UCRL-JC-115029
PREPRINT

Implementation and Performance of a Domain
Decomposition Algorithm in Sisal

Thomas DeBoni and John Feo
Lawrence Livermore National Laboratory

Garry Rodrigue
University of California Davis/Livermore

and
John Muller
State University of New York, Stonybrook

This paper was prepared for submittal to the
Hawaii International Conference on System Sciences,
Maui, Hawaii, January 4-7, 1994

September 23, 1993

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.

RECEIVED
SEP 23 1993

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

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.

Implementation and Performance of a Domain Decomposition Algorithm in Sisal

Thomas DeBoni and John Feo
Lawrence Livermore National Laboratory
Livermore, CA 94550

Garry Rodrigue
University of California, Davis/Livermore
Livermore, CA 94550

John Muller
State University of New York, StonyBrook
StonyBrook, NY 11790

Abstract

Sisal is a general-purpose functional language that hides the complexity of parallel processing, expedites parallel program development, and guarantees determinacy. Parallelism and management of concurrent tasks are realized automatically by the compiler and runtime system. Spatial domain decomposition is a widely-used method that focuses computational resources on the most active, or important, areas of a domain. Many complex programming issues are introduced in parallelizing this method, including: dynamic spatial refinement, dynamic grid partitioning and fusion, task distribution, data distribution, and load balancing. In this paper, we describe a spatial domain decomposition algorithm programmed in Sisal. We explain the compilation process, and present the execution performance of the resultant code on two different multiprocessor systems: a multiprocessor vector supercomputer, and a cache-coherent scalar multiprocessor.

1.0 Introduction

A high-level programming language is a tool that the programmer uses to encode his program for execution on a computer system. Since programming, especially parallel programming, is difficult [10], high-level programming languages provide abstractions intended to simplify the programming task. Most conventional languages were

developed for sequential computing, and do not support parallel abstractions. Proposed extensions increase program development costs, codify architectural characteristics, and are susceptible to subtle time-dependent errors. Functional languages are a class of high-level languages developed specifically for parallel computing. Based on mathematical principles, their semantics hide the complexities of parallel processing, expedite parallel program development, and guarantee determinacy. Parallelism and management of concurrent tasks are not the programmer's responsibility; instead, they are realized automatically by the compiler and runtime system. Functional programs are shorter, simpler, and more portable than equivalent programs written in conventional languages. Perhaps the biggest advantage of functional languages is that they reduce the cost of parallel program development.

Although programming costs are important, speed is paramount. Machine performance is still the measure of success used by most scientific programmers. The "conservative" syntax of Sisal [7], a general-purpose functional language developed by Lawrence Livermore National Laboratory and Colorado State University, supports compile-time optimizations that remove the inefficiencies associated with functional semantics (dynamic memory management and excessive copying). The performance of most sequential codes written in Sisal is comparable to equivalent code written in traditional imperative languages [5]. The Sisal scientific programmer may enjoy the benefits of functional programming without sacrificing performance.

Programming in Sisal does require programmers to learn a new language and, more importantly, a new way of thinking about programs. To "think functional" involves reverting to a thought process closer to mathematics, the fundamental language of science. Programming in such a style keeps the programmer closer to the task at hand, and eliminates many of the opportunities for errors that occur in translation between the domain of discourse and the programming language. The most difficult task is to unlearn the non-mathematical semantics of imperative languages. For example, the Fortran statement

$$x = x + 1$$

has no mathematical meaning.

In this paper, we relate our experiences developing a spatial domain decomposition code in Sisal. Domain decomposition is a widely-used method that focuses computational resources on the most active, or important, areas of a physical system. Many complex programming issues are introduced in parallelizing this method, including: dynamic spatial refinement, dynamic grid partitioning and fusion, task distribution, data distribution, and load balancing. To achieve high-performance, these issues must be addressed. We found Sisal's dynamic nature and functional semantics simplified our programming task. In this paper, we discuss how the Optimizing Sisal Compiler addresses these issues to realize efficient implementations of our code on two different multiprocessor systems: a multiprocessor vector supercomputer and a cache-coherent scalar multiprocessor. Since we made no source changes in porting the code, this study demonstrates the portability and transportability of Sisal programs.

In Section 2, we describe the general principles of functional languages and give a brief overview of the most important Sisal features used in developing our code. In Section 3, we describe the general methodology of domain decomposition, and describe the heat diffusion problem and its solution. Section 4 discusses key sections of the code we wrote to implement domain decomposition in Sisal. In sections 5 and 6, we describe our experiments and present the results of our study. Section 7 discusses moving the code to distributed memory machines, anticipates future work, and presents our conclusions.

2.0 Sisal

Although, research in functional programming languages began over thirty years ago with the definition of Lisp, functional languages are still not widely used. Scientific programmers have hitherto considered functional languages inappropriate for several reasons:

1. Lack of arrays and array operations. The list is the fundamental data structure of most functional languages. This data structure does not support random access of data.
2. Lack of iterative and parallel loop constructs. Most functional languages support only recursion and list comprehensions. The execution costs of these forms of control are significant, compared to loops.
3. Single-assignment semantics. Functional programs bind names to values. Once defined, a name can not be redefined. To modify an aggregate object requires copying the object and then modifying the copy. Copying large arrays reduces performance.
4. Lack of memory management operations. The size and shape of aggregate objects are not pre-declared by the user. The copying of intermediate structures, and the dy-

namic allocation and deallocation of memory at runtime degrades execution performance.

But, these perceptions can be addressed by appropriately defined functional languages [4]. Sisal includes arrays, array operations, and both an iterative and parallel loop construct, without sacrificing functional semantics. Its syntax is close in style to that of formal mathematics. A Sisal program is a specification of what must be computed, couched in declarations of relationships among the units of the program, rather than a specification how the computation must proceed in a sequence of state transformations. The programmer remains closer to the natural domain of discourse in the programming task, reducing the opportunities for errors of translation from problem specification to solution method to program. Sisal's functional semantics enable the compiler to ascertain with certainty the meaning and effects of program units, and to decide how best to optimize execution and resource management. Compile-time analysis routines eliminate the adverse effects of single-assignment semantics and the lack of memory management operations.

A Sisal program is a series of function definitions and applications. Each program unit—be it a numerical or logical expression, a compound statement, or a function—may receive inputs only from its explicit arguments, and may affect the rest of the computation only through its outputs. The time and method of execution cannot alter the results returned by a program unit. This attribute precludes the possibility of race conditions, or time-dependent behavior, and guarantees determinacy.

As mentioned previously, Sisal supports arrays and array operations. The programmer defines array types using the type statement,

```
type Onedim = array [real];  
type Twodim = array [Onedim];
```

The first statement defines a type name for all one-dimensional arrays of real values. The second statement defines a second type, an array of arrays (i.e., a two-dimensional array) of real values. Notice that the array type does not include either index bounds or size. These attributes are defined at runtime, and may be different for each instance of the type. If an array expression returns n values, the resultant array has size n . An array's index ranges are similarly defined by the expression. The dynamic nature of Sisal arrays permits the same source code to execute correctly over all problem sizes without wasting memory.

A second benefit is more subtle. Since a multi-dimensional array is an array of arrays, each component array may have a different lower bound and size—Sisal arrays are *ragged*. Scientific applications typically represent n -dimensional physical systems by n -dimensional arrays whose components correspond to components of the physical system. Domain decomposition applications dynamically increase the number of evaluation loci in active regions of the system, and decrease the number in inactive

regions. The refinement of a physical region results in a corresponding increase in the size of the respective array component. Dynamic changes in the sizes of array components are supported naturally by Sisal's ragged arrays.

A second important Sisal feature is its pair of looping constructs. The first of these, the **for** expression,

```
for <range generator>
  <loop body>
returns <returns clause>
end for
```

is a parallel construct. An instance of the loop body is executed for each element of the range. The returns clause defines how values defined in the body are to be reduced to scalar values or gathered to build aggregate objects. The expression's syntax and semantics guarantee that instances of the loop body are data independent. Compile-time analysis and runtime decisions determine the most efficient way to execute the instances on a given architecture. Possible choices are sequential, vector, concurrent, or concurrent-vector. Typically, if work is insufficient to warrant parallel execution, the runtime system will execute the expression as a sequential task on one processor. If there is sufficient work and the work performed per iteration is approximately equal, the runtime system will partition the expression into equal size slices (number of iterations divided by number of workers) and execute the slices concurrently on different processors. If the work is not equivalent, the runtime system may slice the expression into more pieces than there are workers and assign slices to workers as they become free. This last alternative increases scheduling overhead, but may reduce overall execution time by improving load balance. In domain decomposition, the dynamically changing structures representing the physical system under simulation preclude, or at best render impractical, any static partitioning of data and scheduling of work.

The second loop form is the **for initial** expression,

```
for initial
  <initialization>
while <test> repeat
  <loop body>
returns <returns clause>
end for
```

This construct permits loop carried dependencies, but retains single assignment semantics. The initialization part of the loop constitutes the first "iteration", defining the initial states of the loop-carried values. The loop body is executed repeatedly until the test terminates execution. An instance of the body may refer to any loop-carried name defined in the previous instance by prefixing the name with the keyword **old**. Thus, **old a** refers to the value of **a** on the previous iteration. Note that defining the value of **a** on the present iteration does not destroy the old value. The rebinding of loop-carried names to values is implicit and occurs between iterations. Since scientific applications rely heavily on iteration to model convergent and time-dependent processes, the availability

of an iterative loop expression in Sisal improves performance and expressability. Functional languages that lack iteration must use recursive constructs, and may incur significant performance costs on supercomputer systems.

3.0 Domain Decomposition

Domain decomposition is a general methodology for partitioning a computational problem into pieces of manageable size for semi-independent solution. This technique is used in numerical simulations of physical systems that exhibit a "natural" decomposition. A single system may contain or encompass multiple subdomains of different materials, of different physical or chemical processes, of different spatial or temporal resolutions, or that need different machine or system resources for solution. Adaptive Mesh Refinement [1] is a domain decomposition methodology used in computational fluid dynamics to simulate complex systems. This method adjusts both the spatial and temporal resolution of subdomains in response to features or activities that occur naturally in the problem. For example, in shock studies, the regions of simulated medium through which a shock wave is moving may be refined in both space and time, so that a more accurate simulation may be obtained in those regions. Computational resources are saved by maintaining coarser resolutions in regions of lesser activity. Thus, the method focuses computational resources where they are needed without unnecessary overhead.

The advantage of domain decomposition methods is that the subdomains may be solved in parallel, synchronizing only to communicate boundary data with their neighbors. Typically, the computation to communication ratio is $O(n^2)$, and is large enough to warrant parallel execution on most parallel computer systems. However, the dynamic nature of domain decomposition introduces many complex parallel programming issues that make realizing its full potential difficult. First, the sizes of data objects grow and shrink dynamically. Most conventional languages support only objects of fixed size and require the programmer to explicitly manage memory. Second, the physical boundaries and shapes of subdomains may change during program execution, requiring changes in loop index values and data access patterns. As the shapes and sizes of subdomains change, the original distribution of data and assignment of tasks to processors may become increasingly inefficient. As communication delays increase and load imbalances occur, the benefits of parallel processing decrease. The application may have to redistribute data and tasks to processors every few steps to maintain efficiency.

To study how functional programming addresses these issues, we wrote a Sisal application to solve the heat problem using domain decomposition. Let G be a rectangular grid with $[0 \dots imax + 1]$ by $[0 \dots jmax + 1]$ zones. Assume that each zone on the boundary of G is held at a

constant temperature. Then at iteration i , the temperature, T , at each interior zone (x, y) in G is the arithmetic average of the temperatures at iteration $(i - 1)$ of the zones to the north, south, east, and west. For example, consider the grid in Figure 1. The temperatures of the six interior zones are

$$\begin{aligned}
 T_{1,1} &= \frac{1}{4}(0 + 0 + T_{1,2} + T_{2,1}) \\
 T_{1,2} &= \frac{1}{4}(T_{1,1} + 0 + T_{1,3} + T_{2,2}) \\
 T_{1,3} &= \frac{1}{4}(T_{1,2} + 0 + 0 + T_{2,3}) \\
 T_{2,1} &= \frac{1}{4}(0 + T_{1,1} + T_{2,2} + 0) \\
 T_{2,2} &= \frac{1}{4}(T_{2,1} + T_{1,2} + T_{2,3} + 0) \\
 T_{2,3} &= \frac{1}{4}(T_{2,2} + T_{1,3} + 0 + 0)
 \end{aligned} \tag{1}$$

We can express Equation (1) as the matrix equation

$$\begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 \\ 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix} \times \begin{bmatrix} T_{1,1} \\ T_{1,2} \\ T_{1,3} \\ T_{2,1} \\ T_{2,2} \\ T_{2,3} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2}$$

The coefficient matrix is a blocked tridiagonal matrix. The number of rows, or linear equations, is $imax * jmax$. The number of blocks is $jmax$, and the size of each block is $imax$.

To effect domain decomposition, we divide the problem grid into a fixed number of subgrids or panels. We can solve the subgrids independently, given appropriate boundary information. The application's basic data structure, then, is a fixed sized two dimensional array of subgrids, or panels. Initially, each panel contains the same number of zones. At each step of the simulation, we allow the number of zones in each panel to grow or shrink in both dimensions by a constant factor r . If a panel is refined, the number of zones grows by r^2 ; if a panel is derefined, the number of zones shrinks by r^2 (n.b. – we never let the number of zones shrink below the initial value). Figure 2 shows a problem grid consisting of a 4 by 5 array of panels. The refinement factor is 2. When panels of different spatial resolution abut each other, we average or

replicate shared data to compensate for the different resolutions.

0	0	0	0	0
0	$T(1,1)$	$T(1,2)$	$T(1,3)$	0
0	$T(2,1)$	$T(2,2)$	$T(2,3)$	0
0	0	0	0	0

Figure 1 – A 4 by 5 grid

Figure 2 – A 4 by 5 grid of panels, $r = 2$

4.0 The Sisal program

Since functional programs are closer to mathematical derivations than conventional programs, composing numerical programs is easier. The largest impediment to progress for the novice is the change of mindset from imperative to declarative algorithm expression. Control flow is not a concern, except in the sequential loop form where it is implied by the loop carried dependencies. The sequential transformation of state implicit in imperative programming essentially disappears. The changes of design and programming habits that must be made to program in Sisal are mostly the unlearning of conventional programming practice, and the return to mathematics. We found that the step from mathematics to code is small,

often merely a simple transliteration of the method or algorithm to Sisal program statements. In the following sections, we discuss the key sections of code in our application.

4.1 Function main

First we define our data types. We use two user defined types,

```
type panel = array [real];
type grid = array [array [panel]];
```

The first statement defines an array of reals, and the second statement defines a two-dimensional array of panels. Notice that `panel` is one-dimensional. We found it more convenient to concatenate the rows of a physical panel into a one-dimensional array than to store it in its more natural two-dimensional form.

The main function reads the program's inputs and returns the calculated temperature for each zone in G (i.e., a grid). The function's header defines the formal name and type for each input parameter, and gives the type of each result value. The function's body is a **for initial** expression whose iterations constitute the steps of the convergent numerical solution to the heat problem. Following is a slightly simplified version of the body of the main function.

```
for initial
  step := 0;
  G := initialize(number_of_panels,
                 number_of_zones,
                 boundaries)
while step < max_steps repeat
  step := old step + 1;
  G_temp := solve(..., old G);
  flags := statistics(..., G_temp);
  G := refine(..., flags, G_temp)
returns value of G
end for
```

Three functions comprise the body. The first function solves for new temperature values given the old temperature values. The second function computes certain statistics for each panel and, based on these statistics, decides which panels should be refined or derefined on the next iteration. The third statement refines or derefines the panels, and constructs a new grid. The loop iterates a total of `max_steps` times. The result returned by the **for initial** expression is the value of G on the last iteration.

The program's structure is apparent from the outer loop. Since there is a strict data dependency between steps of the simulation, there is no parallelism at this level—parallelism is nested more deeply within the functions. In the next two sections, we present **function solve** and **function refine** in detail.

4.2 Function Solve

function solve is the mathematical kernel of the application. It takes the temperature values at step i and returns the temperature values at step $i + 1$. It solves Equation (2) for each panel of G independently using the conjugate gradient method. A simplified version of **function solve** follows.

```
let
  top := grid[0];

  core :=
    for each_row in G cross each_panel in row
      first := grid[i,0];
      panel := cg(...,A(...),grid[i, j],B(...));
      last := grid[i, nPr + 1]
    returns array of
      Paste_Edges(first, panels, last)
  end for;

  bottom := grid[nPr + 1]

in
  Paste_Top_and_Bottom(top, core, bottom)
end let
```

A **let** expression comprises the function's body. The **let** clause defines the core of the new grid, and the **in** clause pastes the top and bottom boundaries onto the new core. The need to paste together aggregate objects is a consequence of the lack of memory in Sisal programs. The components of aggregate objects must first be defined, and then concatenated together to form larger objects. In a naive implementation, such concatenations would introduce copying; however, the Sisal compiler determines the final size of the aggregate object, in this case the new grid of temperatures, and inserts code to preallocate space. The location of each component within G is precomputed and passed to the task responsible for defining that component. Consequently, each component is built in place as in an imperative program. The build-in-place analysis is described in [5,9].

Since Equation (2) may be solved independently for each panel, we use a doubly-nested **for** expression to compute the new core. The instances of the body of the nested **for** expression will execute in parallel. Each instance calls the **function cg** to solve a particular instance of Equation (2). The functions A and B return the appropriate coefficient matrix and right-hand-side vector for the given instance, respectively. Notice that parallelism has been introduced implicitly by use of the **for** expressions. The decomposition of the expression into tasks, the scheduling of tasks to processors, and the management of tasks occurs automatically at runtime.

The **function cg** is an exact transliteration of the mathematical derivation of the conjugate gradient method. In the formulation that follows, A is the coefficient matrix of size n by n with block size m , and x is the vector of un-

known temperatures. It is important to keep in mind that the matrix and the grid are not the same thing. The grid consists of $(imax + 2) * (jmax + 2)$ zones of which $n = imax * jmax$ must have their temperatures calculated. Thus, the system of linear equations is n equations in n unknowns. Given an estimate x_i , the conjugate gradient method computes a new estimate x_{i+1} as follows:

$$Mz_i = b - Ax_i$$

$$z_i = M^{-1}Mz_i$$

$$t_i = \begin{cases} 0 & i = 0 \\ \frac{z_i^T Mz_i}{z_{i-1}^T Mz_{i-1}} & i > 0 \end{cases}$$

$$p_i = z_i + t_i p_{i-1}$$

$$a_i = \frac{z_i^T Mz_i}{p_i^T A p_i}$$

$$x_{i+1} = x_i + a_i p_i$$

where M is the matrix containing the major diagonal of A with zeros elsewhere, and z , p , and x are vectors. The method iterates until the inner product of $(b - Ax)$ with itself is less than some error value (we used 10^{-12}).

The Sisal code is

```

for initial
  Ax := Times5(n, m,
               A1, A2, A3, A4, A5, x0);
  Mz := Minus(n, b, Ax);
  z := Times(n, Minv, Mz);
  zMz := Inner(n, z, Mz);
  t := 0.0d0;
  p := z;
  Ap := Times5(r, m,
               A1, A2, A3, A4, A5, p);
  a := zMz / Inner(n, p, Ap);
  x := Plus_Times(n, x0, a, p)
while Inner(n, Mz, Mz) > 1.0d0-12 repeat
  Ax := Times5(n, m,
               A1, A2, A3, A4, A5, old x);
  Mz := Minus(n, b, Ax);
  z := Times(n, Minv, Mz);
  zMz := Inner(n, z, Mz);
  t := zMz / old zMz;
  p := Plus_Times(n, z, t, old p);
  Ap := Times5(n, m,
               A1, A2, A3, A4, A5, p);
  a := zMz / Inner(n, p, Ap);
  x := Plus_Times(n, old x, a, p)
returns value of x
end for

```

where n is the rank of A , m is the rank of a block of A , $A1$ through $A5$ are the non-zero diagonals of A , $Minv$ is the inverse of the major diagonal of A , and $x0$ is the original estimate of x . The function is almost a one-to-one

transliteration of the mathematical equations. The functions `Times5`, `Minus`, `Times`, `Inner`, and `Plus_Times` are all vector-vector computations consisting of a single `for` expression. The functions may be compiled as sequential, vector, parallel, or vector-parallel code. Which is best depends on the functions' nesting level and machine architecture. Since `function cg` is called from within the doubly-nested `for` expression in `function solve`, the Sisal compiler does not parallelize the function; instead, it generates sequential code on scalar processors and vector code on vector processors. The compiler decides how best to compile the function, and the implementation of the decision is left to the runtime system.

4.3 Function Refine

This function is responsible for refining and derefining panels. It performs a large number of memory accesses, but virtually no mathematical computations. Nevertheless, it may be the most difficult routine to write and most expensive routine to execute. Conventional programming languages provide ample mathematical capabilities, but offer little help managing dynamic data structures. Modern computer systems can execute hundreds of millions of floating point operations per second, but slow down significantly when executing large numbers of memory operations. Sisal's functional semantics and dynamic, ragged arrays make writing this function easier.

Since panels may be refined and derefined concurrently, a single `for` expression comprises the function:

```

for panel in G
returns array of
  if flag = -1 then
    derefine(panel)
  elseif flag = 0 then
    panel.1
  else
    refine(panel)
  end if
end for

```

Since G is of type grid, the result of the `for` expression is also of type grid. The `function derefine` averages successive clusters of four elements returning a panel one-fourth the size of the input panel. The `function refine` duplicates each element four times returning a panel four times the size of the input panel.

The key to the efficient implementation of this routine is how Sisal stores multi-dimensional arrays [2,9]. Figure 3 illustrates the internal structure of the grid shown in Figure 2 (note the raggedness of the third level arrays). The compiler inserts code to precompute the size of the resultant grid and to preallocate storage for the two levels of pointers. The compiler also inserts code in the functions `derefine` and `refine` to precompute the sizes of their resultant arrays and preallocate the required memory. The address of the second level pointer for the i -th panel is passed to i -th instance of the loop body. If the panel is refined or derefined, the memory manager is called for

new memory and the new array is stored in that space. The program then assigns the start address of the new space to the pointer. If the panel remains unchanged, the program assigns the start address of the current panel to the pointer, and increments the panel's reference count by one. This last operation is necessary to prevent the panel from being thrown away with the old grid. The deallocation and allocation of memory in Sisal programs is explained in detail in [2,5]. Here, it suffices to state that the memory management operations execute quickly, and the system does not leak memory.

Figure 3 – The internal structure of a 4 x 5 grid of panels, $r = 2$

5.0 Experiments

We designed experiments to study the code's raw speed, parallelism, and transportability. We ran the experiments on an eight processor Cray C-90 vector supercomputer and a four processor, cache-coherent SGI Iris 340. We identified four factors that could affect program performance. The first factor was the number of panels per row and column of the physical grid. Varying the ratio of rows to columns exposed a bias in the Sisal system in favor of row parallelism. Since the potential work imbalance among panels is not evident from the code, the compiler parallelized only the outer, or row, loop of the doubly-nested `for` expression in `function solve`. The compiler determined correctly that parallelizing only the outer loop would saturate the target machine; but did not appreciate the possibility of load imbalance. We forced the compiler to parallelize the inner loop also by setting a command line option. This directive, requiring no changes to the code, was sufficient to achieve the performance and speedups reported in the next section. Sisal's

bias in favor of row parallelism is well known, and some corrective steps have been taken.

The second factor was the minimum number of zones per row and column in each panel. By varying the number of zones per panel, we varied the size of the set of linear equations solved by `function eg`. Spending more time in a program's mathematical kernel usually improves speedups. In our application, the ratio of work to overhead is so large even for small problems, that we saw no appreciable improvement in speedups as we varied the minimum panel size. However, longer vectors did increase the program's raw speed on the C-90.

Increasing the refinement factor increases the work differential between refined and unrefined panels, thereby, creating larger load imbalances. Unfortunately, as the refinement factor increases, the program's memory size grows quadratically. Our systems did not have sufficient memory to fully explore refinement factors greater than 3. Since we saw no degradation in performance due to load imbalances, we left the value at 2.

The fourth factor was different boundary conditions. By varying the constant temperature values on the boundaries, we altered the diffusion of heat across the grid, thereby, generating different refinement and derefinement patterns. We expected the different patterns to reveal any bias in the Sisal system in favor of row or column parallelism, or inability to load balance equally well in both dimensions. Once we forced the Sisal compiler to parallelize the inner loop of the `for` expression in `function solve`, we saw no favoritism towards either rows or columns.

6.0 Results

This study is three studies in one. We report on the cost to develop the domain decomposition algorithm in Sisal, the compiler's ability to optimize the code for different multiprocessor systems, and the code's performance on those systems.

6.1 Programming costs

Once we understood what we wanted to write, it took approximately two-man weeks to write and debug the code (including rewrites to improve performance). Our estimate is approximate because parts of the program evolved from another Sisal code. Note that this code was written in Sisal from first principles, and is not the translation of an extant imperative code. The Sisal program is 802 lines of source code comprising 32 functions. Although, we are experienced Sisal programmers, similar productivity has been reported by novice Sisal programmers [6].

TWINE [8], the Sisal debugger, and the diagnostics provided by the compiler and runtime system [3] assisted our development effort. Since functional programs are determinant, they may be debugged on a single processor

without concern. This attribute of functional languages is underappreciated. Because of the possibility of time dependent errors, an imperative parallel program must be debugged on multiple processors. The Sisal compiler and runtime system are heavily instrumented, and generate extensive diagnostic information. We used that information to identify parts of the code that optimized or performed poorly. In such instances, we tried different formulations of the mathematics until we achieved good performance. Such prototyping is typical.

6.2 Compiler optimizations

There are 72 loops in the source code. On the C-90 the compiler vectorized 51 loops and parallelized 8 loops. On the SGI machine, the compiler fused 27 loops and parallelized 8 loops. Loop fusion is an important optimization. It decreases loop overhead, increases the work to overhead ratio in loops, removes stores and loads, and eliminates intermediate array objects. We have found that loop fusion on the Cray machines tends to hurt vectorization, and have turned the optimization off.

The code constructs 152 arrays. The Sisal compiler preallocates storage and builds 150 of the arrays in place. The two arrays not built-in-place occur in the initialization section of the code, and are built only once. A naive implementation of the code would include 94 array copy operations and 1147 reference count operations. The compiler eliminates all copy operations and 889 reference count operations.

6.3 Performance

We ran our domain decomposition code on two computer systems: an eight processor Cray C-90 vector supercomputer, and a four processor, cache-coherent Silicon Graphics model 340 Iris multiprocessor. The C-90 runs slowly if the code includes many short vectors, or if memory is accessed in strides of powers of two. The SGI runs well if program segments and data fit into its 64KB caches; if not, performance degrades sharply. We experimented with both square and rectangular grids.

We performed several runs on the SGI Iris 340, observing run times, operation rates, and parallel speedups. We used square grids of size 10, 35, and 60. The initial number of zones per panel was 6 by 6, and the refinement factor was 2. For each run we set the left boundary to 10.0 and the other three boundaries to 0.0. Table 1 shows our results. We have seen as much as seven MFLOPS on computationally intensive mathematical kernels, but most large applications run at one to two MFLOPS. Speedups of three on four processors are typical.

Tables 3a and 3b show results for rectangular shaped grids on the SGI system. There are definite shape effects—tall grids show better raw performance and speedup. The reason is that heat diffuses from left-to-right, concentrating the computational work in columns.

The overhead in regions in front of the heat gradient is significant, since the conjugate gradient solver immediately returns a zero solution in these regions. Larger numbers of rows results in more refined regions, amortizing overhead and resulting in better speedups. When we change the boundary conditions so that heat diffuses from top-to-bottom, the wide grids perform better than the tall grids.

We ran several square grid problems on the Cray C-90 systems of size 30 and 100. The initial number of zones per panel 15x15, 25x25, and 75x75. Table 2 shows the results. The code's performance improves as the number of zones per panel increases. In the first and second problems we lose less than one processor out of six and five, respectively (n.b. – we were unable to obtain a time for the second problem on six processors because of a system failure). The speedups on the third problem were slightly worse. Tables 4a and 4b show the performance of rectangular shaped problems on the C-90. The Cray appears less affected by the shape of the grid than the SGI machine.

7.0 Conclusions

We consider this work the first step in developing a robust adaptive mesh refinement code. To date, we are pleased with the results. We have been able to develop a high-performance, transportable domain decomposition code in Sisal. The development time was minimum. The compiler eliminated all unnecessary copy operations, inserted code to manage memory, and generated parallel and vector code as appropriate for the target machines. The runtime system efficiently managed concurrent tasks, and achieved good load balance. On both the C-90 and the SGI machine, the code's MFLOPS and speedups were generally excellent.

Our next step is to add a time component to the simulation. This addition significantly complicates the code. Grids of different resolution advance at different rates. Since boundary values must be exchanged before each time step, communications are no longer regular. In the current code, we partition the grid into a fixed number of panels. In the final code, we expect the number of panels to vary and their shapes to be irregular. Then, how to block panels of the same resolution into larger panels of regular shapes becomes an issue. Finally, we plan to port the code to massively-parallel, physically-distributed memory machines. Prototype Sisal systems exist for such machines. Our code will be an excellent test for these systems as they mature.

Acknowledgements

We wish to thank Cray Research Incorporated for use of their C-90. They provided us dedicated time to run our experiments. Special thanks to Deborah Barnes for arranging the account. This work was supported by

Lawrence Livermore National Laboratory under DOE contract W-7405-Eng-48 and the DOE SERS Program.

References

1. Berger, M. and J. Oliver. Adaptive Mesh Refinement for Hyperbolic PDEs. *Journal of Computer Physics*, vol. 53, 1984.
2. Cann, D. C. Compilation Techniques for High Performance Applicative Computation. Ph.D. thesis, Department of Computer Science, Colorado State University, 1989.
3. Cann, D. C. *The Optimizing Sisal Compiler: Version 12.0*. Lawrence Livermore National Laboratory Manual UCRL-MA-110080, Lawrence Livermore National Laboratory, Livermore, CA, April 1992.
4. Cann, D. C. Retire FORTRAN? A Debate Rekindled. *CACM*, vol. 35, 8 (August 1992).
5. Feo, J.T., D. C. Cann, and R. R. Oldehoeft. A Report on the SISAL Language Project. *Journal of Parallel and Distributed Computing*, vol. 12, 10 (December 1990), pp. 349-366.
6. Hendrickson, C. P. Programming a real code in a functional language. *Proc. Visions of Supercomputing Conference*, Santa Fe, NM, September, 1991.
7. McGraw, J. R. et. al. *Sisal: Streams and iterations in a single-assignment language, Language Reference Manual, Version 1.2*. Lawrence Livermore National Laboratory Manual M-146 (Rev. 1), Lawrence Livermore National Laboratory, Livermore, CA, March 1985.
8. Miller, P. J. and W. Cedeño. *TWINE Version 3.0*. Lawrence Livermore National Laboratory, Livermore, CA, July 1992.
9. Ranelletti, J. E. Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages. Ph.D. thesis, Department of Computer Science, University of California at Davis/Livermore, 1987.
10. Warren, K. H. and E. D. Brooks, Gauss Elimination: A Case Study. *Proc. COMPCON 91*, San Francisco, CA, February, 1991.

10 x 10 x 6 x 6 Problem: 125 627 801 Flop

# workers (n)	runtime (t _n) sec.	MFlops sec.	speedup (t ₁ / t _n)
1	31.16	4.03	1.00
2	19.93	6.30	1.56
3	16.48	7.62	1.89
4	13.80	9.10	2.26
5			
6			

35 x 35 x 6 x 6 Problem: 739 030 601 Flop

# workers (n)	runtime (t _n) sec.	MFlops sec.	speedup (t ₁ / t _n)
1	261.97	2.82	1.00
2	150.26	4.92	1.74
3	110.94	6.66	2.36
4	90.90	8.13	2.88
5			
6			

60 x 60 x 6 x 6 problem: 1 331 609 801 Flop

# workers (n)	runtime (t _n) sec.	MFlops sec.	speedup (t ₁ / t _n)
1	540.62	2.46	1.00
2	305.07	4.36	1.77
3	208.84	6.38	2.59
4	167.59	7.95	3.23
5			
6			

Table 1 - Square grids on the SGI

100 x 100 x 15 x 15 problem: 4 119 798 160 Flop

# workers (n)	runtime (t _n) sec.	MFlops sec.	speedup (t ₁ / t _n)
1	88.2144	46.7	1.00
2	41.8448	98.5	2.11
3	30.1205	136.8	2.93
4	22.5384	182.8	3.91
5	18.8234	218.9	4.69
6	15.6003	264.1	5.65

30 x 30 x 25 x 25 Problem: 4 447 810 556 Flop

# workers (n)	runtime (t _n) sec.	MFlops sec.	speedup (t ₁ / t _n)
1	17.0813	260.4	1.00
2	8.9815	495.2	1.90
3	5.9705	745.0	2.86
4	4.5517	977.2	3.75
5	3.7116	1198.4	4.60
6			

30 x 30 x 75 x 75 Problem: 64 334 663 442 Flop

# workers (n)	runtime (t _n) sec.	MFlops sec.	speedup (t ₁ / t _n)
1	114.9788	559.5	1.00
2	65.7826	978.0	1.75
3	43.3581	1483.8	2.65
4	36.5996	1757.8	3.14
5	31.6337	2033.8	3.63
6	30.1293	2135.3	3.82

Table 2 - Square grids on the Cray C90

8 x 4 x 6 x 6 Problem; 20 624 956 Flop

# workers (n)	runtime (t_n) sec.	MFlops sec.	speedup (t_1 / t_n)
1	4.29	4.81	1.00
2	2.90	7.11	1.53
3	2.11	9.77	2.03
4	1.39	14.84	3.08

4 x 8 x 6 x 6 Problem; 28 837 637 Flop

# workers (n)	runtime (t_n) sec.	MFlops sec.	speedup (t_1 / t_n)
1	6.30	4.58	1.00
2	3.88	7.43	1.50
3	3.83	7.53	1.64
4	3.29	8.77	1.91

Table 3a - Small rectangular grids on the SGI

40 x 4 x 6 x 6 Problem; 118 328 956 Flop

# workers (n)	runtime (t_n) sec.	MFlops sec.	speedup (t_1 / t_n)
1	24.37	4.86	1.00
2	13.71	8.63	1.78
3	9.55	12.39	2.55
4	6.88	17.20	3.54

4 x 40 x 6 x 6 Problem; 50 428 156 Flop

# workers (n)	runtime (t_n) sec.	MFlops sec.	speedup (t_1 / t_n)
1	16.37	3.08	1.00
2	10.04	5.02	1.63
3	8.61	5.86	1.90
4	6.99	7.21	2.34

Table 3b - Large rectangular grids on the SGI

240 x 32 x 16 x 16 Problem; 9 768 679 372 Flop

# workers (n)	runtime (t_n) sec.	MFlops sec.	speedup (t_1 / t_n)
1	99.7700	97.9	1.00
2	51.1051	191.1	1.95
3	32.5352	300.2	3.07
4	24.4978	398.8	4.07
5	19.8248	492.8	5.03

10 x 90 x 50 x 50 Problem; 8 495 654 716 Flop

# workers (n)	runtime (t_n) sec.	MFlops sec.	speedup (t_1 / t_n)
1	21.8905	388.1	1.00
2	11.4273	743.5	1.92
3	8.1468	1042.8	2.69
4	6.5096	1305.1	3.36
5			

30 x 225 x 16 x 16 Problem; 2 776 267 738 Flop

# workers (n)	runtime (t_n) sec.	MFlops sec.	speedup (t_1 / t_n)
1	103.2219	26.9	1.00
2	47.7555	58.1	2.16
3	33.9718	81.7	3.04
4	26.0364	106.6	3.96
5	20.9723	132.4	4.92

90 x 10 x 50 x 50 problem; 69 110 431 036 Flop

# workers (n)	runtime (t_n) sec.	MFlops sec.	speedup (t_1 / t_n)
1	128.6606	537.2	1.00
2	66.8517	1033.8	1.92
3	47.3070	1460.9	2.72
4	36.0207	1918.6	3.57
5	29.0528	2378.8	4.43

Table 4a - Rectangular short vectors on the C90

Table 4b - Rectangular long vectors on the C90

END

**DATE
FILMED**

12/28/93