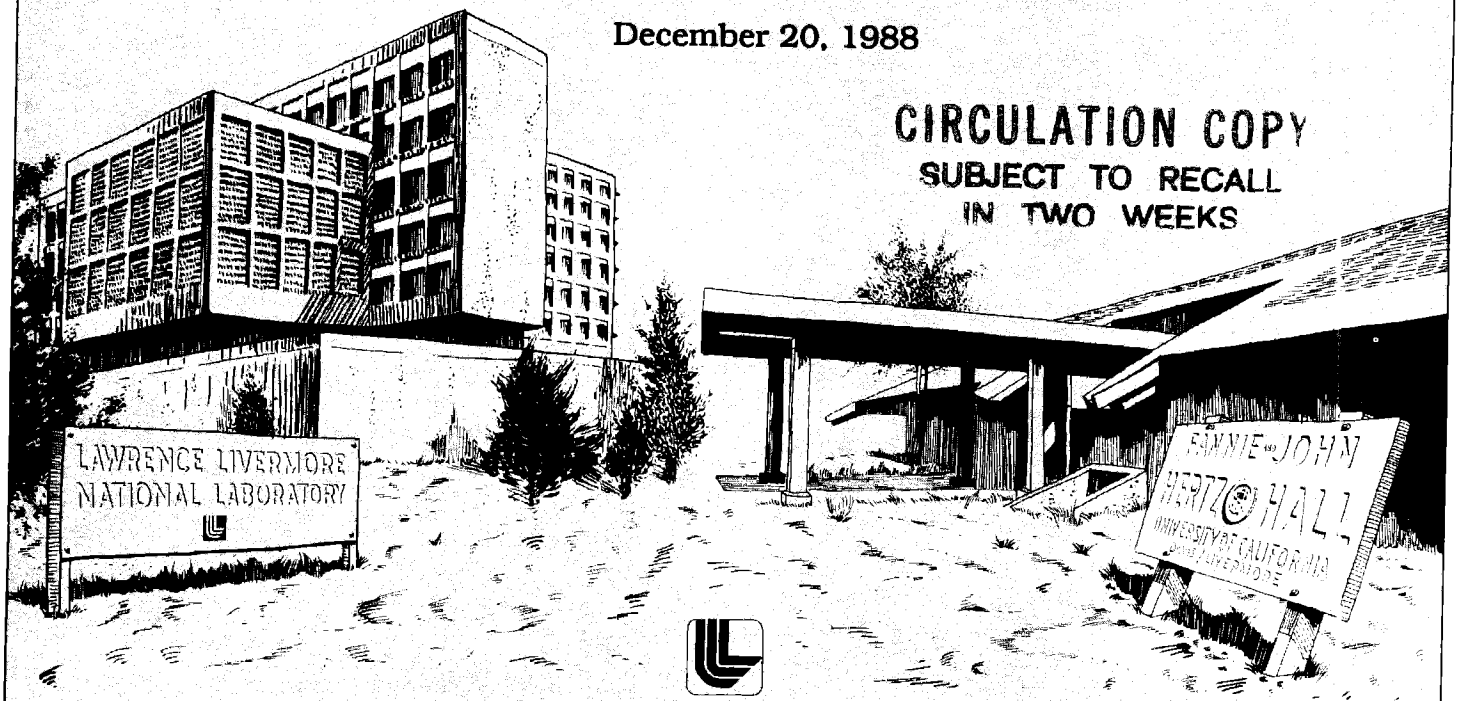
UCRL-98289

An Automatically Partitioning Compiler for Sisal

Vivek Sarkar
Stephen Skedzielewski
Patrick Miller

This paper was prepared for inclusion in
the Conference Proceedings of CONPAR88
Manchester, England

December 20, 1988



Work performed under the auspices of the U.S. Department of Energy by the UCLLNL under contract No. W-7405-ENG-48.

DISCLAIMER

An Automatically Partitioning Compiler for Sisal

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

Stephen Skedzielewski
Computing Research Group, L-306, Lawrence Livermore National Laboratory (LLNL), Livermore, CA 94550.

Patrick Miller
Computing Research Group, L-306, LLNL, Livermore, CA 94550

1. Introduction.

In this paper, we describe a compiler that automatically repackages SISAL [6] programs to achieve the grain size needed to run efficiently on different multiprocessors. The compiler is based on an existing implementation for SISAL on the Sequent Balance multiprocessor [7], and on previous work on automatic partitioning of SISAL programs [9,10]. The granularity of parallelism in the existing Sequent implementation is defined by language constructs (e.g. all function calls are spawned as separate tasks), causing the programming style to dramatically affect multiprocessor performance. It is desirable for the partitioning to be performed automatically, so that the same program can be made to execute efficiently on different multiprocessors and the programmer need not be concerned with granularity issues when developing his program. This is the goal of our automatically partitioning compiler.

The partitioner described in Section 3 operates on IF1 [11], a graphical intermediate language for SISAL programs. It also takes as input a list of parameters describing the target multiprocessor. The partitioner's output consists of IF1 annotations to specify which calls to user-defined functions should be executed as parallel tasks, which Forall expressions should be executed as parallel loops and their appropriate *chunk* sizes.

The Sequent Balance implementation has been retargetted to the Alliant, multi-VAX and Cray X-MP multiprocessor systems. It is currently being modified to support the partitioner's output, rather than just using the default granularity described in [7]. We intend to use the automatically partitioning compiler to study the performance of various SISAL application programs on these different multiprocessors.

2. Runtime Model.

Our runtime model supports three kinds of parallelism in SISAL programs:

1. *Function Calls.* Any function call marked with *Spawn = true* by the partitioner is executed as an independent task. The spawning occurs dynamically whenever the call is reached during program execution.
2. *Loop Slicing.* Any Forall expression can be sliced into independent tasks (much like a function call), since it contains no loop-carried data dependence. The partitioner

selects the Forall expressions to be sliced, and the minimum number of iterations (*chunk* size) that should be executed by a task.

3. *Stream Parallelism.* The stream data type in SISAL sets up a producer/consumer relationship among expressions. The semantics of streams imply sequential access to the stream elements and allow for non-strict evaluation. Thus, a stream is implemented as a shared queue with a fixed buffer size, with the producer adding elements to the tail and the consumers accessing elements from the head.

This model relies on both centralized job queues and global memory. A task can be placed on the *ready* (or *blocked*) queue 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 [1].

3. The Partitioner.

The first step in partitioning is to estimate the execution times of all IF1 nodes and graphs in the program. The average execution times of *simple* nodes provide a basis from which all other execution time estimates are derived. They are initialized according to the performance of simple operations on the given multiprocessor system. Average execution times for *compound* nodes are determined by using average frequency values for Loop and Select subgraphs. In our implementation, these frequency values are obtained from execution profile information. Finally, the average execution time of a function call is assumed to be independent of the call site, and is thus set to the value computed for the corresponding function body. This can be tricky for a recursive call where the execution time of the body depends on the execution time of the call itself. In [10], we show how to assign execution times in the presence of recursive calls.

Given our runtime model, the relevant parameters for partitioning are:

1. The number of processors available.
2. Task creation overhead – time to place a newly created task on the *ready* queue.
3. Task scheduling overhead – time to remove a task from the *ready* queue and initialize it for execution on a processor.
4. Synchronization overhead – time to move the parent task from the *ready* queue to the *blocked* queue and then back to the *ready* queue.

Synchronization overhead occurs when a parent task blocks waiting for a result from a child task. Communication overhead is not relevant to this model because the current compiler targets only shared memory machines.

The simplified partitioning algorithm used in the experiments presented in this paper does not directly use the preceding parameters. Instead, it uses a *threshold* value, T_{min}. The partitioner processes functions in a reverse topological order of the call graph, so that for any non-recursive function call, the callee will always be partitioned before the caller. The idea is to produce the finest partition in which every task has an estimated sequential execution time that is at least T_{min}. Therefore, the parameter T_{min} provides a control on the granularity of the partitioned program. Increasing T_{min} yields a coarser granularity. T_{min} should be chosen so as to dominate the overhead time involved in executing a task on the target multiprocessor.

In future work, we plan to more fully use the multiprocessor parameters listed above in an implementation along the lines of that presented in [9] and [10].

procedure *DetermineTasks*(G)

1. /* First recurse. Note that a depth-first traversal of subgraphs in a compound node ensures that a node is made into a new task "as soon as possible". In this sense, the task partition is at the finest possible granularity for the given T_{min} value. */
   for each compound node N_{comp} in graph G **do**
   (a) **for each** subgraph SG of compound node N_{comp} **do**
      i. **call** *DetermineTasks*(SG)
2. $G.TaskTime := 0$
3. /* Process each node in G. */
   for each node N in graph G **do**
   (a) /* Determine $N.TaskTime$. */
   if N is a compound node **then**
      i. $N.TaskTime := \sum_{SG \in N} SG.freq * SG.TaskTime$
      /* SG is an IF1 subgraph of compound node N. */
   else if N is a function call node **then**
      i. $N.TaskTime := CG.TaskTime$
      /* CG is the IF1 graph of the callee function in node N. */
   else /* N must be a simple node. */
      i. $N.TaskTime := N.SeqTime$
   (b) /* Decide if node N should be a separate task. */
   if ($N.TaskTime \geq T_{min}$) **and** (N is a function call node or a Forall node) **then**
      i. $N.spawn := true$
   else
      i. $N.spawn := false$
      ii. /* $N.TaskTime$ contributes to $G.TaskTime$, since $N.spawn = false$. */
      $G.TaskTime := G.TaskTime + N.TaskTime$
   (c) /* Determine chunk size for a parallel Forall node. */
   if (N is a Forall node) **and** ($N.spawn = true$) **then**
      i. $N.chunksize := \lceil N.freq * T_{min}/N.TaskTime \rceil$

end procedure

Figure 1: Procedure DetermineTasks

The simple approach taken does not directly address the issue of parallelism. Since the algorithm tries to produce the finest granularity partition that satisfies the T_{min} restriction, all the parallelism available at a coarser granularity should continue to be visible in the program. If the program's parallelism only occurred at a finer granularity than T_{min}, then the program is probably unsuitable for execution on the target multiprocessor because it would incur too much overhead.

Of course, the real interaction between parallelism and overhead can be more complex in general. Communication overhead is very sensitive to the partition boundaries and can be more significant than scheduling and synchronization overheads. Further, the parallelism in the program is determined by the execution times of the nodes on the *critical paths* of the program graph. We addressed these issues in a general statement of the partitioning problem presented in [9] and [10]. In this paper, we discuss a simpler approach; but one that has been implemented in a full compiler for real multiprocessor systems.

Procedure DetermineTasks in Figure 1 outlines the algorithm used to select tasks based on T_{min}. DetermineTasks is called for each function's IF1 graph in the program. The procedure operates on an IF1 graph, G, where:

- Each function call node, CN, contained directly or indirectly in G, is assigned a boolean value $CN.spawn$. If $CN.spawn = true$ then the function call is to be spawned as an independent task. Otherwise the function call is to be executed sequentially by the parent task that executes node CN.
- Each Forall node, FN, contained directly or indirectly in G, is assigned a boolean value $FN.spawn$. If $FN.spawn = true$ then the Forall node is sliced into independent tasks. The minimum number of iterations to be executed in one task is given by the value $FN.chunksize$.

Recursive function calls can pose a problem, even in this simple partitioning scheme. What should we do if the execution time of the body of a recursive function is less than T_{min}? One solution would be to set $Spawn = false$ for all the recursive function calls, but that is not interesting because it could serialize the entire program. Our approach is to iteratively perform in-line expansion of recursive calls at compile-time, untill the execution time of the expanded function body exceeds T_{min}. The details of this approach have been omitted, due to space limitations.

4. Experimental Results.

The execution times reported in this section were measured on three different multiprocessor systems:

1. **Sequent** – a Sequent Balance system with 12 NS32032 processors, running the DYNIX operating system. A locally developed *gang daemon* reserves up to 10 processors exclusively for parallel processing jobs. The deamon ensures that interactive jobs are not scheduled on the same processors as parallel processing jobs. Therefore, our measurements are repeatable at any time, with a very small variance.
2. **Alliant** – an Alliant FX/8 computer with 8 Computational Elements. For these experiments we used only six of the eight processors; using more processors led to non-repeatable results due to interference from other time-sharing users.
3. **VAX** – two VAX-11/780 processors connected to 4 megabytes of MA780 shared memory. Our measurements could only be taken when there were no other users on the system.

For the Sequent and VAX systems, we simply report the execution time as *wallclock* time in seconds. The Alliant times reported are the CPU times given in the SISAL runtime statistics.

We present performance results for two SISAL programs. Both programs were partitioned using $T_{min} = 1000$.

4.1. MM – Matrix Multiplication.

A SISAL program for multiplying two 32×32 sized real matrices, using the standard $O(N^3)$ matrix multiplication algorithm.

P	1	2	3	4	5	6	7	8	9	10
T_{orig}	20.0	10.8	7.7	6.1	5.2	4.7	4.3	4.1	4.0	4.0
T_{part}	17.4	8.8	4.6	4.5	4.0	3.5	3.0	2.5	2.5	2.5

Table 1: Execution times of MM on the Sequent ($T_{seq} = 17.4$)

P	1	2	3	4	5	6
T_{orig}	6.2	3.4	2.5	2.0	1.7	1.6
T_{part}	4.9	2.5	1.7	1.3	1.1	1.0

Table 2: Execution times of MM on the Alliant ($T_{seq} = 4.9$)

Table 1 shows the parallel execution time of MM on the Sequent. The sequential execution time is denoted by T_{seq}, T_{orig} denotes the parallel execution time from the original implementation [7], and T_{part} is the time obtained by using the partitioner. We see that the T_{part} numbers are smaller than the T_{orig} numbers. Further, T_{part} shows a relative speedup of $17.4/2.5 = 6.96$ as opposed to T_{orig}'s relative speedup of $20.0/4.0 = 5.0$. Therefore, T_{part} is better than T_{orig}, both in absolute values and relative speedups.

Table 2 shows similar results on the Alliant. The sequential execution time was $T_{seq} = 4.9$. T_{orig} and T_{part} are comparisons of the original Alliant version against the partitioned versions. Once again the T_{part} times are uniformly less than the T_{orig} times, and the relative speedup is better, as well (4.9 vs. 3.9 at six processors).

As a comparison, the sequential execution time on the VAX was $T_{seq} = 13.9$, and the execution times for 1 and 2 processors were:

- $T_{orig} = 16.2, 9.2$, $T_{part} = 14.2, 7.8$

So the observation that T_{part} is better than T_{orig} in absolute values and in relative speedup, holds for the VAX as well.

As discussed in [7], the overhead of maintaining reference counts for arrays can be a significant part of the program execution time. Tables 3 and 4 show the performance obtained by manually simulating a reference count optimization, in which unnecessary reference count operations are removed from the program. We plan to make this optimization automatic in future work. T_{orig} and T_{part} show the same trends in Table 3 as in Table 1. The absolute execution times are better in Tables 3 and 4 because of the reference count optimization. What is even more encouraging is that the relative speedup of T_{part} on the Sequent has now increased to $6.9/0.9 = 7.67$, and on the Alliant to $2.0/0.4 = 5.0$, showing that the reference count optimization also improves the parallelism in the program.

P	1	2	3	4	5	6	7	8	9	10
T_{orig}	9.5	5.4	4.0	3.4	3.0	2.9	2.8	2.8	2.9	3.1
T_{part}	6.9	3.5	2.4	1.8	1.6	1.4	1.2	0.9	0.9	0.9

Table 3: MM on the Sequent with reference counts removed ($T_{seq} = 6.9$)

P	1	2	3	4	5	6
T_{orig}	3.4	2.0	1.5	1.3	1.2	1.1
T_{part}	2.0	1.0	0.7	0.5	0.5	0.4

Table 4: MM on the Alliant with reference counts removed ($T_{seq} = 2.0$)

For the reference count optimization case on the VAX, we measured $T_{seq} = 5.0$, and the execution times for 1 and 2 processors were:

- $T_{orig} = 7.3, 4.2$, $T_{part} = 5.3, 2.7$

4.2. TRANS – Particle Transport.

The TRANS 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 parallel loops.

P	1	2	3	4	5	6	7	8	9	10
T_{orig}	57.2	30.0	21.5	17.4	15.5	14.5	13.9	13.7	13.8	13.6
T_{part}	20.8	10.9	8.8	7.4	7.5	7.5	7.6	7.5	7.6	7.5

Table 5: Execution times of TRANS on the Sequent ($T_{seq} = 20.5$)

Tables 5 and 6 show the parallel execution time of TRANS on the Sequent and the Alliant. These numbers indicate a similar trend as was seen for MM. However, TRANS has less parallelism than MM, leading to a relative speedup of $20.8/7.5 = 2.77$ in T_{part} on the Sequent and $5.9/3.4 = 1.7$ in T_{part} on the Alliant.

As a comparison, on the VAX $T_{seq} = 18.5$ and the execution times for 1 and 2 processors were:

- $T_{orig} = 48.5, 26.7$, $T_{part} = 19.0, 10.2$

As in MM, Table 7 gives the execution times with the reference counts removed. The gain due to partitioning is much more pronounced for TRANS, where we see $T_{orig} = 10.6$ and $T_{part} = 3.2$, for 10 processors on the Sequent and $T_{orig} = 4.2$ and $T_{part} = 1.6$ for six processors on the Alliant.

For the reference count optimization case on the VAX, we measured $T_{seq} = 9.1$, and the execution times for 1 and 2 processors were:

- $T_{orig} = 36.9, 20.2$, $T_{part} = 8.2, 4.1$

The VAX timings contain an anomaly, where the single processor time for T_{part} (= 8.2) is noticeably better than $T_{seq} = 9.1$. This is due to the fact that the VAX timings are sensitive to the presence of other jobs, unlike the Sequent timings. We tried to minimize this factor by only making the measurements when there were no other users on the system. This apparent anomaly is due to the fact that we report *wallclock* times.

P	1	2	3	4	5	6
T_{orig}	24.3	12.3	8.7	6.9	5.9	5.3
T_{part}	5.9	3.3	3.3	3.3	3.3	3.4

Table 6: Execution times of TRANS on the Alliant ($T_{seq} = 5.8$)

P	1	2	3	4	5	6	7	8	9	10
T_{orig}	44.9	23.5	16.7	13.3	11.7	11.1	10.8	10.6	10.5	10.6
T_{part}	9.2	4.8	3.8	3.1	3.1	3.1	3.1	3.2	3.1	3.2

Table 7: TRANS on the Sequent with reference counts removed ($T_{seq} = 9.2$)

The *CPU* times corresponding to T_{seq} and to T_{part} on one processor were 6.97 and 6.98 respectively, showing that T_{seq} did not really use more CPU time than T_{part}.

5. Related Work.

Our compiler is an extension of the compiler developed at Colorado State University (in collaboration with LLNL) for implementing SISAL programs on the Sequent Balance multiprocessor [7]. We use the same runtime model, 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 [9,10].

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 [2]. The *serial combinators* approach for the ALFL language [3,5] 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 (e.g. lazy evaluation, copy avoidance) which make reduction languages like ALFL harder to implement efficiently, compared to single assignment languages like SISAL.

6. Conclusions.

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 that uses a graphical program representation.

Further research is necessary to investigate the performance of various SISAL applica-

P	1	2	3	4	5	6
T_{orig}	21.0	10.6	7.3	5.7	4.8	4.2
T_{part}	2.8	1.5	1.5	1.5	1.6	1.6

Table 8: TRANS on the Alliant with reference counts removed ($T_{seq} = 2.8$)

tion programs on different multiprocessors. One of the biggest challenges in implementing SISAL (or any other functional language) is to not only obtain good speedup in a parallel implementation, but also achieve efficient sequential execution times compared to imperative languages like Fortran, C or Pascal. Our hope is to meet this challenge by combining the automatically partitioning compiler described in this paper with current research in optimizing single assignment languages [8] [4].

7. Acknowledgements.

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.

References.

[1] 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.

[2] 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.

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

[4] K. Gopinath and J. L. Hennessy. *Copy Elimination with Abstract Interpretation.* Technical Report CLaSSiC-87-17, Stanford University, Stanford CA 94305, Feb 1987.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* PhD thesis, Stanford University, Stanford, California, April 1987.

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