Department of

Computer Science

DISTRIBUTED RUNTIME SUPPORT FOR TASK AND DATA MANAGEMENT

Matthew Dennis Haines

Technical Report CS-93-110

August 5, 1993

Colorado State University

PH.D. DISSERTATION

DISTRIBUTED RUNTIME SUPPORT FOR TASK AND DATA MANAGEMENT

Submitted by
Matthew Dennis Haines
Department of Computer Science

In partial fulfillment of the requirements
for the degree of Doctor of Philosophy
Colorado State University
Fort Collins, Colorado
Summer 1993

ABSTRACT OF PH.D. DISSERTATION

DISTRIBUTED RUNTIME SUPPORT FOR TASK AND DATA MANAGEMENT

High-performance computer architectures are evolving into larger and faster systems and, in particular, distributed memory multiprocessors represent the most powerful class of computers built today. Their available resources provide a programmer with the potential for exploiting massive amounts of parallelism in an application, and yet support for high-level programming languages on these machines is sparse. Thus the need is great for software systems that can free the programmer from the implementation details of an architecture.

This dissertation focuses on the design of software support for a high-level functional language on conventional distributed memory multiprocessors. Specifically, we present the design and implementation of a runtime system that provides implicit support for both thread management and data management, and study the effects of latency avoidance and latency tolerance on a set of sample programs that have been written using the Sisal functional language.

Matthew Dennis Haines
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Summer 1993

Advisers:
Professor Rodney R. Oldehoeft
Professor Anton Pedro Willem Böhm

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

'Where shall I begin, please your Majesty?', he asked. 'Begin at the beginning', the King said, gravely, 'and go on till the end: then stop.'

– Lewis Carroll, "Alice in Wonderland"

Modern sequential computers are approaching the fundamental physical limitation that signals cannot travel faster than the speed of light. However, current scientific problems [Gra91] require the computational power of thousands of these sequential computers, and the demand for computational power is ever-increasing. This situation has led computer architects to design multiprocessor computer systems in the hopes of using the collective computational power of the multiple processors to solve these large problems.

Shared memory multiprocessors provide each of the processors equal access to a shared memory, but as the speed of the individual processors increases, the number of processors that can be supported in such a system decreases. Due to this inability of current shared memory multiprocessors to scale past a relatively small number of processors, today's most powerful computers are *distributed memory multiprocessors* [Thi91, Int91, nCU90b], capable of supporting thousands of powerful processors, whose aggregate computing capabilities are sufficient for solving many of the large problems that face our scientific community today.

Unfortunately, these advances in hardware design have not been followed by corresponding advances in software. High-level programming abstractions for these machines are almost non-existent, leaving most programmers the task of explicitly programming these architectures using machine-dependent, low-level abstractions. This approach is error-prone and encumbers the programmer with many details outside of the application domain, such as explicit data distribution and synchronization. In addition to identifying, allocating, and controlling parallel tasks, the programmer must control the distribution of data among the separate memories so as to minimize remote memory latencies. Also, if the machine does not provide hardware support for a single addressing space, the programmer must also insert the appropriate message passing calls needed to exchange data from one remote memory to another.

This dissertation studies the effectiveness of an alternative approach, where programs written in an implicitly parallel, machine-independent programming language are executed efficiently on distributed memory multiprocessors. The compiler creates the parallel tasks, and the runtime system controls the distribution of the tasks for efficient parallel execution. The runtime system also supports a single addressing space for user data structures that allows for flexible distribution schemes and the ability to link the distribution of data with the distribution of the parallel tasks that will access the data so that remote memory references are minimized.

As distributed memory multiprocessors continue to grow in size and computational power, two issues are becoming increasingly important:

- *Detecting enough parallelism* in an application to keep the machine resources busy. As the number of processors in these systems increases into the thousands, the amount of parallelism needed to obtain thousand-fold speedups increases proportionally. Applicative (or functional) languages have demonstrated their ability to expose inherent parallelism in an application as well as to simplify the task of parallel programming [Nik90, FCO90].

- *Managing the parallelism* for efficient execution on a large variety of processor configurations, from a few to thousands. Once the parallelism in an application has been revealed, it is up to the compiler, runtime system, and hardware to manage the parallelism for efficient utilization of the machine resources. Since most large distributed memory multiprocessors have several orders of magnitude difference in the time to access local memory versus the time to access remote memory, managing this latency is necessary to efficient execution. Latency can be *avoided* by keeping many memory accesses local, and *tolerated* by switching to other useful work that can be done in the time it takes to satisfy the remote reference (i.e. *multithreading*). Latency avoidance requires that a code segment exhibit some degree of locality, and that it be mapped to the same node as the data it references. Latency tolerance requires a fast context switching mechanism and much more program parallelism than machine parallelism.

Sisal (Streams and Iterations in a Single Assignment Language) is a functional language that supports data types and operations for scientific computation [MSA$^+$85]. Current implementations of Sisal exist for sequential machines and for shared memory multiprocessor architectures [CO88], including vector [Can92, LSF88], hierarchical memory [WFC91], and dataflow [BS89], where the hardware supports the shared memory abstraction. The Sisal compiler consists of three parts:

1. The *frontend* is responsible for ensuring the syntactic correctness of a Sisal program and translating the Sisal source program into an optimized intermediate dependence graph form called IF1 [SG85].

2. The *backend* is responsible for adding memory requirements to the IF1 graph, performing *build-in-place* and *update-in-place* optimizations to eliminate aggregate copies [Can89], applying a standard set of compiler optimizations, and finally generating C as the target code.

3. The target C code compiled under the native C compiler and linked with the *runtime system* to produce machine-specific object code. The runtime system is responsible for providing the Sisal compiler with two main abstractions: *task management* and *memory management*. Tasks are portions of sequential code that can be executed independently.

The current compiler assumes that all data structures exist in a flat, shared address space and that the runtime routines employ shared queues and locks. While this has provided efficient implementations of Sisal on shared memory multiprocessor architectures

[Can92], it has precluded a straightforward port of the Sisal system to distributed memory multiprocessors.

This dissertation focuses on the design and implementation of runtime support for a shared memory programming paradigm on distributed memory multiprocessors with little or no support from the compiler or hardware. We use this system as a basis for studying the abilities and limitations of a runtime system in providing efficient distributed memory execution. In particular, we describe the design and implementation of a runtime system for the execution of Sisal programs on the nCUBE/2 distributed memory multiprocessor, where the Sisal compiler is unaware of the underlying architecture.

The remainder of this dissertation is organized as follows. Chapter 2 provides the motivation and preliminary background necessary for establishing the research goals. Chapter 3 examines related research and its relationship to our work. Chapter 4 introduces the design of our distributed memory task management system, including multi-level distribution and multithreading. Chapter 5 describes the design of our distributed memory data management system, also known as *VISA*. Chapter 6 introduces the sample programs that we use to evaluate our runtime system. The programs are written in Sisal and cover a wide range of common scientific computations. Chapter 7 presents the results and analysis of several experiments using our sample programs. Chapter 8 concludes and provides ideas for extensions of this research.

Chapter 2

BACKGROUND

Computer Science is no more about computers than astronomy is about telescopes.
– E. W. Dijkstra

This chapter provides the motivation and background for our research. We start with an overview of distributed memory multiprocessors, their architecture and features. Next we examine the issue of detecting parallelism in a program from the language perspective. Finally, we look at the management of parallelism and the effect it can have on the two fundamental issues in multiprocessing.

2.1 Multiprocessor Systems

Distributed memory multiprocessors represent today's most powerful class of computers [Thi91, Int91, nCU90b], and the best hope of achieving the scalable parallelism necessary for solving the *Grand Challenge* problems that face our scientific community [Gra91]. To understand why distributed memory multiprocessors have come to the forefront of high performance computing, we examine the issues that govern today's multiprocessor architectures and then compare both shared memory multiprocessors and distributed memory multiprocessors using these issues.

1. *Single Addressing Space.* A multiprocessor system that supports a single addressing space identifies every memory location in all memory elements with a unique and uniform address from a single contiguous space. The advantages of having a single addressing space include the ability to reference all memory locations using a single addressing scheme, which greatly simplifies the programming of the machine, and the ability to pass procedural parameters using their address rather than their value. For scientific applications that employ many large data structures, the overhead of copying parameters required for passing by value can be overwhelming.

2. *Shared Memory.* If a multiprocessor system supports a single addressing space, then it may additionally support shared memory, which we define as the ability to place a data object anywhere in the memory system without affecting the performance of the corresponding application. This implies that there is a uniform access time to all memory locations in the system. The advantage of having shared memory is that *data distribution*, which is the problem of distributing the program data structures among the participating memory elements for optimal program execution, is trivial, since, by definition, data placement does not affect performance.

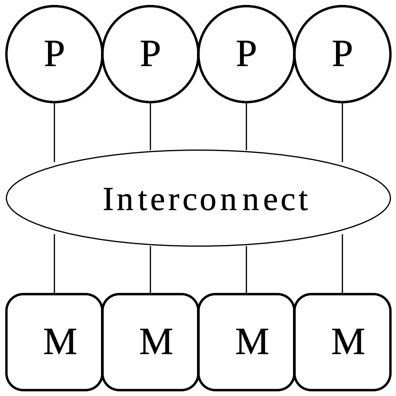

Figure 2.1: Organization of a shared memory multiprocessor

3. *Scalability.* We define a multiprocessor system to be scalable if the architecture can incrementally support a large number[1] of nodes (processing and memory elements), and the addition of more nodes increases the system's performance. Typically, scalability depends on the interconnection scheme used to combine the processing and memory elements. For example, a fully-connected network, in which every processing element is connected to every other processing element, is not scalable (not to mention buildable), since the complexity (hence cost) of the network increases as the square of the number of processing elements. The advantage of a scalable system is that a very large number of processors can be provided and, for many of the problems that face our scientific community, this is essential.

When we examine a multiprocessor system from an organizational view, we see that it contains a set of processing elements, a set of memory elements, and hardware for interconnecting these sets called the *interconnection network*. We restrict our attention to Multiple Instruction, Multiple Data (MIMD) multiprocessors, where each processing element possesses a complete instruction set and is capable of executing code independently of all other processors. Though there are many ways of combining these elements to form a MIMD multiprocessor, two common organizations have emerged: *shared memory* and *distributed memory.*

- **The Shared Memory Multiprocessor Organization**
  A shared memory multiprocessor is one which supports shared memory at the hardware level. This is typically accomplished by connecting the processing elements with the memory elements using a high-speed bus that provides a single addressing space and uniform access time to all memory locations (see Figure 2.1). The advantage of the shared memory multiprocessor is its hardware implementation of shared memory, which frees the programmer and all system software from the difficult task of data distribution and remote data access through messages. The disadvantage of this design is that it does not scale past a relatively small number of processors (on the order of 50), and the number is decreasing as the processor speeds increase.

[1]By today's standards, "a large number of processors" typically means over a hundred.

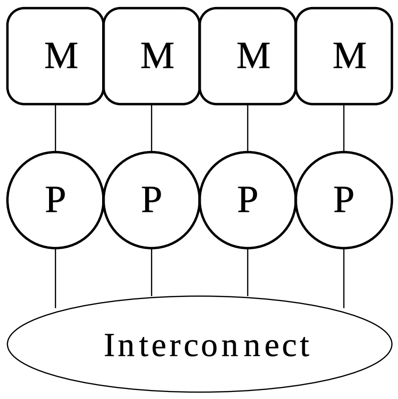

Figure 2.2: Organization of a distributed memory multiprocessor

Another restriction to the scalability of shared memory multiprocessors is the problem of maintaining the coherence of multiple caches [DB82, CF78]. The hardware required to keep the caches coherent adds to the cost of increasing the number of processors, thus further limiting the scalability of the system. Examples of shared memory multiprocessors include the Sequent Symmetry, the Alliant FX, and the Encore Multimax.

- **The Distributed Memory Multiprocessor Organization**
  In a distributed memory multiprocessor, each processing element is combined with a memory element to form a *node*, and the nodes are then connected using a scalable interconnection network, such as a ring, a mesh, or a hypercube [AG89a] (see Figure 2.2). Each node supports a separate addressing space that encompasses the local memory. The advantage of the distributed memory multiprocessor is that it is scalable to a very large number of processors[2]. Distributed memory multiprocessors that offer only a message passing abstraction for sharing data do not have the problem of keeping caches coherent, since the local processor caches contain only local objects. However, as detailed in Chapter 3, there are various hardware and software designs that offer other methods for sharing data on these machines, and may re-introduce the problem of cache coherence. The disadvantage of a distributed memory multiprocessor is the absence of a single addressing space, and thus shared memory. This requires that the programmer (or system software) be responsible for data distribution and synchronization of computations to achieve the required remote data access (i.e. message passing), both of which are very complex and error-prone tasks. Examples of distributed memory multiprocessors include the intel iPSC/860 [Int91] and the nCUBE/2 [HMS$^+$86].

Thus we come to the crux of the multiprocessor design issue: though distributed memory multiprocessors are capable of supporting large numbers of processors, they are more difficult to program due to the problems of data distribution and multiple addressing spaces [PB90]. We will demonstrate this point in Chapter 7. However, the issue of scalability has overpowered the issue of programability since there are many problems that

[2]Current distributed memory multiprocessors contain thousands of processors.

require a very large number of processors to complete in a reasonable amount of time. This had led to several hardware and software approaches that attempt to empower the basic distributed memory design with some of the virtues of shared memory, in effect blurring the once-clear distinction between shared memory and distributed memory multiprocessors. For example, distributed memory multiprocessors are being built with hardware support for a single addressing space, though the time required to access each of the memory addresses is not uniform [Ken92, Fra87]. These NUMA (Non-Uniform Memory Access) machines remove the burden of explicit message passing from the programmer, but the problem of data distribution still remains.

One of the latest hardware approaches is the *Tera Computer System* [AAC+91], which is a descendant of the Denelcor HEP machine [Smi85] and the Horizon supercomputer project [KS88]. From a hardware standpoint, the Tera system is a distributed memory multiprocessor whose basic organization differs from most multiprocessors in that the memory elements and processing elements are not tightly coupled as nodes. Rather, the processing elements and memory elements are separate entities that are combined using a scalable interconnection network. However, from an applications standpoint, the Tera claims to be a shared memory multiprocessor in that it provides a single addressing space and can hide the effects of distributed memory latency by overlapping the memory operations with the work of another task that would eventually have to be done. By hiding the memory latency, Tera asserts that it can achieve shared memory status; the placement of data would not affect the performance of an application since all memory latencies are hidden. Tera employs specialized hardware and multiple levels of parallelism in hopes of successfully masking all long latency operations. Thus we see how Tera and NUMA systems blur the once clean line between shared and distributed memory. The success of the Tera system and its claim of achieving shared memory status is still to be determined. A more detailed examination of Tera can be found in Chapter 3.

2.2 Latency and Synchronization: The Two Fundamental Issues in Multiprocessing

Every multiprocessor architecture, whether shared memory, distributed memory, or a hybrid of these basic designs, must address the two fundamental issues in multiprocessing: *latency* and *synchronization* [AI87].

- **Latency** is defined as the time that elapses between making a request and receiving the associated reply. Though latency is present in I/O systems and and in function calls, latency is generally regarded in terms of *memory latency.* Uniprocessors and shared memory multiprocessors can expect constant time, and therefore constant latency, in accessing memory locations. However, since distributed memory multiprocessors employ interconnection networks that do not provide constant access time, they face a much larger and varying latency. When this latency cannot be avoided by keeping the required data local, or hidden by overlapping it with useful operations, a substantial performance penalty is incurred. Current research projects [AAC+91, CSS+91, Bec92] are focusing on the possibility of trading parallelism for latency. This is done by overlapping parallel execution of program segments with communication primitives so that the communication latency is effectively hidden. In order to hide the latency with overlapping program segments, the program must be partitioned into independent *tasks* that can be executed in parallel. Then, when a memory reference is issued from a task, the system can *switch* from that task to another task ready for execution. This results in a number of currently executing tasks which then need to be *synchronized* so that the program remains determinate.

- **Synchronization** is defined as the temporal ordering of tasks necessary to obtain *determinate* system. Task synchronization is needed for avoiding the read-before-write race of producer-consumer parallelism, for providing a mechanism in which to join a set of tasks at a particular location in the program (i.e. *barrier*), and for providing mutually exclusive access to shared resources. Synchronization may result in the suspension of a task until a particular event occurs, though this does not imply that the processor need be idle; it may be possible for the processor to switch to another task that is not blocked and execute statements from this latter task. Thus the cost of synchronization is the time required to execute the synchronization primitive *plus* either the idle time of a blocked processor, or, if the processor switches processes, the time of two context switches: one to get to a new task and another to return.

The two fundamental issues are very closely related: Multiprocessor systems incur large memory latencies, giving rise to the need for hiding this latency, which implies task switching and therefore synchronization to control the tasks. Processor architectures that have been designed to address the issues of latency and synchronization include multi-threaded [Smi85, AAC$^+$91] and dataflow [AC86, Den80] architectures. However, these architectures typically require unconventional hardware, such as non-von Neumann processors and/or memories tagged with presence bits. This results in high costs for production of special-purpose hardware. Additionally, these systems currently experience performance problems that limit their practical use in the marketplace. Distributed memory multiprocessors, on the other hand, are commercially available today and provide the scalability needed to build powerful computer systems. However, if we are to exploit this growing class of computer systems, we must develop software that can address the issues of latency and synchronization, as well as the problems of multiple addressing spaces and a lack of true shared memory.

This gives rise to the problems of detecting the parallelism that exists in a given algorithm (i.e. the creation of independent tasks), and managing this parallelism with the goal of efficiently executing the program and, if there is enough parallelism to saturate the system, efficiently utilizing it. The extent to which the programmer is responsible for these tasks depends on the amount of support provided by the language, the compiler, the runtime system, the operating system, and the machine architecture. For purposes of portability, correctness, and ease of programming, it is desirable to alleviate the programmer from the implementation-specific details of a machine architecture. Thus we face the challenge of providing the programmer with a high level language capable of abstracting the underlying architecture, implicitly detecting the parallelism in an application, and managing the parallelism for efficient execution on a wide range of multiprocessor systems. Clearly this is a challenging goal.

2.3 On Detecting Parallelism

The effort required to detect the parallelism that exists in a program is strongly influenced by the programming methodology that is used to encode the program.

2.3.1 Imperative Programming Languages

If an *imperative* programming language (such as Fortran or C) is used, then the task of detecting which statements may execute in parallel becomes a difficult one. In fact, one survey [BE92] found that half of the programs in the *Perfect Benchmarks* suite [BCK$^+$89] were not parallelized due to problems with the structure of the potentially parallel loops, and the remaining programs achieved speedups averaging 2.5 out of a possible 8 processors. This poor performance is due to the nature of imperative language semantics: mutable memory locations, which represent the state of the computation, are modified by sequentially-ordered program statements, resulting in a program with no inherent provision for parallelism. This leads to the need for determining which of the sequential statements can be executed in parallel. For two statements to be executed in parallel, they must be *independent* of each other, which means that the execution of one statement does not effect the execution of the other. This "independence" information is gathered using *dependence analysis*. If a programmer wishes to use an imperative language for parallel execution, then either the programmer can perform the dependence analysis and provide parallelizing statements where applicable (*explicit imperative programming*), or a parallelizing compiler can take the programmer's sequential code and attempt to discover the dependence relations needed to provide automatic parallelization (*implicit parallel programming*). Since every imperative program that is to be parallelized must undergo some degree of dependence analysis, we shall introduce those concepts before examining the explicit and implicit imperative programming approaches.

Dependence Analysis

Padua and Wolfe identify four types of data dependence [PW86]: *flow dependence*, *anti-dependence*, *output dependence*, and *loop iteration dependence*. Zima et al. [ZBG86] refer to the first three as *data flow* analysis, and the last as *data dependence* analysis.

- Flow dependence (or *true dependence*) occurs when one statement depends on a value that is computed by a preceding statement. For example, in the code
  S_1: A := B + C
  S_2: D := A + 2
  statements S_1 and S_2 cannot be executed in parallel since S_2 uses the value of A computed in S_1.

- Anti-dependence occurs when a statement assigns to a variable that is used in a preceding statement. For example, in the code
  S_1: A := B + C
  S_2: B := D / 2
  statements S_1 and S_2 cannot be executed in parallel since S_1 uses the value of B computed before S_2, which re-assigns a new value to B.

- Output dependence occurs when two statements assign different values to the same variable. For example, in the code
  S_1: A := B + C
  ⋮
  S_2: A := D + E
  statements S_1 and S_2 cannot be executed in parallel since, if S_1 were to execute after S_2, A would contain the wrong value after this code segment.

- Loop iteration dependence (sometimes called loop dependence) occurs when one iteration of a loop has a dependence with another iteration of the loop. For example, in the code
  for I **in** 1 **to** 3 **do**
  S_1: A(I) = B(I) + C(I)
  S_2: D(I) = A(I-1)
  a dependence in the elements of A flows from iteration $i-1$ to iteration i. **end for**

It is important to note that anti-dependence and loop dependence are, in a sense, *false dependencies*. They arise not because data is being passed from one statement to another, but because the same memory location is used more than once.

Explicit Imperative Programming

In case of explicit imperative programming languages, the user is not only responsible for all of the dependence analysis, but also for inserting the parallelizing and synchronizing statements in the proper place so as to produce a correct and determinate program. Additionally, if the target machine is a distributed memory multiprocessor, the programmer must be concerned with the data distribution and data movement statements. The complexity of the dependence analysis is also exacerbated by the modular style of programming that is often employed, in which programs are separated into logical subprogram units. Subprograms encapsulate the details of an operation, limiting the scope and effectiveness of intra-procedural analysis. Additionally, for imperative languages that allow *aliasing* (such as Fortran and C), dependence analysis must take a very conservative approach, often severely limiting its effectiveness. The result is a program in which the basic meaning of an algorithm is lost among the various explicit statements needed to specify and manage the parallelism, and the applications programmer is forced to master the concepts of data dependence, data distribution, and message passing, in addition to the algorithm being encoded. In fact, one can draw the analogy that explicit imperative programming is the assembly language of parallel processing: cryptic and error-prone yet flexible and powerful. Still, due to the lack of sophisticated software and the flexibility of the explicit approach, many parallel applications have been written to exploit the parallelism of a large machine using explicit parallelization and synchronization constructs.

Implicit Imperative Programming

In the case of implicit imperative programming languages, it is the job of a very intelligent (and complex) compiler to determine the correct dependence relationships required for parallel execution, and then to insert the appropriate parallelization and synchronization primitives required to successfully execute the program in parallel. A *vectorizing* compiler is similar to a *parallelizing* compiler, but the former restricts its attention to restructuring loops for vector execution on a *single* processor, while the latter concentrates on a more complete analysis needed for determining parallel execution of tasks on *multiple* processors. In either case, dynamic variables and aliasing, as well as the modular style of programming, often obscure the dependence relations and force the compiler to make very conservative decisions, which typically result in an under-parallelization (or under-vectorization) of the program. An imperative language compiler might attempt to remove false dependencies (such as output dependence and anti-dependence) using various techniques, but, in general, these are difficult problems and typically done only in a limited number of cases. Some compilers attempt to make up for their limited powers by asking the programmer to *annotate* the code, such as inserting a parallel "for all" statement in the place of a sequential "for" statement. However, annotations simply place the burden of detecting the parallelism back on the programmer (explicit imperative programming), and if a mistake is made, the resulting program may be a *nondeterminate* nightmare. Despite their difficulties, there have been numerous attempts at automatically parallelizing imperative languages [FHK$^+$90, GB90, HA90, IFKF90, ZBG86, Ree90, RA90]. To the extent that these compilers have been able to exploit a limited amount of parallelism in certain types of numerical applications, they have been successful.

The difficulty in getting imperative programming languages to execute efficiently on parallel architectures without sacrificing parallelism or encumbering the user has led to the investigation into other approaches, most notably *object oriented languages* [LG91, CGH89, WY88] and *functional programming languages* [AGP78, MS82, MSA$^+$85].

2.3.2 Functional Programming Languages

Functional programming languages possess several properties that are useful for parallel programming:

- Their operational semantics do not over-specify the order of evaluation, which helps in identifying all forms of parallelism.

- The computation can be represented by a data flow graph that exposes all data dependencies in a program. Data dependence analysis need only focus on loop iteration dependencies.

- Programs are composed of mathematically sound (side-effect free) functions. This guarantees that the order in which functions are evaluated has no effect on the result of the computation (*Church-Rosser property*). Therefore, the textual sequencing given by the programmer does not necessarily define the program's sequencing constraints, and thus only true flow dependence governs the execution order of a program.

- To preserve the history insensitivity of the applicative programming model [Bac78], functional languages support only a single assignment[3] of a value or a definition to a variable. This means that variables are simply a notational convenience for representing mathematical expressions, and thus functional languages are said to be free of "state." A functional program is then a series of mathematically sound functions that, when given the appropriate input data, are *reduced* to a single expression. Variables in the program represent various stages of the reduction. Thus A := X + Y binds the value of X + Y to A throughout the scope of A.

These properties result in a language style that helps to identify the available parallelism in a program, rather than helping to obscure it. However, this attractive property of

[3]Actually, pure functional languages do not support assignment, but we will relax our definition of functional languages to include single-assignment languages, such as Sisal and Id with I-Structures. Since both functional and single-assignment languages operate under the applicative model of computation, they are often labeled *applicative languages*.

implicit parallelism does not come for free, as we will see in the next section. For example, functional languages are traditionally slow due to the problems in maintaining the single assignment semantics of the language, especially with respect to structured data. We will briefly examine two languages that are currently being used to take advantage of the new, high-performance parallel architectures: *Sisal* [MSA+85, BCFO91] and the family of *Id* languages (Id, Id Nouveau, Id90) [Nik87], to which we will cumulatively refer to as *Id*. But first we shall examine the issue of *strictness*, since this seems to be a major battle line for current functional languages [Got91].

The Issue of Strictness

The distinguishing feature of *strict semantics* is that arguments to procedures and data constructors are completely evaluated before the procedure body or data constructor is invoked. On the contrary, in *non-strict* functional languages, array elements may be read before the entire array is initialized (*non-strict data structures*), and a function may be evaluated and possibly even use a returned value before all of its input parameters have been evaluated (*non-strict control structures*) [Tra91]. This gives a non-strict functional language more expressive power than its strict counterpart, since there are some dependencies that are impossible to construct in strict functional languages. For example, consider the following functional code segment that generates a circular list:

```
{a = cons 1 (cons 2 (cons 3 a)); in a}
```

In order for this program to work correctly, the application of `a` in the definition of `a` must remain unevaluated until the end, at which time it establishes the circular link to the head of the list. In a strict functional language, this would either deadlock (waiting for a definition of `a`) or report an error that `a` was referenced before it was defined. In another example, we see how non-strictness can be used to control a possibly infinite amount of computation:

```
{ints_from n = cons n (ints_from (n+1)); in nth 10 (ints_from 1)}
```

In this program, the function `ints_from` creates a list of integers from `n` to infinity, however, the body of the program only needs a list from 1 to 10, since the `nth` function simply returns the n^{th} element of a list. Again, for a non-strict language, the evaluation of `ints_from n` can be delayed until it is known that only 10 elements are needed, while in the strict case this computation results in infinite recursion. For a more formal treatment of non-strictness, the reader is encouraged to examine [Tra91].

The expressiveness of a non-strict functional language is not without its drawbacks. Since non-strict languages can only generate a partial ordering of subexpressions at compile time, the compiler is forced to create a set of *threads*, which must then be ordered at runtime using *presence bits* to start a thread when the value it computes is needed by another thread. The result is that not only are presence bits required for synchronization, but that the resulting thread length is small. On the other hand, a strict language compiler is capable of generating a total ordering (i.e. a sequential program) at compile time. This not only allows for greater control over the thread size needed for efficient parallel execution, but also alleviates the need for presence bits. The result is that strict languages are better suited for execution on conventional von Neumann-based architectures.

Another issue relating to non-strictness is its effect on resource management. We can view a parallel computation as a tree of tasks or activations. A fully-eager, data-driven approach to evaluating the activation tree results in a breadth-first unfolding of the tree, so that at some point the entire tree must be resident in the system, probably swamping the system's resources. Therefore, the job of a *resource manager* is to unfold the activation tree in such a way as to effectively utilize the available resources, but not swamp them. The resource manager receives requests for new activations and must decide which activations to start, and when to start them. This is a difficult decision, based on the current load of the system and, in the case of non-strict languages, the interactions among the threads. This interaction can cause the system to deadlock, since withholding resources from one part of the activation tree may starve the activations in another part of the tree. For this reason, most non-strict functional languages restrict the attention of the resource manager to a certain type of simple loop bodies that can be executed in parallel [Cul90]. In a strict functional language, the lack of activation interaction means that the job of a resource manager is greatly simplified [BT88b].

The way in which functional languages deal with this issue influences many of their implementation characteristics, as we will see in a brief examination of Sisal (a strict functional language) and Id (a non-strict functional language).

Sisal

Sisal (Streams and Iterations in a Single Assignment Language) is an applicative language that supports data types and operations for scientific computation [MSA+85]. The Sisal language was designed with the following goals:

- *Goal:* A general-purpose functional language. *Fact:* Sisal has proved to be an effective language for solving many scientific applications [Can92], though its current implementation has yet to take advantage of the functional parallelism that is predominant in some applications. Also, the usefulness of Sisal outside of the scientific application realm has yet to be established.

- *Goal:* A language whose programs are always *determinate*, which means that the output from a program is independent of the execution order of the program. *Fact:* Determinacy is achieved by the functional semantics of Sisal.

- *Goal:* Execute on both conventional and novel architectures. *Fact:* Sisal executes on both conventional [CO88, Can92, LSF88] and novel [BS89] multiprocessor architectures, as well as most Unix-based uniprocessors. There is also ongoing research in hierarchical-memory [WFC91] and distributed memory [Gri90, HB92] Sisal implementations. The functionality of the language allows for an intermediate representation as a data flow graph, which can then be executed on a dataflow architecture, such as the Manchester Dataflow Machine [BS89]. For conventional implementations, Sisal defines arrays using *strict* semantics. The result is that a sequential evaluation order of array references is always possible, alleviating the need for presence bits.

- *Goal:* Compete with the sequential and parallel execution performance of imperative languages. *Fact:* For some scientific applications, Sisal is very competitive with Fortran (and C) on many conventional multiprocessor and vector architectures [Can92]. This is made possible by the use of *strict* data constructors, which can be analyzed at compile time to possibly eliminate the need for copying arrays at runtime [Can89].

Thus Sisal is an applicative language that, because of its strict semantics, has been efficiently executed on conventional multiprocessor hardware, without the need for hardware synchronization (presence bits). However, Sisal sacrifices a certain amount of expressibility in achieving this goal. For some, this sacrifice is justified, but for those people that *require* the expressibility that a strict language cannot provide, Sisal's efficiency is irrelevant.

Id

Back when data-flow didn't work so well, it seemed a lot more elegant!

– Allan Gottlieb

The Id language [Nik87] exemplifies the tension that can arise between using a purely functional language and supporting efficient execution. At Id's core is a purely functional language that supports non-strict semantics, but surrounding the core is a more complete, non-functional language that supports imperative data structures, accumulators, and non-determinism [AG89b]. Adding these non-functional features to Id was done to improve the programmability and performance of Id for scientific and real-time applications. Implementations of Id exist for novel architectures [Den80, Tra86, PC90], and for conventional architectures using a software interface to handle the non-strictness of the language [CSS+91].

One area of inefficiency caused by the purely functional core of Id is data structure handling. The functional operators for creating and manipulating data structures are *cons* and *update*, respectively, which treat all data structures as lists. The *cons* constructor builds data structures, arrays for example, by creating incrementally-larger arrays until the array size is achieved. This results in a large number of unused intermediate arrays. The *update* constructor alters a data structure by creating a new copy of the array, with the updated value in place of the old value in the new data structure. An *I-Structure* [ANP89] is a single-assignment data storage mechanism that replaces the *cons* constructor for creating arrays, and represents Id's first departure from the purely functional world. This is because the I-Structure arrays can be allocated in one operation and filled using one or more separate operations, whereas a functional implementation allocates and fills the arrays in one operation. Rather than treating arrays as lists, the I-Structure mechanism allocates enough space (from the I-Structure storage) for the array at one time using the *allocate* operator. The array is initialized by writing directly into the elements of the array using the *I-store* operator. Note however that an array element can only be written to once, and if an *I-store* finds that a location is already occupied by a value, an error occurs. Another feature of I-Structures comes from reading the arrays using the *I-fetch* operator. Each element of I-structure storage contains presence bits that inform an *I-fetch* whether or not the value it requires has already been written. If not, the *I-fetch* operator will block until the write occurs. This *self-synchronizing* feature of I-Structures is needed to implement the non-strict semantics supported by the language. However, I-Structures do not solve all of the problems of a functional language. There still remains the problem of the need for copying data structures in the *update* operator, and the larger problem of *state*. Since functional languages are without *state*, it is impossible to efficiently implement *accumulators*, which are typically used for creating a *histogram*. Thus we see the addition of another non-functional feature to Id: accumulators. Id accumulators take the form of a *Fetch-and-Op* without the *Fetch*. That is, they atomically accumulate a value without

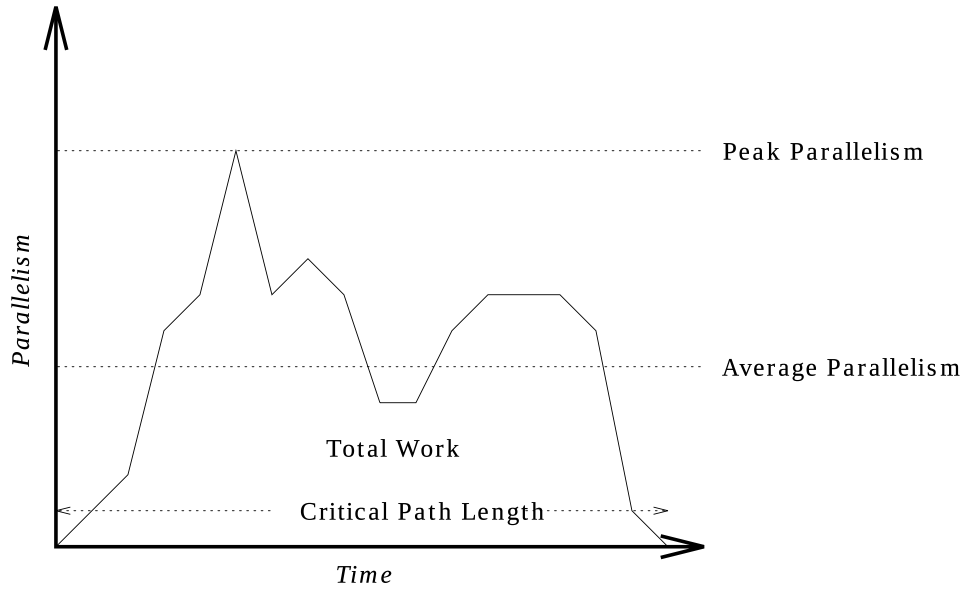

Figure 2.3: A sample parallelism profile

returning its result. With accumulators, a deterministic data-flow solution is possible that closely mimics the multiprocessor solution but avoids the space and time penalties of traditional single-assignment data structures. Determinacy is maintained by specifying, upon creation of the accumulator, how many accumulations are necessary before the result may be read. A *fetch* of the accumulator value then blocks until the accumulator has reached its prescribed limit. To solve the problem of *state* in a more general manner, and to eliminate the need for making copies of data structures, the next layer of Id introduces *M-Structures* [BNA91], which are mutable (i.e. imperative) I-Structures. Like I-Structures, every M-Structure element contains a number of presence bits, but unlike I-Structures, the addition of a *state* bit allows for synchronized mutation of the elements. The *put* operation places a value into an M-Structure element whose state is *empty*, resetting the state to *full*, and the *take* operation removes a value from an M-Structure element whose state is *full*, resetting the state to *empty*. Since the state bit does not contain enough information to allow for more than one update to an element, the language is augmented with a *barrier* operation that allows for explicit synchronization of the program. Though very well defined and controlled, the use of M-Structures takes Id into the imperative model of computation.

Though the additions to the Id core take Id further from its functional beginnings, they are deemed necessary for an efficient implementation of the language that will support scientific and real-time applications.

Sample Problems in Sisal and Id

Before we present two problems that highlight the differences between a strict functional language (Sisal) and a non-strict functional language (Id), we shall briefly discuss some of the measures by which parallel computations are compared. A *parallelism profile* (see Figure 2.3) gives us the amount of parallelism, over time, that is available to be exploited, given an infinite number or processing elements. Parallelism profiles are useful tools for measuring parallel algorithms, yielding the following measurements:

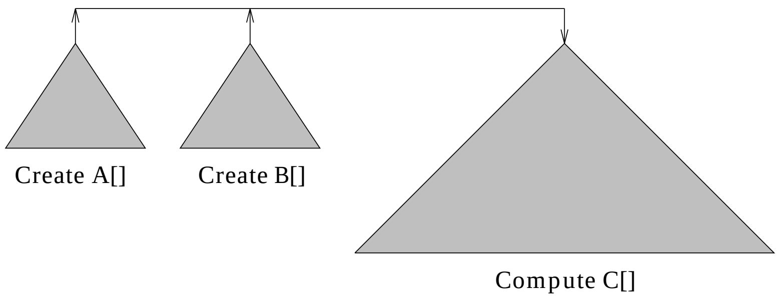

Figure 2.4: Work required for matrix multiply

- *Total work*, which is the total number of operations that have to be performed in order to complete the algorithm, and is represented in the parallelism profile as the area under the parallelism curve.

- *Peak parallelism*, which is the largest number of processors that would ever be needed to satisfy the parallelism of the algorithm, and is represented in the parallelism profile as the maximum in the graph.

- *Average parallelism*, which is the average amount of parallelism that exists over the span of the critical path length, which is represented in the graph as the ratio of total work to critical path length.

- *Critical path length*, which is the minimum amount of time needed to complete the algorithm given an infinite number of processors, and is represented in the parallelism profile as the length of the graph.

Using these measures, it is now possible to compare parallel algorithms for efficiency. The first problem we consider is *Matrix Multiplication*, $C = A \times B$, where A, B and C are each 2-dimensional matrices. For each element $C[i,j]$, we compute the inner-product $A[i,*] \times B[*,j]$. Given that there are $O(n^2)$ elements in C, and assuming the creation of each element in C requires $O(n)$ computations, the *total work* to compute C is $O(n^3)$. Assuming that the inner product is performed sequentially, the *critical path length* of the algorithm is $O(n)$, and the *average parallelism* is therefore $O(n^2)$. A graphical representation of the work done in matrix multiplication is depicted in Figure 2.4, where the creation of A and B requires $O(n^2)$ work and the computation of C requires $O(n^3)$ work.

The strict semantics of Sisal state the the creation of A and B be complete before the computation of C can begin. This allows the resource manager to allocate up to $O(n^2)$ processes for the computation of C without the possibility of deadlock, regardless of the number of processors allocated to execute the $O(n^2)$ processes. Conversely, the non-strict semantics of Id allows the resource manager to allocate up to $O(n^2)$ processes to complete the $O(n^3)$ work, representing the creation of A and B, and the computation of C. However, most of this work, represented by the computation of C, cannot be immediately completed, being blocked until their input values are available. This results in a competition for resources between the processes creating A and B and the processes computing C. Assuming that there are not enough processors to accommodate all $O(n^2)$ processes (which is a good assumption for $n > 100$), the resource manager must *wisely* select which processes are to be executed first, else the system could easily deadlock. For example, deadlock will result if the processes computing C are assigned to processors

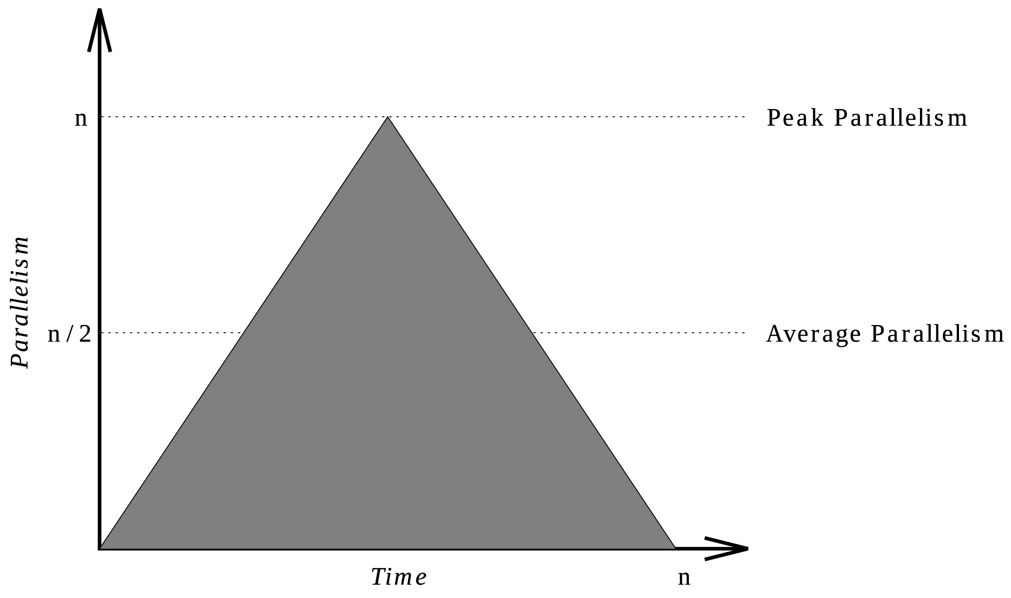

Figure 2.5: Parallelism profile for Wavefront

before the processes creating A and B. Unfortunately, the resource manager cannot distinguish among the various processes, so another scheme is necessary to safeguard against the possibility of deadlock. Current implementations of Id employ a resource management strategy called *k-bounded loops* [Cul90] for only releasing a limited number of processes into the system, but only works for well-defined loops with left-to-right data dependencies. This example demonstrates the problems of non-strictness when faced with a limited amount of resources, as all real systems are.

The second problem we consider is *Wavefront*, in which some elements of a matrix are defined in terms of other elements. A sample Wavefront problem is to build a matrix A defined as:

$$A[1,*] = 1$$
$$A[*,1] = 1$$
$$A[i,j] = A[i-1,j] + A[i-1,j-1] + A[i,j-1]$$

Note that the left and top edges of the matrix are all 1, and the other elements are computed using neighbors to the left and above. Thus the parallel computation proceeds in a "wavefront" from the top left to the bottom right. Since there are $O(n^2)$ elements in the wavefront to be computed, and each computation requires $O(1)$ steps to compute, the *total work* is $O(n^2)$, with an average parallelism of $O(n)$ (see Figure 2.5). While it is possible to program this problem in Sisal to achieve the total parallelism possible, it cannot be done in a straightforward manner, since Sisal does not allow for recursive array definitions. Here we see the limitation of strict languages in expressing a rather simple, and often used, data dependence. The Sisal implementation therefore is a "back-door" approach that uses *streams*, Sisal's version of producer-consumer parallelism, to create the matrix in a row-wise manner. Id, on the other hand, has no problem in expressing this problem exactly as it was defined, since recursive array definitions are possible in a non-strict language. This simple problem demonstrates the power of expressibility in a non-strict functional language. However, Id still has the same resource management difficulties when it comes to executing this problem. $O(n^2)$ processes (for computing the matrix values) are released into the system, but only $O(n)$ of them are capable of executing

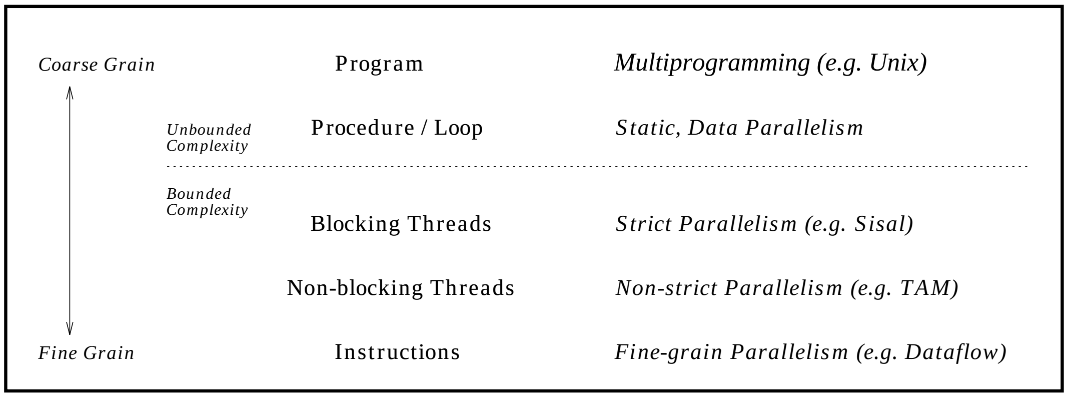

Figure 2.6: The spectrum of parallelism

in parallel (as we can see from the parallelism profile in Figure 2.5). Therefore, not only are system resources wasted, but the possibility of deadlock (a much more serious problem) still exists.

2.4 On Managing Parallelism

The type of parallelism that is exposed implicitly by a language or compiler, or explicitly by a programmer, will determine how the parallelism is to be managed. *Granularity* refers to the size of the schedulable unit of parallelism, called a *grain*, and as we see in Figure 2.6, there is a spectrum of parallelism (i.e. grain sizes) possible:

- *Program* level parallelism occurs when an operating system executes several programs in parallel, often called *multiprogramming* [PS85]. Each program is instantiated as a process, and the operating system schedules all of the available processes according to some scheduling scheme, such as *round robin with multilevel feedback*, as is done in the Unix operating system [Bac86]. For program level parallelism, the granularity is very coarse (i.e. large grains), and the complexity of each grain in *unbounded*, which makes scheduling decisions difficult since the scheduler has no idea how long a process will execute. This means that scheduling must be done at runtime (dynamic scheduling) rather than at compile time (static scheduling).

- *Procedure* or *loop* level parallelism occurs when a programmer (or compiler) decides that two loops (or procedures) can safely execute in parallel, or that different iterations of the same loop may safely execute in parallel. In the case of parallel loops, we call this *data parallelism*, and in the case of parallel procedures, we call this *functional parallelism*. In both cases, the complexity of each grain is still unbounded, and so dynamic scheduling is necessary.

- *Thread* level parallelism occurs when the grain size is a *basic block*. Since basic blocks do not contain loops (by definition [ASU86]) their complexity is bounded and scheduling becomes more manageable. If the threads are allowed to encompass long-latency operations, such as *read* and *write*, then we define them as *blocking* threads. This is the case with the threads used in the current multiprocessor-based implementation of Sisal [Can92]. Threads which are not allowed to encompass long-latency operations are defined as *non-blocking* threads. This is the case with TAM [CSS$^+$91] threads.

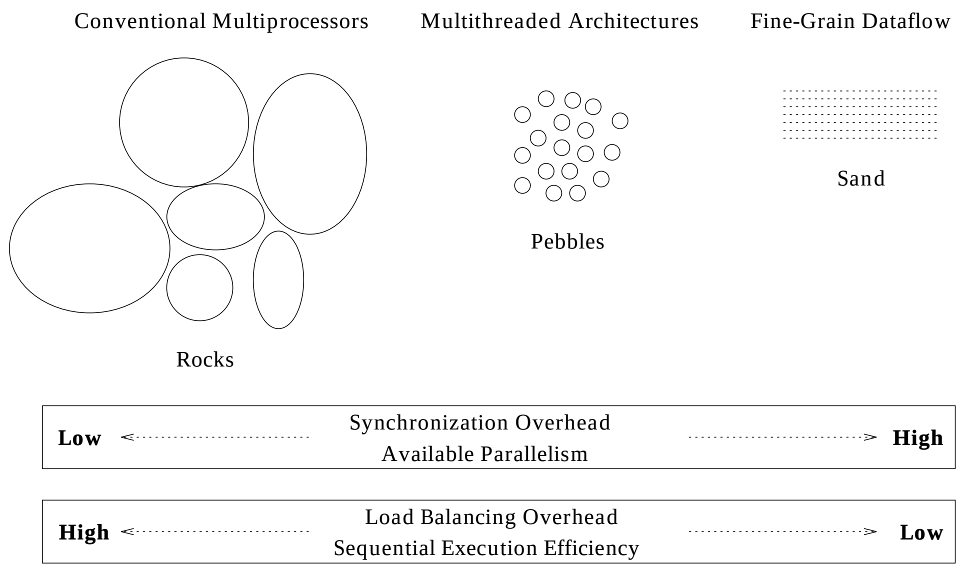

Figure 2.7: Various grain sizes

- *Instruction* level parallelism occurs when individual instructions are identified as being able to execute in parallel, such as in fine-grained dataflow [BS89] and superscalar architectures [Fis87, RYYT89]. The granularity is very fine at this level, which can result in a large amount of parallelism, potentially causing runtime overhead. There are techniques that can be implemented to reduce this overhead. Fine-grained dataflow machines often employ a throttling technique [RS87] to match program and machine parallelism, and superscalar architectures often rely on compiler assistance in locating and scheduling the parallelism [Fis87].

The difficulty in managing the exposed parallelism, with respect to scheduling and load balancing, decreases as the grain size decreases, but the synchronization overhead increases as the grain size decreases (see Figure 2.7). This relationship is reflected in Sarkar's parallelism vs. overhead graph [Sar89], in which the "ideal" granularity is expressed as the intersection of the decreasing overhead curve and the increasing granularity curve. Multi-threaded computation is an attempt at realizing this "ideal" granularity, which for our spectrum of grain sizes, falls somewhere between blocking and non-blocking threads. For the remainder of this discussion, we restrict ourselves to this level of granularity (the pebbles depicted in Figure 2.7).

Once we have decided on a granularity and exposed it in the program, we are faced with the scheduling problem: assign the tasks that result from partitioning the program (i.e. exposing the parallelism) to the available processors so as to minimize the parallel execution time. In general, task scheduling is an NP-complete problem, and the reader is referred to [Sar89] for a more detailed study of the problem and its application to multiprocessors.

Data Distribution

Related to the problem of task scheduling is the problem of *data distribution*. Informally, the problem of data distribution is to divide the data structures that a program uses among the memory elements so as to minimize certain desired measures, such as total execution time or number of remote references. For a shared memory multiprocessor, this is a trivial task. Since, by definition of shared memory, placement of a data structure does not effect performance, the data structures can be allocated (or distributed) anywhere among the shared memory modules. However, for non-shared memory multiprocessors, such as distributed memory multiprocessors and NUMA architectures, data distribution becomes a more serious and complex problem, but one that must be addressed if reasonable performance is to be achieved. There are two important points to be made about data distribution for a distributed memory multiprocessor:

1. The time required to access local memory is typically orders of magnitude less than the time to access remote memory. Therefore, assuming no attempt is made to tolerate the latency, optimal execution time occurs only when all data references are local. However, this is clearly not possible. For example, every processor may need every element of a data structure. If the data structure is to be distributed, then clearly some processors will not have local access to the elements they need. Another problem occurs when the reference pattern is unknown at the time of distributing the data. Also, some interconnection networks (e.g. ring) have faster access times to *neighboring* nodes than to distant nodes. For these non-uniform access machines, non-local data references should be on neighboring nodes as opposed to distant nodes. Again, this is not always possible.

2. The principle of *locality* states that memory references are grouped together in both space (*spatial locality*) and time (*temporal locality*). This implies that if we reference a particular data item, then there is a good probability that we will issue a reference for the same data item very soon (temporal locality), or we will reference another data item that is physically close to the original reference (spatial locality).

If we combine these points, then we have the outline for a data distribution scheme:

- Determine the access pattern.
- Distribute the data so as to maximize the local references.
- Distribute the non-local references so as to maximize the neighboring references (only if there is a discrepancy between neighboring and distant access latencies).
- If a reference is remote, then attempt to make subsequent accesses to this reference local, or attempt to make subsequent references to related items local, or both.

This distribution scheme takes advantage of the observations that were made about the behavior of distributed memory multiprocessors and their programs, but does not address the *feasibility* of the approach. Of the assumptions made, the ability to determine access patterns is by far the most idealistic. This is reflected in the current alternative methods being used for distributing data structures:

- The compiler controls the distribution of data structures. The idea is to distribute the data structures according to some *distribution function*, and then to analyze the array subscripts to determine whether or not, for a particular thread, a given reference is local or remote. If the reference is remote, the the appropriate communication primitives are generated to retrieve the value at runtime. The distribution functions are formalized so that a compiler can make use of them, and this formalization is equally useful when considering other approaches. Although some systems present formal analysis that attempts to determine the best distribution from analyzing the access patterns [OH92], most parallelizing compilers use user-supplied distributions, either in the form of language extensions or pragmas (compiler directives) [HKT92, ZBG86].

- The compiler controls the distribution with the help of run-time profiles [Sar89]. Again, this approach attempts to help the automated distribution process, but rather than have the programmer tell the compiler how the data will be accessed, the compiler simply "watches" several characteristic runs and notes the distribution patterns used for those runs. The compiler then selects a distribution function that will come closest to this observed reference behavior. The advantage this approach has over the pragmas is that the programmer may be unaware of the reference pattern, and thus be unable to help with the distribution. The disadvantage is that if the profiled runs are not characteristic of the actual reference patterns, or if the reference patterns vary with the input data, then this approach may be misleading.

- The programmer controls the distribution explicitly. Since all of the above techniques require intelligent compilers that are not always (or often) available, a common technique for distributing data is for the programmer to explicitly distribute the data and then insert the appropriate communication primitives into the source code, all "by hand." Though this approach requires very little software support (only the message passing interface is needed), the user is required to determine the access patterns and then distribute the data accordingly using explicit message passing primitives. Clearly this contradicts the efforts of raising programming to a higher level of abstraction.

Two Fundamental Issues, Revisited

Having introduced the concepts of tasks, scheduling, and data distribution, we take another look at the issues of latency and synchronization, and the possibility of trading parallelism for latency. We examine a software approach to addressing the two fundamental issues of multiprocessing: the idea of *avoiding* latency using intelligent mapping functions for tying task distribution to data distribution, and the idea of *hiding* latency by overlapping parallelism with latency.

TAM and Id

Dave Culler et al. [CSS$^+$91] present a compiler-based approach to tolerating latency called the *Threaded Abstract Machine* (TAM). TAM provides an execution model for the fine-grain parallelism (non-blocking threads in our spectrum) that is generated by the non-strict language Id. The basic concept is that synchronization, scheduling, and storage management are placed under compiler control by making these operations explicit in the instruction set. TAM provides software support for I-Structures so that the actual hardware does not require presence bits. TAM also implements a software multithreading scheme that switches threads whenever a long latency operation is encountered. This requires that the task switching incur a minimal overhead, which is accomplished by using very lightweight threads that carry little context with them. Since the initiation of a split-phased transaction do not block a thread, it is possible for the compiler to determine how long each thread will execute, and thus is able to perform the scheduling at compile time. Preliminary measurements show that the approach is feasible, but due to the software implementation of I-Structures and multithreading, and the small size of the threads, the overhead is currently too great to compete with imperative language counterparts executing von Neumann code.

2.5 Proposed Research

Our research focuses on the feasibility of providing runtime support for a distributed memory implementation of a strict functional language. Our approach centers on a runtime system that copes with the two fundamental issues of multiprocessing by providing the following services to the compiler:

- Various *task distribution* schemes that allow for the efficient distribution and reduction of large and small numbers of tasks among a variety of processor configurations.
- The ability to *tolerate* latency by providing software support for multithreading.
- A *single addressing space* to support the shared memory model of computation that the Sisal compiler assumes to exist.
- The ability to *avoid* latency by providing a rich set of data mapping functions that allow for data and tasks to be aligned so that remote memory references are minimized.

The VISA runtime system is designed as the culmination of these goals and, more importantly, as a vehicle for studying the effects of latency avoidance and latency tolerance on strict functional-language applications running on conventional distributed memory multiprocessor architectures. Though it may be attractive to implement these ideas either in the compiler or by using special-purpose hardware, a runtime approach has several advantages:

- *Runtime information and adaptability.* A compiler can produce fast code that avoids latency when it can deduce the access patterns and thread lengths of a program. However, this information may not be available until runtime parameters have been processed. Also, many "adaptive" applications change their access patterns and data locations over their execution lifetime, making the overhead of current compiler-based solutions intolerable [KM91]. For this set of applications, compiler-based approaches offer little practical support. A runtime system approach can utilize dynamic information in generating task and data mappings, as well as adapt to changing access patterns.
- *Machine and language independence.* Programmers have long been aware that the language design has a significant impact on how easily an algorithm can be transformed into working code. Even the so-called "general purpose" languages are recognized as being suited for certain problem solving approaches. The transformation process is more tedious and error prone when the conceptual models supported by the language relate only peripherally to the problem-solving model of the programmer. Thus, the need to support various languages is real [PB90]. A runtime system

can be both machine and language independent. This offers the advantage of being able to re-use the concepts of latency avoidance and tolerance for a variety of machines that do not support these features directly, and for languages that rely on them.

- *Short development time.* By implementing these facilities in software, we can measure their effectiveness on a variety of sample applications in a relatively short period of time, as compared with a long hardware development time.

Chapter 3

RELATED RESEARCH

If you steal ideas from one source, that's plagiarism, but if you steal ideas from more than one source, that's research.
– Laurendo Almeida, Brazilian guitarist.

In this chapter we examine some of the related research that not only pertains to ours, but helps to define it. We start by examining related multithreaded architectures, whose primary goal is a hardware solution to the two fundamental problems. Next we examine message passing abstractions that help to define the abstraction implemented in VISA. Finally we survey other approaches that provide for a shared memory programming paradigm, including those implemented at the language level, the operating system level (DSM), and the hardware level.

3.1 Multithreaded architectures

Multithreaded architectures embody the belief that hardware support is needed to efficiently handle the switching and synchronization of lightweight tasks, which is necessary for tolerating the large latencies found in all large multiprocessor systems. The basic idea behind all multithreaded architectures is to run threads of sequential code through a fast von Neumann processor, but to schedule and synchronize the threads using a data-driven approach, as is done for individual instructions in fine-grain dataflow machines.

3.1.1 *T

The *T architecture [Bec92] focuses on designing a processor node that will serve as the building block for a massively parallel machine, with hardware support for multithreaded execution. The *T is centered around the Motorola 88110MP, which is a customized version of the Motorola 88110 Symmetric Superscalar microprocessor that includes hardware support for fine-grain communication and synchronization. The 88110MP instruction set is a superset of the 88110 instruction set, including message transmission and reception instructions, microthread scheduling instructions, and processor configuration instructions. The result is a processor design that hopes to support a threaded model of computation without sacrificing sequential execution speed.

The *T utilizes a register-set model of networking, similar to the J-Machine [DCF$^+$89] except that *T messages are limited to 24 words so that the entire message can be placed into registers for transmission or upon reception. A set of send and receive registers for storing outgoing and incoming messages, and a few special-purpose processor instructions provide a *minimal* message passing interface that can be customized to fit a variety of message passing models.

Multithreading in the *T is supported by the *microthreading model*, which defines a thread as non-blocking code fragment. Normally suspensive operations, such as remote access and synchronization operations, are expressed in terms of *split-phased* instructions, where one thread initiates the operation and arranges to schedule another thread to handle the completion of the operation. This non-blocking model allows for *T to employ a simple microthread stack for storing microthread descriptors ready to execute, providing an extremely fast thread-switching mechanism. Microthreads which are designed to handle messages are termed *message handlers* ("inlets" in [CSS+91]), and microthreads designed to perform computations are called *computation microthreads* ("threads" in [CSS+91]). Special 88110MP instructions support the creation (`fork`), scheduling (`post`), and synchronization (`cfork, cpost`) of microthreads. To avoid expensive interrupts, the `rxpoll` instruction can be used to check for incoming messages and, if a message is present, move the message into the receive registers and schedule a message handler to process the message.

The *T architecture represents a direct approach to marrying the strengths of von Neumann and dataflow architectures. Latency tolerance is supported at the hardware level through the use of specialized processor instructions that provide a minimal interface to the network and hardware support for thread creation, scheduling, synchronization. The result is an architecture that will attempt to efficiently execute a wide spectrum of programs written for a variety of computational models, from dataflow to data parallel.

3.1.2 The Denelcor HEP and Tera Computer Systems

Burton Smith has designed several machine architectures, among which are the Denelcor Hep [Smi85] and the Tera Computer System [AAC+91]. Both are multithreaded architectures that provide hardware support for replicated processor states and split-phased transactions, with the goal of masking long latency operations by efficiently switching among threads. All registers and memory locations contain presence bits for fine-grain synchronization, and all accesses can choose to test them. In order to efficiently switch among the currently executing threads, the HEP and Tera both use the technique of replicating the processor state (i.e. registers) so that a context switch does not have to save and restore ("spill") registers, and thus is very fast. The threads are interleaved using a pipelined processor. The Tera system guarantees that the results of arithmetic and conditional operations are available to the next instruction in the thread, and memory operations are handled through an explicit dependence lookahead field in the instruction (assuming that the compiler can set the fields properly). The memory system on both machines is physically distributed, though Smith claims that these machines are shared memory machines: The placement of data in the system does not affect the performance of the related application. For the HEP, all memory is global, and a thread making a memory request is removed from the pipeline and placed in a waiting area until the memory request has been satisfied. Each processor is capable of supporting up to 64 threads, which turned out to be too few for keeping the processors busy so that all memory latency is hidden.

The Tera represents an attempt to improve upon the HEP design, and to create a machine that has hardware support for many forms of parallelism (heterogeneous parallelism). The thread count was increased from 64 to 128 threads per processor, the pipeline problem was addressed using the explicit dependence field of the instruction set, and the use of local memory caches helps in the execution of threads that exhibit a strong degree of locality. Tera employs two "close" memories that can be used by the compiler to avoid accessing global memory. This local memory is often used to store register spills and thread stacks. Tera also associates four tag bits with each memory location and register (as opposed to 1 presence bit in the HEP): a forwarding bit, a full/empty bit, and two data trap bits. The full/empty bit is used in conjunction with the data trap bits to form several modes of access, including *normal*, in which the full/empty bit is ignored, *future*, in which reads and writes occur only if the full/empty bit is full, and *synchronized*, which is similar to the HEP operation: read only on full and set empty, and write only on empty and set full. Tera also supports various levels of parallelism, including very fine-grained (via the instruction pipeline), fine-grained (loop level parallelism), medium-grained (more than 100 instructions grouped into a "chore"), and course-grained (multiprogramming various tasks).

The HEP and Tera architectures both represent the desire to hide the effects of latency and synchronization by using special-purpose hardware and complicated compilers, whose job it is to identify and exploit the various levels of parallelism that exist in most applications. Though the HEP fell to the wayside for various economic and scientific reasons, it nonetheless represents the first attempt at providing hardware support for multithreading. The Tera Computer System hopes to alleviate the HEP shortcoming by providing an improved set of hardware for supporting multithreading, and a very intelligent compiler that can take advantage of the many forms of parallelism that the Tera architecture is equipped to handle.

3.1.3 EM-4 and EM-5

The EM-4 and EM-5 [SYH$^+$89, YSH$^+$89] are multithreaded machines designed at the Electrotechnical Laboratory (ETL) in Tsukuba, Japan. The EM-4 node contains an Input Buffer, a Fetch and Add Unit, and an Execution Unit, all in a single VLSI chip [SYHY87]. This has the advantage of not incurring off-chip delays when dealing with the dataflow processor, but the disadvantage of being a very specialized chip. Rather than using a simulation, the EM-4 designers have built an 80-node prototype for testing and evaluation. As for the abstract machine, the EM-4 uses the concept of threads (called *strongly connected blocks* in EM-4 terminology) to represent sequential blocks of code that are to be executed using a von Neumann-style processor, but that are switched and scheduled using a dataflow-style matching approach. The EM-5 design is an advanced design of the EM-4 that is capable of supporting up to 16K processors, with an estimated peak processing rate of 3.2 teraflops. The EM-5 design is scheduled to be completed in 1993, and a first prototype is scheduled for March of 1994. Both systems are being targeted for a number of programming models, including imperative, object oriented, and functional.

3.2 Message Passing Systems

A shared memory multiprocessor uses the shared memory to exchange data among the processors, and can use semaphores and monitors for mutual exclusion and synchronization. However, when we move to a distributed memory machine, **message passing** is used for both information exchange and synchronization. Information is exchanged when one processor *send*s a message to another processor, which is required to *receive* it [Tan87]. This characteristic in message passing that "things have to be said twice" makes programming often complicated and error prone. However, like the traditional *goto* statement, *send* and *receive* are very powerful and flexible statements from which other communication primitives can be built. For example two additional system calls, built from the *send* primitive, are *broadcast*, in which the message is sent to every other processor, and *multicast*, in which the message is sent to a subset of the remaining processors. Additionally, the message passing primitives can typically operate in either *blocking* or *non-blocking* mode. In blocking mode, a *send* will wait until the corresponding *receive* has been initiated, and likewise the *receive* will wait until the corresponding *send* has been initiated and the data arrives. This provides for implicit synchronization among the communicating processes. In non-blocking mode, both *send* and *receive* return immediately after posting their requests to the network. For the *send*, this just means that the sending process is not sure when (or if) the corresponding receive is initiated. For the *receive*, if the receive request is not immediately satisfied when the system call is made (i.e. the data to be read was already there), then the call returns and the process must *poll* the communications port for the desired message to know when it arrives. Some operating systems, such as Vertex [nCU90a], provide a message polling capability so that the process does not have to explicitly poll for the message. When the message does arrive, the operating system will issue an *interrupt* to the process on behalf of the message.

3.2.1 The Reactive Kernel

The Reactive Kernel [SSS88] is an operating system kernel that provides the user with a standard message passing abstraction, hiding some of the unnecessary details of communications from the user. The abstraction supports a non-blocking send and both a blocking and non-blocking receive. The non-blocking send simply places the message into the network and returns. The blocking receive waits for a message that matches its key, thus blocking the processor from proceeding. This version of the receive is used for synchronization purposes. The non-blocking receive returns immediately, either succeeding if the desired message was already waiting, or failing of the message has not yet arrived, making no attempt to re-try the receive operation (i.e. polling). Thus it remains the programmer's responsibility to poll for a message.

3.2.2 Active Messages

Active Messages [vECGS92] focuses on providing a minimal message passing interface that utilizes a *split-phased* approach to message passing, rather than the traditional send and receive operations. In the traditional approach to message passing, the send and receive operations can be either blocking (synchronous) or non-blocking (asynchronous) operations. Blocking operations eliminate the need for intermediate copies since buffering is not needed, but both network and processor utilization suffer as a result. Non-blocking operations increase the network and processor efficiency by relaxing the synchronization restrictions, but require that the message be buffered since there is no guarantee that it will be immediately handled on the receiving end.

Active Messages attempts to combine the benefits of each approach, namely high processor and network utilization without intermediate buffering. In addition to data, a message contains a pointer to the handler that will process the message upon arrival. As opposed to a remote procedure call (RPC), the role of a handler is not to perform computation on the data, but rather to get the data out of the network so that buffering

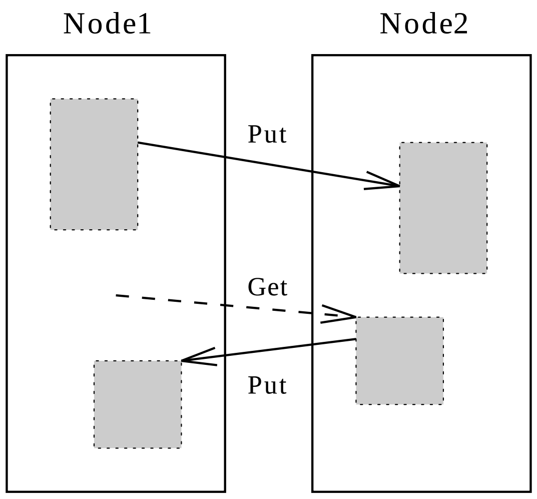

Figure 3.1: Split-phased active message operations

at the destination is not necessary. Using active messages, message passing primitives can now be expressed as split-phased remote memory references (see Figure 3.1). *Put* copies a local memory block into a remote memory at an address specified by the sender. A handler in the put message will take care of placing the data in the proper place on the remote node. Thus the thread can continue operation while the remote handler takes care of placing the datum in the specified location. *Get* retrieves a block of remote memory and makes a local copy. The local handler sends a message to the remote processor, specifying the memory location desired and the location of where to place the datum when it returns. Again, the operation proceeds and the remote handler sets up a *put* message that will subsequently reply with the requested information. Both operations are non-blocking and are handled asynchronously by the remote processors.

3.3 Supporting the Shared Memory Programming Paradigm

Central to this thesis is the support of the shared memory programming model on top of a distributed memory architecture, commonly referred to as a *distributed shared memory (DSM)* or *virtual shared memory (VSM)* system. Two important issues must be addressed when designing a shared memory abstraction:

- *Coherence.* Most DSM systems provide replication of data to increase availability and, if the replicated data is mutable, the system must provide a means to ensure coherence (consistency). There are two levels of consistency that are often supported: *strong consistency* [CF78], which defines a sequential ordering of memory references that ensures a consistent view of memory at all times, and *weak consistency* [DSB86], which requires that a memory system be coherent only at synchronization points. If coherence is necessary, then a *coherence protocol* is employed to guarantee the desired level of consistency. Broadly, these protocols can be classified into three approaches:

  – *Write-broadcast*: Also known as write-update, this policy updates all copies of the mutable data item for each write.

  - *Write-invalidate*: The data item is updated, and all other copies are marked as invalid so that subsequent attempts to read them will force replacement.
  - *Lock-based*: In this approach, reads and writes are preceded and succeeded by lock request and releases.

- *Granularity.* Every DSM system must choose some level of granularity at which the shared memory is supported. The granularity of a coherence strategy is the unit size of a "cacheable" block of data. In a cache system the unit of granularity is called a *block* or *line*, and in a virtual memory system it's called a *page*. Since the granularity of accesses in DSM systems may preclude using true shared memory primitives (such as test-and-set), alternatives are often provided that take advantage of specific implementation details of a DSM system.

DSM approaches can be categorized by the level that implements the shared memory abstraction.

3.3.1 Language-Based Approaches

At the language level, compilers and runtime systems implement a shared addressing space by offering the programmer the use of a limited set of shared data structures, typically arrays. The distribution of these data structures is based on information retrieved from reference analysis (fully-automated), profiling (partially-automated), or user pragmas (manually-automated). When an element is to be accessed, its location is determined and, if remote, padded with communication primitives to get (or put) the datum. The granularity at the language level is typically that of the individual data structure, which allows for greater flexibility in the distribution mechanism. Coherence is typically not an issue since either the compiler doesn't support replicated data structures, or coherence is controlled by ordering access to the data.

- **Fortran D** [FHK$^+$90, HKT92] is a version of Fortran that provides language extensions for specifying both the *problem mapping*, which specifies how arrays should be aligned irrespective of the underlying architecture, and *machine mapping*, which specifies how the arrays should be distributed onto the actual parallel machine. The language extensions `DECOMPOSITION`, `ALIGN`, and `DISTRIBUTE` are used by the programmer to tell the compiler how to align and distribute the arrays. A `DECOMPOSITION` represents an abstract problem or index domain. Each element of a decomposition represents a unit of computation. `ALIGN` is used to map arrays onto decompositions, corresponding to the *problem mapping*. Arrays mapped onto the same decomposition are automatically aligned with one another. `DISTRIBUTE` maps the decomposition onto the finite resources of the physical machine, corresponding to the *machine mapping*.

  The compiler uses the distribution information provided by the user to distribute the program data structures and, if possible, generate the communication *send* and *receive* sets that specify how data is to be exchanged prior to the execution of a parallel loop. When it is not possible to compute these communication sets at compile time, as is the case with indirect subscripts, the compiler generates the code necessary to compute these sets at runtime. Almost all parallelizing Fortran compilers use the *inspector/executor* model [SCMB90] for generating these communication sets at runtime. The *inspector* iterates through the loop once to compute the communication sets and passes this information to the *executor*, who then gathers the necessary data and executes the loop. Assuming the subscripts do not change inside of the loop, the overhead of the inspector is amortized over the execution of the loop. However, if the subscripts do change inside of the loop, as is the case with adaptive mesh problems, the inspector must be executed every time the subscripts change, which in the worst case is every iteration. Since the overhead of the inspector is typically greater (e.g. 1.5x) than the actual execution time of a single iteration [KM91], executing the inspector each for each loop iteration would result in a 150% overhead.

- **Kali** [KM91] provides a set of language primitives that supports a data parallel model of execution using a global naming space. The programmer must specify: the actual processor topology, the distribution of data structures across this topology, and the parallel loops and where they are to be executed. The Kali compiler then maps the virtual processors used by the programmer onto the actual processors and inserts the message passing primitives as needed. As with the FortranD approach, communication sets are generated at compile time if possible, or at runtime using the inspector/executer model. This approach requires more specification that the Fortran D approach, and hence less sophisticated compilers.

- **Linda** [Gel85] is not really a parallel language, but a mechanism for extending sequential languages such as C (C-Linda) and Fortran (Fortran-Linda). Processes communicate via a single name space called *Tuple Space* rather than the more traditional shared memory of message passing paradigms. Tuple Space is addressed *associatively* (by contents) rather than by address, and is manipulated by three atomic operations: *out* adds a tuple to Tuple Space, *read* matches a tuple and returns a copy of it, and *in* matches a tuple and removes it from Tuple Space. Both *read* and *in* block until a matching tuple exists in Tuple Space. If two processes simultaneously try to remove a tuple, one will succeed and the other will block. Since tuples must be removed to be changed (i.e. tuples are immutable), simultaneous updates are automatically synchronized by the *in* primitive. The distributed memory version of Linda associates an identifier with each tuple that identifies the processor that owns the tuple, and any updates are sent to the owner for modification, which is similar to the *owner computes rule* used in Fortran D.

- **Orca** [BT88a] is an object-oriented language based on a shared memory model called the *shared data-object* model, in which shared data are encapsulated in passive (data) objects and accessible only through a set of *operations* defined by the object type. When a parent spawns a child, it may pass any of its objects as *shared* parameters to the child, thus distributing the object among the descendants, possibly on different memory modules. Changes made to an object are visible to other processes that hold the object, thus preserving strong consistency. The distributed implementation is based on replication and migration, and a write-update protocol is used to keep replicated copies consistent.

- **Paralex** [OBAAD91] is a graphical programming language in which the user builds a course-grain dataflow graph that represents the problem to be executed on a heterogeneous network of machines. The graph nodes contain information such as the sequential code that defines the node, the data specifications, and information relating to host selection and fault tolerance required. The compiler organizes the computations according to the data flow graph and inserts message passing primitives if connected graph nodes are on different processors. Consistency is maintained by replicating only immutable data objects.

3.3.2 Operating System-Based Approaches

It was at the level of the operating system that the idea of providing a single addressing space over a set of distributed memories was first implemented, and though there are language- and hardware-based approaches to performing this service, the majority of DSM systems are still based operating system-based. Virtual memory is striped across all of the memories and virtual memory managers, informed by page faults, perform the task of swapping virtual pages so that the remote references are turned into local references. Thus the granularity of most operating system approaches is that of the memory *page*. Much of the design is focused on coherence among the pages, as the entire set of memories can be viewed as a cache for the virtual pages, some of which may contain duplicate information. Virtual memory issues such as thrashing and efficient address translation must also be addressed.

- **Apollo Domain** [LLD$^+$83] was one of the earliest systems to employ the DSM approach for ensuring strong consistency of shared objects in a local area network (LAN) of Apollo workstations and servers. The objects were maintained by a two-level distributed object storage system (OSS). The first layer provided access to local objects and the second layer provided access to remote objects, transparent to the user. A two-level approach is also used to maintain consistency: a lower level for detecting coherence violations and a higher level for locking objects. Domain is unique among the DSM systems for employing a version numbering scheme that does not prevent inconsistencies, but rather it detects them.

- **Ivy** [Li86] provides a shared virtual memory (SVM) space for a group of Apollo workstations connected by a token-ring network. This is similar to the Domain system, except that the granularity of access is a physical page of memory rather than an object. To maintain coherence, Ivy uses a write-invalidate scheme with multiple-readers/single-writer semantics for strong consistency. In this approach, all read-only copies of a page are invalidated when the page is updated. Page ownership is determined using several algorithms, including *centralized, fixed distributed*, and *dynamic distributed.* Ivy also provides synchronization support through the *eventcount* primitive. Variations of Ivy also exist for heterogeneous architectures (**Mermaid** [LSWZ89]) and for Hypercube architectures (**Shiva** [LS89] and **Koan** [LP91]).

- **Choices** [SMC90] is an operating system architecture that uses class hierarchies and object-oriented programming to support the building of customized operating systems for shared memory and network operating systems. Memory objects are either local or remote, and the remote memory object class contains handling routines for page-faults incurred by a processor accessing the remote object. A write-invalidate scheme, similar to Ivy, is used to maintain strong consistency. Choices also provides for locking a page to guarantee atomic updates, which is similar to the *read-write* lock provided in the Clouds system.

- **Clouds** [RAK88] is an object-based distributed operating system that takes an object-oriented approach to separating the address space, which is represented by globally-accessible objects, from the computation, which is represented by threads. Objects are passive entities that comprise the global naming space, and are composed of segments, which provide the granularity of sharing. Threads execute within the context of an object. Since threads may access other objects, they are not associated with a single address space and thus may span machine boundaries. Each node contains a Distributed Shared Memory Controller (DSMC) which owns and maintains the segments created on that node. The DSMC's use a lock-based protocol for strong consistency that unifies synchronization and the transport of data. Both exclusive (read-write) and shared (read-only) locks are supported.

- **Mach** [RTY$^+$87] is a multiprocessor operating system kernel that provides two shared memory abstractions: *copy-on-write* and *read-write*. In copy-on-write sharing, unrelated tasks share an address space without the actual data being copied. However, the first task that attempts to write to the data gets a copy of it and the copies become distinct from this point on. In read-write sharing, a shared memory region is created with one of the following inheritance attributes: *shared, copy*, or *none*. Shared pages are physically shared between the parent and children; i.e. there is exactly one copy of the page. Copy pages give each child a copy of the page, and none specifies that no sharing is to take place between the parent and child. Mach provides strong coherence using a write-invalidate scheme for sharing pages across the network.

- **Mether** [MF90] provides a set of shared memory mechanisms for a network of Suns. Mether differs from most other DSM systems in that it does not provide memory coherence – a process may continue to write to a page without the changes being reflected in the other copies. The other page copies are updated in one of the following three ways: (1) the process with the consistent copy may invalidate all other copies; (2) the process with an inconsistent copy may invalidate its own copy; (3) the process with an inconsistent copy may request a copy of the page. As should be evident, the user is responsible for tailoring the consistency requirements to match the needs of the application.

- **Munin** [BCZ89] is a DSM system that provides multiple consistency protocols and uses a form of weak consistency called *release consistency*, which was introduced by the Dash system. Munin allows the coherence protocol to be tailored to the access patterns of the data, which is determined by the user. Current access patterns supported include:

  - *Read-only.* Objects that are not changed after initialization. Replication is used since these objects cannot become inconsistent.
  - *Migratory.* Objects that are accessed by a single thread at a time, such as a critical section datum. One copy of the datum is kept and a lock-based protocol is used to ensure consistency.
  - *Producer-consumer.* Objects written by one thread and read by another. Replication is used with a write-update coherence protocol, since the number of processors effected by the updates is typically only one.

- *Reduction.* Objects that perform an atomic lock-read-write-unlock operation. While similar to migratory objects, reduction objects are kept in one location and coherence is guaranteed with a lock-based protocol that is inherent to the object.
- *Concurrent-write-shared.* Objects that are concurrently written by multiple threads, without the writes being synchronized. The programmer ensures that the writes occur to different portions of the object, thus alleviating the need for synchronization. Coherence is maintained by merging separate copies of the object after all the writes have been processed.
- *Result.* Objects that are accessed in phases. They are alternately updated in parallel by multiple threads, then read by a single thread. All copies are sent to the read thread for modification.

3.3.3 Hardware-Based Approaches

At the hardware level, the shared addressing space is divided into small blocks, much like cache lines, and special communication processors ensure that the processors have coherent copies of the blocks using a hardware-based cache-coherence protocol, such as *snooping* [KMRS86]. Most hardware-based approaches are similar in spirit to the operating system-based approach, the difference being that the granularity is typically smaller and the coherence protocol is implemented in hardware rather than software.

- **CapNet** [TF90] is a project designed to provide DSM across a wide-area network, with a primary goal of investigation of network support to reduce latency.

- **Dash** [GLL+90] is a hardware-based DSM system that uses an invalidation-based coherence protocol to provide a relaxed form of weak consistency called *release consistency*, in which each shared memory access is classified as either a *synchronization* or an *ordinary* access, and all synchronization access must be weakly-consistent. Access granularity is 16 bytes, which is smaller than any other DSM system surveyed and is possible due to the hardware-based coherence protocols that act much like hardware-based coherence protocols for multiple caches on a shared memory system [BK85].

- **Memnet** [Del88] is a shared token-ring network that provides a close-coupling of processors in a distributed multiprocessor environment. Memnet has an access granularity for shared data of 32 bytes, which is much smaller than the typical page granularity of the operating-system approaches. This is possible due to the dedicated hardware that ensures consistency using a very fast token-ring-based write-invalidate protocol. Memnet does not support virtual memory, and therefore acts more like a cache management system in a shared-memory multiprocessor.

- **Plus** [BR90] is a hardware-based DSM implementation that uses a non-demand, write-update coherence protocol. Pages (unit of granularity) are only replicated at remote nodes at the request of software, otherwise all references are forwarded to the owner of a particular page. Interprocessor coherence is maintained using write barriers, which must be explicitly specified by the program.

- **KSR1** [Ken92], the first computer from Kendell Square Research, is a highly-parallel multiprocessor that is scalable to 1088 processors. Shared memory is obtained through a global name space and processor caches, guaranteed by hardware to be consistent. The patented *ALLCACHE* system maps the single addressing space into the physical caches, as is done is software by the operating system-based DSM systems, and implements a hardware-based coherence scheme to guarantee *sequential consistency*, which is similar to strong consistency. Scalability is achieved by grouping the caches into a hierarchy, starting with the processor and extending out as groups of processors (in rings) are added.

3.4 Relation to Our Research

The previous three sections have examined various research projects in the areas of multithreaded architectures, message passing systems, and distributed shared memory systems. Our research borrows ideas, but differs from each of these areas as follows:

- From the multithreaded arena, our research borrows the idea of tolerating latency by overlapping parallel threads of computation with long latency operations, such as memory access. However, we employ a software-based approach that does not require the use of special-purpose processors and tagged memory (unlike the hardware approaches), and is focused on a strict functional model of computation, unlike the TAM project, which focuses on a multithreaded implementation for a non-strict functional language. Our model of computation is tuned for efficient sequential performance, and we only use multithreading as a technique for examining the effectiveness of latency tolerance in our system.

- From the message passing arena, our research borrows the idea of using very fast asynchronous message passing, as was seen in the implementation of Active Messages. However, unlike Active Messages, our model hides the message passing abstraction within a distributed shared memory scheme so that the programmer is exposed to a shared memory model of computation as opposed to a message passing model.

- From the distributed shared memory arena, our research borrows the idea of providing a shared memory paradigm on top of a set of distributed memories. Our DSM design is embedded in a runtime system, which is similar to the language-based approach in that both provide granularity at the level of the data structure and lack sophisticated (and expensive) support for coherence protocols. We are not restricted to the *owner computes* rule, which specifies that the owner of a datum must perform the computation that defines that datum, and we are not restricted to communication sets for establishing the necessary message passing. Instead, our system uses on-the-fly address translation that works with general distribution functions and non-linear subscripts. However, runtime address translation is expensive, and so we utilize various techniques to eliminate this. Unlike the other hardware- and operating system-based approaches, which implement a more general update scheme that requires the use of a costly coherence protocol, our system employs an *owner writes* scheme for updating shared variables and a *write-broadcast* scheme for updating replicated structures. Rather than implement an expensive coherence protocol, our system relies on the compiler to ensure that the coherence of replicated data structures is not violated.

Chapter 4

TASK MANAGEMENT

The first myth of management is that it exists. The second myth of management is that success equals skill.

- Robert Heller

In this chapter we present the design of a flexible task management system for executing Sisal tasks on a distributed memory multiprocessor. The design focuses on the ability to execute both large and small numbers of tasks on both a large and a small number of processors. However, before we describe the design of the distributed task management system, it is necessary to understand the role and scope of a Sisal task. Therefore we first describe a Sisal task and the task management design that is used in shared memory implementations.

4.1 Sisal Tasks

4.1.1 Sisal Parallelism

The functional language semantics of Sisal ensures that only "true" data dependence constrains the parallelism of a program. Thus *independence* is defined in this context as having no data dependence between functions, loops, or expressions. Sisal allows for the extraction of parallelism at three levels:

- *Function-level parallelism* exists when two functions are independent of each other, and thus may be executed in parallel. Functional language semantics ensure that if functions are independent of one another (in terms of input and result values), the order of evaluation will not affect the outcome of the program (*Church-Rosser property*).
- *Loop-level parallelism* exists when the iterations of a loop are partially independent of each other, and thus may be executed in parallel. Sisal provides two forms of an iterative construct, though the task of determining which loop construct to use is left to the programmer.
  - `for initial`, used for specifying loops which must be executed sequentially or in pipeline fashion[1].

[1]However, the current Sisal compiler, OSC, does not support pipeline parallelism in the `for initial` construct

  - `for`, used for specifying loops in which the loop bodies may execute in parallel, which corresponds to the concept of *data parallelism.*

- *Instruction-level parallelism* exists when two instructions are independent of each other, and thus may be executed in parallel. Sisal programs can be represented by a dataflow graph, which exposes the instruction-level dependencies. Several dataflow architectures, including the Manchester Dataflow Machine [BS89], the RMIT/CSIRO Dataflow Machine (CSIRAC) [AE89], and the ADAM dataflow architecture [Mit92], have used these dataflow graphs to exploit instruction-level parallelism.

The current multiprocessor-based Sisal implementation is optimized for efficient execution of scientific applications. Therefore, although Sisal language semantics allow for the extraction of all three levels of parallelism, the current multiprocessor-based compiler and runtime system support only *loop-level parallelism*, as this is the dominant form of parallelism in many scientific applications.

4.1.2 The Master-Slave Model of Parallel Execution

The current Sisal runtime system employs a *master-slave* model of parallel execution, in which the roles of each process type are defined as follows:

- The **master** process executes all sequential portions of the code, including I/O, initialization, and synchronization. Additionally, the master process is responsible for setting up and initiating parallel execution of a loop, in which the following actions are performed:

  1. The parallel loop is divided into a set of independent *tasks*, where each task represents a contiguous set of loop bodies.
  2. The tasks are distributed among the participating slave processes for parallel execution.
  3. A *barrier* is established, which prevents the master from continuing until all of the slave processes have completed their tasks.

  Once the barrier has been completed, the master continues with the execution of the program.

- The **slave** processes are responsible for executing the parallel *tasks* that are distributed by the master as follows:

  1. Each slave continues to check a *ready queue* for the appearance of a task that needs to be executed.
  2. Tasks are removed from the ready queue and executed by the slave processes in parallel.
  3. The result of the task is either a partial value that needs to be combined with the global value held by the master (i.e. *reduction*), or is a set of array elements representing a portion of a newly-formed array.
  4. After reporting the results, the slave indicates that it has completed this task.

Nested parallel loops imply that the tasks representing the outer parallel loop will contain parallel loops themselves, and thus every slave process must be capable of initiating (or becoming) a master process to facilitate the parallel execution of the inner loop. Therefore it is more convenient to think of masters and slaves as *processes* rather than *processors*, since at any given time, a single processor may be playing the role of either the master of slave.

4.1.3 Slices and Activation Records

Whereas a *task* in Sisal is the general term used for parallel work, a *slice* is, specifically, a contiguous set of parallel loop bodies. The *thickness* of a slice is defined as the number of contiguous iterations it contains, and is determined by the loop bounds, the number of slave processes executing the loop, and the loop distribution strategy. The runtime data structure for a slice is an *activation record*, containing the following minimal set of information:

- A range of execution. Since Sisal slices are defined as a *contiguous* set of loop bodies, a range (*lo, hi*) is sufficient for defining the scope of parallel execution for a task.
- A code pointer, representing the function that the slave will execute. This function consists of a parameterized version of the parallel loop, where the parameters represent the range of execution.
- An argument list. Besides the range of execution, each task may require additional inputs, such as the location of a global array to be operated on, or the values of some global variables. Additionally, an output argument is specified which denotes the location of a global value or pointer to an array that is to be filled. These arguments are packed into a single record by the compiler and included in the activation record as a single argument.
- A unique loop identifier, which will be used in the case of nested parallel loops to identify which barrier counter is to be updated upon receipt of a slice-completion message.

In general, the terms *task*, *slice*, and *activation record* all represent the concept of parallel work in Sisal as a contiguous set of loop bodies and, although the terms are often interchanged, the concept remains clear.

4.2 Shared Memory Task Management

Now that we have defined the unit of parallel work in Sisal, the task, and the method of performing parallel work, the master-slave model, we will describe task management as it exists in the shared memory implementations of Sisal.

At the heart of the shared memory implementation is the *shared ready queue*, in which the master enqueues each of the activation records (or slices) and from which the slaves receive them, as depicted in Figure 4.1. The ready queue is allocated from shared memory, and thus is accessible to the slave processes on all processors. Barrier synchronization is performed by having the slaves set a *done* flag in the activation record on the shared ready queue, and the master counts the number of activations that have completed. When the

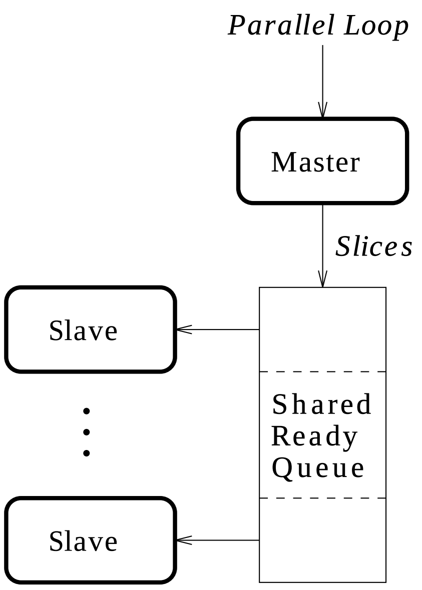

Figure 4.1: Shared memory task organization

count equals the number of activations enqueued, the activations are removed from the queue and the barrier is complete.

The advantage of this approach is that *load balancing* techniques, such as *guided self-scheduling* [PK87], are possible due to the shared nature of the queue. Load balancing is the problem of ensuring that each processor in a parallel machine performs roughly the same amount of parallel work, so that minimal time is lost on barrier synchronizations. Since all of the activation records are placed in a single location, a slave simply gets the next slice from the queue, executes the slice, and returns for more. Slaves continue to execute the slices until none remain, at which time the slaves will wait for more work.

The disadvantage of this approach is that since multiple processors have access to this queue, the queue becomes a *shared resource* that must be accessed using a critical section. The Sisal runtime system uses a lock-based protocol for ensuring mutually-exclusive access to the queue. This means that before any master or slave process may access the queue, a lock must be successfully acquired, and if the lock is not available, then the process must wait. Therefore, *contention* for the shared resource creates a runtime overhead, which is minimally the time required to execute the lock protocol, but can be extended when the process must wait for the lock. Since this overhead due to contention grows with the number of processors in the system, we say that the design does not *scale*. Still, this shared memory design has resulted in efficient implementations for shared memory multiprocessors that can quickly and efficiently access shared data structures [Can92], as the number of processors for these systems is relatively low.

4.3 Distributed Memory Task Management

Executing the Sisal master-slave model of parallel execution in a distributed memory environment implies that a shared ready queue is not available. Therefore, a new method for distributing the tasks to the slaves, and a new method for performing a barrier synchronization is required.

4.3.1 Task Distribution and Barrier Synchronization

Since shared data structures are not supported by hardware in a distributed memory multiprocessor, we give each slave process its own *private* ready queue. The master then sends the activation records to the nodes containing the slave processes using the native message passing mechanisms, where the message is received (asynchronously) and placed onto the private ready queue of the local slave process. The slave monitors its own private ready list, which can now be done without having to obtain exclusive access, and executes any slice that arrives. Resulting values and a completion message are sent back to the master upon completion of each slice.

Barrier synchronization is now performed by having the master process count each of the slice-completion messages, and when the count is equal to the number of slices distributed, the barrier is complete. However, in the case of nested parallel loops, it is not sufficient for a single counter to keep track of the slice-completion messages, since a single processor may receive slice completion messages for both a parallel outer and parallel inner loop, for which it is the master. Thus it is necessary to distinguish among the slice completion records, and this is done by assigning a unique loop identifier to each parallel loop. An identifier is removed from a pool of identifiers that is local to each processor, and is replaced when the barrier for a loop is complete. Also, each processor maintains an array of barrier counters, corresponding to the loop identifiers, so that when a slice-completion message is received, the proper barrier counter can be incremented.

This distributed design has removed the need for obtaining critical section locks to access the ready queue, thus eliminating the overhead for contention and allowing for a scalable number of slaves to be employed. However, the implicit load balancing capabilities of the shared queue have been lost, and thus it is now possible for the system load to become imbalanced. Although various dynamic load balancing schemes could be employed to ensure that the load remains balanced [Cho90], we have chosen not to implement any of these schemes for two main reasons:

1. Dynamic load balancing strategies work by migrating work from an over-utilized node to an under-utilized node, either because the under-utilized node asked for the work (*active load balancing*) or because the over-utilized node decided to give some work away (*passive load balancing*). Either way, parallel work, in our case slices, would migrate from one node to another, disrupting the original distribution pattern. Since our system is designed to minimize the number of remote references by closely tying the distribution of tasks with the distribution of data, a dynamic load balancing technique would disrupt this alignment, possibly creating excessive remote references and causing an increase in the total execution time rather than a decrease.

2. Dynamic load balancing algorithms require updated information about their surrounding processors, resulting in a runtime overhead for exchanging this information. Also, task migration results in even more message passing. Dynamic load balancing is an expensive operation, and will not be undertaken until it is determined that the loads are imbalanced, and that this imbalance could benefit from a load balancing strategy. Thus, there is not enough analysis at this point to justify the added expense of a dynamic load balancing strategy.

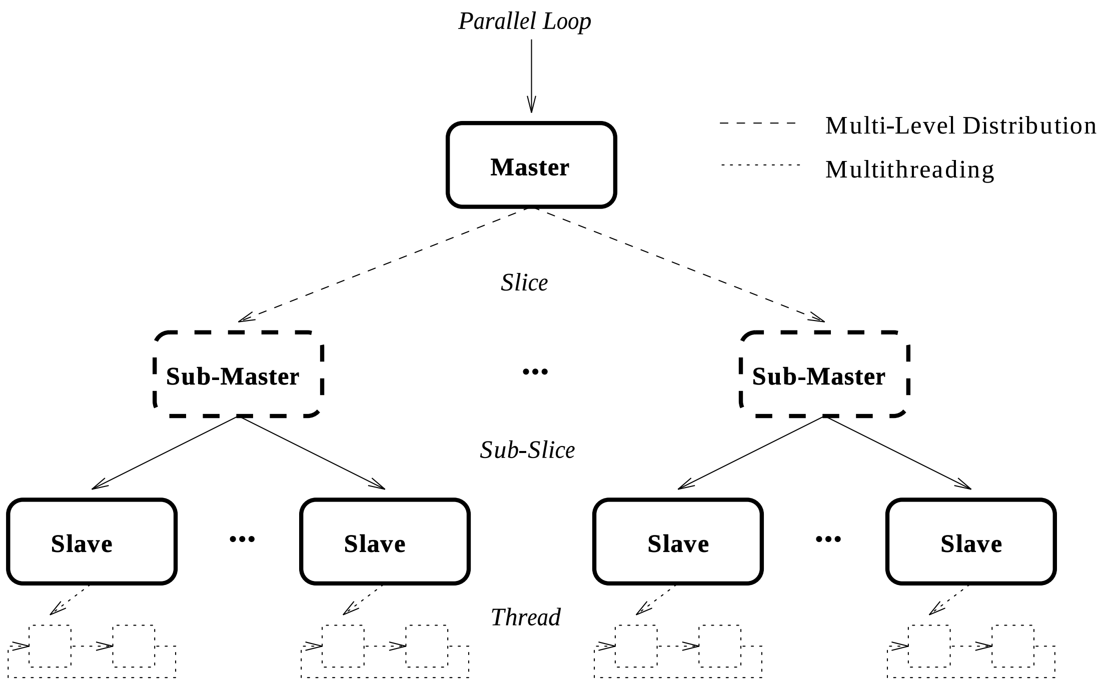

Figure 4.2: Distributed memory task organization

4.3.2 Extending the Master-Slave Model of Parallel Execution

Our distributed memory task management system is intended to execute on a wide variety of distributed memory machines, with sizes ranging from tens to thousands of processors. As previously defined in Section 4.1, Sisal parallelism is expressed in terms of a *task*, which represents a contiguous set of loop bodies that can be executed in parallel, and implemented dynamically as a *slice*, which is defined by an *activation record.* Because distributing these slices over a large number of processors can create a sequential bottleneck, and because the ability to mask latency requires the parallel execution of tasks within a single processor, we have augmented the single-level definition of Sisal parallelism to create a flexible multi-level parallelism hierarchy, as depicted in Figure 4.2. To maintain the high level of execution efficiency that is achieved by *slices* in the single-level design, each level of the hierarchy still corresponds to a contiguous set of loop bodies, where *slices* are “thicker” than *sub-slices*, and sub-slices are thicker than *threads.* Since we have changed the definition of parallel work in the Sisal system, we must also modify the master-slave execution model to account for the various levels of parallelism. Thus we augment the master-slave model with two new models of parallel execution: *multi-level distribution* and *multithreading.*

4.4 The Multi-Level Distribution Model of Parallel Execution

The multi-level distribution (MLD) model of execution is designed to parallelize the distribution of tasks from master to slaves, and parallelize the reduction of values from slaves to master. This becomes necessary when the number of slave processes gets large enough that the sequential distribution and reduction times start to lower the efficiency of the parallel execution time. The MLD model works as follows (refer to Figure 4.2):

- The **master** process divides the parallel loop into a number of *slices*, where the number is called the *fan-out degree.* These slices are then sent to **sub-master** processes, and a barrier is initiated to wait for the completion of the sub-master processes. After the barrier is complete, the master returns execution at the point following the parallel loop.

- Each **sub-master** process will divide its slice into *sub-slices*, and distribute these sub-slices among a subset of the **slave** processes *in parallel*, where the degree of parallelism is equal to the fan-out degree. Each sub-master process then performs a barrier synchronization for the sub-slices it has distributed, and collects the reduction values from the slaves. Upon completion of the barrier, the sub-masters will report their values to the master and indicate that the assigned slice has been completed.

- The **slave** processes execute the sub-slices obtained from the sub-master, and report back to the sub-master with the results of the sub-slice and an indication that the sub-slice has been completed.

The main goal of our MLD design was to enable parallel distribution and reduction without requiring the compiler to re-structure the loop slices, thus remaining independent of the compiler and allowing MLD to be enabled or disabled without having to re-compile.

For MLD execution, the runtime system divides a parallel loop into *fan-out* slices, rather than p slices, where p is the number of participating processors. The slices are then distributed to the *fan-out* sub-master processes, who will further subdivide the slices and distribute them among p / *fan-out* slave processes in parallel. If the loop results in a reduction, then each of the sub-master processes will allocate a local value to accumulate the reduction values from the slave's it controls. When the slave processes complete their slices, they will send their completion message and reduction value back to their sub-master process. When a sub-master completes the barrier for the slaves it initiated, it will report back to the master with the accumulated reduction value and a completion message. When the master completes the barrier for all of the sub-master processes, the loop is complete.

The only new structures required for this design are MLD barriers and intermediate reduction values for the the sub-master processes. Since we utilize the same loop slice method of parallelism, and loop slices are controlled by the runtime system, we meet our design goal of isolating the compiler from the operation of MLD.

As we will see in Chapter 7, the distribution of slices can create a significant sequential bottleneck when the number of processors is in the hundreds. By creating this extra level of sub-masters and sub-slices, MLD serves to parallelize this sequential bottleneck, effectively boosting the parallel efficiency.

4.5 The Multithreaded Model of Parallel Execution

The multithreaded execution model is designed to tolerate remote memory latencies by switching among local *threads* whenever a remote memory reference is initiated. Threads and slices are closely related. A slice is a contiguous set of loop bodies, represented by an activation record, and is executed by a single slave process (on a single processor). Threads are a further subdivision of slices, and are represented by a *thread descriptor*, which is a data structure similar to an activation record but augmented with fields to support the multithreading context switching mechanism. The multithreading model works as follows (refer to Figure 4.2):

- The **slave** process, having received a slice from the master process, divides the slice into *mt* threads, where *mt* is called the degree of multithreading. A thread descriptor is created for each thread, and linked together in a circular list. The threads are then scheduled for execution in round-robin fashion. Whenever a remote reference is initiated in the current thread, the current thread is suspended and the next thread in the list is started (or resumed). This continues until all the threads have completed, indicating that the slice has also been completed. The slave process then reports back to the master process with any reduction value and an indication that the slice has been completed.

Multithreading increases the parallelism in the system from p, the number of processors executing slave processes, to $p \times mt$. The increased parallelism is then used to mask the latency of remote memory references.

By allowing both the multi-level distribution model and the multithreading model to be selectively enabled and disabled, the runtime system possesses the flexibility to implement a wide range of parallel loop distribution and execution strategies.

4.5.1 Multithreading Design

Since the nCUBE/2 multiprocessor provides no hardware support for multithreading, we have designed a *software multithreading system*, adhering to the following design goals:

- *Minimal synchronization overhead for context switching.* The minimal requirement for multithreading to be effective is that two context switches must be faster than one remote memory reference, otherwise it is better to simply wait for the remote value.

- *Minimal impact on activation record structure.* Much of the Sisal runtime system, and thus VISA, relies on the current structure of Activation Records. To minimize the amount of changes to the runtime system, and to allow easy enabling and disabling of multithreading, the design should require minimal changes to the structure of the activation records. We achieve this by having a separate data-structure for a thread: a *ThreadDescriptor* (see Figure 4.3).

- *General purpose.* We do not want to design a multithreading scheme that only works because of some specific knowledge of the operations being performed within a thread.

- *Machine independence.* Finally, we want our design to be machine independent so that it can be easily ported, along with the rest of the VISA system, to other distributed memory multiprocessor platforms. This precludes using assembly language primitives to modify the program counter and stack pointer.

Our multithreading design, adapted from an idea from Rob Pike at AT&T Laboratories [Rob92], is based on the standard Unix system calls *setjmp*, which saves the state of the current computation, and *longjmp*, which restores a previously saved state. Each parallel task received by a worker processor is divided into *mt* threads, where *mt* is called the

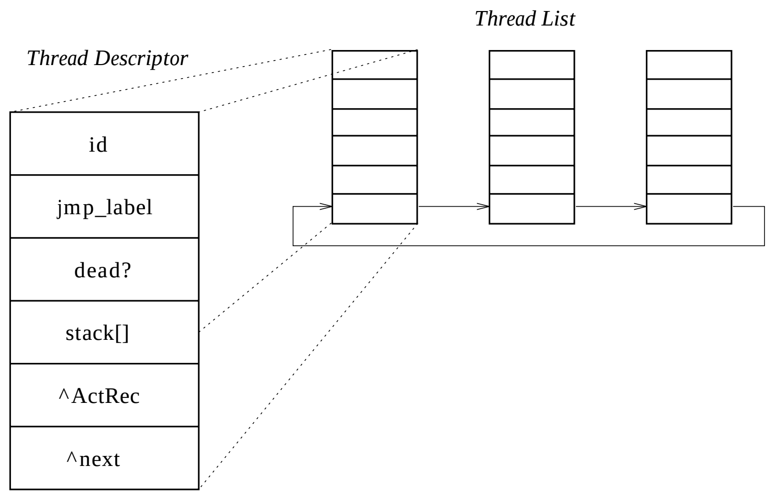

Figure 4.3: Thread descriptor data structure

degree of multithreading. The parallel tasks, and consequently the threads, are guaranteed by the Sisal compiler to be completely independent, thereby allowing arbitrary parallel execution. All threads are executed sequentially by the worker processor, and scheduled in round-robin fashion. One *ThreadDescriptor* (Figure 4.3) is allocated for each of the *mt* threads, and linked together to form a circular list. The fields in the *ThreadDescriptor* structure have the following functions:

- **id** stores an identifier, unique among the current list of threads, and used for matching messages and threads.
- **jmp_label** stores the thread context that *longjmp* will use for starting a new thread, and is initialized by the *setjmp* command.
- **dead?** is a flag that indicates whether or not this thread has terminated and ready to be removed from the list.
- **stack[]** is a pointer to the stack on which the thread will be running. If the threads were to execute from the normal system stack, then the entire stack frame would need to be saved and restored for each context switch, since the next thread would destroy its contents. However, by modifying the **jmp_label** so that the stack for each thread is located in heap space, only the processor registers need be saved and restored for context switches. **stack[]** reserves the heap space necessary for this stack (64K bytes for the current nCUBE/2 implementation).
- **ActRec** is a pointer to the conventional thread representation of the runtime system, the Activation Record (ActRec) structure. This is done to satisfy the goal of keeping the current runtime thread structure intact.
- **next** is a pointer to the next *ThreadDescriptor* structure in the circular list.

Notice that our threads contain a number of loop bodies and consequently are still relatively large. Since the threads are executed sequentially, the amount of parallelism we exploit *per processor* is equal to *mt*, the degree of multithreading. Our threads allow for only one outstanding remote request to simplify the synchronization process. We favor efficient sequential execution above exploiting maximal parallelism.

Multithreaded execution proceeds as follows. Before starting the first thread, a return address is saved using *setjmp*, so that after all the threads have completed, there is a place to return to. The scheduler then selects a thread for execution, which continues until it either completes execution or begins a remote memory reference, implemented as a *split-phased transaction.* In the first phase, the thread sends a *visa_request* to the target processor that contains the desired value, then invokes the scheduler to start another thread. The scheduler selects the next thread from the circular list (if one exists) and begins its execution. The target processor will process the visa_request and send back an associated *visa_reply*, containing the requested remote value. When a visa_reply message arrives during the execution of a thread, its message handler examines the *id* field of the message and, if the *id* does not match the id of the currently executing thread, stores the message in the storage buffer for the proper thread. A *presence bit*, indicating the arrival of the message, is also set.

When a thread is re-enabled by the scheduler from having sent a visa_request, it first examines the presence bit of its storage buffer to see if the visa_reply message arrived and was stored by another thread. If so, this is termed a *hit*, and the message is removed from the buffer, the presence bit is reset, and the value of the remote reference requested earlier is extracted from the message. This completes the second phase of the split-phased transaction.

If the message is not found in the buffer storage, the thread will wait for the message. Since threads are executed in a round-robin fashion, and the message start-up time for the nCUBE/2 (160 μs) is so high compared to the message transport time (2.6 μs per byte) [vECGS92], we assume that messages are received in order. Therefore, waiting for the outstanding message after the first thread switch will minimize the thread switching overhead without much loss of parallelism. This is clearly an architecture-dependent decision, and would have to be re-evaluated for a distributed system with different message passing timing characteristics. We now re-examine our design goals in the context of our design choices.

- *Minimal synchronization overhead for context switching.* Each thread carries its own context in the jmp_label, aside from its runtime stack, which is allocated out of heap space. For the nCUBE/2, the processor state, and thus the size of the jmp_label, is 24 registers. The average time required by the scheduler to save the current context and restore another is 90 μs. Therefore twice this time (180 μs) is required to switch to another thread and then back, and must be less than the minimal round-trip message time for multithreading to be profitable. Since the minimal round-trip message time for the nCUBE/2 is 340 μs [vECGS92], this design meets this primary criterion. However, as we shall see in Chapter 7, this is not the only overhead that multithreading incurs and must cover to be beneficial.

- *Minimal impact on the current activation record structure.* As the circular Thread-Descriptor structure is an extension of the activation record, the remainder of the runtime data structures and routines remains largely unchanged.

- *General purpose.* By manipulating the program counter and stack pointer using standard C routines, this design does not take into account any knowledge of the thread behavior. As long as the **stack**[] space is large enough to hold the stack frames of all called routines, this method will work for general thread code.

- *Machine independence.* All of the routines are written in standard C, and provided that a machine correctly supports the *setjmp* and *longjmp* routines, then this method will work without modification on another machine.

In our design, a loop-slice is divided into *mt* local thread descriptors with adjusted loop index values. While this approach allows the integration of multithreading with minimal impact on the compiler and runtime system, it creates extra input parameter references since each thread must now get the necessary input parameters. By replicating the parameter structure in the VISA system, the extra parameter fetches are local VISA references, but they are still extra references and represent additional overhead for multithreading.

4.5.2 Quantitative Cost Analysis of Multithreading

Multithreading incurs a *startup cost*, C_{start}, averaging 550 μs for creating the circular thread queue and message storage buffers. Fetching the extra thread input-parameters results in $C_{param} = 10\mu s$ per parameter.

If a shared array access *read A[i]* is local, A[i] is fetched without further ado and the thread continues. This happens most of the time in programs with high locality and makes our threads much larger than threads that switch at any memory reference. If the access is remote, a read request package is created and sent off. The source processor spends an average of 200 μs to form the package and initiate the send, and this needs to be done whether multithreading or not. The round trip time for the remote read involves 170 μs message startup time and flight time through the net, 350 μs for the target processor to accept the message, fetch the data, form a reply package and initiate the send back, and 170 μs message startup time and flight time through the network back to the source processor. Therefore, the source processor has $C_{round} = 690\mu s$ to do other work. In this time, the source processor needs to do a context switch to another thread, which involves a longjmp and am setjmp, at a total cost of 90 μs. The difference $B_{switch} = 690 - 90 = 600\mu s$ is what is gained from multithreading, but *only if there is useful work to do in the other thread.* When this is the case, this is called a *successful context switch.*

Multithreading profits if the total gain caused by successful context switches is larger than the initial costs of setting up the circular thread descriptor structure and fetching the extra input parameters. If we call H the number of successful context switches, and P the number of remote parameters needed by each thread, we get the following criterion for multithreading being effective:

$$\Delta Time \approx (H * B_{switch}) - \{C_{start} + (MT - 1) * (P * C_{param})\} > 0 \qquad (4.1)$$

For the above values of B_{switch}, C_{start}, and C_{param}, this implies:

$$H > ((MT - 1) * P + 55)/60 \qquad (4.2)$$

The above inequality must hold for every processor involved, as barrier synchronization at the end of a loop causes the slowest process to govern the computation time.

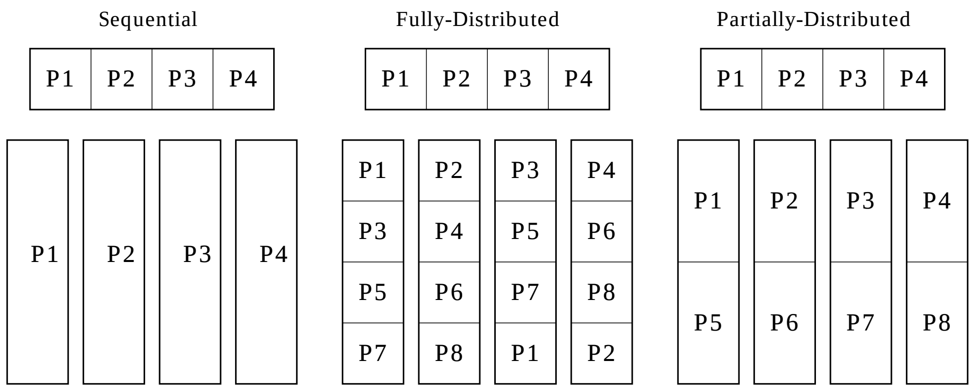

Figure 4.4: Distribution options for nested parallel loops

An implicit form of latency hiding occurs in both the multithreading and non-multithreading cases, when a processor can handle an interrupt while being blocked. This can occur frequently, especially in the non-multithreading case, and amounts to a $350\mu s$ gain.

In Section 7.6, we will evaluate several example programs using this equation of cost, and compare it with the actual timing results obtained from applying multithreaded execution.

4.6 Control Parameters for Parallel Loops

Flat, or single-level, parallel loops offer few choices for parallelization and distribution: if the loop is parallelized, then it is typically distributed among all participating nodes, using either the single-level distribution or multi-level distribution strategy. Nested parallel loops, on the other hand, provide a wide range of distribution options, such as distributing the outer loop for parallel execution and executing the inner loop sequentially (called *sequential inner loop*), distributing both outer and inner loops for parallel execution (called *fully-distributed inner loop*), and fully distributing the outer loop while only partially distributing the inner loop (called *partially-distributed inner loop*). These nested loop distribution schemes are depicted in Figure 4.4, where the outer loop is fully-distributed over the first four processors, and the inner loop is distributed over some set of the 8 available processors.

The first distribution scheme (*sequential inner loop*) minimizes the overhead of the inner loop by executing the loop sequentially, but limits the amount of available parallelism to the size of the outer loop. Any machine parallelism in excess of the outer loop parallelism would be wasted. The second distribution scheme (*fully-distributed inner loop*) allows us to exploit the combined parallelism of both outer and inner loops, and instead of wasting machine parallelism, this strategy often over-saturates the machine resources, causing unnecessary overheads. Therefore, the third scheme (*partially-distributed inner loop*) was created as a hybrid of the first two. When the outer loop parallelism (n) is larger than the number of processors (p), the inner loop is run sequentially, but when p exceeds n, we will distribute the inner loop in a controlled fashion for parallel execution. For example, suppose we are executing a nested parallel loop with 4 outer loop iterations ($n = 4$) and 4 inner loop iterations ($m = 4$) on 8 processors ($p = 8$). Since the outer loop parallelism will not cover the machine parallelism, we will distribute the outer loop to 4 of the processors (P_0, P_1, P_2, P_3) and each of these processors, P_i, will distribute the inner loop to P_i and

Distribution Scheme	*blocksize* Outer	*blocksize* Inner	*start node* Outer	*start node* Inner	*stride* Outer	*stride* Inner
Sequential Inner Loop	n/p	m	P_{id}	P_{id}	1	1
Fully-Distributed Inner Loop	n/p	m/p	P_{id}	P_{id}	1	1
Partially-Distributed Inner Loop	1	mn/p	P_{id}	P_{id}	p/n	1

Table 4.1: Control parameter settings for nested loop distribution schemes

$P_{i+(p/m)}$. This distribution scheme will utilize all the processors while minimizing the overhead of distributing the inner loop.

Since the decision of which distribution scheme to use can only be made after runtime parameters have been established (i.e. size of loops, number of processors), we have equipped our runtime system with the ability to handle the three nested loop distribution schemes discussed, and the flexibility to create others. The actual decision can then be made by a compiler using runtime profiles or other analysis, or by the programmer. This is accomplished by associating three control parameters to each parallel loop: *blocksize*, *start node*, and *stride*. The *blocksize* specifies the thickness of the loop slices, i.e. the number of contiguous loop iterations for each slice, thereby creating $n/blocksize$ slices of the loop to be distributed. By default, *blocksize* is set to n/p so that p slices are created, one per processor, minimizing overhead. The slices are then distributed among the processors, starting with *start node* and continuing at an increment of *stride* until all slices have been distributed. Slices are wrapped in modulo fashion when their number exceeds the number of processors divided by the stride ($n/blocksize > p/stride$). Table 4.1 gives the parameter settings for the three nested loop distribution schemes discussed earlier, where n is the number of outer loop iterations, m is the number of inner loop iterations, p is the number of processors, and P_{id} is the designator for each processor $id = 0 \ldots p-1$.

4.7 Summary

We have introduced the concept of parallel work in Sisal, and the master-slave model of parallel execution used in exploiting this parallelism. We have focused on the distributed task design, where we have augmented with single-level model of distribution and execution with two additional models, *multi-level distribution* and *multithreading*, which allow for a more flexible hierarchy of parallelism. These additional models are necessary to efficiently execute a wide variety of parallel loops on a wide variety of processor configurations. We have outlined the design of our software multithreading scheme, and provided a quantitative cost analysis of its performance. We have also introduced a system of *control parameters* that is used to define how a parallel loop is to be divided and distributed among the participating nodes, which is essential for supporting the various distribution options present in nested parallel loops. As we shall see in Chapter 5, these control parameters correspond to the control parameters used in partitioning and distributing the user data structures. This is a deliberate attempt to align task distribution with data distribution, which is necessary to minimize remote references.

Chapter 5

MEMORY MANAGEMENT

To avoid slow performance, Apple suggests that the amount of virtual memory you select be less than the system RAM.
– INFOWORLD

Central to the current Sisal compiler is the assumption of shared memory, which is required for both system and user data structures. In Chapter 4, we outlined the design of a task management system that eliminates the need for global system data structures by employing a distributed, rather than centralized, task distribution approach. In this chapter, we describe the design of runtime support for a single addressing space and general data decompositions used to manage global user data structures.

Other distributed memory languages [HKT92, KM91, ZBG86] use a compiler-based approach to providing a single addressing space. However, there are several problems with this approach that have kept them from wide-spread use, and motivate our decision to employ a runtime-based design:

- Data distribution and automatic parallelization of a sequentially written program requires extensive dependence analysis that can be hampered by common imperative programming phenomena such as aliasing. Also, symbolic subscript terms with unknown values, coupled subscripts, and nonzero or nonunity coefficients of loop indices often make dependence analysis impossible for even the most sophisticated parallelizing compilers [SLY90]. When static dependence analysis does fail, the alternative is to employ a costly runtime-based scheme for generating the necessary communication sets by actually pre-executing the parallel loop to determine the current access pattern [SCMB90].

- Due to the complexity of these compilers and the difficulties in porting them to new machines, their availability is limited to only a few of the currently available distributed memory multiprocessor systems. Also, the way in which data distribution is controlled and the amount of programmer interaction varies widely from system to system, which can make porting an application from one distributed memory system to another a non-trivial task.

- Programmers have long been aware that the language design has a significant impact on how easily an algorithm can be transformed into working code [PB90]. Even the so-called "general purpose" languages are recognized as being suited for certain problem solving approaches. The transformation process is more tedious and error prone when the conceptual models supported by the language relate only peripherally to the problem-solving model of the programmer. Unfortunately, though the compilation ideas for these compilers are applicable for a wide range of languages, almost

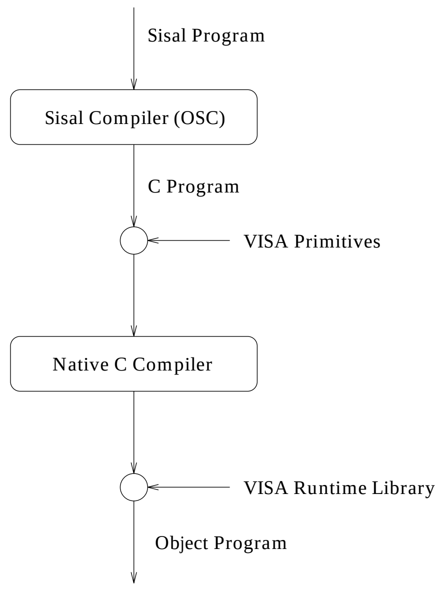

Figure 5.1: Overview of the VISA system

all of these systems offer the same programming language, drastically restricting the choice of languages for distributed memory machines.

- The single addressing space is realized by dividing the global data structure into disjoint portions, and assigning ownership of these portions among the participating processors. Using the "owner computes" rule, the defining computation for each element is carried out by the processor which own that element. This has the effect of fusing the task distribution and data distribution phases, which can result in load or memory imbalance problems when the goals of task and data distribution are opposed. Also, the single addressing space is limited to parallel loops, and thus any data to be shared outside of such loops, such as global argument structures, must still be communicated using message passing primitives.

Considering these difficulties of distributed memory compilers, we have designed a runtime-based approach to providing a single addressing space for the Sisal compiler, rather than modifying the compiler itself to deal with distributed data structures. Address translation is done on-the-fly, which alleviates the need for extensive dependence analysis in order to determine which data will be accessed, and when. Language independence is achieved by designing the system as a stand-alone runtime library that can be accessed by other compilers (see Figure 5.1). Finally, any data structure may be placed into the single addressing space, including shared argument structures, which can be replicated, and atomic variables, which exist in only one node.

5.1 Design Goals

Our goal in designing the VISA runtime system is to eliminate the burden of explicit data management from the programmer, while at the same time providing explicit control over the general distribution of global data structures. Towards this end, VISA provides the following services:

- *Single Addressing Space.* One of the primary difficulties in programming a distributed memory multiprocessor is the lack of a single addressing space for user data structures, such as arrays. This results in encumbering the compiler, or worse yet, the programmer, with the task of distributing data structures and inserting the proper code to fetch and store non-local references. Therefore VISA provides a single addressing space, and a set of associated functions that operate on that space, so that the programmer is given a familiar shared memory model of computation.

- *Mapping Functions.* In association with the single addressing space, VISA provides a method for the compiler or programmer to specify "how" the data is to be distributed across the memories for improved efficiency. The idea is to distribute the data structures in tight accordance with the distribution of parallel tasks, so that *local references are maximized.* New analysis techniques [OH92, HKT92, GB92] can yield the optimal distribution in restricted cases, and in these cases compilers could either insert the communication code directly or pass the distribution information to a runtime system like VISA in the form of a mapping function directive.

- *Split-Phased Transactions.* In Chapter 4, we introduced the design of multithreaded task management, which relies on the ability to perform remote references as *split-phased transactions*, where the request and reply phase are decoupled to allow for thread switching between the two phases. VISA supports split-phased transactions in support of the multithreaded task execution model.

The interaction of the VISA system with the other language components is depicted in Figure 5.1. The compiler augments a parallel program with VISA primitives for allocating and accessing the data structures to be kept in the single addressing space. Any variables not placed in the VISA space are unaffected by the system. The augmented program is then compiled using the native C compiler of choice, and linked with the VISA library to create the object program, which can then be executed on a distributed memory multiprocessor.

5.2 Message Passing Abstraction

All message passing required for accessing remote values is handled by the VISA system through the use of a *message passing abstraction*, supporting both synchronous (blocking) and asynchronous (non-blocking) operations. Since these operations are provided by most host operating systems for distributed memory multiprocessors, VISA can be easily ported to other distributed memory multiprocessors by modifying the message passing abstraction to make the proper native calls.

Specifically, the abstraction supports the following operations:

- *WriteMsg*, a non-blocking send abstraction used for point-to-point communication.

- *Broadcast*, a non-blocking send abstraction used for broadcasting information, such as updates to replicated data structures. If not specifically supported by the underlying system, the Broadcast can be built from the WriteMsg primitive.

- *ReadMsg*, a blocking receive abstraction used for synchronous communication. Selective message screening can be accomplished by specifying a message *key*, which is composed of a message type and sender designator.

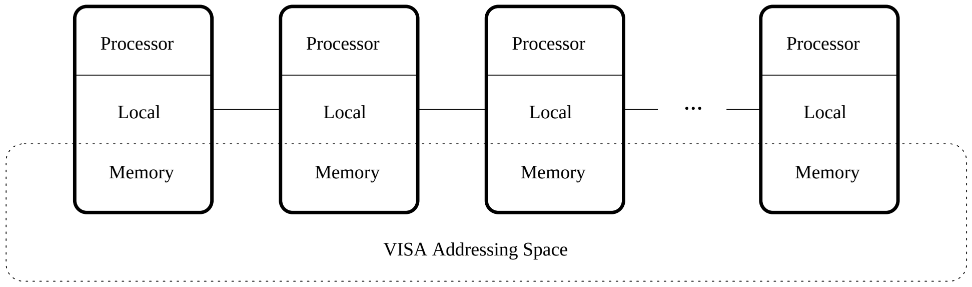

Figure 5.2: The VISA addressing space

- *MsgInterruptHandler*, an asynchronous receive abstraction using interrupts. Asynchronous message reception requires polling at some level to determine when a message arrives and take appropriate action. Most systems, including the nCUBE/2, provide hardware polling for incoming messages, resulting in a hardware trap that is caught by the operating system, and then passed into the user-level in the form of an interrupt. The interrupt causes a VISA message interrupt handler to deal with the message. If the interrupt handler is allowed to be invoked at any arbitrary time during the computation, it cannot modify the global state of the computation. Therefore, either the interrupt handler must be selectively disabled during the times when global data structures are accessed, or it must be prevented from modifying global data structures. The former option requires the placement of expensive system calls for enabling and disabling interrupts around all global data structure accesses, which can be costly and error-prone. Therefore, the VISA system employs the latter option: Any message requiring a global modification is enqueued onto a message list for handling outside of the scope of the interrupt handler.

5.3 Data Distribution

As depicted in Figure 5.2, the VISA address space is allocated in part of the local memory of each participating node. This creates two types of addressing space for each participating node in the system: a shared *virtual* addressing space that spans all of the nodes, and a *local* address space for data visible only to the local node. Each data structure allocated to the VISA space receives a contiguous set of *virtual* addresses that are shared among the nodes and mapped onto *physical* addresses from each node.

Data distribution (or *data decomposition*) determines how the physical storage for a global data structure is to be divided among the participating nodes. The goal is to divide the data structure among the nodes so as to *minimize the number of remote references* caused by the distribution. This implies that the distribution of data must be tied to the access pattern of the parallel computation, and therefore data distribution needs to be flexible to support a wide variety of access patterns. For VISA, data distribution is accomplished by dividing a data structure into a set of blocks, where each block contains *blocksize* elements. The blocks are then allocated to the physical memories of the nodes in round-robin fashion until all of the blocks have been distributed.

To facilitate a variety of distribution schemes, we assign a set of control parameters to each data structure that define the *blocksize* of each block, the *start node* to which the first block is assigned, and the processor *stride* at which the blocks are distributed. These *data* control parameters correspond to the *task* control parameters that are used to distribute

Mapping Function	*blocksize*	*start node*	*stride*	*replicate*
`scalar_map`	n	map_arg	1	No
`replicate_map`	n	P_{id}	1	Yes
`block_map`	n/p	map_arg	1	No
`variable_block_map`	map_arg	P_0	1	No
`interleave_map`	1	map_arg	1	No

Table 5.1: Control parameter settings for various 1D mapping functions

parallel tasks (see Table4.1), thus providing a unified method for tying task distribution to data distribution. A fourth control parameter for data structures specifies whether or not a data structure is to be replicated. Any data structure, from a single variable to an entire array, can be replicated among the nodes in the VISA system. Replication is accomplished by allocating enough local storage from each node to accommodate the entire structure, and broadcasting all writes to the data structure. Rather than implementing an expensive coherence protocol, VISA assumes that the replicated data structures are controlled by the compiler, where explicit synchronization can be provided which minimizes the synchronization required while still maintaining a coherent system.

5.3.1 One-Dimensional Mapping Functions

Table 5.1 details the parameter settings for several one-dimensional mapping functions, where the *map_arg* is passed in from the allocation routine, typically specifying the starting node. Most variables and structures are allocated using either the `scalar_map` or the `replicate_map`, depending on the nature of the variable. For example, a structure containing arguments for a parallel slice routine would be replicated to eliminate the remote references required by each of the nodes executing the parallel slice, whereas a global counter would be allocated using the `scalar_map` to ensure consistency. Data arrays are typically allocated using the `block_map`, which provides an even distribution of the data among the nodes in chunks that are often exploited by the contiguous loop structure of the Sisal tasks. Arrays can also be replicated, and as we will see with two dimensional data structures, the pointer array is replicated to eliminate the need for two remote references when accessing an array element.

5.3.2 Multi-Dimensional Mapping Functions

The current Sisal compiler represents multi-dimensional data structures as structures containing sub-structures. For example, a two-dimensional array is represented as an array of *pointers*, where each element points to the location of a one-dimensional data structure (see Figure 5.3). This is done to conform to the way in which multi-dimensional arrays are represented in both Sisal and C[1]. Mapping functions for multi-dimensional arrays must therefore consider both the pointer arrays as well as the data arrays. Pointer arrays are

[1]True multi-dimensional arrays in C are possible only if the array bounds are given at compile time

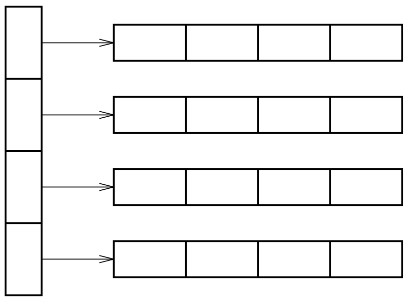

Figure 5.3: Two-dimensional arrays in Sisal

Mapping Function	*blocksize*	*start node*	*stride*	*replicate*
`matrix_row_map`	n	map_arg	1	No
`matrix_col_map`	1	map_arg	1	Yes
`matrix_block_map`	$rbs * n/p$	map_arg	rbs	No

Table 5.2: Control parameter settings for various 2D mapping functions

replicated to guarantee that accessing any element of a matrix will require at most one remote reference.

Assuming that all pointer arrays are allocated using the `replicate_map`, Table 5.2 details how the control parameters are established for each of the data arrays in a two-dimensional matrix. The *map_arg* for these mapping functions represents the starting node, and is typically some function of i, corresponding to the i^{th} row of the matrix. For `matrix_row_map`, the map_arg is typically set to $i/(n/p)$, where n is the number of rows and p is the number of processors. When the number of rows is equal to the number of processors ($n/p = 1$), the i^{th} row is placed on the i^{th} processor, and when the number of rows exceeds the number of processors, each processor gets a group of n/p contiguous rows. However, if an interleaved row allocation is desired, the map_arg can be set to $imodp$ instead. For `matrix_block_map`, the map_arg is typically set to $i/(n/rbs)$, where rbs is a control parameter for the `matrix_block_map` function, and is fixed for a given number of processors to allow for the proper layout of the blocks. Figure 5.4 depicts the distributions for an 8x8 matrix on 4 processors using the *matrix_row_map* with contiguous rows and the *matrix_block_map* with $rbs = 2$.

It is possible to create many different mapping functions, given the ability to modify the data control parameters. This *general* approach to data distribution is necessary to accommodate the various access patterns that applications exhibit, and VISA allows the user to add to the set of available mapping functions so that customized decompositions are possible.

5.3.3 Specification of Mapping Functions

Given a set of mapping functions for controlling the distribution of data structures, there are various methods currently employed by other systems for selecting a mapping function for a given data structure:

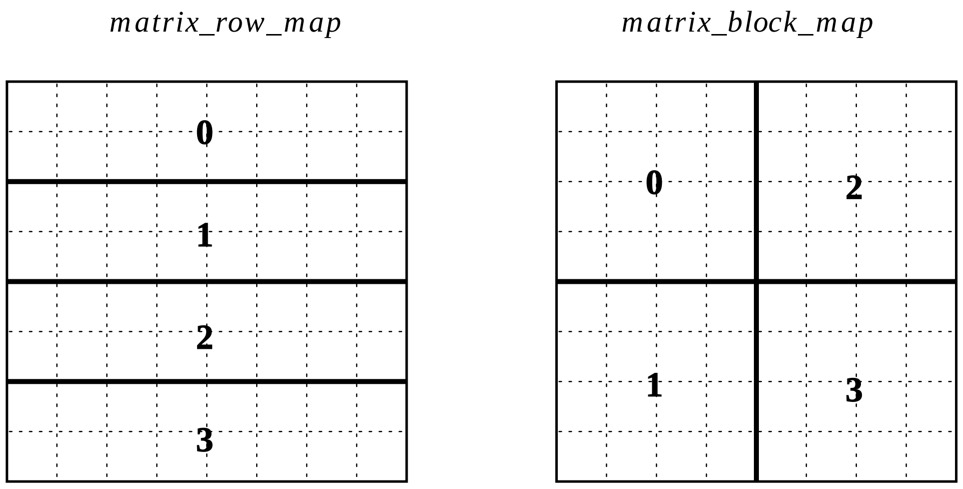

Figure 5.4: `matrix_row_map` and `matrix_block_map` functions

- The compiler controls the distribution of data structures. This is the approach taken by the parallelizing compiler camp [FHK+90, ZBG86]. The basic idea is to distribute the data structures according to some *distribution function*, and then to analyze the array subscripts to determine whether or not, for a particular thread, a given reference is local or remote. If the reference is remote, the the appropriate communication primitives are generated to retrieve the value at runtime. The distribution functions are formalized so that a compiler can make sense of them, and this formalization is equally useful when considering other approaches.

- The compiler controls the distribution with the help of the programmer. This approach is an extension to the compiler controlled approach in that the programmer helps the compiler in identifying the data access patterns by the use of *pragmas*, which are source level compiler directives. Since the programmer may have a better idea as to how the data will be accessed [HKT92], most compilers that perform the data distribution for the programmer will accept these "hints" so that the proper data distribution function can be selected.

- The compiler controls the distribution with the help of run-time profiles [Sar89]. Again, this approach attempts to help the automated distribution process, but rather than have the programmer tell the compiler how the data will be accessed, the compiler simply "watches" several characteristic runs and notes the distribution patterns used for those runs. The compiler then selects a distribution function that will come closest to this observed reference behavior. The advantage this approach has over the pragmas is that the programmer may be unaware of the reference pattern, and thus be unable to help with the distribution. The disadvantage is that if the profiled runs are not characteristic of the actual reference patterns, or if the reference patterns vary with the input data, then this approach may be misleading.

- The programmer controls the distribution explicitly. Since all of the above techniques require intelligent compilers that are not always (or often) available, a common technique for distributing data is for the programmer to explicitly distribute the data and then insert the appropriate communication primitives into the source code, all "by hand." Though this approach requires very little software support (only the message passing interface is needed), the user is required to determine the access patterns and then distribute the data accordingly using explicit message passing primitives. Clearly this contradicts the efforts of raising programming to a higher level of abstraction.

The VISA approach to this complex problem is to provide a comprehensive set of mapping functions that are representative of common scientific data access patterns, and allow for the user to create new mapping functions as needed, such that the mapping function establishes the desired values of the control parameters. The mapping function is then specified upon requesting memory from the single addressing space using the *visa_malloc* function. This allows a compiler that is generating the VISA primitives to invoke *visa_malloc* with the desired mapping function, either obtained from analysis or through user directives. Likewise, a programmer using the VISA primitives directly can select the desired mapping function for each data structure without having to specify the actual message passing details necessary for implementing such a distribution scheme.

5.4 General Address Translation

Address translation is the process of obtaining the physical address of a datum given its virtual address. For a distributed memory multiprocessor, a physical address consists of the tuple *(node, offset)*, where *node* is a node designator and *offset* is the physical address within that node. Since VISA employs a block-based addressing scheme, where the blocksize, starting node, and stride may all vary, it is necessary to store these control parameters, along with other information about each data structure, in a descriptor called a *range_map entry*. The entire VISA space is therefore described by the collection of these entries, called the *range_map table*. The term "range" refers to the fact that, since all data structures are assigned contiguous addresses in both virtual and physical spaces, the range (low, high) is sufficient to represent all of the addresses within a data structure. To ensure local access of the range_map entries, the range_map table is replicated. There is no coherence problem, since each range_map entry is written only once (upon creation by *visa_malloc*).

In addition to the data distribution control parameters, each range_map entry (depicted in Figure 5.5) contains three address ranges for each data structure:

- The *visa_base* represents the range of global virtual (VISA) addresses for this data structure.
- The *local_base* represents the range of local physical addresses of the blocks that are allocated for this data structure.
- The *optimized_base* represents the optimized range of global addresses, as explained in Section 5.5.

After a data structure has been distributed with the *visa_malloc* routine, access requires a translation from the virtual VISA address to the physical address tuple *(node, offset)*, which proceeds as follows:

- The range_map entry for the desired data structure is fetched by the *find_rm()* routine, which is exposed to the compiler so that the range_map entry for a data structure that is to be accessed many times need only be fetched once.

Field	Function
`visa_base`	The range of global virtual addresses
`local_base`	The range of local physical addresses for locally-owned blocks
`optimized_base`	The range of optimized virtual addresses
`nelems`	The number of elements
`size`	The size of each element
`blocksize`	The blocksize (elements per block) used for distribution
`start node`	The node ID on which to begin distributing the blocks
`stride`	The stride at which to distribute the blocks
`replicate`	A boolean to determine if this data structure is replicated
`table_index`	The index into the range_map table for this entry
`next`	A utility pointer

Figure 5.5: Description of a `range_map` entry

- From a *virtual address*, the relative element position within the data structure (`element`) is computed:

  `element = address - low_range`

- Next, the block which contains the desired element is computed:

  `block = element / blocksize`

- Next, the relative offset of the desired element within this block is computed:

  `block_offset = element mod blocksize`

- Next, the node which owns the desired block is computed. If the `replicate` flag is set, then the computed node always equals the local node designator, indicating that each node has a copy of the desired block. Otherwise, we compute:

  `node = (start node + (block * stride)) mod P`

  where `P` is the number of participating nodes.

- If the number of blocks for this data structure exceeds the number of participating nodes, then some (or all) of the nodes will own multiple blocks. Next, the relative block number within the desired node is computed:

  `node_block = block / P`

- Next, the relative offset of the desired element within the desired node is computed:

  `rel_offset = node_block * blocksize + block_offset`

- If the access is local (i.e. `node` is equal to the local node designator) the `rel_offset` is incremented by the local_base from the range_map entry to produce the actual offset in local physical memory, since local_base contains the address of the first byte for this structure:

  `offset = rel_offset + local_base.`

- If the access is remote, a message is sent to the computed node, requesting that the desired datum be fetched and returned. For multithreading support, this is implemented as a *split-phased transaction*, where the first phase involves a sending a `request` to the desired node and the second phase involves waiting for a `reply`. When multithreading is enabled, a thread scheduler is invoked between these two operations to start another parallel thread while the request is being processed. Refer to Section 4.5 for a more detailed description of multithreading and split-phased transactions.

An alternative to this address translation scheme is to have a *fixed* blocksize, start node, and stride for every data structure. For example, consider a virtual addressing space with addresses $0 \ldots 15$ and a fixed blocksize of 2 distributed over 4 nodes as follows:

PE_0	PE_1	PE_2	PE_3
0	2	4	6
1	3	5	7
8	10	12	14
9	11	13	15

The virtual to physical address calculation is then:

- `block = address / fixed_blocksize`
- `block_offset = address mod fixed_blocksize`
- `node_block = block / P`
- `node = block mod P`
- `offset = node_block * fixed_blocksize + block_offset`

If the fixed_blocksize and number of nodes (`P`) are powers of two, the binary representation of the virtual address can be interpreted as consisting of three bit fields:

```
| node_block | node | block_offset |
```

The result is a virtual address from which the physical address tuple can be obtained by an examination of the virtual address bit fields. This type of translation would typically be done in hardware by the memory management unit to provide for very fast address translation. However, since the nCUBE/2 lacks the ability to perform this translation in hardware, we have implemented this fixed_blocksize address translation scheme to compare with the VISA general address translation scheme. Chapter 7 contains the results of this experiment.

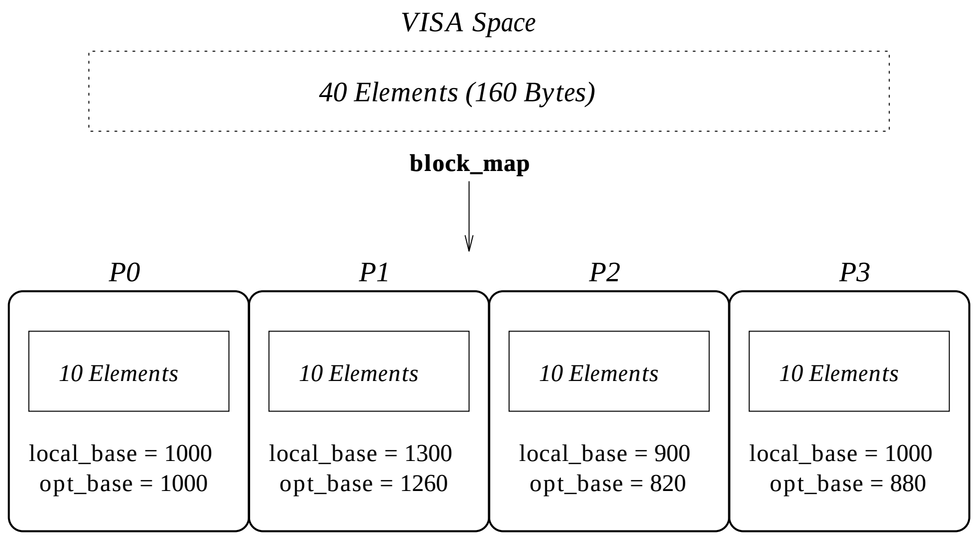

Figure 5.6: Sample VISA data structure with computed optimized_base values

5.5 Optimized Address Translation

One of the first things we noticed about the VISA address translation scheme is that the overhead for translation is minimal when compared with the time required to perform a remote reference, but dominates the time required for a local reference. In addition to exposing the routine which finds a desired range map entry so that range_map entries can be stored locally to avoid repeated searching of the range_map_table, we have designed and implemented an optimization that *eliminates* the need for an address translation when the access is local. We introduce a new function, called *visa_opt*, which re-writes the virtual base address with the structure's *optimized* base address, and establishes a pair of "water mark" registers to hold the low and high values of the range corresponding to the local_base. The optimized_base is the local_base minus the offset necessary to generate a global address that will result in a local access. For example, suppose an array of 40 integers (4 bytes each) is allocated using *block_map* among 4 nodes, as depicted in Figure 5.6, where the local_base values can be different for each node, which is possible since each node manages its local memory independently of the other nodes. Each processor would allocate local storage for *blocksize = 10* elements (40 bytes), and set the local_base accordingly. If, for example, the third node wishes to optimize the base address for this structure, then the optimized value is the local_base minus 20 elements (80 bytes), corresponding to the two blocks of 10 elements each that proceed it in the distribution. Once the base address for a structure has been optimized, any further access to this structure, represented as some offset from the base, will be checked against the low and high water marks. If the computed address falls within the water marks, then the access can proceed without translation, otherwise the address is passed along to the VISA access routines for general address translation and proper remote handling. Special macros are defined to perform the water mark checks, so that the total overhead for a local access has been reduced to the time required for three comparisons.

Chapter 6

SAMPLE PROGRAMS

There never has been, nor will there ever be, any programming language in which it is the least bit difficult to write bad code.
– Lawrence Flon

In this chapter we introduce the set of sample programs that will be used to conduct a variety of experiments with our runtime system, as detailed in Chapter 7. The programs were selected to highlight a particular aspect of the runtime system design, such as loop distribution, data distribution, or multithreading. We present the mathematics and algorithm of each program, the corresponding Sisal code, and the characteristics of the problem, leading to an explanation as to why it was included in the suite.

We note that the programs in our suite are relatively simple programs, often termed "kernels," and not large, involved "benchmarks." The decision to restrict our attention to simple problems and exclude large benchmarks is twofold:

1. We require the programs to be small enough to reason about their individual behavior, in terms of parallelism and data access patterns. This is necessary to properly analyze the effects of our various designs, without which we are left with just numbers.

2. The current Sisal compiler (*OSC V12.0*) lacks both the ability to generate the VISA primitives and the ability to treat a rectangular array as a single data structure, rather than a collection of lower-dimensional data structures. Therefore, implementing large benchmark programs would require a substantial amount of work to insert the necessary VISA primitives and, assuming the benchmark employed multi-dimensional arrays, would result in poor performance. This teaches us very little about the system and the performance of these programs.

6.1 Purdue #1

6.1.1 Problem Description and Algorithm

Purdue Parallel Benchmark #1 [Ric85] approximates the value of the integral of $f(x)$ in the interval [a,b] using the trapezoidal rule:

$$T_N = h * (\frac{f(a) + f(b)}{2} + \sum_{i=1}^{N-1} f(a + i * h))$$

where N is the number of intervals in for the estimate and, $h = (b - a)/N$. Increasing N usually improves the accuracy. For our implementation, we compute $\int_0^\pi \sin(x)$, where sin(x) is computed using a Taylor series of fourteen terms ($\frac{x^1}{1!} - \frac{x^3}{3!} + \ldots - \frac{x^{27}}{27!}$), resulting in 55 floating-point operations per invocation and matching the precision of the nCUBE/2 system sin function.

6.1.2 Sisal Code for Purdue #1

```
% purdue1.sis

define main

global sin (x: double_real returns double_real)

function f (x: double_real returns double_real)
    let
        a := x * x;
        b := x * a / double_real(6.0);
        c := b * a / double_real(20.0);
        d := c * a / double_real(42.0);
        e := d * a / double_real(72.0);
        f := e * a / double_real(110.0);
        g := f * a / double_real(156.0);
        h := g * a / double_real(210.0);
        i := h * a / double_real(272.0);
        j := i * a / double_real(342.0);
        k := j * a / double_real(420.0);
        l := k * a / double_real(506.0);
        m := l * a / double_real(600.0);
        n := m * a / double_real(702.0);
    in
        x - b + c - d + e - f + g - h + i - j + k - l + m - n
    end let
end function % f



function main (a,b: double_real; n: integer returns double_real)
    let
        h := (b-a)/double_real(n);
        ans := for i in 1,(n-1)
            returns value of sum f(a+double_real(i)*h)
        end for
    in
        h * (ans + (double_real(0.5) * f(a)) + (double_real(0.5) * f(b)))
    end let
end function % main
```

6.1.3 Program Characteristics

Purdue #1 contains a large, flat, parallel loop in which

$$\sum_{i=1}^{N-1} f(a + i * h)$$

is computed. Since there are no arrays, this problem isolates the effectiveness of loop distribution.

6.2 Purdue #2

6.2.1 Problem Description and Algorithm

Purdue Parallel Benchmark #2 [Ric85] computes e^* by:

$$e^* = \sum_{i=1}^{n} \prod_{j=1}^{m} (1 + e^{(-|i-j|)})$$

6.2.2 Sisal Code for Purdue #2

```
% purdue2.sis

define main

global etothe (a: double_real returns double_real)

function abs (a: integer returns integer)
    if (a < 0) then -a else a end if
end function % abs

function main (n,m: integer returns double_real)
    for i in 1,n
        p := for j in 1,m
            returns value of product (double_real(1.0) +
                etothe (-double_real(1.0) * double_real (abs(i - j))))
        end for
        returns value of sum p
    end for
end function % main
```

6.2.3 Program Characteristics

Purdue #2 computes a sum-of-products, allowing us to explore the different distribution options for nested parallel loops. With no arrays in this problem, we can isolate the effectiveness of our various nested task distribution techniques.

6.3 Lawrence Livermore Loop #1

6.3.1 Problem Description and Algorithm

This program [Feo87] creates an array X from input arrays Y and Z, and constants Q, R, and T, where X_i is defined as:

$$X_i = Q + (Y_i * (R * Z_{i+10} + T * Z_{i+11}))$$

6.3.2 Sisal Code for Lawrence Livermore Loop #1

```
% lll1.sis

define main

type OneD = array[double_real]

function loop1 (n: integer; Q,R,T: double_real; Y,Z: OneD returns OneD)
    for K in 1,n
        X := Q + (Y[K] * (R * Z[K+10] + T * Z[K+11]))
    returns array of X
    end for
end function % loop1

function main (n: integer; Q,R,T: double_real returns OneD)
    let
        Y := array_fill (1,n,double_real(2.0));
        Z := array_fill (1,n+11,double_real(2.0));
    in
        loop1 (n, Q, R, T, Y, Z)
    end let
end function % main
```

6.3.3 Program Characteristics

This program creates two initial arrays of different sizes (Y and Z), which are then used to create the resulting array (X), which is the same size as the smaller of the input arrays (Y). This allows us to manipulate the blocksize for the arrays and measure the difference in performance when various blocksizes are applied. Also, with a high degree of locality, this program highlights the overheads of multithreading when very few remote references are present.

6.4 Lawrence Livermore Loop #7

6.4.1 Problem Description and Algorithm

This program [Feo87] creates an array A from an input array B and constants R, T, C_1, and C_2, where A_i is defined as:

$$\begin{aligned} A_i &= B_i + R * C_1 + R^2 * C_2 + T * B_{i+3} + T * R * B_{i+2} + T * R^2 * B_{i+1} + \\ &\quad T^2 * B_{i+6} + T^2 * R * B_{i+5} + T^2 * R^2 * B_{i+4} \end{aligned}$$

6.4.2 Sisal Code for Lawrence Livermore Loop #7

```
% ll17.sis

define main

type OneD = array[double_real]

function loop7 (n: integer; R,T: double_real; U,Y,Z: OneD returns OneD)
    for k in 1,n
    returns array of U[k] + R * (Z[k] + R * Y[k]) +
        T * (U[k+3] + R * (U[k+2] + R * U[k+1]) +
        T * (U[k+6] + R * (U[k+5] + R * U[k+4])))
    end for
end function % loop7

function main (n: integer; R,T: double_real returns OneD)
    let
        U := array_fill (1,n+6,double_real(2.0));
        Y := array_fill (1,n,double_real(3.0));
        Z := array_fill (1,n,double_real(4.0));
    in
        loop7 (n, R, T, U, Y, Z)
    end let
end function % main
```

6.4.3 Program Characteristics

With very little task management required, this problem highlights the different memory management and multithreading techniques. In particular, the different array sizes presents the possibility of controlling the distribution of these various arrays for optimal performance.

6.5 Successive Over-Relaxation (SOR)

6.5.1 Problem Description and Algorithm

Successive over-relaxation (SOR) performs a “smoothing” operation on an array by iterating over the array and computing each new A_i element as follows:

$$A_i = \begin{cases} A'_i & \text{if } i = 1 \text{ or } i = n \\ (A'_{i-1} + A'_i + A'_{i+1})/3.0 & \text{otherwise} \end{cases}$$

where A represents the current iteration and A' represents the previous iteration.

6.5.2 Sisal Code for SOR

```
% sor.sis

define main

type OneD = array[double_real]

function sor (n,k: integer; A: OneD returns OneD)
    for initial
        V := A; loop := 0;
    repeat
        loop := old loop + 1;
        V := array[1:1.0e0]
            ||
            for i in 2,n-1
                el := (old V[i-1] + old V[i] + old V[i+1]) / 3.0e0
            returns array of el
            end for
            ||
            array[1:double_real(n)]
    until loop=k
    returns value of V
    end for
end function % sor

function main (n,k: integer returns OneD)
    let
        A := for i in 1,n
            el := if mod (i,2) = 0 then 1.0e0 else double_real(n) end if
        returns array of el
        end for
    in
        sor (n, k, A)
    end let
end function % main
```

6.5.3 Program Characteristics

The access pattern for SOR is fixed over all of the iterations. The outer iterative loop in this program provides a method of controlling the amount of synchronization required, thus highlighting the different task management options.

The Sisal compiler allocates the initial array and two additional arrays that are used as "swap" arrays, where one represents the current iteration and the other represents the previous iteration. This optimization, which is a form of *build-in-place* [Can89], removes the need to create a new array for each loop iteration as the functional semantics imply. Therefore, the two swap arrays need to be distributed in accordance with the input array to maintain the minimal number of remote references.

6.6 Parallel Prefix

6.6.1 Problem Description and Algorithm

A parallel prefix computation is defined in terms of a binary, associative operator $\otimes$ [CLR91]. The computation takes as input the sequence $\langle x_1, x_2, \ldots, x_n \rangle$ and produces as output the sequence $\langle y_1, y_2, \ldots, y_n \rangle$ such that $y_1 = x_1$ and

$$\begin{aligned} y_k &= y_{k-1} \otimes x_k \\ &= x_1 \otimes x_2 \otimes \cdots \otimes x_k \end{aligned}$$

for $k = 2, 3, \ldots, n$. Thus y_k is obtained by applying the operator $\otimes$ in the first k elements in the sequence of x_k, hence the term "prefix". For our prefix program, we have selected addition as the binary, associative operator, and thus we are computing

$$y_k = \sum_{i=1}^{k} x_i$$

6.6.2 Sisal Code for Parallel Prefix

```
% pprefix.sis

define main

type OneD = array[double_real]

function prefix (n: integer A: OneD returns OneD)
    for initial
        v := A; d := 1;
    repeat
        d := 2 * old d;
        v := for i in 1,n
            el := if i+ old d > n then old v[i]
                else old v[i] + old v[i+old d]
            end if
        returns array of el
        end for
    until d >= n
    returns value of v
    end for
end function % prefix

function main (n: integer returns OneD)
    let
        A := array_fill (1,n,1.0)
    in
        prefix (n, A)
    end let
end function % main
```

6.6.3 Program Characteristics

Parallel prefix performs a logarithmic number of iterations over the initial array in which the access pattern changes for each iteration. Specifically, the computation of y_i in the current iteration makes reference to the values y_i and y_{i+d} in the previous iteration, where d is a "distance" variable that starts at 1 and increases by a factor of two for each subsequent iteration. Thus, when d is greater than the distribution blocksize of the initial array, references to y_{i+d} are remote for all y_i.

Parallel prefix represents a problem class in which a large amount of remote references are incurred despite a good initial distribution of the data. Therefore, as latency avoidance is hampered by the variable access pattern, latency tolerance becomes more critical, and this problem highlights the ability of multithreading to effectively tolerate unavoidable remote reference latencies.

As with SOR, the Sisal compiler allocates a set of swap arrays instead of generating a separate array for each iteration. Thus the swap arrays are distributed in accordance with the input array.

6.7 Fast Fourier Transform (FFT)

6.7.1 Problem Description and Algorithm

A **Discrete Fourier Transform** is used to evaluate a polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

of degree n at the roots of unity $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$. Assuming that A is given in coefficient form: $a = (a_0, a_1, \ldots, a_{n-1})$, the result y_k, for $k = 0, 1, \ldots, n-1$, is defined as

$$\begin{aligned} y_k &= A(\omega_n^k) \\ &= \sum_{j=0}^{n-1} a_j \omega_n^{kj} \end{aligned}$$

The y vector is the Discrete Fourier Transform of the coefficient vector, a, written as $y = DFT_n(a)$.

The Fast Fourier Transform (FFT) takes advantage of the special properties of the complex roots of unity to compute the Discrete Fourier Transform of a coefficient vector in time $\Theta(n \; lg \; n)$ as opposed to the $\Theta(n^2)$ time of the straightforward DFT. The FFT employs a divide-and-conquer strategy, using the even-index and odd-index coefficients of $A(x)$ separately to define two new degree $n/2$ polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$:

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1}, \\ A^{[1]}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1}. \end{aligned}$$

The problem of evaluating $A(x)$ at $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ reduces to evaluating the degree $n/2$ polynomials $A^{[0]}$ and $A^{[1]}$ at the points

$$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2,$$

and then combining the results according to the equation

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2).$$

This method recursively divides the n-element DFT_n computation into two $n/2$-element $DFT_{n/2}$ computations to compute the DFT of an n-element vector a, where n is a power of 2.

For our implementation of this algorithm, we select an iterative version of FFT [CLR91], since the recursive version minimizes the loop parallelism and maximizes the function parallelism.

6.7.2 Sisal Code for FFT

```
% fft.sis

define main

type cmplx = record [real: double_real; imag: double_real];
type AC = array[cmplx];
type AR = array[double_real]

global sin (x: double_real returns double_real)
global cos (x: double_real returns double_real)

function add_c (a,b: cmplx returns cmplx)
    record cmplx [real: a.real + b.real; imag: a.imag + b.imag]
end function % add_c

function sub_c (a,b: cmplx returns cmplx)
    record cmplx [real: a.real - b.real; imag: a.imag - b.imag]
end function % sub_c

function mul_c (a,b: cmplx returns cmplx)
    record cmplx [real: a.real * b.real - a.imag * b.imag;
        imag: a.imag * b.real + a.real * b.imag]
end function % mul_c

function iter_shuffle (A: AC returns AC)
    let
        n := Array_Size(A);
    in
        for i in 1,n
            bitrev := for initial
                x := n/2; y := 0 ; b := i-1;
                while x ~= 0 repeat
                    y := if mod(old b,2) = 1 then
                            old y + old x
                         else
```

```
                        old y
                    end if;
                x := old x / 2;
                b := old b / 2
            returns value of y
        end for + 1
    returns array of A[bitrev]
    end for
  end let
end function % iter_shuffle

function fft (A: AC returns AC)
    let
        n := array_size(A)
    in
        for initial
            m := 2; B := A;
        while n >= m repeat
            hm := old m / 2; ndm := n / old m; OB := Old B; om := old m;
            B := for j in 0, ndm-1
                L,R := for i in 1,hm
                    x := 6.28318 / double_real(om);
                    pi := mul_c (record cmplx[
                       real: cos (x*double_real(i-1));
                       imag: -sin (x*double_real(i-1))],
                          OB[j*om + hm + i]);
                    li := OB[j * om + i];
                    Ti := add_c (li,pi);
                    Bi := sub_c (li,pi)
                returns array of Ti array of Bi
                end for;
            LR := L || R;
            returns value of catenate LR
            end for;
        m := old m * 2
        returns value of B
        end for
    end let
end function % fft

function main (n: integer returns AC)
    let
        h := n/2;
        C := for i in 1,n
            returns array of record cmplx[real: double_real(i); imag: 0.0]
        end for
    in
        fft (iter_shuffle(C))
    end let
end function % main
```

6.7.3 Program Characteristics

The one-dimensional iterative FFT algorithm makes successive passes over an array, where the access pattern changes with each iteration, in a fashion that is similar to the parallel prefix problem. Since the variable access pattern will disrupt any initial distribution of the data, this problem will highlight the effectiveness of multithreading at tolerating unavoidable remote references.

This FFT implementation redistributes the array elements using a bit-reversal of the array index and then combines the intermediate array elements in $lg\, n$ "butterfly" computations, where the size of the butterfly doubles at each stage. The Sisal compiler is able to determine that the sub-arrays fit into one result array, and thus pre-builds the sub-arrays "in-place," so that copying the sub-arrays into the final array is avoided. This is another example of the powerful *build-in-place* analysis that is performed by the current Sisal compiler [Can89]. However, the compiler does perform the updates in place, and thus allocates a new array for each of the logarithmic iterations, accounting for a large amount of message broadcasting at each iteration to remove the old array and allocate a new array. Also, the arrays are allocated in "ragged" fashion, which means that various offsets into the array are passed into the slices as the starting array addresses. This poses a problem for the VISA system, which attempts to locate the range map entry for a data structure given the real base address for the structure, not some offset into that structure. A solution to this problem is to delay the offset computation until after the array has been passed to the slices so that each slice can fetch the range map entry using the actual base address and then update the base address using an offset that is now also passed to each slice.

6.8 Laplace

6.8.1 Problem Description and Algorithm

Whereas SOR is a "smoothing" algorithm using a three-point stencil over a one-dimensional array, Laplace is a smoothing algorithm that uses a five-point stencil over a two-dimensional array. Specifically, each new $A_{i,j}$ element is computed as follows:

$$A_{i,j} = \begin{cases} A'_{i,j} & \text{if } i=1,\ i=n,\ j=1,\ j=n \\ A'_{i,j}/2.0 + (A'_{i-1,j} + A'_{i+1,j} + A'_{i,j-1} + A'_{i,j+1})/8.0 & \text{otherwise} \end{cases}$$

where A represents the current iteration and A' represents the previous iteration.

6.8.2 Sisal Code for Laplace

```
% laplace.sis

define main

type OneD = array[double_real];
type TwoD = array[OneD];

function 2d_fill (N: integer returns TwoD)
    for I in 1,N cross J in 1,N
        el := if mod(I+J,2) = 0 then
            double_real(1.0)
        else
            double_real(N)
        end if
    returns array of el
    end for
end function % 2d_fill

function laplace (Init_M: TwoD; N,KMax: integer returns TwoD)
    for initial
        K := 1;
        M := Init_M
    repeat
        K := old K + 1;
        M := for I in 1,N cross J in 1,N
            nM := if I=1|I=N|J=1|J=N then
                old M[I,J]
            else
                old M[I,J] / double_real(2.0) +
                    (old M[I-1,J] + old M[I+1,J] + old M[I,J-1] +
                    old M[I,J+1]) / double_real(8.0)
            end if
        returns array of nM
        end for
    until K >= KMax
    returns value of M
    end for
end function % laplace

function main (N,K : integer returns TwoD)
    let
        A := 2d_fill (N)
    in
        laplace (A, N, K)
    end let
end function % main
```

6.8.3 Program Characteristics

This problem highlights two-dimensional Sisal arrays, mapping functions, and the ability to combine task distribution with data distribution at a multi-dimensional level. The mapping functions attempt to minimize the edges (or boundary elements) in the distribution, since boundary elements require remote references. The five-point stencil also implies more remote references than SOR, giving multithreading the chance to tolerate these latencies.

Since the Sisal compiler does not support true rectangular arrays, the matrix in this program is implemented as an array of rows, where each row is a one-dimensional array of values. Also, each Sisal array is represented using three data structures, two descriptor structures and the actual array, and each of these data structures is placed into VISA space, requiring an additional VISA descriptor for each. Thus, for an $n \times n$ matrix, the Sisal compiler generates $6n+6$ data structures, $5n+5$ of which are replicated. Finally, since Laplace is an iterative algorithm, the compiler generates two additional swap matrices, bringing the total data structure count for the program to $3(6n+6)$, of which $3(5n+5)$ are replicated across all nodes. As we will see in Chapter 7, handling multi-dimensional arrays in this manner has a profoundly detrimental effect on the performance of the program.

6.9 Matrix Multiplication

6.9.1 Problem Description and Algorithm

Matrix multiplication computes $C = A \times B$, where A, B, and C are all $n \times n$ matrices. Specifically, $C_{i,j}$ is computed as:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} * B_{k,j}$$

6.9.2 Sisal Code for Matrix Multiply

```
% matmult.sis

define main

type OneD = array[double_real];
type TwoD = array[OneD];

function 2d_fill (n: integer; v: double_real  returns TwoD)
    for i in 1,n cross j in 1,n
        returns array of v
    end for
end function % 2d_fill

function matmult (n: integer; A,B: TwoD returns TwoD)
    for i in 1,n cross j in 1,n
    returns array of
        for k in 1,n
        returns value of sum A[i,k] * B[k,j]
        end for
    end for
end function % matmult

function main (n: integer; v1,v2: double_real  returns TwoD)
```

```
    let
        A := 2d_fill (n, v1);
        B := 2d_fill (n, v2);
    in
        matmult (n, A, B)
    end let
end function % main
```

6.9.3 Program Characteristics

Matrix multiplication highlights the two-dimensional mapping functions as well as the multithreading system. As with the other two-dimensional programs, the Sisal compiler generates an enormous number of data structures to accommodate the three two-dimensional data structures used in Matrix Multiply.

6.10 Cholesky Factorization

6.10.1 Problem Description and Algorithm

A system of n linear equations and n unknowns can be written in the form $Ax = b$, where A is the matrix of equations, x is the vector of unknowns, and b is the vector of solutions. We say that A has an *LU-decomposition* if A can be factored into a lower triangular matrix L and an upper triangular matrix U, such that $A = LU$. When $U = L^T$ so that $l_{ii} = u_{ii}$ for $1 \leq i \leq n$, the factorization algorithm is called *Cholesky's factorization*, after the mathematician André Louis Cholesky, who proved that if A is a real, symmetric, and positive definite matrix, then it has a unique factorization $A = LL^T$, in which L is a lower triangular with a positive diagonal.

Cholesky's factorization algorithm follows:

1. First, the diagonal elements of L are computed as:

$$l_{kk} = \left(a_{kk} - \sum_{s=1}^{k-1} l_{ks}^2 \right)^{1/2}$$

   for $k = 1, 2, \ldots, n$.

2. Then, using each of the computed diagonal elements, the lower triangular elements are computed as:

$$l_{ik} = \left(a_{ik} - \sum_{s=1}^{k-1} l_{is} l_{ks} \right) / l_{kk}$$

   for $i = k+1, k+2, \ldots, n$.

6.10.2 Sisal Code for Cholesky Factorization

```
% cholesky.sis

define main
```

```
type OneD = array[double_real];
type TwoD = array[OneD];

global sqrt (x: double_real returns double_real)

function cholesky (n: integer; A,LL: TwoD returns TwoD)
    for initial
        col := 1;
        diag := double_real(1.0);
        L := LL;
    while (col <= n) repeat
        col := old col + 1;
        diag := Sqrt (A[old col,old col] -
            for k in 1,old col - 1
            returns value of sum (old L[old col,k] * old L[old col,k])
            end for);
        L := for i in 1,n cross j in 1,i
            returns array of
            if (i = j) & (j = old col) then
                diag
            else  if j = old col then
                (A[i,j] - for k in 1,j-1
                returns value of sum (old L[i,k] * old L[j,k])
                end for) / diag
            else
                old L[i,j]
            end if
            end if
            end for;
    returns value of L
    end for
end function % cholesky

function main (n: integer returns TwoD)
    let
        LR := array_fill(1,n,double_real(0.0));
        L := array_fill(1,n,LR);
        A := for i in 1,n cross j in 1,i
            returns array of
            if i=j then
                double_real((n+i+j)*(n+i+j))
            else
                double_real(i+j)
            end if
        end for
    in
        cholesky (n,A,L)
    end let
end function % main
```

6.10.3 Program Characteristics

This program highlights the effectiveness of data distribution for a lower triangular array problem, and the ability of multithreading to tolerate the remote reference latencies. The triangular matrix is created as a Sisal two-dimensional array, where the size of the rows varies from 1 element to n elements. As Laplace and Matrix Multiply, Cholesky suffers from the problems of Sisal two-dimensional data structures. Additionally, as the Cholesky algorithm iterates through the columns, a new triangular matrix is allocated and the old triangular matrix is freed.

Chapter 7

EXPERIMENTAL RESULTS AND ANALYSIS

The purpose of computing is insight,
not numbers.
– R.W. Hamming

In this chapter we present a set of experiments used in evaluating the various design aspects of the VISA runtime system. In evaluating the performance of our sample programs, which are detailed in Chapter 6, we must choose an input problem size for the programs to use. When arrays are used, a fixed input size introduces the problem that a size which fits into a single node's memory is not large enough to saturate an order of magnitude larger machine configuration, and an array size that saturates a larger machine configuration often doesn't fit in a single node's memory. Therefore, for programs utilizing arrays, we will create three processor configuration groups: 1,2,4, and 8 nodes, 4,8,16, and 32 nodes, and 16, 32, 64, and 128 nodes. We will use the same array size within each group, but will use increasingly larger array sizes between the groups. We will measure the efficiency of our programs relative to the smallest configuration in each group, and call this measure the *relative efficiency (REff).*

7.1 Task Management Techniques

7.1.1 Task Management for Flat Loops

We compare the single-level and multi-level loop distribution schemes for a large unnested (flat) loop by measuring the performance of the Purdue Parallel Benchmark #1. Table 7.1 displays the execution times for this program, which ran for 10^7 iterations, where *Sp* represents the parallel speedup (T_1/T_n) and *Eff* represents the parallel efficiency (Sp/n). These execution times compare favorably to the single processor sequential C program that performs the same computation in 414.96 seconds. The disparity in single processor execution time between Sisal and C results from the fact that the Sisal compiler generates very efficient C code that is often easier for the native C compiler to optimize than hand-coded C programs.

Fan-out corresponds to the number of *sub-master* processes used in multi-level distribution, and *OFO* in Table 7.1 represents the optimal fan-out degree for each machine configuration. Fan-out provides a means to control the amount of *overhead* for multi-level distribution (exemplified by the sequential loop in the master process that distributes the sub-tasks), versus the *gain* of having the sub-masters distribute the slices in parallel. Figure 7.1 depicts this tradeoff for all possible fan-out degrees on 512 processors, where a fan-out degree of 1 represents single-level distribution. In the fan-out region [1..8] there is not enough sub-master parallelism, whereas in the fan-out region [128..512] there is too much overhead. A fan-out degree about equal to the square root of the number of

	single-level distribution			*multi-level-distribution*			
PEs	Time (s)	Sp	Eff (%)	OFO	Time (s)	Sp	Eff (%)
1	355.8472	1.00	100				
2	177.9336	2.00	100	2	177.9343	2.00	100
4	88.9698	4.00	100	2	88.9701	4.00	100
8	44.4940	8.00	100	2	44.4932	8.00	100
16	22.2639	15.98	100	4	22.2601	15.99	100
32	11.1649	31.87	100	4	11.1538	31.90	100
64	5.6461	63.03	98	8	5.6207	63.31	99
128	2.9484	120.69	94	8	2.8936	122.98	96
256	1.7236	206.46	81	16	1.6084	221.24	86
512	1.7582	202.39	40	32	1.3961	254.89	50

Table 7.1: Performance of Purdue #1, single and multi-level distribution

processors appears to be most effective for most applications, and is currently the default in our system.

Table 7.1 shows that the efficiency of this application using both single-level distribution and multi-level distribution is 100% up to 32 processors. For higher numbers of processors, the efficiency starts to decrease, getting much worse for 256 and 512 processors, especially for single-level-distribution. Thus, for this program, 64 processors is the point at which the overhead for sequentially distributing the loop slices starts to have a noticeable detrimental effect on the performance, and multi-level distribution is able to recapture some of the lost performance by parallelizing the distribution phase. For 256 and 512 processors, the gain is substantial: 5% and 10% efficiency, respectively.

Table 7.1 also reflects the effect that the ratio of *computation time to communication time* can have on the performance of the program. As we double the number of processors, the computation time of each parallel slice is *halved*, since the same number of loop bodies are now distributed over twice the number of loop slices, and the communication time is *doubled*, since there are now twice the number of loop slices to be distributed and twice the number of intermediate values to be reduced. Thus the ratio of computation time to communication time, which determines the efficiency of an application, is reduced by a factor of four each time the number of processors is doubled. We can see the effect of this diminishing ratio in the efficiency for 256 and 512 processors. Multi-level distribution reduces the rate at which the communication time increases, thus decreasing the rate at which efficiency is lost. If the initial loop-body computation time were reduced, the ratio would be more dependent on the communication time increase rather than the computation time decrease, and thus the effect of multi-level distribution on slowing the declining efficiency would be greater. To see this effect, we reduce the Taylor series from 14 terms to 7 terms, effectively halving the computation time for each loop iteration. The results of this experiment are displayed in Table 7.2. Comparing the gain in efficiency for multi-level distribution with the first experiment (Table 7.1), we see that the gain is increased. Given this performance gain, and the ability to employ multi-level distribution along with the other runtime options (such as multithreading), we will typically employ multi-level distribution for most of our sample programs, particularly in the last processor configuration grouping (16..128 processors).

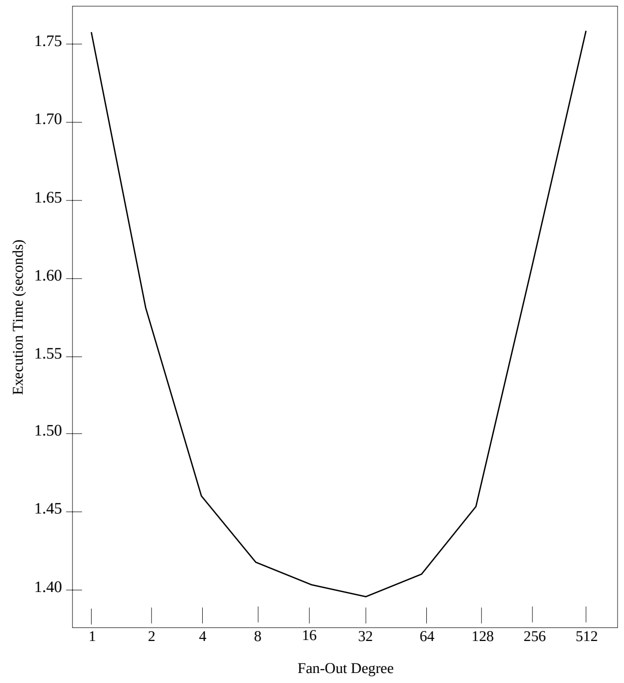

Figure 7.1: Performance of Purdue #1 on 512 processors, various fan-out degrees

	single-level distribution			*multi-level-distribution*			
PEs	Time (s)	Sp	Eff (%)	OFO	Time (s)	Sp	Eff (%)
1	149.8411	1.00	100				
2	74.9305	2.00	100	2	74.9309	2.00	100
4	37.4680	4.00	100	2	37.4684	4.00	100
8	18.7430	7.99	100	2	18.7424	7.99	100
16	9.3885	15.96	100	4	9.3846	15.97	100
32	4.7272	31.70	99	4	4.7161	31.77	99
64	2.4273	61.73	96	8	2.4017	62.39	97
128	1.3394	111.87	87	8	1.2842	116.68	91
256	0.9194	162.98	64	16	0.8029	186.83	73
512	1.3551	110.58	22	16	0.9155	163.67	32

Table 7.2: Performance of Purdue #1, 1/2 Taylor series

	Sequential			*Fully-Distributed*			*Partially-Distributed*		
PEs	Time (s)	Sp	Eff	Time (s)	Sp	Eff	Time (s)	Sp	Eff
1	296.6874	1.00	100						
2	148.3511	2.00	100	147.9001	2.00	100	148.3511	2.00	100
4	74.1801	4.00	100	74.0986	2.00	100	74.1801	2.00	100
8	37.0962	8.00	100	37.2061	7.97	100	37.0962	8.00	100
16	18.5605	15.98	100	18.9324	15.67	98	18.5605	15.98	100
32	9.3051	31.58	100	9.9582	29.79	93	9.3051	31.58	100
64	4.7005	63.12	99	5.8988	50.30	79	4.7005	63.12	99
128	4.7090	63.00	49	4.0136	73.92	58	3.5198	84.29	66
256	4.7261	62.78	25	4.1539	71.42	28	3.5128	84.46	33
512	4.7601	62.33	12	6.9894	42.45	8	5.1946	57.11	11

Table 7.3: Performance of Purdue #2, various inner loop distribution schemes

7.1.2 Task Management for Nested Loops

We now examine the effects of the nested loop distribution schemes, as defined in Table 4.1, on the performance of Purdue Parallel Benchmark #2. Table 7.3 gives the performance numbers for inputs $n = 64$ and $m = 524288$ (2^{19}), where *Sequential*, *Fully-Distributed*, and *Partially-Distributed* describe the inner loop distribution scheme, *Sp* is the parallel speedup, and *Eff* is the parallel efficiency (%). We intentionally select a value of n less than the total machine parallelism of the nCUBE/2 so that the effects of the *fully-distributed inner loop* and *partially-distributed inner loop* distribution schemes can be observed. The results in Table 4.1 confirm the following assumptions about these nested task distribution schemes:

- When the outer loop is sufficiently large to cover the available machine parallelism, the inner loop should be run sequentially to minimize the overhead for parallelism, since more parallelism is not needed.

- When the machine parallelism exceeds the outer loop parallelism, both the *Fully-Distributed* and the *Partially-Distributed* inner loop distribution techniques are effective at utilizing the excess machine parallelism, and the latter method is more efficient than the former method since the overhead of distribution and reduction have been minimized for the amount of parallelism needed to cover the available machine parallelism.

- For large numbers of processors (e.g., 512), the overhead for distributing the inner loop of this program defeats the purpose of utilizing the excess machine parallelism, and the *Sequential* inner loop distribution mechanism provides the best performance, even though many of the processors are not utilized.

- For this program, the inner loop body performs few computations so, as we increase the number of processors, the increase in communication time dwarfs the computation time. Thus this program, with these inputs, should not be executed on more than 128 processors, which provides the highest performance in terms of execution time (3.5128 s) using the *Partially-Distributed* inner loop distribution scheme.

Method		Size	Time(s)	Size	Time(s)
Fixed	1	32768	16.5964	33280	16.9382
Fixed	2048	32768	86.4301	33280	92.7254
Fixed	4096	32768	1.0007	33280	16.7632
Fixed	8192	32768	138.8981	33280	115.0115
Var	No Opt	32768	1.1426	33280	1.1744
Var	Opt	32768	0.1709	33280	0.1699

Table 7.4: Performance of fixed vs. variable blocksize for SOR, 8 PEs

7.2 Address Translation: Fixed Versus Variable Blocksize

To assess the relative performance of the fixed-blocksize address translation scheme outlined in Section 5.4 and the effectiveness of our variable-blocksize address translation scheme with the local base optimization, we study the SOR program running on an eight processor nCUBE/2. The results of our comparison are displayed in Table 7.4, where four cases of the fixed mapping scheme are examined: blocksize = 1, blocksize = 2048, blocksize = 4096, and blocksize = 8192. We also compare the performance of the unoptimized variable-blocksize translation scheme and the optimized variable-blocksize translation scheme, both of which employ a blocksize of n/p.

The outer loop of SOR runs sequentially, whereas the inner loop is distributed over the eight processors in equal sized (n/p) loop slices. When combined with an array of size 32768, a blocksize of 4096 elements provides an ideal distribution, which is reflected in the fixed blocksize of 4096. The fixed blocksize of 8192 elements results in the array occupying only half of the physical memories, resulting in "hot-spot" message traffic to those nodes which possess a portion of the array. Additionally, it creates a mismatch between loop and data distribution. Likewise, the fixed blocksize of 2048 elements creates a mismatch between the loop and data structures, though the smaller blocksize helps to provide a more uniform distribution that reduces the hot-spot message traffic pattern. The interleaved data distribution (blocksize = 1) equally distributes the data and message traffic, though again running out of alignment with the loop.

When using an array of size 33280 elements, *all* fixed-blocksize distribution schemes run out-of-alignment with the loop iteration space, resulting in a drastic loss in performance for the previously optimal fixed blocksize of 4096 elements.

The variable-blocksize address translation scheme, represented in the lower portion of Table 7.4, employs a blocksize of n/p elements which, for this program, always coincides with the loop distribution blocksize. The result is that the variable-blocksize address translation scheme *always* runs in alignment with the loop slices, thereby minimizing the number of remote references, even when the array size is not a multiple of the number of processors.

As can be seen from Table 7.4, the local address optimization described in Section 5.5 has a considerable impact on performance. We therefore, by default, employ the optimized variable-blocksize address translation scheme for all of our sample programs.

PEs	Array Size	Time (s)	REff (%)
1	65536	89.0208	100.0
2	65536	45.4485	97.9
4	65536	23.7399	93.7
8	65536	12.0825	92.1
4	262144	90.1753	100.0
8	262144	45.4274	99.3
16	262144	23.8274	94.6
32	262144	13.7801_4	81.8
16	1048576	90.5167	100.0
32	1048576	46.6536_4	97.0
64	1048576	25.7870_8	87.8
128	1048576	17.0495_{16}	66.4

Table 7.5: Performance of SOR, 100 iterations

7.3 One-Dimensional Arrays

In this section we present performance results for the one-dimensional array programs. Table 7.5 gives performance of the SOR program using the processor configuration grouping discussed earlier. In the last group of processors (16..128) we employ multi-level distribution to reduce the overhead of distributing the large flat loop.

Table 7.6 gives the performance for Lawrence Livermore Loop #7, which creates an array of n elements from an input array of $n + 6$ elements. The default blocksize of n/p elements used by the `block_map` distribution function creates a misalignment between the input and output arrays, resulting in an excessive number of remote references. To minimize the number of remote references, the two arrays are distributed using the same blocksize, namely $(n + 6)/p$. However, the loop must now also be distributed using a blocksize of $(n+6)/p$ iterations per slice, which is accomplished by modifying the blocksize control parameter of the loop. The result, where the performance is given in Table 7.6 under the *Custom Blocking* heading, is that the input and output arrays are properly aligned both with each other and the loop space, resulting in a minimal number of remote references. However, the loop iteration space is now unevenly distributed among the processors, with the first $p - 1$ processors each receiving $(n + 6)/p$ iterations and the final processor receiving the remainder. For this program, the reduced number of remote references outweighs the load imbalance.

Table 7.7 gives the performance for Lawrence Livermore Loop #1, which creates an array of n elements from an input array of $n + 11$ elements. As with Loop #7, we examine the performance of both the default and custom blocking schemes. For this program, the benefit of reducing the number of remote references does not cover the cost of the load imbalance, causing the *Custom Blocking* method to degrade the overall performance of the program.

Table 7.8 gives the performance of the parallel prefix program, and Table 7.9 gives the performance of the FFT program. Both of these programs have access patterns that vary for each iteration of the algorithm, causing a very large number of remote references, regardless of the original array distribution. For example, in the parallel prefix algorithm,

		Default Blocking		*Custom Blocking*		
PEs	Array Size	Time (s)	REff (%)	Time (s)	REff (%)	Sp
1	65536	74.9312	100.0	74.9313	100.0	1.00
2	65536	40.6622	92.1	38.5077	97.3	1.06
4	65536	23.9994	78.1	20.0551	93.4	1.20
8	65536	13.3095	70.4	10.7879	86.8	1.23
4	262144	79.7722	100.0	76.2464	100.0	1.05
8	262144	41.3235	96.5	38.8856	98.0	1.06
16	262144	27.0599	73.7	20.2864	94.0	1.33
32	262144	23.7752	41.9	11.1579	85.4	2.13
16	1048576	83.0454	100.0	76.4765	100.0	1.09
32	1048576	53.0769	78.2	39.2500	97.4	1.35
64	1048576	46.5863	44.6	21.1387	90.5	2.20
128	1048576	64.9865	16.0	13.1271	72.9	4.95

Table 7.6: Performance of Livermore loop #7, 50 iterations

		Default Blocking		*Custom Blocking*		
PEs	Array Size	Time (s)	REff (%)	Time (s)	REff (%)	Sp
1	65536	32.2601	100.0	32.2602	100.0	1.00
2	65536	16.6137	97.1	16.6764	96.7	1.00
4	65536	8.8835	90.1	8.9769	89.8	0.99
8	65536	5.6399	71.5	6.0811	66.3	0.93
4	262144	33.0699	100.0	33.1572	100.0	1.00
8	262144	17.3766	95.2	17.1698	96.6	1.00
16	262144	9.3532	88.4	9.5200	87.1	0.99
32	262144	7.5177	55.0	9.0400	45.8	0.83
16	1048576	33.5528	100.0	33.5487	100.0	1.00
32	1048576	20.4575	82.0	21.0452	79.7	0.97
64	1048576	15.9578	52.6	18.2375	46.0	0.87
128	1048576	20.4108	20.5	25.1038	16.7	0.81

Table 7.7: Performance of Livermore loop #1, 50 iterations

PEs	Array Size	Time (s)	REff (%)
1	65536	10.9807	100.0
2	65536	65.3893	8.3
4	65536	54.7874	4.9
8	65536	36.8320	3.7
4	262144	220.2234	100.0
8	262144	147.3423	74.7
16	262144	92.3348	59.6
32	262144	55.9987	49.1
16	1048576	369.9710	100.0
32	1048576	222.2807	83.2
64	1048576	130.6187	70.8
128	1048576	76.3341	60.6

Table 7.8: Performance of parallel prefix

the computation of A_i requires the values from A_i and A_{i+d} in the previous iteration, where d is the "distance" between the references, and increases from 1 by a factor of two for each iteration. Therefore, when the distance is greater than the blocksize, all A_{i+d} references will be remote. In Chapter 8 we discuss future work that would enable us to *re-map* the array once the distance exceeds the blocksize, effectively resetting the distance to 1. The FFT program performs particularly poorly due, in part, to the large number of remote references caused by the "butterfly" access pattern (similar to parallel prefix), but also because the Sisal compiler is unable to re-use the resulting array at each iteration of the butterfly. The result is that, for each iteration, the old array is deallocated (when the reference count reaches zero) and a new array is allocated. This causes an excessive amount of remote references used in broadcasting information about the old and new VISA data structure descriptors.

Both programs exhibit very little speedup, due to the very low computation to communication ratio. If we examine the total remote reference count for parallel prefix in the first processor group [1..8], we see that there are 0 remote references for 1 processor, 65535 remote references for 2 processors, 130978 remote references for 4 processors, and 196601 remote references for 8 processors. The remote references grow as the number of processors increases, but the total amount of computation remains the same. Given that a remote reference is about 80 times as expensive as either a local reference of a local computation, it is easy to see how the computation to communication ratio drops for this program. The result is that parallelism is not very effective for these two programs, and that parallel execution is only desirable for space considerations.

7.4 Explicit Versus Implicit Programming Styles

As was stated in Chapter 2, a parallel program executing on a distributed memory multiprocessor must address two issues, either explicitly or implicitly:

1. Task management. Parallel execution is achieved by *dividing* the portions of code which may be executed in parallel into parallel tasks, *distributing* the tasks among

PEs	Array Size	Time (s)	REff (%)
1	4096	8.0910	100.0
2	4096	39.1992	10.3
4	4096	36.4452	5.6
8	4096	27.9294	3.6
4	8192	85.4928	100.0
8	8192	81.5629	52.4
16	8192	77.8649	27.4
32	8192	72.6381	14.7
16	16384	190.0012	100.0
32	16384	177.2432	53.6
64	16384	168.7034	28.2
128	16384	162.1386	14.6

Table 7.9: Performance of FFT

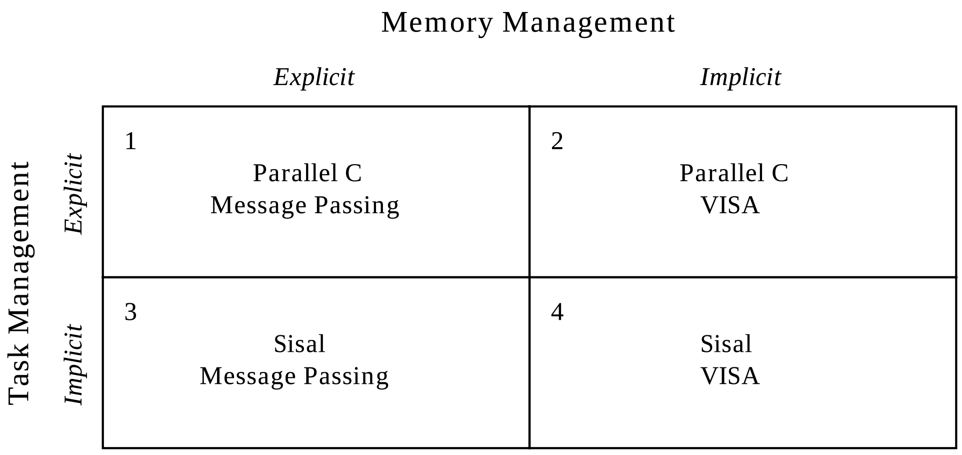

Figure 7.2: Parallel programming style combinations

the participating nodes for parallel execution, and *synchronizing* their results so that the computation remains determinate.

2. Memory management. Global data structures need to be *distributed* among the participating nodes in such a way as to minimize the number of remote references generated by the execution of the parallel tasks. Once a distribution is agreed upon, the program must identify those references that fall outside of the local distribution (i.e. remote), and communicate the request to the node which contains the value.

Given these two orthogonal programming issues, either of which may be handled explicitly or implicitly, there are four possible parallel programming style combinations, as depicted in Figure 7.2:

1. Explicit task management using parallel C and explicit memory management using message passing primitives. Similar to assembly language, this style represents the lowest level of abstraction, but the possibility for the highest level of performance.

2. Explicit task management using parallel C and implicit memory management provided by the VISA runtime system. This style alleviates the programmer from the details of a distributed memory system and explicit message passing.

3. Implicit task management using Sisal and explicit memory management using message passing primitives. This represents a machine-dependent Sisal compiler that has been given the ability to generate explicit distributed memory code, much like the distributed memory Fortran compilers [HKT92, ZBG86]. However, such a modification to the compiler has not been undertaken, and thus we cannot expand on this style in our analysis.

4. Implicit task management using Sisal and implicit memory management provided by the VISA runtime system. This represents the opposite end of the programming effort spectrum from explicit parallel C with message passing.

To measure the relative merits of each style, in terms of programming effort and execution speed, we encode two applications in the three programming styles (1, 2, and 4) specified above. The two codes we selected, SOR and Livermore Loop #7, are designed to highlight the effectiveness of either task or memory management techniques. With very little task management required, Loop #7 highlights the differences between the implicit and explicit memory management styles. The iteration loop in SOR provides a method of controlling the amount synchronization required, thus highlighting the differences between implicit and explicit task management.

Both of these programs were encoded using the three programming styles as follows:

- *Sisal with VISA*. Both codes were transformed into Sisal directly from their mathematical descriptions. The code only specifies *what* is to be computed, not *how* the computations are to proceed. The result is a machine-independent specification of the problem that runs on any machine Sisal supports.

- *Explicit parallel C with VISA*. Moving into explicit task management, the codes have to specify how the parallel loop is to divided among the workers, and how explicit synchronization is to be performed. Memory management is handled by the VISA system, however, for the Livermore Loop #7 code, special registers were employed to cache the values of the B array so that multiple remote references to retrieve the same value were eliminated.

- *Explicit parallel C with message passing*. Moving away from the VISA system, the explicit task management code is augmented with explicit message passing designed to optimize the number of remote references required and perform all remote references before the computation loop is initiated (*pre-fetching*). Also, the communication model is changed from an interrupt-driven request/reply model used in VISA to a synchronous read/write model so that the overhead of the interrupt handler can be avoided. This allows the computation (inner) loop to run completely without remote references. Special buffers are used to hold the pre-fetched values, and synchronous communication phases are necessary to avoid deadlock. The distribution of data among the processors is also explicitly stated, and altering this distribution would require re-coding both the explicit communication and computation phases.

Measure	*LLNL Loop #7*			*SOR*		
	SISAL	C+VISA	C+MP	SISAL	C+VISA	C+MP
Lines of code	25	163	338	24	184	459
Time to encode (hrs)	0.25	2.5	9.5	0.25	3.0	11.0

Table 7.10: Comparison of programming effort, in both time and space

PEs	Array Size	SISAL	C+VISA		C+MP	
		Time (s)	Time (s)	Sp_1	Time (s)	Sp_2
1	65536	1.8002	1.3232	1.36	0.7462	1.77
2	131072	1.8699	1.3868	1.35	0.7479	1.85
4	262144	1.9307	1.3983	1.38	0.7493	1.86
8	524288	1.9322	1.3922	1.39	0.7518	1.85
16	1048576	2.0143	1.3959	1.44	0.7569	1.84
32	2097152	2.2006	1.4029	1.57	0.7673	1.83
64	4194304	2.5794	1.4173	1.81	0.7882	1.80
Ave.				1.47		1.83

Table 7.11: Performance of Livermore loop #7, various programming styles

7.4.1 Results and Analysis

We compare the relative merits of each programming style using two metrics: programming effort and performance. Table 7.10 displays the programming effort in terms of lines of code that the user is responsible for writing, and approximate time it took us to code and debug each of the programs, where, as in all of our tables, SISAL represents the Sisal codes, C+VISA represents the explicit parallel C with VISA codes, and C+MP represents the explicit parallel C with message passing codes. The claim that implicit parallel languages ease the task of programming distributed memory multiprocessors is clearly supported by these numbers. We acknowledge that these measurements are subjective as to the overall programming effort, however, they do paint a realistic picture of the relative difficulties of these programming styles. As we move from Sisal to explicit C with VISA, and to explicit C with message passing, the code becomes increasingly more complex, requires increasingly more lines of code, and becomes more machine-dependent. The question, then, is whether increased performance justifies the additional programming effort.

Table 7.11 gives the execution results for Loop #7, where a constant blocksize of 65536 (2^{16}) double-precision elements is used and *Array Size* represents the total size of the A and B arrays, Sp_1 represents the speedup in going from Sisal to C with VISA (T_{SISAL}/T_{C+VISA}), and Sp_2 represents the speedup in going from C with VISA to C with message passing (T_{C+VISA}/T_{C+MP}). In order to highlight the performance gain achieved by explicit memory management, the blocksize, or number of array elements per processor, was kept constant at 65,536 (2^{16}) double-precision elements. The data reveals that an average speedup of 1.47 is achieved when going from Sisal to explicit C with VISA,

			SISAL	C+VISA		C+MP	
PEs	Blocksize	Ratio	Time (s)	Time (s)	Sp_1	Time (s)	Sp_2
1	65536	.002	114.7980	119.6738	0.96	51.9780	2.30
2	32768	.004	58.2668	60.8672	0.96	41.1032	1.48
4	16384	.008	30.2806	30.4173	0.99	21.0470	1.46
8	8192	.016	15.5127	15.4519	1.00	10.6547	1.45
16	4096	.032	9.1281	8.1962	1.11	5.5524	1.48
32	2048	.063	7.2312	5.0998	1.42	3.1722	1.61
64	1024	.125	8.8509	4.4798	1.97	2.6409	1.69
Ave.					1.20		1.64

Table 7.12: Performance of SOR, various programming styles

which is due to the memory caching optimization rather than the explicit control of tasks. Additionally, an average speedup of 1.83 is achieved when moving from explicit C with VISA to explicit C with message passing, demonstrating the overhead of the VISA system and the effectiveness of the pre-fetching optimization. In terms of space requirements, Sisal uses the minimum: two arrays of size n, one for A and one for B. Explicit C with VISA allocates an additional 7 double-precision locations per array to cache the values of B_i through B_{i+6} so that they need only be retrieved once. Explicit C with message passing also allocates an additional block of 7 elements to store the values of B that reside on the neighboring node.

Table 7.12 gives the execution results for SOR, where a constant array size of 65536 (2^{16}) double-precision elements and 128 iterations is used. Again, the array size is held constant, causing the blocksize to decrease and the ratio of iterations to blocksize to increase as the number of processors increases. This ratio represents the increasing emphasis being placed on task management. In moving from Sisal to explicit C with VISA, there is an average speedup of 1.20, which starts as a performance decrease and gains as the ratio of iterations to blocksize increases, placing greater emphasis of task management on the total execution time. This initial loss in performance is due to the ability of the Sisal compiler to generate code that is highly optimized, which sometimes outperforms normal hand-coded C. However, this small gain is quickly lost as the complex Sisal task management system is outperformed by the hand-coded C task management. In moving from explicit C with VISA to explicit C with message passing, there is an average speedup of 1.64, again representing the overhead of VISA and the effectiveness of pre-fetching all remote references. The low speedup of the explicit C with message passing in going from 1 processor to 2 processors shows the enormous overhead of the synchronization that this problem creates. This drop in speedup is not visible in the other two approaches due to the overhead of the VISA system. In terms of space requirements, explicit C with VISA uses the minimal two arrays of size n, one for the previous iteration and one for the current iteration, and pointers are swapped at the end of each iteration. The Sisal compiler also recognizes this optimization, but generates the two swap arrays only after generating an array to hold the initial values, resulting in a space overhead of n elements. The explicit C with message passing uses only the two necessary arrays, but allocates an additional two elements per processor to hold the pre-fetched remote values from neighboring nodes.

7.4.2 Summary of Data

Sisal with VISA provides implicit management of both tasks and data, and offers reasonable performance while alleviating the programmer from the implementation details of an architecture, resulting in relatively efficient machine-independent code that is portable among a wide range of architectures [Can92]. Furthermore, since the current Sisal compiler is unaware of distributed memory and costs associated with accessing remote data, we expect a performance gain when such information is exploited by the compiler [WF92].

Explicit parallel C with VISA offers the ability to increase the performance of an application, but at the cost of increased code size, programming effort, and machine-dependence. For our simple programs, an average speedup of 1.34 over Sisal is achieved, but at the cost of increasing the code size by an average factor of 7, and increasing the time required to encode and debug the programs by an average factor of 11.

Explicit parallel C with explicit message passing offers the ability to exploit the problem and machine details to obtain the highest performance for a particular machine. For our programs, average speedups of 1.74 over C with VISA, and 2.34 over Sisal are achieved. Once again, this increase in performance is obtained at the cost of increasing program sizes by an average factor of 2 over explicit C with VISA, and by a an average factor of 15 over Sisal, while increasing the time required to encode and debug the programs by an average factor of 4 over explicit C with VISA, and by an average factor of 40 over Sisal.

The results show that although implicit parallel programming can offer reasonable performance, it is possible to increase the performance by taking explicit control over task management or data management. It is the decision of the applications programmer as to whether the increase in performance warrants the increase in programming effort when moving from implicit to explicit programming styles, but the option should nonetheless be available.

7.5 Two-Dimensional Arrays

To evaluate the effectiveness of our runtime system in aligning data and loops, we shall examine the distribution and performance of the Laplace program. Laplace employs a five-point stencil computation, which implies that the computation of all boundary elements for a give distribution will require remote references. Thus our first intuition is to minimize the number of elements on the distribution boundaries. We examine two matrix mapping functions to see how effective they are at minimizing remote references. For our comparisons, let us assume that the matrix size is $256 \times 256 (n = 256)$ and we are using 16 processors $(p = 16)$.

- The *matrix_row_map* mapping function allocates b contiguous rows to each processor, where $b = n/p$. This mapping scheme eliminates all of the interior remote references, leaving only those on every b-th row boundary. Thus we have a total of $(p-2)2(n-2)+2(n-2)$ remote references , which is 7,620 for our example of a 256×256 matrix.

- In the *matrix_block_map* mapping function the p processors are arranged in a $\sqrt{p} \times \sqrt{p}$ grid, each owning a $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of the matrix. In the example case this would lead to a 4x4 grid with 128x128 elements per processor, with 4 corner processors performing 254 remote references each, 8 side processors performing 382

		matrix_row_map		*matrix_block_map*		
PEs	Matrix Size	Time (s)	REff (%)	Time (s)	REff (%)	Sp
1	256x256	17.1284	100.0	18.6020	100.0	0.92
2	256x256	27.8498	30.8	34.3491	27.1	0.81
4	256x256	19.4244	22.0	24.4763	19.0	0.79
8	256x256	20.7248	10.3	22.7965	10.2	0.90
4	512x512	52.2188	100.0	53.0218	100.0	0.98
8	512x512	47.8333	54.5	51.3777	51.6	0.93
16	512x512	58.5527	22.3	63.2369	21.0	0.92
32	512x512	89.9674	7.3	97.4665	6.8	0.92
16	1024x1024	129.7875	100.0	134.5676	100.0	0.96
32	1024x1024	185.1608	35.0	199.6552	33.7	0.93
64	1024x1024	318.1579	10.2	346.8237	9.7	0.92
128	1024x1024	529.3233	3.1	646.9596	2.6	0.82

Table 7.13: Performance of 2D Laplace, row versus block map, 10 iteration

remote references each, and 4 interior processors performing 512 remote references each, producing total of 6,120 remote references.

The results of running Laplace with the two matrix mapping functions are given in Table 7.13. These results are clearly disappointing. The poor performance is caused by the need for replicating the administrative data structures of the two dimensional arrays, creating $3(3n+3)$ Sisal data structures and $3(3n+3)$ VISA data structures (range map entries), for a total of $3(6n+6)$ administrative data structures, $3(5n+5)$ of which must be replicated (see Section 6.8.3). In a one PE machine there is no broadcast, hence the much better sequential performance. Dealing with N-dimensional arrays in this fashion works for shared memory machines, but is clearly unacceptable in a distributed memory machine. The correct way to solve this problem is to have true N-dimensional arrays in Sisal, resulting in one descriptor for the whole structure. Sisal 2.0 [BCFO91] defines true N-dimensional arrays and gives a method of distributing regions of these arrays. A quick fix to this problem given the current version of Sisal is to represent the matrix by a one dimensional structure and rewrite the inner loop of Laplace as follows:

```
M := for k in 1,n*n
       i := (k-1)/n + 1; j := mod(k,n);
      nM := if i=1|i=n|j=1|j=0
            then old M[k]
            else old M[k] / 2.0 + (old M[k-n] + old M[k+n] +
                 old M[k-1] + old M[k+1])/8.0
            end if
      returns array of nM
     end for
```

Table 7.14 presents the results of the improved Laplace program, using a one dimensional array and a block mapping function that corresponds to the *matrix_row_map*

		Sisal 2D Arrays		*True 2D Arrays*		
PEs	Matrix Size	Time (s)	REff (%)	Time (s)	REff (%)	Sp
1	256x256	17.1284	100.0	16.6974	100.0	1.03
2	256x256	27.8498	30.8	18.5326	45.0	1.50
4	256x256	19.4244	22.0	12.4936	33.4	1.55
8	256x256	20.7248	10.3	8.7825	23.8	2.36
4	512x512	52.2188	100.0	41.3844	100.0	1.26
8	512x512	47.8333	54.5	25.4780	81.2	1.88
16	512x512	58.5527	22.3	17.4767	59.2	3.35
32	512x512	89.9674	7.3	14.5955	35.4	6.16
16	1024x1024	129.7875	100.0	50.6113	100.0	2.56
32	1024x1024	185.1608	35.0	36.0326	70.2	5.14
64	1024x1024	318.1579	10.2	29.1644	43.4	10.91
128	1024x1024	529.3233	3.1	26.8367	23.6	19.73

Table 7.14: Performance of improved Laplace (`matrix_row_map`)

function. The results are better, albeit not impressive yet. They can be further improved by employing multithreading (see Section 7.6), and even more so by having the compiler generate block moves allowing a whole row to be communicated between nodes. This is a case where making the compiler aware of the distributed memory architecture, and performing the appropriate analysis and optimization, will provide results superior to the general runtime approach.

Table 7.15 gives the performance of matrix multiply and Table 7.16 gives the performance of Cholesky factorization, both of which employ Sisal two-dimensional data structures. Based on the results obtained for Laplace, it is not surprising to see the poor performance of these other two-dimensional array programs. As with parallel prefix and FFT, the two-dimensional problems all suffer from excessively low computation to communication ratios. Matrix multiply allocates three two-dimensional data structures, one for each input matrix and one for the computed matrix, all of which are allocated using the `matrix_row_map` mapping function, since using the `matrix_block_map` function would require re-writing the matrix multiply algorithm to perform sub-block multiplications. We did consider explicitly transposing the B matrix so that both A and B would be accessed in row-major fashion, which coincides with the distribution. However, this strategy does not alter the number of B elements held locally by any one processor, but alters the way in which the elements are accessed. As a result, the number of remote references during the matrix multiply is the same whether the B matrix is transposed or not, though actually transposing the matrix incurs a substantial number of remote references. Therefore, it is better to leave the B matrix in its original form and access it in column-major fashion. Though matrix multiply does observe some degree of speedup, the overall poor performance is due to the overheads in handling Sisal two-dimensional arrays, which is exacerbated in the Cholesky program.

Table 7.16 gives the dismal performance of the Cholesky program, which employs triangular arrays that are implemented as two-dimensional data structures in Sisal. Therefore, although the actual amount of data space occupied by the triangular arrays is less

PEs	Matrix Size	Time (s)	REff (%)
1	64x64	2.9211	100.0
2	64x64	80.3168	1.8
4	64x64	72.4649	1.0
8	64x64	46.4425	1.0
4	128x128	581.9276	100.0
8	128x128	345.1970	84.3
16	128x128	231.3354	62.9
32	128x128	170.6795	42.6
16	256x256	1625.3946	100.0
32	256x256	1051.7490	77.3
64	256x256	837.5011	48.5
128	256x256	1248.3309	16.3

Table 7.15: Performance of matrix multiply

PEs	Matrix Size	Time (s)	REff (%)
1	32x32	15.4610	100.0
2	32x32	32.5118	23.8
4	32x32	33.4192	11.6
8	32x32	36.5835	5.3
4	64x64	216.2966	100.0
8	64x64	221.6448	48.8
16	64x64	229.6827	23.5
32	64x64	271.6236	10.0

Table 7.16: Performance of Cholesky Factorization

than for a full two-dimensional array, the number of data structures required to represent the array $(6n+6)$ remains the same. Additionally, the Cholesky program allocates a new matrix for each iteration of the algorithm, where an iteration updates a column in the matrix. Therefore, in the 32×32 problem, there are 2 initial matrices created plus 32 subsequent matrices. Since the number of control structures for all 34 matrices would exceed the capacity of the VISA system, old matrices must also be freed at each iteration, which requires a substantial amount of time given that the range map table is represented by an array and must remain in sorted order to facilitate the binary search lookup. Therefore, while deallocating an array and removing the $3n+3$ VISA data structures from the range map table, new allocation request messages pile up in the input buffer until the buffer fills and a message is dropped by the network, resulting in a system failure. This is the case for the largest processor configuration attempting to work on a problem size of 128×128, which is why the table only contains numbers for the first two groupings.

Given the performance of Laplace, matrix multiply, and Cholesky using Sisal two-dimensional arrays, and the improvement in Laplace when going to a true rectangular

	No Multithreading	*Multithreading*		
K	Time (s)	MT	Time (s)	Sp
2	319.7981	16	261.1423	1.23
4	163.2623	16	132.1156	1.24
8	84.3097	16	69.4269	1.21
16	44.8897	16	38.6020	1.16
32	24.6900	16	22.7694	1.08
64	15.5231	16	14.5893	1.07
128	11.0904	8	10.4830	1.06
256	8.7077	8	8.2148	1.06
512	7.9346	4	7.4274	1.06
1024	6.9820	4	6.9516	1.01
2048	6.6903	2	6.6887	1.00
4096	6.5692	2	6.5608	1.00
8192	6.5136	2	6.5265	1.00

Table 7.17: Performance of synthetic program, 8 PEs, 65536 elements, 50 iterations

array data structure, it is evident that a distributed memory Sisal compiler cannot continue to deal with multi-dimensional data structures in the current manner.

7.6 Multithreading

In order to evaluate the multithreading system and the analysis given in Section 4.5.2, we begin our study with a synthetic program in which we can vary the number of remote references. The program successively transforms an array, similar to the SOR program, and performs a remote access for every K_{th} array element to be computed.

Table 7.17 gives the results of this program for 50 iterations, using an array size of 65536 elements, and running on eight processors. MT gives the optimal number of threads per processor, and Sp gives the speedup due to multithreading. The arrays are divided in 8 parts (1 per node) of 8192 elements each. In the case of K=2, there are 4096 remote references per processor per iteration. The result is that there is very little work surrounding the remote references, and so 16 threads are necessary to avoid switching back to a thread before the remote reference has been satisfied. The gain per remote reference is therefore the difference in the multithreading and non-multithreading times ($319.7 - 261.1 = 58.6s$) divided by the number of remote references (4096 per iteration $\times$ 50 iterations), which is $58.6/(50 * 4096) = 286\mu s$. It turns out that in the non-multithreading case, almost all message interrupts to handle remote read requests occur while the processor is waiting for its own remote reference. This is due in part to the highly regular access pattern that is identical for all processors, which creates very synchronous message traffic. Also, almost all context switches are successful in the multithreading case. Therefore the actual gain of $286\mu s$ coincides closely with the estimated gain (using Eq. 4.2) of $600 - 350 = 250\mu s$. This behavior continues for values of K from 2 to 16, at which point the effects of latency hiding are diminished by the overheads of multithreading. Finally, when the number of remote references becomes very small (K=1024), the multithreading overheads govern the behavior.

		No Multithreading		*Multithreading*		
PEs	Array Size	Time (s)	REff (%)	Time (s)	REff (%)	Sp
1	65536	89.0208	100.0			
2	65536	45.4485	97.9	45.9884	96.8	0.98
4	65536	23.7399	93.7	23.9641	92.9	0.99
8	65536	12.0825	92.1	12.1591	91.5	0.99
4	262144	90.1753	100.0	90.6536	100.0	0.99
8	262144	45.4274	99.3	45.5038	99.6	0.99
16	262144	23.8274	94.6	23.8850	94.9	1.00
32	262144	13.7801	81.8	13.8359	81.9	0.99
16	1048576	90.5167	100.0	90.5733	100.0	1.00
32	1048576	46.6536	97.0	46.6836	97.0	1.00
64	1048576	25.7870	87.8	25.7963	87.7	1.00
128	1048576	17.0495	66.4	17.0916	66.2	1.00

Table 7.18: Performance of SOR with multithreading, MT=2, 100 iterations

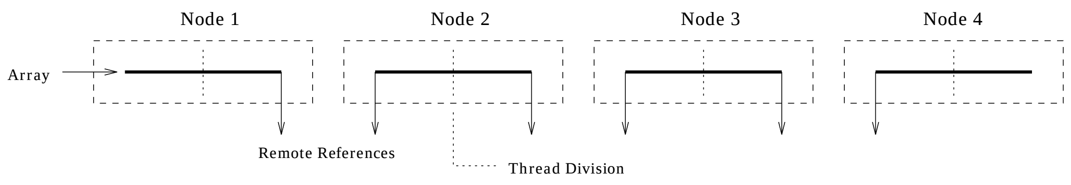

Figure 7.3: Array distribution for SOR program

Table 7.18 provides performance data for SOR with multithreading, where the number of threads is 2 for all cases, *REff%* gives the relative efficiency for the group, and *Sp* gives the speedup caused by multithreading. Due to the locality of the data references in the program and the effectiveness of the mapping function in exploiting this locality, Sisal exhibits good relative speedups, leaving little room for improvement for multithreading. In fact, locality works so well for this program, that there are simply not enough remote references that result in successful switches to cover the costs of setting up multithreading and fetching thread parameters.

Relating these results to our analytical model, the SOR program contains a sequential outer loop that iterates over the array being smoothed, and a parallel inner loop in which each array element is updated using values from the array in the previous iteration. Figure 7.3 shows the distribution of the array onto four nodes, and how these sub-arrays are again divided over the two threads per node. The first node has zero successful switches as its first thread, which has no remote references, executes to completion before the second thread is started. Although the second thread contains a single remote reference, there are no other threads to tolerate the latency of the reference. Therefore, multithreading only infers extra costs (P=0, H=0 in Eq. 4.2). Internal nodes have two successful context switches, which almost covers the costs of multithreading (P=2, H=2 in Eq. 4.2). The last node has only one successful switch, and therefore does not gain from multithreading either (P=2, H=1 in Eq. 4.2). Since barrier synchronization forces all nodes to wait for

		No Multithreading		*Multithreading*		
PEs	Matrix Size	Time (s)	REff (%)	Time (s)	REff (%)	Sp
1	256x256	16.6974	100.0			
2	256x256	18.5326	45.0	11.2740	74.1	1.65
4	256x256	12.4936	33.4	8.9311	46.7	1.39
8	256x256	8.7825	23.8	6.8160	30.6	1.27
4	512x512	41.3844	100.0	26.7381	100.0	1.54
8	512x512	25.4780	81.2	18.2013	73.5	1.39
16	512x512	17.4767	59.2	13.6886	48.8	1.27
32	512x512	14.5955	35.4	11.5710	28.9	1.26
16	1024x1024	50.6113	100.0	36.4239	100.0	1.39
32	1024x1024	36.0326	70.2	27.3604	66.6	1.31
64	1024x1024	29.1644	43.4	23.1235	39.4	1.26
128	1024x1024	26.8367	23.6	20.3319	22.4	1.32

Table 7.19: Performance of Laplace with multithreading, MT=16, 10 iterations

the slowest, this problem, as can be seen in both the cost analysis and the actual data, does not benefit from multithreading.

Table 7.19 gives the performance results for Laplace with multithreading, where the number of threads is 16 for all multithreading cases, *REff%* gives the relative efficiency within the group, and *Sp* gives the speedup due to multithreading. For this experiment, we use the improved version of Laplace that utilizes a one-dimensional array rather than the Sisal two-dimensional arrays. This is done so that the effects of multithreading can be seen in the performance of the program. Going from no message passing (1 PE) to message passing (2 PEs) slows the program down in the non-multithreading case, but since the compute/communicate ratio of Laplace is high (the computation of $O(n^2)$ array elements requires $O(n)$ remote references), parallelizing this code pays off and performance is regained. At the same time, multithreading is effective for Laplace as there are enough remote reference to cover the multithreading overheads, which gives rise to speedups of between 1.26 and 1.65. Clearly, multithreading is effective at tolerating the remote references for this program.

Table 7.20 gives the performance of Livermore Loop #7 with multithreading and Table 7.21 gives the performance of Livermore Loop #1 with multithreading, where the number of threads is 2 for all cases (to minimize overhead), *REff%* gives the relative efficiency within a group, and *Sp* gives the speedup due to multithreading. Both of these programs exhibit a very high degree of locality, and the mapping functions are successful in taking advantage of this locality, generating very few remote references. As a result, neither of these applications are affected by multithreading. In fact, the slight gains or losses seen in the tables are probably more attributable to noise than the actual effects of multithreading.

Table 7.22 gives the performance of parallel prefix with multithreading and Table 7.23 gives the performance of FFT with multithreading, where the number of threads is 16 for all cases (to maximize hits), *REff%* gives the relative efficiency within a group, and *Sp* gives the speedup due to multithreading. Both programs utilize one-dimensional arrays whose access patterns vary with each iteration, so that many remote references are gen-

		No Multithreading		*Multithreading*		
PEs	Array Size	Time (s)	REff (%)	Time (s)	REff (%)	Sp
1	65536	74.9313	100.0			
2	65536	38.5077	97.3	38.5364	97.2	1.00
4	65536	20.0551	93.4	20.0642	93.4	1.00
8	65536	10.7879	86.8	10.8078	86.7	1.00
4	262144	76.2464	100.0	76.2555	100.0	1.00
8	262144	38.8856	98.0	38.9049	98.0	1.00
16	262144	20.2864	94.0	20.3012	93.9	1.00
32	262144	11.1579	85.4	11.1727	85.3	1.00
16	1048576	76.4765	100.0	76.4846	100.0	1.00
32	1048576	39.2500	97.4	39.2721	97.4	1.00
64	1048576	21.1387	90.5	21.1599	90.4	1.00
128	1048576	13.1271	72.9	13.1485	72.7	1.00

Table 7.20: Performance of Livermore loop #7 with multithreading, 50 iterations

		No Multithreading		*Multithreading*		
PEs	Array Size	Time (s)	REff (%)	Time (s)	REff (%)	Sp
1	65536	32.2601	100.0			
2	65536	16.6137	97.1	16.6358	97.0	1.00
4	65536	8.8835	90.8	8.9120	90.5	1.00
8	65536	5.6399	71.5	5.1330	78.6	1.09
4	262144	33.0699	100.0	33.0953	100.0	1.00
8	262144	17.3766	95.2	17.3685	95.3	1.00
16	262144	9.3532	88.4	9.5229	86.9	0.98
32	262144	7.5177	55.0	7.3098	56.6	1.03
16	1048576	33.5528	100.0	33.7273	100.0	0.99
32	1048576	20.4575	82.0	19.5398	86.3	1.05
64	1048576	15.9578	52.6	15.5133	54.4	1.03
128	1048576	20.4108	20.5	20.0372	21.0	1.02

Table 7.21: Performance of Livermore loop #1 with multithreading, 50 iterations

PEs	Array Size	*No Multithreading*		*Multithreading*		
		Time (s)	REff (%)	Time (s)	REff (%)	Sp
1	65536	10.9807	100.0			
2	65536	65.3893	8.3	41.0189	13.1	1.59
4	65536	54.7874	4.9	40.2863	7.0	1.36
8	65536	36.8320	3.7	27.3209	4.9	1.35
4	262144	220.2234	100.0	162.1199	100.0	1.36
8	262144	147.3423	74.7	109.3570	74.1	1.35
16	262144	92.3348	59.6	68.9983	58.7	1.34
32	262144	55.9987	49.1	42.0135	48.2	1.33
16	1048576	369.9710	100.0	276.2995	100.0	1.34
32	1048576	222.2807	83.2	166.7407	82.9	1.33
64	1048576	130.6187	70.8	98.3616	70.2	1.33
128	1048576	76.3341	60.6	57.9285	59.6	1.32

Table 7.22: Performance of parallel prefix with multithreading

PEs	Array Size	*No Multithreading*		*Multithreading*		
		Time (s)	REff (%)	Time (s)	REff (%)	Sp
1	4096	8.0910	100.0			
2	4096	39.1992	10.3	33.1723	12.2	1.18
4	4096	36.4452	5.6	31.7575	6.4	1.15
8	4096	27.9294	3.6	23.8901	4.2	1.17
4	8192	85.4928	100.0	70.0730	100.0	1.22
8	8192	81.5629	52.4	67.7769	51.7	1.20
16	8192	77.8649	27.4	65.8737	26.6	1.18
32	8192	72.6381	14.7	63.2700	13.8	1.15
16	16384	190.0012	100.0	158.8733	100.0	1.20
32	16384	177.2432	53.6	149.3392	53.2	1.19
64	16384	168.7034	28.2	144.3667	27.5	1.17
128	16384	162.1386	14.6	137.5489	14.4	1.18

Table 7.23: Performance of FFT with multithreading

		No Multithreading		*Multithreading*		
PEs	Matrix Size	Time (s)	REff (%)	Time (s)	REff (%)	Sp
1	64x64	2.9211	100.0			
2	64x64	80.3168	1.8	63.2268	2.3	1.27
4	64x64	72.4649	1.0	69.7533	1.1	1.04
8	64x64	46.4425	1.0	45.7500	1.0	1.02
4	128x128	581.9276	100.0	454.9914	100.0	1.28
8	128x128	345.1970	84.3	328.6310	69.2	1.05
16	128x128	231.3354	62.9	226.9965	50.1	1.02
32	128x128	170.6795	42.6	166.4451	34.2	1.02
16	256x256	1625.3946	100.0	1430.3472	100.0	1.12
32	256x256	1051.7490	77.3	1009.6790	70.8	1.04
64	256x256	837.5011	48.5	829.1261	43.1	1.01
128	256x256	1248.3309	16.3	1242.8650	14.4	1.00

Table 7.24: Performance of matrix multiply with multithreading

		No Multithreading		*Multithreading*		
PEs	Matrix Size	Time (s)	REff (%)	Time (s)	REff (%)	Sp
1	32x32	15.4610	100.0			
2	32x32	32.5118	23.8	31.1058	24.9	1.05
4	32x32	33.4192	11.6	32.3402	12.0	1.03
8	32x32	36.5835	5.3	36.5824	5.3	1.00
4	64x64	216.2966	100.0	213.2748	100.0	1.01
8	64x64	221.6448	48.8	219.0985	48.7	1.01
16	64x64	229.6827	23.5	228.3755	23.3	1.00
32	64x64	271.6236	10.0	271.5913	9.8	1.00

Table 7.25: Performance of Cholesky, with multithreading

erated regardless of the original data distribution (i.e., poor locality). For parallel prefix, multithreading does a good job of tolerating the remote references, resulting in an average speedup of 1.36 over the non-multithreading cases. The tendency for the speedup to decrease as the number of processors is increased is due to the increase in task management overheads rather than a decrease in the multithreading efficiency. Multi-level distribution attempts to re-capture some of the lost performance, but cannot remove the decrease altogether.

Table 7.24 gives the performance of matrix multiply with multithreading and Table 7.25 gives the performance of Cholesky with multithreading, where the number of threads is 16 for all cases (to maximize hits), *REff%* gives the relative efficiency within a group, and *Sp* gives the speedup due to multithreading. Although the speedups due to multithreading are negligible, it is not because the multithreading is ineffective at tolerating the remote references, but that the overriding costs in these programs is in managing the Sisal two-dimensional data structures. In other words, the effects of multithreading are dwarfed by the overheads of two-dimensional data structures. As seen with the Laplace example, when these overheads are eliminated, the effects of multithreading are much more visible.

The multithreading results lead to the following conclusions:

1. When the number of remote references is very low, due to the ability of the mapping functions and loop distribution techniques to exploit locality, the percentage of total time due to remote references is very low, and so multithreading will have little impact one way or another. When the number of remote references is extremely small, the costs of multithreading are typically not re-captured.

2. When there are a sufficient number of remote references, which account for a substantial portion of the total execution time, multithreading is effective at tolerating much of the latency resulting from these remote references, resulting in a speedup over the non-multithreading case averaging between 1.20 and 1.40.

3. When the total execution time is dominated by the overhead of administering Sisal two-dimensional data structures, the effects of multithreading are inconsequential. The much larger problem is to efficiently handle multi-dimensional data structures.

Chapter 8

CONCLUSIONS AND FUTURE DIRECTIONS

The future's so bright, I gotta wear shades!
– Timbuckthree

We have presented the design and implementation of a distributed runtime system which provides support for task management and memory management, such as parallel task distribution, data distribution across a single addressing space, and multithreading. Using this system as a basis for supporting Sisal on a distributed memory multiprocessor, we have evaluated the various designs in our system in general, and the ability of a runtime system to avoid and tolerate remote memory references in particular.

8.1 Summary of Results

We found that the runtime system can be effective in controlling the distribution of tasks among a wide variety of processor configurations. In particular a multi-level distribution scheme is effective at reducing the overheads associated with distributing work to a large number of processors. Also, in the case of nested loops, adapting the amount of parallelism in loops to the parallelism in the machine is necessary to control the tradeoff of parallelism versus overhead.

We found that providing a single addressing space to the compiler can result in large overheads for address translation, but that certain optimizations can minimize this overhead, particularly for local references where the overhead is most visible. We compared our variable-blocksize address translation scheme, which allows us to align the data distribution with the task distribution so that unnecessary remote references are minimized, to a fixed-blocksize address translation scheme. Although the fixed-blocksize translation scheme eliminates the need for fetching a data structure descriptor, which is necessary in the variable-blocksize translation scheme, it is difficult to keep the data aligned with the tasks, resulting in excessive remote references. The variable-blocksize address translation scheme typically keeps the data in alignment with the tasks, resulting in performance comparable to the optimal fixed blocksize. Furthermore, optimizing the address translation process to account for local references results in a substantial improvement over both the fixed-blocksize and unoptimized variable-blocksize address translation schemes.

We found that, through a combination of aligning tasks with data, parallelizing the distribution of tasks to a large number of processors, and tolerating the unavoidable remote references using multithreading, we are able to efficiently execute a variety of programs that use one-dimensional data structures exclusively.

We found that, in comparing explicit programming styles to implicit programming styles for distributed memory multiprocessors, there is a tradeoff of performance for programming effort. Specifically, for our small examples, pure implicit programming offers reasonable performance while alleviating the programmer from the implementation details of the target architecture. A hybrid of explicit and implicit programming can be used to increase performance by an average speedup of 1.34 over the purely implicit style, but only at a cost of increasing the lines of code by an average factor of seven and increasing the time required to encode and debug the problem by an average factor of 11. Finally, using a purely explicit style, we can increase performance to obtain an average speedup of 1.74 over the hybrid style and 2.34 over the purely implicit style, but at a cost of increasing the program size by a factor of two over the hybrid style and a factor of 15 over the purely implicit style and increasing the programming time by an average factor of four over the hybrid style and a factor of 40 over the purely implicit style. Therefore, although it is possible to increase the performance of an application by lowering the programming abstraction, this increase in performance often comes at a substantial cost to programming effort. Also, the performance of the implicit style can be improved through multithreading and increased compiler support.

We found that two-dimensional arrays in Sisal generate an excessive number of descriptor data structures, and the management of all these structures at runtime is prohibitive. In fact, the overhead caused by these two-dimensional data structures dominates the execution time of our two-dimensional programs to the extent that the effects of data distribution and multithreading are not visible. However, if true rectangular arrays were supported, the overhead of two-dimensional data structures is greatly reduced, and the effects of multithreading become visible.

We found that multithreading can be effective at tolerating the latency due to remote references when the number of remote references is large enough to cover the cost of multithreading and the effect of the remote references on the overall execution time of the program is noticeable. For this case, multithreading yields a speedup over the non-multithreading case averaging between 1.2 and 1.4. If either the remote references are too few to have an impact on the performance of the program, or other factors, such as two-dimensional data structure manipulation, hide the effects of the remote references, then multithreading is ineffective.

8.2 Future Work

In this section we examine the issues that need to be addressed in providing software support for latency avoidance and latency tolerance.

- **Compiler support**. Although we have created a robust runtime system that is capable of executing code produced by the shared memory Sisal compiler on a distributed memory multiprocessor, it is often the case that when the compiler ignores the presence of a distributed memory hierarchy, the code it generates will result in dismal performance. Below we present a list of items that need to be addressed by a compiler generating code for a distributed memory multiprocessor with underlying runtime support such as provided by our system.

  - *Support for the runtime system primitives.* Since the current Sisal compiler does not know of the VISA runtime system, all of the VISA primitives (listed in Appendix A) must be inserted by hand. While this is an algorithmic procedure that would not be difficult for a code generator to perform, it has nonetheless precluded us from studying large problems, due to the time required to insert the primitives by hand.

- *Support for true rectangular arrays.* We have shown that, for a distributed memory multiprocessor, implementing Sisal two-dimensional arrays in the current manner is prohibitive, and that rectangular arrays can improve the performance greatly. Therefore, before any large problems can be efficiently supported, true rectangular arrays must be supported at the compiler level.
- *Support for function call parallelism.* Though the Sisal language exposes function level parallelism, the current Sisal compiler does not exploit this level of parallelism. For example, a recursive version of FFT would expose a great deal of function level parallelism, but we are forced to use an iterative version of FFT to expand the parallel loops.
- *Support for the multithreading paradigm.* The Sisal compiler currently generates code for a very limited parallel tasking system, though we have been able to add considerable flexibility through runtime support. The multithreading system currently uses this same tasking paradigm, simply dividing each given slice into a number of threads. Since the current compiler generates slice code in such a way that the input parameters are fetched after the slice has begun, each thread on the same processor must fetch these input parameters independently. Clearly this creates an unnecessary overhead for the multithreading system that could be eliminated if the compiler generated slice code in two sections, one for gathering input parameters and another for performing the actual computations. In this manner, the threads could divide after the common parameters have been fetched.
- *Improved analysis.* We have provided runtime support for a number of parameters in a distributed memory implementation, although the intelligence to determine the best configuration of the various options has been left to the programmer to determine using runtime switches. The goal of raising the programmer from this level of detail must be addressed by the compiler in performing analysis that will attempt to compute the optimal settings for the various capabilities. Current research in the area of compiler analysis for distributed memory architectures is abundant, including Sisal-based analysis for data distribution [OH92] and task partitioning [WF92], although neither of these analyses have been actually incorporated into the current Sisal compiler. Also, for regular problems whose access pattern can be determined at compile time, the compiler should be able to help the VISA system by optimizing all local references so that address translation can be avoided, and possibly even setting up a synchronous communication model to fetch known remote references.

- **Support for vectorized messages and pre-fetching**. Perhaps one of the largest costs in the VISA system is the time required for a remote reference, due mainly to the address translation process for both the requesting and replying processes, and the message startup time. A significant improvement can be made by amortizing the translation and startup costs over a large number of remote references that are to be obtained from the same processor, such as an entire matrix row. By supporting vectorized messages and the ability to control pre-fetching these blocks of data, we hope to drastically improve the performance of programs such as matrix multiply.

- **Support for re-mapping.** For programs such as parallel prefix and FFT, where the access pattern changes over some iteration space in the program, it may be beneficial to re-map the data to re-capture local references. For example, in the parallel prefix algorithm, the distance between accesses increases by a factor of two for each of the $lg(n)$ iterations over an array of n elements. When the distance exceeds the blocksize, all references will be remote. At this point it may be beneficial to re-distribute the data, using the vectorized messages described above, and change the address translation maps in such a way as to effectively return the distance to one. Although we intend to provide the mechanism for doing this soon, the harder question is the analysis required to determine when re-mapping is worth performing, since it involves moving (potentially) all of the data and re-broadcasting the new range map entries.

- **Support for other message passing abstractions.** Currently, VISA operates atop an interrupt-driven message passing abstraction that is found on the nCUBE/2 and other distributed memory multiprocessors, such as the Intel Paragon. However, new abstractions, such as provided by the *T architecture, promise to provide very low-level interfaces to the interconnection network, which can be tuned for optimal performance by the VISA system.

- **Support for other languages.** The VISA system currently supports the Sisal compiler for execution on a distributed memory multiprocessor, although the actual VISA primitives are language independent. We hope to explore the option of creating different pathways that lead to VISA from other languages, such as C++, in which the VISA primitives and capabilities would be wrapped into a class definition which controls the access functions.

Given this long list of future directions, we have selected two items that we will pursue first. We will implement support for the vectorized messages and pre-fetching operators. Once this is complete, we will implement a re-mapping capability. However, neither of these improvements will help the Sisal efficiency until compiler support is in place, so we will be working with the Sisal group at Lawrence Livermore National Laboratories in guiding the compiler support.

8.3 Summary

We have introduced our project, which is to study latency avoidance and latency tolerance for an implementation of the implicitly parallel programming language Sisal on distributed memory multiprocessors. To that end we have developed a runtime system that provides support for both task management and memory management. We have outlined the designs of this system, and presented performance results using a variety of sample programs designed to test the various aspects of these designs. We have analyzed the results to determine which methods are successful, which are not, and what can be done to change this.

The need for scalable and portable parallelism is becoming increasingly important in the computing and computational science communities. We must develop software that will allow applications to exploit the speed and power present in today's and tomorrow's large distributed systems. It is our view that we must shield the programmer from the details of programming distributed memory multiprocessors, but not at the expense of performance. Clearly this is a challenging goal.

REFERENCES

[AAC+91] Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allen Porterfield, and Burton Smith. Exploiting heterogeneous parallelsim on a multithreaded multiprocessor. Technical report, Tera Computer Company, Seattle, WA, 1991. Workshop on Multithreaded Computers, Albuquerque, NM, 1991.

[AC86] Arvind and David E. Culler. Dataflow architectures. *Annual Reviews of Computer Science*, 1:225–253, 1986.

[AE89] D. Abramson and G. K. Egan. Implementation of the RMIT/CSIRO Dataflow Machine - CSIRAC. In *Proc. Data-flow Computing : A Status Report*, 1989.

[AG89a] G. Almasi and A. Gottlieb. *Highly Parallel Computing*, chapter 8: Interconnection Networks, pages 279–300. Benjamin/Cummings, Redwood City, CA, 1989.

[AG89b] G. S. Almasi and A. Gottlieb. *Highly Parallel Processing Computing*, pages 193–194. Benjamin/Cummings, Redwood City, 1989. The Rings of Id.

[AGP78] Arvind, K. Gostelow, and W. Plouffe. An asynchronous programming language and computing machine. Technical Report 114a, University of California, Irvine, CA, December 1978.

[AI87] Arvind and Robert A. Iannucci. Two fundamental issues in parallel processing. Technical Report Computation Structures Group Memo 226-6, Massachusetts Institute of Technology, May 1987.

[ANP89] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-Structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4), 1989.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[Bac78] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System.* Software Series. Prentice-Hall, 1986.

[BCFO91] A. P. W. Böhm, D. C. Cann, J. T. Feo, and R. R. Oldehoeft. SISAL 2.0 reference manual. Technical Report CS-91-118, Computer Science Department, Colorado State University, Fort Collins, CO, November 1991.

[BCK$^+$89] M. Berry, D. Chen, P. Koss, D. Kuck, L. Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The Perfect Club Benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, 3(5):5–40, Fall 1989.

[BCZ89] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Shared memory for distributed memory multiprocessors. Technical Report Rice COMP TR89-91, Rice University, April 1989.

[BE92] William Blume and Rudolf Eigenmann. Performance analysis of parallelizing compilers on the Perfect Benchmarks Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[Bec92] Michael J. Beckerle. An overview of the START(*T) computer system. Technical Report MCRC-TR-28, Motorola Cambridge Research Center, October 1992. Revision 2.

[BK85] Bob Beck and Bob Kasten. VLSI assist in building a multiprocessor UNIX system. In *Proceedings of Summer Usenix*, pages 257–275, 1985.

[BNA91] Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-Structures: Extending a parallel, non-strict, functional language with state. Technical Report Computation Structures Group Memo 327, Massachusetts Institute of Technology, March 1991.

[BR90] Roberto Bisiani and Mosur Ravishankar. PLUS: A distributed shared-memory system. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 115–124, May 1990.

[BS89] A. P. W. Böhm and J. Sargeant. Code optimization for tagged-token dataflow machines. *IEEE Transactions on Computers*, pages 4–14, January 1989.

[BT88a] Henri E. Bal and Andrew S. Tanenbaum. Distributed programming with shared data. Technical Report IR-149, Vrije Universiteit, Amsterdam, March 1988.

[BT88b] A. P. W. Böhm and Y. M. Teo. Resource Management in a Multi-Ring Dataflow Machine. In *Proceedings International Conference on Parallel Architecure (COMPAR), UMIST*, 1988.

[Can89] D. C. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, Computer Science Department, Fort Collins, CO, 1989.

[Can92] David Cann. Retire Fortran? A debate rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[CF78] Lucien M. Censier and Paul Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.

[CGH89] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: A language for parallel programming. Technical Report CSL-TR-89-396, Stanford University, October 1989.

[Cho90] Shyamal Chowdhury. The greedy load sharing algorithm. *Journal of Parallel and Distributed Computing*, 9:93–99, 1990.

[CLR91] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Electrical Engineering and Computer Science Series. The MIT Press and McGraw-Hill, 1991.

[CO88] D. C. Cann and R. R. Oldehoeft. Porting multiprocessor SISAL software. Technical Report CS-88-104, Computer Science Department, Colorado State University, Fort Collins, CO, May 1988.

[CSS$^+$91] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In 4^{th} *International Conf. on Architectural Support for Programming Languages and Operating Systems*, 1991.

[Cul90] David E. Culler. Managing parallelism and resources in scientific dataflow programs. Thesis MIT/LCS/TR-446, MIT, March 1990.

[DB82] Michel Dubois and Fayé A. Briggs. Effects of cache coherency in multiprocessors. *IEEE Transactions on Computers*, C-31(11):1083–1099, November 1982.

[DCF$^+$89] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Lethin, Peter Nuth, Scott Wills, Paul Carrick, and Greg Fyler. The J-Machine: A fine-grain concurrent computer. In *Proceedings of Information Processing 89*, Amsterdam, 1989. North Holland.

[Del88] G. Delp. *The Architecture and Implementation of MemNet: A High-Speed Shared Memory Computer Communication Network.* PhD thesis, University of Delaware, 1988.

[Den80] Jack B. Dennis. Data flow supercomputers. *IEEE Computer*, 13(11):48–56, November 1980.

[DSB86] Michel Dubois, Christoph Scheurich, and Fayé A. Briggs. Memory access buffering in multiprocessors. In *The 13th International Symposium on Computer Architecture*, pages 434–442, June 1986.

[FCO90] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, December 1990.

[Feo87] J. T. Feo. The Livermore Loops in SISAL. Technical Report UCID-21159, Lawrence Livermore National Laboratory, Livermore, CA, August 1987.

[FHK$^+$90] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[Fis87] J. Fisher. Wide instruction word architectures: Solving the supercomputer software problem. In *Proceedings of the 1987 International Seminar on Scientific Supercomputers*, 1987.

[Fra87] D. Fram. The BBN advanced computers butterfly parallel processor: A MIMD computer for simulation of complex systems. In *Proceedings of the Third Conference on Multiprocessors and Array Processors*, pages 137–146, January 1987.

[GB90] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. University of Illinois at Urbana-Champaign Technical Report CRHC-90-14, Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, October 1990.

[GB92] Manish Gupta and Prithviraj Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[Got91] Allan Gottlieb. An outsider's view of data-flow. In Jean-Luc Gaudiot and Lubomir Bic, editors, *Advanced Topics in Data-flow Computing*, chapter 22, pages 573–581. Prentice Hall, 1991.

[Gra91] Grand challenges: High performance computing and communication. To Supplement the President's Fiscal Year 1992 Budget, Office of Science and Technology, Washington D.C., 1991.

[Gri90] D. H. Grit. A distributed memory implementation of SISAL. In *Proceedings of the Fifth Distributed Memory Computing Conference*, April 1990.

[HA90] P. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[HB92] Matthew Haines and Wim Böhm. On the design of distributed memory Sisal. Technical Report CS-92-144, Colorado State University, Fort Collins, CO, January 1992.

[HKT92] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[HMS+86] John P. Hayes, Trevor N. Mudge, Quentin F. Stout, Stephen Colley, and John Palmer. Architecture of a hypercube supercomputer. In *Proceedings of the International Conference on Parallel Processing*, pages 653–660, August 1986.

[IFKF90] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[Int91] Intel iPSC/860 specifics. Brochure, 1991.

[Ken92] Kendell Square Research Corperation, Waltham, Massachusetts. *Technical Summary*, 1992.

[KM91] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Computing*, 2(4):440–451, October 1991.

[KMRS86] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive snoopy caching. Technical Report CMU-CSD-86-164, Department of Computer Science, Carnegie Mellon University, 1986.

[KS88] J. Kuehn and B. Smith. The Horizon supercomputing system: Architecture and software. In *Proceedings of Supercomputing*, pages 28–34, Orlando, Florida, November 1988.

[LG91] Jenq Kuen Lee and Dennis Gannon. Object oriented parallel programming experiments and results. In *Proceedings of Supercomputing 91*, pages 273–282, November 1991.

[Li86] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, September 1986.

[LLD$^+$83] Paul J. Leach, Paul H. Levine, Bryan P. Douros, James A. Hamilton, David L. Nelson, and Bernard L. Stumpf. The architecture of an integrated local network. *IEEE Journal on Selected Areas in Communications*, 1:842–857, November 1983.

[LP91] Zakaria Lahjomri and Thierry Priol. KOAN: A shared virtual memory for the iPSC/2 hypercube. Technical Report 597, Campus de Beaulieu, July 1991.

[LS89] Kai Li and Richard Schaefer. A hypercube shared virtual memory system. *Proceedings of the 1989 International Conference on Parallel Processing*, 1:125–131, 1989.

[LSF88] C. Lee, S. Skedzielewski, and J. Feo. On the implementation of applicative languages on shared-memory, MIMD multiprocessors. In *Proceedings of the Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, pages 188–197, New Haven, CT, September 1988. Available in *SIGPLAN Notices* 23, 9.

[LSWZ89] Kai Li, Michael Stumm, David Wortman, and Songnian Zhou. Shared virtual memory accommodating heterogeneity. Technical Report CS-TR-210-89, Princeton University, February 1989.

[MF90] Ronald G. Minnich and David J. Farber. Reducing host load, network load, and latency in a distributed shared memory. In *10th International Conference on Distributed Computing Systems*, pages 468–475, May 1990.

[Mit92] S. Mitrovic. An IF2 code-genrator for ADAM architecture. In J.T. Feo, C. Frerkling, and P.J. Miller, editors, *Proceedings of the Second Sisal User's Conference*, pages 93–109, San Diego, CA, October 1992.

[MS82] J. R. McGraw and S.K. Skedzielewski. Streams and iteration in VAL: Additions to a data flow language. In *Proceedings of the Third International Conference on Distributed Computing Systems*, Miami/Ft. Lauderdale, Florida, March 1982.

[MSA+85] J. R. McGraw, S. K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. Kirkham, W. 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.

[nCU90a] nCUBE, Beaverton, OR. *nCUBE 2 Programmer's Guide*, December 1990.

[nCU90b] nCUBE, Beaverton, OR. *Technical Overview, SYSTEMS*, 1990.

[Nik87] R. Nikhil. Id nouveau: Quick reference guide. Technical report, MIT Laboratory for Computer Science, Cambridge, MA, January 1987.

[Nik90] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo 284-1, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 1990. Supercedes: Id/83s (July 1985) Id Nouveau (July 1986), Id 88.0 (March 1988), Id 88.1 (August 1988).

[OBAAD91] Özalp Babaoğlu, Lorenzo Alvisi, Alessandro Amoroso, and Renzo Davoli. Paralex: An environment for parallel programming in distributed systems. Technical Report UB-LCS-91-01, University of Bologna, February 1991.

[OH92] Michael O'Boyle and G.A. Hedayat. Data alignment: Transformations to reduce communication on distributed memory architectures. In *Proceedings of the Scalable High Performance Computing Conference*, pages 366–371, April 1992.

[PB90] Cherri M. Pancake and Donna Bergmark. Do parallel languages respond to the needs of scientific programmers. *IEEE Computer*, 23(12):13–24, December 1990.

[PC90] G. M. Papadopolus and D. E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the* 17^{th} *Annual International Symposium on Computer Architecture*, June 1990.

[PK87] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[PS85] James L. Peterson and Abraham Silbershatz. *Operating System Concepts*. Addison-Wesley, second edition, 1985.

[PW86] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[RA90] R. Ruhl and M. Annaratone. Parallelization of Fortran code on distributed-memory parallel processors. In *International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[RAK88] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Unifying synchronization and data transfer in maintaining coherence of distributed shared memory. Technical Report GIT-CS-88/23, Georgia Institute of Technology, June 1988.

[Ree90] A. Reeves. The Paragon programming paradigm and distributed memory compilers. Technical Report EE-CEG-90-7, Cornell University Computer Engineering Group, Ithaca, NY, June 1990.

[Ric85] John R. Rice. Problems to test parallel and vector languages. Technical Report CSD-TR-516, Purdue University Computer Science Department, May 1985.

[Rob92] 1992. Personal Communication with Rob Pike, AT&T Laboratories.

[RS87] C. A. Ruggiero and J. Sargeant. Control of Parallelism in the Manchester Dataflow Computer. In *Lecture Notes in Computer Science no. 274*, pages 1–15, 1987.

[RTY$^+$87] Richard Rashid, Avadis Tevanian, Michael Young, David Young, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31–39, October 1987.

[RYYT89] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer, design philosophies, decisions and trade-offs. *IEEE Computer*, pages 12–35, January 1989.

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. The MIT Press, 1989. Research Monographs in Parallel and Distributed Computing.

[SCMB90] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[SG85] S. K. Skedzielewski and J. Glauert. IF1—an intermediate form for applicative languages. Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985.

[SLY90] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An emperical study of Fortran programs for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[SMC90] A. Sane, K. MacGregor, and R. Campbell. Distributed virtual memory consistency protocols: Design and performance. In *Second IEEE Workshop on Experimental Distributed Systems*, pages 91–96, October 1990.

[Smi85] Burton Smith. The architecture of HEP. In J. Kowalik, editor, *Parallel MIMD Computation: The HEP Supercomputer and Its Applications*, pages 41–55. MIT Press, Cambridge, MA, 1985.

[SSS88] Charles L. Seitz, Jakov Seizovic, and Wen-King Su. The C programmer's abbreviated guide to multicomputer programming. Technical Report Caltech-CS-TR-88-1, Caltech, January 1988.

[SYH$^+$89] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single chip processor. In *Int. Symp. on Computer Architecture*, pages 46–53. IEEE, 1989.

[SYHY87] S. Sakai, Y. Yamaguchi, K. Hiraki, and T. Yuba. Introduction of a Strongly-Connected-Arc-Model in a Data-Driven Single Chip Processor EMC-R. In *Data-flow Workshop, IECE*, pages 231–238, 1987. Japanese.

[Tan87] Andrew S. Tanennaum. *Operating Systems: Design and Implementation.* Software Series. Prentice-Hall, 1987.

[TF90] Ivan Ming-Chit Tam and David J. Farber. CapNet–an alternative approach to ultra high speed networks. In *Proceedings, International Communications Conference*, 1990.

[Thi91] Thinking Machines Corporation, Cambridge, Massachusetts. *CM5 Technical Summary*, October 1991.

[Tra86] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical report, MIT Laboratory for Computer Science, August 1986.

[Tra91] K. R. Traub. *Implementation of Non-Strict Functional Programming Languages.* Research Monographs in Parallel and Distributed Computing. MIT Press, 1991.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communications and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[WF92] R. Wolski and J. Feo. An extended data flow model for program partitioning on NUMA architectures. In *Proceedings of the Second Sisal User Conference*, October 1992.

[WFC91] R. Wolski, J. Feo, and D. C. Cann. A prototype functional language implemenation for hierarchical-memory architectures. Technical Report UCRL-JC-107437, Lawrence Livermore National Laboratory, June 1991.

[WY88] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of Object Oriented Programming Systems, Languages, and Applications*, pages 306–314, September 1988.

[YSH$^+$89] Y. Yamaguchi, S. Sakai, K. Hiraki, Y. Kodama, and T. Yuba. An Architectural Design of a Highly Parallel Data-flow Machine. In *IFIP*, pages 1155–1160, 1989.

[ZBG86] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1986.

Appendix A

VISA FUNCTIONS

- *Allocation*
  - V_ADDRESS **visa_malloc** (int *nelems*, int *size*,
    map_function *map*, int *map_arg*)

    This function allocates a block of VISA space (*nelems* * *size* bytes), which will be distributed according to *map*, and returns a pointer to the start of the allocated space. A range_map entry is also created and distributed among the nodes, and local space is allocated, according to the map, to store the data structure.
- *Deallocation*
  - void **visa_free** (V_ADDRESS *address*)

    This function returns the given portion of VISA space to the free pool, removes the corresponding range_map entry from each of the range_map tables, and deallocates the local storage used for storing the structure.
- *Access*
  - range_map_type * **find_rm** (V_ADDRESS *address*)

    Return a pointer to the range_map entry corresponding to the given VISA address. This pointer is then passed into each of the access routines as an argument so that the fetch does not have to be done for each access.
  - char **visa_get_c** (V_ADDRESS *address*, range_map_type **rm*)
    int **visa_get_i** (V_ADDRESS *address*, range_map_type **rm*)
    float **visa_get_f** (V_ADDRESS *address*, range_map_type **rm*)
    double **visa_get_d** (V_ADDRESS *address*, range_map_type **rm*)

    These functions return the desired value from the given VISA address. If the range_map entry *rm* is not defined, then the corresponding range_map entry for this structure will be fetched, which is true for all of the access functions.
  - void **visa_get_m** (POINTER *data*, int *size*, V_ADDRESS *address*,
    range_map_type **rm*)

    This function copies the block of data starting at the given VISA address and for a length of *size* into the local address pointed to by *data*.
  - void **visa_put_c** (char *value*, V_ADDRESS *address*, range_map_type **rm*)
    void **visa_put_i** (int *value*, V_ADDRESS *address*, range_map_type **rm*)
    void **visa_put_f** (float *value*, V_ADDRESS *address*, range_map_type **rm*)
    void **visa_put_d** (double *value*, V_ADDRESS *address*, range_map_type **rm*)

    These functions place *value* into the given VISA address location.

- void **visa_put_m** (POINTER *data*, int *size*, V_ADDRESS *address*, range_map_type **rm*)

  This function copies the local data block of size *size* and pointed to by *data* into the given VISA address location.

- void **visa_update_c** (uchar *red*, char *value*, V_ADDRESS *address*, range_map_type **rm*)
  void **visa_update_i** (uchar *red*, int *value*, V_ADDRESS *address*, range_map_type **rm*)
  void **visa_update_f** (uchar *red*, float *value*, V_ADDRESS *address*, range_map_type **rm*)
  void **visa_update_d** (uchar *red*, double *value*, V_ADDRESS *address*, range_map_type **rm*)

  These functions update the value stored in the given VISA address with *value*, according to the reduction *red*. Currently supported reductions include *V_SUM* and *V_PRODUCT*.