM, August 1992, Vol. 35, No. 8, pp. 81-89.

[2] 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.

[3] 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.

[4] D. Gajski, and J. Peir, "Camp: A Programming Aide for Multiprocessors", *Proc. Int'l Conf. On Parallel Processing*, August 1986, pp. 475-482.

[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] S. Kim, S. S. Pande, D. P. Agrawal, and J. Mauney, "A Message Segmentation Method to Minimize Task Completion Time", *Proc. 1991 Int'l Symposium on Parallel and Distributed Processing*, April-May, 1991, pp. 519-524.

[8] 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.

[9] W. Li, and K. Pingali, "Access Normalization: Loop Restructuring for NUMA Compilers", Technical Report, TR 92-1278, Cornell University.

[10] J. Mauney, D. P. Agrawal, et al., "Computational Models and Resource Allocation for Supercomputers", *Proceedings of the IEEE*, Vol. 77, No. 12, January 1990, pp. 1859-1874.

[11] 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).

[12] K. Pingali, and A. Rogers, "Process Decomposition through Locality of Reference", Technical report TR88-935, Department Of Computer Science, Cornell University, August 1988.

[13] J. Ramanujan, and P. Sadayyapan, "Access Based Data Decomposition for Distributed Memory Machines", *Proc. 6th Distributed Memory Computing Conf.*, April 1991, pp. 196-199.

[14] J. Rose, and G. Steele, "C*: An Extended C Language for Data Parallel Programming", Technical Report PL87-5, Thinking Machines Inc.

[15] 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.

[16] S. Skedzielewski, and J. Glauert, "IF1 – An Intermediate Form for Applicative Languages", Computing Research Group, Lawrence Livermore National Laboratory, 1985.

Implicit Array Copying: Prevention is Better than Cure

Paul Roe and Andrew Wendelborn

Department of Computer Science,
University of Adelaide, GPO Box 498, Adelaide, SA 5001
E-mail: proe, andrew@cs.adelaide.edu.au

Abstract

SISAL is a purely functional language which supports arrays. The difficulty with supporting purely functional arrays is that a prohibitive amount of copying can be necessary. SISAL uses a sophisticated compile-time analysis to alleviate some copying associated with array operations. This compile-time analysis is discussed and some problems with it are exposed. The alternative to curing copying, is to prevent it. To this end we propose a linear ADT, inspired by Wadler's monads work. Our linear ADT may be incorporated into Haskell and other functional languages. It allows efficient functional assignment on arrays. In addition it allows efficient divide and conquer style algorithms to be expressed using array partitioning and concatenation. The latter is particularly useful for expressing parallel array algorithms.

1 Introduction: the copying problem

SISAL is an applicative language designed for scientific processing, and in particular it supports arrays. It is a pure language which does not permit side-effects: there is no destructive assignment in SISAL. This paper assumes the reader is familiar with SISAL [14] and functional programming [2].

SISAL supports several functions on arrays. Functions such as indexing, and size and bounds enquiry present no problems. However other functions such as replace (e.g. `a[i:3]`), array add (e.g. `array_addh (a,e)`) and concatenate (e.g. `a||b`) can result in lots of copying. The copying arises because each of these functions must construct a new array. For example `a[i:3]` creates a new array the same as a except element `i` has the value 3. A straightforward implementation will copy the array a and initialise element `a[i]` to 3. In an imperative language such as Fortran, a programmer would either modify

the array using assignment, or the array would be explicitly copied, and element `i` in the copied array would be initialised to 3.

Notice that the problem is copying and *not* excess storage use (especially if an implementation performs reference counting). Consider the the evaluation of an expression such as `a[i:3]`: an array will be allocated to hold the expressions result, the values of a will be copied into the new array and element `i` of the new array will be initialised to 3. If the expression `a[i:3]` occurs in a context where it contains the last reference to a, then once the expression has been evaluated the storage for a can be freed. Thus only during the evaluation of `a[i:3]` are both the arrays, a and its copy, resident in store.

Array copying as described can make drastic changes to the efficiency of an algorithm. For example insertion sort, an $O(n^2)$ algorithm, becomes an $O(n^3)$ algorithm, if array replace is used. This copying problem effectively discourages the use of replace, array add and concatenate operations, which makes many algorithms hard to express.

Note, there have been some proposals which aim to circumvent the copying problems associated with functional arrays by providing sophisticated implementations of arrays. These implementations use special representations for arrays, such that arrays do not need to be copied in their entirety. Unfortunately these implementations negate some desirable properties of arrays: namely constant time access to elements and a compact representation in contiguous storage. Bloss reports on some experimentation with one of these implementation techniques (trailing) in [3].

2 A cure: compile-time analysis

A lot of research has gone into devising compile-time analyses which recognise when replace operations may be implemented using destructive assignment [3, 5, 7, 9]. Effectively all these analyses perform some form of reference counting at compile-time. This enables the compiler to determine that some arrays become completely dereferenced after a replace operation and hence the replace operation may be correctly implemented by modifying the original array. For example if it can be determined that in *all* contexts after the evaluation of the expression `a[i:3]` the array a becomes completely dereferenced; then the replace operation may be implemented by simply modifying the array a using destructive assignment (in imperative programming terms `a[i] := 3`). Crucially, no copying is required. Of course such analyses are only approximate and cannot detect all expressions where replace may be implemented by destructive updating.

The array add function presents further difficulties since, unlike the replace operation, its result is larger than its argument array. Cann [5] has shown how some iterated array adds may be bounded using an analysis called build-in-place analysis. This means that storage may be allocated once in a single block, resulting in no copying.

Rather than defining array algorithms in terms of replace, array partition and concatenate may be used to define divide and conquer style algorithms. This is particularly useful

for defining parallel algorithms. Array concatenate usually copies its two arguments into a new array. Sometimes when concatenate is used in conjunction with array partition operations, copying is unnecessary. Gopinath has developed an analysis to determine when this optimisation is permissible [7].

3 Problems with compile-time analysis

The basic problem with using compile time analysis to optimise replace and other operations is that the operational behaviour of programs is no longer evident from their text. It is hard to write efficient programs for compilers which perform sophisticated optimisations unless the optimisations are very easy for the programmer to recognise[1]. Writing portable efficient programs becomes practically impossible: since different compilers will perform different optimisations. Generally programs exhibit a loss of operational clarity and it becomes hard to reason about programs' efficiency. In addition compile-time analysis does not work well with separate compilation because such analyses are global.

In the OSC manual [6] in the section on "Warnings, hints and Recommendations", the advice given to the SISAL programmer wanting to write efficient programs is: "As a general rule, write FORTRAN style SISAL to get the best optimised performance"! The OSC compiler does provide information on copying. However this information is difficult for the programmer to use, in order to improve a program's efficiency.

Furthermore all the above problems are compounded by higher order languages. Analyses become more expensive, more complex, more difficult to report and harder for the programmer to understand.

4 The alternative: prevention of implicit copying

Recently two alternatives to using compile-time analyses to optimise implicit copying in programs have been suggested: linear type systems [8, 13, 16, 18] and linear abstract data types [15]. Both these approaches enforce linear manipulation of arrays; thereby guaranteeing that arrays are never shared. This allows replace operations to *always* be implemented by destructive updates. To copy a value (they cannot be shared) an *explicit* copy operation must be used.

4.1 Linear type systems

Linear type systems support linear types. A value having linear type is enforced, by the type system, not to be shared. Thus if arrays are made linear types then replace

[1]Recently on the net, people have enquired about what storage optimisations the OSC compiler performs and how these can be recognised.

operations on arrays may be implemented by destructive updating. Rather than detecting via a compile-time analysis that arrays are not shared, a linear type system enforces this.

In [16] Wadler presents a type system which supports linear and shared types. There is a natural restriction that although linear types may reference shared types, shared types may not reference linear types. This ensures linear types are not shared indirectly via shared types. Wadler's system has been implemented by Wakeling [18]. Two more sophisticated systems are proposed in [8, 13].

4.2 Monads and linear ADTs

In [15] Wadler has proposed using monads to structure functional programs. Monads can also be used to achieve linearity; that is to support the linear manipulation of arrays. This guarantees that arrays are unshared. These monad operations may be encapsulated into an ADT, which may be implemented using destructive updating. We will call such ADTs linear. The main advantage of this approach is that no extension of a conventional functional language is required; only a special implementation of the ADT is required. Hudak is investigating the general implementation of linear ADTs [10]. The next section describes our own research with a linear ADT.

5 A linear ADT

This section describes our own ideas for a linear abstract data type. The ADT described has been inspired by Wadler's ideas on monads [15, 17]. The linear ADT allows replace operations to be implemented using destructive assignment. It also prevents copying when divide and conquer programs are written which utilise array partitioning and concatenation. In order to discuss these approaches a Haskell like lazy functional language will be used [11, 12].

A special ADT is used to manipulate arrays efficiently. The ADT allows replace operations to be implemented by destructive assignment. This is possible because the ADT prevents arrays from being shared. This ADT does not replace Haskell's arrays rather it provides an efficient alternative way of manipulating them. The ADT is not limited to Haskell, the only requirements are a higher order functional language with at least a Hindley/Milner style type system; laziness is not required.

Essentially the idea is this: by only allowing arrays to be manipulated indirectly, via array transformation functions, the sharing of arrays is prevented. In fact our transformation functions on arrays will be very conservative; they will prevent the copying, sharing and discarding of arrays' structure. However, the contents of arrays may be changed. This is similar to work on semantics where commands are often modelled as store transformers, functions from one store to another store, and of course the store is never copied! We call array transformation functions *transformers*. Rather than dealing just with array trans-

formers we deal with general transformers. Transformers are a limited form of function, which can only be combined (composed) linearly. Their implementation is shown below:

```
> data T a b     = Trans (a -> b)
```

This states that the hidden representation of transformers is a record, named "Trans", with a single function component. Notice how polymorphism is required for the transformer type, T, to be fully general.

The rest of this section describes primitive operations on transformers (T: the linear abstract data type). Note that all operations on the linear ADT preserve the structure of arrays.

Transformers may be linearly combined using #. The # function is reverse function composition. Its implementation is shown below:

```
> infixl 9 #
> (#) :: T a b -> T b c -> T a c
> (#) (T f) (T g)        = T (g . f)
```

This operation is used to build linear sequences of transformations, rather like ";", in an imperative language, is used to build linear sequences of commands. The # function is associative. An expression such as `t1 # t2 # t3` may be read as apply transformer `t1`, then apply transformer `t2` and then apply transformer `t3`.

An identity transformer `tid` exists. It is the identity element of # and its implementation is:

```
> tid :: T a a
> tid    = T (\x -> x)
```

A note on arrays: in this paper vectors will be used; these are an instance of Haskell arrays:

```
> type Vec a     = Array Int a
```

For example a vector of `Int` would have type `Vec Int`. The operations used on vectors are indexing, for example `v!3` yields element 3 of `v`, and bounds enquiry, for example `bounds v` would yield `(1,100)` for vector indexed from 1 to 100.

The replace operation is called `assign`: this is like SISAL's replace, and it is a restricted form of Haskell's `//`. The operation given an index and a value produces a transformation on vectors. The transformation maps one vector to another with the appropriate modification.

```
> assign :: Int -> a -> T (Vec a) (Vec a)
```

How do transformers interface with the non-linear conventional world? Two operations are provided: `tap` for applying transformers to conventional types and `read` for reading 'intermediate' values. Both operations must perform some evaluation and copying to ensure transformers do not introduce observable side-effects.

The `tap` function applies a transformer to a value. However since a transformer may destructively update (modify) a value, the value must be unshared. Otherwise, if the value was shared, other parts of the program might 'see' the value change when transformers were applied to it. To achieve this the value is completely evaluated and copied. Sharing may still be present if the value is a function. Hence function types are disallowed; this is achieved by restricting a to be an equality type, that is a ground type:

```
> tap :: Eq a => T a b -> a -> b
```

A real implementation should provide a special input facility for reading arrays directly from a file etc. without having to copy them. This is discussed in the Further research section.

The `read` operation has the following type:

```
> read :: (a -> b) -> (b -> T a c) -> T a c
```

It allows the reading of a value which is to be subsequently transformed. Typically `read` is used to read an element of an array. For example, the read value may be used by an `assign` operation, or the value may be used used to control a transformer (for example `swap` and `tif` in the next section).

The `read` operation may be described thus:

```
tap (read f g) a  =  tap (g (f a)) a
```

Similar to `tap` the read function `f` of `read f g` has its result fully evaluated and copied, in case it is altered by subsequent transformations. Generally this will be efficient since the result of the read function will be an atom to be assigned by an `assign` transformation. Reading of sub-arrays is inefficient and requires further research.

There is a straightforward transformer, which given two transformers produces a pair transformer. Pair transformers may be applied in parallel.

```
> pp :: T a b -> T c d -> T (a,c) (b,d)
```

The zuz function is like a combined zip and unzip. It takes a transformer, which maps a vector of pairs to a vector of pairs, and it produces an isomorphic transformer, which maps a pair of vectors to a pair of vectors. The resulting transformer only works on pairs of equal length arrays.

```
> zuz :: T (Vec (a,b)) (Vec (c,d)) -> T (Vec a,Vec b) (Vec c, Vec d)
```

Note that efficient implementation of zuz is possible, see [19]. (Essentially this operations corresponds to a regular communication operation in a data parallel language.)

In order to write divide and conquer operations on arrays the following operations will be required: `halve`, `cat` and `isunit`. The `halve` transformer halves an array as equally as possible:

```
> halve :: T (Vec a) (Vec a,Vec a)
```

The `cat` function is used to concatenate arrays:

```
> cat :: T (Vec a) (Vec b,Vec b) -> T (Vec a) (Vec b)
```

Given a halving like transformation, the `cat` function produces a transformation which concatenates the result of the halving transformation, thus: `cat halve` = `tid`.

Note, the `halve` and `cat` operations are good for expressing parallel programs using arrays, because they express simple data partitioning. This is especially useful for distributed implementations which must move data between processors.

The `isunit` function is used for controlling divide and conquer functions on arrays. It produces a transformer which is equal to one of two transformers depending on whether or not the array to which the new transformer is applied, is a singleton.

```
> isunit :: (a -> b) -> T (Vec a) (Vec b) -> T (Vec a) (Vec b)
```

It may be described thus:

$$\begin{array}{lcll} \texttt{tap (isunit f t)}\ \langle \texttt{a} \rangle & = & \langle \texttt{f a} \rangle & \\ \texttt{tap (isunit f t)}\ \langle \texttt{a}_1 \ldots \texttt{a}_n \rangle & = & \texttt{tap t}\ \langle \texttt{a}_1 \ldots \texttt{a}_n \rangle, & \text{if } n > 1 \end{array}$$

where angle brackets denote vectors.

6 Some example programs

This section describes some programs written using the linear ADT described in the previous section. As might be expected from the functional nature of the linear ADT, the programs have an FP [1] flavour. This is because we have elevated our array operations from the object (array) level to the function level.

A transformer to swap two elements of a vector is shown below:

```
> swap :: Int -> Int -> T (Vec a) (Vec a)
> swap i j      = read getvals doswap
>                 where
>                 getvals v             = (v!i,v!j)
>                 doswap (ival,jval)    = assign i jval # assign j ival
```

The swap function produces a transformer which reads two values from a vector, and then swaps the two values around.

An "if" function may be expressed thus:

```
> tif :: (a -> Bool) -> T a b -> T a b -> T a b
> tif c t f     = read c g
>                 where
>                 g True    = t
>                 g False   = f
```

Given a predicate and two transformers this produces a transformer which behaves like one of the two transformers, depending on the value of the predicate applied to the value to be transformed.

A `while` function may be defined using the `tif` transformer.

```
> while :: (a -> Bool) -> T a a -> T a a
> while c t     = tif c (t # (while c t)) tid
```

This takes a predicate and a transformer as argument and produces a transformer. It repeatedly applies the transformer until the predicate does not hold on the transformed value. The `while` operation may be described thus:

$$\texttt{tap (while c t) a} = \texttt{s a}$$

where s is the shortest sequence (t # t # ... # t) such that: `not` (c (s a)).

A sequential for loop, similar to `while`, may be defined:

```
> for :: Int -> Int -> (Int -> T a a) -> T a a
> for l h f     | l > h          = tid
>               | otherwise      = f l # for (l+1) h f
```

The for function takes two integer bounds and a function from an integer to a transformer as arguments. The function is applied over a range of integers formed from the two bounds. The resulting sequence of transformers are sequentially composed using #. The for transformer may be informally described thus:

$$\texttt{tap (for l h f) a} = \texttt{tap (t l \# t (l+1) \# t (l+2) \# ... \# t h) s}$$

A sequential scan (prefix) operation may be defined using for:

```
> sscan :: (a->a->a) -> T (Vec a) (Vec a)
> sscan op = read bounds (\(lo,hi) -> for (lo+1) hi f)
>            where
>            f i        = read p (assign i)
>                         where
>                         p a = (a!(i-1)) `op` (a!i)
```

The sscan function may be described thus:

$$\texttt{tap (sscan } \oplus\texttt{)} \langle a_1 \ldots a_n \rangle = \langle a_1, a_1 \oplus a_2, \ldots, a_1 \oplus a_2 \oplus \ldots \oplus a_n \rangle$$

A parallel divide and conquer map operation is shown below:

```
> tmap :: (a -> b) -> T (Vec a) (Vec b)
> tmap f        = isunit f (cat (halve # pp (tmap f) (tmap f)))
```

This takes a function from a to b and produces a corresponding transformation from vectors of type a to vectors of type b. This may be described thus:

$$\texttt{tap (map f)} \langle a_1 \ldots a_n \rangle = \langle \texttt{f } a_1 \ldots \texttt{f } a_n \rangle$$

A parallel divide and conquer scan operation may be defined using tmap:

```
> pscan :: (a->a->a) -> T (Vec a) (Vec (a,a))
> pscan op      = isunit f (cat g)
>       where
>       f x     = (x,x)
>       g       = halve # pp (pscan op) (pscan op) # zuz (tmap h)
>       h ((a,as),(b,bs))       = ((a,as `op` bs),(as `op` b,as `op` bs))
```

This version of scan may be described thus:

$$\texttt{tap (pscan} \oplus\texttt{)} \langle a_1 \ldots a_n \rangle = \langle (a_1, l), (a_1 \oplus a_2, l), \ldots, (l, l) \rangle \text{ where } l = a_1 \oplus a_2 \oplus \ldots \oplus a_n$$

This operation is only defined on arrays whose length is a power of two: since zuz produces a transformer which only works on equal length arrays[2]. Operationally, pscan is parallel, but less sequentially efficient than sscan. Note that neither the sequential or parallel versions of scan perform any array copying.

One might wonder what the benefits of using functional languages are. After all if we want efficient assignment why not use an imperative language? One of the many useful features of functional languages is the ability to define higher order abstractions. For example we may define a more general divide and conquer abstraction which encompasses tmap and pscan thus:

```
> dc :: T (Vec a) (Vec a,Vec a)          ->      -- divide
>       T (Vec b,Vec b) (Vec b,Vec b)    ->      -- post adjust
>       (a -> b)                         ->      -- solve
>       T (Vec a) (Vec b)
> dc d p s      = isunit s (cat (d # pp (dc d p s) (dc d p s) # p))
```

This divide and conquer combinator allows transformations on vectors to be defined. It takes three arguments: a halving (splitting) function, a post adjust function which adjusts results before they are concatenated and a solve function which is applied to individual array elements.

For example using dc, tmap and pscan may be defined thus:

```
> tmap f        = dc halve tid f
```

```
> pscan op      = dc halve (zuz (tmap h)) f
>       where
>       f x               = (x,x)
>       h ((a,as),(b,bs)) = ((a,as `op` bs),(as `op` b,as `op` bs))
```

7 Further research

Transformers as described are somewhat limited. There are three main problems with the present scheme:

[2]This is a standard restriction for the algorithm given; other algorithms can be expressed which work on unbalanced data.

- Nested data structures cannot be transformed, only top level data structures can be transformed (except pairs of arrays).
- The `read` operation must copy the result of the reading function. This is expensive if the result is large. The `tap` function suffers from a similar problem.
- Apart from pairs multiple values cannot be transformed.

Handling nested structures requires a generalisation of the system. For example we require an `assign` transformer having the following type:

```
> assign :: Ix -> T a a -> T (Vec a) (Vec a)
```

This allows one transformer to be applied to a particular element on an array. There are many issues in this area of nested structures and generalising transformers.

The second problem, that of copying for reading, can be solved along similar lines to Wadler in [15]. Essentially another ADT is defined, that of readers. This prevents read values from being shared and yet allows parallel reading. The problem with this approach is its syntactic clumsiness; monad comprehensions may solve this problem [15].

There is less of a copying problem with `tap`. All that is needed are some functions for *defining* arrays which prevent the defined arrays from being shared. (Note, that this copying problem is to do with defining and not manipulating arrays.) One such array defining function, mentioned previously, might read an array from a file; another function is shown below:

```
> tapvec :: Int -> Int -> (Int -> a) -> T (Vec a) b -> b
```

This function constructs a new vector and applies a transformer to it. The first two parameters specify the bounds of the vector. The third parameter is a function to initialise elements of the vector. The fourth parameter, a transformer, is applied to the vector constructed using the previous parameters. This may be specified thus:

$$\texttt{tapvec l h f t} \;=\; \texttt{tap t} \langle \texttt{f l}, \texttt{f (l+1)}, \ldots, \texttt{f h} \rangle$$

The third problem, transforming multiple values, can be overcome by the introduction of a general operation for defining transformers over all tuples. However, a better and more general solution might be to use *records* of values, but this requires an extension to the type system to handle records ...

It is desirable to have a `halve` operation in Haskell to operate on naked arrays `nhalve :: Vec a -> (Vec a,Vec a)`. This operation is efficient to implement, since

unlike cat it requires no copying. With nhalve operations such as reduce may be defined, which can read, but not modify, arrays; for example a vector of numbers can be summed.

Sometimes it can be useful to copy and discard arrays. A simple way to achieve this, in the present scheme, is with the following operations:

```
> tcopy :: T a (a,a)
> fkill :: T (a,b) b
> skill :: T (a,b) a
```

However, these operations do not always interact well with cat, for example consider: cat tcopy. The resulting vector is larger than the vector to which it is applied. One way to prevent this is to have several ADTs and to use Haskell's type classes for overloading the operations. By analogy with Haskell consider the classes Eq and Num. The Eq class is more general that the Num class. Any operations on Eq are defined on Num but not vice versa. The operations described above tcopy, fkill and skill belong to a less general class than the transformer functions previously described, since they allow discarding and duplication of values.

8 Conclusions

We have argued that preventing implicit array copying is better than trying to cure it. To this end we have defined a linear abstract data type. This supports efficient functional assignment, and efficient divide and conquer operations, on arrays. The latter is particularly useful for writing parallel programs.

It might be useful to incorporate a linear ADT into SISAL so that the operational behaviour, efficiency, of programs is apparent from their text. This can greatly ease programming when efficiency is important. However the ADT relies on higher order functions and a polymorphic type system, hence SISAL 1.2 is not suitable, but SISAL 2.0 [4] may be suitable. The linear ADT is particularly useful because it enables divide and conquer algorithms such as parallel map and scan to be expressed, using powerful low level primitives. Divide and conquer algorithms such as parallel scan cannot be efficiently expressed in SISAL.

Although the ADT described here allows some useful programs to be written, it is nevertheless too restrictive for general programming. Further research is necessary to increase the linear ADT's generality.

Acknowledgements. Thanks to my office-mates Brad Alexander and Dean Engelhardt for many interesting discussions and thanks to Andrew Wendelborn for his support and advice.

References

[1] J Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *CACM*, 21(8):613–641, 1978.

[2] R Bird and P Wadler. *An Introduction to Functional Programming.* Prentice Hall International, 1988.

[3] A Bloss. Update analysis and the efficient implementation of functional aggregates. In *1989 ACM Conference on Functional Programming Languages and Computer Architecture, London.* Addison-Wesley, 1989.

[4] A P W Böhm, D C Cann, J T Feo, and R R Oldehoeft. SISAL reference manual (language version 2.0). Draft report, February 1992.

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

[6] D C Cann. *The Optimising SISAL Compiler: Version 12.0*, 1992. Manual with OSC version 12.0.

[7] K Gopinath and J L Hennessy. Copy elimination in functional languages. In *16th Annual ACM Symposium on Principles of Programming Languages, Austin, Texas*, pages 303–314. ACM Press, January 1989.

[8] J C Guzmán and P Hudak. Single threaded polymorphic lambda calculus. In *Proc. 5th IEEE Symposium on Logic in Computer Science*, June 1990.

[9] P Hudak. A semantic model of reference counting and its abstraction. In S Abramsky and C Hankin, editors, *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[10] P Hudak. Mutable abstract datatypes or how to have your state and munge it. Draft Summary, July 1992.

[11] P Hudak and J H Fasel. A gentle introduction to Haskell. *SIGPLAN Notices*, 27(5):T1–T53, May 1992.

[12] P Hudak, S L Peyton Jones, and P L Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*, 27(5), May 1992.

[13] M Odersky. Observers for linear types. In *European Symposium on Programming*, February 1992.

[14] S K Skedzielewski. SISAL. In B K Szymanski, editor, *Parallel Functional Languages and Compilers.* ACM Press, 1991.

[15] P Wadler. Comprehending monads. In *1990 ACM Conference on Lisp and Functional Programming, Nice*, June 1990.

[16] P Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods, Israel*, 1990.

[17] P L Wadler. The essence of functional programming. In *1992 ACM Symposium on Principles of Programming Languages, Santa Fé*, pages 1–14. ACM Press, January 1992.

[18] D Wakeling and C Runciman. Linearity and laziness. In *1991 ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, MA*, pages 215–240, 1991.

[19] C Walinsky and D Banerjee. A functional programming language compiler for massively parallel computers. In *1990 ACM Conference on Lisp and Functional Programming, Nice*, pages 131–138, June 1990.

Mathematical Syntax for SISAL

Dr. Arun K. Arya*
Dr. Arun Arya and Associates

David Woods and Charles Murphy †
University of Massachusetts · Lowell

October 23, 1992

Abstract

This paper presents a mathematical syntax which can be generated from SISAL 2.0 and the tool to generate this mathematical form automatically. This tool is envisioned to be incorporated into a SISAL 2.0 translator. We contend that a mathematical syntax would be closer to a native language for computer users especially those well founded in mathematics. In addition, this mathematical form is suitable for overheads and textbook with the added benefit that the mathematical form is guaranteed to be syntactically correct. This will enhance the understandability presented programs. The example translations contained here are intended as a starting point for standardizing on mathematicalized SISAL and would be interested in others peoples ideas. There is no reason to prevent the extension of our approach to arbitrary functional languages.

1 User Domain Language

Programming languages have traditionally been designed for use by those who have been specially trained in the engineering of software. This software has been built to solve problem in a wide range of specializations, most of which are outside the engineering of software. These developers of software by necessity also need to be cross trained in these specializations.

*arya@cs.umn.edu
†Department of Computer Science, Lowell, MA 01854, {cmurphy,dwoods}@cs.ulowell.edu

Those people in other specializations that need computer support have found it necessary to develop in the programmer's world. This has diverted efforts from directly supporting their own efforts and has enslaved them to working in areas that they may not feel competent in. Very few of us can work effectively in many disciplines.

There are a large number of disciplines that could use computer languages that are geared to the way that each disciplines finds natural to express the solutions to their problems and may describe the initial solutions in this user domain. A number of these disciplines utilize discrete mathematics in discussing solution for their problems and therefore the use of symbolism from discrete mathematics would be a natural choice. Another example of a domain specific languages might be a dataflow style graphical display which might be good for systems engineers, network engineers or signal processing engineers.

We are currently taking a different approach in that we are starting from the programmers language and generating a representation back to a user's language similar to that of discrete mathematics. This display form of the programming language should be equivalent to a user language that would be developed with appropriate development tools.

We chose this mechanism to build these test languages as the development time is shorter. One could (and eventually will have to) build visual editors and parsers to examine the effectiveness of these user languages but would add several software layers to frontend, for example, the SISAL compiler. Others have input text that mimic the output form to generate visual form. We have chosen to transform the SISAL 2.0 language[1] into a discrete mathematics form that does appear to be closer to what one might expect. There is an amount of programming "noise" (such as type declarations) that still exists in the output form but as the goal is to generate a form in which users could develop in, we have left this in.

2 Bottom-up SISAL Parser

The current parser for SISAL 1.2 is a top-down implementation in C with no existing parser for SISAL 2.0 that we are aware of. We have used L-attributed LALR parser generator tools, which are capable of resolving reduce-reduce conflicts at parse time, to parse SISAL 2.0. These are used to produce a bottom-up parser which generates a mathematical syntax for SISAL programs automatically. We also provide a method to customize the output form to meet individuals preference. We use LaTex as the output form for the parser.

There are two primary language parsing tools which are used. The first is *lcpg*[2]. This

program acts as a preprocessor to convert a parser specification with actions specified in C into a form that a YACC style program could interpret. The lcpg form has several advantages over YACC. The lcpg input language does not use the YACC notion of $ digit symbols to represent terminals and non-terminals values in the action portion of the specification, but replaces these with the symbol name preceded by a $. This greatly increases the readability of the specification and eliminates the errors caused by inserting actions in the right hand side. In addition, lcpg treats inherited attributes as arguments to, and synthesized attributes as results returned by nonterminals that are viewed as procedure calls. The other interesting notion in lcpg is that it allows the compiler developer to specify conflict resolution rules that will allow one to resolve a reduce/reduce conflict at parse time. This obviously requires a different YACC style process which is called CYACC (for Conflict YACC).

These tools allow the development of L-attributed LALR parsers without a lot of grammer rewriting in a clean, easy to comprehend fashion. This speeds up the development process.

3 Mathematical Syntax

There are many reasonable mappings from SISAL 2.0 to a math form. We are currently using a mapping from SISAL to a form that is similar to that of discrete mathematics. Some of the more important mappings are: array indices, reserved words, mathematical operators, reduction symbols, language operator symbols, greek names etc., and general mathematical denotations.

As this is intended to be a strawman mathematical syntax, this should be viewed as a starting point for a proper mathematical syntax for SISAL designed by an appropriate committee. Further, there is no reason to prevent the extension of this approach to other functional languages. We must also recognize that there is not one uniform syntax that all will recognize as the best. It is hoped that a form can be found that is in the users domain that could be executable.

3.1 Identifiers

There is a transliteration of greek letter and special function names into their mathematical form. Therefore identifiers such as alpha, beta, gamma, delta, and epsilon are displayed as

$\alpha\ \beta\ \gamma\ \delta\ \epsilon$. The uppercase letters must be a different source id as SISAL is a case insensitive language. Currently the names biggamma, bigdelta, bigtheta are displayed as $\Gamma\ \Delta\ \Theta$. We also can use this method to give special symbols such as sum and product the familiar forms $\sum\ \prod$.

Other identifiers have display forms that are similar to there SISAL but it is the first occurrence of an identifier that determines the capitalization. In addition, when arrays and streams elements are accessed we use a subscripted notation to indicate which elements. A hold over from SISAL is the ...notation as an array index.

We have chosen to leave function, modules, and programs definition statement very similar to SISAL form. The comma separator has been replaced $\times$ along with returns replaced by $\Rightarrow$. For example

function Reduce (pivot : integer $\times$ A : TwoD $\times$ B : OneD $\Rightarrow$ TwoD $\times$ OneD)

is the visual form for

Function Reduce(pivot:integer, A:TwoD, B:OneD returns TwoD,OneD)

The prototype form of a function is

function MatMult (TwoDim $\times$ TwoDim $\Rightarrow$ TwoDim)

3.2 Types

Types are a remnant from the programming language. To identify types the boldface word **type** is used followed by $\equiv$ and the type definition. The type definition has the basic form

$$\frac{\textbf{component\_name or class}}{\text{type of component or class}}$$

For example:

$$\frac{\textbf{array}\ [\ldots\ ,\ \ldots]}{\text{numeric}}$$

is the class array with 2 dimensions that is composed of the data type numeric.

One can juxtapose several of these together to form a record

$$\frac{\text{a}}{\text{integer}}\ \frac{\text{b}}{\text{real}}\ \frac{\text{c}}{\text{integer}}$$

or forming a union by separating the components with a |.

$$\frac{\text{a}}{\text{integer}}\ |\ \frac{\text{b}}{\text{real}}\ |\ \frac{\text{c}}{\text{integer}}$$

SISAL has the capability of allowing a programmer build up the set of values a type may contain by using the + to indicate set union and − for set difference.

3.3 General Expressions

Immediately after the function definition is the return expression list that is evaluated to and therefore the return values of the function. In order to extract values from inside iteration construct, the iterator returns an expression list that is composed of component from inside the iterator. To visualize this, we have place the expression list that is extracted from the iterator and placed under a left arrow. We would have preferred an arrow the length of the expression but has not been implemented in Tex. An alternative might have been to use ⊣ the the right of the expression list.

Expression lists are organized as one expression on a line separated by a comma. Multiple line expression would not have the comma to separate. There has been a suggestion that we inclose an expression list in a extended parenthesis that we have not yet implemented.

Bindings are indicated by ← and works over expression lists.

Special note on the visual form of array updates and generation of complete arrays. An array update takes on the form of a binding that references on the left hand side the components of the array being updated. On the right hand side is the expression list of values the array is being updated with. The example

$$G_{2\ldots r-1,\ 2\ldots c-1} \leftarrow M$$

would update the values of the corresponding components of the G array from the M array leaving the undesignated components the same value as before.

Similarly array generators treat the type component as the value and subscript with the values being updated (the entire array). On the right hand side is a list of expressions that may preceded by a selection of the output array components that will be generated by the expression list. These right hand side entities are enclose in parenthesis. The example

$$\textbf{array}\ \text{numeric}_{\ i \in 1...size(A,1),\ j \in 1...size(B,2)} \leftarrow \left([\text{i}\ ,\ \text{j}\ ] \sum (\text{A}_{\ i,\ ...}\text{B}_{\ ...,\ j}) \right)$$

would create an array with the size of the first dimension of array A by the size of the second dimension of array B. The value of the array would in the i,j components would be filled by the vector product from the A and B arrays.

Sisal let expressions predefines values for use inside the scope of the let expression. We have chosen to transform the let into the postdefine form using where notation. The let body is place before the declarations separated by the **where** .

SISAL selection constructs are the if and case. Each of these construct make be mapped to the same visual form. The visual selection is indicated by a large left curly brace that delimits horizontally the visual form that is affected. Immediately to the right of the brace is a vertical set of expression lists. Associated with each set on the last line of the set is a condition either starting with an **if** followed by a condition or an *otherwise*.

The visual form of

```
if ( G[i,j] = 1 & Total > 5 ) then 0
   elseif ( Total ~= 3 ) then 1
   else G[i,j]
   end if
```

is

$$\begin{cases} 0, & \textbf{if}\ \text{G}_{\ i,\ j} = 1 \bigwedge \text{Total}\ > 5 \\ 1, & \text{Total}\ \neq 3 \\ \text{G}_{\ i,\ j}, & \textit{otherwise} \end{cases}$$

3.4 Iterators

Iterators have two basic forms in SISAL. There is the *for* loop constructs and the reduction operators. The reduction operators are treated as a special function with a internal mapping produced in the same manner as identifiers. The loop is indicated by the $\forall$. Underneath this symbol is the SISAL *in* clause which is the set notation that indicates the values over which the iterator will range. For example

$$\underset{j \in 2 \ldots c-1}{\overset{j \neq i}{\forall}}$$

Above the $\forall$, we have placed the predicate from the for-test component in SISAL indicating a while loop. A *NOT* preceding the predicate indicates the *until* form. The above expression could be derived from

```
for j in 2..c-1 while j ~= i
do
```

There are two special cases of the *in* clause of *for* expressions. The first is the dot product form. This is indicated by listing all of the *in* expressions under one $\forall$. The other is the cross product form which is indicated by a separate $\forall$ symbol for each *in* expression.

$$\underset{\substack{j \in 2 \ldots c-1 \\ i \in 2 \ldots d-1}}{\forall}$$

could be derived from

```
for j in 2..c-1 dot i in 2..d-1
do
```

and

$$\underset{j \in 2 \ldots c-1}{\forall} \quad \underset{i \in 2 \ldots d-1}{\forall}$$

could be derived from

```
for j in 2..c-1 cross i in 2..d-1
do
```

There may be a declaration component immediately after the for. This is represented by a where clause bound to the scope of the loop. For instance

```
for j in 2..c-1 ; k = 10
do ...
```

would be represented as

$$\underset{j \in 2 \ldots c-1}{\forall} \ldots$$
where
$k \leftarrow 10$

The body of the for loop is a flattened expression list and is placed after the $\forall$ symbols.

4 Examples

4.1 Life

```
program GameOfLifeExample

% John Conway's Game of Life.  Values of the Grid are 0's or 1's.  A
% cell has 8 heighbors.  Each iteration updates cells as follows: --If a
% c ell has a 1 and greater than five of its neighbors have 1's, then it
% should be updated to a 0.  -- If a cell has a 0 and has more than
% three or fewer then three neighbors have 1's, then it should be
% updated to a 1.  -- otherwise the value of a cell remains unchaned.
% The simulation iterates n times.  The border of the grid never
% changes.  The lower bound of each dimension is assumed to be 1
```

```
type Grid = array[..,..] of integer;

function Compute( G:Grid, i,j:integer returns integer )
  let
    Total := G[i+1,j-1] + G[i-1,j] + G[i+1,j+1] + G[i-1,j-1] +
             G[i-1,j] + G[i-1,j+1] + G[i,j-1] + G[i,j+1];
  in
    if ( G[i,j] = 1 & Total > 5 ) then 0
    elseif ( Total ~= 3 ) then 1
    else G[i,j]
    end if
  end let
end function

function DoWork ( G:Grid, r,c:integer returns Grid )
  let
    M := for k in [2..r-1] cross j in [2..c-1] do
      returns array[2..r-1,2..c-1] of Compute(G,k,j)
      end for;
  in
    G[2..r-1,2..c-1: M ]

  end let
end function

function Life( n:integer, G:Grid returns Grid )
  let
    r := size(G,1);
    c := size(G,2);
  in
    for i in [1..n] do
      G := doWork(G,r,c);
    returns G
    end for
  end let
end function

end program
```

program GameOfLifeExample

type Grid $\equiv \underset{\text{integer}}{\textbf{array}\ [\ldots\ ,\ \ldots]}$

function Compute (G : Grid $\times$ i , j : integer $\Rightarrow$ integer)

$$\begin{cases} 0, & \textbf{if } G_{i,j} = 1 \bigwedge \text{Total} > 5 \\ 1, & \text{Total} \neq 3 \\ G_{i,j}, & \textit{otherwise} \end{cases}$$

where

Total $\leftarrow G_{i+1,j-1} + G_{i-1,j} + G_{i+1,j+1} + G_{i-1,j-1} + G_{i-1,j} + G_{i-1,j+1} + G_{i,j-1} + G_{i,j+}$

function DoWork (G : Grid $\times$ r , c : integer $\Rightarrow$ Grid)

$G_{2\ldots r-1,\ 2\ldots c-1} \leftarrow M$

where

$M \leftarrow \textbf{array}_{2\ldots r-1,\ 2\ldots c-1}\{\overleftarrow{\text{Compute}}\ (G\ ,\ k\ ,\ j\ )\} \underset{k\in 2\ldots r-1}{\forall}\ \underset{j\in 2\ldots c-1}{\forall}$

function Life (n : integer $\times$ G : Grid $\Rightarrow$ Grid)

$\{\overleftarrow{G}\}\ \underset{i\in 1\ldots n}{\forall}\ G \leftarrow \text{doWork}\ (G\ ,\ r\ ,\ c\ )$

where

r $\leftarrow$ size (G , 1)

c $\leftarrow$ size (G , 2)

4.2 Matrix Multiply

```
% This program illustrates the defiition and use of a matrix package
% comprising a matrix multiplication operation.

interface matrixRoutines
  type TwoDim = array [..,..] of numeric;
  function MatMult( TwoDim, TwoDim returns TwoDim );
end interface

module MatrixRoutines

  function Matmult( a, B : TwoDim returns TwoDim )
    if size(A,2) ~= size(B,1) then error[TwoDim]
    else array numeric [i in 1..size(A,1), j in 1..size(B,2):
          [i,j] sum(A[i,..] * B[..,j])]
    end if
  end function
end module

program MatrixMultiplyExample

  from MatrixRoutines: MatMult;

  type TDR = array[..,..] of real;

  function MatMult( A, B: TDR returns TDR )
     MatrixRoutines.MatMult( A, B )
  end function

end program
```

interface matrixRoutines

type TwoDim $\equiv \underset{\text{numeric}}{\textbf{array}\ [\ldots\ ,\ \ldots]}$

function MatMult (TwoDim $\times$ TwoDim $\Rightarrow$ TwoDim)

module MatrixRoutines

function MatMult (a , B : TwoDim $\Rightarrow$ TwoDim)

$$\begin{cases} \textbf{error}\,[\text{TwoDim}\,], & \textbf{if size}\ (\text{A}\ ,\ 2) \neq \text{size}\ (\text{B}\ ,\ 1) \\ \textbf{array}\ \text{numeric}_{\,i \in 1 \ldots size(A,\,1),\ j \in 1 \ldots size(B,\,2)}\,([\text{i}\ ,\ \text{j}\,] \textstyle\sum (\text{A}_{\,i,\,\ldots}\text{B}_{\,\ldots,\,j})), & \textit{otherwise} \end{cases}$$

program MatrixMultiplyExample

from $\underset{\text{MatMult}}{\text{MatrixRoutines}}$

type TDR $\equiv \underset{\text{real}}{\textbf{array}\ [\ldots\ ,\ \ldots]}$

function MatMult (A , B : TDR $\Rightarrow$ TDR)

MatrixRoutines.MatMult (A , B)

4.3 Gaussian Elimination

```
program GaussExample

type OneI = array [..] of integer;
type OneD = array [..] of double;
type TwoD = array [..] of OneD;

function Reduce(pivot:integer, A:TwoD, B:OneD returns TwoD,OneD)
     let
          mults := A[..,pivot] / a[pivot,pivot];
     in
          for row in A at [i] do
               nrow,
               nB := if i = pivot then
                         row / A[pivot,pivot],
                         b[i] / A[pivot,pivot]
                    else
                         row - mults[i]*A[pivot],
                         B[i] - mults[i]*B[pivot]
                    end if
          returns array of nrow,
               array of nb
          end for
     end let
end function

function Main ( n:integer,A:TwoD,B:OneD returns OneD)
     for i in [1..n] do
          A,B := Reduce(i,A,B);
     returns B
     end for
end function
end program
```

program GaussExample

type OneI $\equiv \frac{\textbf{array}\ [\ldots]}{\text{integer}}$

type OneD $\equiv \frac{\textbf{array}\ [\ldots]}{\text{double}}$

type TwoD $\equiv \frac{\textbf{array}\ [\ldots]}{\text{OneD}}$

function Reduce (pivot : integer $\times$ A : TwoD $\times$ B : OneD $\Rightarrow$ TwoD $\times$ OneD)

$$\begin{array}{l}\textbf{array}\ \overleftarrow{\{\text{nrow}\}},\\ \textbf{array}\ \{\text{nb}\}\end{array} \underset{row \in Ai}{\forall} \text{nrow , nB} \leftarrow \begin{cases} \frac{\text{row}}{A_{\,pivot,\,pivot}} & \\ \frac{b_{\,i}}{A_{\,pivot,\,pivot}}, & \textbf{if i} = \text{pivot} \\ & \\ \text{row} - \text{mults}\ {}_{i}A_{\,pivot} & \\ B_{\,i} - \text{mults}\ {}_{i}B_{\,pivot}, & \textit{otherwise} \end{cases}$$

where

$$\text{mults} \leftarrow \frac{A_{\,\ldots,\,pivot}}{a_{\,pivot,\,pivot}}$$

function Main (n : integer $\times$ A : TwoD $\times$ B : OneD $\Rightarrow$ OneD)

$$\overleftarrow{\{B\}} \underset{i \in 1 \ldots n}{\forall} A\ ,\ B \leftarrow \text{Reduce (i , A , B)}$$

4.4 Quicksort

```
program QuickSortExample
type Info = array of integer;

function QuickSort(Data: Info returns Info)
     if (size(Data)<2) then Data
     else let Pivot := Data[liml(Data)];
          Low, Mid, High := for E in Data do
               returns
                    array of E when E < Pivot,
                    array of E when E = Pivot,
                    array of E when E > Pivot
               end for
     in
          QuickSort(Low) || Mid || QuickSort(High)
          end let
     end if
end function
end program
```

program QuickSortExample

type Info $\equiv \underset{\text{integer}}{\textbf{array}}$

function QuickSort (Data : Info $\Rightarrow$ Info)

$$\begin{cases} \text{Data}, & \textbf{if size (Data)} < 2 \\ \text{QuickSort (Low)} \,||\, \text{Mid} \,||\, \text{QuickSort (High)} \\ \quad \textbf{where} \\ \quad \text{Pivot} \leftarrow \text{Data}_{\,liml(Data)} \\ \quad \text{Low, Mid, High} \leftarrow \begin{array}{l} \overleftarrow{\textbf{array}}\,\{\text{E} : \text{E} < \text{Pivot}\}, \\ \textbf{array}\,\{\text{E} : \text{E} = \text{Pivot}\}, \\ \textbf{array}\,\{\text{E} : \text{E} > \text{Pivot}\} \end{array} \forall_{E \in Data}, & \textit{otherwise} \end{cases}$$

5 Conclusions

In this paper we have briefly described a mathematicalized form for SISAL 2.0 and the tools used to generate this form automatically. We feel that this form in it's final version could allow a wider range of backgrounds to understand and implement programs in the spirt of SISAL development.

This methodology is also capable of other tasks. One example that we have done in the past is the visualization of intermediate forms such as IF1[3]. This visualized form gives a user an immediate feel for the information contained in a tree or a graph and we are also working on a dataflow representation of the graphs from the IF1 form.

References

[1] A.P. Böhm and D.C. Cann and R.R. Oldehoeft and J.T. Feo. *SISAL Reference Manual Language Version 2.0.* Livermore, CA.

[2] Arun K. Arya. Lcpg: An l-attribute c parser generator. Technical Report R-88-002, University of Lowell, 1988.

[3] Stephen Skedzielewski et al. IF1: An intermediate form for applicative languages. Technical report, Lawrence Livermore National Laboratory, Livermore, CA, 1985.

An Approach for Optimizing Recursive Functions

Steven M. Fitzgerald and Linda Wilkens

Department of Computer Science
University of Massachusetts Lowell
Lowell, MA 01854
{sfitzger,lwilkens}@cs.ulowell.edu

Abstract

Within the SISAL programming environment, existing optimization techniques do not address recursive functions. This limitation restricts the performance of many applications. One approach is to transform recursive solutions into their iterative counterparts, thus allowing existing techniques to optimize the resulting form. In this paper, we discuss the issues involved in converting a recursive function into an equivalent iterative form. An outline of an algorithm which performs a source-to-source transformation on tail recursive SISAL functions is presented. We then discuss the issues involved in generalizing this approach.

1 Introduction

Much of the expressive power of functional programming comes from the use of recursive decomposition. Recurrences play a central role in numerical computation [Gao90]. To address execution efficiency, optimization techniques, such as build-in-place [Ran87] and update-in-place [Can89], have been developed. As a precursor to these optimization techniques, functions are typically in-lined. Since the calling depth of recursive functions is not known until execution time, these functions cannot be in-lined. Without additional support for recursion, the performance that can be achieved for numerically intensive applications is limited since the current optimization techniques do not consider recursion.

In the current test suites, recursion is avoided because its use leads to poor performance. Although a recursive program can always be translated into an equivalent iterative one [BB86], requiring the user to perform the translation may introduce errors and confusion. If a problem lends itself naturally to a recursive decomposition, the corresponding recursive program is clearer and easier to prove than the equivalent iterative program [BB86]. Requiring the

user to express the algorithm in a contrary manner is unproductive even when it yields better performance.

Within the SISAL programming environment, recursion is currently handled at run-time, resulting in high overhead costs. A stack is used both to maintain information about the calling sequence and to store temporary results. To address the lack of optimizations for recursive functions, several approaches are possible. One approach is to convert a recursive function automatically into an equivalent non-recursive form [Boi92, Bir80, FLS87, Böh88]. When possible, this conversion should take place as early as possible in the compilation process to take advantage of other existing optimizations. The state introduced by removing recursion, however, is not readily expressible in an applicative programming language. An alternative approach is to revise existing optimization techniques to consider recursion.

In this paper, we examine the approach of converting a recursive function into an equivalent iterative one. We first discuss some of the issues involved in this process. We then present an outline of an algorithm to remove tail recursion. Although this translation process is straightforward, this exercise provides a base for developing transformation techniques for a larger class of recursive programs.

2 The Task

In this section, we examine some of the issues involved in the automatic conversion of a recursive function into an equivalent iterative form within an applicative environment. For clarity, SISAL 1.2 source code is used in our discussion, although we intend the conversion process to be performed in the context of an intermediate language, for example IF1 [SG85] or IF2 [WSYR86]. Using an intermediate language allows both state information to be introduced and existing optimizations to be performed, while keeping the conversion process language-independent. First, some definitions are provided.

> **Definition**: *Simple Recursion* exists when all recursive calls to a function appear in the definition of that function.
>
> **Definition**: *Tail Recursion* occurs when each recursive calls in function is the last operation performed by a particular function invocation.
>
> **Definition**: A *Principal Call* is the initial call to a recursive function.
>
> **Definition**: A *Dependent Call* is a function call within the definintion of a recursive function. principal call.
>
> **Definition**: A *Continuation Path* is an execution path that induces another recursive call.

Definition: A *Termination Path* is an execution path that allows the recursive function to terminate without inducing another recursive call.

Execution of recursion requires that state information be maintained. This information, which resides on a run-time stack, allows the context of a recursive call to be restored upon return. This is necessary since many activations of the same function may be in different phases of execution. To convert recursive functions into their iterative forms, state information must be considered. There are forms of recursion that rely less heavily on state information than others. We concentrate on these forms of recursion because we want to limit the introduction of state information. Additionally, emphasis on these recursive functions helps to identify the issues involved in the conversion of other forms of recursion. Specifically, tail recursion is examined in order to establish a base from which further issues can be explored.

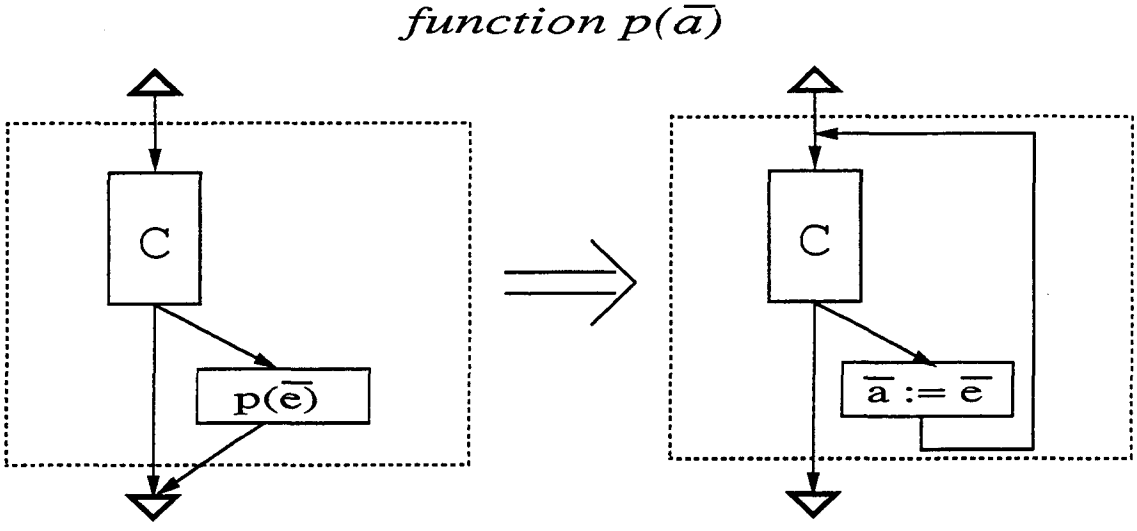


Figure 1: Transformation of Tail Recursion into a Loop

In tail recursion, the call sequence does not have to be maintained. The recursive function can be directly transformed into a loop structure, eliminating function-call overhead. Figure 1 depicts this conversion. Notice the values of the expressions, $\bar{e}$,[1] passed to the recursive call must be evaluated and assigned to loop variables, $\bar{a}$. These values are then used in flow-graph C during succeeding

[1] denoting a set of expression, say $e_1 \ldots e_n$,

executions of the loop. In the transformation process, the continuation path is redirected to the top of the function, while the termination path remains unchanged.

```
function fac (N :integer returns integer)

  if N >= 2 then
      N * fac (N - 1)
  else
      1
  end if
end function % fac
```

Figure 2: SISAL 1.2 Recursive Definition of Factorial

```
function fac (N :integer returns integer)

  for initial
      accum := 1;
      NewN  := N;
  while (NewN >= 2) repeat
      accum := old NewN * old accum;
      NewN  := old NewN - 1;
  returns value of accum
  end for
end function % fac
```

Figure 3: SISAL 1.2 Iterative Definition of Factorial

To maintain the semantics of the function, other modifications must be made to the function. This can be seen by examining the differences in the recursive and iterative forms of a function computing factorials. These functions are presented in Figures 2 and 3, respectively. Notice there are two execution paths in the recursive form. The continuation path is repeatedly traversed until the condition statement causes a dependent call to function "fac" to terminate. All preceding function invocations are then terminated as each dependent call returns its value.

In the iterative form, a local variable, *accum*, is introduced to carry the result of the partial computation from one iteration of the loop to another.[2] This is

[2]This variable is initialized to the *identity* value for the operation used in the computation.

necessitated by the change in the method of computation. In the recursive version, a series of expressions are partially evaluated, and are then completed upon return of the dependent call; thus "fac(1)" is computed first. The return value of the dependent call is then used to determine the next largest factorial. The values needed to complete the evaluations are normally stored on a run-time stack, which is not needed in the iterative form.

It is straightforward to transform the factorial function by hand, but the transformation method cannot be readily applied to other forms of recursion. This example has identified an approach for addressing state information without the use of a run-time stack. An outline of an algorithm to convert tail recursive functions into equivalent iterative ones is presented in the next section. An example of converting quicksort using the algorithm is also provided. We then discuss some of the issues involved in eliminating general recursion.

3 Outline of the Transformation Algorithm

Below we provide an outline of an algorithm to perform a source-to-source transformation on SISAL programs, removing tail recursion. The current algorithm is limited since it only considers functions that have a single continuation path and a single termination path. Additionally, the function operating on the return value is limited to a single binary operator. This operator must have an identity value and this value must be provided. A detailed example using the algorithm is presented in Section 4. Notice, meta-symbols are delimited by "<" and ">".

TRANSFORMING TAIL RECURSIVE SISAL FUNCTIONS
INTO EQUIVALENT ITERATIVE ONES

Given: A function definition "p"$(f_1, \ldots, f_n)$ which is tail recursive with only one termination path and only one continuation path. The *identity* value of the operator combining the results of the recursive call, e.g., <UNIT> for <OP>. Without loss of generality the function has the following template.

```
function p(f1, ..., fn)
  if (<COND>) then
    ...
    p(f1, ..., fn) <OP> <EXPR>
    % or <EXPR> <OP>  p(f1, ..., fn)
  else
    <TERMEXPR>
  end if
end % function
```

1. Create the new function using the template:

```
if (<COND>) then
    for initial
        <INIT>
      until ~(<COND>)
      repeat
          ...
          <BODY>
      returns <RETURN>
    end for
else
  <TERMEXPR>
end if
```

2. Initialize the loop variables, replacing <INIT>
   (a) for each formal parameter of the function "p", create a loop variable, and initialize it to its corresponding formal parameter, e.g., $tf_1 := f_1$.
   (b) create a loop variable, and initialize it to the unit value of the <OP>, i.e., "accum" := <UNIT>. The variable is used to accumulate the results of the loop.
3. Copy the boolean expression controlling access to the continuation path into both <COND>s. In the <COND> associated with the "until" statement, substitute all formal parameters with their corresponding loop variables, created in step 2a.
4. Substitute each reference to a formal parameter with a reference to its corresponding loop variable, i.e., <EXPR>$[f_n \backslash tf_n]$.
5. Replace <BODY> with the following statements:
   (a) "accum" := <EXPR> <OP> "old accum"
   % "accum" := "old accum" <OP> <EXPR>
   Note the position of <EXPR> *w.r.t.* <OP>
   (b) $tf_1 := a_1; \ldots tf_n := a_n$; (Evaluating the parameters for the recursive call.)
6. Replace <RETURN> with "accum" <OP> <TERMEXPR>
   % or <TERMEXPR> <OP> "accum"
   Note the position of <EXPR> *w.r.t.* <OP>

4 An Example

In this section, we describe the conversion process by transforming the recursive function quicksort. This function, depicted in Figure 4, has been extracted from the SISAL 1.2 reference manual [MSA+85, pg D-2]. For brevity, we have omitted the definition of the function Split. There are two recursive calls in this function, only one of which is transformed by our algorithm. The binary operator "||" (array concatenation) is associative, however the SISAL compiler parses it as a left associative operator. Therefore, the recursive call "quickSort(R)" is associated with the last expression executed and it is removed via the transformation algorithm.

```
function quickSort(Data :Info returns Info)

    if array_size(Data) >= 2 then
      let
        L, Middle, R := Split(Data)
      in
        quickSort(L) || Middle || quickSort(R)
      end let
    else
      Data
    end if

end function % quicksort
```

Figure 4: Recursive Definition of Quicksort

The first step in the algorithm is to create a template for the new function. This new template differs from the original template only in the structure of the continuation path; a for-loop has been inserted. From the original function, we can syntactically identify the following code fragments which are used in both functions.

- <COND> ≡ array_size(Data) >= 2
- <EXPR> ≡ quickSort(L) || Middle
- <OP> ≡ ||
- <TERMEXPR> ≡ Data

Inserting these values into the new function's template yields

```
function quickSort(Data :Info returns Info)

  if (array_size(Data) >= 2) then
       for initial
           <INIT>
        until ~(<COND>)
        repeat
           ...
           <BODY>
        returns <RETURN>
      end for
  else
      Data
  end if
end % function
```

The loop that has been inserted within the continuation path simulates the recursive behavior of the original function. Four parts of the loop must be defined, the <INIT>, <COND>, <BODY> and <RETURN>. Additionally, the original body of the continuation path needs to be inserted into the body of the loop.

In the initialization section, two loop variables are introduced. The first variable, *tData*, is used to mimic passing the parameter which corresponds to the formal argument *Data*, and is initialized to its value. As the loop executes, *tData* is updated to reflect the value of the next parameter to the recursive call being simulated. The second variable, *accum* is used both to store and to combine the partial results as they are evaluated. The initial value of *accum* must not affect the first partial result. Since the operator <OP> combining the results is array concatenation, the empty array is used to initialize *accum*. In general, the accumulation variable is initialized to the identity value for <OP>. The identity value must be know *a priori* and supplied to the algorithm.

Entry to the continuation path is controlled by the <COND> expression, "array_size(Data) >=2". Similarly, execution of the loop continues as long as this condition holds. We must, however, replace *Data* with the loop variable *tData*. In a recursive function which contains several continuation paths, the disjunction of all the conditions may be used as the test criteria, i.e., until (<COND1>| ... |<CONDN>).

In the <BODY> section, the loop variables must be re-initialized for the next iteration. This operation is performed after the non-recursive expressions are evaluated. Thus, the statement "L, Middle, R := Spilt(tData)" is inserted

into the loop body first.[3] Next, the effect of evaluating the recursive expression is simulated by combining the results produced by the previous iteration of the loop with the remaining portion of the recursive expression, <EXPR>. This result is then stored into *accum*. Finally, the loop variable, *tData* which mimics the formal parameter of the recursive function is re-initialized, i.e., "tData := R".

Because the concatenation operator is not commutative, operand placement is significant when combining the results. In the new function, old *accum* is the left operand of the operator ||. However in the original function, the recursive call "quickSort(R)" is the right operand. As the loop is executed, the left portion of the subarray *tData*, which is now sorted, must be positioned both to the right of past results and to left of future results. At the top of each loop, *accum* contains the left portion of the final sorted array while the new partial results are generated by the expression "`(quickSort(L) || Middle)`". These arrays are combined and stored into the new accumulated array which is used in the next iteration. In this manner, as the right portion of each subarray is sorted, the result is appended to the previously sorted subarrays.

Immediately prior to loop termination, *accum* contains the entire sorted array except for the final value of *tData*. This value is then appended to the end of *accum* in the *returns* statement of the for loop. Additionally, the keywords "value of" have been inserted to make the final statement syntactically correct. The final form of the transformed function is depicted in Figure 5.

5 General Recursion

To establish a base for research, a class of recursive functions which do not require a run-time stack has been examined. For the transformation approach to be useful, other forms of recursion need to be considered. Although all recursive functions can be effectively computed via a loop construct, a major concern is maintaining state information across loop boundaries. The state information can be maintained by using a run-time stack, but this approach tends to result in poor performance.

A recursive call can occur in the middle of a function. Consider Figure 6 which depicts a graph of a simple recursive function and how it might be transformed. Subgraphs C and D represent the operations that are performed before and after each recursive call, respectively. The function is restructured to allow C to be executed repeatedly, via a loop. A similar change is made to D. In this situation, the context of the recursive call must be maintained. The number of times which the subgraphs C and D are executed must be the same. Additionally, the values which flow directly from subgraph C to D must be preserved. These problems do not occur in tail recursion since the values used

[3] The *let* construct has been removed for syntactic reason.

```
function quickSort(Data :Info returns Info)

  if array_size(Data) >= 2 then
    for initial
        tData := Data;
        accum := ARRAY Info [];
    until ~(array_size(tData) >= 2)
    repeat
        L, Middle, R := Split(old tData);
        accum := old accum || (quickSort(L) || Middle) ;
        tData := R;
    returns value of accum || tData
    end for
  else
    Data
  end if
end function
```

Figure 5: Transformed Quicksort

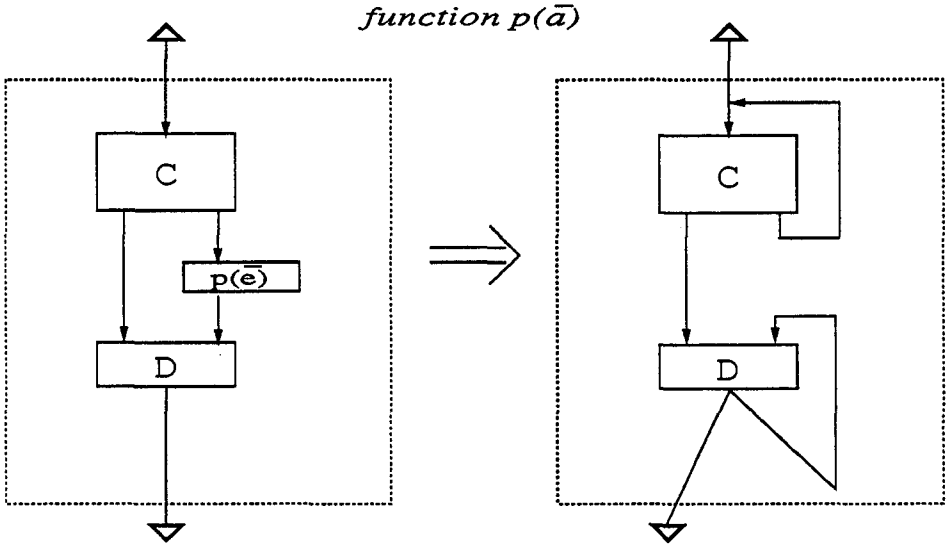


Figure 6: Graph of General Single Recursion

in conjunction with the recursive call are used immediately.

To maintain proper sequencing of subgraphs, counters could be used. Although using counters introduces state variables, it has the advantage of reducing the memory requirements associated with a run-time stack. The counter would be incremented each time the program loops through subgraph *C*. A test would be performed at the end of the function to determine if subgraph execution is balanced. This is depicted in Figure 7. Each time this test fails, the counter would be decremented and subgraph *D* would be executed.

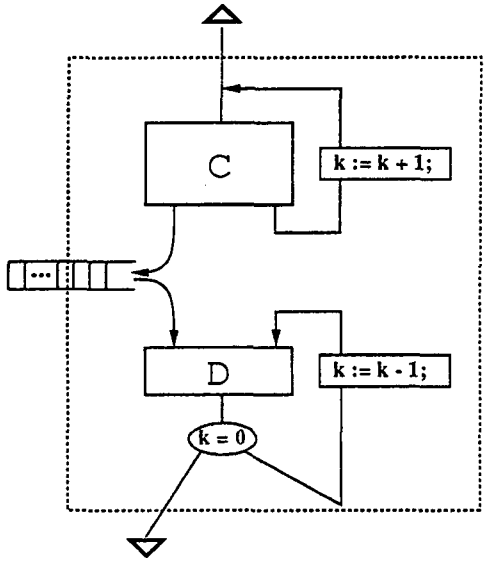


Figure 7: Graph of Counter Controlling Execution

The data values that flow directly to subgraph *D* from *C* must be saved as succeeding values are produced by subgraph *C*. Additionally, as subgraph *D* executes, the data values are required in the opposite order in which they are produced. A stack could be used both to store these values and to reverse their order.

There are two approaches to addressing the additional state introduced by the stack. One approach is to manage a local stack, similar in the way in which the run-time system handles its stack. Although the number of values stored may be reduced, this approach does not allow other optimizations to be performed. In an alternate approach, a flexible array could be introduced to simulate a stack. The push and pop operations could be implemented by *array_addh* and *array_remh*, respectively.[4] The optimizations developed for

[4] Array_addh and array_remh are two of SISAL's primitive operations for arrays.

arrays, for example update-in-place [Can89], could then remove some of the inefficiencies in this approach.[5]

6 Preliminary Indications

To gather evidence for the effectiveness of the transformation algorithm, a small program suite of hand transformed code was timed on a sequential architecture, the Vax 6520, running Ultrix. Although we are concerned about the performance of these algorithms on parallel architectures, establishing results on a sequential architecture allows us to analyze our results without the additional variables introduced by parallel execution. The runs were made at off hours and were repeated 200 times to reduce the effects of system fluctuations. These timings are only preliminary but serve as a general indication of the benefits which might be achieved using this approach.

The comparison of the first two recursive programs, Factorial and Fibonacci, are presented in Figure 8 and 9, respectively. Due to their short execution times, each function was executed 100000 times, and the total time was divide by 100000. As one would expect, the iterative versions of the functions perform substantially better than their recursive counterparts. The non-linear growth of the iterative form of the Fibonacci function is due to the second recursive call, which is not be transformed by our existing algorithm.

To test the transformation approach on applications which use arrays, timings were produced for quick sort and merge sort. A range of array sizes from 256 to 8192 elements were used to produce a set of timings. Unfortunately, the results obtained vary greatly, and we are in the process of analyzing the data.

7 Work in Progress / Future Work

To establish the merits of this approach, we will gather empirical results. Initially, we will perform timings on hand modified source code. Since conversion introduces a sequential loop, our analysis will also weigh the potential loss of parallelism against the reduction in function call overhead. In algorithms with multiple recursive calls, e.g. quicksort, a function call can be used to initiate work on a different processor. Removal of both recursive calls could affect the amount of parallelism that is detected in the program. We believe, however, the cost of function call overhead is substantial enough to warrant conversion, in most cases.

SISAL source code was used for clarity in our discussion of the conversion process. Future research will be performed using IF1 and IF2 as the source language. This will allow us to concentrate on fewer semantic constructs as well

[5]Update-in-place may be able to identify that the array has a single owner and allow updates to be done in place.

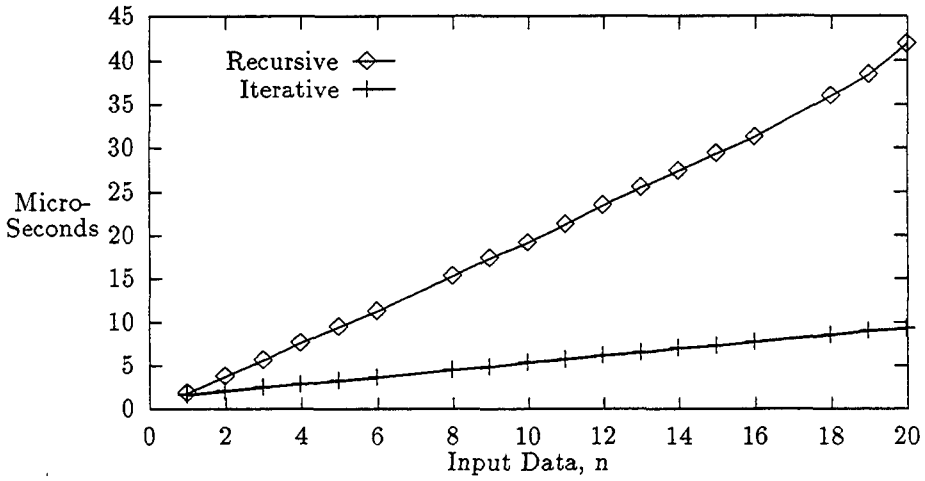


Figure 8: Timing for Factorial Function, fac(n)

as to detach the process from a particular source language. Additionally, this will give us the flexibility to introduce state information without compromising the benefits of other optimization techniques.[6]

Research is underway to extend the applicability of the conversion approach. In the cases we have examined so far, recursion has been limited to simple nodes. When recursion is part of a compound node, we expect different issues to be raised, dictating special handling. For example, removal of the remaining recursive call in the body of the quicksort function (Figure 5) may require special handling since it appears in the body subgraph of a loop.

According to our definition of tail recursion, the factorial program in Figure 2 is not tail recursive. Notice that the value N is multiplied with the return value of each recursive call. The multiplication is part of subgraph D since it is executed after the recursive call (refer to Figure 10a). We have been able to fold this computation into the resulting loop. Figure 10b, however, illustrates a more complex graph which our present algorithm cannot handle.

In our research, we will also develop a taxonomy of recursive functions. The classification will be based both on the methods needed to transform a recursive function and on the characteristics of the resulting iterative version.

[6]That is, optimization techniques, such as update-in-place, will not have to consider the possible effects of the state that is introduced the conversion process.

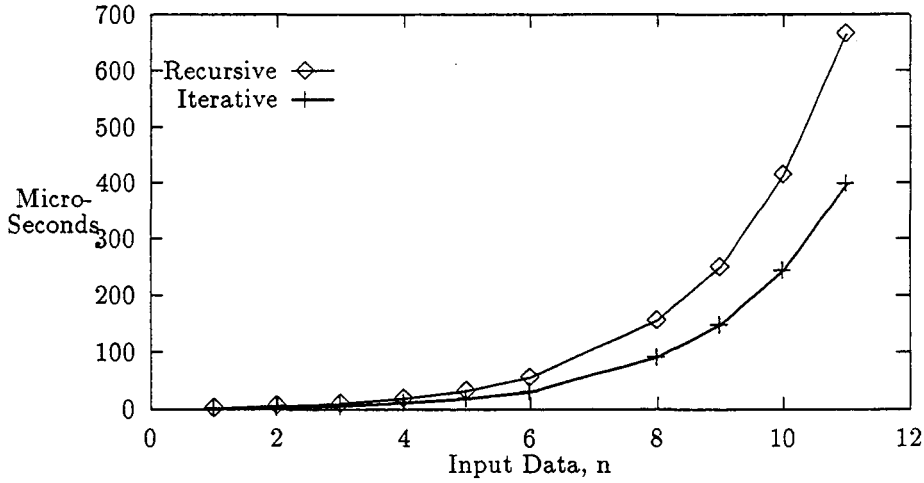


Figure 9: Timing for the Fibonacci Function, fib(n)

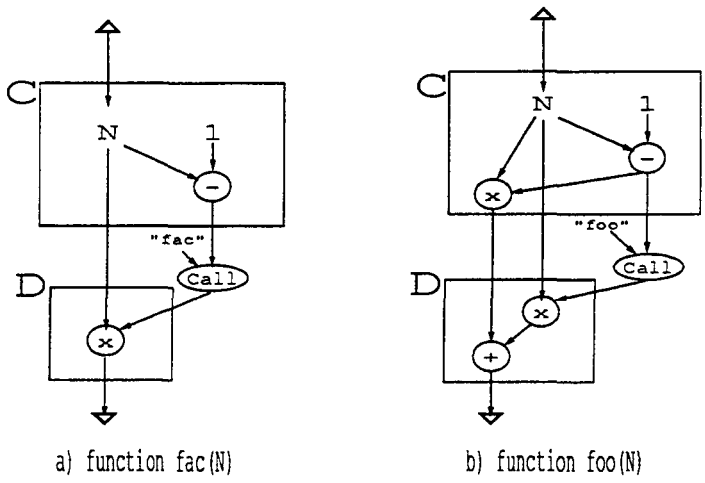


Figure 10: Partial definition of recursive functions, "fac" and "foo"

8 Summary

Optimization techniques have been developed for applicative languages. These optimizations have dealt primarily with the execution efficiency of array constructs. Recursion is an important language feature that has not been addressed in these optimizations. To enhance the existing optimizations, several approaches are possible. An approach that appears to have merit is to convert a recursive function into a non-recursive form. In this paper, we have identified several issues involved in this transformation process.

We have presented an outline of an algorithm to remove tail recursion from SISAL programs. Research is in progress both to generalize this method and to migrate the process to the IF1 and IF2 level. Although removing recursion involves the introduction of state variables, this approach has the advantage of both reducing the memory requirements associated with a run-time stack and eliminating function call overhead. If this approach is successful, it relaxes the need for existing optimizations, such as build-in-place, to consider recursion. Additionally, it may expose other opportunities for optimizations which will both enhance the performance of existing SISAL programs and encourage the development of new SISAL applications.

References

[BB86] Pierre Berlioux and Philippe Bizard. *Algorithms: The Construction, Proof, and Analysis of Programs.* John Wiley & Sons, 1986.

[Bir80] R. S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, December 1980.

[Böh88] Corrado Böhm. Reducing recursion to iteration by means of pairs and n-tuples. *Foundation of Logic and Functional Programming*, Lecture Notes in Computer Science:58–66, 1988.

[Boi92] Eerke A. Boiten. Improving recursive functions by inverting the order of evaluation. *Science of Computer Programming*, 18:139–179, 1992.

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

[FLS87] Kay-Ulrich Felgentreu, Wolfram-M. Lippe, and Friedmann H. Simon. Optimizing static scope Lisp by repetitive interpretation of recursive function calls. *IEEE Transactions on Software Engineering*, 13(6):628–635, June 1987.

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

[MSA+85] James McGraw, Stepen 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, manual m-146 edition, March 1985.

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

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

[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, manual m-195 edition, November 1986.

Implementing Arrays in SISAL 2.0

R. R. Oldehoeft
Computer Science Department
Colorado State University

1 Introduction

In this report we outline an implementation technique for arrays in SISAL 2.0 [2] as well as in other parallel programming languages. To preserve the efficiency of array access required for parallel systems with either shared memory or distributed memory, we rely on organizational opportunities first observed while working through the design of SISAL 2.0.

We had three goals in the redesign of array facilities for SISAL. First, the currently implemented version of SISAL, 1.2, has some redundancy among array operations that we wanted to simplify without losing expressive power or convenience. Second, experience showed us that we need true multi-dimensional arrays (as well as arrays of arrays). Third, other languages with arrays define operations analogous to those on scalars that work element-by-element on array operands, so we defined this expressive power for SISAL arrays.

The distribution of array objects among the processor-memory pairs of distributed-memory parallel systems is a subject of significant current study. The development of explicit distribution techniques embedded in programming languages is exemplified by PCF Fortran [7]. Fortran D [5] uses programmer-defined declarations as well as compiler optimizations to improve efficiency. Automatic data alignment is the goal of several projects [4, 6, 8].

Another major group of research projects aims to preserve the illusion of a single shared memory on distributed-memory multiprocessors. These distributed shared-memory systems may be tied to programming languages [3] or may be operating system based [1, 9, 10]. Different systems implement versions of coherence, but most keep track of fixed-size pages containing shared data.

Our goal is to provide an efficient implementation that is similar for both shared-memory and distributed-memory parallel systems. The design uses data structures that work well in machines with shared memory supported by interconnection hardware (not necessarily with uniform access time), and that can be made to maximize the effectiveness of distributed shared memory supported by system software.

Before discussing implementations, we need a few concepts from the revised language.

array-ref	::=	primary [selector]
selector	::=	selector-part [, selector-part] ···
		[{ diag-spec [, diag-spec] ··· }]
selector-part	::=	expression | [value-id **in**] triplet
triplet	::=	[expression] .. [expression] [.. expression]
diag-spec	::=	value-id [**dot** value-id] ···

In the above, brackets [ ] are lexical elements, and [ ] enclose syntactically optional forms. A trailing ellipsis, ···, indicates that the preceding may appear zero or more times.

In SISAL 2.0, a *selector* is a comma-separated sequence of items. Each has one of the following two forms:

- An *integer-valued expression.* This form only identifies some portion of a larger array structure; ordinary subscripting is a simple example.
- A *triplet.* The first expression gives a lower bound, the second expression defines an upper bound, and the third is the stride between subscript values for consecutive elements. This form is useful for several purposes in different contexts. It may describe the extent of an array in the dimension associated with this position, it may select a subarray of an extant array, or it may define which elements to update within an extant array.

The optional triplet label "value-id **in**" is used for selection of diagonal subarrays. As in other languages that support subarray selection, the comma separator in array selector syntax specifies a cross product of the index sets and gives subarray selection that is rectangular. In addition, one would like to select *diagonal* subarrays. By identifying triplets that will proceed through their extents together, we specify a diagonal dimension in the array. Syntactically, one names ranges by preceding them with "value-name **in**", and specifies which value-names will vary together in what we will refer to as a *group.* The **dot** notation, exemplified below, connects the names of triplets in a group. Subscript ranges thus identified must have the same extents. Names on triplets take on the values in the ranges. The dimensions of the subarray are ordered, left to right, by the first appearance of a name (in a group with other names) on a triplet. For example, consider the following definitions, assuming that `A[1..4, 1..4, 1..10]` is a three-dimensional array:

```
B := A[i in 1..4,     j in 1..4,     1          {i dot j}];
C := A[x in 4..1..-1, y in 4..1..-1, 5          {x dot y}];
D := A[t in 3..1..-1, 4,             u in 1..3 {t dot u}];
```

Then these arrays are defined by the following elements of `A`:

```
B  = A[1,1,1], A[2,2,1], A[3,3,1], A[4,4,1]
C  = A[4,4,5], A[3,3,5], A[2,2,5], A[1,1,5]
D  = A[3,4,1], A[2,4,2], A[1,4,3]
```

All three are one-dimensional, and

```
E := A[ i in 1..4, j in 4..1..-1, 1..10 {i dot j} ]
```

specifies a [1..4,1..10] array of 40 elements composed from codiagonal "lines" in each "plane." Here the subarray reference has two groups. Group 1 consists of the two triplets labeled with i and j (defining the first dimension of the result), and group 2 has only the unlabeled triplet (defining the second dimension) residing by itself. As usual, each of these selected subarrays has unit lower bound and unit stride in each dimension. Note that diagonal subarray definition is not limited to simple 45° diagonals. By using different strides in the associated triplets, other orientations are possible (as long as extents match).

2 Shared Memory

The shared-memory array implementation in this section is in some ways conventional. But the subarray and transpose capabilities will be important in the next section where they contribute to preserving both "array-ness" and partitionability for distributed shared memory. We also outline an implementation of the general diagonal subarray technique described earlier.

2.1 Physical Data Space

The storage for arrays contains a rectangular layout of values. We can organize rectangular storage in either row-major or column-major fashion; a language manual need never specify how it is done. A column-major order simplifies FORTRAN interfacing, but since this implementation makes array transpose trivial, either will work with FORTRAN 90, assuming that language also uses a "dope vector" approach like the one described next.

2.2 Dope Vectors

An array value is a pointer to a "dope vector," a term apparently first used in [11]. Our dope vector has these fields.

- *A reference count.* This allows multiple uses of an array value to share a dope vector.
- *Physical space pointer.* We need this to find the physical space for deallocation, not for subscripting.
- *Physical reference count pointer.* All dope vectors that share all or part of the physical array point to the same physical reference count. The physical reference count is kept separate from the physical data space to make "build-in-place" operations possible.

- *Logical base.* This is a pointer to the place in storage where the array element subscripted by [0, ..., 0] resides. Often this element is nonexistent, but "LB" makes subscripting more efficient.

- *Number of dimensions.* We use n for this value.

- *Packets.* Define n packets, one for each dimension of the array. A packet for dimension j contains three values:

  1. L_j, the lower bound.
  2. U_j, the upper bound.
  3. M_j, the *multiplier*. Let S_k be the extent of the array in dimension k, with S_0 the size of an array element. Then the multipliers for column-major storage are

$$\begin{aligned} M_1 &= S_0 \\ M_j &= M_{j-1}S_{j-1},\ j = 2,\ldots,n \end{aligned}$$

  For row-major array storage, the only change is in the multiplier computation:

$$\begin{aligned} M_n &= S_0 \\ M_j &= M_{j+1}S_{j+1},\ j = n-1,\ldots,1 \end{aligned}$$

  In either case, each multiplier is the address distance from the array's "origin element" to the adjacent element in each dimension.

The logical base LB is easy to compute:

$$LB = (\text{address}([L_1,\ldots,L_n]) - \sum_{k=1}^{n} L_k \times M_k$$

A subscript reference to the element $[i_1, i_2, \ldots, i_n]$ translates to the computation

$$LB + \sum_{k=1}^{n} i_k \times M_k$$

For example, if the the three-dimensional array `A` used in the diagonal subarray example above is stored in memory at address 1 in row-major order, and each element takes 4 units of addressable storage, then its dope vector will be

LB	n	L_1	U_1	M_1	L_2	U_2	M_2	L_3	U_3	M_3
-203	3	1	4	160	1	4	40	1	10	4

and element `[2,3,4]` is found at address $-203 + 2 \times 160 + 3 \times 40 + 4 \times 4 = 253$. Note that the reference count and pointer fields were omitted for simplicity.

2.3 Some Operations

In addition to single element subscripting, we describe the implementation of two other array operations in terms of manipulations on dope vectors. These operations work as we wish them to because, in each dimension, the distance between adjacent elements is a constant.

2.3.1 Transpose

Simply copying a dope vector with packets in reverse order (and keeping the same LB) implements an n-dimensional transpose operation. Because manipulation of dope vectors is not possible in the source language, the transpose operation should be an intrinsic function. One must increment the physical reference count since both the original array and its transpose are sharing all the elements.

To see why this works, element $[i_1, \ldots, i_n]$ is found at address

$$LB + i_1 \times M_1 + \cdots + i_n \times M_n$$

In a transpose, this same element is indexed via $[i_n, \ldots, i_1]$ and its address is the same:

$$LB + i_n \times M_n + \cdots + i_1 \times M_1$$

For example, the transpose of A above is described by the dope vector

LB	n	L_1	U_1	M_1	L_2	U_2	M_2	L_3	U_3	M_3
-203	3	1	10	4	1	4	40	1	4	160

and element [4,3,2] is at address $-203 + 4 \times 4 + 3 \times 40 + 2 \times 160 = 253$.

2.3.2 Subarray Selection

Subarray selection is a powerful notation for expressing algorithms on arrays. Surprisingly, both rectangular and diagonal selection can be unified in a single algorithm that produces first-class arrays as results.

Rectangular Selection When some subscript elements are triplets, they specify subarrays instead of single elements. The result is a new dope vector derived from the original one which shares the physical array space (one must increment the physical space reference

count). The dope vector for completely rectangular selection is computed this way:

$$
\begin{aligned}
n' &= \text{number of triplets} \\
L'_j &= 1,\ j = 1, \ldots, n' \\
U'_j &= (u_j - l_j) \div s_j + 1,\ j = 1, \ldots, n' \\
M'_j &= \text{address}([l_1, \ldots, l_j + s_j, \ldots, l_n]) - \text{address}([l_1, \ldots, l_j, \ldots, l_n]),\ j = 1, \ldots, n' \\
&= M_j \times s_j,\ j = 1, \ldots, n' \\
LB' &= \text{address}([l_1, \ldots, l_n]) - \sum_{j=1}^{n'} M'_j
\end{aligned}
$$

Each multiplier is the address distance between the subarray's "origin element" and the adjacent element in each dimension.

As an example, the subarray `A[3..1..-1, 2, 1..9..2]` of our three-dimensional `A` whose dope vector was given above is described by the two-dimensional dope vector

LB	n	L_1	U_1	M_1	L_2	U_2	M_2
513	2	1	3	-160	1	5	8

Diagonal Selection First, we explain how to derive a dope vector for a subarray specified by triplets that are all in the same group:

```
A[i in l1..u1..s1, ...  , j in ln..un..sn {i dot ...  j}]
```

for an array `A` with n dimensions. The result is one-dimensional, a line in n-space. All extents $(u_j - l_j) \div s_j + 1$ must be the same, and the subscripts $[l_1, \ldots, l_n]$ and $[u_1, \ldots, u_n]$ must be legal. The derived dope vector contains

$$
\begin{aligned}
n' &= 1 \\
L'_1 &= 1 \\
U'_1 &= (u_1 - l_1) \div s_1 + 1 \\
M'_1 &= address([l_1 + s_1, \ldots, l_n + s_n]) - address([l_1, \ldots, l_n]) \\
&= \sum_{j=1}^{n} s_j \times M_j \\
LB' &= \text{address}([l_1, \ldots, l_n]) - M'_1
\end{aligned}
$$

The subarray `A[i in 1..3, j in 4..2..-1, k in 5..7 {i dot j dot k}]` is one-dimensional and characterized by the dope vector

LB	n	L_1	U_1	M_1
13	1	1	3	124

Subarray Selection in General We can unify rectangular and completely diagonal subarray selection in a general way. A subarray reference contains a mix of simple expressions and named and unnamed triplets. Some of the named triplets may be **dot**ted to show that their ranges are to vary together in a *group*. Each unnamed triplet and each named but un**dot**ted triplet resides in a separate singleton group. Triplets $t_{g_1}, t_{g_2}, \ldots, t_{g_n}$ reside in group g. The groups together must define dimension directions from a single array element, the "origin element" of the subarray.

Let G be the number of resulting groups, ordered by the left-to-right appearance of the first triplet in each. The dope vector for the subarray is defined this way:

$$
\begin{aligned}
n' &= G \\
L'_g &= 1,\ g = 1, \ldots, G \\
U'_g &= (u_{t_{g_1}} - l_{t_{g_1}}) \div s_{t_{g_1}} + 1,\ g = 1, \ldots, G \\
M'_g &= \text{address(origin element with 1 step applied from group } g) - \\
&\quad \text{address(origin element)},\ g = 1, \ldots, G \\
LB' &= \text{address(origin element)} - \sum_{g=1}^{G} M'_g
\end{aligned}
$$

To exemplify general subarray selection, recall that `E`, defined in an earlier example, is `A[i in 1..4, j in 4..1..-1, 1..10 {i dot j}]`. Here $G = 2$, with group 1 containing the first two triplets **dot**ted together, and the third triplet is in group 2. The origin element is at `[1,4,1]`. Then `E`'s dope vector is computed:

$$
\begin{aligned}
n' &= 2 \\
L'_1 &= 1 \\
L'_2 &= 1 \\
U'_1 &= 4 \\
U'_2 &= 10 \\
M'_1 &= \text{address}([2,3,1]) - \text{address}([1,4,1]) = 120 \\
M'_2 &= \text{address}([1,4,2]) - \text{address}([1,4,1]) = 4 \\
LB' &= \text{address}([1,4,1]) - (120 + 4) = -3
\end{aligned}
$$

Both rectangular and diagonal subarray definition results in a dope vector that is usable in an implementation in the same way as any other array dope vector. No special cases arise: A subarray may be subscripted, further subarray'ed, updated, passed as an actual function parameter, used in arithmetic operations, and so forth.

3 Distributed Memory

In multicomputer systems with separate physical memories, issues arise in the placement of array segments for efficient execution. We assume a *shared virtual memory* implementation wherein a single virtual address space is shared by all processors, as discussed above. Pages may reside in one or more of the physical memories, depending on the system design.

Two issues need to be addressed to exploit a shared virtual memory efficiently. First, in the distribution of array pieces (rows, columns, or blocks, for example) across processor-memory pairs, *false sharing* must be avoided. This can occur if we naively place, in a single virtual page, array elements from more than one piece. Then writes to the pieces by different processors are updates of the same virtual page, and many messages are required to keep the virtual page involved coherent. We avoid this by expanding the individual pieces to virtual page boundaries with "phantom" elements, denoted in diagrams by □.

Second, we want to continue the shared-memory illusion afforded by the system and described in the previous section for actual shared memory. That is, we will continue to use dope vectors of conventional content to access arrays. We will also use dope vectors to define the array parts that are distributed to different processors in such a way that they will efficiently access those virtual pages containing just the needed subarrays. By continuing the shared-memory illusion at the higher abstraction level of array data structures, we avoid multiple special cases that a compiler and run time system would have to handle.

In addition to the notation for describing arrays in actual shared memory, we will use two more symbols. N is the number of addressable units in a virtual page. P is the number of processors that build and access the distributed array parts—hence P is also the number of array parts and is known before the array is built at run time.

One can use a variety of forms for array parts to be distributed. Among these are sets of contiguous columns, sets of contiguous rows, or subrows or columns (parts of a one-dimensional array, or two-dimensional blocks of a two-dimensional array, for example). See [4, 6] for discussions of how a compiler might come to a decision about an efficient decomposition of an array into parts. Here we explore the efficient implementation of some of these decisions.

We now consider a general two-dimensional subarray distribution, and its generalization and simplifications. We assume a row-major array storage layout, but the development for column-major arrays is just as easy. The development also assumes that arrays are one-origin, which can be generalized without difficulty.

3.1 Two-Dimensional Block Distribution

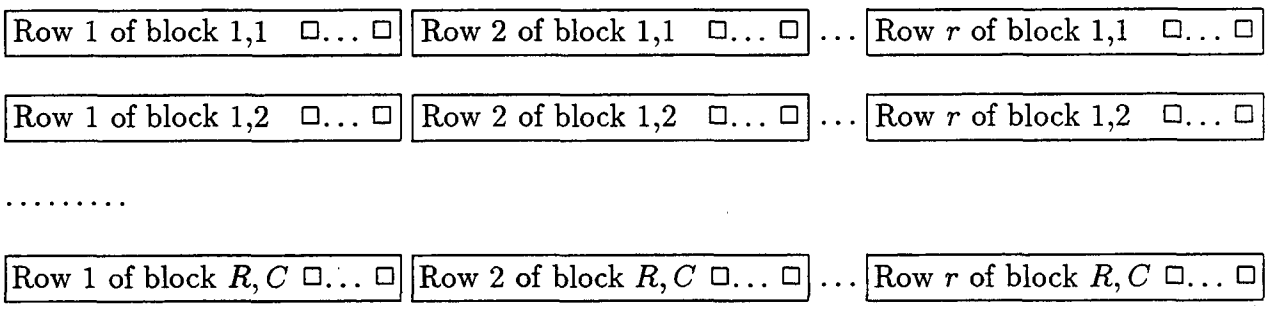
Consider an array `A` with type `array[1..u1, 1..u2] of T`. We will divide `A` into R rows and C columns of rectangular blocks, each with r subrows and c subcolumns (except perhaps the last row and column of blocks).

$$R = (u_1 - 1) \div r + 1,\ C = (u_2 - 1) \div c + 1$$

The number of processors is $P = R \times C$. Each element of type `T` takes S_0 space; we assume that S_0 divides N. In each block, we pad each subrow with phantom elements to use an integral number of pages, so each subrow starts at the beginning of a virtual page. Each subrow takes $(c \times S_0 - 1) \div N + 1$ pages. Let c_A be the number of columns in a padded subrow:

$$c_A = (N \div S_0) \times ((c \times S_0 - 1) \div N + 1)$$

Instead of A, consider B of type `array[1..R, 1..C] of array[1..r, 1..cA] of T`. This can in turn be considered to have type `array[1..R, 1..C, 1..r, 1..cA] of T`. In virtual storage, the layout for our block-decomposed array is:



The following dope vector describes the storage:

LB	n	L_1	U_1	M_1	L_2	U_2	M_2	L_3	U_3	M_3	L_4	U_4	M_4
	4	1	R	$M_2 \times C$	1	C	$M_3 \times r$	1	r	$M_4 \times c_A$	1	c	S_0

Note that M_3 is computed using c_A, not c.

Then a source reference to `A[i,j]` is translated by the compiler to

```
B[(i-1)/r+1, (j-1)/c+1, (i-1)mod(r)+1, (j-1)mod(c)+1]
```

and normal four-dimensional subscripting successfully access all elements.

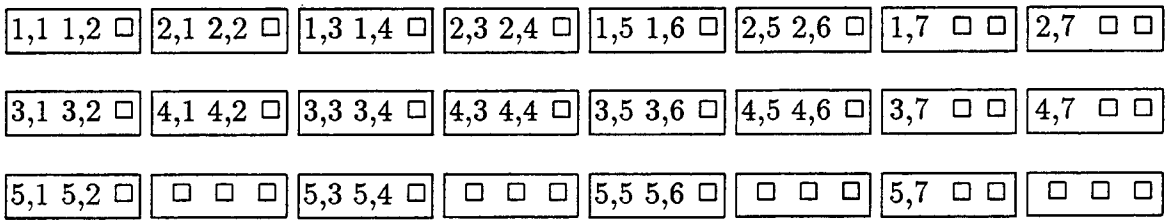
Since $P = R \times C$ processors in parallel will build and further reference each block, we need to build P two-dimensional dope vectors so each will reference exactly the block residing in unique virtual pages. This is accomplished by the subarray operations:

```
B[1, 1, .., ..]  B[1, 2, .., ..]  ...  B[1, C, .., ..]
B[2, 1, .., ..]  B[2, 2, .., ..]  ...  B[2, C, .., ..]
                    ...   ...   ...
B[R, 1, .., ..]  B[R, 2, .., ..]  ...  B[R, C, .., ..]
```

As a concrete example, consider A of type `array[1..5, 1..7]` to be built in parallel on $3 \times 4 = 12$ processors. The distribution by blocks 2×2 in size is:

1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,1	4,2	4,3	4,4	4,5	4,6	4,7
5,1	5,2	5,3	5,4	5,5	5,6	5,7

For simplicity $S_0 = 1$ in all examples. Suppose $N = 3$. The desired virtual memory layout (with $c_A = 3$) is:



The dope vector we need to access the entire space as an ordinary array is (in all examples, we will assume the first element is at virtual address 1):

LB	n	L_1	U_1	M_1	L_2	U_2	M_2	L_3	U_3	M_3	L_4	U_4	M_4
-33	4	1	3	24	1	4	6	1	2	3	1	2	1

Dope vectors for each of the 12 processors to define and manipulate the blocks are formed by conventional subarray operations. For example, `B[2, 3,..,..]` is the `[2,3]` block and is described by the dope vector:

LB	n	L_1	U_1	M_1	L_2	U_2	M_2
+33	2	1	2	3	1	2	1

Each processor uses one-origin subscripting to access its block.

3.2 Generalizations

The reader will be able to see that a block distribution of higher-dimensional arrays into higher-dimensional blocks is easily done. However, in [4] the argument is made that to effectively exploit parallelism in large-scale distributed-memory multiprocessors, one needs only to think of one- and two-dimensional arrays of processors, and distribute the other dimensions within blocks to them.

3.3 Row Distribution

Some algorithms will work better if a contiguous group of entire rows is given to each processor instead of a block of subrows and subcolumns. Think of this as a special case in which $c = u_2 \Rightarrow C = 1$. This alone eliminates the dimension $1..C$, but one can simplify further. Here one only needs to add columns to the right side of the array A so each row begins on a virtual page boundary. The array we actually store is `B[1..u1, 1..cA]` and is distributed to processors in subarrays

$$\texttt{B[(j-1)*r+1 .. j*r, ..]}, j = 1, \ldots, P$$

Returning to our example, using $P = 3$ processors with $r = 2$ whole rows per block, the array is distributed as follows:

1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,1	4,2	4,3	4,4	4,5	4,6	4,7
5,1	5,2	5,3	5,4	5,5	5,6	5,7

If $N = 3$, the desired virtual storage layout is:

1,1 1,2 1,3	1,4 1,5 1,6	1,7 □ □	2,1 2,2 2,3	2,4 2.5 2,6	2,7 □ □
3,1 3,2 3,3	3,4 3,5 3,6	3,7 □ □	4,1 4,2 4,3	4,4 4,5 4,6	4,7 □ □
5,1 5,2 5,3	5,4 5,5 5,6	5,7 □ □			

The dope vector for the entire array is:

LB	n	L_1	U_1	M_1	L_2	U_2	M_2
-9	2	1	5	9	1	7	1

and the dope vector for the middle subarray is (for example):

LB	n	L_1	U_1	M_1	L_2	U_2	M_2
+9	2	1	2	9	1	7	1

3.4 Column Distribution

It is sometimes appropriate to distribute contiguous column sets. This is also a special case of rectangular block distribution with $r = u_1 \Rightarrow R = 1$. As in the preceding, we can store a two-dimensional array and distribute two-dimensional subarrays. Here we pad columns with elements so each begins on a virtual page boundary. However, since we will continue to think about arrays as row-major, we require a new wrinkle.

The desired virtual memory storage layout is column-major, so groups of columns can be distributed. The entire array is still accessible in conventional (row-major) ways by building the overall dope vector differently. The actual layout is conventionally addressable by applying a transpose operation to a column-major dope vector describing the augmented array. Subarray dope vectors are constructed for access to distributed column groups via normal subarray operations to this "unusual" overall dope vector.

We continue the same example: `A[1..5, 1..7]`, $P = 3 \Rightarrow c = 3$. The distribution we want is:

1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,1	4,2	4,3	4,4	4,5	4,6	4,7
5,1	5,2	5,3	5,4	5,5	5,6	5,7

Let $N = 3$, so r_a = actual rows = 6. The desired virtual memory layout is:

1,1 2,1 3,1	4,1 5,1 □	1,2 2,2 3,2	4,2 5,2 □	1,3 2,3 3,3	4,3 5,3 □
1,4 2,4 3,4	4,4 5,4 □	1,5 2,5 3,5	4,5 5,5 □	1,6 2,6 3,6	4,6 5,6 □
1,7 2,7 3,7	4,7 5,7 □				

The transpose of a column-major dope vector addresses this space:

LB	n	L_1	U_1	M_1	L_2	U_2	M_2
-6	2	1	7	1	1	5	6

The subarray dope vector for the middle column set is:

LB	n	L_1	U_1	M_1	L_2	U_2	M_2
6	2	1	5	1	1	2	6

3.5 One-Dimensional Array Distribution

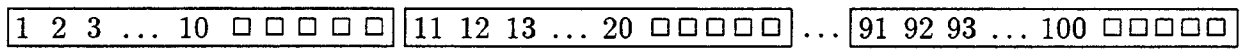
We can think of this as block distribution of a two-dimensional array with only one row, so r, R, and r_A do not exist. Beginning with an array A of type `array[1..u2] of T`, we want a one-dimensional array `B` of one-dimensional arrays: `array[1..C] of array[1..c] of T`.

A two-dimensional dope vector describes the entire array:

LB	n	L_1	U_1	M_1	L_2	U_2	M_2
	2	1	C	$c_A \times S_0$	1	c	S_0

A reference to `A[j]` must be translated by the compiler to `B[(j-1)/c+1, (j-1)mod(c)+1]`.

As an example, consider `A[1..100]`, with $P = 10$ and $N = 15$. The desired virtual storage layout is:



A reference to `A[j]` becomes `B[(j-1)/10+1, (j-1)mod(10)+1]` using the dope vector:

LB	n	L_1	U_1	M_1	L_2	U_2	M_2
-15	2	1	10	15	1	10	1

The dope vector for the j^{th} processor is:

LB	n	L_1	U_1	M_1
$-15 + (j-1) \times 15$	1	1	10	1

4 Conclusions

This report describes implementation methods for multidimensional arrays in SISAL 2.0 that can be useful for other languages as well. The methods are similar for both shared-memory and distributed-memory parallel computer systems to simplify compiler and run-time software. The distributed-memory version uses SISAL 2.0 techniques for subarray definition to align distributable subarrays in shared virtual memory to eliminate false sharing while preserving the ability to address the array as a whole in standard ways.

References

[1] J. Bennett, J. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory. Technical Report COMP-TR89-99, Rice University, November 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] D. Gelernter. Generative communication in Linda. *ACM Trans. Programming Languages and Systems*, 7(1):80–112, January 1985.

[4] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Trans. Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[5] S. Hiranandani, K. Kennedy, and C-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *CACM*, 35(8):66–80, August 1992.

[6] K. Knobe, J. Lukas, and G. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[7] B. Leasure. PCF Fortran: Language definition, version 3.1. Technical report, The Parallel Computing Forum, Champaign, IL, August 1990.

[8] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *J. Parallel and Distributed Computing*, 13:213–221, 1991.

[9] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proc. 5th ACM Symp. on Distributed Computing*, pages 229–239, August 1986.

[10] U. Ramachandran, M. Ahamad, and M. 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.

[11] Kirk Sattley. Allocation of storage for arrays in ALGOL 60. *Communications of the ACM*, 4(1):60–65, January 1961.

FOL : An object oriented extension to the SISAL language (Extended Abstract)

Marc Pantel, Marcel Gandriau, Patrick Sallé
IRIT/ENSEEIHT,
2, rue Charles Camichel,
31071 TOULOUSE,
FRANCE,
{pantel,gandri,salle}@irit.fr

October 2, 1992

Abstract

This paper describes an improvement of the reusability model of the SISAL language. The study of the object oriented paradigm limits leads us to propose several extensions promoting a higher reusability scheme. The paper contains the description of the FOL language and of its translator to SISAL as well as insights in our future work on impure effects.

Introduction

The SISAL language is part of a project whose goal is to demonstrate that functional programs executed on classic "Von Neumann" architectures can be as efficient as any imperative languages (see [BCFO91]). This will enhance the development of this expressive and reliable programming paradigm, well-adapted to the realization of logical algorithmic proofs. The first part of this paper develops the reasons why functional programming and reusability matter. The second part describes the usefulness of an advanced data description model. The third part is a concise description of the FOL language and of its translator to SISAL . The last part concludes and presents the research subjects which will be developed around the FOL language.

1 Improving efficiency in program development

One of the most sought goal of computer science is the improvement of the efficiency in program development. The use of functional languages and the development of reusability are two of the possible answers to this problem.

1.1 Usefulness of pure functional languages

The first computation model which has been intensively used was the Von Neumann's engine. Although this model was close to the way computer worked, its high degree of complexity made quite impossible the study of the logical properties of algorithm written in this model. Other much simpler mathematical models, such as the typed λ-calculus could be used to work out logical proofs of algorithm, but the translation from Von Neumann's imperative style was not trivial. The best solution was to use languages which could be easily translated in one of these models (see [Bac78]). The functional languages, and in particular SISAL , are based on the typed λ-calculus.

1.1.1 Provable algorithm

The simplicity of the λ-calculus mathematical model enables to develop several kinds of proofs about the correction and confluence of algorithms. This goal which has driven many research projects in computer science is part of the quest for higher efficiency in software development. Proofs could be developed with any mathematical model, but only simple ones will permit the development of efficient enough systems.

1.1.2 Quasi mathematical expressiveness

The syntax of functional languages mimics the definition of mathematical functions. This propriety eases the development of programs by mathematicians and engineers, who do not have to learn quite esoteric language syntax to express their equational problems.

1.1.3 Reliability of program

One of the major problems when using software written by someone else is that you do not know exactly what modification the code does to its environment, the so called "side effects". A software library cannot be declared safe and so cannot be used in any case in such a situation. Pure functional languages forbid access to memory cells, a function can only use the value of its parameter and

return a new value, it cannot modify its caller's environment. Thus, the use of functional languages guarantees that a function in a library can be safely used.

These problems motivated the definition of the SISAL language which proved that reliable functional languages could be as efficient as imperative ones without resorting to ad hoc non-"Von Neumann" architecture (see [Can]).

1.2 Usefulness of reusability

Programmers spend most of their time rewriting already existing software components. Either they know nothing of the existence of these softwares, or their requirements are slightly different and impose complete rewriting of the software.

In the last decade, these problems have been investigated by the software engineering community in two different ways. The first approach consisted in improving the description of existing softwares, so that it would be easier to look for old sources than writing new ones. The second approach, which is our field of study, tries to elaborate new paradigms of programming, so that slight changes in software requirements do not involve rewriting of all software components, but only a mere addition of new sources corresponding to the differences.

Research in knowledge modeling proposed to improve the data models as a means to enhance reusability.

2 Data model description

The evolution of programming languages has first stressed the formalization of behavior before studying data description. The next step in programming languages evolution has been the adoption of knowledge modeling mechanism to describe data types. The concept of class has become the basis of the object oriented model which is currently one of the leading trend in the development of new programming languages.

2.1 Classic class based model

Research in knowledge modeling proposed the association of data and behaviors as a stronger semantic model of programs. The inheritance and genericity mechanism enabled to reduce the size of the description of complex systems by improving reusability. However, this goal still encounters some problems.

2.1.1 Data + Behavior = Class

In the object oriented paradigm, the description of programs is based on ideas issued from knowledge modeling. An abstract object is represented by its decomposition in sub-objects and by the description of its behaviors. Such a representation is called a class. Several mechanisms, both syntactic and semantic, simplify the description of complex systems by classes. All along this paper, the word behavior will be used to describe a function part of a class. Its choice was motivated by our will to show that the same model could be used with different evaluation paradigms.

2.1.2 Composition, Genericity and Inheritance as Reusability schemes

The oldest and more efficient reuse concept is the composition. It consists in describing a class as a sum of its different parts and its behavior as a composition of their behaviors. This scheme is very useful for programs which only change the top of their structural decomposition; but if the changes concern only the bottom of the decomposition, a lot of components have to be modified or reprocessed by the compilers.

A new kind of reuse scheme was thus needed, which would enable users to describe new parts of a software system as slight different versions of old parts and to integrate them with a minimum of work. The first step in this direction was the conception of generic mechanism. Some bottom parts of a software component decomposition were specified only by their interface with the system. Every other part having the same interface could be substituted for these.

This scheme was very interesting for the development of storage types such as list, tree or stack, but users had to fully describe every behavior of the interface, even if only one changed from other parts. The next step was the development of the inheritance mechanism which enabled users to describe a new component as an extension of an old one. Only the description of the changed behavior and the new ones were needed. The data type of these systems could now be partially ordered. Each class having no ancestor is the root of an inheritance tree. The dynamic binding mechanism resulting from the combination of inheritance, polymorphism and overloading solves the problem of reprocessing the top part of a system when changes are made in the bottom parts.

To describe the fact that the class $\mathcal{B}$ is an heir of class $\mathcal{A}$, the following notation is used: $\mathcal{A} \searrow \mathcal{B}$.

2.1.3 Limits of the model

The improvements introduced by these approaches were noticeable in several cases, particularly in the implementation of abstract data types and user interface management systems (see [VL88]). In general, studies have shown an

improvement in efficiency when reusability has been stressed as one of the design principles (see [LHKS92]). A paradox can be found here, object oriented languages, designed to improve reusability, burden users with the necessity to predict every use of a software component before implementing it.

Another major problem with the standard class model is the difficulty of unifying several class hierarchies. For example, abstract syntax trees are very useful for formal manipulation of mathematic formulae. Each node of these trees can have several sons depending on the type of the node and of the number of the parameters, that the function or operator it represents, have. If you have efficient class implementations of unary, binary, ternary and n-ary tree coming from different sources, you would like to merge them to solve this problem. This merging is impossible in the standard object oriented system if you cannot modify the code of both classes.

2.2 The key to a higher reusability

To improve the easiness of reusability scheme, the object oriented approach has been enhanced with two new paradigms in our project.

2.2.1 Implicit Genericity

Several object oriented languages do not include a genericity mechanism. This one can easily be emulated by using polymorphism and inheritance, but this approach lacks of the strong typing properties of explicit genericity. Genericity can be a source of problems in the case of software reuse. You have to imagine every possible use of a software component in order to describe as generic every part of the components which could be of any use as a generic part.

In order to avoid this problem, we decided to include in our language an implicit genericity mechanism. Every field of a class is considered to be generic. This permits the definition of a stronger type system. Consider the following example:

```
class Pair < Any>
     Fst, Snd : Any
end class;

class Point < Pair[ Fst, Snd : Real]>
end class;

class Complex < Pair[ Fst, Snd : Real]>
     alias Fst as Re, Snd as Im
end class;
```

```
class Vector < Pair[ Fst, Snd : Point]>
end class
```

An instance of class Pair cannot be affected to a variable of type Point, Complex or Vector. This typing scheme is stronger than the genericity simulation using inheritance and polymorphism.

Our implicit genericity implies that each field of the object is generic with its type constrained to be a subtype of the original type of the field.

2.2.2 Late abstraction

We have shown that the inheritance relation limits drastically software reuse in some cases because only the leaf of the inheritance tree can grow. We have devised a scheme enabling the growth of its roots, so that users can combine similar classes and give them a common father in the inheritance tree. The major problem is the definition of the similarity relation between class behaviors. The intuitive semantic description of a behavior is its signature, i.e. the types of its parameters. This relation is necessary but not sufficient, the set of behavior with the same signature is quite larger than the set of similar behaviors. Currently, the use of this only relation requires that the user provides the lacking information. Then, coherence checks are performed by the system. The next step will be to use the scheme for using signature isomorphism to search functions in libraries (see [Rit91]) to reduce the set of similar behaviors. Other opportunities come from the theory of effects which can be synthesized algebraically and would require no other work from the user.

To describe the fact that the class $\mathcal{A}$ is an abstraction from the class $\mathcal{B}$ and $\mathcal{C}$, the following notation is used: $\mathcal{A} = \mathcal{B} \odot \mathcal{C}$.

The following example shows the use of late abstraction to combine two pre-existing classes.

```
class BinaryTree < Any >
     Info : Any;
     Left, Right : BinaryTree;
     function Process ( par : Any; returns Any )
          let LeftC, RightC :=
               Left.Process( par ),
               Right.Process( par ) in
               ...
          end let
     end function
end class;
```

```
class TernaryTree < Any >
     Fst, Snd, Trd : TernaryTree;
     function Compute ( par : Any; returns Any )
          let FstC, SndC, TrdC :=
               Fst.Compute( par ),
               Snd.Compute( par ),
               Trd.Compute( par ) in
               ...
          end let
     end function
end class;

abstract Tree = BinaryTree, TernaryTree;
```

An instance of the class Tree is a tree which can have two or three sons at each node. A value of Tree type can be assigned to every field of Tree, BinaryTree or TernaryTree type in the initial classes.

Up to now, a semantic description of behaviors, sufficient enough to enable an automatic synthesis of the abstracted class, has not yet been designed. Thus, we still need other information provided by the user. The system can be used in two ways, either the user specifies some or all the common behaviors and the system checks whether they can really be unified, or the user asks the system which behaviors are compatible and selects among these which are the common ones.

The fist rule for comparing behaviors is that each behavior must have the same arity. Then the type compatibility of the signature must be checked with the following laws:

$$\frac{E \vdash f : U_1 \rightarrow V_1, g : U_2 \rightarrow V_2, \exists U_3, U_1 \leq U_3, U_2 \leq U_3, \exists V_3, V_1 \leq V_3, V_2 \leq V_3}{\exists h = f \odot g, h : \{U_1 \rightarrow V_1\} \& \{U_2 \rightarrow V_2\}}$$

This rule means that a common behavior can be defined if there exists both a common supertype of the parameter and of the result of both behavior.

The value of these supertypes U_3 and V_3 are computed with the following laws:

$$\begin{aligned} U_1 \leq U_2 &\Rightarrow U_3 = U_2 \\ U_1 \geq U_2 &\Rightarrow U_3 = U_1 \\ \exists U = U_1 \odot U_2 &\Rightarrow U_3 = U \\ V_1 \leq V_2 &\Rightarrow V_3 = V_2 \\ V_1 \geq V_2 &\Rightarrow V_3 = V_1 \\ \exists V = V_1 \odot V_2 &\Rightarrow V_3 = V \end{aligned}$$

The type $\{U_1 \rightarrow V_1\} \& \{U_2 \rightarrow V_2\}$ means that either the function takes a parameter of type U_1 and returns a value of type V_1, or it takes a parameter of

type U_2 and returns a value of type V_2. This type is part of the $\lambda\&$ typed λ-calculus developed by Giuseppe Longo at the LIENS (see [CGL92] and [Cas92]). There are slight differences due to the fact that each occurrence of the type of the class in its definition is substituted by the type of the late abstraction.

For example, after this substitution, the BinaryTree, TernaryTree and Tree classes will be equivalent to the following code:

```
-- This is the abstracted class, there is no implementation of its
-- behaviors in its definition, they will be done in the subclasses

class Tree < Any >
     Info : Any;
     function ProcessCompute ( par : Any returns Any )
          deferred
     end function
end class;

class BinaryTree < Tree >
     Info : Any;
     Left, Right : BinaryTree;
     function ProcessCompute ( par : Any returns Any )
          let LeftC, RightC :=
               Left.ProcessCompute( par ),
               Right.ProcessCompute( par ) in
               ...
          end let
end class;

class TernaryTree < Tree >
     Fst, Snd, Trd : TernaryTree;
     function ProcessCompute ( par : Any returns Any )
          let FstC, SndC, TrdC :=
               Fst.ProcessCompute( par ),
               Snd.ProcessCompute( par ),
               Trd.ProcessCompute( par ) in
               ...
          end let
     end function
end class;
```

The similar function `Process` and `Compute` have been combined in the unique function `ProcessCompute` and the class `Tree` has been created. It can be applied to both classes because it is defined in the class `Tree` and both inherit from it.

These extensions extend the possiblity of reusability in the FOL language. The next part will describe concisely the language itself.

3 Overview of the FOL language

The subject of the study leading to the conception of the FOL language was *"Introduction of object oriented paradigm in logic and functional languages"*. Its first part led to the realization of the CIEL programming language which uses logic resolution as an evaluation mechanism for a class based language (see [Gan88]). This study concerned mainly the class model and the associated type inference system.

Some of its results pointed out several problems about the merging of classic object oriented features and non-imperative paradigms. One of the major problems was that objects are often recognized as means of storing data and are required to be persistent and prone to side effects. The resolution mechanism of CIEL involved copying of data and forbade side effects as every logic based language does. In the functional paradigm, side effects are possible and sometimes very useful for the user. Thus, the second part of the study, concerning the functional paradigm, was split in two parts. The first one concerning the data type model and reuse problem, led to the definition of FOL . The second one, related to the problem of data persistence and impure effects, will be the subject of future work. The choice of using a classic functional language as an assembly language permitted to limit the study to the data model description parts of the language.

3.1 Its syntax

FOL being an extension of SISAL , its syntax was designed, so that classic SISAL sources could be translated without any problems. The definition of a class is loosely modeled after the definition of a record type.

3.1.1 The class structure

The definition of a class in FOL is constituted by the following parts:

- The ancestor of the class
- New fields and redefinition of old ones
- New functions and redefinition of old ones

The definition of new behaviors which are a mere composition of the class field behaviors can be achieved by a mechanism of parallel evaluation. This means that the behavior of each of the concerned fields will be applied, leading

to the elaboration of a new instance, which will be the same for all the field behaviors.

A few examples of FOL classes, describing a part of a formal derivation system, are given below:

```
class Expr < Any >

--  The behavior Derive is deffered which means that it must be
--  defined in each of Expr subclasses

     function Derive ( returns Expr )
          deferred
     end function
end class;

class Const < Expr >
     Value : Integer;
     function Derive ( returns Const )
          returns instance Const [ Value : 0 ]
     end function
end class;

class Variable < Expr >
     function Derive ( returns Const )
          returns Const [ Value : 1 ]
     end function
end class;

class Plus < Expr >
     Left, Right : Expr;
     function Derive ( returns Plus )
          returns instance Plus [ Right : Right.Derive;
                                  Left : Left.Derive ]
     end function
end Plus;
```

```
class Times < Expr >
     Left, Right : Expr;
     function Derive ( returns Plus )
          let LocalLeft, LocalRight :=
               instance Times [ Left : Left.Derive;
                                Right : Right ],
               instance Times [ Left : Left;
                                Right : Derive.Right ] in
               returns instance Times [ Left : LocalLeft;
                                        Right : LocalRight ]
          end let
     end function
end class;

class Main
     function Execute ( returns Null )
          let val := instance Plus
               [ Left : instance Times
                    [ Left : instance Const
                         [ Value : 2 ];
                    Right : instance Variable ];
               Right : instance Variable ]; in
               val.Derive
           end let
     end function
end class
```

3.2 Its semantics

The FOL semantics is a combining of the semantics of classic imperative object oriented languages as Eiffel (see [Cox87] and [Mey88]) and classic typed functional languages as ML, Miranda or Haskell (see [Tur86], [HJW92], [BW88], [CH90], [Mau92] and [Wik87]).

3.2.1 Type checking system

FOL type system is an extension of the Damas Milner polymorphic λ-calculus with *overloading*, *subtyping* and *intersection types*.

The subtyping relation means that you can assign to a variable of a given type every value of a lower type. So, a type structure must be a super structure of each of its supertype, which means that every structural action which can be applied to a given type should be appliable to every lower type.

The subtyping relation in FOL is built from the inheritance and the abstraction relation. We associate a type τ to each class $\mathcal{C}$. We use the following notation for this association: $\mathcal{C} \rightsquigarrow \tau$. The rules for subtyping are the following:

$$\frac{\mathcal{A} \rightsquigarrow \alpha, \mathcal{B} \rightsquigarrow \beta, \mathcal{A} \searrow \mathcal{B}}{\alpha > \beta}$$

This rule means that the type inferred by the heir of a class is inferior to the type inferred by this class.

$$\frac{\mathcal{A} \rightsquigarrow \alpha, \mathcal{B} \rightsquigarrow \beta, \mathcal{C} \rightsquigarrow \chi, \mathcal{A} = \mathcal{B} \odot \mathcal{C}}{\beta < \alpha, \chi < \alpha}$$

The type α in the previous rule is such that: $\forall \tau, \beta < \tau \vee \chi < \tau \Rightarrow \alpha < \tau$.

This rule means that the type inferred by an abstraction is the nearest type, greater than the types inferred by the abstracted classes.

$$\frac{U_2 \leq U_1 \wedge V_1 \leq V_2}{U_1 \rightarrow V_1 \leq U_2 \rightarrow V_2}$$

This subtyping rule enables to assign a function to a variable of a higher type. This function will have to accept every argument the higher typed one would have accepted, so its parameter type should be higher. It will also have to return a value whose type is accepted by every assignment made by applying the higher function. So its return type must be lower than the return type of the higher one.

$$\frac{\forall i \in \mathcal{I}, \exists j \in \mathcal{J}, U_{2,i} \leq U_{1,j} \wedge V_{1,j} \leq V_{2,i}}{\{U_{1,j} \rightarrow V_{1,j}\}_{j \in \mathcal{J}} \leq \{U_{2,i} \rightarrow V_{2,i}\}_{i \in \mathcal{I}}}$$

This rule extends the previous one to the case of overloaded functions. Each of the possible functions of the higher type must correspond to one of the functions of the lower type. This one can have more overloaded functions than the higher one.

The use of records to describe class components requires the following type-checking rule which describes the record sub-type relation:

$$\frac{\forall i \in [1..m]\ U_i \leq V_i}{\langle\langle u_1 : U_1; \ldots; u_{m+n} : U_{m+n}\rangle\rangle \leq \langle\langle v_1 : V_1; \cdots; v_m : V_m\rangle\rangle}$$

It means that an heir of a given class can have more fields than this one has and that it can redefine the type of its fields if the new ones are subtypes of the old ones.

These type checking rules are based on an extension of the Damas Milner system proposed by Giuseppe Longo at the LIENS (see [CGL92] and [Cas92]). It is an extension of the typed λ-calculus with subtyping and overloading called $\lambda\&$.

3.2.2 Behavior selection system

Classic behavior selection in the Damas Milner type system consists in choosing to apply the function whose signature corresponds to the list of its parameter types. In the case of a curryfied function, we have the following rule:

$$\frac{E \vdash x : U, f : U \rightarrow V}{E \vdash f(x) : V}$$

In FOL , the selection scheme is more complex due to the nature of the subtyping and overloading relations. So the previous rule becomes:

$$\frac{E \vdash x : U, f : V \rightarrow W, U \leq V}{E \vdash f(x) : W}$$

This rule enables to call a function with parameters whose types are subtypes of the function signature.

The following rule shows which behavior will be chosen in case of overloading:

$$\frac{E \vdash x : U, f : \{U_1 \rightarrow V_1\} \& \cdots \& \{U_n \rightarrow V_n\}, U_i = \min_{j \in 1 \cdots n} U_j \leq U}{E \vdash f(x) : V_i}$$

There are several signatures for the same fuction name. The chosen one must be the closest to the types of the parameters in respect to the subtyping relation.

This rule extends the classic object oriented dynamic binding mechanism: Usually, the selection of the correct behavior is made by looking at the real type of the first parameter. The curryfication mechanism extends this selection to the real type of each parameter.

3.3 From FOL to SISAL

The SISAL language provides all the mechanisms for the implementation of our reusability paradigm. Therefore, we decided to implement the FOL language as a translator to SISAL source.

3.3.1 Translation scheme

Each FOL class will be translated in a record type corresponding to the fields of the class and in a list of function definitions corresponding to the behaviors of the class. The name of the class will be appended to each SISAL definition to reduce the possibility of conflict with pure SISAL code.

The most complex part of the scheme is the implementation of the dynamic binding mechanism. A variable of a given type may contain a value of one of its subtype. This may only be implemented using type sets. One way of implementing late binding is to define a type set for each branch of the subtype

hierarchy. Each function will be defined once for each different signature and the application of the correct definition will be done using a type case structure.

The translation of a class generates the following SISAL constructs: a record type made of the class fields and of its ancestors' ones, a type set which is the union of the new record and of its subclass type sets, the definition of each function corresponding to the behavior of the class. For each existing behavior in the inheritance hierarchy, a dispatching function is generated. This one takes type sets as parameters and selects which class behavior is used with a type case on its argument runtime types.

The previous example about formal derivation will be translated as the following SISAL code:

```
-- This is the translation of the Expr class

type R_fol_Expr = empty;

-- This type will enable the manipulation of subtype variable

type TS_fol_Expr = R_fol_Expr
     + TS_fol_Const
     + TS_fol_Variable
     + TS_fol_Plus
     + TS_fol_Times;

-- This is the translation of the Const class

-- The prefix R_fol_ indicates the record associated to a class

type R_fol_Const = record [ Value : Integer ];

-- The prefix TS_fol_ indicates the type set used to implement
-- polymorphism

type TS_fol_Const = R_fol_Const;

-- The prefix F_fol_''class name'' indicates the SISAL implementation
-- of a class behavior

function F_fol_Const_Derive ( returns TS_fol_Const )
     returns record R_fol_Const [ Value : 0 ]
end function;
```

```
-- This is the translation of the Variable class

type R_fol_Variable = record [ Value : Integer ];

type TS_fol_Variable = R_fol_Variable;

function F_fol_Variable_Derive ( returns TS_fol_Const )
     returns record R_fol_Const [ Value : 1 ]
end function

-- This is the translation of the Plus class

type R_fol_Plus = record [ Left, Right : TS_fol_Expr ];

type TS_fol_Plus = R_fol_Plus;

function F_fol_Plus_Derive ( P_fol_obj : R_fol_Plus returns TS_fol_Plus )
     returns record R_fol_Plus [
          Left : F_fol_Derive ( P_fol_obj.Left );
          Right : F_fol_Derive ( P_fol_obj.Right )]
end function

-- This is the translation of the Times class

type R_fol_Times = record [ Left, Right : TS_fol_Expr ];

type TS_fol_Times = R_fol_Times;

function F_fol_Plus_Derive ( P_fol_obj : R_fol_Plus returns TS_fol_Plus )
     returns record R_fol_Plus [
          Left : F_fol_Derive ( P_fol_obj.Left );
          Right : F_fol_Derive ( P_fol_obj.Right )]
end function
```

```
-- This dispatch function is the key to the dynamic binding mechanism

-- The prefix F_fol_ indicates the dynamic binded behavior common to
-- every class

function F_fol_Plus ( P_fol_obj : TS_fol_Expr returns TS_fol_Expr )
     case type ( P_fol_obj )
          R_fol_Const : returns F_fol_Const_Derive ( P_fol_obj );
          R_fol_Variable : returns F_Variable_Derive ( P_fol_obj );
          R_fol_Plus : returns F_fol_Const_Plus_Derive ( P_fol_obj );
          R_fol_Times : returns F_fol_Times_Derive ( P_fol_obj );
     end case
end function
```

3.3.2 Optimization

The dynamic binding mechanism consumes a lot of CPU time. Thus, using a static binding detection scheme to reduce its use is a very efficient optimization. In the case of static binding, the global performances are these of classic SISAL code. Currently, the detection scheme concludes when an instance of a class cannot have been over-typed, i. e. in the body of the let defining this variable.

3.3.3 The librarian

To avoid parsing every class each time we use the translator, some information about all the classes previously parsed, is kept in the librarian. This part of the system is also in charge of the generation of the dispatching function and of the backpatch of the type sets for each class whose heirs have changed.

4 Conclusions and Perspectives

The first part of our project about improvement of the user condition by providing better languages and systems leads to the realization of the FOL language. The evaluation mechanism used is the pure typed λ-calculus. The second part of the project, currently under investigation, studies the modeling of impure effects. This will provide the easiness brought by impure effects to the programmer without losing the reference transparency of the model. The backbone of this work will be the study of the linear type system (see [Gir87], [Laf88], [Ode], [Ode92], [Wad90a] and [Wad91]) and the monad programming paradigm (see [Wad90b] and [Wad92]). A new semantic description of behaviors based on constraint equations and effect systems using algebraic reconstruction algorithms (see [JG] and [JG88]) will reduce the need for user intervention in the late abstraction mechanism.

References

[Bac78] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978. 1977 ACM Turing Award Lecture.

[BCFO91] A. P. W. Böhm, D. C. Cann, J. T. Feo, and R. R. Oldehoeft. Sisal reference manual, language version 2.0. Technical report, Lawrence Livermore National Laboratory, December 1991.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programing*. Prentice Hall, 1988.

[Can] David Cann. Retire fortran? a debate rekindled. Technical report, Lawrence Livermore National Laboratory.

[Cas92] Giuseppe Castagna. Strong typing in object-oriented paradigms. Technical Report 92-11, Laboratoire Informatique de l'Ecole Normale Supérieure, May 1992.

[CGL92] Guiseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. Technical report, Laboratoire d'Informatique de l'Ecole Normale Supérieure, Paris, France, February 1992.

[CH90] Guy Cousineau and Gérard Huet. The caml primer. Calcul Symbolique, Programmation et Intelligence Artificielle RT 122, Institut National de Recherche en Informatique et en Automatique, Domaine de Voluceau, Rocquencourt BP 105, 78153 Le Chesnay Cedex, France, September 1990.

[Cox87] Brad J. Cox. *Object Oriented Programming, An Evolutionary Approach.* Addison Wesley, 1987.

[Gan88] Marcel Gandriau. *CIEL: Classes et Instances En Logique.* Thèse de docteur ingénieur, Institut National Polytechnique de Toulouse, May 1988.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–102, 1987.

[HJW92] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Haskell a non-strict, purely functional language. Technical report, Yale University and University of Glasgow, 1992.

[JG] Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. Technical report, Ecole des Mines de Paris and Laboratory for Computer Science, Massachusetts Institute of Technology.

[JG88] Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. Technical report, Ecole des Mines de Paris and Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[Laf88] Yves Lafont. The linear abstract machine. *Theoretical computer science*, 59(1,2):157–181, March 1988.

[LHKS92] John A. Lewis, Sallie M. Henry, Dennis G. Kafura, and Robert S. Schulman. On the relationship between the object oriented paradigm and software reuse: An empirical investigation. *Journal of Object Oriented Programming*, 5(4):35–42, July 1992.

[Mau92] Michel Mauny. Functional programming using caml light. Technical report, Institut National de Recherche en Informatique et Automatique, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, France, April 1992.

[Mey88] Bertrand Meyer. *Object Oriented Software Construction.* Prentice Hall, 1988.

[Ode] Martin Odersky. How to make destructive updates less destructive. In *Proc. 18th Symp. on Principles of Programming Languages.*

[Ode92] Martin Odersky. Observers for linear types. Technical report, Department of Computer Science, Yale University, Yale University, Box 2158, Yale Station, New Haven, CT 06520, February 1992.

[Rit91] Mikael Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1):71–89, January 1991.

[Tur86] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Lecture Notes in Computer Science*, volume 201. Springer Verlag, 1986.

[VL88] John M. Vlissides and Mark A. Linton. Applying object oriented design to structured graphics. In *Proc. of the USENIX C++ Conference, Denver, Colorado*, October 1988.

[Wad90a] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 1990.

[Wad90b] Philip Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*. North Holland, 1990.

[Wad91] Philip Wadler. There's no substitute for linear logic. Technical report, University of Glasgow, University of Glasgow, G12 8QQ, Scotland, December 1991.

[Wad92] Philip Wadler. The essence of functional programming. In *Proc. 19th Symp. on Principles of Programming Languages*, January 1992.

[Wik87] Åke Wikström. *Functional Programming using Standard ML*. Prentice Hall, 1987.

Twine: a Portable, Extensible Sisal Execution Kernel*

Patrick J. Miller

November 4, 1992

1 Introduction

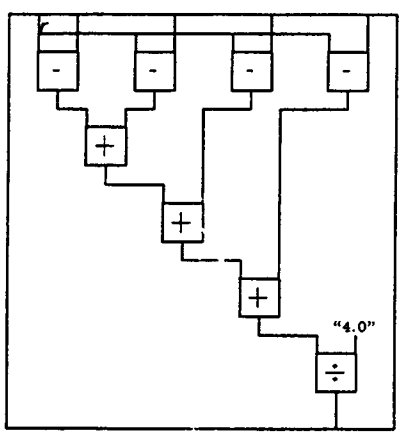
Functional programs are very much like black boxes – inputs are applied to programs that produce outputs. We need tools to help us peer into these boxes and find out what's going on. Some of the traditional tools programmers use to augment programs (print statements, monitors, histogrammers) are impossible to describe in the functional paradigm. This paper presents ways to instrument a computational engine that is simulating a functional execution. The TWINE engine executes a serialized, compiled version of Sisal's intermediate form IF1. The tool generation environment allows moderately sophisticated users to write packages that can instrument running Sisal programs. The engine provides a clean abstraction of the execution, so the tool writer does not have to worry as much about the internal execution of the program. The engine talks to tools through an *event-driven* interface. This paper gives an overview as to how the engine works, and how Sisal codes are compiled for it. The paper concludes with an example which constructs an instrument which monitors array sizes.

2 Overview

TWINE is primarily a sequential execution engine for IF1 like languages (hereafter denoted IFx). IFx codes are denoted by acyclic, hierarchical graphs with iteration, recursion, and function application implied by the semantics of its operations. Operations are denoted by nodes of the graph, while values are carried on the edges of the graph. These graphical programs are serialized to allow *frame based* sequential execution. These frames are mapped to functions (YIFT conversion) that define automatic values and that drive the engine. The engine contains instruction definitions

* This work was supported (in part) by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

```
define Avg
function Avg(x,N,E,W,S : real
       returns real)
       ((x-N)+(x-E)+
        (x-W)+(x-S))/4.0
end function
```



$\Longleftrightarrow$

```
X 14   "avg"
E  8 1  0 1   Real   %na=T8
N  1   Minus
E  0 1  1 1   Real   %na=x
E  0 2  1 2   Real   %na=N
N  2   Minus
E  0 1  2 1   Real   %na=x
E  0 3  2 2   Real   %na=E
N  3   Minus
E  0 1  3 1   Real   %na=x
E  0 4  3 2   Real   %na=W
N  4   Minus
E  0 1  4 1   Real   %na=x
E  0 5  4 2   Real   %na=S
N  5   Plus
E  1 1  5 1   Real   %na=T1
E  2 1  5 2   Real   %na=T2
N  6   Plus
E  5 1  6 1   Real   %na=T3
E  3 1  6 2   Real   %na=T4
N  7   Plus
E  6 1  7 1   Real   %na=T5
E  4 1  7 2   Real   %na=T6
N  8   Div
E  7 1  8 1   Real   %na=T7
L      8 2   Real   "4.0"
```

Figure 1: Dataflow specification of a Sisal code fragment

(a sort of μ-code table), I/O managment routines, dynamic memory (Heap) mangment routines, an ALU interface, event signal generators (explained below), a system state stack, and some system status variables.

2.1 Serialization

IFx codes are really dataflow graphs, so execution order is not explicit, but rather is implicitly controlled by data dependency information contained in the edges. Many parallel and sequential orderings are possible. Since TWINE is sequential, we only concern ourselves with sequential orderings. Consider the code fragment, dataflow graph, and associated IF1 code in Figure 1. For the sequential engine to work, we must serialize the nodes (operations) in graph and assign frame offsets to the edges (values). Many orders are possible: some orders follow the original source order, some follow a more optimized path (e.g. common sub-expressions eliminated and migrated), and some orders may be more eager in evaluation. Figure 2 shows the

```
Integer x; Integer N;
Integer E; Integer W;
Integer S; Integer T1;
Integer T2; Integer T3;
Integer T4; Integer T5;
Integer T6; Integer T7;
Integer T8;

T1 := x-N;
T2 := x-E;
T3 := T1+T2;
T4 := x-W;
T5 := T3+T4;
T6 := x-S;
T7 := T5+T6;
T8 := T7/4.0;
```

Source Ordering

```
Integer x; Integer N;
Integer E; Integer W;
Integer S; Integer T1;
Integer T2; Integer T3;
Integer T4; Integer T5;
Integer T6; Integer T7;
Integer T8;

T1 := x-N;
T2 := x-E;
T4 := x-W;
T6 := x-S;
T3 := T1+T2;
T5 := T3+T4;
T7 := T5+T6;
T8 := T7/4.0;
```

Eager Ordering

Figure 2: Serializing an IFx graph into a frame

source order and eager order for the code in Figure 1.

2.2 YIFT conversion

TWINE converts these serialized graphs into C code that interfaces with the engine. This is done with a tool called YIFT (Yet another Intermediate Form Translator). IFx types are represented with C structures, and edge names and literals are pooled. A table of IFx graphs is constructed with pointers to C functions which contain the serialized graphs. Figure 3 shows a segment of the YIFT output. Note that no semantic interpretation is done. Operations are denoted only by IFx opcodes. The engine and its runtime make all execution decisions. YIFT just recreates the IFx graph in C form.

2.3 TWINE Engine

The TWINE engine is a runtime library that is linked with the YIFT'ed code. Conceptually, it consists of a μ-code table, an ALU interface, a dynamic memory manager, a system state interface, and an event interface. Figure 4 shows the structure of these components. The YIFT'ed code is plugged into the engine (dashed box), and the Controller manages the execution. Later, we will see how to design "Packages" to interface with the engine.

```
#include "SC.h"
#define NodeTable Frag_NT
#define SourceTable Frag_ST
...
/* Type Table */
ForwardBasic(Type6);
DefineBasic(Type6,{"Real"},IF1Real);

/* Edge Name Pool */
static char ENP[][16] = {
        "NULL","EDGE_9.1_TO_1.1","S","X", ...
};

/* Literal Pool */
static char LP[][4] = {
        "4.0",  /* 1 */
};
LiteralPool{
        Literal(Type6,LP+0),
};
...
NodeStuff NodeTable[TableSize] = {
        EntryInformation(1,1,"Frag"),
        {1,1,5,NTypes+0 NTypes+0,NULL,NodeTable+1,
          Graph,TRUE,{7,8,NULL,NULL,GP(GN+0),GP(AVG_),GP(AVG_Setup)},{3}}};

/* Code */
GlobalFunction(AVG_) {
        EnterGraph(1,7,2,1);

        /* Argument/Result pointers are placed on a stack */
        ArgInput(1,1,CP(ENP+3));/* X */
        ArgInput(2,2,CP(ENP+6));/* N */
        LocalOutput(1,1,CP(ENP+7)); /* EDGE_2.1_TO_6.1 */
        /* And passed to a node execution function in the kernel */
        NodeExecute(2,2,1,NTypes+0,NTypes+0,NULL,
                NodeTable+1,Simple,FALSE,{135},{4});

...

        LocalInput(1,7,CP(ENP+13));/* EDGE_8.1_TO_9.1 */
        LiteralInput(2,1);/* 4.0 */
        ArgOutput(1,1,CP(ENP+1)); /* EDGE_9.1_TO_1.1 */
        NodeExecute(9,2,1,NTypes+0,NTypes+0,NULL,
                NodeTable+1,Simple,FALSE,{122},{4});

        ExitGraph(1);
}
```

Figure 3: YIFT translation

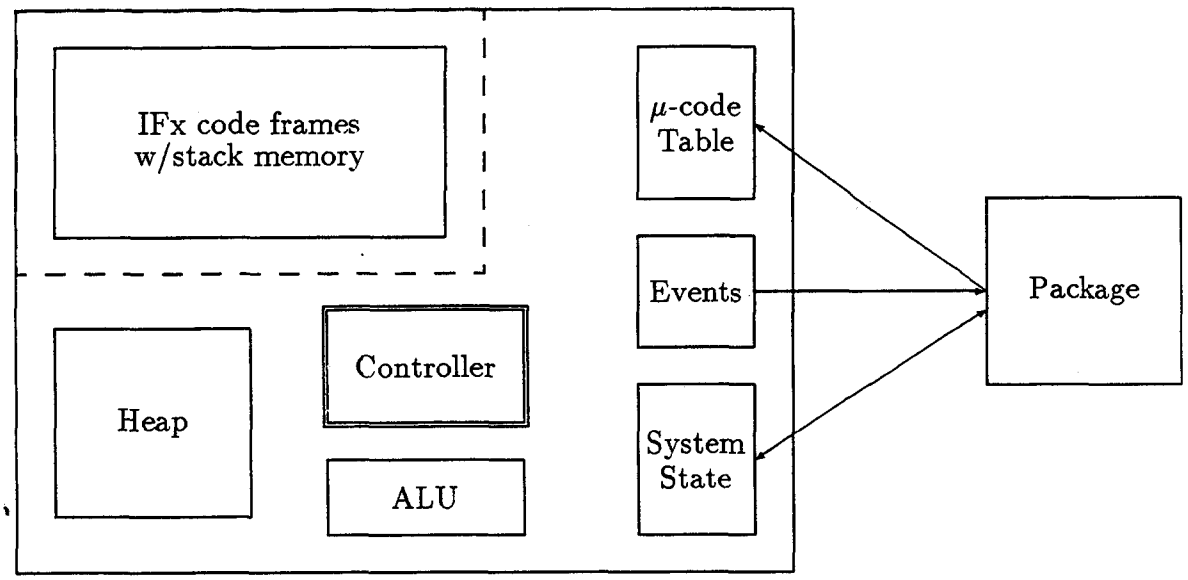


Figure 4: The TWINE engine

The μ-code table lets packages modify and extend the IFx semantics. For instance, if a new IFx node with op code 225 were developed, the package could execute:

```
void NewRoutine();
OpTable[225].Code = NewRoutine;
OpTable[225].Name = "New OP";
```

to provide the new definition. Similarly, the package could execute

```
void MyPlus();
OldPlus = OpTable[IFPlus].Code;
OpTable[IFPlus].Code = MyPlus;
```

to override the definition for an existing operation. The dynamic memory manager controls access to IFx dynamic objects. All dynamic objects share a common, extendible structure known as a bag. These bags are really restricted B-trees (BAG = B-Tree Aggregate) that can be arbitrarily extended (although access time is proportional to the height of the tree). Garbage collection scavenges unreferenced bags periodically to keep the heap compact. Different ALU's can be compiled into the TWINE kernel. This allows TWINE to act as a high level interface to new mathematics libraries. This facility can be used to make TWINE simulate Cray arithmetic and precision on a workstation for example. The system state and event generator will be discussed along with packages.

Table 1: Kernel Events

Signal	Event
PreStart	Access at program startup. Allows control of t.out options
Start	Execution is about to begin.
Function	A function graph is executing.
Graph	A graph is executing.
EndGraph	A function or graph is terminating.
PreNode	A simple or compound operation is executing.
PostNode	A simple or compound operation is terminating.
Error	An error value has been generated.
Forall	A IFForall compound is executing.
Complete	Execution ends.

3 Packages

The kernel can create stand-alone programs in the manner of a emulator, but other than results, we can retrieve no information about the execution. TWINE allows instrument packages to interface with IFx executions. The kernel and instrument package runs as co-routines, passing control back and forth. Control is passed from kernel to package when interesting "events" take place. The package scans the system state, does its work, an lets control pass back to the kernel.

3.1 Event Interface

The kernel signals interesting events to the instrument package. Table 1 lists the event signals that the kernel may generate. When these events occur, they are delivered to the `PACKAGE_HOOK` function in the instrument package. This function should decode the hook, take appropriate action, and return. The kernel will then continue computing until the next event. Figure 5 shows a very simple instrument package that counts function invocations.

3.2 System State

Sometimes, the event signal contains all information an instrument package needs. For example, the package in Figure 5 only needed to know that a function was being invoked, not which one. The package can get more information from the kernel. Most of this information is contained in the system execution stack. At the time of the signal, the top of the stack will contain the node or graph currently being executed. Entries lower in the stack contain the dynamic chain of graphs, compound operations, and functions leading up to program entry. Consider the code:

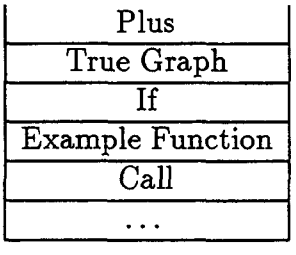
```
void
PACKAGE_HOOK(hook)
        PackageHookType hook;
{
        static int count = 0;

        switch( hook ) {
        case PackageFunction:
                count++;
                break;
        case PackageComplete:
                printf("%d function calls\n",count);
                break;
        }
}
```

Figure 5: A simple PACKAGE_HOOK function

```
function Example(n :  integer returns integer)
       if ( n < 0 ) then n+1 else n-1 end if
end function
```

If an event were signaled at the n+1 operation, then the stack would look like



Each entry in the system execution stack contains pointers to inputs, outputs, local data, and structure. The instrument can see the entire active state of the computation by tracing back through the stack. Functions are provided that return the top entry in the stack or that return useful information about the computation. Some system variables can be set that can affect computation or I/O formatting. Table 2 lists some of these.

Table 2: Important System Interface Functions and Variables

Name	Kind	Description
`TopOfEnvironmentStack()`	Function	Returns top entry of stack
`GetFunctionNode(x)`	Function	First function lower than entry x
`FibrePrint(x)`	Function	Display x to output
`PrintMax`	Variable	Max number of elements to print (0=∞)
`InfiniteLoopCount`	Variable	Max number of iterations or recursions (0=∞) before setting `InfiniteLoopFlag`
`InfiniteLoopFlag`	Variable	TRUE if in an infinite loop or recursion Stays TRUE until explicitly reset
`IFormat`	Variable	Integer format for output
`RFormat`	Variable	Real format for output
`DFormat`	Variable	Double format for output

4 A simple example

This section will show how to interface a simple monitoring tool to a TWINE program. It will describe some of the structure of system stack entries and show how to get at runtime values. It will also show how to use the `IF1OBJECT` union type.

Suppose we have a Sisal program in which we wish to know the average size of an array or stream that is being scattered in a Forall loop. From IF1 semantics, we know that these arrays enter an `IFAScatter` operation. If we check every node entry event, we can detect all `IFAScatter` operations. We can then make a running total of array sizes and entry counts. At the end of the program, print the average.

4.1 Event Handler

Since we use an event-driven interface, we use an event handler function `PACKAGE_HOOK` to interpret events. Figure 6 shows the outline of the handler function for the array average instrument. The include files `SC.h` and `SCLib.h` define TWINE structures and accessors. The functions `AddLocals` and `InitializeGraphLocals` are required. Their purpose is to bind names to edges for debugger instruments, and they are not important here. The event handler function is simply a switch to decode the events we need to instrument. Here, the `NodeCheck` function (described later) will check for `AScatter` operations, and the `Finish` function will print out the final average.

4.2 Getting at the system stack

When an event occurs, the top of the system stack contains the currently executing node. The routine `NodeCheck` will need to get information about that node. The

```
#include "SC.h"
#include "SCLib.h"

void AddLocals() {}              /* Required function */
void InitializeGraphLocals() {} /* Ditto */

int     N = 0, Total = 0;

void NodeCheck()
{
...
}

void Finish()
{
  if ( N > 0 ) {
    printf("# %d occurrences. %g avg size\n",
      N,((double)(Total))/((double)(N)));
  } else {
    printf("# No occurrences\n");
  }
}

void PACKAGE_HOOK(hook)
     PackageHookType    hook;
{
  switch(hook) {
   case PackagePreNode:  NodeCheck(); break;
   case PackageComplete: Finish();    break;
  }
}
```

Figure 6: Array average instrument event handler function

```
void NodeCheck()
{
#define IFAScatter      (114)

  Environment   *Top;
  NodeInfo      TopNode;
  int           EdgeCount;
  IF1OBJECT     *ArrayEdge;

  /* Get the top element of the system stack */
  Top = TopOfEnvironmentStack();
  TopNode = Top->INFO;

  /* Make sure its an AScatter node! */
  if ( OpCodeOf(TopNode) == IFAScatter ) {
    N++;                          /* Found one */

    /* Get the length of the array being scattered */
    EdgeCount = INARITY(TopNode); /* Always be one for AScatter */
    if ( EdgeCount > 0 ) {
      ArrayEdge = Top->IN[0];   /* Zero based! */
      Total     += ArrTS(ArrayEdge);
    }
  }
}
```

Figure 7: The NodeCheck function

---

body of the NodeCheck function is shown in Figure 7. We first get the top value in the system stack The structure of this "environment" entry is shown in Table 3. The node information is then retrieved into a "nodeinfo" structure. Various accessors allow the instrument programmer to get at information within this structure. Here, we find the AScatter nodes by checking the op code of the instruction, and we make sure the expected edges really exist. Then, we get the "IF1 object" from the list of instruction inputs and use an object accessor to find the size of the array and add it to the running sum. Table 4 and Table 5 show some of the important node, object, type, and bag accessors defined by TWINE. We can use these accessors to gather many kinds of information about the running process. The instrument is linked into a TWINE program (versions 1.3 and later) with the -g option. Suppose the instrument has been compiled into a file `instrument.o`, then execute the following

Table 3: TWINE System Stack Entry Structure

Environment	System Stack Entries
NAME	Name of graph, function, or instruction
INFO	Points to IFx node structure information
IN	A list of inputs to graph, function, or instruction
OUT	A list of outputs (if computed yet)
VARS	A list of local values (graphs and compounds only)
VARCOUNT	Number of locals

Table 4: Node, Type, Object, and Bag Accessors

Node Accessors (NodeInfo)		
INARITY	int	Number of input edges
OUTARITY	int	Number of output edges
IType	*TypeD	List of expected input types
OType	*TypeD	List of expected output types
NodeIDOf	int	Local offset in graph
ParentOf	NodeInfo	Parent graph or compound
ChildCountOf	int	Number of children in graph
ChildrenOf	*NodeInfo	List of children nodes
SubNodeCountOf	int	Number of sub-graphs
SubNodeOf	*NodeInfo	List of sub-graphs
OpCodeOf	int	IF1 opcode
GraphNameOf	*char	Name of graph
GraphCodeOf	FUNCTION	Code to execute
IsSimple	int	1 if is an IF1 simple node
IsCompound	int	1 if is an IF1 compound node
IsGraph	int	1 if is an IF1 graph node
IsExported	int	1 if listed in the define statement
Type Accessors (TypeD)		
TypeEntryOf	EntryType	IF1 type (IF1ARRAY, IF1BASIC, ...)
KindOfBasic	BasicType	IF1Boolean, IF1Character, ...
ElementTypeOfArray	TypeD	Base type for arrays
ElementTypeOfStream	TypeD	Base type for streams
FieldCountOfRecord	int	Number of fields for this record type
FieldTypesOfRecord	*TypeD	List of field types
TagCountOfUnion	int	Number of tags for this union type
TagTypesOfUnion	*TypeD	List of tag types

Table 5: Node, Type, Object, and Bag Accessors (continued)

Object Accessors (*IF1OBJECT)		
TypeOf	TypeD	Type structure for object
BasErr	SisalBoolean	True if an error value
BVal	SisalBoolean	Boolean value of object
CVal	SisalCharacter	Character value of object
DVal	SisalDouble	Double value of object
IVal	SisalInteger	Integer value of object
RVal	SisalReal	Real value of object
ArrIsOK	SisalBoolean	TRUE if error array
IsArrLBErr	SisalBoolean	TRUE if lower bound is an error
ArrLB	SisalInteger	Lower bound of array
ArrPS	unsigned int	Prefix size
ArrTS	unsigned int	Total size
ArrCol	BagPtr	Collection of objects
BagPos(n,A)	unsigned int	Position of nth element in array A
FunName	*char	Name of function constant
FunCode	FUNCTION	Code for function
RecErr	SisalBoolean	TRUE if record is an error
RecCol	BagPtr	List of field values
UniErr	SisalBoolean	TRUE if union is an error
UniTag	SisalInteger	Variant tag for union
UniVal	BagPtr	A one element bag holding the value
IsEmptyObject	int	1 if object hasn't been defined
Bag Accessors (BagPtr)		
PointerIntoBag(b,n)	*IF1OBJECT	Get nth object from bag b

```
% twine -g=instrument.o foo.sis ...
```

This adds the instrument to the `t.out` executable.

5 Conclusion and Warning

TWINE provides a simple instrument interface to running IFx computations. The TWINE debugger shows how powerful these instruments can be. The underlying engine supports the original IF1 semantics with error values and an event-driven instrument interface. Many kinds of instruments can be written without explicit knowledge of the kernel and without modifying the kernel. This is a great advantage over the original Sisal debugger/interpreter system (DI) which required integration of all tools into the execution kernel.

TWINE is not all powerful, however, because it executes only the strict IF1 semantics. This means that TWINE executions do not take advantage of the memory optimizations available in OSC [CO91]. Unless the tool takes this into account, it may give skewed results. Right now, TWINE does not implement the IF2 buffer type, nor does it support lazy evaluation of streams. Future revisions of TWINE will support these features.

References

[CO91] David Cann and R. R. Oldehoeft. *A Guide to the Optimizing SISAL compiler.* Manual UCRL MA-108369, Lawrence Livermore National Laboratories, September 1991.

[MC92] Patrick Miller and Walter Cedeno. *A User's Guide to TWINE.* Manual, Lawrence Livermore National Laboratories, November 1992.

[MSA*85] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert 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.

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

[SY85] Stephen Skedzielewski and R. Kim Yates. *Fibre: An external form for Sisal and IF1 data objects.* Manual M-154, Lawrence Livermore National Laboratories, Livermore, CA, January 1985.

Investigating the Memory Performance of the Optimising SISAL Compiler

Dean Engelhardt and Andrew Wendelborn*
University of Adelaide

September 23, 1992

Abstract

This paper describes additions made to the runtime system of OSC v12.7 which allow details of memory usage to be collected at discrete time steps. We also discuss experiments we have carried out with the aid of this tool, and describe how these experiments have helped us gain a better understanding of the memory characteristics of OSC and its optimisations. We further show how this understanding has enabled us to hand-optimise the memory performance of our programs to produce optimal memory performance.

1 Introduction

One of the greatest advantages SISAL has over imperative languages such as C is that it has in OSC [2] an implementation which can provide comparable execution-time performance, while removing from the programmer the burden of explicitly managing the memory over which the program runs. It can be argued that even languages such as FORTRAN force this burden upon programmers in so much as the programmer must manually handle reuse of storage to produce efficient code. In contrast to programs written in these 'memory-explicit' languages, SISAL programs abstract completely over memory by means of provided data types and single assignment semantics, and leave the task of inferring the details of memory allocation to the compiler and runtime system. Cann's OSC goes through several optimisation stages to construct and modify memory-explicit representations (expressed in IF2[7]) of the program being compiled, with C code being produced from these representations as the final output of the compiler.

The question naturally arises as to whether these memory-explicit forms produced by such a compiler are as efficient as code written directly in memory-explicit languages like FORTRAN and C. Numerous reports have been made asserting a comparable execution time performance between SISAL programs compiled with OSC and their imperative counterparts [4, 3, 1], but few if any reports have addressed issues of memory performance.

*Authors' Address: Department of Computer Science, University of Adelaide, GPO Box 498, Adelaide 5001, Australia. E-mail: {dean,andrew}@cs.adelaide.edu.au

This paper describes local additions made to the OSC runtime system which allow for the construction of a memory profile of executing SISAL programs. From these profiles graphical output can be generated using the PostScript[1] plotting program described in [6]. With the aid of this profiling tool and its graphical output we have investigated the memory performance of some simple programs, and deduced some characteristics of OSC's optimisations with respect to memory.

2 Profiling Additions to OSC

This section gives a brief background on OSC's memory management system in order that the reader may gain some impression of exactly what statistics we are collecting with our profiling system. Following this appears a description of the operation and output of this system.

The OSC memory system is a slightly modified implementation of that presented in [5]. It consists of two parts: a boundary tag pool and a number of block caches (one per worker process). The boundary tag pool is essentially a large region of memory which is broken into blocks by the presence of "zero blocks"; record structures which hold information about the memory which immediately follows in the pool. These zero blocks are linked together to form a list. A block within the boundary tag pool is in one of three states: free, allocated, or cached. Cached blocks arise as a result of the runtime system deallocating an allocated block. Rather than simply returning the block to a free state, the system marks it as cached in the hope that at some stage in the future an allocation request will occur for an amount of memory very close to, or exactly the same as, the size of the cached block. In that instance, the cached block would be granted to the calling program rather than requiring the allocator to split a free block to fill the request. Each worker process maintains a list of the (cached) blocks that have been deallocated by it — these lists are referred to as the workers' block caches.

Thus the process that takes place when an allocation request is received by the OSC runtime system is as follows: firstly a check is made of the requesting worker process' block cache to determine whether a block of exactly right or near size is present. If such a block is found it is allocated to the worker. If no such block is found, the runtime system searches the boundary tag pool to find a free block larger than the size requested. If it finds such a block it splits it into two blocks (by inserting another zero block into the pool) and allocates the appropriate portion to fill the request. If no such block is found the system is forced to flush all the block caches present within the system, and coalesce all the free blocks that result. After such a cache flush another search of the boundary tag pool for a block larger than that requested occurs, and if this second search fails, the runtime system terminates with a message to the effect that memory is full.

The provision of runtime memory profiling for the OSC system has required several changes to the allocation and signal handling code of the runtime system. Firstly, to allow for a better accounting of what allocated memory in the boundary tag pool represents, we have included an extra field in the zero block structure mentioned above. This extra field holds information

[1] Postscript is a trademark of Adobe Systems Incorporated

about what type of IF2 node brought about the allocation whose memory immediately follows the zero block. We have modified the allocation portion of OSC's runtime system to update the *creator* field of a zero block upon allocation of the block of memory it corresponds to. We have further added code to the allocation subsystem to traverse and report on the memory present within the boundary tag pool and the block caches. To perform this memory reporting at discrete time steps during the program's execution, we have made modifications to the signal handling portion of the system. Specifically we have arranged for an operating system timer to be set up at the commencement of execution of the user program. A command line option to the `s.out` binary controls the timeslice duration with which this timer is initialised. The user program executes as normal until the timer expires and the OS sends a signal to the OSC runtime system. The runtime system catches this signal, stops the timer, suspends execution of the user program and invokes the memory reporting routine mentioned above. A summary of the information gleaned from this traversal is appended to a memory profile file called `s.prof`. Once this reporting is complete, the OS timer is reset to the original timeslice duration and the execution of the user program is resumed.

At each timestep the following information is gathered via a traversal of OSC's caches and boundary tag pool and appended to the `s.prof` file:

- Amount of time actual (non-profiling) code has been running,
- Number of allocation calls made during the last time step,
- Number of these allocation requests that were handled by OSC's block cache,
- Number of times the cache needed to be flushed in the last time step,
- Amount of memory currently allocated within the system,
- A breakdown of this allocated memory by the type of IF2 node that caused it to be allocated,
- Number of free blocks presently within the boundary tag pool and their sizes,
- Number of blocks present within OSC's cache and their sizes.

The graphical plots which appear throughout the remainder of this paper are drawn from the IF2 breakdown of allocated memory.

The signal handling semantics we have mentioned above are somewhat simplified; the actual system as implemented requires certain sections of the runtime system to be made critical sections, that is not to be interruptable by signals from the timer. In addition to this, the process of stopping all worker processes in a multiprocessor system (where any number of these processes may be within critical sections) before traversal of the boundary tag pool, adds complexity to the signal handling semantics. Details of these more complicated aspects of our modified system are not given here, since these are largely irrelevant to the remainder of this paper.

3 Experiments with Memory Consumption

The main motivation for profiling the memory performance of the OSC system was to investigate the memory characteristics of the system and to compare those to the expected performance that could be achieved by writing the equivalent program in C or FORTRAN. This section presents two sets of experiments carried out with the modified runtime system and documents the results obtained from them. The profiles which appear are from a modified OSC 12.7 running on a SPARCstation under SunOS. Thus all loops are run sequentially.

The first series of experiments were with programs that constructed non-regular array structures. Shown below (in figure 1) are two SISAL codes which construct such structures; both structures are two dimensional triangular matrices of integers, the first with columns of monotonically increasing length (as the matrix is traversed row-wise), the second with monotonically decreasing columns. The memory profiles of running each of these programs with program input 2000 is shown in figures 2 and 3.

```
type aa = array [array [ integer ]];
function main (n: integer returns aa)

  for initial
    arr := array aa [];
    cnt := 1
  while (cnt < n) repeat
     arr := array_addh (old arr, array_fill (1,old cnt,1)) ;
     cnt := old cnt + 1;
  returns value of arr
  end for

end function
```

```
type aa = array [array [ integer ]];
function main (n: integer returns aa)

  for initial
    arr := array aa [];
    cnt := n
  while (cnt > 1) repeat
     arr := array_addh (old arr, array_fill (1,old cnt,1)) ;
     cnt := old cnt - 1;
  returns value of arr
  end for

end function
```

Figure 1: Code for increasing and decreasing ragged array programs

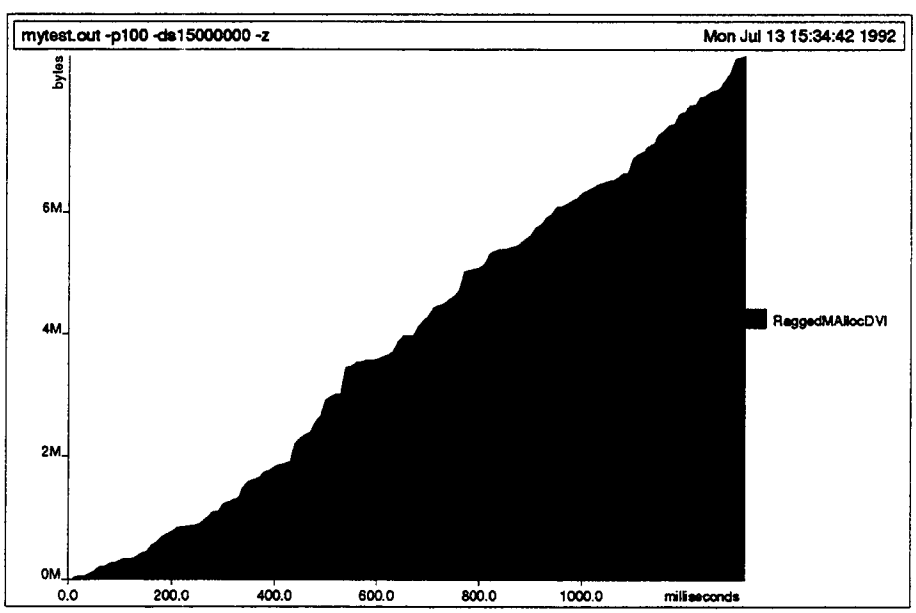


Figure 2: Profile for increasing column program of figure 1 (n=2000)

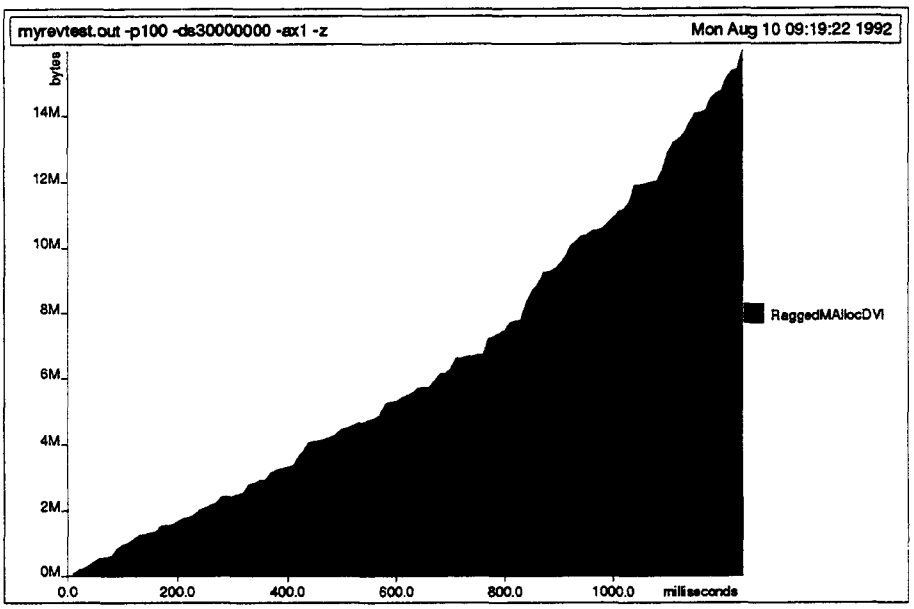


Figure 3: Profile for decreasing column program of figure 1 (n=2000)

These profiles show a memory performance that is of the expected 'shape'; that is the amount of memory allocated monotonically increases. However, the amount of memory used by the programs is significantly different. The increasing column program has a peak memory usage of approximately 8Mb, a figure which compares favourably with an equivalent C program which would need to allocate space for about 2 million integers each of which occupies 4 bytes on the SPARCstation. However, the decreasing column program has a peak memory usage approximately twice this amount, despite the fact that the same number of elements of the same type are being stored. An explanation for this disparity has yet to be determined.

A second series of experiments we have performed with the modified OSC system compare the memory requirements of three programs which calculate the same value but use different SISAL constructs to do so. The problem to be solved is very simple: a function named **sub** takes an integer as its argument and returns an integer which is the result of constructing a large triangular array and reducing it via SISAL's **sum** reduction. The code for **sub** is shown in figure 4, however details of the calculation are not as important as the function's expected memory consumption. We would expect a steep increase of memory during the construction of the array, and a steep fall following the reduction.

The particular calculation we perform is a summation of 30 calls to this function with differing arguments. There are essentially three ways of expressing this calculation: directly as a sum of function invocations, as a product form loop, or as an iterative-style loop. The SISAL code for each version is shown in figures 5, 6. The memory profiles which result from running these programs are shown in figures 7, 8, and 9 respectively. The argument passed to the program was 500 for the direct addition program and 1000 for the other two. It was not possible to complete a run of the direct addition program with 1000 as memory consumption exceeded that available on out system.

The graph for the product form (figure 8) is essentially what we would intuitively expect all

```
type aa = array [array [ integer ]];

function sub (n: integer returns integer)

let
  tmp :=
    for initial
      arr := array aa [];
      cnt := 1
    while (cnt < n) repeat
      arr := array_addh (old arr, array_fill (1, old cnt, 1));
      cnt := cnt + 1
    returns value of arr
    end for
in
  for j in tmp
  returns value of sum j[1]
  end for
end let

end function
```

Figure 4: Code for the function sub

```
function add (n: integer returns integer)

sub (n+1) + sub (n+2) + sub (n+3) + sub (n+4) + sub (n+5) +
sub (n+6) + sub (n+7) + sub (n+8) + sub (n+9) + sub (n+10) +
sub (n+11) + sub (n+12) + sub (n+13) + sub (n+14) + sub (n+15) +
sub (n+16) + sub (n+17) + sub (n+18) + sub (n+19) + sub (n+20) +
sub (n+21) + sub (n+22) + sub (n+23) + sub (n+24) + sub (n+25) +
sub (n+26) + sub (n+27) + sub (n+28) + sub (n+29) + sub (n+30)

end function
```

Figure 5: SISAL code for direct addition program

```
function forall (n: integer returns integer)

for i in 1,30
returns value of sum sub (n+i)
end for

end function
```

```
function iter (n: integer returns integer)
for initial
  i := 1
while (i < 30) repeat
  i := old i + 1
returns value of sum sub (n+i)
end for
end function
```

Figure 6: SISAL code for product form and iterative style addition programs

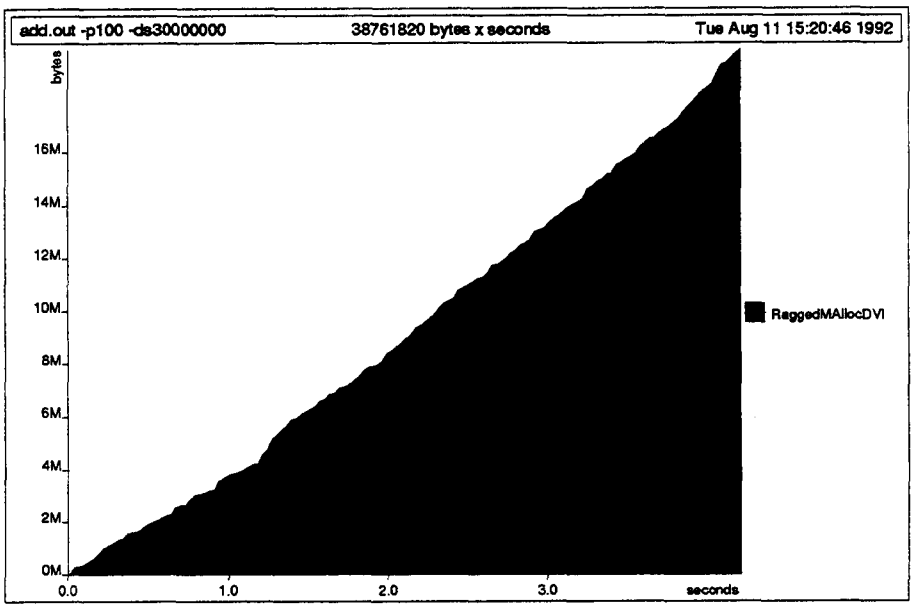


Figure 7: Memory profile of direct addition program of figure 5 (n=500)

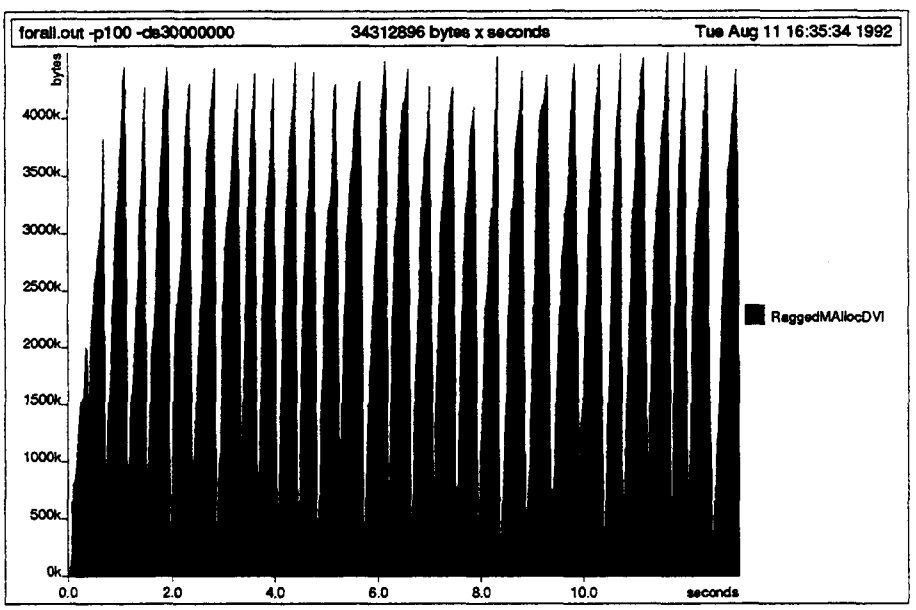


Figure 8: Memory profile of product form addition program of figure 6 (n=1000)

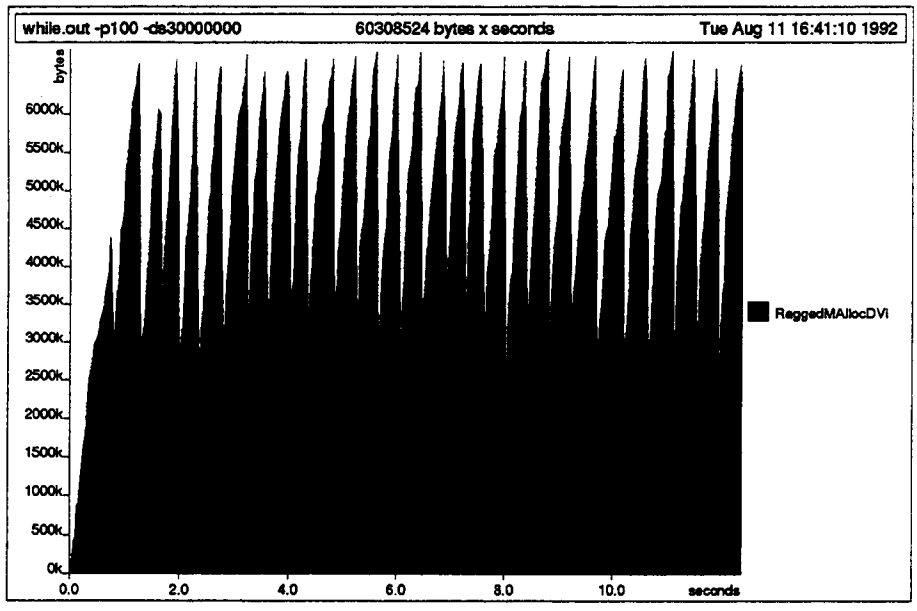


Figure 9: Memory profile of iterative addition program of figure 6 (n=1000)

three graphs to look like. That is, it consists of 30 peaks, each corresponding to a call to sub. The iterative program profile (figure 9) is similar, but shows noticeably higher memory usage than the product form. The simple addition program exhibits very poor memory performance, essentially being a monotonic allocation of memory. The runtime system in this case is obviously not performing deallocations when possible.

To investigate the differences between the performance of these three versions of the same calculation, we examined the C code produced by the OSC compiler for the different programs. Notably we found that for the simple addition case, the compiler completely inlined all calls to function sub producing a monolithic block of C code which included 30 copies of the code for sub. The C code for sub included a call to the runtime memory allocation subsystem, but *not* a corresponding deallocation of that memory following the reduction of the produced array. This explains the observed monotonically increasing memory usage. By placing such deallocations in the C code for the direct addition program, a much better memory performance was obtained. Profiles of runs of this patched program with input data 500 and 1000 are shown in figures 10 and 12 respectively. It is worth noting that an almost identical improvement in memory performance is made simply by forcing OSC to compile sub into a C function rather than inlining it.

Similar observation of the C code produced by OSC for the iterative program revealed the presence of a 'memory leak' of a similar nature. The semantics of SISAL are such that the initialisation section of a while loop counts as the first iteration of the loop as far as reductions are concerned. Thus, the C code generated for the iterative program contains two distinct copies of the code for sub; one corresponding to the initialisation section of the SISAL loop, and a second copy within a C for loop which corresponds to the body of the SISAL loop. The memory allocated by the copy of the code for sub within the C loop is deallocated as the last statement of that loop. However the memory allocated in the initialisation code is *not* deallocated. This explains the large block of memory figure 9 shows being held onto for most of the program's execution. By placing these deallocations within the C code, the

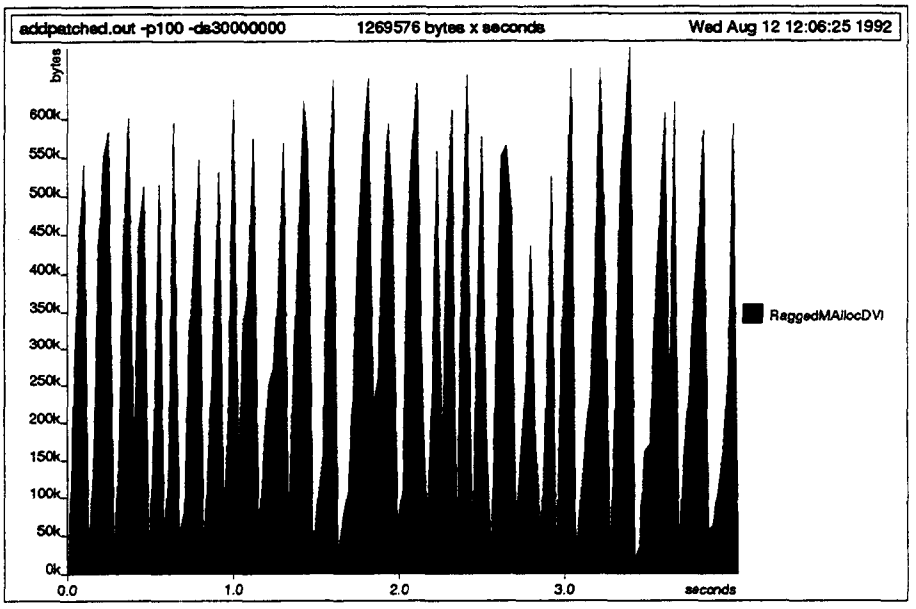


Figure 10: Memory Profile of modified version of direct addition program (n=500)

profile shown in figure 11 was obtained. The memory performance of this modified iterative program is now comparable to the performance of the product form program. Note, however that a run of the modified direct addition with program input of 1000 (the same as for the runs of the product form and iterative form) yields the profile shown in figure 12. The peak memory consumption in this program is approximately half that of the modified iterative and product form programs. This seems to indicate that in these programs, memory is still being unnecessarily held onto beyond its usefulness.

4 Experiments with Storage Reuse

A second area of OSC 12.7's memory performance we chose to investigate with our profiling tool was memory reuse. That is, whether the IF2 analyses and operations the compiler performs include a consideration of possible reuse of the memory allocated to an expression's input parameters to build its output. For example, in the program shown in figure 13, the expression which calculates `b` from `A` can be readily performed by destructively overwriting the memory allocated to `A`, since:

1. `A` is never used after the calculation of the expression in question,
2. The memory size of the expression's input (`A`) and output (`b`) are identical,
3. Each element in the result array depends solely on the corresponding element in the input array.

If OSC performs any memory reuse optimisations we would expect the generated C code for this program to calculate `b` by such a destructive writing operation. In this case we would expect the memory profile of this program to be fairly flat (that is, memory usage should be

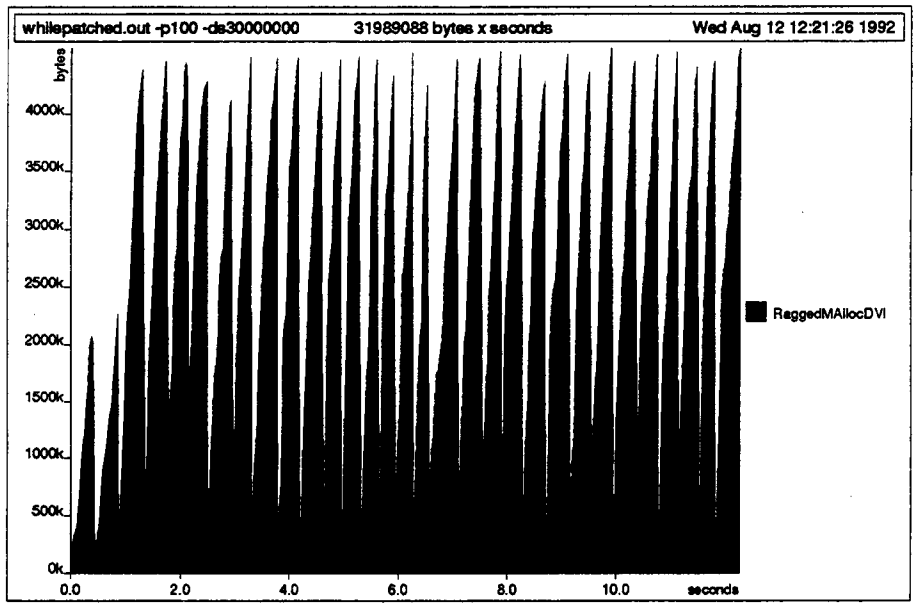


Figure 11: Memory Profile of modified iteration program (n=1000)

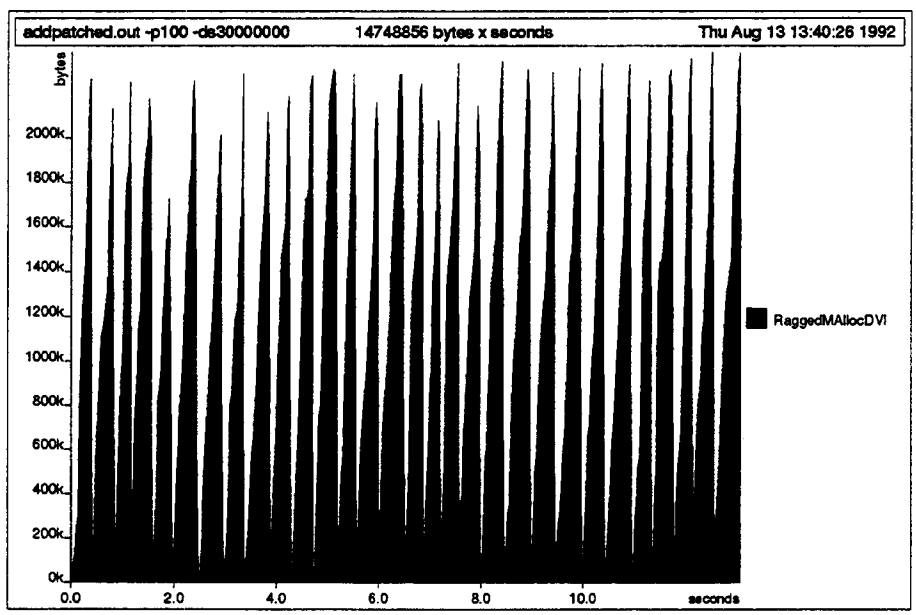


Figure 12: Memory Profile of modified direct addition program (n=1000)

constant). The observed profile, shown in figure 14, shows a flat initial region (corresponding to the calculation of c), followed by the allocation of extra memory for b. Thus we conclude that the opportunity for memory reuse is not being acted upon. This essentially halves the maximum size of the problem which may be solved with this program on a given machine.

```
function main (A: array[integer] returns array[integer], integer)

let
  n := array_limh (A);
  c := for i in 1,n
       returns value of sum i
       end for;
  b := for i in A
       returns array of i + c
       end for
in
  b,c
end let

end function
```

Figure 13: Memory reuse example

5 Future Work

The system we have described in this paper so far is a modified version of OSC 12.7. For purposes of experimenting with SISAL's stream data type (which is not properly supported by the newer OSC versions), we also use an earlier release of the OSC system. By making similar modifications to this system we have provided equivalent profiling functionality for this compiler. Some preliminary runs of this profiler have shown memory performance for identical programs run under the older system is significantly different to that observed for the program run under the newer system. In the cases we have examined so far, the pattern of memory usage under the old OSC is less prone to the jagged peaks present in the profiles of the programs discussed in section 3. Additionally, the peak memory usage of programs under the early OSC system seems generally to be lower.

In the future we hope to perform more experiments with the stream version of OSC in an attempt to isolate the effects stream parameters to the runtime system (such as buffer sizes) have on memory usage. We would also like to use our profiled stream system to allow us to examine the memory usage of a SISAL streams implementation of the `clausify` program presented by Runciman and Wakeling in [6]. In particular it would be interesting to determine whether the program modifications which lead to improvements in the LML program, lead to similar improvements in the SISAL program. We have already investigated this with a strict (array) SISAL implementation of the program with mixed results. The non-strictness afforded by streams should, however, provide a much more even comparison with the (lazy) LML program.

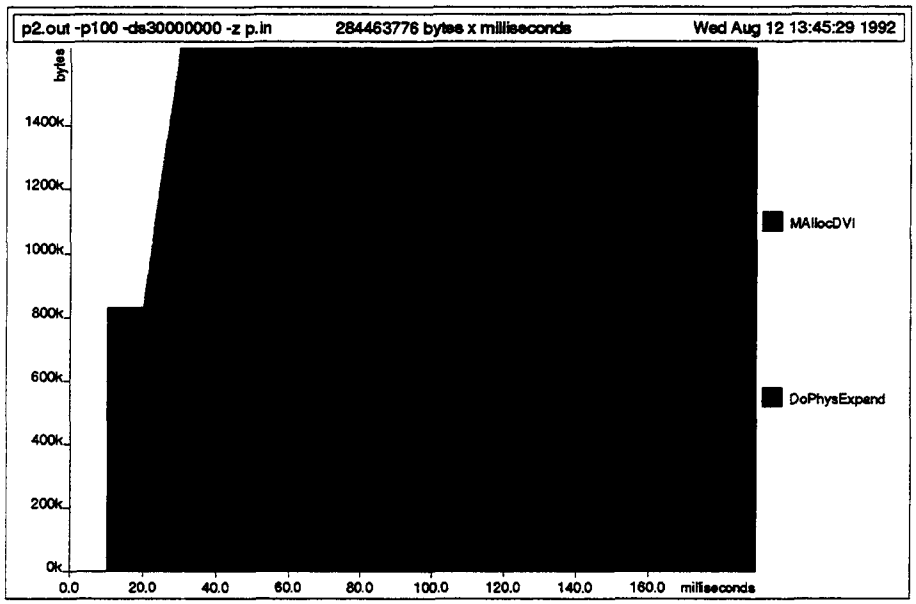


Figure 14: Profile for the memory reuse example of figure 13

6 Conclusions

In this paper we have presented a tool which allows dynamic information concerning memory allocation within the OSC system to be collected. We have shown a number of simple experiments and their results which give us some insight into the memory characteristics of the OSC system. Our experiments with memory consumption showed that in most cases the amount of memory consumed by an OSC aggregate type is roughly equivalent to that occupied by the same kind of aggregate in a language like C. Consider for example the monotonically increasing column triangular array mention in section 3: the SISAL structure consumed about 8Mb (according to the profile), while a similar C structure would consist of 2 million elements of 4 bytes each plus structuring overhead. However, in other cases, such as the decreasing column triangular array, the OSC system's representation of data structures is significantly more expensive than a similar structure represented directly in C.

The experiments documented with the direct addition, product form addition and iterative addition programs showed that in many cases the C code output from the compiler 'loses' memory by not performing deallocations when memory is no longer needed. In the direct addition program the effect of this memory loss was acute since memory for temporary arrays allocated by the code for the function sub is never deallocated which leads ultimately to the unnecessary accumulation of blocks of memory which are no longer required for the computation. Since the addition of C code to perform these deallocations at the appropriate places in the code is simple, we conclude that either insufficient analysis is performed to recognise these deallocation opportunities, or that the provision of a memory efficient system has not been a priority in the development of OSC. The memory reuse experiments we have presented in section 4 seem to indicate at least that the possibilities for reuse of memory present in the situation we tested is either not detected or not acted upon.

One of the initial goals of constructing a profiling system for OSC was to provide some help

to programmers who wish to fine tune the memory performance of their programs. That is, our original goal was to produce a system which reported the same kind of programmer-useful data as that produced by Runciman and Wakeling's system described in [6]. The tool we have produced falls significantly short of this goal; our profiles only relate memory usage back to IF2 nodes that caused the memory to be allocated, rather than to SISAL constructs. There are essentially two related difficulties we encountered in attempts to provide such a user tool:

1. The fact that the final code which is executed is the result of many optimisations makes it very difficult to relate an event in the running version of the program back to its source,

2. The task of propagating sufficient information to allow better correlation between the final code and the original source is complex since it involves extensive modification to an already complicated and largely undocumented system.

Even if the second problem were overcome, it is still uncertain as to whether sufficient accountability is attainable to make such a tool of practical use to SISAL programmers.

The tool as it stands is, however, useful to those in need of guidance in fine tuning the performance of a program. Our experiments with C code modifications to OSC output have been motivated, guided and evaluated by the profiles we have obtained from the executing program. In the case of the direct addition program we have discussed, this guidance has led to dramatic decreases in the peak and average memory usage of the program, and indeed increased the maximum size of the problem we could solve on our machine. This last concern is perhaps the most important from the perspective of SISAL as a scientific programming language; in many cases numeric algorithms operate over very large aggregate structures which will inherently consume large quantities of memory. Any tool that will lend a programmer guidance with hand optimising a program's memory characteristics will essentially be allowing problems to be solved whose size would normally preclude them from solution on the available hardware. We have demonstrated that our profiling tool can provide guidance in attaining the maximal problem size solvable on a particular machine running OSC.

References

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

[2] D C Cann. *The Optimising SISAL Compiler: Version 12.0*, 1992. Manual with OSC version 12.0.

[3] D C Cann and R R Oldehoeft. High performance parallel applicative computing. Technical Report CS-89-104, Colorado State University, February 1989.

[4] J T Feo and D C Cann. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, (10):349–366, 1990.

[5] R R Oldehoeft and S J Allen. Adaptive exact-fit storage management. *Communications of the ACM*, 28(5):506–511, May 1985.

[6] C Runciman and D Wakeling. Heap profiling of lazy languages. Technical Report 172, University of York, UK, April 1992.

[7] M Welcome, S Skedzielewski, R Yates, and J Ranalletti. IF2 an applicative language intermediate form with explicit memory management. Technical Report M-195, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.