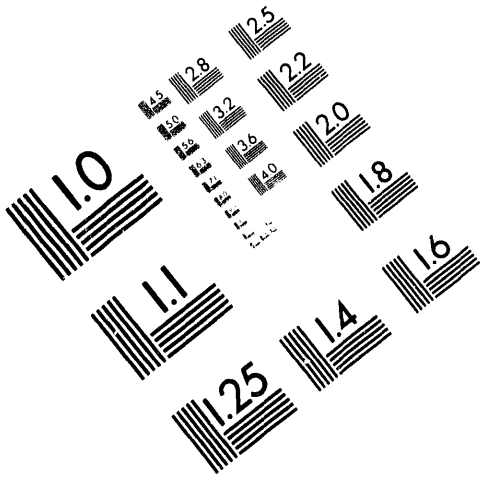
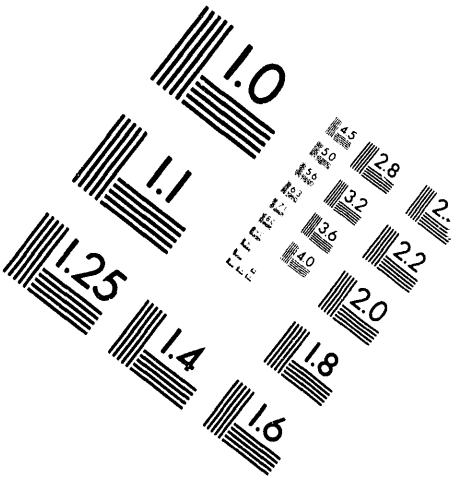
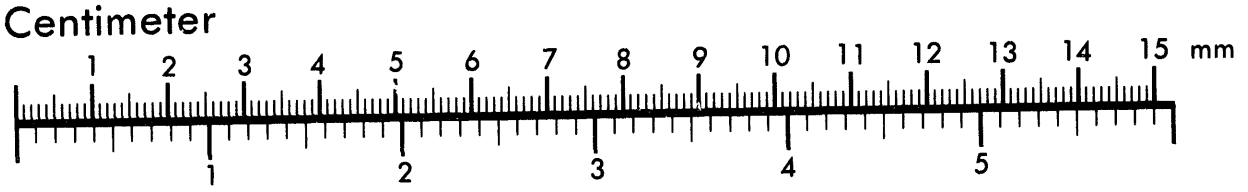


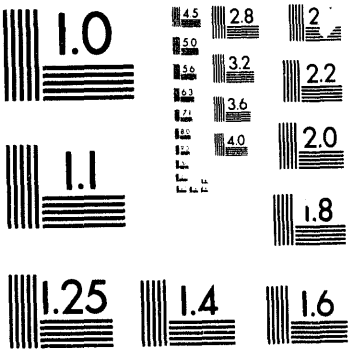
Association for Information and Image Management

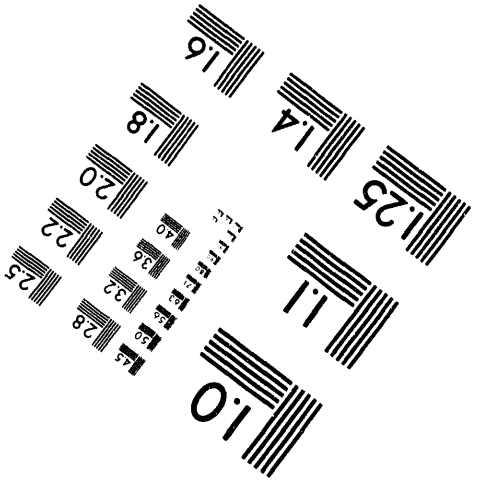
1100 Wayne Avenue, Suite 1100
Silver Spring, Maryland 20910

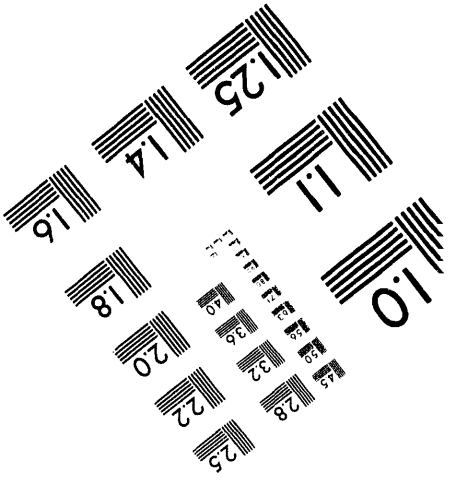
301/587-8202











MANUFACTURED TO AIIM STANDARDS
BY APPLIED IMAGE, INC.

1 of 1

Conf-9408147--1

UCRL-JC-116015
PREPRINT

Realizing Parallel Reduction Operations in Sisal 1.2

Scott M. Denton,
John T. Feo,
and
Patrick J. Miller

This paper was prepared for submittal
to the Working Conference on Parallel
Architectures and Compilation Techniques
(PACT '94)

August 24-26, 1994, Montreal Canada

February 1994

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

Lawrence
Livermore
National
Laboratory

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Recycled
Recyclable

DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Realizing Parallel Reduction Operations in Sisal 1.2

Scott M. Denton, John T. Feo and Patrick J. Miller [1]

Computer Research Group
Lawrence Livermore National Laboratory
Livermore, CA 94550

Abstract: *A parallel job consists of sets of concurrent and sequential tasks. Often the tasks compute sets of values that are reduced to a single value or gathered to build an aggregate structure. Since reductions may introduce dependencies, most languages separate computation and reduction. For example, Fortran 90 and HPF provide a rich set of predefined reduction functions, but only for extant arrays. Sisal 1.2 is unique in that reduction is a natural consequence of loop expressions. Unfortunately, the language supports only seven reduction operations. In this paper, we present compilation techniques that recognize pairs of for expressions in Sisal 1.2 as computation-reduction expressions. Our techniques work without any language or intermediate form extensions; however, we recognize only certain forms. We describe how we implement pairs of computation–reduction expressions as single parallel loops, and we present performance numbers that demonstrate the utility of our techniques.*

1.0 Introduction

Reduction operations are an important and essential component of parallel programming. Often the tasks of a parallel job compute sets of values that are reduced to a single value or gathered to build an aggregate structure. For example, a set of tasks may compute the temperature at each point in the state space, and then average the temperatures. Alternatively, the temperatures could be gathered into an array, or graphed. A recurring theme in particle codes is the calculation of forces between particles. Since the forces are symmetric, we want to calculate each force only once and then accumulate the forces on the affected particles. Because reduction operations occur frequently in application programs, they are good targets for optimization. The efficient expression and implementation of reduction operations can reduce the cost of parallel programming.

Reduction operators are functions of sets of values (arrays, lists, etc.). The memory used to store the result is shared by the tasks computing the individual values. Since reductions may introduce

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

dependencies, most languages separate the computation and reduction tasks. For example, Fortran 90 [1] and HPF [2] provide a rich set of predefined reduction functions, but only for extant arrays.

```
C  find the minimum value in z_array
   z_min = MINVAL(z_array)

C  return the first location of the minimum value in z_array
   z_min_loc = MINLOC(z_array)
```

Despite the lack of explicit memory, functional languages also support reduction operations. Haskell [3] provides reduction and accumulation operations on extant lists or list expressions.

```
-- compute the sum of the integers 1 through 10
sum[1..10]

-- return a table of the number of occurrences of each value
-- within bounds in list z
accumArray (+) 0 bounds [i := 1 | i <- z, (inRange bounds i)]
```

Sisal 1.2 [4] is unique in that reduction is a natural consequence of loop expressions. The reduction operation appears as a keyword in the returns clause of the *for* or *for initial* expression.

```
% find the minimum value in z_array
for z in z_array
returns value of least z
end for

% compute the sum of the integers 1 through 10
for i in 1, 10
returns value of sum i
end for
```

The Sisal compiler can overlap computation and reduction, and implement both tasks to take best advantage of the underlying architecture.

A short coming of all languages is that they support only a general set of predefined reduction operations. Many applications, however, require very specific reduction operations. The HPF Journal of Development [2] has suggested additional language features be added for user-defined reduction functions, and in the final section of this paper we discuss syntax for user-defined reductions in Sisal 90. Without special language features, a user must use extant loop forms to express the computation and reduction operations, and rely on the compiler to generate efficient code. For example, the Fortran D compiler [6] seeks to recognize reductions for optimization in traditional imperative code.

In this paper we present compiler techniques to identify pairs of computation–reduction expressions in Sisal 1.2. We describe how we manipulate the code's intermediate form to construct a single parallel loop similar to the loops constructed for the seven intrinsic reduction operations. Our techniques assume no language or intermediate form extensions; however, we recognize only certain forms. Section two presents the form of computation–reduction expressions we recognize, and the constraints that the expressions must satisfy. Section three illustrates the rewiring of the intermediate form, and discusses implementation issues. Section four presents performance numbers demonstrating the utility of our techniques. In section five, we discuss the syntax for user-defined reductions in Sisal 90, the analysis required to insure determinancy, and the possible implementations of different classes of reduction operations.

2.0 Computation–reduction expressions

The Sisal 1.2 language definition supports seven reduction operations: sum, product, least, greatest, array, stream, and catenate. The reductions may appear in the returns clause of *for* or *for initial* expressions. While useful, the reductions are inadequate. For example, finding the first location of the minimum value of an array cannot be expressed efficiently in Sisal. A programmer must either write two *for* expressions,

```
min_value := for x in A returns
             value of least x end for;
min_index := for x in A at i returns
             value of least i when x = min_value end for;
```

or one *for initial* expression

```
min_index := for initial
               i := 1;
               min_value, min_index := A[1], 1
             while i < array_size(A) repeat
               i := old i + 1;
               min_value,
               min_index := if A[i] < old min_value then A[i], i
                            else old min_value, old min_index
                            end if
             returns value of min_index
             end for
```

The first solution doubles the computation's overhead, and the second solution eliminates all parallelism.

The situation is more dire if we want to generate a set of values, and then count or accumulate the values of different types. For example, consider a set of n particles and m bonds. Each bond represents a force between two particles. We can calculate the forces in parallel, but must use a *for initial* expression to calculate the total force on each particle

```
Total_energy,
Force_update := for bond in 1, m
                  ii, jj := end_points(bond);
                  Bond_energy  := Energy(bond);
                  Force_record := force(ii, jj, Positions)
                returns value of sum Bond_energy
                        array of Force_record
                end for;

Force_out    := for initial
                  i := 0;
                  Forces := array_fill(1, n, 0.0)
                while i < array_size(Force_update) repeat
                  i  := old i + 1;
                  ii := Force_update[i].ii;
                  jj := Force_update[i].jj;
                  f  := Force_update[i].force;
                  Forces := old Forces[ii: old Forces[ii] + f;
                                       jj: old Forces[jj] - f ]
                returns value of Forces
                end for
```

On highly parallel computer systems, the presence of the *for initial* expression curtails the code's efficiency—an effect of Amdahl's Law. Notice that the size of the sequential code grows linearly with problem size. On medium or small systems, there may be insufficient memory to store the intermediate array of force records. The extra storage may increase the number of page faults and secondary memory accesses, diminishing performance.

Programmers writing in an imperative language do not face this problem. They can write a single parallel loop that includes a critical section to control access to the force array,

```
do ibond = 1, m
     call end_points(bond, ibond, ii, jj)
     f = force(ii, jj, ...)
     lock(Force_out)
          Force_out(ii) = Force_out(ii) + f
          Force_out(jj) = Force_array(jj) - f
     unlock(Force_out)
end do
```

Since the force calculation is typically much longer than the critical section, the concurrent tasks will contend for the lock infrequently. The code is parallel, efficient, safe, and minimizes memory use.

We have realized generated code similar to the imperative code, but without explicit locks, by extending the Sisal compiler to recognize pairs of computation–reduction expressions. The optimization is applied to pairs of *for* and *for initial* expressions that satisfy the following criteria:

1. the *for initial* expression does not depend on any descendant of the *for* expression,
2. the *for initial* expression depends on the *for* expression for only an array of values,
3. the *for initial* expression consumes each value of the array,
4. the initialization clause of the *for initial* expression is independent of the array of values, and
5. the *for initial* expression has no loop carried dependencies other than an index value and the shared accumulator.

Shared accumulator refers to the scalar value or aggregate structure returned by the *for initial* expression. The *for* and *for initial* expression presented earlier satisfy the five criteria. Our compiler merges the two expressions into a single parallel loop as explained in the next section. Currently, we do not prove that the reduction function is commutative; consequently, the user may introduce non-determinism. Some Sisal aficionados argue that the introduction of non-determinism in such a tightly controlled manner is good because it expands the domain of Sisal programming; others disagree. In the final section, we discuss our ideas regarding analysis and implementation techniques to guarantee determinancy. Our Sisal 90 compiler will provide this analysis.

3.0 Rewiring the graphs

Consider the expressions for `Force_update` and `Force_out` given in the previous section. Figure 1 is a logical view of the IF1 graphs [5] of the two expressions. The top node is a parallel *for*

computation. It has three subgraphs: generator, body, and returns. The generator defines a set of index values. An instance of the body is executed for each value, and each body computes two values, `Bond_energy` and `Force_record`. These values are passed to the returns subgraph that sums the bond energies and gathers the force records into an array. The bottom node is a sequential *for* computation. It has four subgraphs: initial, test, body, and returns. The initial subgraph initializes the index value `i` and the shared accumulator `Forces`. The body is executed once for each force record. The body updates two elements of `Forces` and passes the new array to returns subgraph. The returns subgraph selects the final value of `Forces` and passes it out from the compound node. We refer to this implementation as unoptimized.

Since the two expressions met the five criteria listed in the previous section, our compiler transforms the graph shown in Figure 1 into the graph shown in Figure 2. The first node initializes the shared accumulator `Forces` and passes it to the second node. The second node is a parallel *for* computation. Its generator is identical to the generator of the original *for* compound node. Its body and returns subgraphs are compositions of the body and returns subgraphs, respectively, of the original compound nodes. Since `Forces` is a shared resource, we place a lock about any read and write accesses to insure mutual exclusion. Notice that we have eliminated the test subgraph in the original graph, and that we no longer build the array of force records. We refer to this implementation as optimized.

There are a variety of ways to build the new graph and control access to the shared accumulator. Instead of locking all of `Forces`, we could lock individual elements or sections of the array. Maintaining a lock per element would be expensive unless the memory had presence bits. Since the Sisal runtime system slices *for* expressions into sets of iterations, we can eliminate the lock from the body by having each set of iterations initialize and maintain a local accumulator. As the sets finish, we "merge" the local accumulators to derive the final result. Such an implementation reduces the number of lock operations and contention for the lock, but uses more memory. Moreover, if the merge operator is different than the reduction operation, as in the example used in this paper, the compiler would have to synthesize it automatically.

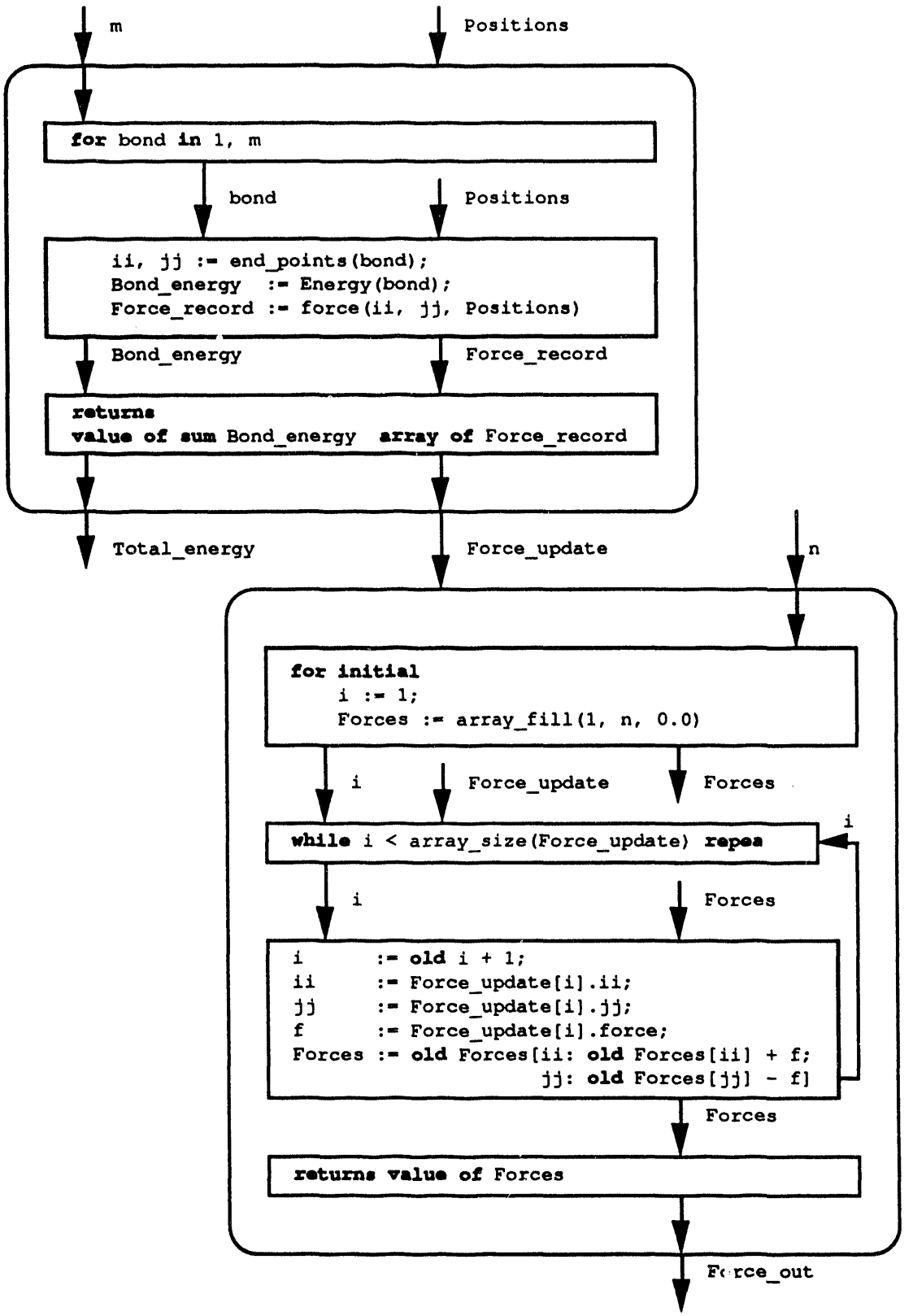


Figure 1 – A pair of computation–reduction expressions

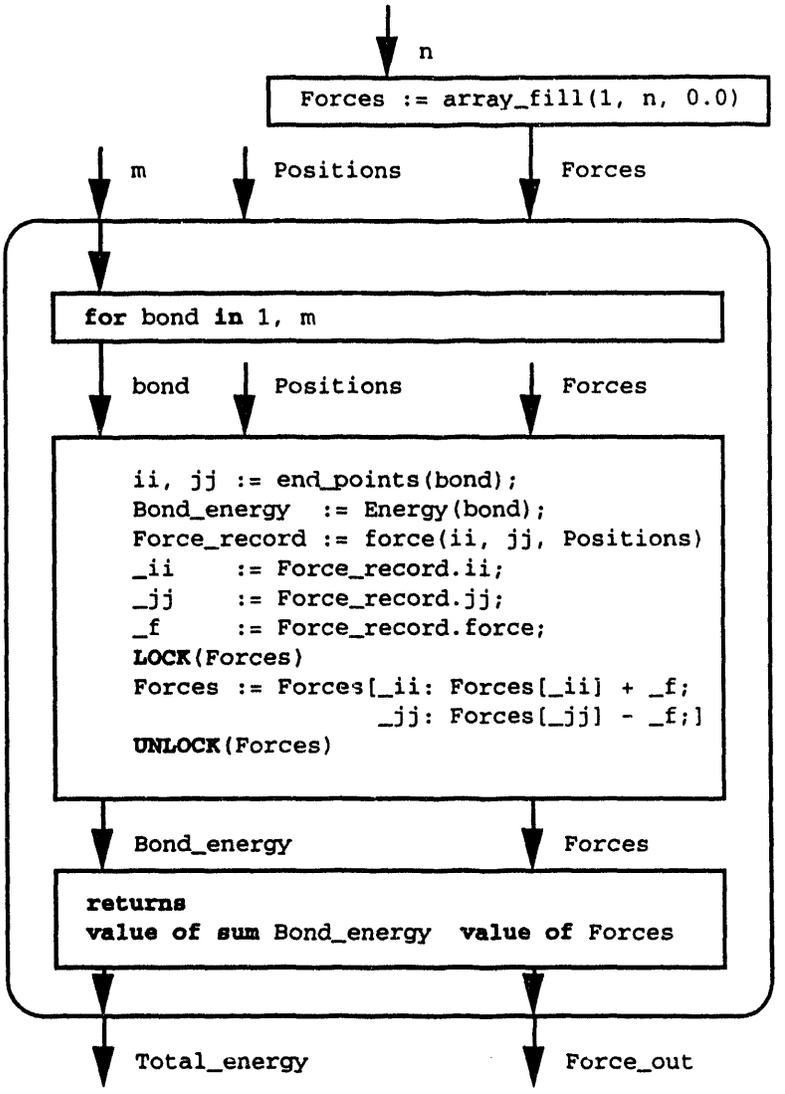
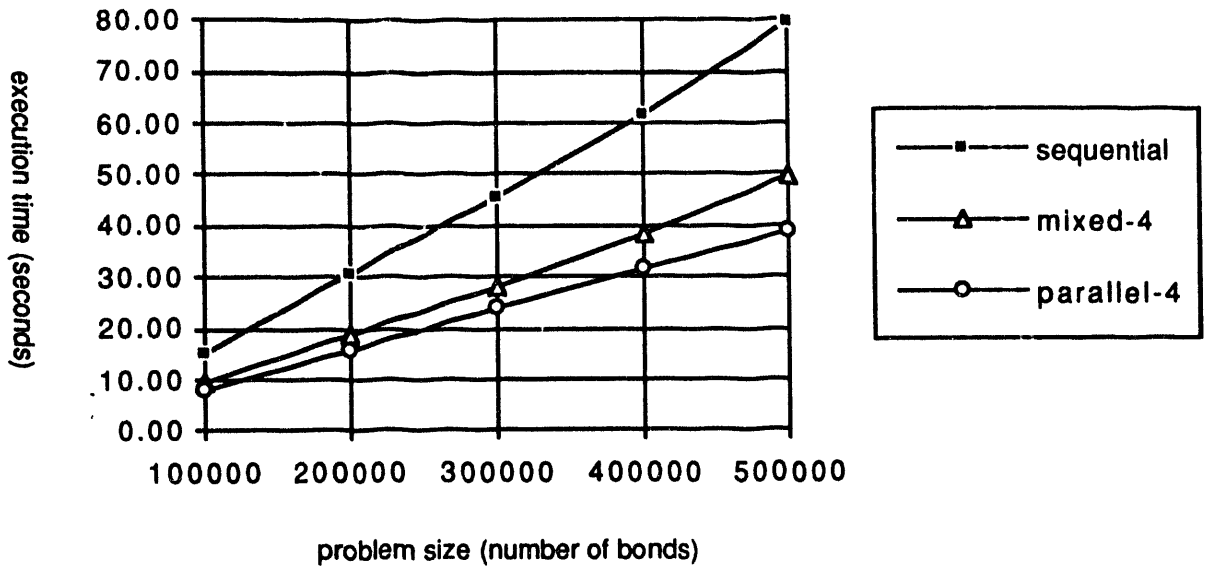


Figure 2 – A parallel computation–reduction graph

4.0 Performance

We ran a series of experiments to evaluate the performance of our optimization. We used a computation and reduction expression from molecular dynamics similar to the expressions used in the previous section. Table 1 gives the execution times and space requirements for different problem sizes. Graph 1 shows the graph of the execution times. Sequential and mixed are the one and four processor execution times, respectively, of the unoptimized implementation (Figure 1). Parallel is the four processor execution time of the optimized implementation (Figure 2). As expected, the



Graph 1 – Execution times of computation–reduction expressions

2). As expected, the optimized code uses less space than the unoptimized code. It also runs faster; however, we did not always see appreciable performance gains.

There are many factors that influence the execution time of the optimized code versus the execution time of the unoptimized code: the time to set and release locks, the time to write and read a record, lock contention, size of the computation expression, size of the reduction expression, number of processors, etc. If the size of the computation expression is at least p (number of processors) times greater than the size of the reduction expression, then there is little lock contention. Essentially, the concurrent tasks contend for the lock the first time, and then become staggered arriving at the critical section at different times. In the molecular dynamics code, all computation expressions are much larger than the reduction expressions. However, small reduction expressions minimize the effect of parallelizing the reduction operation. On large systems, Amdahl's Law may magnify the effect, but then the large number of processors increases lock contention. Overall, the optimized code may run five to twenty percent faster than the unoptimized code, but the real savings is in memory costs. The largest force computation in the molecular dynamics code for a simulation of 20,000 atoms, if unoptimized, generates approximately a 64MB array! That's a lot of memory.

Sequential Reduction	11.6 MB 9.37 sec	23.2 MB 18.97 sec	34.8 MB 28.25 sec	46.4 MB 38.18 sec	58.0 MB 49.76 sec
Parallel Reduction	8.4 MB 7.73 sec	16.8 MB 15.60 sec	25.2 MB 23.83 sec	33.6 MB 31.47 sec	42.0 MB 39.02 sec
Problem Size	100000	200000	300000	400000	500000

Table 1 – Time and memory usage of computation–reduction graphs

5.0 Future work

We plan to include specific syntax in Sisal 90 to support user defined reduction. A possible form for the computation expression is:

```
for bond in 1, m
  ii, jj := end_points(bond);
  Bond_energy := Energy(bond);
  F           := Force(ii, jj, Positions)
returns sum of Bond_energy
        Force_histo(n) of ii, jj, F
end for
```

The reduction function might be written as:

```
reduction Force_reduction(n, ii, jj: integer; F: real
                  returns array[real])
  for initial
    Forces := array_fill(1, n, 0.0)
  repeat
    new Forces := Forces[ii: Forces[ii] + F;
                         jj: Forces[jj] - F ]
  returns new Forces
  end for
end reduction  % Force_reduction
```

The names enclosed in parentheses prior to the keyword **of** at the reduction call site are values required to initialize the reduction; i.e., consumed in the initialization clause of the reduction function. The names listed to the right of the keyword **of** are the set values computed by each instance of the

body of the computation expression and reduced by the reduction. The reduction function is a *for initial* expression. In Sisal 90, we use the keyword **new** in place of the keyword **old**. The *for initial* expression has an implied test: the body executes once for each set of reduction values computed by the computation expression.

To insure determinate results, we are developing analysis to classify reductions according to their commutativity. Deciding the commutativity of an arbitrary function is difficult, and few function are commutative in general. However, most reductions are either comparative or accumulative. These types of functions are easier to analyze and a greater number are commutative. Preliminary studies seem to indicate that reductions divide into classes with variable degrees of commutativity. For each class, there exists a set of implementations of the member functions that insure determinate results. For example, we can insure that a non-commutative reduction return determinate results by enforcing sequential exeuction. We hope to report shortly on our analysis, implementation procedures, and performance of those implementations.

Acknowledgments

This work was supported by Lawrence Livermore National Laboratory under DOE contract W–7405–Eng–48.

References

1. J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 Handbook: Complete ANSI/ISO Reference*, chapter 13. Intertext/McGraw-Hill, 1992.

2. High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0.* Rice University, Houston, TX, May 1993.

3. P. Hudak and J. H. Fasel. A Gentle Introduction to Haskell. *ACM SIGPLAN Notices*, 27(5):T1–T53, May 1992.

4. J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. *SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2.* Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

5. S. Skedzielewski and J. Glauert. *IF1: An Intermediate Form for Applicative Languages.* Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985.

6. C. Tseng and J. H. Saltz. Compilation and runtime support for massively parallel processors. Supercomputing '93, Tutorial F3, November 1993.

DATE
FILMED
10/17/94
END