A Comparison of Implicit and Explicit Parallel Programming

Vincent W. Freeh

TR 93-30a

A Comparison of Implicit and Explicit Parallel Programming

Vincent W. Freeh

TR 93-30b

Abstract

The impact of the parallel programming model on scientific computing is examined. A comparison is made between SISAL, a functional language with implicit parallelism, and SR, an imperative language with explicit parallelism. Both languages are modern, high-level, concurrent programming languages. Five different scientific applications were programmed in each language, and evaluated for programmability and performance. The performance of these two concurrent languages on a shared-memory multiprocessor is compared to each other and to programs written in C with parallelism provided by library calls.

August 1, 1994

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

[1] A version of this paper will appear in the *Journal of Parallel and Distributed Computing.*
[2] This research was supported by NSF grants CCR-9108412 and CDA-8822652

1 Introduction

With most scientific computing applications, tension exists between the size of the problem (or the precision of the solution) and the amount of computational time required. This forces evaluating a smaller problem or calculating a less precise solution in order to obtain timely results. The extra processing power of parallel computers relieves some of this tension, and today parallel computers are common and relatively cheap. Thus, we can expect more and more scientists to turn to parallel computing.

There are three main approaches to programming parallel computers: parallelizing compilers, implicit parallel programming languages, and explicit parallel programming languages [Cof90]. A parallelizing compiler creates a parallel program from sequential source code. Thus, existing sequential programs become parallel programs, and the programmer need not learn a new language. Unfortunately, this approach has two major drawbacks. First, the compiler usually cannot discover all the available parallelism in a program. Second, the best parallel solution to a problem often differs from the best sequential solution [Cof90].

An implicit parallel programming language, such as Id or SISAL, relies on a compiler to exploit the parallelism inherent in a program. These languages have no constructs to create and manage parallelism, and hence, program design and development is generally simpler than in an explicit parallel language. However, like a parallelizing compiler for a sequential language, a compiler for an implicit parallel language has to create and manage parallelism. Thus, like in the above approach, the compiler may not discover all the parallelism available [Cof90].

An explicit parallel programming language provides constructs, such as **cobegin**, that allow the programmer to create and manage concurrency. Modern concurrent languages, such as Ada and SR, have parallel constructs integrated into the language. Others take an existing sequential language, such as C or Pascal, and add library routines to manage concurrency. Because parallelism is explicit, the programmer can write an efficient program and tune it for peak performance. However, the need to code parallelism explicitly makes this approach more difficult than either of the other approaches [Cof90].

This paper compares using an implicit parallel language, SISAL, to using an explicit parallel language, SR. SISAL(Streams and Iteration in a Single Assignment Language) is a general purpose functional language [FCO90]. It is implemented using a dataflow model, meaning program execution is determined by the availability of the data, not the static ordering of expressions in the source code. The compiler can schedule the execution of expressions in any order, including concurrently, as long as it preserves data dependencies. Appendix A contains a brief overview of SISAL.

SR (Synchronizing Resources) is a general purpose imperative language with processes, communication, and synchronization integrated into the language [AOC+88, AO93]. It has shared variables, semaphores, synchronous and asynchronous message passing, remote procedure call, and rendezvous. Sequential, shared-memory, and distributed programs can be written in SR. Appendix B contains a brief overview of SR.

We evaluate SISAL, because it is one of the few applicative languages that has respectable performance on a general-purpose machine, and SR, because it is powerful, flexible, and easy to use and understand. Furthermore, both are public domain and have implementations on many different machines. We compare the two approaches for programmability and performance. For programmability, we evaluate the ease of expressing an algorithm and compiling and debugging a program. We compare the performance of SISAL and SR programs to each other and to a hand-tuned program written in C with parallelism provided by library calls.

Section 2 describes SISAL and SR programs that solve five scientific problems and compares the performance of those programs. Section 3 compares of the programmability of the two approaches.

	Application	*Work Load*	*Data Sharing*	*Synch.*
1	**Matrix Multiplication**	Static	Light	None
2	**Jacobi Iteration**	Static	Medium	Medium
3	**Adaptive Quadrature**	Dynamic	None	Medium
4	**LU Decomposition**	Static	Heavy	Heavy
5	**Mandelbrot**	Dynamic	Light	None

1. Only synchronization is termination detection. All shared data is read-only.
2. Edge sharing with neighbors. Compute maximum change between iterations.
3. Fork/join parallelism. No data sharing.
4. Decreasing work. Every iteration disseminate values to all.
5. Variable amount of work.

Figure 1: Applications evaluated and comments about each.

Section 4 contains a summary and some conclusions. Appendices A and B briefly describe SISAL and SR, respectively. Appendix C contains Jacobi iteration programs written SISAL and SR. Finally, Appendix D compares the sizes of the programs evaluated in this paper.

2 Applications, Programs, and Performance

In order to compare SISAL and SR, we wrote programs for five applications: matrix multiplication, Jacobi iteration, adaptive quadrature, LU decomposition, and Mandelbrot. Ideally, the two languages would be evaluated using many large applications. However, the evaluation of a single large application requires more expertise and time then is available to us; furthermore, the results are likely to be specific to that application only. Nonetheless, the results presented here show the flavor, strengths, and weaknesses of the languages. Moreover, in a large application most of the time is spent executing a small portion of the code [Knu71]; thus, our performance results provide an indication of the expected performance of a larger application. We compare the performance of these programs to each other and to equivalent C programs, which provide a performance baseline and show the overheads introduced by SISAL and SR.

Each subsection below describes one scientific programming problem, discusses the implementation in both SISAL and SR, and compares the performance of programs written in SISAL, SR, and C. Figure 1 summarizes the five applications and shows three properties of each: work load, data sharing, and synchronization. *Work load* is a characterization of whether the number of tasks or the amount of work per task can be determined statically at compile time or whether it has to be determined at run time. *Data sharing* is a measure of the extent to which data is shared. *Synchronization* is a measure of the amount of interprocess synchronization.

The programs evaluated in this paper are efficient in the sense that they are representative of production-quality programs. For example, in matrix multiplication ($C = A \times B$), the inner product uses a row of A and a column of B. Because the C programming language stores matrices in row-major order, accessing a matrix column-wise results in poor cache utilization. Therefore, all matrix multiplication programs transpose the B matrix to improve cache efficiency. Other programs use similar optimizations.

Both SR and C have explicit parallelism; consequently, the C programs solve the problem in essentially the same way as the SR programs. Although the C programs are very similar to the SR programs, they were significantly harder to develop. This paper does not discuss the programmability of the C programs because its focus is languages specifically design for parallelism.

Testing Details

All performance tests were conducted on a Silicon Graphics Iris 4D/340 shared memory multiprocessor. It has four 33-Mhz MIPS processors, 64 Mbytes of main memory, 64 Kbytes of data cache, 64 Kbytes of instruction cache, 256 Kbytes secondary data cache, runs Irix V4.0.1, and has MIPS floating-point units. The Sisal compiler used is `osc 1.2 V12.9.2`. The SR compiler used is `sr V2.2.2`. The C compiler used is `/bin/cc`; the parallelism for C programs is provided through library calls to the Silicon Graphics `mpc` library. Both `osc` and `sr` generate C code, which is also compiled by `/bin/cc`.

The Sisal compiler we had available does not have function call or recursive parallelism, even though this is a natural aspect of the language. We added it by modifying the compiler back end to generate code for the Filaments package, which supports efficient fine-grain parallelism on shared-memory multiprocessors, distributed-memory multicomputers, and clusters of workstations [EAL93, FLA94].

The performance results presented in this paper are the median of at least three separate executions. The reported time reflects only the performance of the applications; it does not include any initialization or finalization of the run-time systems. Performance tests were conducted in multi-user mode when the machine was very lightly loaded. Even so, there are some Unix daemons running in the background competing for the processors. Some tests were made in single-user mode to determine what effect this has on the multi-user times. Single- and multi-user times were the same for the cases with 1, 2, or 3 processors; however, the multi-user times were slightly higher occasionally on four-processor tests.

2.1 Matrix Multiplication

Matrix multiplication solves $C = A \times B$, where A, B, and C are $n \times n$ 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 A and B matrices are read-only; C is write-only. If each process computes distinct elements of C, there is no contention when writing these elements. Therefore, the algorithm requires no interprocess coordination, and synchronization is needed only to detect termination.

All programs (Sisal, SR, and C) transpose the B matrix before multiplying, as discussed above. (The times for the non-transposed programs are 2-5 times greater on a 500×500 matrix.)

2.1.1 Sisal Program

The Sisal program consists of a parallel `for` loop in two dimensions that returns the values of elements of C. The body of each loop contains another `for` loop that computes the inner product. Another 2-dimensional `for` loop transposes the B matrix. Figure 11 in Appendix A contains a Sisal matrix multiplication program.

2.1.2 SR Program

Because parallelism is explicitly coded in SR, the SR program must create processes and distribute the work among them. However, the decomposition is simple because the work load is regular: every

	400×400			500×500			600×600		
CPUs	*Sisal*	*SR*	*C*	*Sisal*	*SR*	*C*	*Sisal*	*SR*	*C*
1	34.5	60.5	28.2	66.9	118.	56.2	115.	215.	95.6
2	17.7	30.7	14.6	34.5	60.0	29.3	59.6	114.	52.9
3	12.3	20.8	10.2	23.7	40.8	21.2	40.9	77.8	38.0
4	9.12	16.0	8.73	19.2	31.1	16.7	31.9	60.4	29.7

Figure 2: Matrix multiplication (in seconds).

point in C requires the same amount of work. Therefore, to balance the load, each process will compute an equal number of points. Furthermore, to minimize the data needed by each process and to improve caching, the program distributes the entire rows to one process. The program assigns a horizontal strip of C to each process, which will read the corresponding strip of A and every element of B.

The SR matrix multiplication program is simple. First, create the A and B matrices. Then, create a process for each horizontal strip, which computes its portion of C in a two-dimensional **fa** (for-all) loop. After the processes are finished, the SR run time system calls the **final** code block[1], which can process the resulting matrix.

2.1.3 Performance Comparison

For matrix multiplication the SISAL program is usually 10-20% slower than the C program. The time of the SR program is almost twice that of SISAL. Most of the overhead in SISAL and some of the overhead in SR are due to the complicated C codes that are generated by osc and sr. The generated code is necessarily more complicated because SISAL and SR are much higher-level than C, and hence the codes generated are less efficient the what can be written directly in C.

The performance of the `sr`-generated executable is much worse than the `osc`-generated executable, primarily because the compilers handle array dereferencing differently. Although both compliers generate a complex expression, consisting of multiple pointer dereferences and index or offset calculations, `osc` optimizes this expression when it is in a loop. It "caches" a pointer to the array data in a local variable, which achieves two performance gains. First, each array reference in the loop is now a simple pointer dereference, instead of a complex expression (this is called common sub-expression elimination). Second, at the end of each iteration, (if possible) the pointer is incremented so it points to the "next" element, saving the cost of re-evaluating the complex expression on each iteration (this is called strength reduction). Consequently, for code that accesses arrays in a loop, the performance of `osc` is comparable to `cc`. In contrast, `sr` generates a complex expression for every array access and is not comparable to `cc`.

2.2 Jacobi Iteration

Jacobi iteration is a technique for solving Laplace's equation. In two dimensions with constant, uniform boundaries, Laplace's equation is:

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

[1] In SR the user may specify a **final** code block, which will be executed when the program shuts down. The **final** code, if it exists, is always the last block of user code that is executed.

CPUs	*ijac*	*bjac*	*brjac*
1	66.5	62.4	58.6
2	35.2	38.0	36.7
3	27.6	28.8	26.3
4	23.3	24.1	21.5

Figure 3: Sisal Jacobi iteration (100×100, in seconds).

The region of interest is discretized onto a 2-dimensional mesh, U. Jacobi iteration uses the following equation to compute the $(k+1)$th approximation of U from the kth approximation:

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

The algorithm iterates until the solution converges; i.e., until the maximum change (δ_{MAX}) that occurs at any point is less than some threshold (ϵ).

Like matrix multiplication, the work load for Jacobi iteration is static and regular. However, processes share the data that are on the boundary of each partition; therefore, synchronization is required to ensure that neighboring processes are finished accessing the boundary data before it is over-written. This synchronization can be accomplished with a barrier, which ensures that all processes have arrived before any are allowed to proceed.

2.2.1 Sisal Programs

In order to explore performance trade-offs, three distinct Jacobi iteration programs were written in SISAL (Figure 3). The first SISAL program (*ijac*: *interior* Jacobi) creates an $n \times n$ array and updates each point using one of nine different equations. The primary function call takes an array and returns both the maximum change of any point and an updated array. If the maximum change is greater than the threshold ($\delta_{\mathrm{MAX}} > \epsilon$), then another iteration (function call) is performed. Because (1) is only defined for the interior points, it is necessary to decide if each point is adjacent to the boundary: If it is in the interior, Equation (1) is used, otherwise a special equation is used. For example, the following equation computes the new values of the top row (the $north_j$'s are the constant values of the top boundary):

$$u_{1,j}^{(k+1)} = \frac{1}{4}(north_j + u_{2,j}^{(k)} + u_{1,j-1}^{(k)} + u_{1,j+1}^{(k)}).$$

Similar equations are needed to compute the points adjacent to the *south, east,* and *west* boundaries, and slightly different equations are needed for the four corner points. A complex *if* expression determines which of the nine equations to use. This expression is expensive because it is comparable to the computation required to update a point, which otherwise contains only 4 additions, a multiply, and a compare.

The second program (*bjac*: *boundary* Jacobi) eliminates the complex *if* expression in *ijac* by using an $(n+2) \times (n+2)$ array that contains a boundary on each edge. Equation (1) updates each of the $n \times n$ interior points because all have four neighboring points. First, the new interior points are computed, and then the boundary is appended to the array. The appending of arrays in *bjac* involves (logically) copying $4n+4$ boundary elements.[2]

[2]The `osc` compiler recognizes that the north and south boundaries rows do not change, and "appends" the top

	75 × 75			100 × 100			150 × 150		
CPUs	*Sisal*	*SR*	*C*	*Sisal*	*SR*	*C*	*Sisal*	*SR*	*C*
1	24.6	117.	22.9	58.6	209.	51.3	178.	772.	162.
2	12.5	59.8	11.4	36.7	105.	25.0	90.9	381.	76.0
3	11.1	39.2	7.47	26.3	70.8	16.8	70.2	254.	47.4
4	8.54	30.5	5.82	21.5	53.4	12.5	52.8	196.	35.5

Figure 4: Jacobi iteration (in seconds).

The third program (*brjac*: *boundary, rows only*) is a hybrid of the first two approaches. Only the *east* and *west* boundaries are appended to the array, reducing the amount of data that is needed and copied at the expense of an *if* expression. The *if* expression in this program is much less expensive than the one in *ijac* because it is simpler (three versus nine arms) and is only evaluated once per row (n times) rather than once per point (n^2 times). The *brjac* program is the fastest of the SISAL programs tested.

2.2.2 SR Program

For the SR Jacobi iteration program, the design and development were straightforward. The program consists of initialization and finalization code, plus the Jacobi iteration kernel that is executed by the worker processes parallel. Initialization consists of reading run time parameters, creating data structures, and forking processes. Finalization consists of printing the solution after the workers finish. In the worker code, new values are computed every iteration, the maximum difference is updated, and the processes synchronize. The synchronization is done with two barriers. Between the barriers one process checks for convergence. The SR program uses a boundary around the array, so there is no complicated *if* statement.

The SR program uses a static decomposition because the work load is known at compile time. Additionally, processes are given entire rows to compute in order improve the cache performance. Each process works on approximately n/p rows, where p is the number of worker processes (and the number of processors); processes are given contiguous rows because it is simpler and there is no performance advantage afforded by a more complicated scheme.

The maximum change (δ_{MAX}) is maintained locally by each process, instead of using a single variable global to the program. This eliminates the contention that would occur when the global is accessed concurrently. However, after every iteration, one process has to "reduce" the local maxima into the global and check for convergence ($\delta_{MAX} < \epsilon$). Although this improves performance while updating the points, it requires an extra barrier per iteration. In particular, the reduction cannot be done until all processes finish updating their points (the first barrier), and it must be completed before any process start updating points for the next iteration (the second barrier).

2.2.3 Performance Comparison

The times for the Jacobi iteration programs are shown in Figure 4. Those for SISAL are from the best program (*brjac*). All tests were run with a convergence tolerance of 10^{-4}.

The C program is efficient and scales very well; sometimes it gets better-than-linear speedup.

and bottom rows by copying a pointer to these rows. Thus the appending of the boundary actually requires only $2n + 4$ copies.

```
function quad(a,b: real) returns A: real
    c := (a + b)/2.0                # calculate mid point
    # compute three areas using trapezoidal (or similar) rule
    A_full := area(f(a), f(c), c − a)
    A_halves := area(f(c), f(b), b − c) + area(f(a), f(b), b − a)
    if |A_full − A_halves| < ε then A := A_halves
    else A := quad(a, c) + quad(c, b)
end
```

Figure 5: Adaptive quadrature algorithm.

Each process accesses $O(n^2/p)$ elements; therefore, as the number of processes increases, the size of the data required by each process decreases. If the data fits in the cache, there is a huge increase in performance, which accounts for the better-than-linear speedup.

The `osc` compiler was able to build the new array in place even though it is created concurrently by distinct processes. Additionally, in *bjac* and *brjac* the append operations are performed in place. These optimizations are absolutely necessary for the SISAL program to run efficiently. However, unlike C, `osc` does not reuse arrays. Consequently, the fastest SISAL program, *brjac*, is 5-15% slower than the C program on a single processor.

SR is again much slower than C, even though the C and SR programs reuse the arrays by changing their roles after each iteration (old becomes new and new becomes old). There are six array accesses per point on every iteration, and there is very little work per point. Therefore, the array accessing overhead is significant.

2.3 Adaptive Quadrature

Numeric quadrature computes the definite integral of a function over an interval ($\int_a^b f(x)dx$) by dividing the interval into subintervals. The area of each subinterval is approximated (using a method like the trapezoidal rule), then the approximations of the subintervals are added together to obtain the approximation for the entire interval. In adaptive quadrature the subintervals are determined dynamically. A function estimates the area of a subinterval and the areas of the right and left halves of this subinterval. If the difference between the sum of the two smaller areas and larger area is small enough, the approximation is returned. Otherwise the function recursively (and in parallel) computes the areas of the left and right subintervals and returns the sum of the two subareas. Figure 5 contains pseudocode for this algorithm.

In order for the function to be integrated by adaptive quadrature, it must be continuous and computable everywhere on the interval. Our programs use $f(x) = e^x$, because the slope varies throughout. This ensures an irregular work load; therefore, the amount of work is not known at compile time. Furthermore, it is easy to verify the results analytically.

2.3.1 Sisal Program

The recursive program in Figure 5 is quite simple to write in SISAL. However, `osc` does not parallelize function calls. This is unfortunate because recursion is the simplest and most natural way to express many algorithms.

To provide function call parallelism, the `osc` compiler was modified to produce Filaments code. The modified compiler, `fsc`, links the generated code with the Filaments run time library, which provides efficient fine-grain parallelism, and in particular, fork/join or function call parallelism

CPUs	*SISAL* osc	*SISAL* fsc	*SR* *bag*	*SR* *prune*	*C*
1	26.7	23.7	25.8	18.2	7.79
2		12.1	15.3	9.10	3.87
3		8.17	13.3	6.07	2.61
4		6.65	22.5	5.64	2.18

Figure 6: Adaptive quadrature (interval: 1.0 to 20.0, in seconds).

[FLA94]. Modifications were made to the `osc` run time system and to the code generation phase of the back end. Because the modifications to the back end of the compiler were confined to the code generator, `fsc` retains the optimizations performed by `osc` [FA95].

2.3.2 SR Programs

We wrote three different adaptive quadrature programs in SR. The simplest SR program, *co*, uses the SR **co** (**cobegin**) statement to invoke the two recursive calls in parallel (the next to last line in Figure 5). Because this statement creates a process for each call, it results in an explosion of processes.

The second program, *bag*, uses a *bag of tasks*, which contains all tasks to be done, and worker processes, which continuously remove and execute tasks [And91]. In this algorithm, we create work by inserting a task (describing the work) into the bag; eventually a worker removes and executes this task. Instead of forking two processes as is done in the *co* program, *bag* places one task (subinterval) in the bag and continues working on the other task. Because a task may be executed by any worker, the bag balances the work among the workers. Additionally, there is no explosion of processes, like occurs in the **co** program. This program uses local partial totals of the area to avoid contention at a global variable. Each worker accumulates it own partial total of the area of all the intervals it has evaluated. When all the work is done, these partial areas are summed in the **final** code block to obtain the total area.

The third program, *prune*, also uses a bag of tasks, but limits the number of tasks inserted into the bag to improve performance. Because there is very little work required per task, the overhead of inserting and removing tasks is considerable. A worker does not insert a task into the bag if there is sufficient parallelism already. It inserts a task only if $T_{in} - T_{out} < \delta$, where T_{in} (T_{out}) is the number of tasks inserted (removed) by the worker, and δ is a parameterized limit. In this case a small δ ensures that sufficient tasks are inserted to balance the load; this program uses $\delta = 3p$.

In SR a bag-of-tasks program is easy to implement because SR both provides powerful message passing primitives and detects termination. In SR message passing is many-to-many, so a message queue suffices as a global bag: A worker simply sends a message to insert a task and receives a message to remove a task. An SR program will terminate when all processes are blocked receiving a message and there are no more messages to be delivered. At which point the `final` code block is called, which adds the partial areas to obtain the final result.

2.3.3 Performance Comparison

As noted above, the SISAL compiler did not expose any parallelism in the programs we wrote, so only the single processor time is reported. Furthermore, it is 47% slower than *prune*. In addition, the non-pruning SR program, *bag*, is as fast as the SISAL program, even though *bag* inserts and removes every task (resulting in more than 122,000 sends and receives). The SISAL program uses recursive calls, it is a poor implementation; a recursive SR program finishes in 18.1 seconds—as fast as *prune* and 32% faster than the SISAL program.

The times shown in the `fsc` column of Figure 6 were obtained using `fsc`, a modified SISAL compiler [FA95]. The speedup is nearly perfect and on a single processor it is better than SISAL. The Filaments library performs pruning, which explains the excellent speedup and the good single processor performance.

The *co* program is not very efficient. It took 8 times longer (7.08 versus 0.85 seconds) than *bag* on a small interval $[1, 10]$. Therefore, it was not tested on the full interval. In the SR program *bag*, contention for the bag increases as the number of workers increases, causing the performance to deteriorate to the point where with four workers it runs 70% slower than with three workers. This is because the bag is simultaneously accessed by many processes, and these accesses must be serialized. The program *prune* limits the number of tasks that are inserted, which both reduces contention and eliminates some insertions and deletions.

2.4 LU Decomposition

LU decomposition solves the linear system: $Ax = b$ [PFTV88], by decomposing A into lower- and upper-triangular matrices, such that $A = LU$. Then the linear system becomes $A = LUx = b$ and the solution, x, is obtained by solving two triangular systems $Ly = b$ and $Ux = y$, using back-substitution. We use Doolittle's method to calculate L and U, which is defined as follows:

$$u_{ij} = \begin{cases} a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} & i \leq j \\ 0 & i > j \end{cases} \tag{2}$$

$$l_{ij} = \begin{cases} \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) & i > j \\ 1 & i = j \\ 0 & i < j \end{cases} \tag{3}$$

The n^2 equations in (2) and (3) depend on one another. In particular, the kth row of U depends on the first $k-1$ rows of U and the the first $k-1$ columns of L. Similarly, the kth column of L depends on the first k rows of U and the first $k-1$ columns of L. A *single-pass* algorithm for LU decomposition is:

do $k := 1$ **to** $n \rightarrow$
  parallel do $j := k$ **to** $n \rightarrow$ compute u_{kj} **od** # using (2)
  parallel do $i := k$ **to** $n \rightarrow$ compute l_{ik} **od** # using (3)
od

This algorithm requires storage for L, U, and A. Because L and U are triangular, the storage requirement is $2n^2$ elements.

There is also an *incremental* algorithm that can create L and U in the same storage provided for A. It computes the summations in (2) and (3) one term at a time (incrementally) and stores the intermediate results in place. The first row of U is identical to the first row of A, so it is unchanged. Similarly, the first column of L only needs to be be normalized by $u_{11}(a_{11})$. The following pseudocode shows the main loop of incremental algorithm:

do $k := 2$ **to** $n \rightarrow$
  parallel do $i := k$ **to** $n, j := k$ **to** $n \rightarrow$
    $a_{ij} := a_{ij} - a_{ik} a_{kj}$
  od

parallel do $i := k + 1$ **to** $n \rightarrow a_{ik} := a_{ik}/a_{kk}$ **od**
od

On each iteration of the sequential **do** loop, the elements in row i and column j obtain their final values, leaving $(n-k)^2$ "active" elements remaining in the lower right corner of the matrix. According to (3) each element $l_{ij}, i > j$, is divided by u_{jj} (a_{jj}); as shown in the pseudocode above.

The incremental algorithm has three advantages over the single-pass algorithm: less synchronization, finer parallelism, and better data locality. First, there is one less phase per iteration; consequently, there is one less barrier. Second, the parallelism is finer, and therefore, more scalable. On the kth iteration in the single-pass algorithm at most $(n-k)$ values can be updated in parallel; in the second algorithm at most $(n-k)^2$ values can be updated in parallel. Lastly, there is little locality between iterations in the single-pass direct algorithm, because a process accesses a different set of data each iteration. In the incremental algorithm, however, all updated points were accessed on the previous iteration.

In both algorithms the last operation in computing the elements of L is a division by u_{jj}. If u_{jj} is close to zero, roundoff errors can be introduced. The usual way to reduce the effect of roundoff error is to *pivot*: interchange the rows (or columns) so that the largest element is moved to the diagonal. Because we are solving a linear system, rows (or columns) can be interchanged without affecting the solution.

2.4.1 Sisal Programs

Both algorithms were programmed in SISAL. The single pass program, *slu*, is a simple translation of equations (2) and (3) into SISAL code. Because two-dimensional arrays in SISAL do not have to be rectangular, *slu* creates L and U as triangular matrices, saving space without having to use a complicated indexing scheme. The L matrix is transposed, because it is created and accessed column-wise. This way, on each iteration, the program concatenates a column to the previously computed columns of L.

A second program, *slup*, uses the single-pass algorithm and also performs pivoting. After determining the rows to pivot, the A matrix and what has been created of the L matrix are pivoted. The U matrix does not need to be pivoted because the rows that are exchanged have not yet been computed and appended to U. Pivoting the A matrix involves exchanging two rows, which can be accomplished by exchanging pointers to the rows. However, because the L matrix is stored in column major order, pivoting involves exchanging two elements in each row of L.

The *ilu* program implements the incremental algorithm. This program, a modified version of one that came with the SISAL distribution, is not as natural or elegant as the *slu* program. Although the active portion of the matrix decreases, *ilu* always creates a new $n \times n$ matrix. On iteration k all active elements are updated and copied to the new array: $a'_{ij} := a_{ij} - a_{ik}a_{kj}$; the inactive elements are simply copied to the new matrix as is: $a'_{ij} := a_{ij}$; and active elements below the diagonal in column k, are normalized: $a''_{ik} := a'_{ik}/a'_{kk}$. There is tremendous copying of inactive values from the old matrix to the new matrix. Additionally, this program performs pivoting.

2.4.2 SR Program

Only the incremental algorithm was programmed in SR, because it is the more efficient algorithm. Two aspects of the program are interesting. First, the work load decreases with every iteration, so a block decomposition does not yield a balanced work load; hence, the program uses a cyclic decomposition. In particular, a worker updates the elements in n/p rows, in a cyclic or modulo

CPUs	*slu*	*slup*	*ilu*
1	188.	534.	154.
2	112.	232.	85.7
3	84.3	170.	65.1
4	71.7	141.	57.4

Figure 7: SISAL LU decomposition (500×500, in seconds).

	400×400			500×500			600×600		
CPUs	*Sisal*	*SR*	*C*	*Sisal*	*SR*	*C*	*Sisal*	*SR*	*C*
1	82.3	27.5	18.0	154.	53.5	36.6	279.	92.7	61.5
2	47.1	19.0	8.94	85.7	37.2	18.5	154.	64.2	32.2
3	36.1	12.7	6.66	65.1	25.1	12.6	117.	46.2	22.8
4	31.6	9.65	4.19	57.4	22.0	9.62	101.	39.8	17.7

Figure 8: LU decomposition (in seconds).

pattern (worker w is assigned rows $w + ip, 0 \le i < n/p$). This results in a work load that is almost evenly balanced throughout the computation.

The second interesting aspect is pivoting, which requires synchronization. On the kth iteration, first, the process that is assigned the kth row scans the active values in the row and selects the largest element as the pivot; this determines the column that will be pivoted with column k. Then all processes perform the pivot by exchanging the elements in the kth and the pivot column in parallel. A barrier occurs before and after the first step in pivoting to ensure correctness: The pivoting process must wait for the other processes to finish updating on the previous iteration, and the other processes cannot start updating this iteration until the pivot column has been selected. Because the other processes need the new values of the kth row to update their points, the pivoting process updates row k while it is selecting the pivot.

2.4.3 Performance Comparison

The SR program is much faster than the SISAL programs, even though the performance of the SR program is hampered by inefficient array accessing. The *ilu* program performs poorly because the SISAL program builds the same-sized matrix each time, and the problem size shrinks at each step. The single-pass programs, *slu* and *slup*, suffer because more memory is needed and there is little data locality. Due to a memory optimization bug in `osc`, we had to use rectangular arrays for L and U instead of triangular arrays; therefore, *slu* and *slup* used 3 times as much memory as *ilu* (instead of just twice as much). Unfortunately, *slup* does not exchange two elements in place during the pivot; therefore, it must copy all n elements in each row to a newly allocated row, performing the pivot while copying.

2.5 Mandelbrot

The Mandelbrot set is in of the two-dimensional plane of the complex numbers [Dew85]. When the operation in Figure 9 is applied to complex numbers, the ones outside the Mandelbrot set grow

```
function point (c: complex) returns count: integer
    var z: complex := c
    count := 0
    do count < limit ∧ |z| < 2.0
        count := count + 1
        z := z^2 + c
    od
end
```

Figure 9: Function which calculates the value of points in Mandelbrot.

to infinity very quickly. The Mandelbrot set is visualized by setting the value of point (x, y) (or equivalently, $c = x + yi$) to the value of *count* returned by the function `point` in figure 9. Points that are not in the Mandelbrot set diverge quickly and receive a value near 0. Points that are in the set do not diverge and receive the value of *limit*. The interesting parts of the image are at the boundaries of the set, points with values between 0 and *limit*. The parameter *limit* determines the number of unique values that will be in the image.

To create an image of the Mandelbrot set, one selects a region and a resolution. The region is any rectangle containing some of the Mandelbrot set, which extends approximately from $-2 - 1.25i$ to $0.5 + 1.25i$. Choosing the above region selects the entire Mandelbrot set, whereas choosing the region from $-0.75+0i$ to $0.5+1.25i$ zooms in on the upper-right quarter of the Mandelbrot set. The resolution determines the number of points in the resulting image. The value of a point depends only on its x and y coordinates; therefore, each point can be calculated independently.

This problem was chosen because it has an irregular, dynamic work load, requires no synchronization other than termination detection, and has no data sharing. Because of the dynamic work load, a block decomposition of the problem will likely be unbalanced—with different amounts of work required in each partition.

2.5.1 Sisal Program

The SISAL function that creates the Mandelbrot image was called from a C language program. This application uses the foreign language interface of `osc`, because SISAL does not have any I/O native to the language.[3] The SISAL Mandelbrot function returns a 2-dimensional array that is the image. The C program writes this array to a file in an image format, so that it can be view using a standard viewer.

Because the operation in Figure 9 is so simple, there is only one way to code the Mandelbrot function in SISAL. The SISAL Mandelbrot function consists of a two-dimensional for-all loop over all the points in the image. In the body of the loop is a while loop (called a *for-initial* loop in SISAL) that performs the operation in Figure 9.

2.5.2 SR Program

The SR program uses a worker process per processor and a bag of tasks. In previous problems the bag was used because the work grows dynamically. In this program the bag balances the problem

[3]The Fibre system (described in Appendix A) provides limited I/O using the Fibre format only. If a different format is required, a user must either translate to and from Fibre, or write a program that will read or write the data and invoke SISAL from inside this program. We chose the latter approach.

	600 × 600 image			700 × 700 image			1000 × 1000 image		
CPUs	*Sisal*	*SR*	*C*	*Sisal*	*SR*	*C*	*Sisal*	*SR*	*C*
1	25.5	36.3	23.2	34.6	50.1	31.8	69.1	75.3	40.4
2	12.9	18.3	11.8	17.5	24.9	16.0	35.0	37.7	20.1
3	8.57	12.1	7.96	11.7	16.6	9.69	23.5	25.1	13.0
4	6.49	9.13	5.84	8.80	12.4	7.71	17.9	18.9	9.91

Figure 10: Mandelbrot (255 gray-levels, in seconds).

because the amount of work per task varies. The program initially inserts N tasks into the bag, one for each row. The workers continuously remove tasks from the bag. Because all tasks are initially in the bag, the workers never insert a task. For each task received from the bag, the worker computes the values for all points in the corresponding row. This algorithm was very simple to write in SR because the worker code is trivial and the bag of tasks is easy to program in SR. There is no pruning, because there is no explosion of tasks.

2.5.3 Performance Comparison

The times for the three Mandelbrot programs on The SR program is much slower than C on one processor; again, this is because of the array accessing overhead.

The SISAL program is competitive with C and scales well. By default a SISAL program partitions the work into the same number of pieces as there are workers. However, this partitioning is unbalanced and the speedup is poor, because the overall time is the time of the slowest process. Using the run-time flags provided by `osc`, the work is partitioned into many more partitions than there are workers. Similar to the bag-of-task programs, the workers repeatedly get new partitions until there are no more.

3 Programmability Comparison

This section compares the ease of programming implicit and explicit parallelism. SISAL and SR use the functional and imperative models, respectively. Although the model primarily defines the approach, the language also has a tremendous impact on programmability. Moreover, a compiler determines not only the quality of code, but affects the usability of the language. We compare SISAL and SR in terms of the model, the language, and the compiler. Although a feature can have an impact on more than one level, it is mentioned only in the most appropriate subsection.

3.1 Models: functional versus imperative

The functional model provides a very high level of abstraction. In particular, a functional language program does not depend on the underlying architecture, and it is deterministic (in a correct program, a specific input produces the same output every time). On the other hand, the abstraction provided by the imperative model is at a comparatively low level. It is characterized as having a program state (i.e., variables) that is explicitly manipulated by the program.

In the functional model, the value of a "variable" does not change once it is defined and the result of a function does not depend on the context in which the function is called. In other words, a functional language has no *side effects* and is *referentially transparent*. Therefore, expressions can be evaluated in any order, and a variable can be replaced by its value (and vice-versa), providing flexibility in the order of execution of the expressions [Hug90]. Furthermore, it is easier to reason about and make assertions of a program that is referentially transparent—helping greatly in program verification. Moreover, debugging is much simpler in a deterministic language [Bac78, AE88].

The functional model has two significant handicaps: no state and no asynchrony. Without state, some applications, such as data bases, cannot be written. Furthermore, because of determinism, asynchronous programs (such as interrupt-driven device drivers) also cannot be written.

Because the imperative model is lower level than the functional model, the programmer has greater flexibility in expressing an algorithm and tuning the program for efficiency. However, the programmer also has greater responsibility. For example, because imperative languages have side effects, statements cannot be arbitrarily rearranged by the compiler; therefore, the programmer must ensure that the statements are ordered efficiently. Furthermore, because of asynchrony imperative programs are generally not deterministic, so the programmer has to ensure correctness by employing mechanisms that avoid race conditions.

3.2 Languages: Sisal versus SR

Although a language owes much of its character to the underlying model, every language is designed for a specific purpose and has its own strengths and weaknesses. Sisal is designed primarily for parallel scientific applications (it is intended to be used instead of FORTRAN). SR is a general purpose concurrent programming language that supports both distributed- and shared-memory parallel programs, as well as sequential programs.

The high level of abstraction provided by Sisal has two major benefits. First, every Sisal program executes correctly on every machine (that is supported by the compiler). Second, because parallelism is implicit, there is no parallel code in Sisal programs. There no need for the programmer to manage processes or to communicate because the compiler (or run-time system does this). Moreover, the programmer does not need to partition and load balance a problem.[4] Therefore, the functional abstraction provides portability and simplicity. However, the major disadvantage of Sisal is the same as its major advantage: implicit parallelism. The programmer must rely on the compiler and its parallelization, even if a better solution is known. For example, very efficient algorithms are known for both Jacobi iteration and LU decomposition; however, neither algorithm is expressible in Sisal.

Two other limitations of Sisal are the lack of input/output[5] and the lack of globals. Because there is no I/O, all parameters must be passed into the main function at the beginning of execution, and all results must be returned by this function when the program terminates. Input cannot be read interactively, and all output data must be kept in memory until the program terminates. The lack of globals forces the programmer to pass all information using function parameters. Both of these limitations increase the number of parameters to a function, which can be become very large. For example, the main function to the Australian Weather Kernel that has 43 input and 12 output parameters [Ega93].

The primary advantage of the explicit mechanism in SR is that the programmer has complete control over the parallelism. This provides flexibility and does not require, nor rely, on compiler analysis. However, there are three major disadvantages. First, the programmer must control the parallelism, even if control is not desired. This not only means more code must be written and debugged, but, because it involves concurrency, the code can be problematic. For example,

[4] Although there is no need (or ability) to partition or load balance a problem at the language level, the compiler may require some hints from the programmer to accomplish this efficiently.

[5] Although not native to Sisal, limited I/O is provided by `osc` through Fibre, see Appendix A.

often shared data must be modified only in critical sections, which then must be identified and protected. Fortunately, SR minimizes this difficultly by providing clean syntax and semantics for parallel constructs. The second disadvantage is that the programmer must decompose the problem. Although this is often trivial, as in matrix multiplication and Jacobi iteration, it can be very difficult, as in adaptive quadrature. The third disadvantage is that SR programs are not independent of the machine architecture. The programs described in this paper are for shared-memory machines; significant editing and testing are necessary to port them to a distributed-memory machine.

3.3 Compilers: `osc` versus `sr`

A compiler is an implementation of a language, which ultimately is responsible for the usability of the language. The optimizing SISAL compiler (`osc`) supports high-performance, parallel scientific applications. Consequently, the primary focus is to be competitive with parallelizing FORTRAN compilers (i.e., efficient for-all parallelism). The SR compiler (`sr`) focuses on efficiently implementing the concurrent constructs of the SR language on many different platforms. Because SISAL has implicit parallelism and other exotic features, any SISAL compiler has to perform many complex analyses, such as extracting parallelism. In contrast, SR is an imperative language with explicit parallelism, so `sr` is a relatively straightforward compiler with no exotic techniques.

Two major features of `osc` deserve mention because they greatly effect the performance of `osc` executables. First, the `osc` compiler uses three techniques to eliminate excess copying: scheduling expressions, updating in-place, and building in-place. Only one program we tested (*slup*) proved too difficult for `osc` to analyze and subsequently optimize. The second major feature of `osc` is partitioning. The compiler makes an estimate of the cost of executing each function; this cost determines the static partitioning of the problem. Consequently, the programmer does not have to partition the problem.

Two minor features of `osc` also deserve mention. First, it employs many standard compiler optimizations, such as inlining, loop unrolling, and strength reduction. This improves the quality of the generated code and, consequently, the `osc`-executable. Secondly, tokens are inserted if `osc` finds a syntax error in the program source. This helps a novice user learn the language, because the correct token is almost always inserted.

Two drawbacks to `osc` also deserve mention. First, static estimating cannot always produce a partition that evenly balances the load. Consequently, the user must evaluate the program and tune it through the various compile- and run-time parameters, defeating a major advantage of implicit parallelism. Second, `osc` does not parallelize function calls, even though this parallelism is implicit in the language, limiting the problems that SISAL and `osc` effectively solve.

The `sr` compiler translates SR source into C that is linked with the `sr` run-time system. Because SR has explicit parallelism and is an imperative language, `sr` does not partition the problem or schedule expressions; rather, the programmer does. Furthermore, `sr` lacks some standard optimizations, such as code hoisting, which explains some of the inefficiency of `sr` relative to `osc`. Consequently, `sr` is much simpler than `osc`. The `sr` compiler is robust, fast, and fully featured. Although, the performance of `sr`-executables, especially array-intensive codes, can be poor, `sr` efficiently implements the concurrent aspects of SR, such as multi-threading and message passing.

4 Summary and Conclusions

There are two primary trade-offs between SISAL and SR. The first is implicit versus explicit parallelism. In SISAL the compiler automatically implements the parallelism, whereas in SR the programmer manually implements it. Clearly, implicit parallelism is better only if the compiler creates an efficient program. On the other hand, explicit parallelism is better when there is a big improvement in performance. When the performance of the approaches is similar, the ease of programming favors implicit parallelism.

The second trade-off is the ease of programming versus the flexibility to control the program execution. Although the implicit parallelism of SISAL allows codes to be shorter and simpler, it comes at the expensive of control. For example, in Jacobi iteration the SISAL programs were shorter and simpler than the SR program. However, the SR program uses only two arrays, whereas the SISAL programs allocate and de-allocate an array on each iteration. The programmer knows that the array can be re-used on the next iteration; however, this cannot be expressed in SISAL. Two of the implementors of SISAL have reached the same conclusion; [BO94] states that some SISAL programmers "have found that they need greater freedom to express algorithms."

SISAL provides a high-level abstraction that hides many details from the programmer. Because the `osc` compiler schedules expressions and partitions the problem based the result of its analysis, small changes to SISAL source can result in huge changes to the `osc` output. As a result, the programmer must view `osc` is a "black box," because the effect of source code modifications is unpredictable. In contrast, in `sr` output is very similar to SR source; therefore, `sr` is predictable, which makes tuning for efficiency simpler in SR than in SISAL.

We make three conclusions. First, although `osc` is mature and well-developed, there is no reason to expect it to get better. It may not represent the limit of implicit parallelism, but a project this mature cannot be expected to produce dramatic new results. Furthermore, it indicates that new gains in this area using similar approaches will not likely be forthcoming.

Conversely, we expect improvements in SR. The biggest drawback to SR is its performance. However, it is sequential aspects of the code that are inefficient. Many of these aspects are efficiently implemented by `osc`; therefore, `sr`'s efficiency can be improved with techniques that are used in `osc`.

The third conclusion is that SISAL is very good for its limited problem domain, which is almost exclusively "loop-parallel" applications. For example, SISAL does not support applications that are interactive or asynchronous; nor does it support applications that require state, input, or output.

5 Acknowledgements

Greg Andrews provided many organizational, technical, and grammatical suggestions and spent countless hours reading and editing the drafts of this paper. Dawson R. Engler provided the C versions of Mandelbrot and LU decomposition. David Lowenthal provided the SR version of LU decomposition and the matrix multiplication, adaptive quadrature, and Jacobi iteration programs in C. He also provided many good technical suggestions. David Mosberger helped get `osc` up and running on our machines. Finally, Gregg Townsend provided assistance with the SR compiler.

References

[AE88] Arvind and Kattamuri Ekanadham. Future scientific programming on parallel machines. *J. Par. and Dist. Comp.*, 5:460–493, 1988.

[And91] Gregory R. Andrews. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, Redwood City, California, 1991.

[AO93] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language.* Benjamin/Cummings, Redwood City, California, 1993.

[AOC+88] G. Andrews, R. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend. An overview of the SR programming language and implementation. *TOPLAS*, 10(1):51–86, January 1988.

[Bac78] Jim Backus. Can programming be liberated from the von Neumann style? *CACM*, 21(8), August 78.

[BO94] A. P. W. Böhm and R. R. Oldehoeft. Two issues in parallel language design. *Transactions on Programming Languages and Systems*, 16(6):1675–1683, November 1994.

[Cof90] Michael H. Coffin. *Par: An Approach to Architecture-Independent Parallel Programming.* PhD thesis, University of Arizona, Tucson, AZ 85721, August 1990.

[Dew85] A. K. Dewdney. Computer recreations. *Scientific American*, pages 16–24, August 1985.

[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.

[Ega93] G. K. Egan. Implementing the kernel of the Australian region weather prediction model in SISAL. In John T. Feo, editor, *SISAL '93*, pages 11–17, Livermore, CA, October 1993. Lawrence Livermore National Laboratory, Computer Research Group.

[FA95] Vincent W. Freeh and Gregory R. Andrews. `fsc`: a Sisal compiler for both distributed- and shared-memory machines. In A. P. W. Böhm and John T. Feo, editors, *Proceedings of the High-Performance Functional Computing Conference*, pages 164–172. Lawrence Livermore National Laboratory, April 1995.

[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.

[Hug90] John Hughes. Why functional programming matters. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 17–42, Reading, MA, 1990. University of Kent, Addison-Wesley.

[Knu71] Donald E. Knuth. An empirical study of Fortran programs. *Software—Practice and Experience*, 1(2):105–33, April-June 1971.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, Cambrigde, 1988.

[SY88] Stephen Skedzielewski and Robert Kim Yates. Fibre: an external format for SISAL and IF1 data objects. Report M-154, Lawrence Livermore National Laboratory, April 1988.

A Sisal

Sisal is a functional dataflow language intended for use on general purpose multiprocessors of all architectures. It has been implemented on shared memory multiprocessors, vector processors, and a variety of uniprocessors. The goals of the language are to support both general purpose computing and large scale scientific computing, and to generate highly efficient code [FCO90]. Sisal is expression-based, deterministic, and lacks side-effects.

A Sisal program primarily consists of functions and for loops. There are two types of loops in Sisal: the parallel *for-all* and the sequential *for-initial*. The expressions in the body of a *for-all* loop might be executed in parallel and, therefore, must be independent. The Sisal code fragment in Figure 11 illustrates *for-all* parallelism. The function `multiply` returns the product of the two 2-D matrices `A` and `B`. The `returns` clause may contain *reduction* operations. For example, the innermost `for` expression contains the following clause:

```
returns value of sum A[i,k] * B[k,j].
```

All n products are added together and a single scalar value is returned. The `returns` clause in outer *for-all* expression reduces the n^2 `Cij`'s into a 2-D array.

The other form of the for loop, the *for-initial*, is a sequential while loop. The function in Figure 12 computes the factorial of n using a *for-initial* loop. The keyword `old` refers to the value of the variable in the previous iteration.

Fibre [SY88] is an external data representation for Sisal, which only uses the ASCII character set. The input to a Sisal program is the arguments to the entry function; the output is the return parameters from the entry function. Fibre input must all occur before any of the Sisal user functions are invoked; furthermore, all the output occurs at the end, after all the user code has been executed. Consequently, there is not interactive I/O in Sisal; all input must occur before the program is started and all output must occur after the program has terminated. In [SY88], the authors state:

> Fibre is not designed to be the primary form for input and output in IF1; it is intended to be used as a tool for the early developers of Sisal programs.

Unfortunately, Fibre is the only form for input and output available with the current Sisal compiler.

```
% Multiply A[1..n, 1..n] and B[1..n, 1..n].
%
function multiply(n: integer; A, B: RealArray2D returns RealArray2D)
     for i in 1, n cross j in 1, n
          Cij := for k in 1, n
                 returns value of sum A[i,k] * B[k,j]
                 end for
     returns array of Cij
     end for
end function % multiply
```

Figure 11: Matrix multiplication function in Sisal.

```
function fact (n: integer returns integer)
  for initial
    i := 1
    f := 1
  while i < n repeat
    f := old f * old i;
    i := old i + 1
  returns
    value of f
  end for
end function
```

Figure 12: Iterative factorial function in SISAL.

```
function quicksort ( Data : IntArray returns IntArray )

    function Split (Data : IntArray  returns IntArray, IntArray, IntArray )
        for E in Data
        returns array of E when E < Data[ 1 ]
                array of E when E = Data[ 1 ]
                array of E when E > Data[ 1 ]
        end for
    end function

    % routine body
    %

    if array_size( Data ) > 2 then
       let
          L, Middle, R := Split( Data )
       in
          Main( L ) || Middle || Main( R )
       end let
    else
       Data
    end if

end function % quicksort
```

Figure 13: Recursive quicksort in SISAL.

B SR

The SR programming language is a general purpose concurrent language. SR has many high-level and parallel programming constructs. The code fragment in Figure 14 shows matrix multiplication. The fragment illustrates some of the sequential programming aspects of SR. The **fa** is the *for-all* statement; this statement is similar to the FORTRAN **do** statement. Parallelism can be accomplished through process creation or through the **co** (concurrent) statement. Each arm of the **co** is executed in parallel; the process blocks at the matching **oc** until every arm is finished. The code fragment in Figure 15 shows three **co** statements. The first starts two processes: a producer and a consumer. The second starts `N` identical processes, each with its own parameter. The last **co** statement computes the N^2 innerproducts in matrix multiplication in parallel.

SR also includes several message passing constructs. Messages can be sent using `send` or `call` and serviced with a `receive` statement or by a `proc` (procedure or process). The combination of these four allows the user to do asynchronous message passing (`send/receive`), rendezvous (`call/receive`), procedure call, possibly remote (`call/proc`), and dynamic process creation (`send/proc`).

```
var sum: real
var A, B, C: [N][N] real

fa i := 1 to N, j := 1 to N ->
  C[i,j] := 0.0
  fa k := 1 to N -> C[i,j] +:= A[i,k] * B[k,j] af
af
```

Figure 14: Sequential matrix multiplication in SR.

```
co producer() // consumer() oc

co (i := 1 to N) worker(i) oc

co (i := 1 to N, j := 1 to N) innerproduct(i,j) oc
```

Figure 15: Examples of SR **co** statements.

C Jacobi Iteration Codes

This appendix shows some of the differences between programming in SISAL and SR. The source for all of the SISAL and SR programs used in this paper is available at the following URL:

`http://www.cs.arizona.edu/people/vin/programs.tar.gz`

C.1 Sisal Jacobi Iteration

The Jacobi code for the SISAL program *bjac* is shown in this section. There are four functions in this program: `main`, `jacobi`, `jac_row`, and `jac_point`. First, the function `main` creates the initial $(\mathtt{N}+2) \times (\mathtt{N}+2)$ array and invokes `jacobi`. The function `jacobi` iteratives until the solution converges. Each iteration it invokes `N` instances of `jac_row`, which returns an $\mathtt{N}+2$ element array. These arrays are concatenated together, along with the constant top and bottom rows to create the new, updated $(\mathtt{N}+2) \times (\mathtt{N}+2)$ array. Similarly, `jac_row` invokes `N` instances of `jac_point` and concatenates the constant left and right points before returning the updated row. The function `jac_point` computes the new value of a point, using a 4-point stencil.

```
define main

type OneD = array[double_real];
type TwoD = array[OneD];
type edges = record [ n, s, e, w: double_real ];

%
%  Calculate the new value of A[i,j] using a 4-point stencil
%  Return new value and difference
%
function jac_point(A: TwoD; i, j: integer returns double_real, double_real)
  let
    Mij := (A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1])/4.0d0
  in
    Mij, abs(Mij - A[i,j])
  end let
end function

%
%  Update a row of A.
%  Return an updated row (with m+2 elements) and the maximum difference
%
function jac_row(A: TwoD; i, m: integer returns OneD, double_real)
  let
    Rj, dj := for j in 1,m
           Rij, dij := jac_point(A, i,j)
         returns array of Rij
                 value of greatest dij
         end for;
  in
    array_addl(array_addh(Rj, A[i,m+1]), A[i,0]), dj
```

```
  end let
end function


%
%  Perform Jacobi iteration until convergence (delta < eps)
%
function jacobi(n,m: integer; A: TwoD; eps: double_real returns TwoD,double_real)
  for initial
    C := A;
    delta := eps
  while delta >= eps repeat
    Cis, delta := for i in 1, n
                    Ci, di := jac_row(old C, i, m);
                  returns array of Ci
                          value of greatest di
                  end for;
    C := array_addl(array_addh(Cis, A[n+1]), A[0])
  returns value of C
          value of delta
  end for
end function

%
%  Create the initial array and call jacobi
%
function main(n, m: integer; start, eps: double_real returns TwoD, double_real)
  let
    B := record edges[n:1.0d0; s:0.0d0; e:1.0d0; w:1.0d0];
    A := for i in 0, n+1 cross j in 0, m+1
           Aij := if i = 0 then  B.N
                  elseif i = n+1 then B.S
                  else
                    if j = 0 then B.W
                    elseif j = m+1 then B.E
                    else start
                    end if
                  end if
         returns array of Aij
         end for
  in
    jacobi(n,m,A,eps)
  end let
end function
```

C.2 SR Jacobi Iteration

The key components of SR are *resources* and *globals*. A resource is the main unit of encapsulation in SR; it is similar to a module in other languages. A program contains one or more resources. A global is a collection of objects in an address space; it is basically an instance of a unparameterized resource.

The SR code to perform Jacobi iteration has four components: `params`, `Arrays`, `barrier`, and `Jacobi`. The first global is `params`, which is used to read values on the command line. It is used in this program purely for convenience: the run time parameters are given default values, and command-line parsing is outside of the main routines.

```
global params
  var N, M: int
  var W := 1
  var North := 1.0, South := 0.0, East := 1.0, West := 1.0
  var Start := 0.0, Epsilon := 0.000001
  var verb := false
body params
  getarg(1,N); getarg(2,M)
  getarg(3,W)
  getarg(4,Start); getarg(5,Epsilon)
  getarg(6,verb)
  getarg(7,North); getarg(8,South); getarg(9,East); getarg(10,West)
end params
```

The second global declares and initializes the two grids. It imports the global `params`, so the variables declared in `params` are visible.

```
global Matrices
  import params
  var A, B: [0:N+1,0:M+1] real
body Matrices
   # initialize the matrix A
  fa i := 1 to N, j := 1 to M ->
    A[i,j] := Start
  af

  fa i := 1 to N ->
    A[i,0] := West
    A[i,M+1] := East
    B[i,0] := West
    B[i,M+1] := East
  af

  fa j := 0 to M+1 ->
    A[0,j] := North
    A[N+1,j] := South
    B[0,j] := North
```

```
    B[N+1,j] := South
  af
end Matrices
```

A third global implements a dissemination barrier using a 2-dimensional array of flags. This component exports two operations: `initBarrier()` and `barrier()`.

```
global bar
   op initBarrier(w : int) {call}
   op barrier(who : int) {call}
body bar
   var W := 0
   var flag :  [*][*]int
   var logW : int

   proc initBarrier(w)
      W := w

      fa i := 1 to log(W,2), j:= 0 to W-1 -> flag[i,j] := 0 af

      logW := int(log(W,2))
   end

   proc barrier(id)

      var partner: int
      var dist := 1

      fa i := 1 to logW ->
         partner := (id + dist) mod W
         do flag[i,id] != 0 -> skip od
         flag[i,id] := 1
         do flag[i,partner] = 0 -> skip od
         flag[i,partner] := 0
         dist *:= 2
      af

   end

end     # global bar
```

The last component is the main resource. An instance of this resource is created when the program is started. This resource imports three globals, so one instance of each of those components is created when `Jacobi` is created. The body of the main resource initializes the barrier, records the start time, and creates `W` worker processes. `Jacobi` contains three sections of code: the body, the worker process, and the **final** section. The main Jacobi iteration loop has been unrolled once to both improve efficiency and to change to rolls of the `A` and `B matrices` (in the top half of the loop `B` is updated from `A`; in the bottom half `A` is updated from `B`).

```
resource Jacobi
  import params, Matrices, bar
  external startwatch(val i: int)
  external stopwatch(val i: int)
  op go()

body Jacobi()
  var which: int
  var deltas[0:W-1]: real

  initBarrier(W)
  var start := age()

  process worker(id := 0 to W-1)
    var startrow := id * N/W + 1
    var endrow := (id+1) * N/W
    var temp: real
    var delta: real

    if verb -> write("worker", id, startrow, endrow) fi

    do true ->
      delta := 0.0
      fa i := startrow to endrow, j := 1 to M ->
        temp := (A[i,j-1] + A[i,j+1] + A[i-1,j] + A[i+1,j])/4.0
        delta := max(abs(temp - A[i,j]), delta)
        B[i,j] := temp
      af
      deltas[id] := delta

      barrier(id)
      if id = 0 ->
        fa i := 1 to W-1 ->
          delta := max(deltas[i], delta)
        af
        if delta < Epsilon ->
          which := 0
          deltas[0] := delta
          stop
        fi
      fi
      barrier(id)

      delta := 0.0
      fa i := startrow to endrow, j := 1 to M ->
        temp := (B[i,j-1] + B[i,j+1] + B[i-1,j] + B[i+1,j])/4.0
        delta := max(abs(temp - B[i,j]), delta)
        A[i,j] := temp
```

```
      af
      deltas[id] := delta

      barrier(id)
      if id = 0 ->
        fa i := 1 to W-1 ->
          delta := max(deltas[i], delta)
        af
        if delta < Epsilon ->
          which := 1
          deltas[0] := delta
          stop
        fi
      fi
      barrier(id)
    od
  end

  final
    var et := age() - start
    write("Time:", et)
    write("Converges to ", deltas[0])

    if verb ->
      if which = 1 ->
        fa i := 1 to N ->
          fa j := 1 to M -> writes(A[i,j], " ") af
          write()
        af
      [] else ->
        fa i := 1 to N ->
          fa j := 1 to M -> writes(B[i,j], " ") af
          write()
        af
      fi
    fi
  end
end Jacobi
```

D Comparison of Code Sizes

	Sisal	SR	C
Matrix Multiplication	19	51	92
Jacobi Iteration	58-62	136	296
Adaptive Quadrature	36	62-99	210
LU Decomposition	49-76	105	195
Mandelbrot	31	90	123
Total	235-266	549	1142

Figure 16: Lines of code.

Figure 16 summarizes the size of the programs in each of the languages. The count does not include blank lines and comments. As expected, the higher-level the language, the fewer the lines of code. The Sisal source is 2-3 times smaller than the SR source, which is 2-3 times smaller than the C source.

Sisal is smaller for three reasons: no variable declarations, no I/O code, and implicit parallelism. Variables are implicitly declared in Sisal, and they are explicitly declared in the other two language; variable declarations account for between 5 and 20 lines. The lack of I/O in Sisal is not an advantage, so while the code is smaller, the program is not as flexible or powerful. The effect of implicit parallelism is discussed in Section 2 of the paper.