UCRL-LR-122354

Optimizing Parallel Reduction Operations

Scott M. Denton

June 1995

Lawrence Livermore National Laboratory

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

UCRL-LR-122354
Distribution Category UC-705

Optimizing Parallel Reduction Operations

Scott M. Denton
University of California, Davis

Master of Science Thesis

Manuscript date: June 1995

LAWRENCE LIVERMORE NATIONAL LABORATORY
University of California • Livermore, California • 94551

[illegible]

Optimizing Parallel Reduction Operations

By

Scott Michael Denton
BSCS, Northeast Louisiana University, 1985

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

John Feo

[illegible]

[illegible]

Committee in Charge

June 1995

Scott Michael Denton
June 1995
Computer Science

Optimizing Parallel Reduction Operations

Abstract

A parallel program consists of sets of concurrent and sequential tasks. Often, a reduction (such as array sum) sequentially combines values produced by a parallel computation. Because reductions occur so frequently in otherwise parallel programs, they are good candidates for optimization.

Since reductions may introduce dependencies, most languages separate computation and reduction. The Sisal functional language is unique in that reduction is a natural consequence of loop expressions; the parallelism is implicit in the language. Unfortunately, the original language supports only seven reduction operations. To generalize these expressions, the Sisal 90 definition adds user-defined reductions at the language level.

Applicable optimizations depend upon the mathematical properties of the reduction. Compilation and execution speed, synchronization overhead, memory use and maximum size influence the final implementation. This paper

1. Defines reduction syntax and compares with traditional concurrent methods
2. Defines classes of reduction operations
3. Develops analysis of classes for optimized concurrency
4. Incorporates reductions into Sisal 1.2 and Sisal 90
5. Evaluates performance and size of the implementations

Acknowledgements

My wife Nikki, who forced me to complete it.

My thesis advisors for advice and encouragement.

Dr. Patrick Miller, who should have been on the front page.

Contents

Acknowledgements iii

List of Tables vi

List of Figures vii

1 Introduction 1

1.1 What is a reduction? 1

1.1.1 Reduction operators and functions 2

1.2 How are they used? 4

1.2.1 Imperative languages and libraries 4

1.2.2 Functional languages 7

1.2.3 Applications for user-defined reductions 8

2 Implementation classification 12

2.1 Automatic classification 12

2.1.1 Analysis 12

2.1.2 Classes of reductions 13

2.1.3 Directives 14

2.2 Sequential and gathered reductions 15

2.3 Independent reductions 16

2.4 Accumulated reductions 17

2.5 Selected reductions 20

2.5.1 Indexed minimum reduction function 20
2.6 Additional code optimizations 21

3 Reductions in Sisal 90 **23**
3.1 Syntax and semantics 23
3.1.1 User-defined reduction language definition 24
3.1.2 Sisal 90 source examples 25
3.2 Dataflow in IF Intermediate Form 26
3.2.1 Standard IF optimizations 26
3.2.2 UReduce node 27

4 Performance study **30**
4.1 Sisal 1.2 classification prototype 30
4.2 Sisal 90 compiler results 34
4.2.1 Execution time and memory consumption 34

5 Conclusions **37**
5.1 Conclusions 37
5.1.1 Performance and Expressibility 37
5.2 Future Work 38

Bibliography **39**

A Code Samples **43**
A.1 Parallel array permutation build 43
A.2 Parallel global sum 44
A.3 Parallel counting histogram 45
A.4 Parallel first minimum 46
A.5 Parallel first minimum index 47

List of Tables

3.1 Reduction node translations and their zero trip return values 27
3.2 *UReduce* node embedded in the returns subgraph of a *Forall*. 28
3.3 *UReduce* port assignments 29
4.1 Time and memory usage of prototype accumulated reductions 32

List of Figures

1.1 Reduction operator vs. function 2

1.2 Reduction operator tree 3

2.1 Sequential and gathered executions of computation and reduction . . 15

2.2 Parallel independent computation and reduction and pattern match . 16

2.3 Parallel accumulated reductions: lock and master merge 19

2.4 Parallel selected reduction with candidate per processor 20

3.1 IF1 data flow optimizations 26

3.2 IF0 *UReduce* node for global sum 29

4.1 Execution times of prototype computation-reduction expressions. . . 33

4.2 Execution times of permuted update in-place. 35

4.3 Execution times of scalar reductions. 36

Chapter 1

Introduction

1.1 What is a reduction?

A reduction transforms a set of values computed sequentially or in parallel. It gathers to define a scalar value or transforms to build an aggregate structure. Examples are:

- global sum operators
- finding the minimum value or index in an array
- histogram functions
- calculating the incident force on a molecular particle

An array sum reduction produces a scalar value by adding together all array elements. The first minimum reduction returns the first occurrence of the minimum value in a list or array. A histogram is a transformational reduction that counts the number of occurrences of a value in one array and stores the count in another. A recurring theme in particle physics codes is the calculation of bond forces. Since the forces between particles are symmetric, each force is calculated once but accumulated on both affected particles.

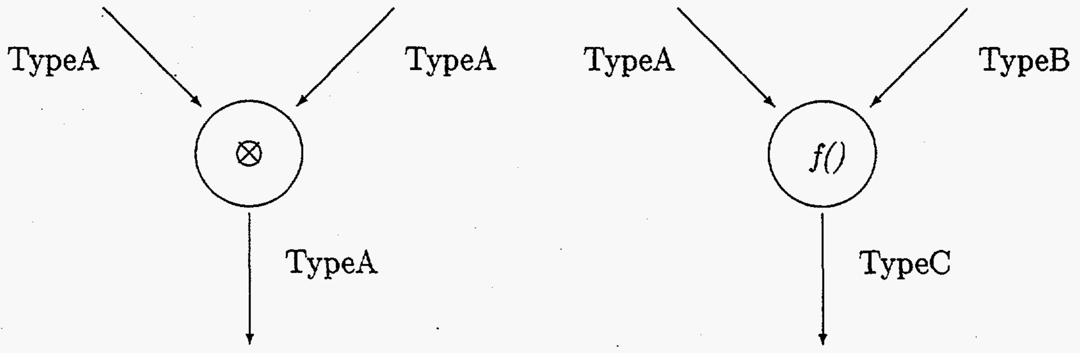


Figure 1.1: Reduction operator vs. function

1.1.1 Reduction operators and functions

A reduction consists of a set of input values, an operator or function, and an optional identity value. For the purposes of this discussion, an operator takes inputs of the same type and produces a value of the same type on output. A function takes inputs of different types and returns an output of possibly a third type as shown in Figure 1.1.

Conceptually, a reduction *operator* is an infix binary operator placed between each of the input values. An example of a scalar reduction operator is global array sum

input values $[5.0, 3.2, 6.0, 1.4, 2.0]$

reduction operator $+$

identity value 0.0

expression $0.0 + 5.0 + 3.2 + 6.0 + 1.4 + 2.0$

- **left** $((((5.0 + 3.2) + 6.0) + 1.4) + 2.0)$
- **right** $(5.0 + (3.2 + (6.0 + (1.4 + 2.0))))$
- **tree** $((5.0 + 3.2) + ((6.0 + 1.4) + 2.0))$
- **random** $((6.0 + 1.4) + ((5.0 + 3.2) + 2.0))$

Note that since scalar sum is commutative and associative (barring machine round-off), order of reduction is not important for many applications. The expression may be evaluated left-to-right, right-to-left, or in random groupings [5]. A tree mapping

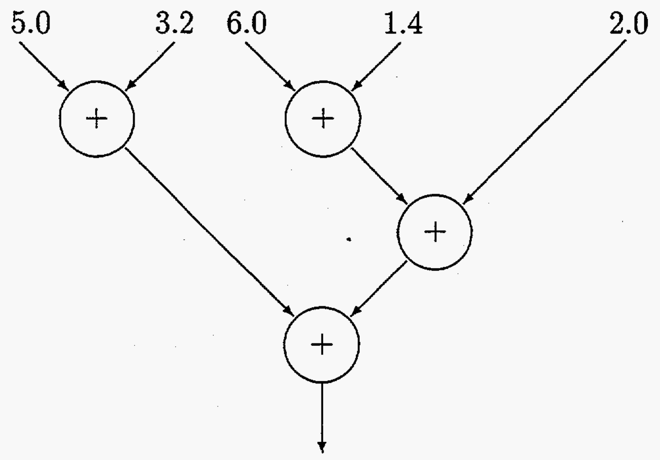


Figure 1.2: Reduction operator tree

is shown in Figure 1.2. The mapping most appropriate for a particular computer architecture may be chosen.

A reduction *function* takes different types. Consider a histogram with four bins represented as a four element array. The identity value is an array of four zeros. The input values are tuples to be accumulated into the histogram. The first element of a tuple selects the bin and the second is the value to add. The histogram function $h()$ applies each input value pair to the bins consecutively.

size and input values $4 : [\{4,1\},\{4,10\},\{2,1\},\{1,1\},\{1,1\}]$

reduction function $h(array, \{index, value\})$

identity value $[0,0,0,0]$

expression $h(h(h(h(h([0,0,0,0],\{4,1\}),\{4,10\}),\{2,1\}),\{1,1\}),\{1,1\})$

merge $M(\ h(h(h([0,0,0,0],\{4,1\}),\{4,10\}),\{2,1\}),$
$h(h([0,0,0,0],\{1,1\}),\{1,1\})))$

operator $[0,0,0,0] \oplus F(\{4,1\}) \oplus F(\{4,10\}) \oplus F(\{2,1\})$
$\oplus F(\{1,1\}) \oplus F(\{1,1\})$

To optimize for parallel machines, this nested invocation is normally split into local groupings for each processor, then merged for the final result. This implementation requires an initial value for each partial result and a merge function to combine local results. Consider a two processor system. The initial value for each processor is the identity value, a four-element zero array. The merge function $M()$ is array addition by element ($\oplus$). $M()$ combines the two temporary array values computed on the two processors. Neither the initial value nor the merge function is usually specified by the source code of the reduction function, and must be synthesized.

A histogram function can also be converted into a histogram operator by expanding types to match. The function $F()$ in the example converts the pair $\{4, 1\}$ to $[0, 0, 0, 1]$. Array addition may then be used directly.

1.2 How are they used?

Computation-reduction pairs are a mainstay of traditional scientific programming. In a calculation evolving over time, iterative computation and reduction update the previous cycle to produce the new state. Because the efficient expression and implementation of reduction operations can reduce the cost of parallel programming, current compilers and libraries have targeted the most common variations.

1.2.1 Imperative languages and libraries

Languages like Fortran, C, and C++ have been used extensively for scientific programming. These languages use an explicit control flow and memory model. Compilers for these traditional sequential languages, such as Cray CF77 [6] and the Fortran D compiler [21], often find parallelization optimizations by pattern matching. The following inner product from the Livermore Loops [9] is a simple example.

```
C       Livermore Loop 3
        Q = 0.0
        DO 100 k = 1, 100
            Q = Q + Z(k)*X(k)
    100 CONTINUE
```

Parallel pragmas may be given to force a parallel implementation such as in the SGI f77 compiler [18]. The *DO* loop is sliced so that each processor operates on a contiguous range of values. This method is limited to scalar reduction operators such as array sum:

```
      SUM = 0.0
C$DOACROSS LOCAL(I), REDUCTION(SUM)
      DO 10 I = 1, N
          SUM = SUM + A(I)
   10 CONTINUE
```

Since reductions may introduce dependencies, most languages separate the computation and reduction tasks. As an example, Fortran 90 [1] and HPF [14] provide a rich set of predefined reduction functions on extant arrays. In this data parallel model, the array memory may actually be distributed across all of the processors.

```
C  find the minimum value in z_array
      z_min = MINVAL(z_array)
C
C  return the first index of the minimum value in z_array
      z_min_loc = MINLOC(z_array)
```

Neither of these methods provides for general reductions defined by the user. To compensate for these language deficiencies, message passing libraries for sequential threads on parallel computers often allow extension of their predefined reductions. The MPI [8] message passing interface adds elementwise array reductions defined by the user.

```
C  MPI predefined reduction to sum x's from all processors in Fortran
      CALL MPI_REDUCE(x, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
C
/*
 * MPI array product of complex numbers in C
 * Define type, register function, call myProd()
 */
   MPI_Type_contiguous( 2, MPI_DOUBLE, &ctype );
   MPI_Type_commit( &ctype );
   MPI_Op_create( myProd, True, &myOp );
   MPI_Reduce( a, answer, 100, ctype, myOp, root, comm );
```

Explicit locks are available as an extension in some shared memory compilers like PDDP [22]. User-defined reductions are still complex. Witness the code to reproduce the predefined PDDP_SUM() global add:

```
C Global sum of elements in array a(1:asize)
        subroutine globalsum(sum, a)
        shared real sum, a() ! shared by all processors
        real partial ! local to each processor
        integer i, asize
        lock sumlock ! critical section lock variable
        integer width
        MASTER ! only one processor initializes
        sum = 0
        ENDMASTER
        BARRIER ! wait for initialization
        asize = pddp_size(a)            ! length of array a
        partial = 0 ! local partial sum
        width = (asize+_NPROCS-1)/_NPROCS ! slice per processor
        do i = (_IPROC-1)*width+1, MIN(_IPROC*width, asize) ! slice
            partial = partial + a(i)
        end do
        LOCK(sumlock) ! critical section
            sum = sum + partial
        UNLOCK(sumlock)
        end
```

1.2.2 Functional languages

Functional languages [2] map input values to output values; since there is no explicit memory model there can be no variable aliasing, side effects, or time dependent (non-determinate) errors. With the program text defining a data dependency graph, the availability of inputs defines the order of execution. All synchronization, communication, and scheduling are implicit.

Despite the lack of explicit memory, functional languages also support optimized reduction operations. Haskell [12] 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 [16] is unique in that reduction is a natural consequence of loop expressions. The reduction operation appears as a keyword in the *returns* clause of a *for* expression.

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

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

The Sisal 1.2 language definition supports only seven reduction operations: sum, product, least, greatest, array, stream, and catentate. Because the reduction is part of the loop, the current optimizing compiler can often overlap computation and reduction for these intrinsics. It generates tasks to take best advantage of the underlying architecture.

1.2.3 Applications for user-defined reductions

A shortcoming of all current languages is the support of only a limited set of predefined reduction operations. Many applications require very specific reductions. Sisal 1.2 does not provide an intrinsic to find the index of the first minimum value. A new intrinsic would be required:

```
% find the index of minimum value in z_array
z_min_index := for z in z_array
               returns value of index_least z   % NOT AVAILABLE
               end for
```

The available Sisal 1.2 solutions are not optimal. The first doubles the computation's overhead,

```
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;
```

and the second solution eliminates all parallelism.

```
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 situation is more dire when generating a set of values to be counted or accumulated by type as in a histogram. 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. This computation cannot be done with the intrinsic reductions.

For example, consider a set of n particles and m bonds. Each bond represents a force between two particles. In the code fragment below, the function end_points returns the indices of the two particles participating in the bond. Function energy returns a record with the input indices and the energy of the bond that is a function of the particle positions. Force_update is a list of these records. Force_out is an array of cumulative bond energies, one entry per particle.

```
Force_update :=
    for bond in 1, m
        index_1, index_2 := end_points(bond);
        Force_record := energy(index_1, index_2, Positions)
    returns 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;
        index_1 := Force_update[i].index_1;
        index_2 := Force_update[i].index_2;
        be := Force_update[i].bond_energy;
```

```
            Forces := old Forces[index_1: old Forces[index_1] + be;
                                 index_2: old Forces[index_2] - be ]
        returns value of Forces
        end for
```

The force update array can be calculated in parallel, but the calculation of total force on each particle uses a sequential *for initial*. 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 (sliced by the compiler across processors) that includes a critical section to control access to the force array,

```
    do ibond = 1, m
        call end_points(bond, ibond, index_1, index_2)
        be = energy(index_1, index_2, Positions)
        lock(Force_out)
            Force_out(index_1) = Force_out(index_1) + be
            Force_out(index_2) = Force_out(index_2) - be
        unlock(Force_out)
    end do
```

Since the energy 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. An alternative is to use two loops with a temporary array per processor,

```
    iproc = processor_number();
    do ibond = 1, m
        call end_points(bond, ibond, index_1, index_2)
        be = energy(index_1, index_2, Positions)
        Force_temp(iproc, index_1) = Force_temp(iproc, index_1) + be
        Force_temp(iproc, index_2) = Force_temp(iproc, index_2) - be
    end do
    lock(Force_out)
        do ipart = 1, n
            Force_out(ipart) = Force_out(ipart) + Force_temp(iproc, ipart)
        end do
    unlock(Force_out)
```

To address the issues of functionality and complexity, the HPF Journal of Development [13] and the HPF-2 draft [15] have suggested additional language features be added for user-defined reduction functions in Fortran. The final form of these features has not been determined. A problem with any of these approaches is that reductions take place on extant arrays. The size of the values to reduce becomes a limitation on problem size due to memory constraints [7]. Sisal 1.2 does not require extant arrays but lacks user-defined reductions. This paper describes how language features were included and implemented in Sisal 90 [19] to address both of these issues.

Chapter 2

Implementation classification

2.1 Automatic classification

2.1.1 Analysis

This paper classifies reductions to reveal possible methods of implementation for multiprocessor computers. There are two types of classification: mathematical properties and data dependencies. Classifications determine the range of possible implementations. The hardware characteristics of a real machine with finite primary memory and parallel overhead further restrict the possible implementations. Our analysis selects an implementation based upon the relationships between machine- and application-specific sizes and overhead costs. For example, we mark a parallel loop sequential if physical memory is exceeded or the parallel overhead outweighs the parallel speedup achievable.

For the techniques tested, the factors used to select an implementation when classification allowed a choice were

- Size
  - Parallel reduction state replicated across processors exceeds memory
  - Parallel gathered reduction values exceeds memory threshold

- Execution time
  - Parallel task overhead large compared to parallel work
  - Parallel shared memory lock contention too high for number of processors used

We encode these cost criteria in IF2 [23]. The first phases of the IF2 optimization in the OSC compiler derive cost estimates and sizes for the execution. Costs are attached to the computation nodes with a *%cc* mark. The latter phases of the compiler use data dependencies, execution cost, synchronization, communication, and tasking startup estimates to select an implementation.

2.1.2 Classes of reductions

The amount of concurrency classifies the top level. This specifies whether the reduction is to be fused with the computation. The mathematical properties of the reduction determine the next level, the range of possible reduction implementations. The analysis and effect of classification are the focus of this paper.

1. **sequential:** match the reduction semantic, a single thread of execution (sequential computation, sequential reduction)
2. **gathered:** save parallel computation results in a temporary array for later sequential reduction (parallel computation, sequential reduction)
3. **parallel:** fuse portions of the sequential reduction with the parallel computation (parallel computation, parallel reduction)
   (a) **independent:** only one update to each element (a permutation for example)
   (b) **accumulated:** multiple updates are combined in a shared accumulator for the final result (global sum or histogram)

(c) **selected:** a conditional update replaces the previous value (first minimum)

(d) **scanned:** updates depend upon a recurrence relation, a non-associative function of the previous update

2.1.3 Directives

Source level pragmas may be used to directly specify a complete implementation or general classification. Since OSC is an automatic optimizing compiler, this method should only be needed when performance from analysis is unavailable.

The upper level directives match the implementation classification. A gathered loop reduction may consume too much memory, so a sequential version is selected by the user with *returns sequential* overriding the default. To invoke analysis to find parallel optimizations of the reductions as described below, the *parallel* directive may be used.

```
#pragma returns sequential
#pragma returns gathered
#pragma returns parallel     % run parallel analysis
```

Hints are available to direct the analysis for each of the *parallel* implementation classes.

```
#pragma returns independent
#pragma returns accumulated
#pragma returns selected
#pragma returns scanned
```

In addition, for accumulated and selected reductions, details of the method to combine values may be specified. This is discussed in more detail in later sections.

```
#pragma reduction merge(master)
#pragma reduction merge(tree)
```

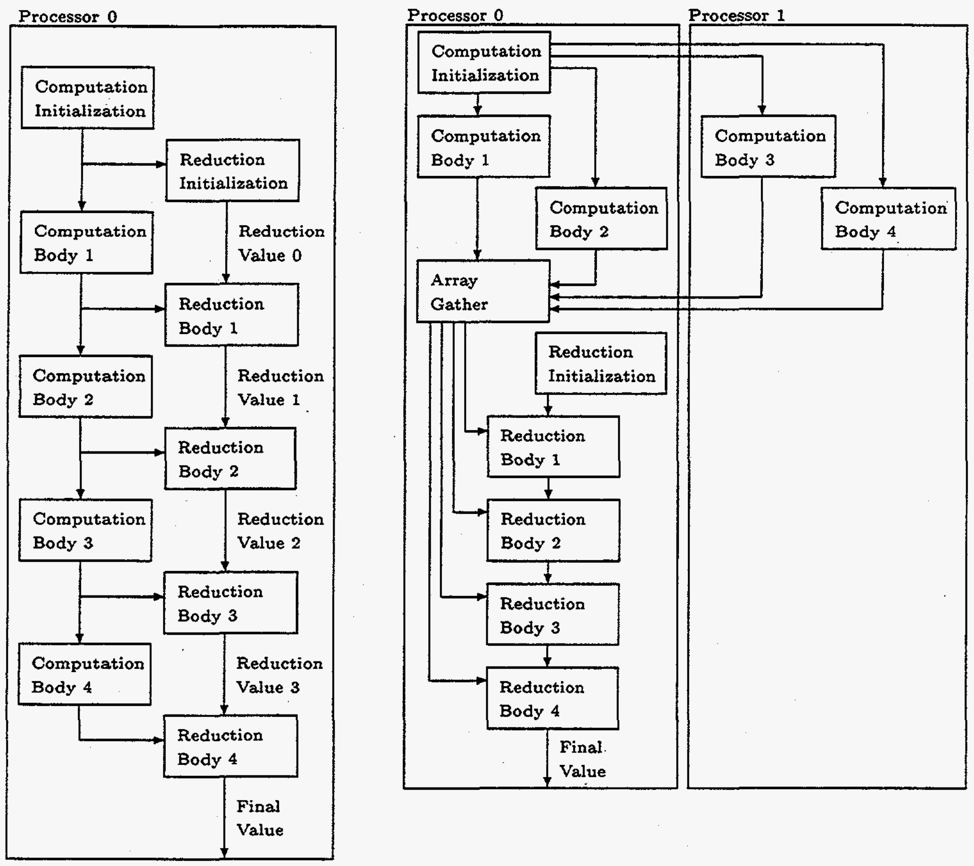


Figure 2.1: Sequential and gathered executions of computation and reduction

```
#pragma reduction merge(lock)
#pragma reduction merge(candidates)
```

2.2 Sequential and gathered reductions

In the first part of Figure 2.1, the sequential implementation execution proceeds from the top down. Nothing is done concurrently. The computation and reduction initializations are run. After that, each computation and reduction pair are executed in turn using results from the prior step. A sequential implementation may be the fastest correct implementation for a given reduction for several reasons:

1. data dependencies may restrict concurrency,

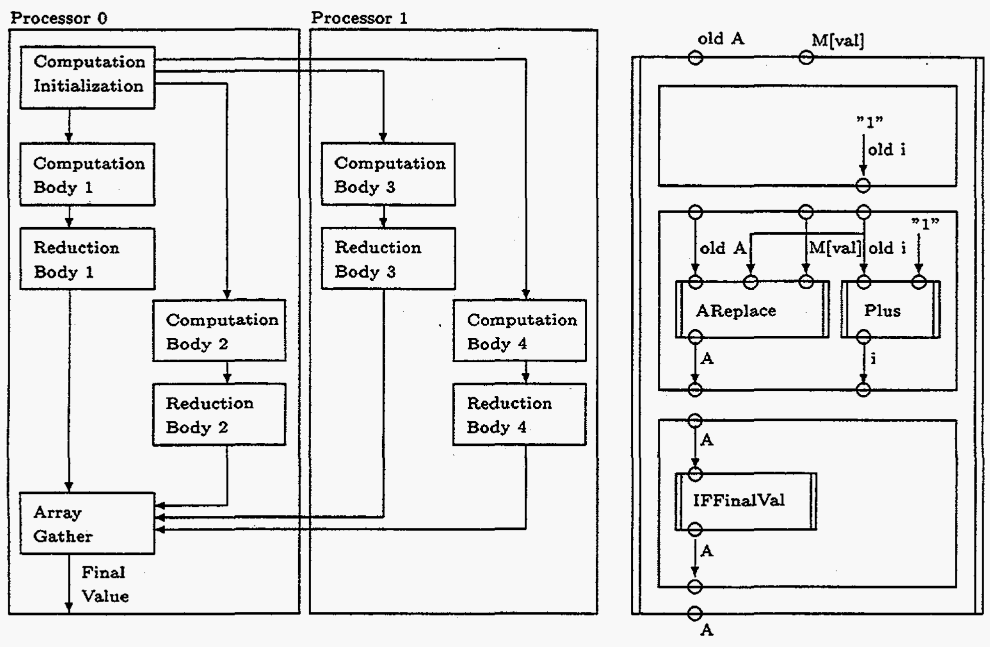


Figure 2.2: Parallel independent computation and reduction and pattern match

2. for small amounts of work, overhead may cause parallel slowdown rather than speedup,

3. memory is minimized since there are no large gathered temporaries.

A gathered implementation runs the computation in parallel and saves the results in a vector temporary. The reduction is then run sequentially. In the second example in Figure 2.1, two processors run in parallel. The four values are gathered in a temporary array. The reduction is then run sequentially by accessing elements from the array to produce a final value. The size of the temporary is a limitation for many applications. These problems are addressed by additional analysis for selection of a parallel implementation by class or user directive.

2.3 Independent reductions

Updates to the reduction value may be independent as in Figure 2.2. A reduction result array is given an initial value. Then each element of the array is updated no more than once. The separate values generated in each body are gathered from the multiple processors and placed in the result array. Permutations of the complete index set are a common example. The value at every index is some one-to-one and onto mapping from an original set of indexes that cover the array. Typically an index set is permuted and used to assemble the final values in the new order as in FFT algorithms [17]. The general analysis to prove uniqueness in the update of each element of the array is approachable by integer linear programming as discussed in [11].

The version implemented is a substantially simpler version, pattern-matching on the increment of the index variable. As also shown in the figure, an array *old A* and multiple values *M[val]* are input at the top and produce the new value *A* at the bottom. The values are placed at the monotonically increasing element *old i*. This index starts at 1 and increments by 1 for each value. Since no elements are multiply assigned, the array build can be done independently. Once analysis shows independent updates to the array elements, a fast and and simple parallel version is implemented. On a shared memory multiprocessor, the updates are performed in place with no locks or other synchronization.

2.4 Accumulated reductions

Accumulated reductions update a shared value in the reduction body. Examples are the Sisal 1.2 sum and Sisal 90 histogram. Analysis of the data dependencies determines the possible optimizations. Commutative and associative reductions remove the requirement for sequential execution. For example, with initial value of *old product* set to 1, compare the commutative and associative expression

```
product := old product * val
```

with a non-commutative and non-associative expression requiring sequential execution,

```
product := old product * (old product + val)
```

The first reduction body may be implemented in any order, even in parallel if the multiply-update is an atomic operation. The second reduction body will produce different results when executed out of order. Consider input multiple values $(2, 5)$ to this second reduction. Left-to-right produces $3 * (3 + 5) = 24$ while right-to-left produces $6 * (6 + 2) = 48$. Pattern matching on data dependencies is used to select reductions for optimization. The previous accumulator value can be updated by add, subtract, multiply, or floating-point divide of values which are not a function of the accumulator. This automatic analysis is similar to the pragmas that must be inserted by hand in the latest HPF-2 draft [15].

The first stage analysis reveals that parallel optimization may be done. Additional analysis defines two possible implementation subclasses. On shared memory machines, the accumulated values may be updated in place in a locked critical section. On any shared or distributed memory processors, intermediate per-processor results may be combined with a synthesized merge operator and initial identity element. Because of the limited number of operations on base types, we use a look-up table similar to Table 3.1. The method used for a particular shared memory application is chosen by using estimates of the ratio of computation to lock overhead and conflict in the critical section. One other criteria is available memory size. If it is not possible to duplicate the accumulator on every processor in the merge implementation, the locked version must be used.

The *accumulated* with a *merge(lock)* version is shown in Figure 2.3 and global sum in Appendix A.2. The initial value for the reduction and computation is set in a master sequential region. Each computation may then proceed in parallel. The update of the shared accumulator is performed in a locked critical section. The read, update, write must be an atomic operation to prevent race conditions. The default lock method forms a critical region for all processors. Any processor access to the code updating the shared accumulator will be sequentialized. A processor executing the computation is unaffected. For load balance, more work requests (slices) may be allocated than the number of available processors. As each slice is finished, a new

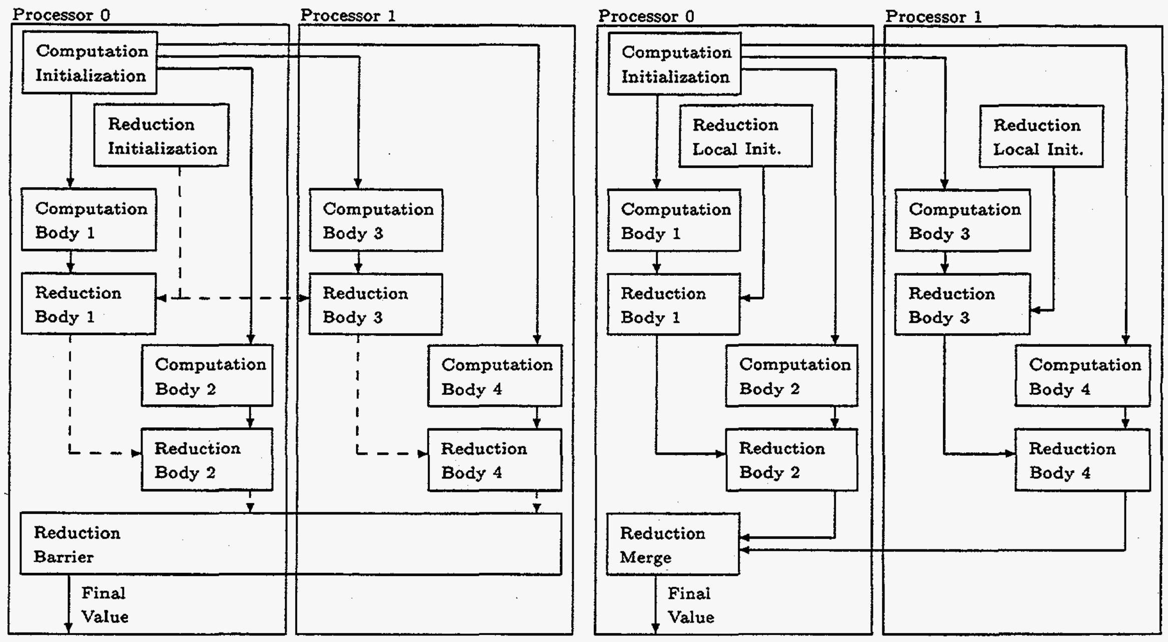


Figure 2.3: Parallel accumulated reductions: lock and master merge

one from the run queue is requested. This spreads the work more evenly among the processors.

Building a local accumulation temporary exploits locality and reduces contention at the expense of memory. For this reason, distributed memory machines favor this implementation. Figure 2.3 and the code in Appendix A.3 show how a local reduction value is initialized on each processor. This local value is synthesized by analyzing the operator used in the update of the accumulator. For the *plus* and *minus* operators, each element in the local accumulator starts at zero while *times* and *divide* start at one. Integer divide may not be done in any order however. Consider $(10/1)/5 = 2$ with $10/(1/5) = \infty$. Integer roundoff loses information. For the boolean operators, logical *and* starts at true (one) and logical *or* at false (zero).

Each processor computes a local reduction value. For shared memory, each processor stores its value in a known location. The master processor then waits until each processor finishes before retrieving the results and accumulating sequentially.

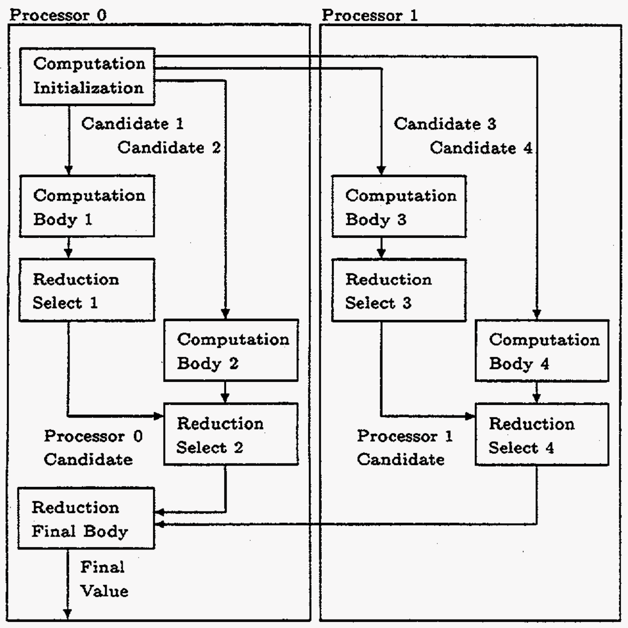


Figure 2.4: Parallel selected reduction with candidate per processor

2.5 Selected reductions

The Sisal 1.2 keyword *least* as shown in Appendix A.4 implements the first minimum reduction. It is an example of the *selected* reduction. For each multiple value to be reduced, an *if* statement guards the replacement of the result. The final value is eventually selected from only one iteration.

In Figure 2.4, the general form of the optimization is shown. For the first minimum reduction operator, the reduction is run left-to-right for each processor's input values. Then, the local value from each processor is supplied to the master processor (*merge(master)*) for the final merge. The merge reduction is identical to the local reduction.

2.5.1 Indexed minimum reduction function

An additional complexity occurs when optimizing a *selected* reduction function rather than an operator. Since less data is returned by the reduction than is supplied as inputs; the candidate inputs must be saved for the final merge.

An example is the first minimum index reduction in Appendix A.5. Notice that the reduction only returns the index value. However, the minimum value is used also, to select the final result. They are both part of the reduction state. For the master to reduce the result, a candidate from each processor must present its complete candidate state. Therefore, the reduction implementation on each processor is modified to save the candidate inputs. The master merge is the original reduction run across all candidates.

2.6 Additional code optimizations

By fusing portions of the sequential reduction into the parallel computation, additional possibilities for optimization are revealed. The code's execution time is decreased by eliminating:

- function calls by inlining,
- unnecessary copying and record builds,
- unnecessary locks and large critical section size,
- unnecessary reference counts.

A reduction call can be made just like a function call. General purpose registers are saved to stack memory and arguments are passed to the call. Pulling the reduction inline eliminates this overhead.

When the reduction is part of the body of the iterative computation, computation values are used as they are produced. A large temporary array to hold the intermediate multiple values to be reduced is not needed. Since the consumer can use the produced values immediately, structured record variables are never stored to memory. This would normally involve an extra pack/unpack for each record. Instead, a record typically remains in registers for immediate use.

The reduction consumer directly follows the computation producer. Only one reference to the computation value is made. Therefore, no reference count on the produced value is needed. When the variable goes out of scope, it is no longer needed and can be deleted or reused. This eliminates the extra locks normally used to count the references to shared data.

In the simplest implementation, locks would guard every shared reduction accumulator. By producing local copies only one lock per processor is needed. The size of the critical section can also be reduced by moving calculations that do not involve the accumulator outside. The values will be calculated in the computation loop body.

Chapter 3

Reductions in Sisal 90

3.1 Syntax and semantics

Sisal *for* constructs are implicitly parallel loop expressions. The generator produces an independent loop body for each element in its range.

```
for range_generator
    loop_body
returns reduction of loop_multiple_values
end for
```

In the following range generators, each i is available for use in its corresponding loop body.

```
for i in 1, 10 ...          % 10 bodies
for i in A ...              % one body for each element in A
for i in A cross j in B ... % one body for each possible pair
for i in C at j, k          % one body for each element in 2D array
```

Loop multiple values created in the loop bodies are supplied as input values to the reduction in sequence. For example, a loop can return the final value that was bound to a loop name, build an array from the sequence, or take the sum of the list. To eliminate keywords, the seven Sisal 1.2 sum, product, least, greatest, catenate, value, array, and stream reductions are predefined reductions in the Sisal 90 definitions file. Because of the separation of the variable name space, problematic name clashes with user functions and variables are eliminated. The invocation of either the predefined or user-defined reduction `sum` is the same.

```
for i in 1, n
    x := i*i;
returns sum of x             % find sum of squares
```

3.1.1 User-defined reduction language definition

A reduction definition in Sisal 90 is similar to a Sisal function definition with an embedded sequential loop, the *for initial.* The key difference is that the call of the reduction may appear only in the *returns* clause of a *for* expression since it takes loop multiple values for input. These multiples are the values produced by the multiple parallel bodies of the *for* loop.

```
reduction name( initial_values repeat loop_multiple_values
     returns result_types )
     for initial
          reduction_values
     repeat
          reduction_body
     returns value of reduction_values
     end for
end reduction
```

As the template above shows, the keyword *reduction* is followed by the reduction name, input values, and result types. The reduction is run in three phases. In the first phase, the initial values are used to produce a default reduction value. If no multiples are ever supplied (a zero-trip loop), this default value will be returned. In the second phase, the multiples are supplied to the body in order, for calculation and update to the final reduction values. The third phase simply returns the final reduction value.

3.1.2 Sisal 90 source examples

Consider the predefined *sum* reduction from the previous section. In this sample similar to Appendix A.2, it is coded as a user-defined reduction.

```
reduction sum( repeat i:integer returns integer )
    for initial
        total := 0
    repeat
        total := old total + x
    returns value of total
    end for
end reduction
```

The initial values (input values before the *repeat*) are blank since the sum reduction always begins at zero. There is one loop multiple value x supplied in sequence by the *for* loop at the call site. The result type is the same as the type of the multiple. In the first phase, *total* is initialized to 0. The second phase adds each multiple to the *old total* to produce a new *total.* In the last phase, the value of *total* is returned. If no multiples are supplied to the reduction, the final value of *total* is the initial value, 0.

The counting histogram in Appendix A.3 is similar. At index *histo_update*, a single element of the array is incremented by one to count the number of items in that histogram bin. The other examples in Appendix A follow a common pattern. State values are initialized, then updated with each parallel computation result. All are explained in more detail in the next chapter.

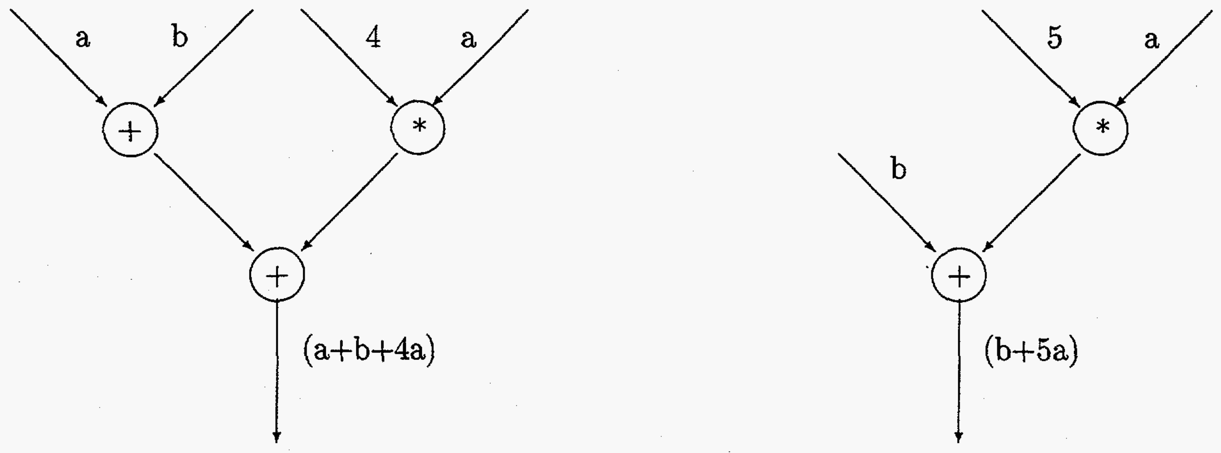


Figure 3.1: IF1 data flow optimizations

3.2 Dataflow in IF Intermediate Form

3.2.1 Standard IF optimizations

The Sisal 90 parser translates the source program into the IF1 [20] intermediate level data flow representation. Nodes in IF1 denote operations such as add or divide while edges represent values that are passed from node to node as in Figure 3.1. Functions are graph boundaries that surround groups of nodes and edges. Types can be attached to each edge or function. Nodes with embedded subgraphs are called compound nodes. These forms reveal the opportunity for many value-oriented optimizations in the Sisal OSC compiler [3] such as function inlining, record and array fission, loop invariant removal, common subexpression elimination, loop unrolling, and dead code removal.

Additional optimizations are performed by the OSC compiler by a second translation to IF2 [23], a representation with explicit memory management. Build-in place and update-in-place analysis is used to eliminate memory copy and management overhead. Loops that warrant parallel and vector execution are automatically made concurrent. Finally, variable temporaries are selected and C code is generated for final compilation to an executable.

Although the Sisal 1.2 intrinsic reductions can be written in Sisal 90, they are

Reduction	Literal	IF1 Equivalents	Zero trip result	
sum	SUM	Reduce(SUM)	0	*type*
product	PRODUCT	Reduce(PRODUCT)	1	*type*
least	LEAST	Reduce(LEAST)	*max*	*type*
greatest	GREATEST	Reduce(GREATEST)	*min*	*type*
catenate	CATENATE	Reduce(CATENATE)	[]	*type*
array	ARRAY	AGather	[]	array [*type*]
stream	STREAM	SGather	[]	stream [*type*]
user defined	*user defined*	None	initial	returns *type*

Table 3.1: Reduction node translations and their zero trip return values

predefined for convenience. During the OSC type resolution phase, type information is added to reduction node edges, and then the reductions are inlined using their IF1 equivalent definitions. This allows existing compiler optimizations to be used directly. Table 3.1 shows the reductions and their IF1 equivalents.

3.2.2 UReduce node

Without analysis, user-defined reductions are translated directly into IF1. Such a translation often obscures the high-level reduction expression. Therefore, Sisal 90 adds compound *UReduce* nodes that specify the reduction operation; they are left unexpanded in the IF0 [10] high level representation. All IF0 nodes can be translated to IF1 with only the loss of succinctness and possibility of high-level optimization. Leaving the node in IF0 allows for efficient implementations based upon low-level control dependencies, sizes, and the amount of determinism required by the application.

As Sisal 90 user-defined reductions remain unexpanded only in the returns clause of parallel *for* loops, IF0 *UReduce* nodes appear only in the returns subgraph of parallel *Forall* compound nodes, as shown in Table 3.2. Matching the three phases of a reduction, the *UReduce* node contains three subgraphs: initialization, body, and returns. The first graph sets up any initial values. The body graph takes multiples from the *Forall* loop node containing it. The *UReduce* returns graph contains the merge and *FinalValue* node. When a *UReduce* node is used in the returns graph of

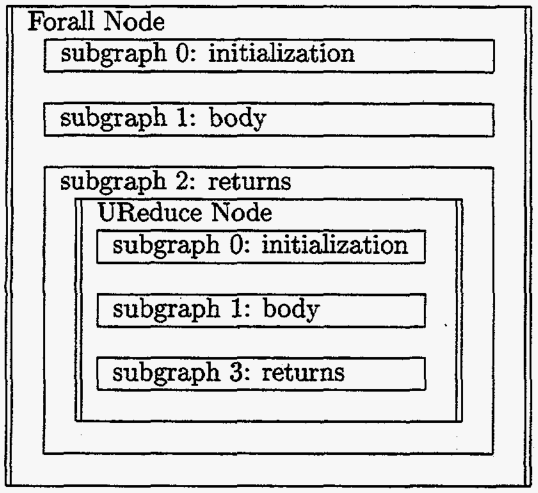
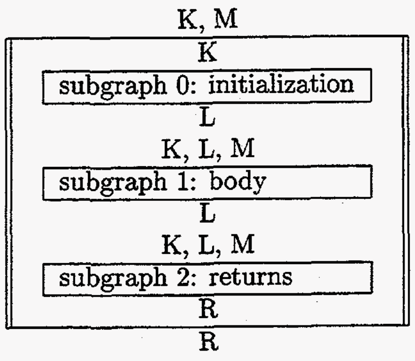


Table 3.2: *UReduce* node embedded in the returns subgraph of a *Forall.*

a *Forall*, it enables postprocessing of the multiples produced by the *Forall* subgraph body.

The IF language defines the ports on nodes and graphs. These ports are the locations where data edges attach. The definition of a node operation also defines the port number on which a value will be input or output. The port positions for the compound *UReduce* node are shown in Table 3.3.

The IF1 for the scalar sum example is shown in Figure 3.2. Values set in the *for initial* clause, in this case *total*, of the reduction appear in the first subgraph. In the body subgraph, the *old total* is incremented with an *IFPlus* operation to produce the new *total* at the bottom of the subgraph. On the first iteration through the body, *old total* is the value supplied by the initialization subgraph. Later iterations feed the new value of *total* back in to the *old total* port since they share the same port number. The returns subgraph takes the final value from the body and returns it. This research seeks to optimize reductions for parallel machines in light of these data dependencies.



Class	Usage	Port Range
K	values imported into node	$1 \ldots n_K$
M	multiples	$n_K + 1 \ldots n_K + n_M$
L	loop carried values	$n_K + n_M + 1 \ldots n_K + n_M + n_L$
R	results of node	$1 \ldots n_R$

Table 3.3: *UReduce* port assignments

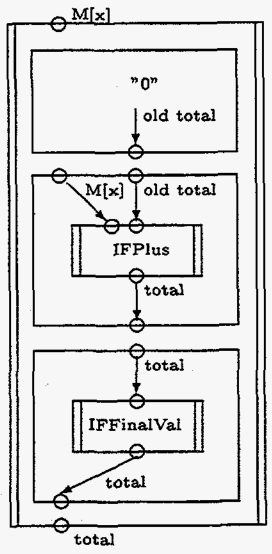


Figure 3.2: IF0 *UReduce* node for global sum

Chapter 4

Performance study

4.1 Sisal 1.2 classification prototype

Before defining the reduction syntax and detailing all classes, I used the OSC Sisal 1.2 compiler to prototype user-defined accumulated reductions. A parallel *for* loop followed by a call to a sequential loop function is recognized as a computation-reduction pair [7]. The parallel computation produces multiple update values in an array. A sequential loop consumes all values in order to produce the final accumulated array. These two bodies fuse to form a single parallel loop with a critical section. The compiler pulls most of the sequential calculations into the parallel computation. The general pattern mathing template is:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Parallel computation loop followed by a function call
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

let
    updates := for val in 1, n
                   update := f(val)
               returns array of update
               end for;
```

```
in
    reduce_function(old_accum, updates);
end let

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sequential function using computation values
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function reduce_function(old_accum:accum_type updates:update_type
    returns accum_type)
    for initial
        i := 0
        accum := old_accum;
    while i < array_size(updates) repeat
        i := old i + 1;
        index := g(i);
        new_val := old accum[index] + updates[i];
        accum := old accum[index: new_val];
    returns value of accum
    end for;
end function
```

The *-r <function>* option signals the OSC compiler to attempt to perform the required analysis to turn the function into an optimized reduction. Since this technique was only meant to be used as a prototype, no changes to the Sisal 1.2 language were made. Only functions operating on extant arrays are recognized. The natural Sisal computation-reduction syntax is not extended to handle general reductions. That is saved for the Sisal 90 language. The sequential to parallel fusion optimization is applied to pairs of *for* computations and *for initial* reduction expressions that satisfy a complex set of criteria.

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

Table 4.1: Time and memory usage of prototype accumulated reductions

1. The *for initial* expression depends directly on the *for* expression, and does not depend on any descendant of the *for* expression.

2. The initialization clause of the *for initial* expression is independent of the array of values consumed.

3. The *for initial* expression consumes every value of the produced array, once and in order.

4. The *for initial* expression has no loop carried dependencies other than an index value and the shared accumulator.

5. The *for initial* expression depends on the *for* expression for only an array of values.

A series of experiments were used to evaluate the performance of this optimization. An expanded version of the molecular dynamics example from Section 1.2.3 was used. Table 4.1 gives the execution times and space requirements for different problem sizes. Because the test computer had only 64MB of core memory, larger problem sizes for the sequential reduction could not be run. Memory page swapping would have skewed the results. The savings in this case was not just the execution speed, but memory size.

Figure 4.1 shows the graph of execution times for the sequential and the original unoptimized four processor implementations (using the sequential reduction). The optimized four processor (using a parallel reduction) execution time is compared. As expected, the optimized code uses less space than the unoptimized code. For this

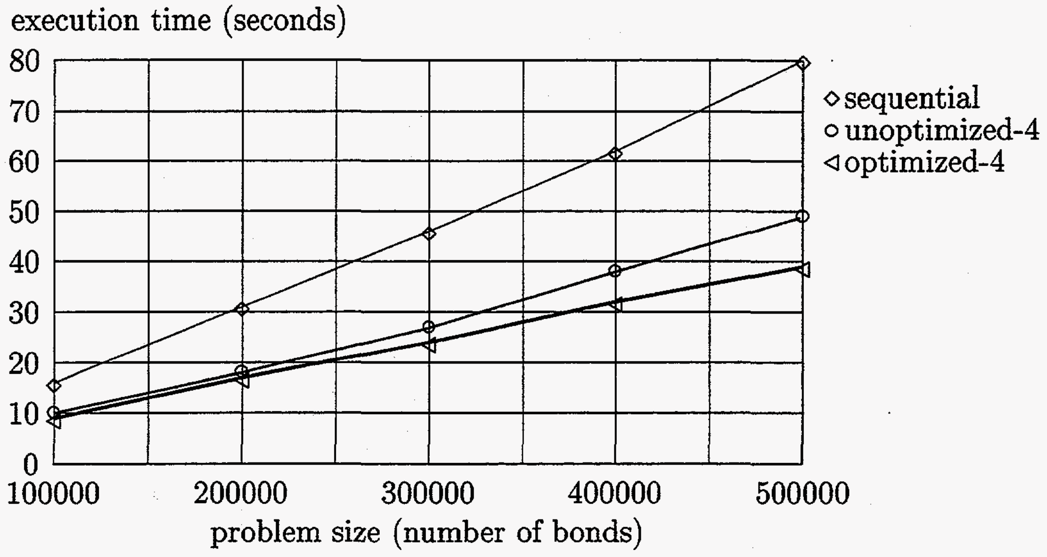


Figure 4.1: Execution times of prototype computation-reduction expressions.

example, it also runs faster; however, appreciable performance gains did not always appear.

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.

In response to these findings, a hand-coded version trading the locks for local temporaries was produced. Similar results were observed. Local memory usage was up from the locked version since processors had an entire intermediate copy. The speedups were however more consistent since lock contention did not increase with the number of processors.

Both methods have application areas. Local memory is not always a scarce commodity so using local temporaries is a good tradeoff on large memory machines. When memory is tight, the lock technique is more appropriate. The factors influenced the implementations available in Sisal 90.

4.2 Sisal 90 compiler results

User-defined reductions were added to the Sisal 90 language as described. Development of the *sequential* and *gathered* classes [19] continued during my addition of the *independent*, *accumulated* and *selected* subclasses. I measured one example of each subclass by direct specification with a *#pragma*. Chapter 2 of this document describes these samples.

1. *independent* array permutation build in Appendix A.1
2. *accumulated merge(lock)* scalar sum in Appendix A.2
3. *accumulated merge(master)* histogram in Appendix A.3
4. *selected merge(master)* first minimum in Appendix A.4
5. *selected merge(candidates)* first minimum index in Appendix A.5

After tuning the optimization implementations, I added analysis to verify the applicability of specific user pragmas. The compiler, however, does not accept the *-O3* flag or *#pragma returns parallel* to invoke nondirected optimization analysis.

4.2.1 Execution time and memory consumption

The OSC Sisal 1.2 compiler performs comparably with optimized Fortran in its application areas [4]. This study compares reductions written in Sisal 90 with equivalent

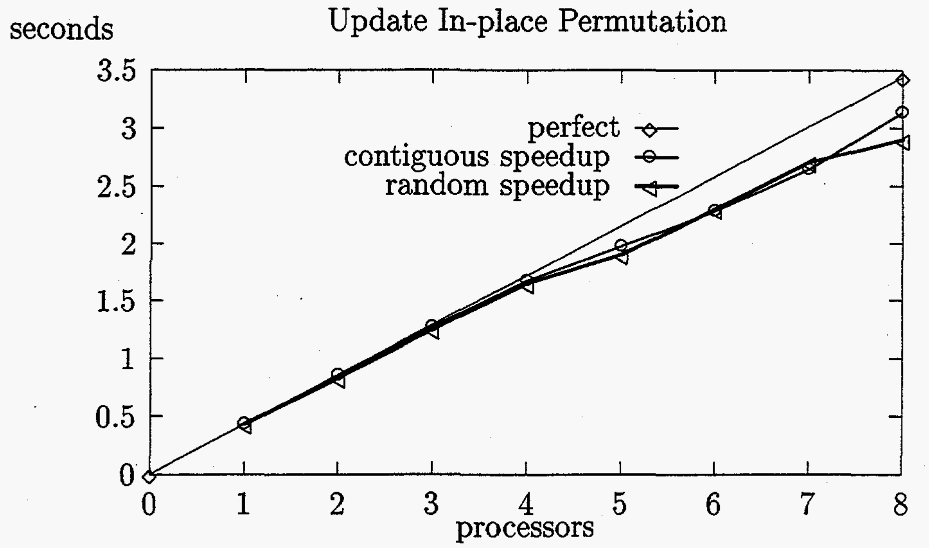


Figure 4.2: Execution times of permuted update in-place.

Sisal 1.2 benchmarks. For constructs unique to Sisal 90, simpler constructs in Sisal 1.2 establish an execution time and memory usage lower bound.

For example, a Sisal 90 *independent* array update in-place with permuted indexes can only be expressed in Sisal 1.2 as a sequential construct. The permutation update is however a parallel operation that can be optimized without any locks. Therefore, a parallel array build in-place in Sisal 1.2 sets the lower bound for comparison.

In the Sisal 1.2 array build in-place, each processor has a contiguous range of values to produce. There are only cache conflicts at the boundaries of the range. A permutation of indexes however, will supply each processor with random locations to update, which changes the cache utilization. To test this, an optimized implementation was run on an 8 processor SGI Challenge with and without actually permuting the indexes; both still ran from 1 to n just like the ordinary parallel array build. The caching effects for the two different versions are negligible on a heavily-loaded machine as shown in Figure 4.2. Both used the minimal array memory.

The execution time and memory is identical for the scalar Sisal 1.2 intrinsic and Sisal 90 user-defined reduction versions: least, greatest, sum, and product. The

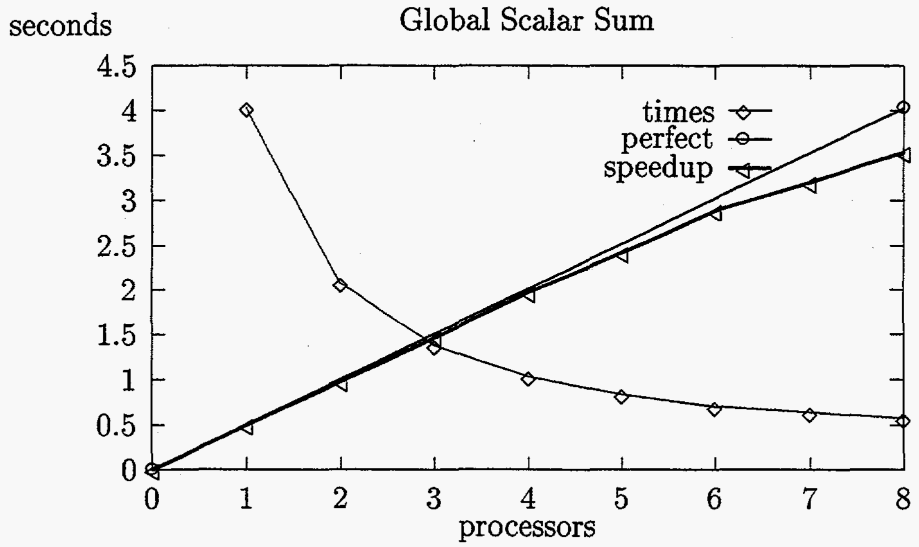


Figure 4.3: Execution times of scalar reductions.

emitted code is functionally equivalent, with differences only in the names of the temporaries used. This shows that the scalar *accumulated merge(lock)* and *selected merge(lock)* can be effectively implemented with linear speedup as shown in Figure 4.3 on a heavily loaded SGI (*load average: 30*).

The *accumulated merge(master)* histogram can actually be compared against a parallel Sisal 1.2 benchmark. Although Sisal is implicitly parallel, it is possible (using poor programming style) to write an outer loop that is sliced among all (1... nprocs) processors as shown in Appendix A.3. The performance and memory usage is nearly identical for the two versions. The Sisal 90 user-defined reduction with implicit parallelism is much simpler. The histogram reduction *histo_reduc* is initialized to size and each element *i* in the update array is reduced. Local histograms are computed on each processor and reduced sequentially on the master in both versions. The *selected merge(candidates)* implementation can also be expressed in Sisal 1.2 using a similar technique as shown in Appendix A.5. Comparable performance and memory use was observed.

Chapter 5

Conclusions

5.1 Conclusions

5.1.1 Performance and Expressibility

When executed on highly parallel machines, the presence of sequential expressions (such as the Sisal `for initial`) diminish the code's efficiency, an effect of Amdahl's Law. I initially sought to maximize the amount of parallelism. Through attempts to increase the execution performance of reduction expressions, the techniques minimized memory as well in many implementations. Several applications critically need reduced memory size to fit within the computer's real memory. Swapping large memory programs to disk as they run greatly curtails execution performance. In addition, the type of memory a program uses matters. Shared memory is a more precious commodity and usually much slower than local memory. Also, performance increases by using local memory because of lack of contention. The number of parallel instructions does not entirely reveal the total execution time of a program.

However, parallelizing overhead sometimes outweighs the gains. A good implementation must be chosen by careful analysis. In a locked reduction, if the parallel computation is thin and the reduction is thick, lock contention is common. In the reverse case with even amounts of work, the processors contend the first time and then are staggered. It must be considered that the greater the number of processors, the greater the opportunity for multiple processors to attempt to enter a critical section simultaneously. Time to broadcast initial values and gather results from multiple processors in comparison to expected speedup must analyzed. Without implementation classes and cost estimates or user pragmas feed into the analysis, parallel slowdown can occur. Fortunately, as problem sizes increase, the extra overhead can be paid.

This research was originally motivated by problems faced in certain molecular dynamics applications. Not only could they not be expressed succinctly, they would not run on the available hardware. By adding user-defined reductions and time and memory optimizations, these codes can now be run. The source code is very readable and completely portable. The problems are parallelized to run efficiently. Expressing the algorithm in a simple, implicitly parallel way leads to good performance. This achieves the goals of the Sisal 90 language by implementing a straightforward reduction syntax.

The definition of classes of reduction operations aided optimization. Analysis revealed opportunities for concurrent execution. Compiler directives were added to override default choices. A prototype in Sisal 1.2 showed the utility of the techniques, especially for memory savings. The Sisal 90 compiler incorporated optimizations for examples from each class of reduction. The generated codes demonstrated the efficiency of fusing parallel computations with sequential reductions.

5.2 Future Work

Additional research is almost exclusively in analysis. High performance implementations already appear in imperative code as programmers struggle for improvements. Analysis to automatically produce their results requires additional compiler development. Fewer hand-generated pragmas specifying an implementation will be needed. From a single source code, the compiler should produce better optimized programs for multiple machine targets.

The most straightforward extension of this work is to add an *-O3* compiler flag to automatically select an implementation from the three top-level classes: *sequential*, *gathered*, and *parallel*. An even more important addition would be to extend the simple *independent* analysis to handle a larger number of cases using work such as in [11]. Without explicit declaration, most permutations of complete index sets should still be recognized. If the indexes are data dependent and are calculated infrequently, a run-time check would also allow update-in-place optimizations.

Though HPF-2 Fortran is imperative, more functional elements are being added. This research should be applied to HPF reduction pragmas to eliminate them as well. Since the high performance computing community is coming upon some common themes, cross-fertilization will continue.

Bibliography

[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] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988. ISBN 0-13-484189-1.

[3] D. C. Cann. *The Optimizing SISAL Compiler.* Lawrence Livermore National Laboratory, Livermore, CA, April 1992.

[4] D. C. Cann. Retire Fortran? A Debate Rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[5] P. Castillo. A mathematical framework for reduction operators. Master's thesis, University of Puerto Rico, Mayaguez, Mayaguez, Puerto Rico, December 1994.

[6] Cray Research, Inc., 2360 Pilot Knob Road, Mendota Heights, MN 55120. *CFT77 Reference Manual*, sr-0018 edition, October 1988.

[7] S. Denton, J. Feo, and P. Miller. Realizing parallel reduction operations in SISAL 1.2. In *Parallel Architectures and Compilation Techniques (PACT) '94.* IFIP Transactions, 1994.

[8] J. Dongarra, D. Walker, et al. Mpi: A message-passing interface standard. Technical Report 1.0, Message Passing Interface Forum, http://www.mcs.anl.gov/mpi/index.html, 5 May 1995.

[9] J. T. Feo. The livermore loops in sisal. Technical Report UCID-21159, Lawrence Livermore National Laboratory, Livermore, CA, August 1987.

[10] J. T. Feo, P. J. Miller, S. Skedzielewski, S. M. Denton, and C. J. Solomon. Sisal 90. In *High Performance Functional Computing Conference*, Livermore, CA, April 1995. Lawrence Livermore National Laboratory.

[11] D. Garza and W. Bohm. Uniqueness analysis of array comprehensions using the omega test. In *Proceedings of the First International Static Analysis Symposium*. Lecture Notes in Computer Science 864, Springer-Verlag, 1994.

[12] P. Hudak and J. H. Fasel. Report on the programming language Haskell: A non-strict, purely functional language. In *SIGPLAN Notices*. ACM, May 1992.

[13] High Performance Fortran Forum. *High Performance Fortran Journal of Development: Version 1.0.* Rice University, Houston, TX, May 1993.

[14] High Performance Fortran Forum. *High Performance Fortran Language Specification: Version 1.0.* Rice University, Houston, TX, May 1993.

[15] High Performance Fortran Forum. *HPF-2 Scope of Activities and Motivating Applications: Version 0.8.* Rice University, Houston, TX, November 1994.

[16] 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.

[17] J. Sequel and D. Bollman. Even and quarter-even prime length symmetric ffts and their sisal implementations. In *Proceedings Sisal '93 CONF-9310206*. Lawrence Livermore National Laboratory, Springer-Verlag, October 1993.

[18] Silicon Graphics Inc., Mountain View, CA. *Fortran 77 Programmers's Guide*, 007-0711-030 edition, 1991.

[19] S. Skedzielewski, J. Feo, P. Miller, and S. Denton. *Sisal 90 Language Reference Manual.* Lawrence Livermore National Laboratory, Livermore, CA, 0.9 edition, April 1995.

[20] S. Skedzielewski and J. Glauert. IF1: An intermediate form for applicative languages. Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985.

[21] C.-W. Tseng and J. H. Saltz. Compilation and runtime support for massively parallel processors. Supercomputing '93 Tutorial F3, November 1993.

[22] K. Warren, B. Gorda, and E. D. B. III. Using pddp. Technical Report DRAFT for review, Lawrence Livermore National Laboratory, Livermore, CA, October 1984.

[23] M. L. Welcome, S. K. Skedzielewski, R. K. Yates, and J. E. Ranelletti. If2: An applicative language intermediate form with explicit memory management. Manual M-195, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.

Appendix A

Code Samples

All examples are shown with explicit pragmas to identify their class. With analysis, these pragmas are not required.

A.1 Parallel array permutation build

Index set 1. . .n is permuted and used to produce an array of computed values in-place.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sisal 90 array permutation build reduction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

reduction square_shift(n,shift:integer repeat i:integer returns
    array[integer])
    for initial
        a := array_fill(0, n-1, 0);
    repeat
        a := old a[ (i+shift) mod n ! i*i ]
    returns value of a
    end for
end reduction

#pragma returns independent
function main(n, shift:integer returns array[integer])
    for i in 1, n
    returns square_shift(n, shift) of i
    end for
end function % main

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sisal 1.2 array reduction (not equivalent but used as benchmark)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
define main

function main(n:integer returns array[integer])
    for i in 1, n
    returns array of i*i
    end for
end function % main
```

A.2 Parallel global sum

A local array slice sum from each processor is added together on the master processor.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sisal 90 global sum reduction (without using predefined reduction)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#pragma reduction merge(lock)
reduction tsum(initval:integer repeat x:integer returns integer)
    for initial
        total := initval
    repeat
        total := old total + x
    returns value of total
    end for
end reduction

#pragma returns accumulated
function main(n:integer returns integer)
    for i in 1, n
    returns tsum(10) of i
    end for
end function % main

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sisal 1.2 global sum reduction (using predefined reduction)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

define main

function main(n:integer returns integer)
    10 + for i in 1, n
         returns value of sum i
         end for
end function % main
```

A.3 Parallel counting histogram

A local counting histogram array is produced for each processor slice and merged columnwise on the master processor.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sisal 90 parallel histogram reduction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#pragma reduction merge(master)
reduction histo(histo_size:integer repeat histo_update:integer
    returns array[integer])
    for initial
        bins := array_fill(1, histo_size, 0);
    repeat
        bins := old bins[ histo_update ! old bins[histo_update]+1 ]
    returns value of bins
    end for
end reduction

#pragma returns accumulated
function main(histo_size:integer; histo_updates:array[integer]
    returns array[integer])
    for i in histo_updates
    returns histo(histo_size) of i
    end for
end function

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sisal 1.2 parallel histogram reduction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

define main

function histo_func(histo_size:integer; histo_updates:array[integer];
    start, finish:integer returns array[integer])
    for initial
        i := start;
        local_histo := array_fill(1, histo_size, 0);
    while i<=finish repeat
        i := old i + 1;
        j := histo_updates[old i];
        local_histo := old local_histo[j: old local_histo[j]+1];
    returns value of local_histo
    end for
end function

function main(nproc:integer; histo_size:integer;
        histo_updates:array[integer] returns array[integer])
        let
            %
            % Create an array from local histograms (in parallel)
            %
            slice_size := array_size(histo_updates)/nproc;
            remainder := mod(array_size(histo_updates), nproc);
            local_histos :=
                for i in 1, nproc
                    start := (i-1)*slice_size + 1 + min(i-1, remainder);
                    finish := i*slice_size + min(i, remainder);
                    %
                    % Create a local histogram (sequential)
                    %
                    local_histo := histo_func(histo_size, histo_updates,
                        start, finish)
                returns array of local_histo
                end for
        in
            %
            % Combine the local histograms (sequential)
            %
            for j in 1, histo_size
                column_sum :=
                    for i in 1, nproc
                    returns value of sum local_histos[i][j]
                    end for
            returns value of catenate array [1: column_sum]
            end for
        end let
    end function
```

A.4 Parallel first minimum

The minimum value on each processor is selected. The master processor then finds the minimum of all candidates sequentially.

```
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Sisal 90 first minimum reduction (without using predefined)
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    #pragma reduction merge(master)
    reduction first_min(repeat x:integer returns integer)
        for initial
            min_val := $MAXINT;
        repeat
            min_val := if x<min_val then x else old min_val;
    returns value of min_val
    end for
end reduction

#pragma returns selected
function main(n:integer returns integer)
    for i in 1, n
    returns first_min of i
    end for
end function % main

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sisal 1.2 first minimum reduction (using predefined reduction)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

define main

function main(n:integer returns integer)
    for i in 1, n
    returns value of least i
    end for
end function % main
```

A.5 Parallel first minimum index

Index and value of first occurrence of the minimum value on each processor slice is returned to the master processor. Master sequentially reruns the reduction and returns the index.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sisal 90 parallel first minimim index reduction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

#pragma reduction merge(candidates)
reduction firstmin_index(repeat x,i:integer returns integer)
    for initial
        min_val := $MAXINT;
        min_index := 0;
    repeat
        min_val, min_index :=
            if x < old min_val then x, i
            else old min_val, old min_index
            end if;
    returns value of min_index
    end for
end reduction
```

```
#pragma returns selected
function main(a:array[integer] returns integer)
    for i in array_size(a)
    returns firstmin_index of a[i], i
    end for
end function

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Sisal 1.2 parallel first minimim index reduction
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

define main

function local_cand(a:array[integer]; start, finish:integer
    returns integer, integer)
    for initial
        i := start;
        local_val := a[start];
        local_index := start;
    while i<=finish repeat
        i := old i + 1;
        local_val, local_index :=
            if a[old i] < old local_val then
                a[old i], old i
            else
                old local_val, old local_index
            end if
    returns value of local_val
            value of local_index
    end for
end function

function main(nproc:integer; a:array[integer] returns integer)
    let
        %
        % create an array of local candidates (in parallel)
        %
        slice_size := array_size(a)/nproc;
        remainder := mod(array_size(a), nproc);
        local_cand_vals, local_cand_indexes :=
            for i in 1, nproc
                start := (i-1)*slice_size + 1 + min(i-1, remainder);
                finish := i*slice_size + min(i, remainder);
                %
                % create a local candidate (sequential)
                %
                local_cand_val, local_cand_index :=
                    local_cand(a, start, finish);
            returns array of local_cand_val
```

```
                    array of local_cand_index
            end for
    in
        %
        % Rerun reduction on all candidates (sequentially)
        %
        for initial
            i := 1;
            final_val := local_cand_vals[i];
            final_index := local_cand_indexes[i];
        while i<=nproc repeat
            final_val, final_index :=
                if local_cand_vals[old i] < old final_val then
                    local_cand_vals[old i], local_cand_indexes[old i]
                else
                    old final_val, old final_index
                end if;
            i := old i + 1
        returns value of final_index
        end for
    end let
end function
```

Technical Information Department · Lawrence Livermore National Laboratory
University of California · Livermore, California 94550