UCRL-102737 Rev. 1
PREPRINT

# POSC — a Partitioning and Optimizing Sisal Compiler

Vivek Sarkar
IBM Research, Yorktown Heights, NY

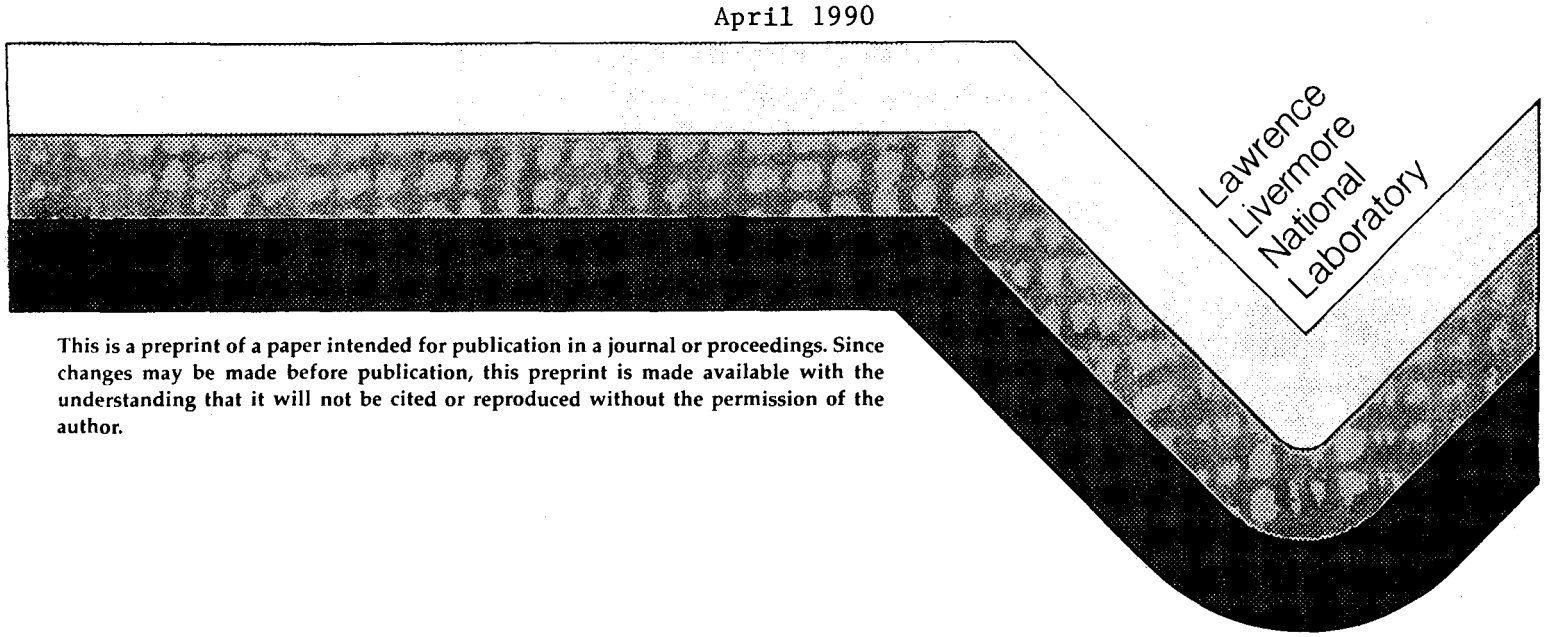
and

David C. Cann
Lawrence Livermore National Laboratory

This paper was prepared for the ICS-90:
1990 International Conference
On Supercomputing

Amsterdam, Holland, June 11-15, 1990

April 1990



---

UCRL--102737-Rev.1

DE90 010679

# POSC — a Partitioning and Optimizing SISAL Compiler

Vivek Sarkar
IBM Research
T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

David Cann*
Computing Research Group, L-306
Lawrence Livermore National Lab.
P.O. Box 808
Livermore, CA 94550

**Abstract**

Single-assignment languages like SISAL offer parallelism at all levels—among arbitrary operations, conditionals, loop iterations, and function calls. All control and data dependences are local, and can be easily determined from the program. Various studies of SISAL programs have shown that they contain massive amounts of *potential parallelism*. There are two major challenges in converting this potential parallelism into real speedup on multiprocessor systems. First, it is important to carefully select the *useful parallelism* in a SISAL program, so as to obtain good speedup by trading off parallelism with overhead. Second, it is important to do *sequential optimizations*, so that the sequential components (tasks) of the SISAL program have comparable execution times with sequential languages such as Fortran, Pascal and C. The POSC compiler system described in this paper addresses both issues by integrating previous work on efficient sequential implementation of SISAL programs with previous work on selecting the useful parallelism in a SISAL program. The combined approach is validated by real speedup measurements on a Sequent Balance multiprocessor.

## 1 Introduction

In this paper, we describe a compiler that automatically compiles programs written in the single-assignment language, SISAL [MSA+85], for efficient concurrent execution on different multiprocessors. This compiler resulted from experience with the SISAL Compiler (SC) [OC88], the Optimizing SISAL Compiler (OSC) [Can89], and from previous work on automatically partitioning SISAL programs [SH86,Sar89c]. We call the new compiler POSC—a Partitioning and Optimizing SISAL Compiler.

In the SC and OSC compilers, the program parallelism to be exploited is defined by language constructs—only Foralls, function calls, and loops that produce or consume streams are eligible for execution as parallel tasks—causing the programming style to dramatically affect multiprocessor performance. In fact, to avoid a potential situation of excessive tasking overhead with no gain in parallelism, function call parallelism was excluded from the experimental results for SC and OSC presented in [OC88] and [Can89]. In the POSC compiler described in this paper, the program parallelism to be exploited is determined automatically based on the control and data dependences in the program, the node execution times, and the multiprocessor overhead parameters. The parallelism chosen by POSC includes selected Foralls as well as calls to new "task functions" created by the partitioner. By selecting the program's task partition automatically, the same program can be made to execute efficiently on different multiprocessor systems, and the programmer is freed from considering granularity issues during program development.

The OSC compiler was developed at Colorado State University and Lawrence Livermore National Laboratory, and provides a portable fork-join implementation of SISAL using its own microtasking runtime system [CLOS87,Ric89]. The runtime system has already been ported to the Alliant, Encore and Sequent

*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.

multiprocessors and the Sun and Vax uniprocessors. The techniques described in this paper have been implemented to obtain the POSC compiler as an extension to OSC. The goal of our research is to use POSC to study the performance of various SISAL application programs on different multiprocessors.

The rest of the paper is organized as follows. Section 2 describes OSC, the optimizing SISAL compiler developed at Colorado State University and Livermore, which forms the basis for the POSC compiler. Section 3 describes the fork-join execution model supported by the runtime system used by OSC and POSC. Section 4 describes how automatic execution profiling is performed in POSC, and how the execution frequency information is used to estimate average program execution times. Section 5 discusses the problem of reordering nodes in a program to expose more parallelism for a fork-join execution model, as well as the reordering algorithm used in POSC. Section 6 describes the partitioning algorithm used in POSC to select useful parallelism from the potential parallelism in the program. Section 7 presents experimental results for the concurrent execution of SISAL programs on a Sequent Balance multiprocessor system. Section 8 discusses related work, and section 9 wraps up with conclusions and a discussion of future work.

# 2 OSC — an Optimizing SISAL Compiler

SISAL is a *single-assignment* programming language with *value-oriented semantics*—it does not have memory-update operations. All SISAL programs must satisfy the *single-assignment rule*, which states that each variable has at most one value assigned to it at runtime. Because of this rule, a variable is really a name for a *value*, rather than a name for a storage location. Whenever a new value is computed, it must be assigned to a new variable or a new instance of a variable. Therefore, single-assignment languages have no storage-related *anti* or *output* data dependences [KKP+81] and yield more parallelism than programming languages with memory-update operations and other side effects.

Unfortunately, without intelligent compilation, implementations of applicative languages like SISAL must copy data to satisfy the single-assignment rule. The use of arrays, which is common in scientific computation, makes a naive copying approach exorbitantly expensive in both time and space, thus hiding any gains from parallel execution. In SISAL, copying results from two classes of operations: those that incrementally *construct* aggregates (e.g. `array_concatenate`), and those that incrementally *update* extant aggregates (e.g. `array_replace`). Fortunately, for most SISAL programs, copy operations are inherent only to the language's semantics, and not to the algorithms in the programs.

OSC eliminates most of the copying that results from incremental aggregate constructions and incremental aggregate updates by employing special *build-in-place* and *update-in-place* analyses at compile-time. It also applies numerous conventional machine independent optimizations to further improve program performance. The compilation process in OSC proceeds as follows [Can89]:

1. Front end translation of SISAL to IF1
2. Construction of a monolithic IF1 program
3. Build-in-place analysis and translation to IF2
4. Update-in-place analysis
5. Code generation

First, a front end translates SISAL source to IF1, a graphical intermediate form for applicative languages [Sim86,SG85]. IF1 graphs are acyclic and comprise simple nodes, compound nodes, graph nodes, edges, and types. Simple nodes denote primitive operations such as addition, comparison, array replacement, etc. Compound nodes define structured expressions such as conditionals and loops. Graph nodes encapsulate functions and the subgraphs of compound nodes. Edges define the data communication among nodes, just as in dataflow graphs. Types are used to label IF1 edges with information about the transmitted data.

After the front end translation, the compiler combines all functions in the program into a monolithic IF1 program to guarantee the availability of complete inter-procedural information during optimization. The monolith is then processed by a machine-independent IF1 optimizer that performs graph normalization[1], function inlining, invariant code movement, common subexpression elimination, global common subexpression elimination, loop fusion, constant folding, and dead code removal [SW85].

The next phase in OSC is the build-in-place analyzer, which is an implementation of the techniques presented in [Ran87]. This optimization attacks the incremental construction problem by preallocating array storage wherever the final size of the array can be computed, either as a compile-time constant or as a compiler-generated expression that can be evaluated at runtime. As a result of this optimization, the compiler translates IF1 to a lower level intermediate form called IF2 [WSYR86]. IF2 is a superset of IF1 and includes operations that directly reference and manipulate memory. IF2 also includes artificial dependence edges for defining synchronization constraints on memory accesses, reference count pragmas for aggregate management, and mark pragmas for specifying aggregate access rights (immutable, mutable, possibly immutable).

Following build-in-place analysis, OSC subjects the IF2 program to update-in-place analysis to tackle the incremental update problem. Here graphs are restructured to help identify update operations that can execute in-place, and to improve chances for in-place operations at runtime where static analysis fails. The analysis proceeds in three phases. Phase one prepares each IF2 graph for later analysis by the insertion of special aggregate *duplication* nodes to decouple copy logic from all aggregate modifiers in the program. The goal of the remaining phases is to eliminate any unnecessary duplicators. Phase one also includes the annotation of each edge transmitting an aggregate with a pragma to explicitly express the program's worst-case reference count behavior. This is possible at compile-time as aggregate data in SISAL must always be acyclic. Phase two reorders the nodes in each graph, where possible, by inserting artificial dependence edges. The inserted edges delay the execution of aggregate modifiers until completion of the related read operations. This phase also removes all reference count operations that are unnecessary because of the node reordering. Lastly, phase three eliminates the unnecessary duplicate operations introduced in the first phase, and annotates all edges transmitting aggregates with the appropriate access rights. In general, all three phases are interprocedural, and are applicable to nested aggregates and loop expressions.

The last step in OSC is code generation. After applying the optimizations described above, OSC translates the optimized IF2 program into an equivalent program in C. We chose the C programming language as the target language to increase compiler portability and expedite compiler development. A drawback of this approach is that program performance will depend on the C compiler being used. Most C compilers have a reputation for producing inefficient code, compared to optimizing compilers for other languages such as Fortran and Pascal.

Sections 4, 5 and 6 describe various extensions to OSC to obtain POSC: the partitioning and optimizing SISAL compiler presented in this paper. In the current implementation of POSC, these extensions (which consist of estimation of execution times, node reordering, and task partitioning) are all performed just after the construction of the monolithic IF1 program (step 2). In the future, we would like to move these phases to after step 4, so that the estimation of execution times can use IF2-level information for greater accuracy, and the node reordering phase can also take into account the artificial dependence edges inserted by the build-in-place and update-in-place analyzers.

# 3 Fork-Join Execution Model

Because of its applicative nature, SISAL offers parallelism at several levels and can be targeted to a wide range of concurrent execution models; for example, dataflow [GKW85], macro-dataflow [SH86,Sar89c], fork-join execution with shared-memory [Can89], message-passing [GDLT86], systolic arrays [GS87], SIMD, and vector. In this section, we describe the fork-join execution model supported by OSC's runtime sys-

[1] The graph normalization phase simply restructures the intermediate form so to eliminate special cases.

tem [CLOS87]. The model can be easily and efficiently supported by all commercially available shared-memory multiprocessors. The runtime system was originally implemented on the Sequent Balance multiprocessor, and has been ported to the Alliant and Encore multiprocessors, as well as the Sun and Vax uniprocessors.

As in other microtasking systems, the OSC runtime system begins by creating a *worker* operating system process for each processor to be used in the multiprocessor system. After this, no other operating system service (except for I/O) is voluntarily requested by the runtime system. The SISAL program is compiled into a set of concurrently executable instruction streams, called *tasks*. Each worker process repeatedly picks a new task from the *ready list* and executes the task till it blocks or terminates. The tasking operations relevant to this paper are [2]:

- FORK *function call* — dynamically create a new task for the function call. Allocate and initialize its task control block (TCB) and runtime stack. Insert the new TCB in the ready list. Also increment the child count of the caller. (Supported by procedures GetStack and RListEnQ in [CLOS87].)
- SLICE *forall loop* — divide the iteration range by the runtime parameter, *LoopSlices*, and slice the Forall loop into *LoopSlices* tasks. Each task is created and inserted in the ready list, as in a FORK. The default value of *LoopSlices* equals the number of workers. If the Forall contains any (associative) reduction operations, each slice computes its partial result, and the parent task combines the partial results to obtain the final value. (Supported by procedure LoopSlicer in [CLOS87].)
- JOIN — suspend the current task, if any of its child tasks are still executing. The current task will only be moved to the ready list when all its children have completed execution. (Supported by procedure Sync in [CLOS87].)
- TERMINATE — terminate the current task and mark its TCB for deallocation. Decrement the parent's child count. If the count becomes zero and the parent task is blocked due to a JOIN operation, then move the parent task to the ready list. (Supported by procedure TermMe in [CLOS87].)

The runtime model relies on centralized task queues and shared memory. [CLOS87] discusses techniques implemented to reduce the size of critical sections in the runtime system. A task can be placed on the ready list or blocked list by any processor, and is available for execution by any processor. All its state can be restored by any processor, and all its data references are global. Structured data objects are allocated in heap storage and reference counts are maintained to decide when an object's space can be reclaimed. A more detailed description of the runtime model is given in [CLOS87].

From the compiler's viewpoint, the two constructs that generate parallelism are parallel loops and function calls. Recall that the second step in OSC (after the front end) is the construction of a monolithic IF1 program, in which function inlining has been performed wherever possible. Function call parallelism is exploited in POSC by having the partitioner select appropriate sets of IF1 nodes as new tasks, and then creating explicit IF1 functions for those tasks. Therefore, the function call tasks seen by the runtime system may be user-defined functions or task functions created by the partitioner. The IF1 program generated by the partitioner contains a very simple interface to specify the task partition:

1. For each function call node, a boolean flag, *dofork*, indicates if a FORK operation should be performed on the call or not. Note that there may be two calls to the same function, with different values of *dofork*.
2. For each Forall node, a boolean flag, *doslice*, indicates if a SLICE operation should be performed on the Forall or not. The experiments performed for this paper assume that the default value for *LoopSlices* (= number of workers) is to be used for each Forall node with *doslice* = *true*. Later on, we plan to have the partitioner specify the number of slices, or perhaps the slice thickness, for each Forall node with *doslice* = *true*.

[2] The runtime system also provides operations for managing streams, and for blocking when a task's memory allocation request cannot be satisfied. However, these events do not occur in the benchmark programs considered in this paper, since they do not use streams or run out of memory.

The JOIN and TERMINATE operations need not be specified by the partitioner, since they are automatically deduced by OSC's code generator (step 5) based on control and data dependences, and the locations of FORK and SLICE operations.

# 4 Execution Profiling and Estimation of Execution Times

An important prerequisite for the node reordering and task partitioning algorithms described later in Sections 5 and 6, is that all IF1 nodes be labeled with execution times, and all IF1 graph nodes be labeled with execution frequencies. In previous work [Sar89c], we designed and implemented a framework based on automatic execution profiling for determining average program execution times in a SISAL program. Automatic execution profiling is an empirical means of obtaining average loop frequencies and branch probabilities in a program. The idea is that the programming environment automatically gathers and stores average frequency values from previous executions of the program, and the frequency values are then used by the compiler to derive average execution times.

In the original implementation [Sar89c], execution profiling was implemented as an extension to the IF1 interpreter, DI [SYO87]. However, it is impractical to obtain frequency values from the interpreter for large program inputs. So, we extended OSC to optionally produce a sequential program with extra code to compute the execution frequency information. The extra code consists of *counter* variable declarations, initializations, and updates for tracking the execution frequency of each IF1 graph and subgraph in the program. At the end of program execution, all the counter values are dumped into a trace file, which is integrated into the IF1 file by a post-processor.

The frequency information obtained from automatic execution profiling is stored as node *pragmas* in IF1. Each subgraph of a compound node is labeled with a frequency value which gives the average number of times the subgraph is executed in a single execution of the parent compound node. The frequency value of a function graph gives the total number of times the function is called in a single execution of the program.

Apart from frequency values, the other input necessary for estimating average execution times is the set of execution time values for all *simple* nodes on the target architecture. We will not discuss the possible techniques for obtaining the costs of simple nodes. A straightforward approach is to count the number of instructions required to implement a simple node. A more careful estimation is required when considering pipelined architectures, vector instructions or the effects of cache usage. For applicative languages like SISAL, it is vital to also consider copying costs when estimating execution times. The current implementation of POSC performs execution time analysis at the IF1 level, before the build-in-place and update-in-place analyses occur at the IF2 level, and can therefore only make an approximate estimate of copying costs. In the future, POSC will perform execution time analysis at the IF2 level, where copying operations are explicitly visible as *duplication* nodes.

We now describe how average execution times are computed for all nodes in the IF1 program (see [Sar89c] for more details). The algorithm for determining average execution times is inter-procedural, so that the execution time determined for a function is passed on to all its call sites. This property dictates that execution time analysis be performed in a bottom-up traversal of the call graph. Recursive functions are also handled in this framework, as described below.

Consider an IF1 function in which execution times are known for all simple nodes and all function calls. Then, a simple linear-time algorithm can be used to obtain all execution times in a bottom-up traversal of the function's IF1 graph hierarchy, while following two simple rules:

1. $TIME(G) = (\textit{local costs}) + \sum_{N \in G} TIME(N)$,
   the average execution time of IF1 graph $G$ is the sum of the average execution times of all nodes in the graph, and any local costs for graph $G$ (for example, instructions executed in a prologue or epilogue representing startup or finishing costs).

2. $TIME(C) = (\textit{local costs}) + \sum_{G \in C} FREQ(G) \times TIME(G)$,
   the average execution time of compound node $C$ is the sum of the product of each subgraph's average frequency and execution time. We also need to add in any local costs for the compound node that were not included in any of the subgraphs' execution times.

The above approach is sufficient for computing all average execution times in a program with an acyclic call graph (which implies that it has no recursive calls). A cyclic call graph is handled by first identifying its *strongly connected components* (SCC's) [AHU74]. We distinguish between an *external* call (between functions in different SCC's) and an *internal* call (between functions in the same SCC). Clearly, the execution time of a function call depends on whether it is external or internal. An external call includes the total recursive computation in the SCC. The execution time of an internal call depends on the recursion depth at the time of the call. However, at compile-time, we need to compute a single value for the execution time of an internal call. Our approach is to assume that all internal calls in an SCC have the same average execution time, and then to compute the average value over all execution instances of the internal calls (over all recursion depths). This is the only computation that uses the execution frequencies of function graphs. The details of this computation are given in [Sar89c]. After the average execution time for all internal calls in an SCC has been obtained, all other execution times in the SCC can be computed by the algorithm outlined above.

# 5 Node Reordering

Section 2 already discussed the importance of node reordering in OSC for build-in-place and update-in-place optimizations. OSC's node reordering is performed by introducing artificial dependence edges that represent reordering constraints. The code generator is then free to choose any ordering that satisfies all the original dependences and the artificial dependences.

In this section, we discuss how node reordering can be a crucial issue for parallelism in a fork-join execution model. Consider the following SISAL function called **averages**:

```
function averages(n:integer returns integer, integer)
  let
    sum1 := for i in 1, n
            returns value of sum f(i)
            end for ;
    avg1 := sum1 / n ;
    sum2 := for i in 1, n
            returns value of sum g(i)
            end for ;
    avg2 := sum2 / n
  in
    avg1, avg2
  end let
end function
```

Function **averages** simply computes the average values of `f(i)` and `g(i)` over the range $1 \leq i \leq n$.

The main body of function **averages** contains four nodes, which correspond to the four definitions (assignments) in the `let` expression. The fork-join code generated for the original node ordering looks like:

1. SLICE forall loop for `sum1` ;
2. JOIN ;
3. *compute* `avg1 := sum1 / n` ;

4. SLICE forall loop for **sum2** ;
5. JOIN ;
6. *compute* **avg2 := sum2 / n** ;
7. *return* **avg1, avg2** ;

Note that there is a JOIN operation between the two Forall's, even though they can be invoked concurrently. This JOIN operation is due to the computation of **avg1**, but it can be eliminated by reordering the nodes so as to obtain the following fork-join code:

1. SLICE forall loop for **sum1** ;
2. SLICE forall loop for **sum2** ;
3. JOIN ;
4. *compute* **avg1 := sum1 / n** ;
5. *compute* **avg2 := sum2 / n** ;
6. *return* **avg1, avg2** ;

Not only does the new order expose more parallelism, but it also reduces overhead by using one JOIN operation instead of two.

The above example shows that node reordering is a crucial issue for increasing parallelism in a fork-join execution model. Our approach in POSC is to perform a node reordering pass before task partitioning. The goal of the node reordering pass is to recursively reorder the nodes in all graphs of an IF1 program, so as to expose the maximum amount of potential parallelism. Task partitioning will later select a desirable subset of the potential parallelism as useful parallelism, but it need not worry about node reordering when doing so. Details of the node reordering algorithm used in POSC are presented in [Sar89b], which also contains experimental results demonstrating that the algorithm works well for real program graphs.

# 6 Task Partitioning

As outlined at the end of Section 3, an IF1 *task partition* in POSC simply identifies the Forall nodes that should be *sliced*, and the function call nodes that should be *forked*. Therefore, a *task* is either a Forall slice or a function call that is forked. The function call nodes that are forked may be user-defined functions or new *task functions* created by the partitioner. In general, any *convex* subgraph of an IF1 graph may be replaced by a special task function. However, in the current implementation of the POSC partitioner, we only consider making a task function out of a single IF1 node (usually a compound node, due to granularity reasons). With this restriction, an implicit *task tree* can be defined by simply specifying a subset of the IF1 nodes that are marked as *task nodes*. A task node, $TN$, uniquely defines a task in the partition, which consists of $TN$ and all nodes contained within $TN$ that do not belong to some other task. The task tree is therefore implicitly defined by the IF1 node hierarchy. In the future, we will extend the POSC partitioner so that it can make a task out of any convex subgraph, as was done in [SH86,Sar89c].

Before describing how a task partition is selected in POSC, we need to describe how a task partition is *evaluated*. An important feature of our work is that we present a single objective cost function that can be used to compare two different task partitions and decide which one is better. This is in contrast to other work ([Cam85,HG85], for example) where the objectives are stated separately as maximizing parallelism and minimizing overhead, without saying how the two should be traded off with each other.

Let $P = \{T_i\}$ be a *task partition* for an IF1 function. The cost of partition $P$ on multiprocessor $M$ is defined to be [SH86,Sar89c]:

$$COST(P,M) = \max\left(\frac{CP}{(SEQTIME/NUMPROCS)}, 1 + \frac{TOTAL\_OVHD}{SEQTIME}\right)$$

where

- $TFREQ(T_i)$ = total execution frequency of task $T_i$, for a single execution of the current function
- $TIME(T_i)$ = sequential execution time (excluding overhead) of task $T_i$
- $SEQTIME = \sum_i TFREQ(T_i) \times TIME(T_i)$ is the total sequential execution time of the current function (excluding overhead)
- $OVHD(T_i)$ = total overhead (task creation, scheduling, synchronization, communication) of task $T_i$
- $TOTAL\_OVHD = \sum_i TFREQ(T_i) \times OVHD(T_i)$, is the total overhead incurred by all tasks in the current function
- $CP$ = estimated parallel execution time of the task partition on an unbounded number of processors assuming that task $T_i$ takes time = $TIME(T_i) + OVHD(T_i)$
- $NUMPROCS$ = number of processors in the target multiprocessor

$COST(P,M)$ nicely expresses the trade-off between parallelism and overhead as a *max* function of the following two terms:

1. The *critical path* term, $CP/(SEQTIME/NUMPROCS)$, which is the estimated critical path length of the partitioned program, normalized to $(SEQTIME/NUMPROCS)$, the "ideal" parallel execution time on $NUMPROCS$ processors.
2. The *overhead* term, $1 + TOTAL\_OVHD/SEQTIME$, which equals 1 plus the estimated total overhead in the program normalized to $SEQTIME$.

For a given multiprocessor, if the granularity of the task partition is too fine, the value of the overhead term will be large owing to excessive overhead. If the granularity is too coarse, the value of the critical path term will be large owing to lost parallelism. $COST(P,M)$ is minimized at an optimal intermediate granularity.

We now outline the partitioning algorithm currently implemented in POSC. Functions in the program are partitioned in a bottom-up traversal of the call graph, so that the main program is the last function to be partitioned. For any function call, the callee's function will be partitioned before the caller's function. For simplicity, we assume that the input program is non-recursive in the current implementation of POSC's partitioner. Partitioning of recursive functions was supported in our earlier work on macro-dataflow [SH86, Sar89c]. In the future, we plan to use a similar approach to extend POSC to handle partitioning of recursive functions.

Currently, the POSC partitioner accepts 6 overhead parameters (details of the corresponding tasking operations were given in Section 3):

1. $T_{fork}$, the time spent by the parent task to *fork* a function call.
2. $T_{slice}$, the time spent by the parent task to fork a single *slice* of the Forall.
3. $T_{startup}$, the time spent in activating a new task from the ready list.
4. $T_{terminate}$, the time taken for a task to *terminate* itself.

5. $T_{suspend}$, the time taken for a task to *suspend* itself during a *join* operation. This overhead component can be performed concurrently with the execution of the child tasks.

6. $T_{restart}$, the time spent in reactivating a suspended task, when its last child complete executions.

These parameters account for all the scheduling and synchronization overhead incurred during program execution.

For each function, the partitioning algorithm attempts to minimize the cost function defined above, $COST(P, M)$. The general structure of the partitioning algorithm is:

1. Start with the finest granularity partition that marks each node as a task function and slices each Forall.

2. Merge all small tasks, $T_i$, for which $CP(T_i) - T_{startup} \leq T_{fork}$. This is a simple optimization that checks if it's always more efficient to execute a task sequentially than to fork it. If $T_i$ is a Forall slice, then we should use $T_{slice}$ instead of $T_{fork}$.

3. Merge all sequential tasks, $T_i$, for which there are no other tasks that can be executed in parallel with $T_i$. This is a simple optimization that checks if $T_i$ is going to execute sequentially anyway. In both steps 2 and 3, task $T_i$ is merged with its parent task in the task tree.

4. Repeat steps 5 and 6 till no further merging is possible; that is, till all nodes have been placed in the same task. Store the best cost function value obtained among all partitions generated during the iterations of steps 5 and 6.

5. Pick the task (say $T_a$) with the largest value of

$$F(T_a) = \frac{TFREQ(T_a) + \sum_{T_c \text{ child of } T_a} TFREQ(T_c)}{1 + (\ \#\ \text{children of } T_a)}$$

   as the first candidate for merging. If $T_a$ is the root task, then the value used for comparison is

$$F(T_a) = \frac{\sum_{T_c \text{ child of } T_a} TFREQ(T_c)}{(\ \#\ \text{children of } T_a)}$$

   Since the reduction in total overhead, when merging a child task into its parent task, is proportional to the total execution frequency of the child task, $F(T_a)$ gives the *average* reduction in the total overhead, if task $T_a$ is merged with its parent or with one of its children.

6. Evaluate the parent and children tasks of $T_a$ as candidates for merging. Of these tasks, pick the one (say $T_b$) that yields the smallest value of $CP$ when merged with $T_a$.

7. Reconstruct the partition with the best cost seen during steps 5 and 6.

8. Merge any remaining task $T_i$ with its parent, if the merge will further reduce the cost function. This is a simple clean-up phase, to locally improve the final partition obtained by steps 4, 5, 6 and 7.

The main issue in the partitioning algorithm is the choice of tasks to be merged in each iteration. In step 3, task $T_a$ with the largest value of $F(T_a)$ is chosen as the primary candidate for merging. This heuristic focuses on the task with the largest average largest reduction in the *overhead term* of the cost function. In step 4, task $T_b$ is chosen as the one that yields the smallest $CP$ value when merged with $T_a$. This heuristic attempts to minimize the *critical path term* of the cost function. Further technical details on the partitioning algorithm are given in [SH86,Sar89c].

| Program | mode | 1 processor | 2 processors | 5 processors | 10 processors |
|---|---|---|---|---|---|
| MATMULT | Fortran | 32.05 | | | |
| | SEQ | 50.70 | | | |
| | PAR | 44.94 | 23.68 | 13.26 | 19.01 |
| | P2, P5, P10 | 36.13 | 18.30 | 7.39 | 3.77 |
| CYK | Fortran | 17.03 | | | |
| | SEQ | 18.76 | | | |
| | PAR | 320.46 | 174.32 | 145.73 | 161.93 |
| | P2, P5, P10 | 20.27 | 10.44 | 4.92 | 3.31 |
| CNTAB | Fortran | 174.90 | | | |
| | SEQ | 168.68 | | | |
| | PAR | 144.00 | 73.44 | 31.00 | 17.27 |
| | P2 | 149.86 | 76.48 | | |
| | P5, P10 | | | 32.02 | 16.95 |
| TRANS | SEQ | 12.37 | | | |
| | PAR | 14.95 | 9.07 | 8.49 | 9.21 |
| | P2 | 11.97 | 7.10 | | |
| | P5 | | | 4.30 | |
| | P10 | | | | 6.27 |

Table 1: Execution times on the Sequent (in seconds)

# 7 Preliminary Experimental Results

In this section, we present the execution times after applying the partitioner to the following four SISAL programs:

1. MATMULT: This program is a standard $O(N^3)$ algorithm for multiplying two $N \times N$ matrices of double precision numbers. The results presented in this paper are for $N = 100$.

2. CYK: This program contains the $O(N^3)$ Cocke-Younger-Kasami parsing algorithm based on dynamic programming [HU79]. The input used consists of the trivial but ambiguous grammar, $\{A \rightarrow AA, A \rightarrow a\}$, and a string of $N$ $a$'s to be parsed. The results presented in this paper are for $N = 100$.

3. CNTAB: This program calculates chi-square, degrees of freedom, and two measures of association for a two-dimensional contingency table of integers [PFTV86]. The results presented in this paper are for a table of size 1000 x 1000.

4. TRANS: This program computes a finite element method solution of linear Boltzman equations, to calculate particle flux through a space. The main calculation is a sequential outer loop containing two FORALL loops, which themselves contain a variety of nested FORALL loops. This is the largest of the four programs, with 568 lines of SISAL code and 25 functions. The IF1 graph for TRANS has 1826 nodes and 3811 edges (610 of the edges are literals).

The execution times presented in this section were gathered on a Sequent Balance[3] 21000 using *gang daemon* software developed at Livermore [ID87]. This software helps reduce interference between parallel and non-parallel jobs.

As controls, we present execution times without partitioning, as well as after forcing all FORALL loops to execute in parallel (without regard to profile data). The following abbreviations are used in the performance tables to describe the various modes of compilations and executions presented:

[3]Sequent Balance is a trademark of Sequent Computer Corporation.

1. SEQ: Compiled for sequential execution (no partitioning).
2. PAR: Compiled so that all FORALL loops execute in parallel (naive partitioning).
3. P2: Partitioned for 2 processor execution using profile data.
4. P5: Partitioned for 5 processor execution using profile data.
5. P10: Partitioned for 10 processor execution using profile data.

We present Fortran times for all programs except TRANS. A Fortran version of this program was not available when we performed the experiments. After some preliminary investigation, we observed a fork-cost and startup-cost of about 500 and 200 microseconds respectively for the Sequent SISAL implementation. These values correspond to 169 and 66 cycles on the Sequent Balance. So, the overhead values used by the partitioner were $T_{fork} = T_{slice} = 169$, and $T_{startup} = 66$. For simplicity, we set $T_{terminate} = T_{suspend} = T_{restart} = 0$ in obtaining the results described in this section.

Table 1 shows the performance data obtained for all four programs. Examining the data, we see that the partitioner yielded parallel execution times that were superior to the PAR approach ("slice all Foralls") for all programs, with the exception of CNTAB where the times were nearly identical. The largest improvement was in CYK, where the execution time of PAR on 10 processors was about 50 times larger than that of P10 on 10 processors. In CNTAB, the 2-processor and 5-processor execution times for both P2 and P5 were slightly larger than for PAR because the P2 and P5 partitions sliced all the Forall loops and created a task function. Therefore, P2 and P5 contain one more task than PAR, and the difference in execution times is due to the overhead of that one extra task. This problem will get fixed as we tune the execution time values for the simple nodes, as well as tune the tasking overhead values so to more accurately represent the parallel execution times of our implementation. However, note that the extra task did pay off on 10 processors, where P10 had a smaller execution time than PAR.

For CNTAB and MATMULT, we see a surprising result where the sequential execution time of P2 is smaller than the sequential execution time of SEQ. Here partitioning resulted in better register allocation in the innermost loops. Also note that execution of P2 on one processor for MATMULT, CYK, and CNTAB resulted in execution times competitive with Fortran. This illustrates OSC's ability to eliminate copying in SISAL programs and generate efficient sequential code.

The results for TRANS show an anomaly, where P10 has a larger execution time on 10 processors than P5 on 5 processors. As far as we can tell, this anomaly occurred because the partitioner did not consider truncation effects in the slicing of Forall loops; that is, it assumed that the iterations could be equally distributed among all processors. TRANS has many loops with small numbers of iterations (2, 4, etc.), and so it is likely to have been seriously affected by this assumption. Truncation effects were taken into account in the macro-dataflow partitioner implemented for SISAL [SH86,Sar89c], and we plan to extend POSC to do the same.

# 8 Related Work

The POSC compiler presented in this paper is an extension of the OSC compiler developed at Colorado State University and Lawrence Livermore National Laboratory [Can89]. POSC uses the same runtime model as OSC, except that the task granularity is now determined by the partitioner. The design of the partitioner is based on previous work on automatic partitioning of SISAL programs [SH86,Sar89c]. Combining these two efforts has made POSC the first compiler system to generate fork-join machine code for executing SISAL programs on shared-memory multiprocessors using global partitioning techniques based on program execution profiles and overhead values.

The general problem of determining the optimal granularity of program decomposition has been addressed in other work. Some partitioning issues for implementing SISAL on a 16-way Transputer-based

message-passing multiprocessor are discussed in [GDLT86]. The *serial combinators* approach for the ALFL language [Gol88,HG85] deals with partitioning program graphs into tasks, as in our compiler. However, serial combinators are not allowed to sacrifice any potential parallelism, leading to a much finer granularity partition than our SISAL tasks. In our partitioner, the central issue is the tradeoff between parallelism and overhead, which allows the partition to be formed at any arbitrary granularity. Further, there are several implementation issues (for example, lazy evaluation) which make reduction languages like ALFL harder to implement efficiently, compared to single-assignment languages like SISAL.

In earlier work [SSM89], we tried to extend OSC's precursor, SC [OC88], so that it would also perform automatic partitioning. In that work, we chose a very simple partitioning strategy based on a *granularity threshold value*, $T_{min}$. The idea was to produce a partition with the largest number of tasks, such that each task had an execution time of at least $T_{min}$. Even that simple approach had a reasonable payoff, compared to the "slice all loops" approach taken by SC. However, that approach did not consider any trade-off between parallelism and overhead, and, in many cases, would produce poorer partitions than the approach described in this paper.

# 9 Conclusions and Future Work

In this paper, we have presented the design of an automatically partitioning compiler that can be used to target the same SISAL program to a range of shared-memory multiprocessors. Such a system greatly simplifies the problems of creating, debugging and porting efficient parallel programs on different multiprocessors. Though the partitioning techniques have been implemented for SISAL, the basic approach is general and is applicable to any environment where a graphical program representation can be obtained.

In the past, one of the biggest challenges in implementing SISAL (or any other single-assignment language) has been to achieve efficient sequential execution times compared to imperative languages such as Fortran, C and Pascal. We feel that this challenge has been largely met, based on the success of recent work on efficient sequential implementation of single-assignment languages [Ran87,GSH88,GH87,Gop89,Can89]. This belief is also validated in the comparisons with sequential Fortran execution times presented in this paper. It now becomes important to turn our attention to efficient parallel implementations. The POSC compiler system is an important step in that direction since it integrates into one system previous work on efficient sequential implementation of SISAL [Can89], along with previous work on selecting the useful parallelism in a SISAL program [SH86,Sar89c]. Further research is now necessary to investigate the performance of various SISAL application programs on different multiprocessors.

There are several minor enhancements that we plan to incorporate into POSC, in the near future. To make execution profiling more efficient and convenient, we will include the profiling optimizations presented in [Sar89a], and also extend POSC so that profiling can be done during a parallel execution of the program. As mentioned at the end of Section 5, we plan to extend the node reordering algorithm so that it approaches a level decomposition when all nodes have equal (or nearly equal) execution times. We also plan to extend the partitioner so that it takes truncation effects into account when slicing loops, and specifies the number of slices (or alternatively, the chunk size) to be used in a Forall that has been chosen for parallel execution. After completing these enhancements, we plan to do several experiments to measure the parallel execution times of various application programs, along the lines of the measurements presented in Section 7.

Some of the major extensions planned for POSC in the future are as follows. First, we would like to make the estimation of execution times more accurate, by performing that phase at the IF2 level where memory operations have been made explicit. Further, the pseudo-edges added in IF2 will also need to be satisfied by the node reordering algorithm. Finally, we will extend the partitioner so that it can handle recursive functions using the techniques introduced in [Sar89c], and can also place more than one node in a task function.

## Acknowledgements

We are grateful for the computer system resources provided by the Computer Science Department at Colorado State University, which were used to obtain the experimental results presented in this paper.

## References

[AHU74] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[Cam85] M. L. Campbell. Static allocation for a dataflow multiprocessor. *Proceedings of the 1985 International Conference on Parallel Processing*, pages 511–517, 1985.

[Can89] D. C. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, Computer Science Department, Fort Collins, CO, 1989.

[CLOS87] David C. Cann, Ching-Cheng Lee, R. R. Oldehoeft, and S. K. Skedzielewski. SISAL multiprocessing support. Technical Report UCID-21115, Lawrence Livermore National Laboratory, Livermore, CA, 1987.

[GDLT86] J. L. Gaudiot, M. Dubois, L. T. Lee, and N. Tohme. The TX16: A highly programmable multimicroprocessor architecture. *IEEE Micro*, 6(10):18–31, October 1986.

[GH87] K. Gopinath and J. L. Hennessy. Copy elimination with abstract interpretation. Technical Report CLaSSiC-87-17, Stanford University, Stanford CA 94305, Feb 1987.

[GKW85] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[Gol88] Benjamin Goldberg. *Multiprocessor Execution of Functional Programs.* PhD thesis, Yale University, New Haven, Connecticut, 1988.

[Gop89] K. Gopinath. *Copy Elimination in Single Assignment Languages.* PhD thesis, Stanford University, Stanford, California, 1989.

[GS87] Thomas Gross and Alan Sussman. Mapping a single-assignment language onto the warp systolic array. *Functional Programming Languages and Computer Architecture Proceedings*, pages 347–363, September 1987.

[GSH88] K. Gharachorloo, V. Sarkar, and J. L. Hennessy. Efficient implementation of single assignment languages. *ACM Conference on Lisp and Functional Programming*, pages 259–268, July 1988.

[HG85] P. Hudak and B. Goldberg. Serial combinators: "optimal" grains of parallelism. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 382–399. Springer-Verlag, New York, NY, September 1985.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[ID87] Eugene D. Brooks III and Gregory A. Darmohray. A parallel programming library and gang scheduler for DYNIX. Technical report, Lawrence Livermore National Laboratory, Livermore, CA, May 1987.

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. *Conference Record of 8th ACM Symposium on Principles of Programming Languages*, 1981.

[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.

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

[PFTV86] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numberical Recipes.* Cambridge University Press, New York, NY, 1986.

[Ran87] John E. Ranelletti. *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages.* PhD thesis, University of California at Davis, Computer Science Department, Davis, California, 1987.

[Ric89] T. R. Richert. Efficient task management for SISAL. Technical Report 89-111, Computer Science Department, Colorado State University, July 1989.

[Sar89a] Vivek Sarkar. Determining average program execution times and their variance. *Proceedings of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 24(7):298–312, July 1989.

[Sar89b] Vivek Sarkar. Instruction reordering for fork-join parallelism. *Submitted to ACM SIGPLAN '90.*, 1989.

[Sar89c] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachusetts, and Pitman Publishing, London, U.K., 1989. An earlier version of this book is available as the author's Ph.D. dissertation, Technical Report CSL-TR-87-328, Stanford University, April 1987.

[SG85] Stephen Skedzielewski and John Glauert. IF1—an intermediate form for applicative languages. Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985.

[SH86] V. Sarkar and J. Hennessy. Partitioning parallel programs for macro-dataflow. In *Proceedings of the ACM Conference on Lisp and functional programming*, pages 202–211, August 1986.

[Sim86] Rea Simpson. SISAL compiler user manual. Manual M-191, Lawrence Livermore National Laboratory, Livermore, CA, September 1986.

[SSM89] V. Sarkar, S. Skedzielewski, and P. Miller. *An Automatically Partitioning Compiler for SISAL.* Cambridge University Press, 1989.

[SW85] S. K. Skedzielewski and M. L. Welcome. Data flow graph optimization in IF1. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 17–34. Springer-Verlag, New York, NY, September 1985.

[SYO87] S. K. Skedzielewski, R. K. Yates, and R. R. Oldehoeft. DI: An interactive debugging interpreter for applicative languages. In *Proceedings of the ACM SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques*, pages 102–109, June 1987.

[WSYR86] M. L. Welcome, S. K. Skedzielewski, R. K. Yates, and J. E. Ranelletti. IF2: an applicative language intermediate form with explicit memory management. Manual M-195, University of California Lawrence Livermore National Laboratory, November 1986.