`fsc`: A Sisal Compiler for Both Distributed- and Shared-Memory Machines

Vincent W. Freeh
Gregory R. Andrews

TR 95-01

`fsc`: A Sisal Compiler for Both Distributed- and Shared-Memory Machines

Vincent W. Freeh and Gregory R. Andrews

TR 95-01

Abstract

This paper describes a prototype SISAL compiler that supports distributed- as well as shared-memory machines. The compiler, `fsc`, modifies the code-generation phase of the optimizing SISAL compiler, `osc`, to use the Filaments library as a run-time system. Filaments efficiently supports fine-grain parallelism and a shared-memory programming model. Using fine-grain threads makes it possible to implement recursive as well as loop parallelism; it also facilitates dynamic load balancing. Using a distributed implementation of shared memory (a DSM) simplifies the compiler by obviating the need for explicit message passing.

February 21, 1995

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]First published in the "High-Performance Functional Computing Conference", April 1995.
[2]This work was supported by NSF grants CCR-9108412 and CDA-8822652.

1 Introduction

It is difficult to create a correct and efficient parallel program; this difficulty is compounded because each architecture may require a different program. It would be easier if programmers could write one straightforward program that is portable across the variety of sequential, vector, and parallel processors. However, the end goal of parallel programming is performance, so the program also has to execute efficiently.

Functional languages have the potential to provide a simple, portable, and efficient solution to parallel programming. Using a functional language often reduces the programmer's effort by making programs simpler (no explicit concurrency or memory management), easier to maintain (smaller programs), easier to test (deterministic results), and easier to understand (equals means equals). In addition, functional programs have an abundance of implicit concurrency, because loop iterations and independent expressions can be evaluated in parallel. The main challenges are to implement functional programs both efficiently and portably on a variety of machines.

SISAL is a functional language [M+85] that has proven useful for programming numerically intensive scientific applications, especially parallel applications [Can92a]. The optimizing SISAL compiler, `osc`, creates code for many different shared-memory and vector processors (e.g., SGI, Sequent, Encore Multimax, Cray X-MP, and Cray 2) [Can92b]. It does not, however, create code for distributed-memory machines.

This paper describes a prototype compiler, `fsc`, that supports distributed-memory multiprocessors as well as other machines. It does so by modifying the back end of `osc` to use the Filaments package, a subroutine library that provides efficient fine-grain parallelism and a shared-memory programming model using only standard hardware and software [EAL93, FLA94]. Additionally, the prototype supports recursive parallelism, which is not supported by `osc` even though it is implicitly available in SISAL itself. The prototype was produced in about three months of work by a single programmer. Still, the code produced by `fsc` already achieves good speedup on both shared- and distributed-memory machines for most of the applications tested.

The next section gives an overview of relevant aspects of the Filaments package. Section 3 discusses the SISAL programming language, the `osc` compiler, and the modifications made to `osc` to create the prototype `fsc` compiler. Section 4 discusses the performance of three programs, Section 5 mentions related work, and Section 6 gives concluding remarks.

2 Filaments Runtime System

The Filaments package is a library of C code that currently runs on clusters of workstations (Suns running SunOS or Solaris, and DEC-5000s running Mach), shared-memory multiprocessors (Sequent Symmetry and SGI Iris), and a distributed-memory multi-computer (Intel Paragon). It has been designed to support parallel scientific applications and is intended to be a target for a compiler. The package is relatively small (less than 7,000 lines of C code), simple (about 20 subroutine calls), and efficient [EAL93, FLA94].

Filaments has two key abstractions: fine-grain threads and shared memory. The fine-grain execution model can be thought of as the least common denominator for concurrency, because it supports coarse- and medium-grain tasks as well as fine-grain ones. The shared-memory programming model provides a simple and natural way to program.

A *filament* is a very lightweight thread. Each filament is an independent unit of work, and there may be thousands in an application. There are two kinds of filaments: *iterative*, which are used in loop-based applications such as matrix multiplication and Jacobi iteration, and *fork/join*, which are used in recursive applications such as adaptive quadrature and quicksort. Once created, filaments are executed concurrently by server threads.

Iterative filaments synchronize using *barriers*, and fork/join filaments use the *join* primitive. The package also provides reduction operations, which are integrated with barrier synchronization. A distributed shared memory (DSM) system provides a logically shared memory on distributed-memory machines. Hence, filaments, whether executing on a shared-memory (SM) or distributed-memory (DM) machine, can communicate by reading and writing shared variables. No explicit message passing is required on DM machines.

The Filaments package employs a number of techniques to help make it efficient. First, filaments are stackless and independent. Second, the DSM is multi-threaded to overlap communication and computation, thus eliminating communication latency in most applications. Third, the package employs techniques such as inlining and pruning to reduce the overhead of executing filaments. Finally, a reliable datagram protocol is used to reduce buffering and copying of message data (DSM pages) and to provide scalability. These features are discussed below and discussed in detail in [EAL93] and [FLA94].

A filament does not have a private stack; it consists only of a code pointer and state (usually just the arguments). Consequently, context-switching is inexpensive, filaments can be created quickly, and very little space is used. On each processor a server thread executes filaments, one at a time. A filament executes in the server's stack, just as a procedure call does. An executing filament is not pre-empted: It can block only at a barrier or reduction (waiting for all other filaments to arrive) or a join (waiting for its children to finish). These two restrictions on filaments (independent and non pre-emptable) allow for the efficient creation and execution of thousands of filaments, yet provide the flexibility to support parallel scientific programming.

On a DM machine, a remote page fault takes a long time, so the Filaments package overlaps communication and computation. This is accomplished by using multiple server threads and asynchronous communication. When a filament accesses a remote page, a page fault occurs. A request is made for the remote page, the executing server thread is suspended, and another server thread (executing another filament) is started. When the requested page arrives, the suspended server—and any others that are waiting for the same page—is moved to the ready list and eventually resumes executing the filament that faulted. A server thread is not pre-empted when a remote page is received. It is momentarily suspended while the incoming message is read; then it is resumed.

Even though filaments can be executed quite efficiently, there are many techniques employed by Filaments to further reduce the overhead. The body of a filament can be inlined to eliminate procedure call overhead. Filaments are usually created in loops, in which case the arguments to the filaments form a regular pattern. At runtime—immediately before execution—analysis can be done on the argument list to determine if the arguments can be generated rather than read from memory. When this optimization is possible, it not only reduces the number of instructions but eliminates many loads—increasing the effectiveness of the cache. In fork/join applications there are generally thousands of filaments created. Forking filaments is expensive, so once "enough" filaments are forked, the package starts *pruning*. Instead of forking, a new filament is executed immediately as if it were a procedure call. There is significantly less overhead in a procedure call and return than in a fork and join, and the vast majority of fork/join filaments are pruned.

There are two choices for communication in Unix: TCP and UDP. TCP is costly and does not scale; UDP is not reliable. We created a new protocol (built on top of UDP) that provides reliable, scalable, low overhead datagram communication. This protocol uses scatter/gather to avoid extra copying of page data, and it has semantics such that page data is never buffered. The Filaments protocol eliminates at least three copies relative to TCP—two on sending and one on receiving—because TCP buffers all messages.

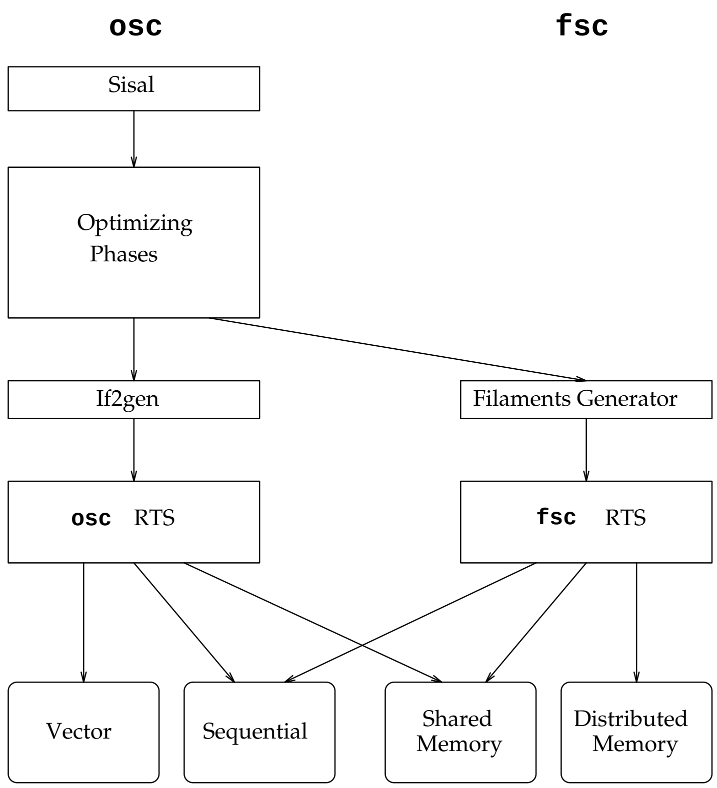


Figure 1: Comparison of `osc` and `fsc` components

3 Details of the `fsc` Compiler

SISAL is a general purpose functional language [FCO90]. It is probably best described as a *dataflow* language because the order of program execution is determined not by the static ordering of expressions in the source code but rather by the availability of the data. Because an expression can be evaluated as soon as all its operands have been determined, the compiler can schedule expression evaluation in any order—including concurrently—that preserves the data dependencies.

Version 12 of the optimizing SISAL compiler (`osc`) creates efficient code for execution on various sequential, vector, and shared-memory machines [Can92b]. (This compiler does not support distributed memory, although the majority of the new supercomputers have distributed memory architectures.) The `osc` compiler translates SISAL source into C or Fortran code, which is then linked with the `osc` runtime system. Many optimizations are performed by `osc` including build-in-place and update-in-place analysis of arrays to reduce copying. The compiler also partitions the problem into subproblems that can be executed in parallel. The `osc` execution model consists of one shared queue containing slices of work and a server thread executing these slices on each processor. It creates at least p slices, where p is the number of processors. The load imbalance is statically calculated by using a cost estimate of the work. Well-balanced programs use exactly p slices; the greater the load imbalance, the greater the number of slices created.

We have created a prototype compiler from `osc` that uses the Filaments runtime system (RTS) instead of the `osc` RTS. The components of the two compilers are shown in Figure 1. The prototype SISAL-to-Filaments compiler (`fsc`) is different from `osc` because it has a fine-grain execution model, supports recursive parallelism, and runs on distributed-memory machines. The prototype was created directly from `osc` both to use all the features of `osc` and to minimize the amount of work required. The majority of the optimizations provided by `osc` are part of `fsc` since the code generator is the only component unique to `fsc`.

In creating `fsc` the following modifications were made to `osc`:

- changed from coarse-grain to fine-grain execution,
- added distributed initialization, and
- added distributed management of shared memory.

The first modification was to support the fine-grain execution model of Filaments. The other two are needed to support execution on distributed-memory machines.

To exploit the fine-grain execution model, `fsc` creates as many filaments as possible, rather than creating some number of slices based on a static evaluation of the workload. Therefore, the number of filaments is based on the problem size, not on the underlying architecture. The code generation of parallel `for` nodes was modified to create filaments and then execute them. To do this the body of the `for` node is made into a filament. For example, in matrix multiplication `fsc` creates n^2 filaments in a two-dimensional loop and starts execution after all have been created. Each filament computes a dot product and writes an element into the result matrix.

There are multiple processes and address spaces on a distributed-memory machine. Therefore, `fsc` creates an address space, an initialization thread, and one or more server threads on each processor. Many of the `osc` initialization files were modified to perform distributed initialization, which includes starting a process in all address spaces, establishing communication between these processes, and creating separate initial work for each processor. In constrast on a shared memory (SM) machine, both the `osc` RTS and Filaments RTS use a single address space containing an initial thread and p server threads (one per processor).

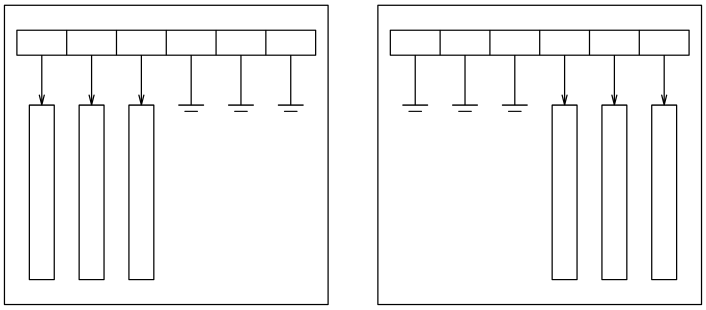
The shared memory allocation and deallocation routines of `osc` had to be modified to use the Filaments DSM. In `osc` shared memory is dynamically allocated (and deallocated). In `fsc` the shared memory is divided into p partitions, one for each processor, and each processor allocates shared memory from its assigned partition. On deallocation, memory is returned to the same partition from which it was allocated. To avoid excessive faulting, objects are placed on pages owned by the processor on which they are created and new objects are placed on new pages. This is accomplished with a "begin object" directive which causes the next memory allocation to begin on an unused page. For an object that requires multiple allocations, subsequent allocations will come from the same page or an unused page.

In `osc` objects are created lazily; `fsc` retains this though a multi-phase allocation scheme. For a scalar object, first the shared memory is allocated, then the object is created and initialized, and finally the value of that object is disseminated to all other processors via a reduction.

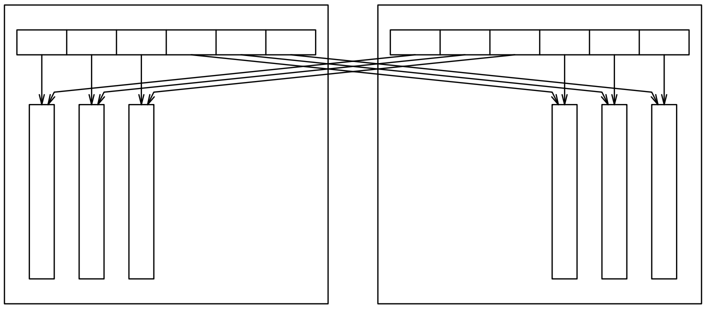
For an aggregate object that is created by all processors, first each processor allocates, creates, and initializes its portion of the object, and then a reduction joins the portions together. For example, the following SISAL code creates an $N \times N$ matrix called `A`, whose elements are $A_{i,j} = f(i,j)$:

```
A := for i in 1, N cross j in 1, N
  returns array of f(i,j)
  end for
```

In a block partition, the first N/p rows of `A` are created on the first processor, the second N/p rows on the second processor, and so on. Each processor creates a top-level array and an array



(a)



(b)

Figure 2: Allocation of 2D array in DSM, before reduction (a) and after reduction (b)

for each of the N/p rows (see Figure 2(a)). After all the rows have been created, a reduction is performed on the top-level array—this reduction corresponds to the `returns` clause. This is shown in Figure 2(b). Each processor has a copy of the top-level array—but the rows are shared.[1]

Cloning appears to be safe because of the single-assignment semantics of SISAL. But `osc` has many memory optimizations that result in arrays being updated. In this case, further reductions are needed to regain a consistent state of memory. Full implementation of updating as supported by `osc` requires locks and reference counts. A distributed protocol to implement locking and reference counting is non-trivial and beyond the scope of the work reported in this paper.

Using a fine-grain execution model has several advantages. First, it more closely represents the abstract model of the SISAL language itself. Second, the partitioning phase of `osc`, which represents a significant amount of the code, may not be needed in a compiler that uses a fine-grain RTS. Static partitioning is problematic in the presence of indefinite loops and conditional statements and may require tuning through compilation and runtime parameters. Lastly, having many small tasks also makes it easier to perform load balancing and data partitioning, allowing dynamic and adaptive systems to perform both (for example, see the Adapt system [LA94]).

Obviously, creating `fsc` from `osc` has some advantages but there is a downside, too. In particular, there is some cost in translating from the fine-grain model of SISAL to the coarse-grain model

[1]Every processor reads the top-level array; the rows, however, are often read by only one processor. Cloning the top-level array reduces page traffic and only slightly increases the overall storage. If the rows were cloned, storage usage would increase greatly and it would be much more difficult to maintain consistent memory.

Machine/Compiler	Processors		
Shared Memory	1	2	4
`osc`	72.0	37.8	21.6
`fsc`	73.8	40.6	25.6
Distributed Memory	1	2	4
`osc`	85.5	—	—
`fsc`	87.2	46.1	29.8
C & library	84.0	45.7	25.7

Figure 3: Matrix multiplication, Times in seconds

of `osc` then back to the fine-model of `fsc`. Because of the structure of `osc`, we were unable to reuse filaments in Jacobi iteration.[2] Ideally, filaments are created once and executed many times: The number of filaments and the task of each filament do not change between iterations—only the data in the array are changed. In a hand-coded Filaments program, filaments are reused in this manner.

4 Performance

Three applications were tested. Each was compiled using both the `osc` and `fsc` compilers, and each was run on a shared- and a distributed-memory machine. The shared-memory (SM) tests were conducted on a Silicon Graphics Iris 4D/340 shared-memory multiprocessor, having four 33-Mhz MIPS processors. The distributed-memory (DM) tests were conducted on a cluster of four Sun Sparc-1 workstations connected with a 10Mbs Ethernet. The Sisal 1.2 compiler, `osc v12.9.2`, was used to compile the Sisal programs; and it formed basis for the `fsc` compiler. We compare DM `fsc` performance to hand-coded C programs that use the Filaments communication library, because `osc` does not have a distributed implementation.

4.1 Matrix Multiplication

Our matrix multiplication program solves $C = A \times B$, where A, B, and C are square matrices. For each point in C, we compute the inner product

$$c_{i,j} = \sum_{k=1}^{N} a_{i,k} \times b_{k,j}$$

The performance of the matrix multiplication programs is is shown in Figure 3. In order to have the tests run in about the same overall time, the SM tests used a 512×512 matrix and the DM tests used a 360×360 matrix.

Shared-Memory Tests

The `fsc` program is slightly slower (2.5%) on one machine than the `osc` program. This is due to the cost of creating and executing filaments. Both compilers achieve good speedup on the SM

[2] The `osc` code-generation phase partitions a `for` node and creates the code for the body and the `returns` clause in three distinct places. This division is quite different from how the Filaments package is used. Therefore, we could not create reusable filaments in `fsc` in the time we spent on this project.

Machine/Compiler	Processors		
Shared Memory	1	2	4
`osc`	120	68.8	50.6
`fsc`	127	75.4	62.4
Distributed Memory	1	2	4
`osc`	51.6	—	—
`fsc`	69.6	51.4	45.1
C & library	38.6	21.5	15.5

Figure 4: Jacobi Iteration, Times in seconds

machine, although the speedup of `osc` is better than that of `fsc`. The performance of `fsc` relative to `osc` decreases as the number of processors increases. Filaments are created sequentially by one processor, so creation time is the same for all tests. Additionally, the filaments themselves are in the cache for (at best) only one processor, resulting in many cache misses on the other processors when they go to execute the filaments.

Distributed-Memory Tests

On a single processor, the C program was fastest, followed by the `osc` program. This is expected because the C and `osc` programs are very similar, but the `osc` runtime has a little bit of overhead, accounting for the 2% increase. The `fsc` program is about 2% slower than the `osc` program due to the cost of creating and executing the nearly 130,000 filaments. Both the C and `fsc` programs scale very well. Matrix multiplication is easy to parallelize, so this is not surprising.

4.2 Jacobi Iteration

In this section we discuss programs that solve Laplace's equation using Jacobi iteration. Laplace's equation in two dimensions with constant, uniform boundaries is

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Jacobi iteration uses the following approximation of the difference equation:

$$u_{i,j}^{(k+1)} = \frac{1}{4}(u_{i-1,j}^{(k)} + u_{i+1,j}^{(k)} + u_{i,j-1}^{(k)} + u_{i,j+1}^{(k)})$$

The equation above computes the $(k+1)$th approximation from the kth approximations. The algorithm terminates when the maximum change in any point is less than some convergence tolerance. The performance of the Jacobi iteration programs is shown in Figure 4. The tests use a 100×100 matrix. Because the SM machine is faster, the SM tests use a smaller convergence tolerance than the DM tests (10^{-5} versus 10^{-3}).

Shared-Memory Tests

The performance of the `fsc` program on the SM machines is not as good as the `osc` program. This is primarily because filaments are not reused. Instead of creating 10,000 filaments once, 10,000 are created on each of the 360 iterations. Additionally, when a set of filaments is created once and repeatedly executed, the Filaments package can perform some additional optimizations that greatly reduce the cost of executing filaments.

Machine/Compiler	Processors		
Shared Memory	1	2	4
`osc`	80.9	—	—
`fsc`	82.7	46.1	22.9
Distributed Memory	1	2	4
`osc`	74.0	—	—
`fsc`	84.6	37.5	29.3

Figure 5: Adaptive Quadrature, Times in seconds

Distributed-Memory Tests

On a single node the C program is significantly faster than the other programs. This is mostly due to memory allocation. The other two programs allocate a matrix in each of the 360 iterations—because each row is allocated separately, there are at least 101 allocations and deallocations each iteration. The `fsc` program is much slower (35%) than the `osc` program because filaments are not being reused.

The C program gets very good speedup, whereas the `fsc` gets very little. The good speedup of the C program is because it uses explicit asynchronous message passing to overlap communication and computation.[3] The poor speedup of `fsc` is due to the same lost optimizations incurred by the SM program, plus an extra reduction and paging costs. The `returns` clause in Sisal roughly corresponds to a reduction in `fsc`. Often a `for` loop will have more than one `returns` clause. This results in multiple reductions in `fsc`. Since reductions are relatively expensive operations, it is important to minimize them. The `fsc` Jacobi iteration program has two reductions per iteration, whereas the hand-coded C program has only one. The cost of the extra barrier grows with the number of processors: It is insignificant with one processor, but accounts for about 4 seconds of the total time of the 4 processor test. The C program explicitly transmits one row (800 bytes) of data to each neighbor using just one message per neighbor. The `fsc` program implicitly transmits one page (4096 bytes) of data to each neighbor using two messages per neighbor. This is not an inherent limitation of the approach, because a hand-coded Filaments program performs within 10% of the C program [FLA94].

4.3 Adaptive Quadrature

Adaptive quadrature approximates the integral

$$\int_a^b f(x)dx$$

by dividing the interval into subintervals. The area of each subinterval is approximated (using a method like the trapezoidal rule), then the approximations are added together to obtain the approximation for the area of the entire interval. The subintervals are determined dynamically: A recursive function estimates the area of an interval, and the areas of the right and left halves of the interval. If the difference between the sum of the two smaller areas and larger area is small enough, the approximation for the area is accepted and the function returns the area of the

[3] First, data is sent to neighboring processors. Then the interior points, which can be updated with local data, are updated. Lastly, the boundary points are updated after receiving data from the neighbors. If the data arrive while the interior points are being updated, full overlap of communication and computation is achieved.

interval. Otherwise the function will recursively (and in parallel) approximate the areas of the left and right subintervals and return the sum. Adaptive quadrature was chosen because it is an irregular problem: the amount of work is not known at compile time. The results of the adaptive quadrature tests are shown in Figure 5.

Shared-Memory Tests

The `osc` compiler has not implemented recursive parallelism, so there is only one performance number for `osc`. On the SM machine, the overhead of `fsc` is small (2%) and the speedup is good (1.8 and 3.6 on two and four processors, respectively). Pruning and load-balancing provided by Filaments package are the reasons for the good performance.

Distributed-Memory Tests

The single processor performance of `fsc` is worse than that of `osc`. On SM `fsc` is 2% slower than `osc`, but on DM it is 14% slower. We cannot account for the large difference. The speedup of the DM `fsc` program is good: 2.3 and 2.9 on 2 and 4 processors. We suspect the super-linear speedup shown on the two-processor test is due to the slowness of the one-processor performance. Compared to the `osc` time, the speedup is not super-linear.

5 Related Work

Two other systems have compiled SISAL for execution on distributed-memory machines. Distributed Memory SISAL from Colorado State University [HB92] is closely related to `fsc`. It uses a DSM, but employs coarse-grain execution. Additionally, their DSM does not achieve as much overlap of communication and computation as `fsc`, because of the lack of many fine-grain tasks and the use of a stack to hold faulted threads.

The distributed-memory SISAL compiler from North Carolina State University uses a coarse-grain, explicit message passing model [PAM94]. Data are transferred between processors using explicit communication operations. Significant analysis is required both to insert and minimize the communication between processors.

Two systems for executing ALFL programs on shared- and distributed-memory machines are described in [Gol88]. An ALFL program and its data are represented as a graph. In these systems a graph reduction "engine" exhaustively applies "reductions" to the nodes in the graph. The shared-memory system has nearly linear speedup; the distributed-memory system gets very poor speedup. Because both systems use graph reduction, they have very large overhead and are not competitive with imperative programs.

The paper [Nik89] shows how Id can be translated to dataflow graphs, then to P-RISC (Parallel-RISC) code, and finally to machine code. The target machine can be a shared- or a distributed-memory machine. The main similarity of this to `fsc` is the architecture-independent intermediate form (P-RISC). Like `fsc`, the P-RISC model uses fine-grain execution. However, it uses a distributed-memory model and hence explicit communication; also, it has not been implemented.

There is a lot of work regarding fine-grain parallelism and distributed shared memory systems that is related to the Filaments package. See [EAL93, FLA94] for details.

6 Conclusions

The `fsc` prototype was created in a very short time (approximately three person-months). Unlike `osc`, it executes on distributed-memory machines as well as shared-memory machines. The Filaments package provides this architecture-independence. Additionally, `fsc` supports recursive parallelism, allowing parallel execution of recursive programs, such as adaptive quadrature and quicksort. The `fsc` compiler should also be easier to use than `osc` because the programmer does not have to tune any compilation or runtime parameters.

Performance of many of the test programs is very good. However, `fsc` does not perform as well as expected, particularly in the Jacobi iteration tests. Previously, we have demonstrated that a hand-coded Filaments program can execute these applications efficiently on a shared-memory multiprocessor [EAL93] and a cluster of workstations [FLA94]. Therefore, we believe poor performance is not an inherent limitation of the approach.

Acknowledgements

Dave Lowenthal gave much advice and encouragement during the project. Then he spent many hours reviewing the paper and provided many invaluable comments. The reviewers of the abstract also provided many good comments that were used to improve the paper.

References

[Can92a] David Cann. Retire Fortran? A debate rekindled. *CACM*, 35(8):81–89, August 1992.

[Can92b] David C. Cann. The optimizing SISAL compiler. Report UCRL-MA-110080, Lawrence Livermore National Laboratory, April 1992.

[EAL93] Dawson R. Engler, Gregory R. Andrews, and David K. Lowenthal. Shared Filaments: Efficient support for fine-grain parallelism on shared-memory multiprocessors. TR 93-13, Dept. of Computer Science, University of Arizona, April 1993.

[FCO90] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the SISAL language project. *J. of Par. and Dist. Computing*, 10(4):349–366, December 1990.

[FLA94] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *First Symposium on Operating Systems Design and Implementation*, pages 201–213, November 1994.

[Gol88] Benjamin Goldberg. Multiprocessor execution of functional programs. *International J. of Parallel Programming*, 17(5):425–473, October 1988.

[HB92] Matthew Haines and Wim Bohm. Software multithreading in a conventional distributed memory multiprocessor. Technical Report CS-92-126, Colorado State University, September 1992.

[LA94] David K. Lowenthal and Gregory R. Andrews. Adaptive data placement for distributed-memory machines. TR 94-33, University of Arizona, December 1994.

[M+85] James R. McGraw et al. SISAL language reference manual version 1.2. Technical Report M-146, Lawrence Livermore National Laboratory, March 1985.

[Nik89] Rishiyur S. Nikhil. The parallel programming language Id and its compilation for parallel machines. In *Proc. Workshop on Massive Parallelism: Hardware, Programming nad Applications*, October 1989.

[PAM94] Santosh S. Pande, Dharma P. Agrawal, and Jon Mauney. Compiling functional parallelism on distrbuted-memory systems. *IEEE Parallel and Distributed Technology*, pages 64–76, Spring 1994.