217102

UCRL-JC-107491
PREPRINT

Implementing Functional Languages to Exploit Locality

Rich Wolski
John Feo
and
David Cann
Lawrence Livermore National Laboratory
Livermore, CA

This paper was prepared for submittal to
the Third IEEE Symposium on Parallel
and Distributed Processing Conference
Dallas, Texas, December -5, 1991

June 3, 1991

Implementing Functional Languages to Exploit Locality

Rich Wolski, John Feo, and David Cann

*Computer Research Group (L-306), Lawrence Livermore Nat. Lab.,
P.O. Box 808, Livermore, CA 94550*

Abstract: *In the quest for high performance, no obstacle has been as persistent or unyielding as memory latency. It was hoped that dataflow's fine-grain asynchronous model of execution might defeat the memory latency problem. Unable to realize efficient fine-grain systems, the dataflow community is now studying medium-grain and coarse-grain implementations which, like conventional execution models, suffer the effects of memory latency. In this paper, we describe a functional language implementation that automatically exploits locality on cache-coherent multiprocessors. Our system achieves performance improvements reaching 20% for some programs. This study lends further support to the superiority of the functional paradigm for parallel processing.*

1. Introduction

In the quest for high performance, no obstacle has been as persistent or unyielding as memory latency. Computer scientists have devised countless hardware and software schemes to ameliorate its effect, including: compiler-time reorganization of instruction streams, multiple functional units, cache memories, hierarchical memory systems, hierarchical processor topologies, microtasking, multiple-thread architectures, and dataflow. Parallel computing systems have further exasperated the problem. As systems grow, so do the physical distances between processors and memories. Physicists speculate that the hard limit for on-chip to off-chip memory delay is approximately 1 to 10 [Stone91]. Over the next decade we can expect to see ratios in the 1 to 20 or 1 to 30 range. Effectively exploiting large multiprocessor systems requires a careful balance between parallelism and locality.

It was hoped that dataflow's fine-grain asynchronous model of execution might defeat the memory latency problem. Since this model naturally identifies all instructions that can execute, there are always a large number of instructions ready to fire. It was thought that processors would be kept continually busy, effectively hiding memory latency. Unfortunately, fine-grain dataflow exposes too much parallelism [Bohm85, Culler90]. System resources are quickly swamped. Unable to develop an effective throttling mechanism, the dataflow community is now studying medium-grain and coarse-grain implementations which, like conventional execution models, suffer the effects of memory latency.

Sisal [McGraw85] is a general-purpose functional language being developed at Lawrence Livermore National Laboratory and Colorado State University. The semantics of functional languages adheres to the principals of mathematics. Sisal programs are a set of mathematically sound, side effect free functions. The programmer defines "what" is to be solved and not "how" it is to be solved. The Sisal compiler and runtime system are responsible for executing the program on the target architecture in the most efficient manner. The challenge, therefore, is to build machine specific Sisal systems that can effectively exploit the particular hardware of each target machine. On cache-coherent multiprocessors, the language system must tightly control where instructions execute; otherwise, programs will suffer a large number of cache misses resulting in poor execution times.

Current Sisal implementations for shared-memory multiprocessor systems employ a medium-grain, dynamic dataflow execution model [Feo90]. Task scheduling occurs at runtime without global control or knowledge of prior scheduling decisions. Consequently, locality is not exploited. In this paper, we report the changes we made to the Optimizing Sisal Compiler (OSC) and runtime system to automatically exploit locality on cache-coherent multiprocessors. These changes improved performance, but increased the complexity of both the compiler and the runtime system. The question we want to answer is "does the gain in performance justify the greater compiler complexity and runtime overhead?" As we shall see, the answer to this question is "Yes!"

In sections two and three we describe the original Sisal implementation and the changes we made. Section four compares the performance of the original and enhanced systems on four example programs run on a Sequent Symmetry (a cache-coherent multiprocessor). In section five we consider what we have learned and discuss future work.

2.0 The Original System

To date, Sisal implementations exits for a variety of shared-memory multiprocessor systems, including the Sequent Balance and Symmetry, Encore Multimax, Alliant FX/80, and Cray X-MP and Y-MP. These implementations are highly efficient. Sisal programs now run on these machines as fast as equivalent Fortran programs [Feo90, Cann90, Cann91a, Cann91b]. With the exception of several optimizations to improve vectorization on the Alliant and Cray architectures, all the implementations are essentially equivalent. Of greater interest is the fact that the Sisal programmer need not be concerned with the differences in the machines' architectures.

2.1 The Compiler

The Optimizing Sisal Compiler (OSC) translates Sisal source code into a hierarchical dataflow graph and then applies a series of optimizations. Ultimately, the optimized graph is translated into C and the native C compiler is called to generate machine code. The Sisal compiler goes to extreme effort to eliminate the data copying that is so prolific in functional language implementations that religiously adhere to side effect free semantics. In general, a simple reordering of operations combined with the insertion of code to precompute aggregate sizes eliminates the need for most copying while preserving program semantics [Feo90]. Runtime reference counting is used when compile time analysis fails. The reordering of operations also reduces the number of reference count operations required to preserve and recycle storage at runtime. We have shown that all these techniques are on the average 98% successful [Cann91b]. Besides optimizing and compiling the code, the compiler is responsible for packaging **for** expressions and inserting code to build activation records and invoke parallel execution. A **for** expression is a loop without carried dependencies of any kind; the individual iterations can execute in parallel in any order. The compiler and the runtime system are responsible for organizing the iterations into subtasks best suited for the hardware of the target machine.

2.2 Parallel loop support

To reduce overhead, the runtime system makes only modest demands of the host operating system. At program initiation, a command line option specifies the number of operating system processes to be instantiated. We call these processes workers, and their number remains constant throughout execution. We assume that the host operating system will bind the workers to the same processors for the duration of the program, regardless of preemption and system load. The workers themselves act like drones in a hive: they repeatedly carry out work on behalf of the queen (the program). The work takes the form of lightweight threads. The Sisal compiler packages the threads into parameterized functions, and inserts code to distribute the work via a shared run-queue. Access to the run-queue is on a first-come first-serve bases, so the pattern of work distribution across the machine is random.

The number of threads formed from a **for** expression is a command line option which defaults to the number of workers. Except in expressions with unequal amounts of work in each iterations, we have found no advantage in creating more threads than workers. For example, when smoothing a 1000x1000 byte image using 10 workers, the system breaks the work load into 10 chucks (each of which smooths 10,000 bytes of the image). The expectation is that each worker will pick up only one thread, resulting in an even distribution of the work. In rare circumstances a worker will fail to pick up a thread, causing a load imbalance. Since the run-queue is shared, there is no guarantee that a worker will consistently process the same portion of the image on successive iterations. Obviously, this thwarts the exploitation of locality and was the first part of the system that we changed.

The system described above controls parallelism according to a master-slave model. The master is responsible for distributing parallel work, while the slaves wait for activation records describing the various threads to arrive in the shared run-queue. They then race to extract the work, and complete the specified computations. Then they return for more work. The activation records identify the compiler packaged functions to be executed, the required arguments, and the domain of execution (iteration subranges, etc.). Returning to the previous example, the master would initialize 10 activation records, link them to form a list, and equeue the list on the run-queue (see Figure 1). If the compiler can determine that the bounds of a parallel loop do not change throughout the computation, it introduces code to build and initialize the list only once. The list is saved and re-used whenever the loop is executed. This is an important runtime optimization which saves the master from allocating, initializing, and deallocating the list of activation records more than once.

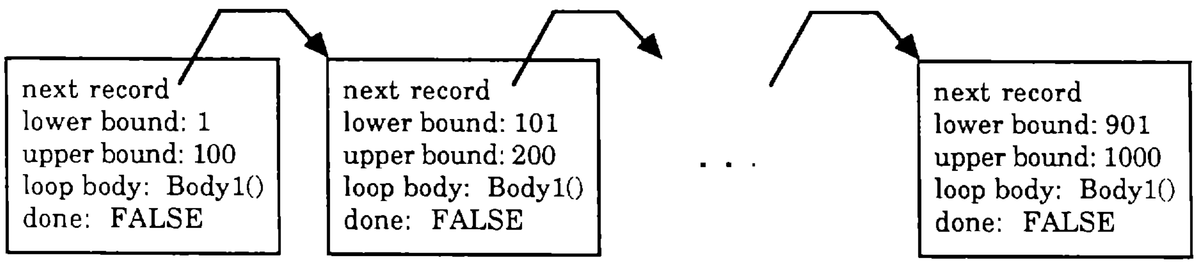

Figure 1 – A List of Activation Records

Figure 2 shows the pseudo-code for that portion of the master and slave processes from the original Sisal implementation that manipulate the shared run-queue. Note that both the master and the slaves require exclusive access to the run-queue, which is realized by a single global lock. The slaves "spin" waiting for the master to change the head of the run-queue to point to the first activation record. After the activation records are enqueued, all the slaves rush to acquire access and grab a record. As an optimization, the master holds out the first record and executes it first. Then, while waiting for the other threads to complete, the master attempts to dequeue records from the run-queue as if it were a slave. Each record has a done flag (initialized to false). When each thread completes the done flag in the respective record is set to true. When all the done flags for the loop are true, the master continues on to the next phase of computation. Note that the master only temporarily assumes the role of a slave. All the workers (master and slaves) use their own stacks to execute the threads. Because Sisal does not allow the intermediate results of a parallel loop to be visible outside the defining iteration, deadlock among the workers is not possible. This stack-based system is different from the system described in [Feo90]. It eliminates the need for a *blocked queue* and context switches.

2.3 Storage subsystem

In addition to managing parallel loop execution, the runtime system supports a dynamic storage subsystem optimized for the data structures found in Sisal. The original system is a boundary-tag system with multiple entry-points to a circular list of free blocks. A worker picks an entry-point on the basis of its own integer identifier. The free list search is parallelized for increased performance as is the deallocation process. To increase the speed of allocation and deallocation, an exact fit caching mechanism lies between the boundary tag system and the requesting worker. It uses a working set of different sizes of recently freed blocks, where blocks of the same size form a sublist. To eliminate synchronization overhead, each worker has its own cache. This results in

```
Master()
  if activation records have not been built then
     build activation records
  end if
  lock(shared run queue lock)
  enqueue all records except the first
  unlock(shared run queue lock)
  execute the routine specified in the first record
  set the done flag in the first record to TRUE
  do while there is a done flag that is FALSE
     Slave_process()
  end do


Slave()
  do forever
     Slave_process()
  end do


Slave_process()
  do while the run queue is not empty
     lock(shared run queue lock)
     if there is a record on the run queue then
        dequeue a record
        unlock(shared run queue lock)
        call the routine specified in the record
        set the done flag in the record to TRUE
     else
        unlock(shared run queue lock)
     end if
  end do
```

Figure 2 – Pseudocode for the Master and Slave processes

faster execution, but can cause exact-fit misses if the size desired is in another worker's cache.

To benefit later discussions, we briefly describe how the original system builds a matrix in parallel. In Sisal, such a structure is by definition a vector of vectors. This allows the definition of ragged arrays and improves copy elimination analysis, but at the cost of increased storage management. When an array is built by a **for** expression, the master allocates the outermost vector before parallel execution begins. Then the slaves allocate their range of row vectors on the fly. Later, the last user of the matrix recycles all the pieces, making sure that each deallocated vector lands in the exact-fit cache of its originating slave. This helps to evenly distribute the free storage and reduce contention for the shared boundary tag subsystem during subsequent allocations attempts. For the smoothing example, the matrix created on iteration i is read on iteration $i + 1$ and then deallocted. The freed storage is used on iteration $i + 2$ to build the new matrix. We will see that this unnecessary memory management activity has high execution overhead and thwarts the exploitation of locality.

3.0 Modifications to exploit locality

Because Sisal insulates the programmer from the target machine, it places the burden for mapping and tuning each program on the language system. The key to automatically exploiting locality in Sisal programs is to schedule the producer and consumers of a data object on the same processor. In the case of an aggregate object, we want to distribute the aggregate's components across the processors, and schedule the producer and consumers of each component to the same processor. Throughout the following discussion, we will use the term *data object* to refer to a primitive data object; that is, a non-aggregate object. Since the original Sisal implementation schedules producers and consumers dynamically without knowledge of previous scheduling decision, we implemented a new scheduling policy.

Many of the problems commonly encountered in scientific computing iterate through sequences of parallel loops. The iterative or outer control loop executes until some convergence criterion is satisfied. On each iteration, the inner parallel loops is invoked to realize the actual computational work. In general, each successive parallel loop takes the results of the previous parallel invocation as input. For example, consider the problem of image smoothing. The main iterative loop invokes a parallel loop and then tests for the satisfaction of convergence. Each successive parallel loop refines the results of the previous iteration. To exploit locality in this example, it is important to insure that each processor refines the same portion of the image on each iteration. The shape of each region, or *partition*, of the array effects the amount of information that must be communicated between regions [Abraham90]. While previous work on the topic of partition shape demonstrates a theoretical advantage to using non-rectangular polygons for certain problems, evidence indicates that the extra control overhead required negates any reduction in communication overhead. For now, we have chosen not to change the existing Sisal partitioning mechanism (which partitions objects into rectangles).

To summarize, we decided to divide parallel loops into rectangular partitions and then schedule the dependent partitions from successive iterations on the same processor. Consequently, intermediate results will reside in the appropriate processor caches between iterations. The enhanced system preserves data locality only for applications in which data generated by the i-th iteration of the producer are used by the i-th iteration of the consumer. We refer to this as *i-to-i locality*. While a more substantial compile-time analysis of array subscripts will allow us to support more complicated producer–consumer relationships (for example, matching the i-th iteration of the producer with the j-th iteration of the consumer), we believe that most scientific codes can benefit from an i-to-i matching. Moreover, before making wholesale changes to the compiler, we wanted to gain a feel for the potential performance improvements that could be had through subscript analysis. The results of this study indicate that a more general system that supports i-to-j locality is warranted.

3.1 Binding Work to Workers

To bind parallel work to specific workers, we chose to implement a distributed run-queue (one queue per worker). We then modified the runtime system to schedule related producer and consumer computations to the same worker. In the enhanced system, the master builds the activation records as in the original system, but now distributes the records among the various worker run-queues in a predefined order—an order preserved across iterations. Each worker first checks its own run-queue for an activation record, and then, if empty, checks the shared queue for work. Figure 3 shows the modified master and slave pseudocode for the distributed run-queue version. Changes from the original version are shown in italics.

3.2 Preallocation of aggregate data structures

Even though our distributed run queue implementation enabled Sisal to exploit cache locality for a large range of scientific applications, initially we did not see the performance improvement that we had anticipated. After studying the problem for some time, we concluded that the overhead associated with the runtime memory management functions was obscuring the performance improvements. As explained in Section 2.2, OSC allocates data objects on demand, and deallocates data objects as soon as all their consumers have completed. Further, OSC implements all objects as collections of data structures linked by pointers. For example, a two-dimensional array consists of three data

```
Master()
  if activation records have not been built then
     build activation records
  end if
  do for each slave
     lock the slave's queue
     enqueue a record in the slave's queue
     unlock the slave's queue
  end for
  if there are any remaining records then
     lock(shared run queue lock)
     enqueue all records except the first
     unlock(shared run queue lock)
  end if
  call the routine specified in the first record
  set the done flag in the first record to TRUE
  do while there is a done flag that is FALSE
     Slave_process()
   end do


Slave_process()
  do while this slave's queue and the shared run queue are not empty
    if there is a record in this slave's queue then
       dequeue that record
    else
       lock(shared run queue lock)
       if there is a record in the shared run queue then
          dequeue a record
          unlock(shared run queue lock)
       else
          unlock(shared run queue lock)
           goto end do
       end if
    end if
    call the routine specified in the record
    set the done flag in the record to TRUE
  end do
```

Figure 3 – Pseudocode for the enhanced Master and Slave processes.

structures for each row (an array row header, and array storage header, and the storage itself) and another set of three structures for a vector of row pointers which associates the rows at the outer dimension. Allocating and deallocating an entire aggregate object can be computationally expensive, and can force the program data out of caches as memory management data structures are accessed. We realized that we would have to eliminate most, if not all, of the memory management operations from the inner parallel loop.

Fortunately, Sisal's functional semantics make it possible for OSC to prebuild data objects. In the smoothing example discussed above, OSC analyzes the array indexing patterns and data dependencies to determine that a 1000x1000 byte image is produced during each iteration. Furthermore, each iteration only requires the 1000x1000 byte image produced during the previous iteration to form the new image. Thus, the compiler introduces code to allocate two 1000x1000 byte arrays before the process begins and at the end of the iterative loop to switch between the two arrays. Consequently, all memory management operations occur outside the work loop. Of course, not all applications conform to a model in which data structures remain uniform in size throughout the computation, but when they do, or if the size shrinks from a maximum that can be deduced (as in Gaussian Elimination), the semantics of Sisal provides enough information for storage preallocation.

Once we added this optimizations to prebuild two copies of aggregate objects and switch between them, we observed the performance benefits from locality that we expected. For example in the case of Gauss-Jordan without pivoting (discussed in Section 4.1) we originally observed only a 12% improvement using the enhanced system. After implementing the memory optimization we realized a 20% improvement. We hypothesized that the 8% loss in performance was either a result of the extra execution time required to do memory management, or the extra space in the cache that is required (for both code and data). Figure 4 details a comparison of the difference in execution times for OSC running Gauss-Jordon without pivoting. The solid columns show the difference in execution time for OSC without memory management (using prebuilt data structures). The hashed columns show the execution time difference for OSC with memory management. All the data is for Gauss-Jordon without pivoting. Since the time differences are relatively equal, we surmise that the loss in performance is due to extra execution overhead and not cache interference. We feel it is important to note that memory management overhead can easily overshadow the benefits of exploiting data locality.

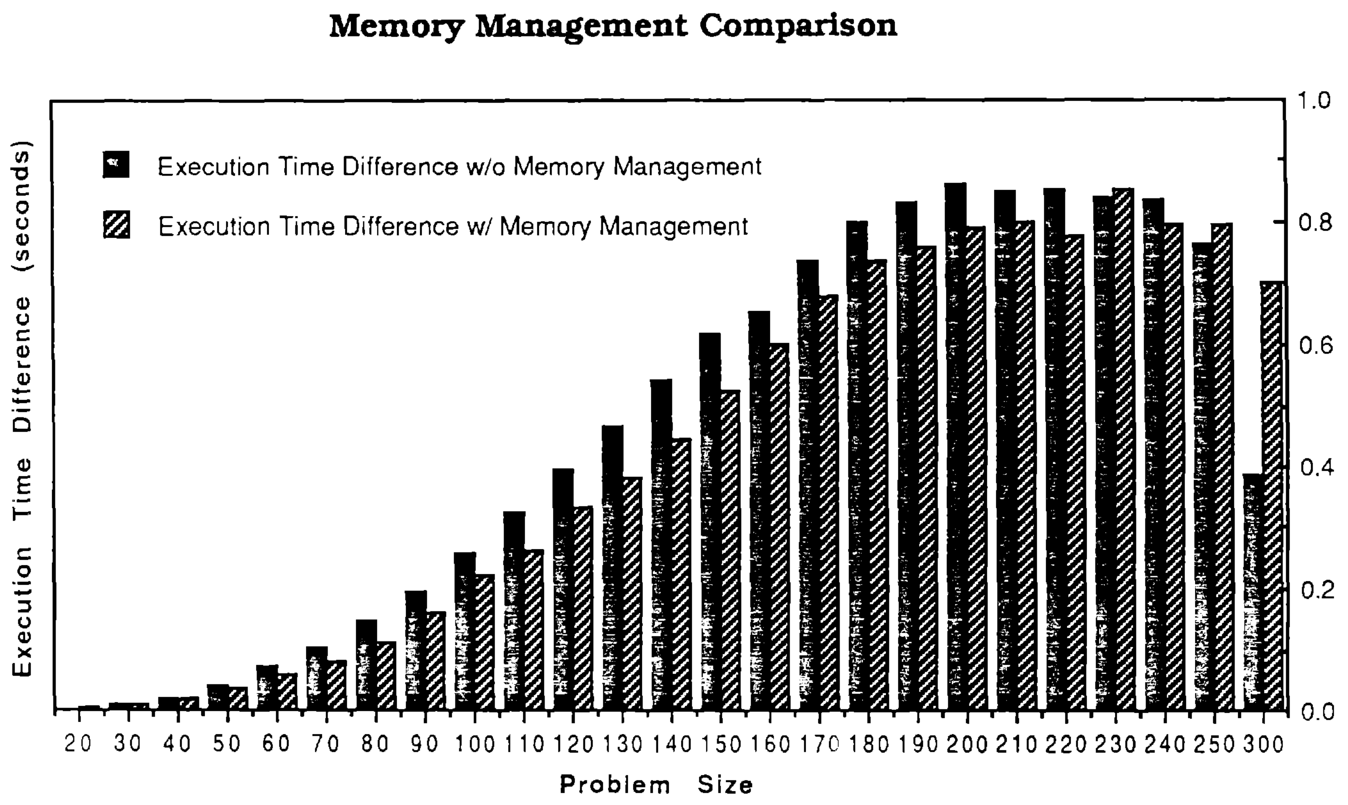

Figure 4 -- Memory Management Execution Times

4.0 Results

To evaluate the cost of exploiting locality we tested the enhanced system on four applications: Gauss-Jordan with and without pivoting, a 5-point stencil, and a particle-in-cell code. The four applications were written in double-precision arithmetic. We ran all the experiments on the Sequent Symmetry at Southern Methodist University. This machine has 20 Intel 80386 processors, 32MB of main memory, and a 32K copy-back cache per processor. The system includes Weytek floating-point coprocessors and automatically maintains cache coherency. For each experiment we used 10 processors and to account for normal variations in execution time we took the average over 5 runs. We report the percentage improvement in execution time of the enhanced version of OSC over the original system which we define as:

$$improvement = \frac{(original\ execution\ time - enhanced\ execution\ time)}{original\ execution\ time}$$

Owing to the functional semantics of Sisal, we could not prescribe a particular order of execution or organization of subtasks. All we could do was express the algorithms and rely on the compiler and runtime system to organize the computation in such a manner to exploit the Symmetry's hardware to best advantage.

4.1 Gauss-Jordon Linear System Solver without Pivoting

The first program we chose to examine was an in-place Gauss-Jordon linear system solver without pivoting. This algorithm solves the linear system

$$A \bullet x = b$$

Since this is a regular problem with no communication between partitions (only the pivot row is shared), we expected to see a large performance improvement for this program. We considered systems from 10 to 400 equations. The Sisal program is comprised of an outer sequential loop executed n times and an inner parallel loop executed $O(n^2)$ times, where n is the number of equations. On each iteration of the outer sequential loop, the master selects the next pivot row and passes it to the inner parallel loop. Each worker then reduces its slice of A and b by that row. Since we ran on 10 processors, each slice of A is $n/10$ rows by n columns and each slice of b is $n/10$ elements. Notice that only the pivot row is fetched from main memory; the rest of the system is distributed among the processor caches.

Figure 5 shows the percentage improvement for Gauss-Jordon without pivoting as a function of problem size. At first, the improvement in performance increases as the problem size grows, increasing the ratio of execution time versus loop overhead. The improvement then levels off at approximately 20%. As the problem size grows, it eventually spills from the cache increasing the number of accesses to main memory. Consequently, the observed improvement of the enhanced over the original system decreases beginning with problem size 130x130. For this size problem we are exploiting 84% of the cache. Recall that Sisal maintains both the current and previous copy of A and b array; thus for a 130x130 size problem, each worker owns $2 \bullet (13 \bullet 130 + 13)$ double-precision elements for a total of 27KB.

2.2 Gauss-Jordon Linear System Solver with Full Pivoting

Next we added full pivoting to the Gauss-Jordon solver. As each worker reduces its slice of A on iteration i, it records the largest element in the slice not in a previously selected pivot row. On completion of the reduction phase each worker passes the largest element it found and the row index of that element to the master. The master then selects

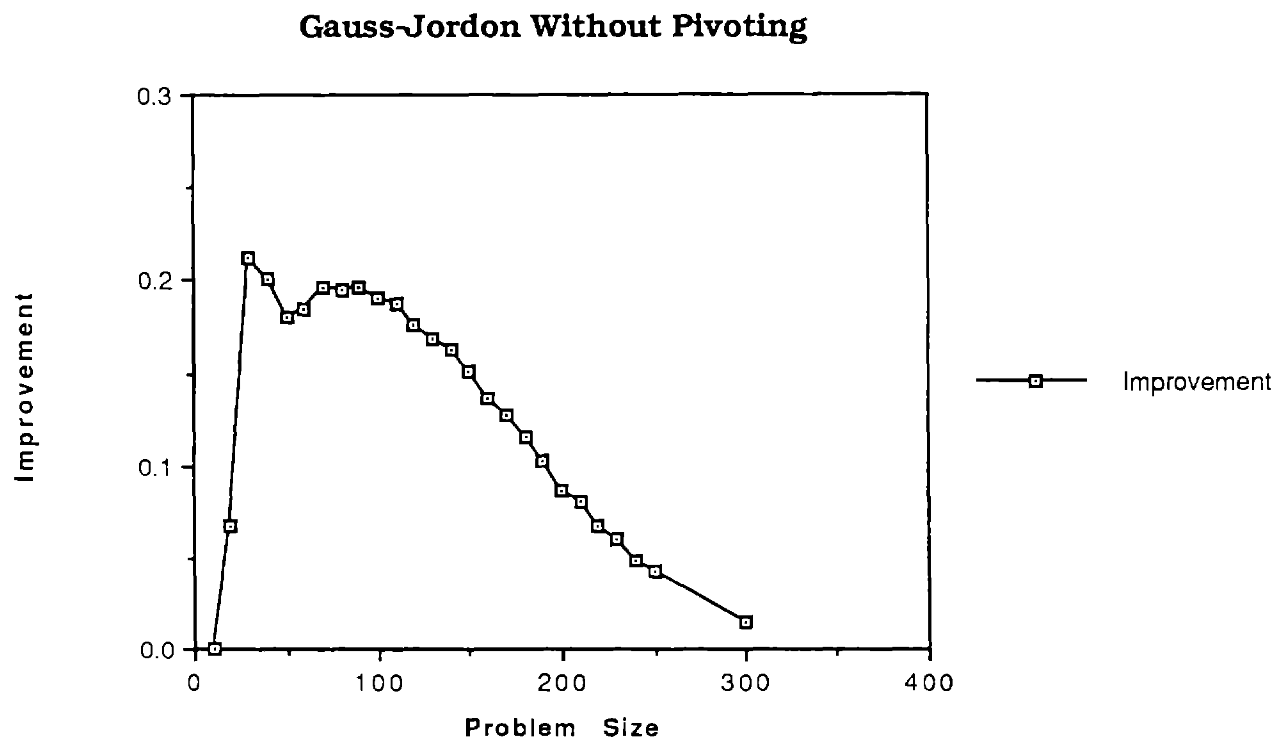

Figure 5 -- Gauss-Jordon Non-pivoting Improvement

the new pivot row, say row j, by finding the largest of these elements. Before initiating the next reduction phase, the master places the largest element on the main diagonal by interchanging rows j and $i + 1$. Note that the selection of the pivot row in the Sisal algorithm is distributed over the workers, thereby, minimizing communications. The only values written and read from main memory are the 10 maximum values, the 10 row indices, and rows i and j.

Figure 6 shows the percentage improvement for Gauss-Jordon with full pivoting. We see the same shaped curve with a maximum improvement of approximately 12%. We expect a smaller percentage improvement in the case of full pivoting contrasted with no pivoting because the former increases the number of instructions executed without increasing proportionally the opportunities to exploit locality. Figure 7 compares the differences in execution times of the two algorithms. Since the heights of the columns are essentially equivalent, we conclude that the advantage of the enhanced system over the original system is constant. Thus, the loss in improvement in the pivoting case is due primarily to increase execution times and not increased bus traffic.

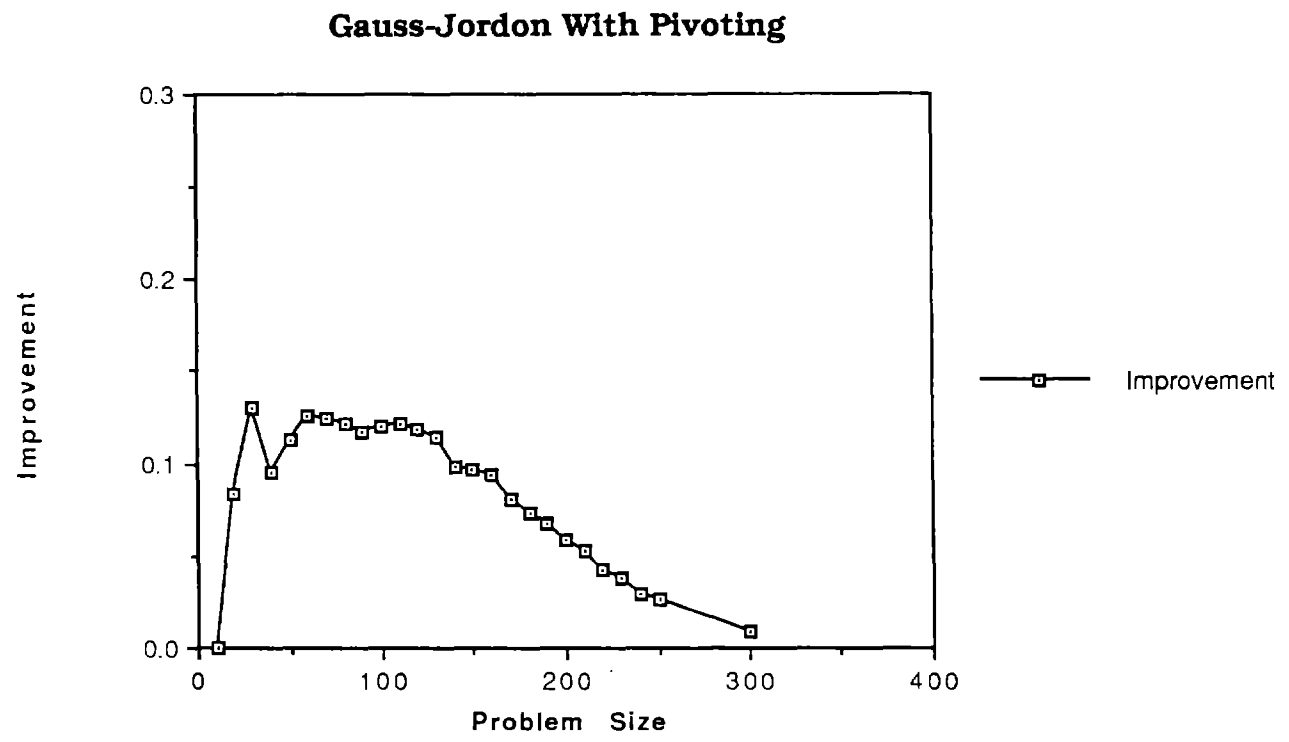

Figure 6 -- Gauss-Jordon Pivoting Improvement

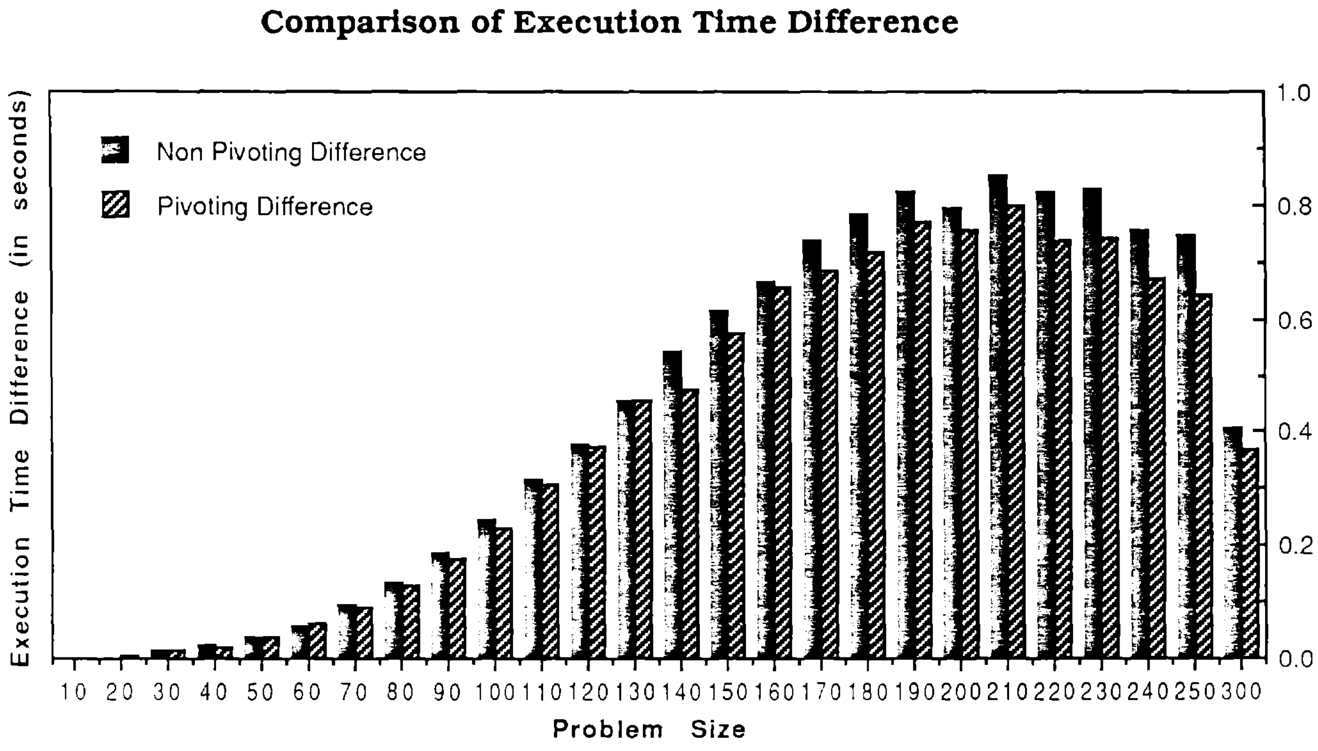

Figure 7 -- Execution Time Difference Between Pivoting and Non-pivoting Gauss-Jordon

2.3 *A Five-Point Stencil*

The third problem we studied was a five-point stencil. Stencils of various complexities appear in many scientific computations. Here we consider a simple stencil which calculates the weighted average of the center element with its north, south, east, and west neighbors. The algorithm is comprised of an outer iterative loop and a parallel inner loop. The outer iterative is executed until a convergence criterion is satisfied. The inner parallel loop computes

$$A[i,j] = \frac{(4.0 \bullet A[i,j] + A[i-1,j] + A[i+1,j] + A[i,j-1] + A[i,j+1])}{8.0}$$

for $2 \leq i,j \leq n$. Note that the boundary elements do not change.

The program exhibits less data locality than the previous two examples. To smooth a row of A requires the row above it and the row below it; but since the Sisal runtime system slices A into chunks, only the rows on slice boundaries have to be communicated between processors; consequently, each processor reads and writes two rows per iteration. Because the computation per element is short, for thin chunks, communication delays form a significant portion of the overall execution time.

For this example program, we varied the number of rows from 20 to 400. By increasing the number of rows, we made each slice of the array successively thicker, increasing the proportion of computations to communications. Since increasing the number of columns does not change the proportion of computations to communications, we kept the number of columns fixed at 80. Figure 8 summarizes the results for the five-point stencil. We see the same shaped curve as before, but with a maximum improvement of 9%. As before we use approximately 80% of the cache before beginning to spill out; however, the decrease in improvement as the problem size grows is more gradual than in Figures 5 and 6. We have no explanation for this disparity.

2.4 *Particle-in-Cell*

The final program we chose to examine was a particle-in-cell calculation. It is an extension of a PIC code developed at Los Alamos National Laboratory. The program simulates the evolution of a system of charged particles confined to a rectangular cylinder. The particles are subject to only electromagnetic forces. A two-dimensional grid is superimposed on the cylinder and the electric potential is calculated at each grid

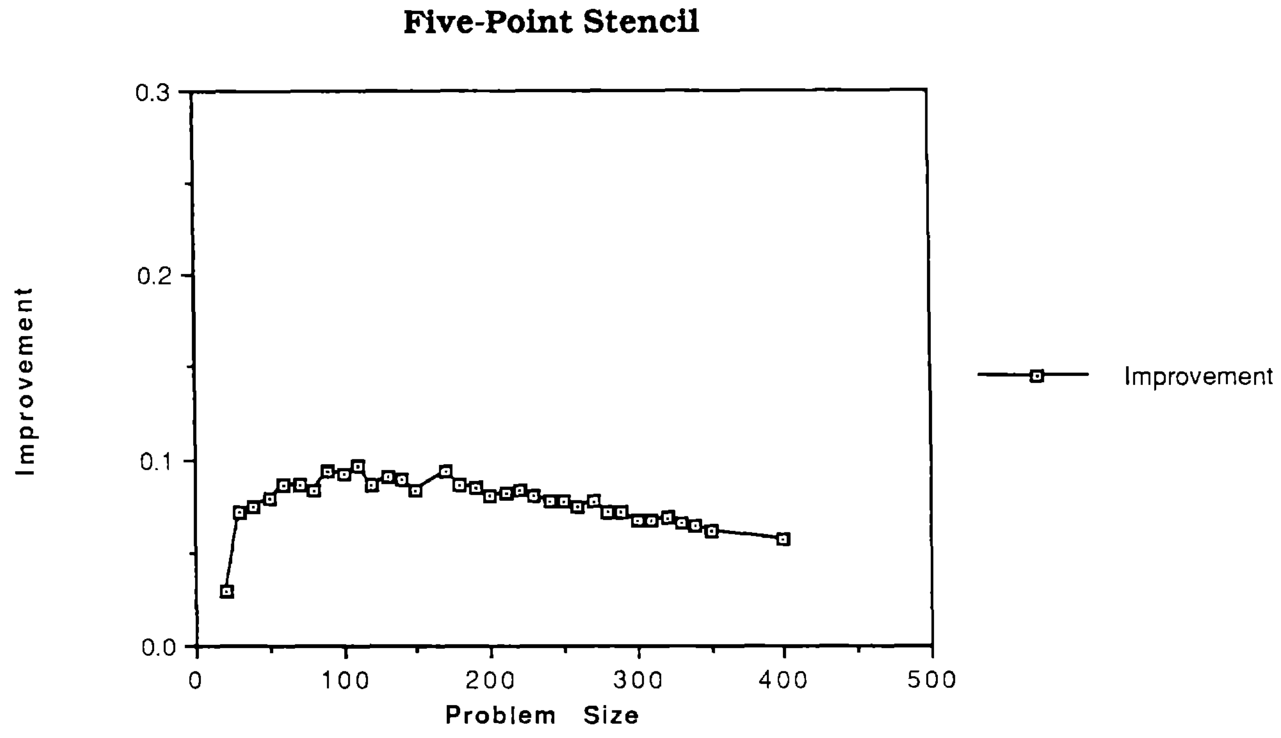

Figure 8 -- Five-point Stencil Improvement

intersection. Particles contribute to and are affected only by the potential at the four grid points defining the enclosing cell. At each time step, the algorithm calculates the acceleration, velocity, and new position of each particle. It then computes the new electric potential at each grid intersection. The program's parameters are: the number of time-steps, the number of particles, and the grid dimensions.

We wrote the Sisal code such that the particle array is evenly distributed across the processors. Each processor calculates the contribution of its particles to the grid. The master then merges these grids to compute the composite force at each grid intersection. The workers read this grid to calculate the new acceleration, velocity, and positions of their particles. Thus for each iteration each worker writes and reads a grid from main memory. Note that the particles never move from their assigned processor. In this implementation the grids move, but the particle data structures remain fixed. Although the computation per slice is large, if the grid partitions are too fine, communication costs will dominate.

Figure 9 shows 100 time-step runs for 500, 1000, and 2000 particles on a 25x25 point grid. For these numbers of particles, a 25x25 grid is fine enough to affect, but not dominate, the program's overall execution time. The percent in improvement of the enhanced over the

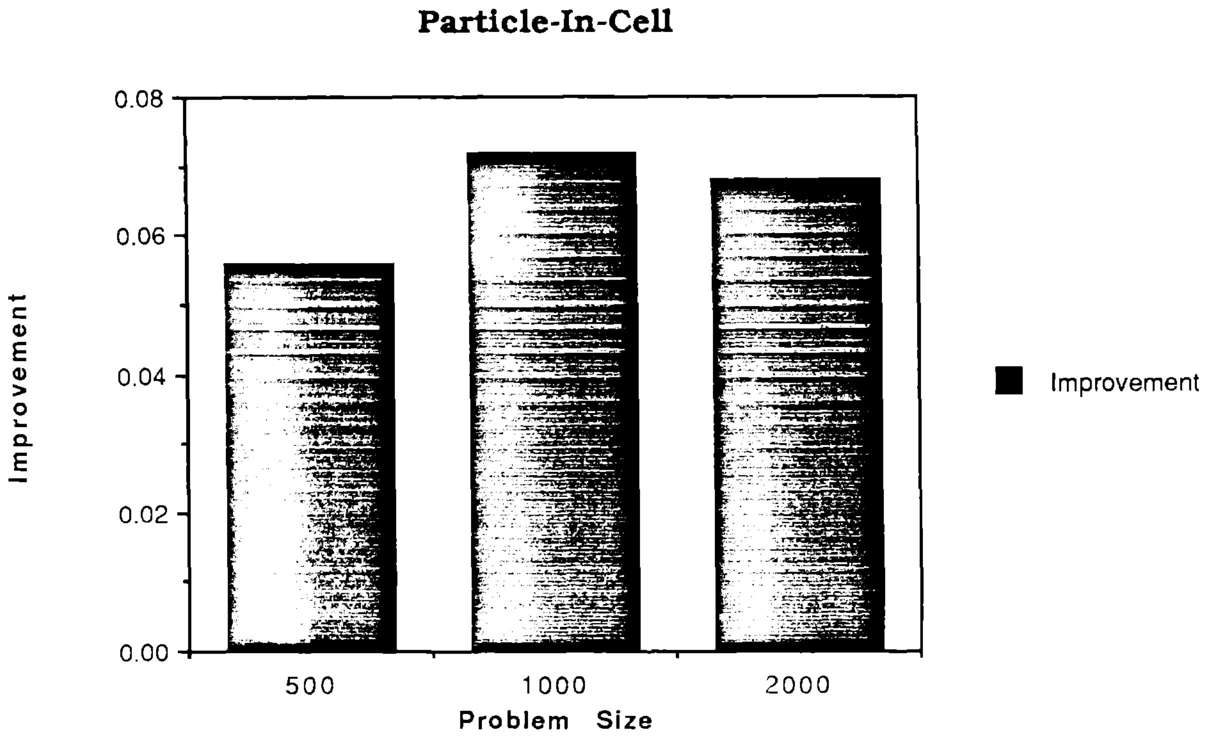

Figure 9 -- Particle-In-Cell Improvement

original version is approximately 8%. Of the four programs we tested, this is by the far the largest and most complicated. It is a real code that exercises all aspects of the Sisal compiler and runtime system. Since we realized a nontrivial improvement, we are encouraged to adopt the enhanced system and pursue more sophisticated techniques for exploiting locality.

5.0 Conclusions and future work

The results of our study show that Sisal language systems can exploit locality and that the improvement in performance outweighs the increase in the complexity of the compiler and runtime system. Consequently, we believe a more comprehensive effort to exploit locality in Sisal and other functional languages is warranted. To achieve the same degree of locality in imperative parallel programming, is very difficult [Warren91]. This study lends further support to the superiority of the functional paradigm for parallel processing. Remember that all we did was express the algorithms, the rest was done automatically by the compiler and runtime system.

We see little reason to support nonrectangular partitions. The overhead of managing such partitions in the Sisal system would be great, as predicted for any real system by Abraham and Hudak [Abraham90]. The computation to communication ratio for nonrectangular partitions would have to be substantially better than the ratio for rectangular loops to justify the increase management costs. There is also the question of increased scheduling costs as the number of partitions increase. Presently, the enhanced system supports only *i*-to-*i* producer–consumer relationships. We recognize the importance of supporting general producer–consumer relationships, and believe that the subscript analysis techniques used by automatic parallelizing compilers can help solve the general problem. We are currently investigating the applicability of these techniques.

While most commercially available multiprocessor systems have a two-level memory (cache and main memory), future large-scale systems will almost certainly employ more than two levels of memory. Much more sophisticated methods than the ones described here will be required to effectively exploit locality on these machines. For example, the BBN TC2000 has cache, local, shared, and interleaved memory. We are currently developing and testing a prototype system based on the distributed run-queue model for the TC2000. We hope to report on this work shortly.

Acknowledgements

We would like to thank Southern Methodist University for use of their Sequent Symmetry. This work was supported (in part) by the U.S. Department of Energy under contract number W-7405-Eng-48 to Lawrence Livermore National Laboratory, the Mathematic Sciences Program of the Office of Energy Research (U.S. Department of Energy), and Lawrence Livermore National Laboratory's Massively Parallel Computing Initiative.

References

[Abraham90] Abraham, S.G. and Hudak, D.E. Compile-time Partitioning of Sequentially Iterated Loops to Minimize Cache Coherency Traffic, *Proc. ACM Int. Conf on Supercomputing*, 1990.

[Bohm85] Bohm, A.P.W., J. Gurd, and J. Sargeant. Hardware and Software Enhancement to the Manchester Dataflow Machine. *Proc. IEEE Spring Computer Conference,* February 1985, pp. 420-423.

[Cann91a] Cann, D.C. Vectorization of an Applicative Language: Current Results and Future Directions,*Proc. COMPCON 91,* San Francisco, CA, February 25, 1991, pp. 396-402.

[Cann91b] Cann, D.C. Retire FORTRAN? A Debate Rekindled, submitted to *Proc. Supercomputing 1991,* Albuquerque, NM, November, 1991.

[Cann90a] Cann, D.C. J.T. Feo, and DeBoni, T.M., SISAL SISAL 1.2: High Performance Applicative Computing, *Proc. 2nd IEEE Symposium on Parallel and Distributed Computing,* Dallas, TX, December, 1990.

[Cann90b] Cann, D.C. and J.T. Feo. SISAL versus FORTRAN: A Comparison using the Livermore Loops. *Supercomputing '90,* New York, NY, November 1990.

[Culler90] Culler, D.E. *Managing Parallelism and Resources in Scientific Dataflow Programs*, MIT Technical Report LCS/TR-446 (Ph. D. dissertation), MIT, Cambridge, MASS, March 1990.

[McGraw85] McGraw, J. R. et. al. *Sisal: Streams and iterations in a single-assignment language, Language Reference Manual, Version 1.2.* Lawrence Livermore National Laboratory Manual M-146 Rev. 1), Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[Feo90] Feo, J.T. D.C. Cann and R.R. Oldehoeft, A Report on the SISAL Language Project, *Journal of Parallel and Distributed Computing*, vol. 12, **10** (December 1990), pp. 349-366.

[Stone91] Stone, H, Keynote Address, *IEEE 5th Int Parallel Processing Symposium*, Anaheim, CA, April 1991.

[Warren91] Warren, K.H. and Brooks, E.D. Gauss Elimination: A Case Study,*Proc.,COMPCON 91,* San Francisco CA, February 25, 1991, pp. 57-61.