THE DESIGN, IMPLEMENTATION AND EVALUATION OF JADE: A PORTABLE, IMPLICITLY PARALLEL PROGRAMMING LANGUAGE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Martin C. Rinard
September 1994

© Copyright 1994 by Martin C. Rinard
All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Monica Lam
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

John Hennessy

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Anoop Gupta

Approved for the University Committee
on Graduate Studies:

Abstract

Over the last decade, research in parallel computer architecture has led to the development of many new parallel machines. These machines have the potential to dramatically increase the resources available for solving important computational problems. The widespread use of these machines, however, has been limited by the difficulty of developing useful parallel software. This thesis presents the design, implementation and evaluation of Jade, a new programming language for parallel computations that exploit task-level concurrency.

Jade is structured as a set of constructs that programmers use to specify how a program written in a standard sequential, imperative language accesses data. The implementation dynamically analyzes these specifications to automatically extract the concurrency and map the computation onto the parallel machine. The resulting parallel execution preserves the semantics of the original serial program.

We have implemented Jade on a wide variety of parallel computing platforms: shared-memory multiprocessors such as the Stanford DASH machine, homogeneous message-passing machines such as the Intel iPSC/860, and on heterogeneous networks of workstations. Jade programs port without modification between all of these platforms.

We evaluated the design and implementation of Jade by parallelizing several complete scientific and engineering applications in Jade and executing these applications on both shared-memory and message-passing computational platforms. Our evaluation focuses on two properties of Jade: how well it supports the process of developing parallel programs and how well the resulting programs perform. We found that Jade does not usually impose an onerous programming burden - the vast majority of the changes required to parallelize the applications were confined to small, peripheral sections of the code and did not perturb the overall structure of the program. The coarse-grain applications perform very well, exhibiting

almost linear speedup up to 32 processors on both computational platforms. Several of the finer-grain applications suffer from some Jade-specific performance problems, but some of these problems could be eliminated with a more advanced Jade implementation.

Acknowledgments

During the course of this research I benefited enormously from the knowledge, expertise and commitment of my colleagues. My advisor, Monica Lam, had a profound impact on my personal and professional development. Her constant refusal to accept anything less than my absolute best forced me to grow as a researcher and allowed me to experience the joy of discovering my own hidden abilities and talents. John Hennessy and Anoop Gupta took an interest in Jade from the beginning. Their support inspired me to explore the basic concepts and practical impact of my ideas; their feedback helped to keep the project focused on the important issues.

Dan Scales and Jennifer Anderson played an important role in the development of Jade. Their insightful contributions in early design meetings helped to elucidate the key principles on which Jade is based. Dan also made a major contribution to the implementation effort by building the Jade front end.

Mark van Schaack, Caroline Lambert, Brian Schmidt, Jun Ye, Ray Browning, Jason Nieh, Ed Rothberg, Dan Scales and Jennifer Anderson all helped to develop the Jade applications discussed in this thesis. Their efforts enhanced my understanding of the practical implications of using Jade.

Many people helped me deal with the vagaries of the different computational environments. Dave Nakahira and Jonathan Chew made sure DASH stayed up; Mark Heinrich showed me how to use the DASH performance monitor. Dan Yergeau kept the iPSC/860 up and running. Duane Northcutt and Jim Hanko at Sun Labs made it possible for Jade to run on the High Resolution Video Machine while David Chenevert, also at Sun Labs, enabled me to use the Mica multiprocessor.

I would like to thank my colleagues in the DASH and SUIF groups for creating a

wonderful research environment. The energy and ability of the people in these two groups made my years in graduate school some of the most exciting and stimulating of my life. I particularly enjoyed my interaction with Saman Amarasinge, Jennifer Anderson, Rohit Chandra, Kourosh Gharachorloo, Aaron Goldberg, Mary Hall, John Heinlein, Amy Lim, Margaret Martonosi, Dave Ofelt, Kunle Olukotun, Charlie Orgish, Ed Rothberg, Margaret Rowland, Dan Scales, J.P. Singh, Vijayaraghavan Soundararajan and Chau-Wen Tseng. Tom Rokicki, my roomate of many years, and several people who entered graduate school with me, Craig Chambers, Edith Cohen, Tom Henzinger and Alex Wang, helped to keep my life interesting and rewarding.

Early in my graduate career I wrote several papers with Vijay Saraswat. I am grateful to Vijay for introducing me to the research and publication process and showing me how rewarding it can be. When I first came to Stanford I worked with Vaughan Pratt. I admire Vaughan's unique understanding of concurrency and am grateful to Vaughan for sharing it with me. Experiencing Vaughan's perspective gave me a framework for understanding concurrency that helped me enormously during the initial design of Jade.

Finally, I would like to thank Ann McLaughlin for her support, companionship and love. She has been with me throughout the whole process, and I cannot imagine what it would have been like without her.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Existing parallel machines present two fundamentally different programming models: the shared-memory model [83, 52, 73] and the message-passing model [130, 68, 131]. Even machines that support the same basic model of computation may present interfaces with significantly different functionality and performance characteristics. Developing the same computation on different machines may therefore lead to radically different programs [126, 114], and it can be difficult to port a program written for one machine to a machine with a substantially different programming interface [89].

Given the rapid development of parallel machines, this lack of portability presents a significant obstacle to the widespread use of parallel computation. Organizations that develop or purchase software will be reluctant to invest in parallel software if it will not easily port to new architectures.

A second problem is that the software must manage many of the low-level aspects associated with mapping a computation onto the parallel machine. For example, the software must decompose the program into parallel tasks and assign the tasks to processors for execution. For the program to execute correctly, the software must generate the synchronization operations that coordinate the execution of the computation. On message-passing machines the software must also generate the message-passing operations required to move data through the machine.

For some parallel programs with simple concurrency patterns the applications programmer can generate this management code without too much difficulty, and its direct

incorporation into the source code does not significantly damage the structure of the program. In general, however, an explicitly parallel programming environment complicates the programming process and can impair the structure and maintainability of the resulting program. To generate correct synchronization code, the programmer must develop a global mental model of how all the parallel tasks interact and keep that model in mind when coding each task. The result is a decentralized concurrency management algorithm scattered throughout the program. To function effectively, a new programmer attempting to maintain the program must first reconstruct, then understand both the global synchronization algorithm and the underlying mental model behind the algorithm. Explicitly parallel environments therefore destroy modularity because they force programmers to understand the dynamic behavior of the entire program, not just the module at hand.

The need for efficient execution compounds the complexity of the program development process. Performance degradation may come from the dynamic overhead of the concurrency management code, excess communication, poor load balancing or a lack of concurrency. Efficient execution may therefore require elaborate, highly optimized concurrency management algorithms or complex parallelization strategies that simultaneously minimize the amount of communication and maximize the amount of exposed concurrency. For maximum performance the software may have to tailor the computation to the specific machine at hand. Requiring the applications programmer to generate this software can dramatically complicate the programming process and impair portability.

Finally, nondeterministic execution exacerbates all of the problems outlined above. Parallel machines present an inherently nondeterministic model of computation. If the programming environment exposes this nondeterminism to the programmer, it complicates the debugging process by making it difficult to reproduce and eliminate programming errors.

1.1 Explicitly Parallel Systems

Programmers have traditionally developed software for parallel machines using explicitly parallel systems [88, 130]. These systems provide constructs that programmers use to create parallel tasks. On shared-memory machines the programmer synchronizes the tasks using low-level primitives such as locks, condition variables and barriers. On message-passing

machines the programmer must also manage the communication using explicit message-passing operations such as send and receive. Explicitly parallel systems are only one step removed from the hardware, giving programmers maximum control over the parallel execution. Programmers can exploit this control to generate extremely efficient computations. The problem is that these systems present a complex programming environment, directly exposing the programmer to all of the problems outlined in the previous section.

1.2 High-Level Languages

Language designers have responded to the complexity of explicitly parallel programming by providing high-level languages. Each such language presents a set of abstractions that simplify and structure the way the programmer thinks about and expresses a parallel algorithm. These abstractions insulate the programmer from the complexity of directly controlling the parallel execution and typically offer a safer, more specialized model of parallel computation.

The implementation of each language encapsulates a set of algorithms for managing concurrency. In effect, each implementation is a reusable package encapsulating the knowledge of how to exploit a specific kind of concurrency on a specific machine. Ideally, the programmer expresses the computation in an abstract, machine-independent way. The implementation then manages the concurrency and performs the machine-specific optimizations required to map the computation efficiently onto the current hardware platform. The resulting parallel program is easier to develop and readily ports to new machines.

Existing approaches support restricted computational paradigms. Languages such as Fortran 90 [105] and C* [111] support the exploitation of regular, data-parallel forms of concurrency. Programmers using these languages view their program as a sequence of operations on large aggregate data structures such as sets or arrays. The implementation can execute each aggregate operation in parallel by performing the operation on the individual elements concurrently. This approach works well for programs that fit the data-parallel paradigm. Its success illustrates the utility of having the language implementation, rather than the programmer, control the parallel execution and data movement.

Parallelizing compilers automatically parallelize serial programs. Because the serial

programming paradigm is significantly simpler than explicitly parallel paradigms, parallelizing compilers liberate programmers from many of the problems associated with writing explicitly parallel programs. Like the data-parallel approach, this approach works well for a specific kind of parallelism: the loop-level parallelism present in scientific programs that manipulate dense matrices. The complexity of the highly tuned, machine-specific code that parallelizing compilers generate [135] illustrates the need for high-level abstractions that shield programmers from the low-level details of their computations.

1.3 Jade

Both the parallelizing compiler and the data-parallel approaches are designed to exploit regular concurrency available *within* a single operation on aggregate data structures. An orthogonal source of concurrency also exists *between* operations on different data structures. In contrast to data-parallel forms of concurrency, task-level concurrency is often irregular and may depend on the input data or on the results of previous computation. The tasks' computations may be heterogeneous, with different tasks executing completely different computations.

Jade [108, 109, 110] is a high-level, implicitly parallel language designed for exploiting task-level concurrency. A Jade programmer starts with a program written in a sequential, imperative language, then uses Jade constructs to describe how the program accesses data. The Jade implementation analyzes the data access information to automatically exploit the task-level concurrency present in the computation. The resulting parallel execution preserves the semantics of the original serial program.

The Jade language design is based on an analysis of the respective strengths of the programmer and implementation. The implementation discovers the concurrency and handles the low-level details of mapping the computation onto the machine. The programmer helps the implementation perform these activities by providing high-level, application-specific granularity and data usage information. Specifically, the programmer provides the following information.

- A decomposition of the data into the atomic units that the program will access.

- A decomposition of the sequential program into tasks.
- A specification of how each task may access data.

The implementation checks each task's accesses to ensure that it respects its access specification.

Almost all other parallel languages that are designed to exploit task-level concurrency present a control-oriented paradigm. They provide constructs that programmers use to directly create and control the parallel execution. Jade takes a fundamentally different approach. It is a declarative language designed to express information about data, not control. The research presented in this thesis explores the advantages and limitations of this approach.

1.3.1 Scope

We designed Jade as a focused language tailored for a specific kind of concurrency. Within its intended domain it provides a safe, effective programming environment that harmonizes with the programmer's needs and abilities. Ideally, Jade will become integrated into a more comprehensive programming system that encompasses all of the different kinds of concurrency. Each part of the system will specialize in a specific kind of concurrency, and the parts will work together to allow programmers to easily exploit all of the different kinds of concurrency present in their applications. We next describe the kind of concurrency that we designed Jade to exploit.

Because Jade requires the programmer to play a role in the parallelization process, it is counterproductive to use Jade for forms of parallelism (such as instruction-level parallelism or loop-level parallelism in programs that manipulate dense matrices) that can be exploited automatically. Jade is instead appropriate for computations that are beyond the reach of automatic techniques. Such computations may exploit dynamic, data-dependent concurrency or concurrently execute pieces of code from widely separated parts of the program. Jade is especially useful for coarse-grain computations because it provides constructs that allow programmers to help the system identify large parallel tasks.

Jade's high-level abstractions hide certain aspects of the underlying parallel computing environment. While hiding these aspects protects the programmer from much of the

complexity of explicit concurrency, it also narrows the scope of the language by denying the programmer control over the corresponding aspects of the parallel computation. It is therefore inappropriate to use Jade for programs that demand highly optimized, application-specific synchronization and communication algorithms.

Finally, Jade forces programmers to express the basic computation in a serial language. It is therefore counterproductive to use Jade for computations that are more naturally expressed in an explicitly parallel language.

1.3.2 Advantages

Jade is designed to extend the sequential programming paradigm to programs that exploit task-level concurrency. Within its application domain the Jade approach delivers significant programming advantages. The first major advantage is portability. We have implemented Jade as an extension to C on a wide variety of computational environments: uniprocessors, shared-memory multiprocessors, distributed-memory multiprocessors, message-passing machines and heterogeneous collections of workstations. Jade applications port without modification between all of these environments.

Jade promotes modular parallel programming. Jade programmers only provide local information about how each task accesses data. There is no code in a Jade program that manages the interactions between tasks. The programmer can therefore concentrate solely on the behavior of the task under development, and need not develop a detailed mental model of the computation's global concurrency structure. These locality properties allow programmers to develop the abstractions required for modular parallel programming. For example, Jade programmers can build abstract data types that completely encapsulate the Jade constructs required to guide the parallelization process [107].

The Jade implementation encapsulates all of the concurrency management code required to exploit task-level concurrency. Jade programmers can therefore concentrate on the semantics of the actual computation, rather than struggling with the low-level synchronization and communication code required to coordinate the parallel execution. Because the programmer does not directly control the parallel execution, the Jade implementation has the freedom it needs to apply machine-specific optimizations and implementation strategies.

Jade supports several kinds of access specifications. The most basic access specifications only allow programmers to describe how tasks will read and write data. Programs that only use these access specifications are guaranteed to preserve the semantics of the underlying serial program. Such programs therefore execute deterministically, which simplifies debugging.

Jade also allows programmers to provide additional high-level information about the operations tasks perform. For example, programmers can specify that certain operations on a given data structure commute (i.e., generate the same result regardless of the order in which they execute). [1] Allowing the programmer to express such information extends the range of applications that programmers can effectively develop using Jade.

Access specifications help the implementation map computations efficiently onto parallel machines. In message-passing environments access specifications allow the implementation to preserve the abstraction of a single address space. Because the implementation knows which pieces of data each task will access, it can automatically transfer the accessed data to the processor that will execute the task. This functionality frees the programmer from the complexity of using message-passing operations to route data through the parallel machine.

Access specifications also allow the implementation to apply communication optimizations. A priori knowledge of data access patterns enables the implementation to enhance the locality of the computation by executing tasks on processors with locally available copies of the accessed data. The implementation also optimizes the execution by concurrently fetching different pieces of accessed data and using excess concurrency to hide the latency of accessing remote data.

1.4 Evaluation

The Jade design and implementation is based on an abstract analysis of how the abilities of parallel programmers, the needs of parallel computations and the functionality of the implementation may interact to support the effective generation of parallel applications.

[1] It is the programmer's responsibility to ensure that the operations actually commute - the implementation does not check this property. See Section 2.2.9.3 for a complete discussion of this issue.

While such an analysis may lead to an elegant, internally consistent language, the language design project remains incomplete without an evaluation of how well the language will actually work in practice. For a parallel language such an evaluation should explore at least two aspects of the design and implementation:

- **Programmability** How well does the language support the development of parallel applications?
- **Performance** How well do the resulting applications perform on different hardware platforms?

We address these questions by developing a set of scientific and engineering applications in Jade and running the applications on a variety of hardware platforms. This experience enables us to perform an initial evaluation of how well Jade works in practice and provides insight into the strengths and weaknesses of the design and implementation.

We believe the field of parallel programming languages suffers from too many paper designs, too few serious implementations and a severe lack of experience developing complete applications. Language designers need feedback from actual applications experience to understand the application space and to address the practical needs of parallel programmers. Our applications experience contributes to the field as a whole by providing insight into the characteristics of the parallel applications and giving specific examples of the problems that the use of Jade either introduces or eliminates.

1.5 Organization

This rest of thesis is organized as follows. Chapter 2 presents the Jade programming language. It outlines the basic concepts of the language, then provides a detailed description of the language constructs. Chapter 3 describes the implementation of Jade on a variety of computational platforms. It presents the key concurrency management algorithms encapsulated inside the Jade implementation. Chapter 4 presents our experience developing complete scientific and engineering applications using Jade. We describe the software development process for each application, then present performance numbers for the application running on both shared-memory and message-passing platforms. Chapter 5 presents

a taxonomy of parallel programming systems. Chapter 6 discusses several directions for future research and Chapter 7 presents the conclusions.

Chapter 2

The Jade Language

The goal of Jade is to preserve the sequential programming paradigm for computations that exploit task-level concurrency. Jade satisfies this goal by taking a declarative approach to parallel execution. Jade programmers direct the parallelization process by augmenting a serial program with granularity and data usage information. The resulting program ports without modification to a wide range of parallel architectures and preserves the modularity properties of the original serial program.

This chapter presents a detailed description of Jade. It describes both the fundamental concepts behind Jade (objects, tasks, and access specifications) and the relationship between these concepts and concurrent execution. It describes the concrete expression of these concepts in the constructs of the language. It analyzes the design choices implicit in the structure of Jade and presents a rationale for the final design.

2.1 Fundamental Concepts

Jade is based on three fundamental concepts: shared objects, tasks and access specifications. Shared objects and tasks are the mechanisms the programmer uses to specify the granularity of, respectively, the data and the computation. The programmer uses access specifications to specify how tasks access data. The implementation analyzes this data usage information to automatically extract the concurrency, generate the communication and optimize the parallel execution. The next few sections introduce the three fundamental concepts.

2.1.1 Shared Objects

Jade supports the abstraction of a single mutable memory that all parts of the computation can access. Each piece of data allocated in this memory is called a shared object. The programmer therefore implicitly aggregates the individual words of memory into larger granularity objects by allocating data at a certain granularity. Shared objects are atomic: no piece of memory can be part of more than one object.

2.1.2 Tasks

Jade programmers explicitly decompose the serial computation into tasks by identifying the blocks of code whose execution generates a task. Programmers can create hierarchically structured tasks by creating tasks which, in turn, decompose their computation into child tasks. In many parallel programming languages tasking constructs explicitly generate parallel computation. Because Jade is an implicitly parallel language with serial semantics, Jade programmers ostensibly use tasks only to specify the granularity of the parallel computation. The Jade implementation, and not the programmer, then decides which tasks execute concurrently.

2.1.3 Access Specifications

In Jade, each task has an access specification that declares how it (and its child tasks) will access individual shared objects. It is the responsibility of the programmer to provide an initial access specification for each task when that task is created. As the task runs, the programmer may dynamically update its access specification to more precisely reflect how the remainder of the task accesses shared objects.

The simplest access specifications declare how a task will read and write shared objects. Jade also supports more sophisticated access specifications that provide more semantic information about how tasks access objects. For example, Jade currently supports access specifications that declare that tasks' accesses to a given object commute. The Jade paradigm generalizes to include access specifications that provide an arbitrary amount of information about how tasks access objects.

2.1.4 Parallel and Serial Execution

The Jade implementation analyzes access specifications to determine which tasks can execute concurrently. This analysis takes place at the granularity of individual shared objects. For access specifications that only declare reads and writes, the dynamic data-dependence constraints determine the concurrency pattern. If one task declares that it will write an object and another declares that it will access the same object, there is a dynamic data dependence between the two tasks and they must execute sequentially. The task that would execute first in the serial execution of the program executes first in all parallel executions. If there is no dynamic data dependence between two tasks, they can execute concurrently.

This execution strategy preserves the relative order of reads and writes to each shared object. If a program only declares read and write accesses, the implementation guarantees that all parallel executions preserve the semantics of the original serial program and therefore execute deterministically.

More sophisticated access specifications may allow the implementation to further relax the sequential execution order. For example, if two tasks declare that their accesses to a given object commute, the implementation has the freedom to execute the tasks in either order. In this case the implementation determines the execution order dynamically, with different orders possible in different executions.

2.1.5 Execution Model

As a task runs, it executes its serial computation. It may also decompose its computation into a set of subcomputations by serially creating child tasks to execute each subcomputation. When a task is created, the implementation executes a programmer-provided piece of code that generates its access specification. As the program runs, the implementation analyzes tasks' access specifications to determine when they can legally execute.

When a task can legally execute, the Jade implementation assigns the task to a processor for execution. In the message-passing implementation, the processor on which the task will execute issues messages requesting the remote shared objects that the task will access. The implementation then moves (on a write access) or copies (on a read access) the objects to

that processor. When all of the remote shared objects arrive, the implementation executes the task. When the task finishes, the implementation may enable other tasks for execution.

The Jade implementation dynamically checks each task's accesses to ensure that it respects its access specification. If a task attempts to perform an access that it did not declare, the implementation will detect the violation and generate a run-time error identifying the undeclared access.

2.2 Basic Jade Constructs

In this section we describe the basic constructs Jade programmers use to create and manipulate tasks, shared objects and access specifications. We designed Jade to be implemented as an extension to an existing sequential programming language. This approach preserves much of the language-specific investment in programmer training and software tools. The current implementation of Jade is an extension to C.

2.2.1 Shared Objects

The Jade memory model is based on the abstraction of a single global mutable memory. Programmers access data in Jade programs using the same linguistic mechanism as in the original C program. Jade extends the standard memory model by segregating pieces of data that multiple tasks may access from data that only one task may access.

The Jade type system uses the `shared` keyword to identify shared objects. Figure 2.1 shows how to declare a shared double, a shared array of doubles, and a shared structure. Programmers access objects using pointers. Figure 2.1 also illustrates how to use the `shared` keyword to declare a pointer that points to a shared object; the `shared` is inserted just before the `*` in the pointer declaration.

It is possible for shared objects to, in turn, contain pointers to other shared objects. To declare pointers to such objects, the programmer may need to use several instances of the `shared` keyword in a given declaration. Figure 2.2 contains the declaration of a pointer to a shared pointer to a shared double and the declaration of a shared structure which contains an array of pointers to shared vectors of doubles.

```
double shared x;
double shared A[10];
struct {
  int i, j, k;
  double d;
} shared s;

double shared *p;
```

Figure 2.1: Jade Shared Object Declarations

```
double shared * shared *q;

struct {
  int n, m;
  double shared *data[N];
} shared s;
```

Figure 2.2: More Jade Shared Object Declarations

It is also possible for shared objects to contain function pointers. Jade enforces the rule that everything that shared objects point to must be shared. So, Jade also has the concept of a shared function. Figure 2.3 presents a simple example to illustrate the syntax.

```
int shared plus_one(int a) { return(a+1); }

int shared (*f) ();
f = plus_one;
```

Figure 2.3: Jade Shared Function Declarations

Finally, it is illegal for shared objects to contain union types. The implementation must know the type of all data in shared objects so it can apply the format translation required in heterogeneous environments. It is impossible for the implementation to determine the type of data stored in a union.

Programmers can allocate shared objects dynamically using the `create_object` construct, which takes as parameters the type of data in the object and the number of elements of that type to allocate. If the number of elements is one the programmer can omit the parameter. The `create_object` construct returns a pointer to the allocated object. Figure 2.4 contains a simple example.

Many parallel machines have multiple physical memories. Typically, each processor is associated with a specific memory, and it takes longer for a processor to access the remote memories of other processors than to access its own local memory. In such situations the programmer may wish to control where the implementation allocates shared objects. Jade supports the concept of an unbounded set of virtual processors; at run time the implementation maps each virtual processor onto a physical processor. Jade provides an object allocation construct (the `create_at_object` construct) that allows the programmer to specify that the implementation should allocate the shared object in the memory associated with a given virtual processor. The first parameter to the `create_at_object` construct is the number of the virtual processor (the virtual processors are indexed by the non-negative integers). Figure 2.4 contains a simple example.

```
double shared *p;
p = create_object(double, 10);

double shared *ap;
int vp = 1;
ap = create_at_object(vp, double, 10);
```

Figure 2.4: Jade Shared Object Creation

The programmer may know that some objects will be accessed by every processor. The message-passing implementation can take advantage of this information to broadcast the object to all processors whenever it is updated, rather than serially sending the object to the processors upon request. The programmer can create such an object by passing a virtual processor number of -1 to `create_at_object`.

Programmers can also implicitly allocate a shared object by declaring a global variable to be shared. While procedure parameters or local variables may point to shared objects,

it is illegal for the parameter or variable itself to be shared. It is therefore impossible to allocate shared objects on the procedure invocation stack. In Figure 2.5, `x`, `A` and `s` must be global.

```
double shared x;
double shared A[N];
struct {
  int i, j, k;
  double d;
} shared s;
```

Figure 2.5: Jade Global Shared Object Declarations

Strictly speaking, in Jade every global variable is a shared object and must be declared using the `shared` keyword. In practice, this restriction makes it impossible to use the standard C include files. So, the current implementation generates warnings, not errors, for non-shared global variables.

2.2.2 Deallocating Objects

The Jade programmer is responsible for informing the Jade implementation when the computation will no longer access an object. The implementation can then reuse the object's memory for other objects or for internal data structures. The programmer uses the `destroy_object` construct to deallocate an object. The construct takes one parameter: a pointer to the object.

Any language that allows the programmer to explicitly deallocate memory faces the potential problem of dangling pointers when the programmers deallocate objects before their last use. This problem can become especially severe in parallel contexts if the programmer does not correctly synchronize the deallocation with other potentially concurrent uses. Just as Jade preserves the serial semantics for reads relative to writes, it preserves the serial semantics for all accesses relative to deallocations. Jade therefore eliminates the problem of having one task deallocate an object while another task concurrently accesses it. Of course, if the serial program accesses an object after its deallocation, the corresponding

Jade program will suffer from the same error.

2.2.3 Part Objects

In some situations the natural allocation granularity of the data may be finer than the desired shared object granularity in the parallel computation. For example, the Jade sparse Cholesky factorization algorithm in Section 2.3.2 manipulates a data structure that contains pointers to several dynamically allocated index arrays. In the parallel computation the desired shared object granularity is the data structure plus the index arrays. In such situations Jade allows programmers to aggregate multiple allocation units into a single shared object. The programmer creates such objects by declaring that some of the objects to which a shared object points are part of that object. As Figure 2.6 illustrates, the programmer declares such pointers using the `part` keyword.

```
struct {
  vector part *column;
  int part *row_index;
  int part *start_row_index;
  int num_columns;
} shared matrix;
```

Figure 2.6: Jade Part Object Declarations

Programmers dynamically allocate part objects using the `create_part_object` construct. The first parameter is a pointer to the shared object that the part object is part of. The second and third parameters are the type and number of data items in the part object. The part object is allocated in the same memory as the shared object of which it is a part. Figure 2.7 contains an example that illustrates how to allocate part objects. Programmers are also responsible for deallocating part objects when they are done with them; Jade provides the `destroy_part_object` construct for this purpose.

The part concept nests; it is possible for part objects to in turn contain pointers to other part objects. The implementation enforces the restriction that if one part or shared object points to a given part object, no other part or shared object points to the same part object.

```
matrix.row_index = create_part_object(indices, int, N);
matrix.col_index = create_part_object(indices, int, 200);
destroy_part_object(matrix.row_index);
```

Figure 2.7: Jade Part Object Creation

The set of all shared and part objects and pointers to part objects from shared and part objects therefore forms a forest of trees. The implementation enforces this property by requiring that the right hand side of every assignment to a part pointer be either `NULL` or a `create_part_object` expression.

2.2.4 Local Pointers

The Jade implementation ensures that tasks respect their access specifications by dynamically checking each task's accesses to shared objects. If the implementation dynamically checked every access, the overhead would unacceptably degrade the performance of the application. Jade therefore provides a mechanism in the type system that programmers can use to make the implementation perform many of the access checks statically rather than dynamically. The programmer can usually drive the overhead down to one dynamic check per object per task, which generates negligible amortized dynamic checking overhead.

The mechanism is that the programmer can declare a pointer to a shared object, and restrict how the program will access data using that pointer. Such a pointer is called a local pointer; Figure 2.8 contains several examples which demonstrate how to declare local pointers.

```
double local rd *rp;
double local wr *wp;
double local rd wr *rwp;
```

Figure 2.8: Jade Local Pointer Declarations

In Figure 2.8, the program can only read shared objects via `rp`, write shared objects via `wp` and read and/or write objects via `rwp`. The implementation statically enforces these

access restrictions. It performs a dynamic check when the program sets the local pointer to point to a shared object. The code fragment in Figure 2.9 illustrates which accesses are checked statically and which are checked dynamically.

```
void proc(double shared *p, int i, int j)
{
  double local rd *rp;
  double local rd wr *rwp;
  double d;

  rp = p;         /* checked dynamically for read access  */
  d = *rp;        /* checked statically  for read access  */
  d += p[i]       /* checked dynamically for read access  */

  rwp = &(p[j]); /* checked dynamically for read and
                    write access                         */
  *rwp += d;      /* checked statically  for read and
                    write access                         */
}
```

Figure 2.9: Local Pointer Usage

Local pointers introduce a complication into the access checking. If a task changes its access specification to declare that it will no longer access a shared object, the implementation should ensure that the task has no local pointers to that object. One way to do this is to count, for each shared object, the number of outstanding local pointers each task has that point into that object. In this case implementation could preserve the safety of the parallel execution by generating an error if a task with outstanding local pointers declared that it would no longer access the object. This feature is currently unimplemented.

It is impossible for one task to use another task's local pointer. The implementation enforces this restriction by forbidding shared objects to contain local pointers and forbidding a task to pass one of its local pointers as a parameter to a child task (Section 2.2.7 explains how tasks pass parameters to child tasks).

It is important to keep the concepts of shared and local pointers separate. The `create_object` and `create_at_object` constructs return shared pointers. The only

```
int shared s;
proc(int shared *p)
{
  int shared *lp;
  int local rd *rp;
  lp = (&s /* shared pointer */);
  rp = (&(p[5]) /* local rd pointer */);
  lp = &(p[1]); /* illegal - lp is a shared pointer,
                            &(p[1]) is a local pointer */
  *(p + 2  /* local wr pointer */) = 4;
}
```

Figure 2.10: Shared and Local Pointers

other way to get a shared pointer is to apply the `&` operator to a shared global variable. When applied to shared objects (with the exception for global variables noted above), the `&` operation yields local pointers. When applied to shared pointers, the pointer arithmetic operations yield local pointers (the implementation uses the surrounding context to determine the appropriate access restriction). The code fragment in Figure 2.10 illustrates these concepts.

A shared pointer always points to the first element of a shared object, but a local pointer can point to any element of a shared object. Shared pointers can travel across task boundaries, but no task can access another task's local pointers. It is illegal to store local pointers in shared objects.

2.2.5 Private Objects

Jade programs may also manipulate pieces of data that only one task may access. Such pieces of data are called private objects. All pieces of data allocated on the procedure call stack are private objects. Jade also provides a memory allocation construct, `new_mem`, that programmers can use to dynamically allocate private objects. `new_mem` has the same calling interface as the C `malloc` routine, taking as a parameter the size of the allocated private object. There is also the `old_mem` routine for deallocating private data; it has the

same calling interface as the C `free` routine. Programmers declare variables that deal with private objects using the normal C variable declaration syntax.

The Jade implementation enforces the restriction that no task may access another task's private objects. The implementation enforces this restriction in part by requiring that no shared object contain a pointer to a private object. The declarations in Figure 2.11 are therefore illegal, because they declare shared objects that contain pointers to private objects.

```
double * shared *d;  /* illegal declaration */
struct {
  double *e;
  int i, j, k;
} shared s;          /* illegal declaration */
```

Figure 2.11: Illegal Declarations

Of course, it is possible for private objects to contain pointers to shared objects. The declarations in Figure 2.12 are legal, because they declare private objects that contain pointers to shared objects.

```
double shared *A[N];
struct {
  double B[N];
  double shared *data;
  int i, j, k;
} s;
```

Figure 2.12: Legal Declarations

2.2.6 Summary of the Jade Data Model

Jade classifies data according to the following hierarchy:

- **Shared Object** An allocation unit that multiple tasks can access. There are two kinds of shared objects:

  - **Normal Shared Object** A global object or an object created by either the `create_object` or `create_at_object` construct.
  - **Part Object** An object that is part of a normal shared object. All part objects are created by the `create_part_object` construct.

- **Private Object** An allocation unit that only one task can access. Every piece of data allocated on the procedure call stack is a private object, as are pieces of data created by the `new_mem` construct.

Jade classifies pointers according to the following hierarchy:

- **Pointer to a Shared Object** A pointer which points into the single memory that multiple tasks can access. There are three kinds of pointers to shared objects:
  - **Shared Pointer** `create_object` and `create_at_object` return shared pointers. The `&` operation applied to a global identifier generates a shared pointer, and pointer to a shared function is also a shared pointer. All shared pointers point to the start of the normal shared object.
  - **Part Pointer** A pointer in a shared object that points to the start of a part object.
  - **Local Pointer** A pointer that points into a shared object. Each local pointer has an access restriction that limits how the programmer can access data using the pointer.
- **Private Pointer** A pointer into a private object.

There is one major restriction associated with how programmers can use the different kinds of pointers and objects: shared objects cannot contain local or private pointers. This restriction, along with the restriction that no parent task can pass a local pointer or private pointer to a child task as a parameter, maintains the integrity of each task's data. It ensures that no task can access another task's private objects or use another task's local pointers or private pointers.

2.2.7 The `withonly` Construct

Jade programmers use the `withonly` construct to identify tasks and to create an initial access specification for each task. The syntactic form of the `withonly` construct is as follows:

```
withonly { access specification } do (parameters) {
  task body
}
```

The `task body` section identifies the code that executes when the task runs. The code in this section cannot contain a `return` statement that would implicitly transfer control out of one task to another task. It is also illegal to have `goto`, `break` or `continue` statements that would transfer control from one task to another task.

The `task body` section executes in a naming environment separate from the enclosing naming environment. The `parameters` section exists to transfer values from the enclosing environment to the task. The `parameters` section itself is a list of identifiers separated by commas. These identifiers must be defined in the enclosing context. Their values are copied into the task's context when the task is created. When the task executes it can access these variables. The only values that can be passed to a task using the parameter mechanism are base values from C (`int`, `double`, `char`, etc.) and shared pointers. This restriction, along with the restriction that shared objects contain no pointers to private objects, ensures that no task can access another task's private objects.

2.2.8 The `access specification` Section

Each task has an access specification that declares how that task may access shared objects. When the implementation creates a task, it executes the `access specification` section, which generates an initial access specification for the task. This section can contain arbitrary C constructs such as loops, conditionals, indirect addresses and procedure calls, which gives the programmer a great deal of flexibility when declaring how a task will access data and makes it easier to specify dynamic parallel computations.

2.2.9 Basic Access Specification Statements

The `access specification` section uses access specification statements to build up the task's access specification. Each access specification statement declares how the task will access a single shared object. We next describe the basic access specification statements.

2.2.9.1 `rd(o)`

The `rd(o)` (read) access specification statement declares that the task may read the object `o`. Tasks can concurrently read the same object. The implementation preserves the serial execution order between tasks that declare a read access and tasks that declare any other access to the same object. If a task in the message-passing implementation declares that it will read an object the implementation ensures that an up-to-date copy of that object exists on the processor that will execute the task before the task executes.

2.2.9.2 `wr(o)`

The `wr(o)` (write) access specification statement declares that the task may write the object `o`. The implementation preserves the original serial execution order between tasks that declare a write access to a given object and all other tasks that declare any access to the same object. If a task in the message-passing implementation declares that it will write an object the implementation moves the object to the processor that will execute the task before the task executes.

2.2.9.3 `cm(o)`

The `cm(o)` (commuting) access specification statement declares that the task may read and write `o`, and that its access commutes with the accesses of all other tasks that also declare a commuting access to `o`. The implementation executes such tasks in some serial order, but the order may vary from execution to execution. The implementation preserves the original serial execution order between tasks that declare a commuting access to a given object and all other tasks that declare any other access to the same object. If a task in the message-passing implementation declares a commuting access to an object the

implementation moves the object to the processor that will execute the task before the task executes.

Programmers are intended to conceptually map the commuting declaration to a specific operation that the program performs. For example, the programmer may assign the commuting declaration to the increment operation. Whenever a task increments an object it declares a commuting access. Because increments commute the parallel execution will generate the same result as the serial execution.

Jade does not currently support multiple commuting operations which do not commute with each other. This situation could arise, for example, if a program contained tasks that incremented an object and tasks that multiplied the object by a constant. In this case the programmer would need a different commuting declaration for each operation. The implementation would preserve the sequential execution order for tasks that declared different commuting operations on the same object. Although Jade currently contains only one commuting access specification statement, the implementation's algorithms do support multiple commuting declarations. It would be trivial to provide several commuting access specification statements.

2.2.9.4 `de(o)`

The `de(o)` (deallocate) access specification statement declares that the task may deallocate `o`. The implementation preserves the original serial execution order between tasks that declare a deallocate access to a given object and all other tasks that declare any access to the same object. It is an error for a task to attempt to access an object after it has been deallocated.

2.2.9.5 Combination Access Specification Statements

For convenience the implementation supports several combination access specification statements. For example, the `rd_wr(o)` access specification statement declares that the task may read and write `o`. The `de_rd(o)` access specification statement declares that the task may read and deallocate `o`.

2.3 A Programming Example

In this section we show how to use Jade to parallelize a sparse Cholesky factorization algorithm. This algorithm factors a sparse, symmetric, positive-definite matrix. The example illustrates how to use Jade's object and `withonly` constructs, and demonstrates how Jade programs can exploit dynamic concurrency.

2.3.1 The Serial Algorithm

The serial algorithm stores the matrix using the data structures declared in Figure 2.13. Figure 2.14 shows a sample sparse matrix and the corresponding data structures. Because the matrix is symmetric, the algorithm only needs to store its lower triangle. The factorization algorithm repeatedly updates the data structures that represent this lower triangle.

The columns of the matrix are packed contiguously into one long vector of doubles. The `columns` global variable points to this vector. The `start_column` global variable tells where in the vector each column of the matrix starts. The `j`'th entry of the `start_column` array gives the index (in the `columns` array) of the first element of column `j`. The `row_index` global variable stores the row indices of the nonzeros of the matrix. The `i`'th element of `row_index` is the row index of the `i`'th element of the `columns` array.

```
double *columns;
int    *row_index;
int    *start_column;
int    num_columns;
```

Figure 2.13: Data Structure Declarations for Serial Sparse Cholesky Factorization

Figure 2.15 contains the serial code for this algorithm. The algorithm processes the columns of the matrix from left to right. It first performs an internal update on the current column. This update reads and writes the current column, bringing it to its final value in the computation. The algorithm then uses the current column to update some subset of the columns to its right. For a dense matrix the algorithm would update all of the columns to

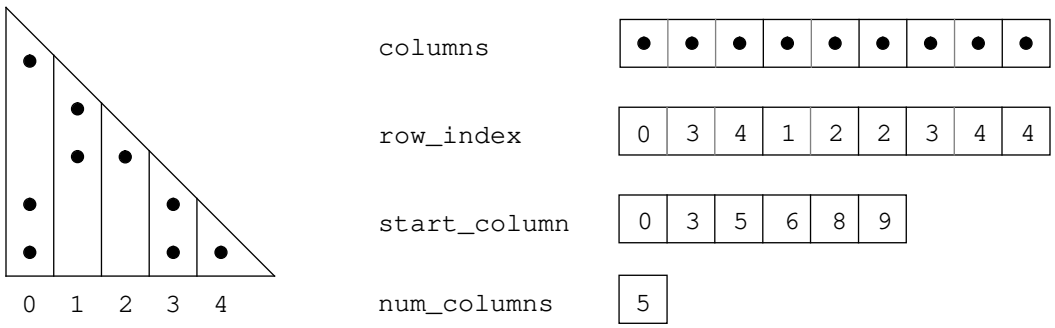

Figure 2.14: Data Structures for Serial Sparse Cholesky Factorization

```
factor()
{
  int i, j, first, last;
  for (j = 0; j < num_columns; j++) {
    /* update column j */
    InternalUpdate(j);
    first = start_column[j] + 1;
    last  = start_column[j+1] - 1;
    for (i = first; i <= last; j++) {
      /* update column row_index[i] with column j */
      ExternalUpdate(j, row_index[i]);
    }
  }
}
```

Figure 2.15: Serial Sparse Cholesky Factorization Algorithm

right of the current column. For sparse matrices the algorithm omits some of these updates because they would not change the updated column.

2.3.2 The Jade Algorithm

The first step in parallelizing a program using Jade is to determine the appropriate data granularity. In this case the programmer decides that the parallel computation will access

the matrix at the granularity of the individual columns. The programmer must therefore decompose the `columns` array so that each column is stored in a different shared object. The new matrix is structured as an array of column objects. The programmer also decides that the parallel computation will access the structuring data (the `num_columns`, `row_index` and `start_column` data structures) as a unit. Because these data structures are dynamically allocated for the particular matrix at hand, the programmer structures them as part objects of a single matrix object. Figure 2.16 gives the new data structure declarations, while Figure 2.17 shows the sample matrix and the new data structures.

```
typedef double shared *vector;
struct {
  vector part *_column;
  int part *_row_index;
  int part *_start_row_index;
  int _num_columns;
} shared matrix;
#define column matrix._column
#define row_index matrix._row_index
#define start_row_index matrix._start_row_index
#define num_columns matrix._num_columns
```

Figure 2.16: Data Structure Declarations for Jade Sparse Cholesky Factorization

Because the data structures have changed, the programmer must first modify the `InternalUpdate` and `ExternalUpdate` routines to use the new matrix data structure. The programmer then inserts the `withonly` constructs that identify each update as a task and specify how each task accesses data. Figure 2.18 contains the new `factor` routine.

2.3.3 Commuting Updates

In the presented `factor` routine the programmer has only declared that each external update reads and writes the updated column. In this algorithm, however, all external updates to the same column commute. The programmer could express this additional

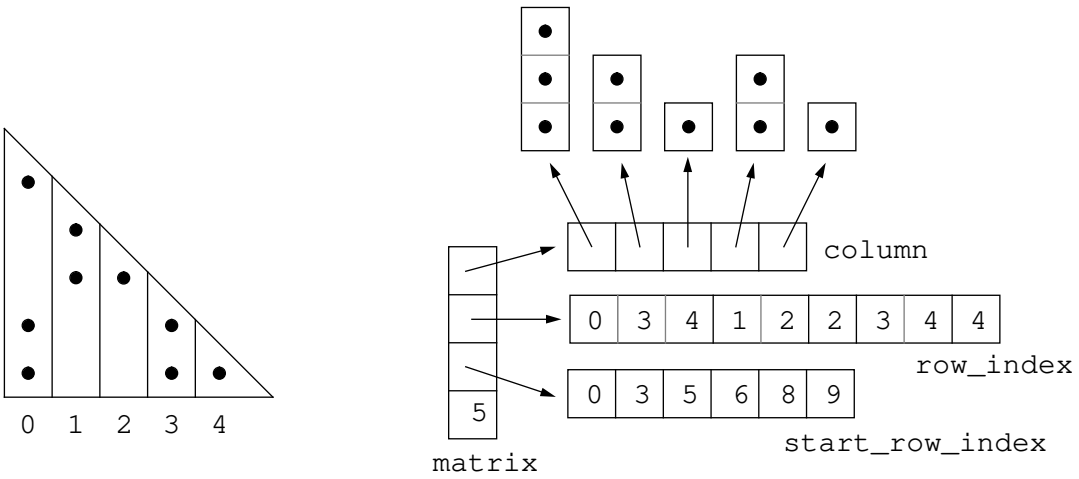

Figure 2.17: Data Structures for Jade Sparse Cholesky Factorization

information by changing the `rd_wr(column[row_index[i]])` access specification statement to `cm(column[row_index[i]])`.

Exposing the fact that external updates commute gives the implementation the freedom to schedule the updates in any order. This freedom may pay off in increased performance. With commuting declarations the implementation can schedule an external update from column `i` to column `j` as soon as the internal update to column `i` finishes and there is no other update to `j` in progress. In effect, the commutativity information allows the implementation to dynamically adjust the external update schedule to take into account the specific situation in each execution of the program. If the implementation enforced the serial execution order for updates, the external update could be forced to wait for another external update to execute. This delay would unnecessarily degrade the performance if the other external update could not yet execute.

2.3.4 Dynamic Behavior

Conceptually, the execution of the `factor` routine on our sample matrix generates the task graph in Figure 2.19. When the program executes, the main task creates the internal and external update tasks as it executes the body of the `factor` procedure. When the implementation creates each task, it first executes the task's `access specification`

```
factor()
{
  int i, j, first, last;
  for (j = 0; j < num_columns; j++) {
    withonly {
      rd_wr(column[j]);
      rd(&matrix);
   } do (j) {
      /* update column j */
      InternalUpdate(j);
    }
    first = start_column[j] + 1;
    last  = start_column[j+1] - 1;
    for (i = first; i <= last; j++) {
      withonly {
        rd_wr(column[row_index[i]]);
        rd(column[j]);
        rd(&matrix);
      } do (i,j) {
        /* update column row_index[i] with column j */
        ExternalUpdate(j, row_index[i]);
      }
    }
  }
}
```

Figure 2.18: Jade Sparse Cholesky Factorization Algorithm

section to determine how the task will access data. It is this dynamic determination of tasks' access specifications that allows programmers to express dynamic, data-dependent concurrency patterns. Given the access specification, the implementation next determines if the task can legally execute or if the task must wait for other tasks to complete. The implementation maintains a pool of executable tasks, and dynamically load balances the computation by assigning these tasks to processors as they become idle. In a message-passing environment the implementation also generates the messages required to move or copy the columns between processors so that each task accesses the correct version of each

column. As tasks complete, other tasks become legally executable and join the pool of executable tasks. In effect, the Jade implementation dynamically interprets the high-level task structure of the program to detect and exploit the concurrency.

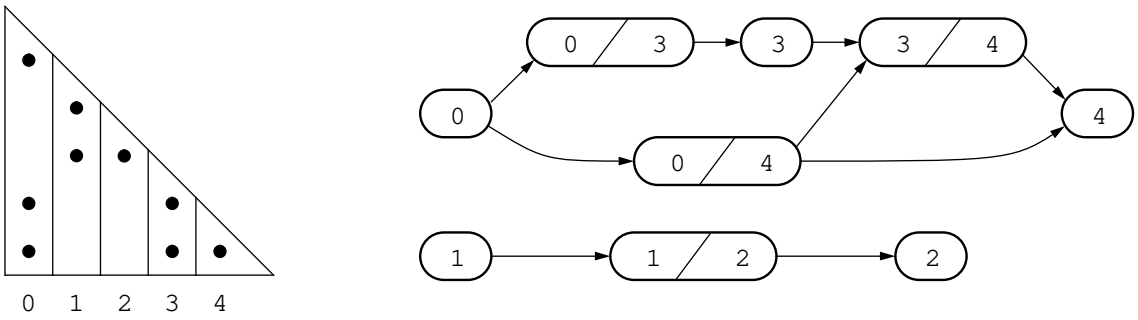

Figure 2.19: Task Graph for Jade Sparse Cholesky Factorization

2.3.5 Modularity

The sparse Cholesky factorization example illustrates how Jade supports the development of modular programs that execute concurrently. Each access specification only contains local information about how its task accesses data - each task is independent of all other tasks in the program. Even though the tasks must interact with each other to correctly synchronize the computation, the Jade implementation, and not the programmer, automatically generates the synchronization using the access specifications and the original serial execution order.

Jade's modularity properties support standard software engineering activities. Jade programmers can build abstract data types that completely encapsulate the code required to exploit concurrency available both within and between operations on the abstract data type [107]. Each operation contains Jade constructs that specify how it will access the data structures that implement the state of the abstract data type. The programmer simply writes a serial program that invokes the appropriate operations in the correct order, and the Jade implementation automatically relaxes the serial execution order to exploit the available concurrency. It is possible to exchange serial and parallel implementations of a given operation or even reimplement the abstract data type from scratch without touching the rest of the program. The clients of the abstract data type remain oblivious to the presence or

absence of parallel execution.

It is important to realize that explicitly parallel languages do not support the construction of such seamless abstract data types. The problem is correctly sequencing the executions of operations on the abstract data type. If the program contains parallel tasks that invoke operations on the same abstract data type, the programmer must insert synchronization code to ensure that the operations execute in the correct order. When the correct execution order depends on both the semantics of the tasks and the semantics of the abstract data type, it is impossible to encapsulate the synchronization code inside the implementation of the abstract data type. Tasks that invoke operations on abstract data types must therefore contain synchronization code that interacts with the synchronization code in other tasks to help generate the correct order. Even though this synchronization code exists only to mediate access to the abstract data type, it appears outside the abstract data type's implementation. Changes to the implementation of the abstract data type may therefore generate widespread changes across the rest of the program as the synchronization code in the tasks adjusts to the new implementation.

2.4 Programmer Responsibilities

Programmers and programming language implementations cooperate through the medium of a programming language to generate computations. To achieve acceptable performance, programmers must often adjust their programming styles to the capabilities of the language implementation. Any discussion of programmer responsibilities must therefore assume an execution model that includes a qualitative indication of the overhead associated with the use of the different language constructs. This discussion of programmer responsibilities assumes the dynamic execution model outlined above in Section 2.1.5 in which all of the concurrency detection and exploitation takes place as the program runs. Substantial changes to the execution model would change the division of responsibilities between the programmer and the implementation and alter the basic programming model. For example, an aggressive implementation of Jade that statically analyzed the code could eliminate almost all sources of overhead in analyzable programs. Such an implementation would support the exploitation of finer-grain concurrency.

The most important programming decisions Jade programmers make deal with the data and task decompositions. In this section we discuss how the decomposition granularities affect various aspects of the parallel execution and describe what the programmer must do to ensure that the Jade implementation can successfully parallelize the computation.

2.4.1 Data Decomposition

The decomposition of the data into shared objects is a basic design decision that can dramatically affect the performance of the Jade program. The current implementation performs the access specification, concurrency analysis and data transfer at the granularity of shared objects. Each object is a unit of synchronization. If one task declares that it will write an object and another task declares that it will read the same object, the implementation serializes the two tasks even though they may actually access disjoint regions of the object. In the message-passing implementation, each object is also a unit of communication. If a task executing on one processor needs to access an object located on another processor, the implementation transfers the entire object even though the task may only access a small part of the object.

Several factors drive the granularity at which the programmer allocates shared objects. Because each object is a unit of synchronization, the programmer must allocate shared objects at a fine enough granularity to expose an acceptable amount of concurrency. Because each object is a unit of communication, the programmer must allocate shared objects at a fine enough granularity to generate an acceptable amount of communication.

There is dynamic access specification overhead for each shared object that the task declares it will access. The programmer must allocate shared objects at a coarse enough granularity to generate an acceptable amount of access specification overhead. The message-passing implementation also imposes a fixed time overhead for transferring an object between two processors (this comes from the fixed time overhead of sending a message), and a fixed space overhead per object for system data structures. The programmer must allocate objects at a coarse enough granularity to profitably amortize both of these per-object overheads.

There is a natural granularity at which to allocate the data of any program. While

the programmer may need to change this granularity to effectively parallelize the program, requiring extensive changes may impose an unacceptable programming burden. In practice, programmers seem to use several standard reallocation strategies. These include decomposing large aggregates (typically arrays) for more precise access declaration, grouping several variables into a large structure to drive down the access specification overhead and replicating data structures used to hold intermediate results in a given phase of the computation. While determining which (if any) reallocation strategy to apply requires a fairly deep understanding of the computation, actually performing the modifications is often a fairly mechanical process. It may therefore be possible to automate the more tedious aspects of the modification process.

2.4.2 Task Decomposition

The task decomposition may also significantly affect the eventual performance. The basic issue is the dynamic task management overhead. The Jade programmer must specify a task decomposition that is coarse enough to profitably amortize this overhead. But the issue cuts deeper than a simple comparison of the total Jade overhead relative to the total useful computation. In the current implementation of Jade, each task creates its child tasks serially. Serial task creation serializes a significant part of the dynamic task management overhead. Even if the total Jade overhead is relatively small compared to the total useful computation, this serial overhead may artificially limit the performance. There are two things the programmer can do to eliminate a bottleneck caused by serial task creation: 1) parallelize the task creation overhead by creating tasks hierarchically, or 2) make the task granularity large enough to profitably amortize the serial task creation overhead. In some cases the programmer can apply neither of these strategies and must go elsewhere to get good performance.

2.5 Discussion

In any programming language design there is a trade-off between the range of computations that the language can express and how well the language supports its target programming

paradigm. Jade enforces high-level abstractions that provide a safe, portable programming model for a focused set of applications. Tailoring the language for this application set limits the range of applications that Jade supports. In this section we discuss both the advantages of using Jade in its target application domain and the limitations that supporting this domain well imposes.

2.5.1 Advantages

The sequential model of computation is in many ways much simpler than explicitly parallel models. Jade preserves this sequential model and inherits many of its advantages. If a program's access specifications only declare reads and writes, it preserves the serial semantics. Programmers can therefore develop the entire program using a standard serial language and development environment. When the program works, the programmer can then parallelize it using Jade, secure in the knowledge that adding Jade constructs cannot change the semantics of the program.

Preserving the serial semantics also supports the process of developing a Jade program from scratch. If a Jade program only declares reads and writes, it executes deterministically. Deterministic execution dramatically simplifies the debugging process by allowing programmers to reliably reproduce incorrect program behavior. Deterministic execution also simplifies the programming model. Jade programmers need not struggle with the complex phenomena such as deadlock, livelock and starvation that characterize explicitly parallel programming.

Strictly speaking, commuting declarations obviate the guarantee of deterministic execution. The implementation cannot check that the actual accesses of tasks that declare commuting accesses do in fact commute. If the accesses do not commute, different runs may generate different results as the task execution order varies from run to run. In practice we expect programmers to use commuting declarations in a deterministic way by mapping the commuting declaration onto a specific operation (such as an increment operation) that commutes with itself.

Serial programming languages promote modularity because they allow the programmer to focus on the dynamic behavior of the current piece of code. Jade preserves the modularity

advantages of serial languages because programmers only provide local information about how each task accesses data, not global information about how parallel tasks interact. The concurrency management algorithm is embedded in the Jade implementation, not the Jade program. New programmers can function effectively with a detailed understanding of selected pieces of the program; changes are confined to the modified tasks.

Jade preserves the abstraction of single address space. Even in message-passing environments, Jade programs access data using a single flat address space with the implementation automatically managing the movement of data. This abstraction frees programmers from the complexity of managing the flow of data through the machine.

2.5.2 Limitations

Jade's enforced abstractions maximize the safety and portability of Jade programs but prevent the programmer from accessing the full functionality of the parallel machine. This lack of control limits the programmer's ability to optimize the parallel execution. All Jade programs must use the general-purpose concurrency management algorithms encapsulated inside the Jade implementation. Because Jade denies the programmer control over many aspects of the process of exploiting concurrency, programmers cannot use highly optimized, application-specific synchronization or communication algorithms. In some cases the programmer may need to use a lower-level programming system to make the application perform acceptably.

It is always possible to execute any Jade program serially, in which case all data and control flow forward in the direction of the sequential execution. Some parallel computations, however, are most naturally or most efficiently structured as a set of cooperating tasks that periodically interact to generate a solution. Jade does not support the cyclic flow of data and control required to structure the computation this way. While it is often possible to express the computation in Jade, the resulting Jade program usually generates more tasks (and more task management overhead) than the cyclic computation.

Consider, for example, a standard iterative grid relaxation algorithm such as Successive Over Relaxation (SOR) [48]. A programmer using an explicitly parallel language could parallelize the algorithm by subdividing the grid into blocks and assigning one task to each

block. At every iteration each task would communicate with its neighbor tasks to acquire the latest values of the boundary elements, then recompute the values of the elements in its block. The tasks would continue their computation, interacting until the algorithm converged [127].

In this parallel computation data flows cyclically through the tasks - over the course of the computation each task both generates data that its neighbor tasks read and reads data generated by its neighbor tasks. Jade's serial semantics, however, means that if one Jade task produces data that another Jade task reads, the first task cannot also read data produced by the other task. To express computations such as the parallel SOR computation described above in Jade, the programmer must create one Jade task per block at every iteration, instead of one task per block for the entire computation as in the explicitly parallel program. While it may be more convenient to express the program in Jade (the Jade program is closer to the original serial program), the additional task management overhead may impair the performance of the Jade program if the task size is small relative to the task management overhead.

Some parallel algorithms dynamically adjust their behavior to adapt to the varying relative execution times characteristic of parallel computation. The tasks in a parallel branch and bound search algorithm, for example, may visit different parts of the search tree in different executions depending on how fast the bound is updated. Such algorithms nondeterministically access different pieces of data in different executions. In some cases the program itself may generate different results. Because Jade's abstractions are designed to support deterministic computations, it may be impossible to express such algorithms in the current version of Jade. Upon further examination, however, one can see that the nondeterminism often arises from the way the different parts of the computation access data. It would be possible to support many of these computations by extending Jade to support the expression of their nondeterministic data access patterns.

2.6 Advanced Constructs

We next present several advanced constructs and concepts that allow the programmer to exploit more sophisticated concurrency patterns.

2.6.1 Task Boundary Synchronization

In the model of parallel computation described so far, all synchronization takes place at task boundaries. A task does not start its execution until it can legally perform all of the accesses that it will ever perform. It does not give up the right to perform any of these accesses until it completes. This form of synchronization wastes concurrency in two cases: when a task's first access to an object occurs long after it starts, and when a task's last access to an object occurs long before it finishes. Figure 2.20 contains an example of both kinds of unnecessary synchronization(assuming that `p`, `q`, `r` and `s` all point to different objects).

```
extern double f(double d);
extern double g(double d);
extern double h(double d, double e);
void proc(double d, double shared *p, double shared *q,
                    double shared *r, double shared *s)
{
  withonly { wr(p); } do (p, d) {
    *p = f(d);
  }
  withonly {
    rd(p); wr(q); rd(q); wr(r);
  } do (p, q, r, d) {
    *q = g(d);
    *r = h(*q, *p);
  }
  withonly { rd(q); wr(s); } do (q, s) {
    *s = g(*q);
  }
}
```

Figure 2.20: Task Boundary Synchronization Example

In this example all three tasks execute serially. But the first task should be able to execute concurrently with the statement `*q = g(d)` from the second task, since there is no data dependence between these two pieces of code. Similarly, the statement `*r = h(*q, *p)` from the second task should be able to execute concurrently with the third

task.

To allow the programmer to overlap the execution of the parts of tasks that do not conflict, Jade supports both a more elaborate notion of access specification and several new language constructs. There is a new construct (the `with` construct) that programmers can use to update a task's access specification, and several new access specification statements. These constructs allow the programmer to more precisely specify the timing of the individual accesses to shared objects, which in turn may expose additional concurrency. In the example in Figure 2.20, the programmer can use these constructs to expose the pipelining concurrency available between these three tasks.

2.6.2 The `with` Construct

Programmers use the `with` construct to dynamically update a task's access specification to more precisely reflect how the remainder of the task will access data. Here is the syntactic form of the `with` construct:

```
with { access specification } cont;
```

Like the `access specification` section in the `withonly` construct, the `access specification` section in the `with` construct is an arbitrary piece of code containing access specification statements. The difference between the two is that the `access specification` section in the `withonly` construct establishes a new access specification for a new task, while the `access specification` section in the `with` construct modifies the current task's access specification.

2.6.3 Advanced Access Specifications

An access specification is a set of access declarations; each declaration declares how a task will access a given object. Declarations come in two flavors: immediate declarations and deferred declarations. An immediate declaration gives the task the right to access the object. The basic access specification statements described in Section 2.2.9 generate immediate declarations. A deferred declaration does not give the task the right to access the object. Rather, it gives the task the right to change the deferred declaration to an immediate

declaration, and to then access the object. The deferred access specification statements described in Section 2.6.4 generate deferred access declarations.

A task's initial access specification can contain both deferred and immediate declarations. As the task executes, the programmer can use a `with` construct to update its access specification.

Deferred declarations may enable a task to overlap an initial segment of its execution with the execution of an earlier task in cases when the two tasks would execute serially with immediate declarations. For example, if one task declares that it will immediately write an object and another task declares that it will immediately read that same object, the implementation completely serializes the execution of the two tasks. If the second task (in the sequential execution order) declares a deferred access, it can start executing before the first task finishes or even performs its access. When the second task needs to actually perform its access, it uses a `with` construct to change the deferred declaration to an immediate declaration. The second task then suspends at the `with` until the first task finishes.

Deferred declarations do not allow the programmer to change the order in which tasks access objects. In the above example the second task must perform its access after the first task. If the first task declared a deferred access and the second task declared an immediate access, the second task could not execute until the first task finished.

2.6.4 Deferred Access Specification Statements

There is a deferred version of each basic access specification statement; the programmer derives the deferred version by prepending `df` to the original basic statement. Specifically, Jade provides the following deferred access specification statements.

- `df_rd(o)` Specifies a deferred read declaration.
- `df_wr(o)` Specifies a deferred write declaration.
- `df_de(o)` Specifies a deferred deallocate declaration.
- `df_cm(o)` Specifies a deferred commuting declaration.

If used in a `with` construct, a deferred access specification statement changes the corresponding immediate declaration to a deferred declaration. If used in a `withonly` construct, it generates a deferred declaration in the access specification.

2.6.5 Negative Access Specification Statements

Jade programmers can use a `with` construct and negative access specification statements to eliminate access declarations from a task's access specification. There is a negative version of each basic access specification statement; the negative version is derived by prepending `no` to the original basic statement. Specifically, Jade provides the following negative access specification statements.

- `no_rd(o)` Eliminates read declarations.
- `no_wr(o)` Eliminates write declarations.
- `no_de(o)` Eliminates deallocate declarations.
- `no_cm(o)` Eliminates commuting declarations.

Negative access specification statements may allow a task to overlap a final segment of its execution with the execution of later tasks in cases when the tasks would otherwise execute sequentially. Consider, for example, a task that performs its last write to an object, then uses a `with` construct and a negative access specification statement to declare that it will no longer access the object. Succeeding tasks that access the object can execute as soon as the `with` construct executes, overlapping their execution with the rest of the execution of the first task. If the first task failed to declare that it would no longer access the object, the succeeding tasks would suspend until the first task finished.

2.6.6 Pipelined Concurrency

The programmer can use the `with` construct and the advanced access specification statements to exploit the pipelining concurrency available in the example in Figure 2.20. The programmer first uses the `df_rd(p)` access specification statement to inform the implementation that the second task may eventually read `p`, but that it will not do so immediately.

This gives the implementation the information it needs to overlap the execution of the first task with the statement `*q = g(d)` from the second task. When the second task needs to read `p`, it uses a `with` construct and the `rd(p)` access specification statement to convert the deferred declaration to an immediate declaration. At the same time, the `with` construct uses the `no_wr(q)` statement to declare that the second task will no longer write `q`. This gives the implementation the information it needs to overlap the execution of the third task and the statement `*r = h(*q, *p)` from the second task. Figure 2.21 contains the final version of the program.

```
extern double f(double d);
extern double g(double d);
extern double h(double d, double e);
void proc(double d, double shared *p, double shared *q,
                    double shared *r, double shared *s)
{
  withonly { wr(p); } do (p, d) {
    *p = f(d);
  }
  withonly {
    df_rd(p); wr(q); rd(q); wr(r);
  } do (p, q, r, d) {
    *q = g(d);
    with { rd(p); no_wr(q); } cont;
    *r = h(*q, *p);
  }
  withonly { rd(q); wr(s); } do (q, s) {
    *s = g(*q);
  }
}
```

Figure 2.21: Pipelined Concurrency Example

2.6.7 Hierarchical Objects

In the Jade programming model presented so far, the initial access specification must declare every access that the task will ever perform to existing shared objects. This is, in general, an unacceptable restriction. Some computations (for example, many computations that perform tree traversals) contain tasks that determine precisely which objects they will access only as they execute.

Jade supports these computations by allowing programmers to structure objects hierarchically. The programmer can specify that one object is a child of another object. Instead declaring precisely which objects a task will access, the programmer may specify only that the task will access some of the child objects of the parent object. When the task determines precisely which child object it (or a child task) needs to access, it can refine its access specification to reflect the new information. The hierarchical object mechanism allows programmers to create tasks whose computations traverse hierarchical data structures.

The programmer specifies the parent/child relationship when the child object is created. The `create_child_object` construct creates a child object. This construct takes three parameters: a shared pointer to the parent object, the type of the data items in the object, and the number of items in the object. The programmer can control where the child object is allocated using the `create_at_child_object` construct. Figure 2.22 shows how to create child objects.

```
int shared *parent;
int shared *child;
int vp;

child = create_child_object(parent, int, N);
child = create_at_child_object(parent, vp, int, N);
```

Figure 2.22: Child Object Creation

Jade provides additional access specification statements for declaring that tasks may declare accesses to child objects. For each basic access specification statement there is a corresponding child access specification statement. The child statement is derived by

prepending `ch` to the basic statement. Specifically, Jade provides the following child access specification statements.

- `ch_rd(o)` Declares that the task or its child tasks may declare a read access to a child object of `o`.
- `ch_wr(o)` Declares that the task or its child tasks may declare a write access to a child object of `o`.
- `ch_cm(o)` Declares that the task or its child tasks may declare a commuting access to a child object of `o`.
- `ch_de(o)` Declares that the task or its child tasks may declare that they will deallocate a child object of `o`.

Immediate declarations also give tasks the right to declare that they will declare an access to a child object. The difference between immediate and child declarations is that child declarations do not give the task the right to access the parent object. This extra precision may allow the Jade implementation to exploit more concurrency available between tasks that only access child objects. Specifically, if tasks that could otherwise execute concurrently declare commuting child accesses, they can still execute concurrently. If the tasks declared immediate commuting accesses they would execute in some serial order.

For pragmatic implementation reasons the current Jade implementation does not allow a task to declare that it will simultaneously access both a parent object and one of its child objects. This restriction implies that once a task declares that it will access a child object, it can never again access the parent. Therefore, Jade programs usually create a child task to perform each new traversal of a hierarchical data structure.

2.6.8 The `block` Construct

Programmers use all of the constructs presented so far to describe how a program accesses data. There is one construct, the `block` construct, that programmers use to directly control the concurrency. Here is the syntactic format of the `block` construct:

```
block {
  code
}
```

If the programmer surrounds a piece of code with a `block` construct, control does not proceed past the block until all of the tasks created in the block have finished. Programmers often use `block` constructs to separate different phases of the program. This prevents unwanted interactions between the phases and makes it easier to measure the execution time of each phase.

2.7 Access Specifications and Concurrency

The current Jade implementation uses a conservative approach to exploiting concurrency. (see Chapter 5 for a discussion of other approaches). It does not execute a task until it knows that the task can legally perform all of its declared accesses. This section defines how the access specifications of tasks interact to generate parallel and serial execution.

Each access specification consists of a set of access declarations. Each access declaration is generated by an access specification statement and gives the task the right to access a given object. Access declarations also impose serialization constraints on the parallel execution. The nature of the constraint depends on the semantics of the declared accesses. The current Jade implementation supports mutual exclusion constraints and constraints that force tasks to execute in the same order as in the original serial program.

The set of serialization constraints is a lattice. Table 2.23 shows the serialization constraint lattice for the current version of Jade. In principle this lattice could expand to include arbitrary constraints. The serialization constraint between two tasks is the least upper bound of the set of serialization constraints induced by the cross product of the two access specifications. The two access declarations must refer to the same shared object to impose a constraint.

Tables 2.1 and 2.2 present the induced serialization constraints for the current version of Jade. In these tables the access declaration from the first task (in the sequential execution order) is in the leftmost column of the table, the access declaration from the second task is on the top row of the table.

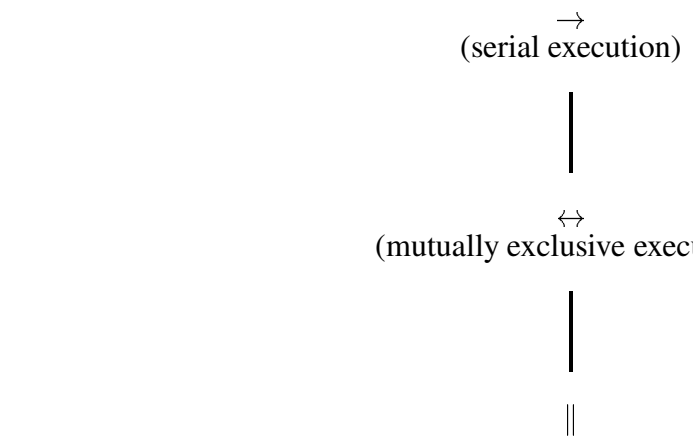

Figure 2.23: Serialization Constraint Lattice

	rd(o)	ch_rd(o)	wr(o)	ch_wr(o)	de(o)	ch_de(o)
rd(o)	$\|\|$	$\|\|$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
df_rd(o)	$\|\|$	$\|\|$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
ch_rd(o)	$\|\|$	$\|\|$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
st(o)	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
df_st(o)	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
ch_st(o)	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$

In this table $st \in \{$wr,de,cm$\}$.

Table 2.1: Induced Serialization Constraints – Part I

	cm(o)	ch_cm(o)	df_st_2(o)
st_1(o)	$\rightarrow$	$\rightarrow$	$\|\|$
df_st_1(o)	$\rightarrow$	$\rightarrow$	$\|\|$
ch_st_1(o)	$\rightarrow$	$\rightarrow$	$\|\|$
cm(o)	$\leftrightarrow$	$\|\|$	$\|\|$
df_cm(o)	$\|\|$	$\|\|$	$\|\|$
ch_cm(o)	$\|\|$	$\|\|$	$\|\|$

In this table $st_1 \in \{$rd,wr,de$\}$ and $st_2 \in \{$rd,wr,de,cm$\}$.

Table 2.2: Induced Serialization Constraints – Part II

The two tables address different possibilities for the second task's declaration. Table 2.1 describes the induced constraints when the second task declares an immediate or child read, write or deallocate access. The region of $\|$ entries in the upper left hand corner of the table says that two read entries (of any kind) impose no serialization constraint. The rest of table contains $\rightarrow$ entries, which specify that all of the other possible combinations in the table serialize the two tasks.

Table 2.2 gives the serialization constraints when the second task declares an immediate or child commuting declaration or a deferred declaration of any kind. The last column of the table, which contains all $\|$ entries, says that if the second task declares a deferred access there is no serialization constraint. The upper left hand corner handles the case when the first task declares any access except a commuting access and the second task declares a commuting access. All of the entries are $\rightarrow$, which specifies that the declarations serialize the tasks. Finally, the lower left corner handles the possibility of two commuting declarations. These declarations impose no serialization constraint unless both declarations are immediate declarations. In this case the $\leftrightarrow$ specifies that the tasks must execute serially, but can execute in either order.

There is one somewhat subtle point about the serialization constraints. There may be serialization constraints between a child task and its parent task. Because the child task executes before the remainder of the parent task, in Tables 2.1 and 2.2 the child task's access declaration would appear in the leftmost column while the parent task's access declaration would appear in the top row. If there is an induced serialization constraint between the child and parent tasks, the parent task must suspend until the child task finishes or executes a `with` construct that eliminates the constraint.

For Jade's conservative approach to exploiting concurrency to succeed, the implementation must know ahead of time a conservative approximation of the accesses that each task and its child tasks will perform. The Jade implementation therefore requires that each task's access specification correctly summarize how it (and its child tasks) will access shared objects. From the programmer's perspective, this requirement takes the form of several rules which govern the use of access specifications.

- To access or deallocate a shared object, a task's access specification must contain an immediate access declaration that enables the access. Table 2.3 summarizes which

access declarations enable which accesses.

- If a task's initial access specification contains an access declaration on a given object, its parent task's access specification (at the time the task is created) must also contain one of the following access declarations:
  - A corresponding access declaration on the same object. The declaration must be either a deferred, immediate or child access declaration.
  - A corresponding access declaration on the object's parent object. The declaration must be either an immediate or a child access declaration.

  Table 2.4 summarizes the rules.

- A `with` construct can change a deferred access specification to a corresponding immediate or child access specification, an immediate to a corresponding deferred or child access specification, a child to a corresponding deferred or immediate access specification, or eliminate an access specification. A `with` construct can also declare that the task will access a child object if the task's access specification contains a corresponding child access declaration on the parent object. In this case the `with` construct must also eliminate the access declaration on the parent object. Table 2.5 summarizes the rules.

- When a task creates an object that has no parent object, its access specification is automatically augmented with deferred read, write, deallocate and commuting access declarations for that object.

- When a task creates a child object, its access specification does not change.

There are additional restrictions associated with commuting access declarations. To prevent deadlock, the Jade implementation does not allow a task to execute a `with` construct that declares an immediate access to any object when the task's access specification already contains an immediate commuting declaration. The implementation also prevents a task from creating any child tasks while its access specification contains an immediate commuting declaration.

Access Declaration	Enabled Accesses
rd(o)	read from o
wr(o)	write to o
de(o)	deallocate o
cm(o)	read from o and write to o

Table 2.3: Enabled Accesses

If child task declares	Parent task must declare one of
st(o)	*st*(o), df_*st*(o), ch_*st*(o), *st*(po) or ch_*st*(po)
df_*st*(o)	*st*(o), df_*st*(o), ch_*st*(o), *st*(po) or ch_*st*(po)
ch_*st*(o)	*st*(o), df_*st*(o), ch_*st*(o), *st*(po) or ch_*st*(po)

In this table po is the parent object of o and $st \in \{$rd,wr,de,cm$\}$.

Table 2.4: Access Specification Rules for the withonly Construct

If a with declares	Task must declare one of
st(o)	*st*(o), df_*st*(o), ch_*st*(o), *st*(po) or ch_*st*(po)
df_*st*(o)	*st*(o), df_*st*(o), ch_*st*(o), *st*(po) or ch_*st*(po)
ch_*st*(o)	*st*(o), df_*st*(o), ch_*st*(o), *st*(po) or ch_*st*(po)
no_*st*(o)	*st*(o), df_*st*(o) or ch_*st*(o)

In this table po is the parent object of o and $st \in \{$rd,wr,de,cm$\}$.

Table 2.5: Access Specification Rules for the with Construct

2.8 Language Design Rationale

We designed Jade to test the hypothesis that it is possible to preserve the sequential imperative programming paradigm for computations that exploit task-level concurrency. In this section we first present the requirements and goals that drove the language design. We then discuss the specific design decisions that determined the final form of the language.

To preserve the sequential imperative programming paradigm, we structured Jade as a declarative extension to a sequential language. We believed that to adequately test the basic hypothesis, Jade had to preserve the following key advantages of the sequential paradigm:

- **Portability** It must be possible to implement Jade on virtually any MIMD computing environment.

- **Safety** Jade programs must not exhibit the complex failure modes (such as deadlock and livelock) that characterize explicitly parallel programming. Jade must preserve the serial semantics and provide guaranteed deterministic execution.
- **Modularity** Jade must preserve the modularity benefits of the sequential programming paradigm.

We also wished to maximize the utility of Jade as a parallel programming language. The following goals structured this aspect of the design process:

- **Efficiency** Because Jade was designed to exploit coarse-grain, task-level concurrency, we expected the dynamic task management overhead to be profitably amortized by the large task size. The important design goal was to minimize (and hopefully eliminate) any performance overhead imposed on the sequential computation of each task. The resulting design imposes almost no such overhead.
- **Programmability** We wished to minimize the amount of programmer effort required to express a computation in Jade.
- **Expressive Power** We wished to maximize the range of supported parallel applications. In particular, the programmer had to be able to express dynamic concurrency.
- **Implementability** We needed to build prototype implementations of Jade quickly and with limited manpower. Several of our design decisions support this goal.

In the following sections we discuss how these design goals and requirements drove specific design decisions.

2.8.1 Implicit Concurrency

Jade is an implicitly parallel language. Rather than using explicitly parallel constructs to create and manage parallel execution, Jade programmers provide granularity and data usage information. The Jade implementation, and not the programmer, is responsible for managing the exploitation of the concurrency.

The implicitly parallel approach enables the implementation to guarantee safety properties such as freedom from deadlock and, for programs that only declare reads and writes, deterministic execution. Because the implementation controls the parallel execution, it can use concurrency management algorithms that preserve the important safety properties.

Encapsulating general-purpose concurrency management algorithms inside the Jade implementation makes it easier to develop parallel applications. These algorithms allow every Jade programmer to build the parallel program on an existing base of sophisticated software that understands how to exploit concurrency in the context of the current hardware architecture. This software relieves programmers of the cognitive and implementation burden of developing a new concurrency management algorithm for each parallel application.

The implicitly parallel approach also preserves the modularity benefits of the sequential programming paradigm. Jade programmers only provide local information about how each task accesses data. There is no need for programmers to deal with or even understand the global interactions between multiple tasks. This property promotes modularity and makes it easier to maintain Jade programs.

2.8.2 Task Model

We believe that determining an appropriate task decomposition requires application-specific knowledge unavailable to the implementation. The final solution, to have the programmer identify blocks of code whose execution generates a task, gives the implementation the task decomposition with a minimum of programmer effort. Experience with Jade applications suggests that the task decomposition follows naturally given the programmer's high-level, domain-specific understanding of the problem.

2.8.3 Access Specifications

Jade requires the programmer to provide a specification of how each task will access data. The alternative would be to generate the `access specification` sections of the tasks automatically. This would have required the construction of sophisticated program analysis software, which would have significantly complicated the implementation of Jade. Given the programmer's high-level, application-specific knowledge, it is reasonable to assume that

in general the programmer could generate more efficient and precise access specifications than an automatic program analysis tool. Experience with Jade applications demonstrates that, given an appropriate shared object structure, the programmer can easily generate the access specifications.

Access specifications give the implementation advance notice of precisely which objects a task will access. The implementation can exploit this information to apply communication optimizations such as concurrently fetching multiple remote objects for each task. Many other parallel systems [85, 15, 19, 44] only discover when tasks will access data as the tasks actually perform the access, and lack the advance information required to apply sophisticated communication optimizations.

Having the programmer generate the access specification does complicate one aspect of the implementation: the implementation cannot assume that the access specifications are correct, and must detect violations of access specifications. This imposed dynamic checking overhead, which in turn drove the design of local pointers.

2.8.4 Local Pointers

The current Jade implementation dynamically checks every access that goes through a shared pointer. It would have been possible to eliminate much of this overhead by developing a front end that analyzed the program to eliminate redundant checks. The development of such a front end would have complicated the implementation of the language and extended the development time. We instead decided to provide the local pointer mechanism. Although this mechanism requires programmer intervention, it allows all but one dynamic check per object per task to be performed statically and minimizes the amount of effort required to construct an effective Jade front end. Experience with Jade applications suggests that using local pointers effectively is a fairly mechanical process that does not fundamentally affect the difficulty of writing a Jade program.

2.8.5 Pointer Restrictions

In Jade, shared objects cannot contain local or private pointers. The restriction on private pointers exists to prevent one task from accessing another task's private objects. The

restriction on local pointers exists to prevent one task from using another task's dynamic access checks. Recall that all references to local pointers are statically checked. Consider the following scenario. One task aquires a local pointer (generating a dynamic access check) and stores it in a shared object. Another task comes along, reads the shared object to get the local pointer and dereferences the local pointer. The access via the local pointer goes unchecked, and there would be a hole in the access checking mechanism. The restriction prevents this scenario.

The restrictions that ensure shared pointers always point to the beginning of shared objects promote portability. The implementation can implement shared pointers using globally valid identifiers and avoid the problems associated with transferring hard pointers between machines.

2.8.6 Shared and Private Data

Jade segregates pieces of data that multiple tasks can access from pieces of data that only one task can access. The alternative would be to eliminate the distinction and have all data be potentially shared. There are two kinds of private data: local variables allocated on the procedure call stack and dynamically allocated memory. Allowing local variables to be shared would significantly complicate the implementation. Consider the following scenario, which can only happen with shared local variables. A procedure creates a task, which is given a shared pointer to a local variable. The procedure then returns before the task runs. For the task to access valid data, the local variable must be allocated on the heap rather than on the procedure call stack. The implementation would then have to automatically deallocate the local variable when no outstanding task could access the variable.

Allocating potentially shared local variables on the heap would complicate the allocation and deallocation of local variables, which in turn could degrade the serial performance of the system. Allowing shared local variables would not have eased the implementation of any existing or envisioned Jade application. We decided the best point in the design space was to make all local variables private.

The other kind of private data is dynamically allocated data. Making all dynamically

allocated data shared would have much less of an effect on the implementation. Still, it would involve both unnecessary space overhead (for system data structures used with shared objects), unnecessary time overhead (for dynamic access checks) and unnecessary programming overhead (for declaring accesses to private data). Because we wanted to preserve the serial execution performance of the system, we decided to provide dynamically allocated private data.

2.8.7 Commuting Declarations

Commuting declarations make it easier to express certain kinds of applications in Jade and expand the range of potential Jade applications. Many parallel computations contain commuting operations that access externally produced data. Exposing the commutativity allows the implementation to perform each operation as soon as the rest of the computation produces the required data. In the absence of the commutativity information the computation would execute the operations in the original serial execution order. This order would waste concurrency if the computation produced the data required for later operations before the data required for earlier operations.

Commuting declarations also interact synergistically with hierarchical objects to eliminate certain kinds of sequencing constraints. Consider a program that generates a collection of tasks, each of which performs a commuting operation on a dynamically determined child object of a given parent object. Each task would first declare a child access on the parent object, then refine its access specification after determining which child object to access. If the tasks only declared read and write accesses, the implementation would prevent each task from performing its operation until every previous task (in the sequential execution order) had refined its access specification. This approach wastes concurrency when a task waits for previous tasks to refine their access specifications, especially if the previous tasks access different child objects.

When each task declares that the child object operations commute, each operation can execute as soon as the task refines its access specification and there are no other operations on that object in progress. Tasks do not have to wait for previous tasks to refine their access specifications.

In the current implementation it is the programmer's responsibility to map commuting declarations to operations that do in fact commute. If the operations do not commute, the program may execute nondeterministically. In practice we do not expect programmers to have difficulty using commuting declarations deterministically.

2.8.8 Allocation Units and Access Specifications

Jade access specifications are declared in terms of shared objects; each shared object is an allocation unit. Unifying the allocation and access declaration granularities makes it difficult to express algorithms that concurrently read and write different parts of a single object. In such algorithms the basic access granularity is finer than the allocation granularity. The programmer may be able to express the algorithm in Jade by reallocating the object so that the allocation granularity matches the access declaration granularity. The problem is that the programmer must then change all the code that accesses the object to reflect the new structure. Programs that access the same data at different granularities in different parts of the program exacerbate the problem by forcing the programmer to periodically reallocate the data at the new granularity.

An early version of Jade [80] avoided this problem by decoupling the units of allocation and synchronization. The units of synchronization (called tokens) were abstract, and their correspondence with the actual data was completely conceptual. The lack of an enforced correspondence dramatically increased the flexibility of the language. Programmers started dynamically modifying the access declaration granularity of particular pieces of data, using tokens to represent abstract concepts like the right to change the access declaration granularity.

In some respects using the old version of Jade imposed less of a programming burden than using the current version because the programmer never had to change the way the program allocated and accessed data. A major drawback of the old version was that it did not enforce the access specifications. Because the implementation was not aware of the correspondence between data and tokens, it could not check the correctness of the data usage information. Another restriction associated with the lack of an explicit correspondence between data and tokens was that the implementation could not determine

which pieces of data each task would access just by looking at its access specification. The implementation could not automatically implement the abstraction of a single address space on message-passing machines by transferring the data at the access declaration granularity in response to each task's access specification.

Ideally, we would be able to combine many of the advantages of the two versions of Jade by allowing the programmer to create multiple access declaration units per object. The programmer would still allocate data at the original granularity, but the language would allow the programmer to partition each object's data into finer granularity access declaration units. A problem with such an extension is that it complicates the access checking. Each access to an object involves a pointer to that object and an index into the object. In the current scheme only the pointer is checked because all indices are equally valid. In a scheme that allowed a finer access declaration granularity, the implementation would also have to check the index against a data structure storing the valid regions of the object.

The problem would get worse for local pointers. In the current scheme accesses via local pointers involve no access checks and are as efficient as accesses via private pointers. In the absence of sophisticated static analysis, the alternative scheme would require the implementation to check the index on all local pointer accesses. This would significantly degrade the serial performance of tasks that repeatedly accessed shared objects.

2.8.9 Allocation Units and Communication

In the message-passing implementation each object is a unit of communication. Transferring the entire object wastes bandwidth when a task actually accesses only a small part of the transferred object. Making each object a unit of communication also requires that each object be fully resident in the accessing processor's memory module. Machines without virtual memory cannot execute programs that access objects bigger than the physical memory associated with each processor. One of the Jade applications (the Volume Rendering application) actually fails to run on one platform because of this restriction.

One alternative is to distribute fixed-size pieces of objects across the memory modules and transfer pieces of objects on demand. The shared-memory implementation implicitly does this by using the shared-memory hardware, which distributes pieces of objects across

the caches at the granularity of cache lines. Page-based software shared-memory systems apply this principle at the level of pages, using the page fault mechanism to detect references to non-resident parts of objects. Another strategy is to have the front end augment every access to a shared object with a software check to see if the data is locally available. If not, the implementation would automatically generate a message to fetch the accessed piece of the object. All of these strategies allow the system to use the whole memory of the computing environment to store large objects and can drive down the amount of wasted bandwidth.

While each of these strategies addresses a fundamental shortcoming of the current Jade communication strategy, they all have drawbacks. Page-based approaches require the implementation to interact with the paging system of the resident operating system. In many operating systems the implementations of the user-level fault handling primitives are inefficient [7], and some operating systems do not provide these primitives at all. On the other hand, using a strategy that dynamically checked each access would impose substantial overhead on each access to a shared object.

Another alternative is to decompose objects into finer communication units under program control and associate an access declaration unit with each communication unit. Each task would then declare precisely which communication units it would access and the implementation would move and copy data at the access declaration granularity.

Supporting multiple communication units per object would raise several interesting memory management issues. The key feature is that the each object would occupy a contiguous chunk of the address space, but each processor might only access several widely separated communication units. The simplest way to store the communication units would be to allocate local storage on the accessing processor for the entire object and then store each accessed unit into this storage. The data in the accessed units would then be valid, while the rest of the object would be invalid. The approach has the advantage that it preserves the object's original index space. The generated parallel code would access the parts of the object using the same indices that the serial program uses to access the complete object.

The disadvantage of this approach is that it wastes parts of the address space. On machines with no support for sparse address spaces the unaccessed pieces of the object

could occupy a large part of physical memory, causing poor utilization of physical memory. On systems that support sparse address spaces this is less of a concern because the pages holding the unaccessed section of the object would remain unmapped and not occupy physical memory. Even these systems could suffer from internal page fragmentation if the communication units were significantly smaller than the page size or if the communication units did not occupy contiguous parts of the object. In a block decomposition of a matrix, for example, each block occupies a non-contiguous part of the matrix's index space.

Another way to implement multiple communication units per shared object would be to allocate a separate piece of memory for each communication unit and store the unit contiguously in this piece of memory. The pieces of memory could be allocated on demand as the program accessed communication units. This approach would promote good memory utilization, but require the implementation to translate accesses from the object's old index space to the communication unit's new index space. In some cases the implementation could apply sophisticated compiler analysis to perform the translation statically, but in general the translation would have to take place dynamically. Performing the translations dynamically would degrade the performance of serial task code.

2.9 Summary

Jade is fundamentally a declarative language that programmers use to specify how parts of a serial program access data. It is designed to support the exploitation of the task-level concurrency inherently present in parallel computations that have a serial semantics.

Jade is built on the three concepts of shared objects, tasks and access specifications. Objects are the units of synchronization and, on message-passing machines, the units of communication. Tasks are the units of computation. Access specifications bridge the gap between computation and data by declaring how tasks access objects. The Jade implementation analyzes the access specifications to automatically extract the concurrency and manage the communication.

Jade provides a safe, portable programming paradigm for exploiting task-level concurrency. Jade programs do not exhibit the complex failure modes such as deadlock that

characterize explicitly parallel computing environments. Programs that contain no commuting access specification statements are guaranteed to preserve the semantics of the original serial program and to execute deterministically.

Chapter 3

The Jade Implementation

The Jade programmer and the Jade implementation each have particular strengths that are best suited for performing different parts of the process of parallelizing the computation. The programmer's strength is providing the high-level, application-specific knowledge required to determine an effective data and computation granularity. The implementation's strength is performing the analysis required to discover parallel tasks, executing the detailed bookkeeping operations required to correctly synchronize the resulting parallel computation, and providing the low-level, machine-specific knowledge required to efficiently map the computation onto the particular parallel machine at hand.

The Jade language design partitions the responsibility for parallelizing a computation between the programmer and the implementation based on this analysis of their respective strengths. This division means that the Jade implementation encapsulates algorithms that automatically perform many important parts of the parallelization process. The programmer obliviously reuses these algorithms every time he or she writes a Jade program.

We have demonstrated the viability and applicability of these algorithms by implementing Jade on many different computational platforms. Jade implementations currently exist for shared-memory machines such as the Stanford DASH machine [82] and the Silicon Graphics 4D/340 [12], for message-passing machines such as the Intel iPSC/860 [16], and for heterogeneous networks of workstations. While no implementation currently exists for shared-memory machines with incoherent caches such as the Cray T3D, it would be possible to implement Jade on such machines.

The performance of many parallel applications depends on their communication behavior. The data usage information in Jade programs gives the implementation the information it needs to automatically apply communication optimizations. The current Jade implementation exploits the data usage information to apply a scheduling heuristic designed to improve the locality of the computation. This heuristic attempts to execute tasks on processors that have locally available copies of the objects that the task will access. The message-passing implementation also optimizes the computation by concurrently fetching the remote objects that a task will access, using excess concurrency in the application to hide the latency of accessing remote objects, and switching to a broadcast protocol for objects that are accessed by all processors.

Jade presents a restricted model of parallel computation. When appropriate, the implementation uses optimized, special-purpose algorithms tailored for the specific situations that arise in the context of Jade. There are cases, however, when the implementation uses general-purpose algorithms suitable for use in other parallel and distributed systems. Specifically, the implementation contains an algorithm for locating objects in a message-passing system, an efficient consistency mechanism that avoids some of the performance overhead associated with invalidate and update protocols, and scheduling algorithms for use in systems that have advance knowledge about how the computation will access data. It would be possible to extract these algorithms from the Jade implementation and package them for reuse in other systems.

3.1 Overview

In this section we describe the functionality and optimizations that the Jade implementation provides. Strictly speaking, there are two Jade implementations: one for shared-memory platforms and one for message-passing platforms. While each implementation is tailored for its own specific computational environment, the implementations do share many basic responsibilities and mechanisms. Both implementations are completely dynamic, consisting of a run-time system and a simple preprocessor which emits C code. Both implementations perform the following activities to correctly execute a Jade program in parallel.

- **Concurrency Detection** The implementation analyzes access specifications to determine which tasks can execute concurrently without violating the serial semantics.
- **Synchronization** The implementation synchronizes the parallel computation.
- **Scheduling** The implementation assigns tasks to processors for execution.
- **Access Checking** The implementation dynamically checks each task's accesses to ensure that it respects its access specification. If a task violates its access specification, the implementation generates an error.
- **Controlling Excess Concurrency** The implementation suppresses excessive task creation to avoid overwhelming the parallel machine with tasks.

The message-passing implementation has several additional responsibilities associated with implementing the abstraction of a single address space in a message-passing environment.

- **Object Management** The message-passing implementation moves or copies objects between machines as necessary to implement the abstraction of a single address space.
- **Naming** The message-passing implementation maintains a global name space for shared objects, including an algorithm that locates remote objects.
- **Format Translation** In heterogeneous environments the implementation performs the format conversion required to correctly transfer data between machines with different data formats. Appendix A describes how the implementation performs the data format translation.

The scheduling algorithms for both the shared-memory and message-passing implementations contain several mechanisms which optimize the assignment of tasks to processors.

- **Load Balancing** The implementation tracks processor use and dynamically balances the load by scheduling enabled tasks onto idle processors.

- **Locality** Because the implementation knows which objects each task will access, it can apply a locality heuristic. This heuristic is designed to enhance the locality of the computation by executing tasks on processors with locally available copies of accessed objects.

While shared-memory machines manage the movement of data in hardware, message-passing machines allow the software to control this movement. The message-passing implementation exploits this control and its knowledge of how tasks access data to apply the following communication optimizations.

- **Migration** The implementation migrates objects for fast local access.
- **Replication** The implementation replicates objects for concurrent read access.
- **Consistency** The implementation uses an efficient consistency mechanism that avoids the overhead of invalidate or update protocols.
- **Block Transfer** If a task will access a remote object, the implementation generates one communication operation that transfers the entire object.
- **Concurrent Fetches** If a task declares that it will access multiple remote objects, the implementation fetches the objects concurrently.
- **Adaptive Broadcast** The implementation tracks object usage patterns in the computation, switching to a broadcast protocol for objects that all processors access.
- **Object Piggybacking** The implementation eliminates remote access latency and excess messages by piggybacking objects onto task messages.
- **Hiding Latency with Concurrency** The implementation uses excess concurrency to hide latency. It assigns multiple executable tasks to each processor and fetches remote objects for one while executing another.

The remainder of this chapter is structured as follows. Section 3.2 describes the object queue mechanism. This is a key mechanism used by both the shared-memory and message-passing implementations to extract concurrency and synchronize the parallel computation.

Section 3.3 presents an overview of the shared-memory implementation. We devote the majority of this section to describing the shared-memory scheduling algorithm and its locality heuristics.

Section 3.4 presents an overview of the message-passing implementation. This section describes the object migration and replication algorithms, an optimized consistency algorithm based on version numbers, the message-passing scheduling algorithm and its locality heuristic and several communication protocols that keep replicated implementation information consistent. This section also describes a general-purpose entity location algorithm that could be used in other parallel or distributed systems.

Section 3.5 describes several algorithms that the two implementations share. In particular, we describe the access checking algorithm and the algorithm used to suppress excessive task creation.

3.2 Object Queues

In this section we discuss the mechanism that the Jade implementation uses to extract concurrency and synchronize the computation. We first describe the mechanism that the implementation uses for read and write declarations, then generalize to include deallocate, disjoint and commuting declarations.

3.2.1 Read and Write Declarations

There is a queue associated with each object that controls when tasks can access that object. The implementation uses these queues to detect concurrency and synchronize the computation. Each task that accesses an object has an entry in the object's queue declaring the access. Entries appear in the queue in the same order as the corresponding tasks would execute in a sequential execution of the program. The implementation initializes a normal object's queue with an entry that declares all possible deferred accesses for the task that created the object. A child object's queue is initially empty.

Immediate write entries are enabled when they reach the front of the queue. Immediate read entries are enabled when there are only read entries before them in the queue. Deferred

entries are always enabled. The purpose of deferred entries is to prevent later tasks from executing prematurely. A task is enabled (and can legally execute) when all of its object queue entries are enabled.

When a task is created, it inserts an entry into the queue of each object that it declares it will read or write. If the task's parent task has an entry in the object queue, the new task inserts its entry just before the parent task's entry. If the parent task has no entry in the object queue, the object must be a child object and the parent must have an entry in the parent object's queue. In this case the implementation inserts the task's entry at the end of the object queue. This insertion strategy ensures that tasks' entries appear in the object queues in the sequential execution order.

A task may update its queue entries to reflect the changes in its access specification. These changes may cause the task to suspend, or they may cause other tasks to become executable. When a task finishes its execution it removes all of its entries from the object queues. These removals may cause other tasks to become executable.

The shared-memory implementation keeps each object queue consistent by giving each queue operation exclusive access to the queue. In the message-passing implementation queue operations also execute sequentially. The queue migrates as a unit on demand between processors.

3.2.2 Evaluation of the Object Queue Mechanism

The biggest drawback of the object queue mechanism is its monolithic nature. Performing object queue operations sequentially may cause serialization that would not be present with a different synchronization mechanism. Consider a set of tasks that all read the same object. The object queue will serialize the insertion of their declarations into the object queue. If the tasks have no inherent data-dependence constraints, the task queue insertions may artificially limit the parallelism in the computation.

It is possible to break the artificial object queue serialization using a mechanism that hierarchically numbers object versions. Each task could compute which version of each object it would access given only the version numbers of the objects in the parent task's access declaration. A task would run only when the correct versions of the objects it would

access became available. This mechanism would allow multiple parent tasks to create child tasks that accessed the same object with no additional synchronization.

3.2.3 Correctness of Object Queue Mechanism

In this section we discuss why the object queue mechanism correctly synchronizes the computation. There are two aspects to the correctness of a synchronization mechanism for Jade programs: 1) at every point of the computation there is at least one task that can execute (i.e. the synchronization mechanism is deadlock free) and 2) no task reads the wrong value from a shared object. We first consider Jade programs without child objects, then remove this restriction.

If the program contains no child objects, then tasks' entries appear in the object queues in the same order that the tasks would execute in a serial program (the serial entry order property). This is true of the original state of an object queue, which contains one entry from the task that created the object. Subsequent object queue operations preserve the serial entry order property. If a task completes or eliminates an access declaration, the implementation removes its entries from the object queue. The new sequence of entries is then a subsequence of the original sequence of entries, and the property holds.

The only other queue operation is the insertion of a new entry. The new entry appears before the entry of its parent task. In the original sequence all entries which appeared before the parent task entry belonged to tasks which executed before the parent task. The child task's entry appears after all of these entries, which is correct because in the sequential execution the child task executes after all of these tasks. Similarly, the child task's entry appears before all of the entries after its parent task's entry, and the child task should execute before all these tasks. Finally, the child task's entry appears before its parent task's entry. This is again correct because the child task should execute before the remainder of its parent task.

The serial entry order property implies the deadlock freeness of the object queue mechanism. The sequential execution order is a total order on the tasks, so at every point in the computation there is at least one task such that every one of that task's entries are at the front of their object queues. This task can execute.

To establish that no task reads the wrong value from a shared object, we establish that all parallel executions preserve the relative order of reads and writes to the same object. We establish that if a write occurs before a read to the same object in a sequential execution, the write will occur before the read in all parallel executions. A similar argument establishes that the read will take place before any subsequent (in the sequential execution order) writes. See [108] for a more formal treatment of this argument.

Consider a write that happens before a read in a sequential execution. There are several cases: 1) the same task performs both the read and the write, 2) the task that performs the write is an ancestor of the task that performs the read, 3) the task that performs the read is an ancestor of the task that performs the write, and 4) two different tasks perform the read and the write, and neither is an ancestor of the other. We show that in all parallel executions the write occurs before the read.

In case 1 the operations of the task always execute in the same order as in the serial execution, so the write occurs before the read. In case 2 the parent task performs the write before it creates the reading task. In case 3 the first task on the ancestor chain of tasks from the parent task to the writing task is created before the read is performed. This child task and every task in the ancestor chain inserts a write entry into the object queue. These entries occur before the parent task's entry. By the time the parent task attempts to perform the read, either the write will have occurred, or there is and will continue to be a write entry from the ancestor chain in the queue until the write is performed. This write entry will prevent the parent task from performing the read before the write.

In case 4 the writing task and the reading task share at least one common ancestor task. Find the least common ancestor task (the ancestor task with no child task that is also a common ancestor task). This ancestor task has two child tasks, one of which (the first task) either performs the write or is an ancestor of the writing task. The second task either performs the read or is an ancestor of the reading task. The first task inserts a write entry into the queue and is created before the second task, which inserts a read entry into the queue. Because the two tasks have the same parent the write entry appears in the queue before the read entry.

All of the ancestors of the writing task between the writing task and the first task must insert write entries into the queue. By the time the reading task is created and inserts its

entry into the queue, either the write will have occurred, or there is and will continue to be a write entry before it in the queue until the write occurs. This write entry will prevent the read task from performing the read until the write task performs the write.

We now extend the argument to handle child objects. For cases 1 and 2 the argument is identical to the one above. For cases 3 and 4 the argument is as follows. Find the least common ancestor task of the writing task and the reading task. There is a chain of tasks from the ancestor task to the write task. If the ancestor declares that it will both read and write the object when it creates the first task on the write task chain, the argument above for normal objects holds. Otherwise the ancestor task declares that it will both read and write an ancestor of the accessed object. Next consider the chain of tasks from the ancestor task to the read task. Both this read chain of tasks and the write chain of tasks discussed above walk down the chain of objects from the ancestor object to the accessed object. At each step of the object chain each task inserts its entry at the end of the object's queue. We will now argue that, for each object queue, the write chain inserts its entry into the queue before the read chain does.

The program starts walking down the write task chain before it starts walking down the read task chain, so the write entry is inserted into the first object's queue before the read entry. At every step of the object chain the write task chain has a write entry in the current object's queue. As the read task chain walks the object chain, it inserts read entries into the object queues. To take each step from a parent object to a child object, the stepping read task must have an immediate entry in the parent object's queue. If the write chain has yet to take the corresponding step, there will still be a write entry in the parent object's queue before the read chain's read entry. This write entry will prevent the read task from executing, and the read task chain will never pass the write task chain on the way down the object chain to the accessed object. The writing task will insert its entry into the accessed object's queue before the reading task, and will perform the write before the read.

Given the above argument, it is possible to see why the implementation enforces the restriction that each task declare that it will access at most one object in a hierarchy of objects. Without this restriction, the ancestor task could have entries in several of the object queues, and the entries could get out of order as the task chains walk the object chain.

For child objects we relax the serial entry order property. It is possible for two tasks

that read the same object to get out of order in the object queue. This of course does not affect when the tasks acquire the right to access the object because Jade allows concurrent reads, and both tasks will acquire the right to read the object at the same time. So, for child objects the implementation only preserves the serial entry order property for writes relative to reads. This is still enough for the deadlock freeness argument presented above to go through.

3.2.4 Extensions for Deallocate Declarations

Deallocate declarations also insert entries into the object queue. A deallocate entry is enabled when it is the first entry in the queue. The correctness condition for deallocations is that all previous tasks that access the object complete their accesses before the object is deallocated. An argument similar to the one in Section 3.2.3 establishes that the object queue mechanism preserves the serial execution order for deallocations relative to other accesses to the same object.

3.2.5 Extensions for Commuting Declarations

Commuting declarations introduce an extra level of complexity into the synchronization algorithm. There are two kinds of synchronization constraints that apply to commuting declarations: serialization constraints (the implementation must preserve the serial order for commuting accesses relative to other accesses) and exclusion constraints (the commuting accesses must execute in some serial order).

The implementation enforces serialization constraints using the object queue mechanism. Each commuting declaration inserts an entry in the object queue. Child and immediate commuting entries become enabled when there are only commuting entries before them in the queue.

The implementation enforces exclusion constraints with an exclusion queue. Associated with each object is an exclusion queue. If a task declares an immediate commuting access it will insert an exclusion entry into the corresponding exclusion queue before acquiring the right to access the object. The entry is enabled when its entry becomes the first entry in the exclusion queue. The task itself becomes enabled when all of its object queue and exclusion

queue entries are enabled. The task removes its exclusion entry when it completes or uses a `with` construct to eliminate the immediate commuting declaration.

The implementation avoids deadlock by properly sequencing the queue insertions. When a task is created it inserts all of its object queue entries into the object queues according the algorithm in Section 3.2.1. It then waits for all of its object queue entries to become enabled. It then sorts its immediate commuting entries and progressively inserts the exclusion entries into the corresponding exclusion queues. It inserts each exclusion entry only after all previous exclusion entries are enabled.

We first establish that the mechanism outlined above is deadlock free. At any point in the execution there is a set of tasks whose object queue entries are all enabled. By the serial entry order property described in Section 3.2.3 this set is not empty. These tasks are either enabled or in the process of inserting exclusion queue entries and waiting for them to be enabled. Conversely, if a task has any entries in exclusion queues, all of its object queue entries are enabled. The Jade implementation ensures this property by preventing a task from declaring any immediate accesses or creating any child tasks while its access specification contains an immediate commuting declaration. These are the only two declarations that can cause one of the task's enabled entries to become not enabled.

In effect, the exclusion queues implement a mutual exclusion lock on each object, and the tasks acquire the locks in the sort order. Every task that holds an object lock is either enabled or will become enabled as soon as it acquires the rest of its locks. Because the tasks acquire locks in the sort order, one task must eventually acquire all of its locks and become enabled.

We next address the correctness conditions. The exclusion queue mechanism ensures that commuting updates to the same object execute in some serial order. An argument similar to the one in Section 3.2.3 establishes that the object queue mechanism preserves the serial execution order for commuting accesses relative to other kinds of accesses.

3.3 The Shared-Memory Implementation

In this section we discuss the implementation of Jade for multiprocessors with hardware support for shared memory. Because the hardware implements the Jade abstraction of a

single address space, the implementation is only responsible for finding the concurrency, synchronizing the computation, and mapping the tasks efficiently onto the processors.

3.3.1 A Task Lifetime

In this section we summarize the dynamic behavior of the shared-memory implementation by tracing the lifetime of a task. Figure 3.1 gives a state-transition diagram of a task's lifetime. We describe the activities that take place in each state in turn.

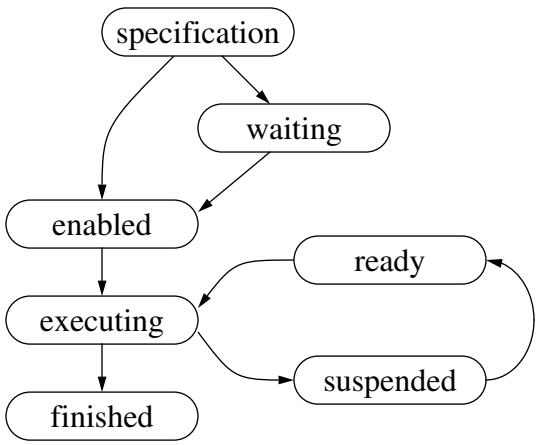

Figure 3.1: Shared-Memory State-Transition Diagram

3.3.1.1 The Specification State

To execute a `withonly` construct, the implementation first allocates a task data structure. This data structure contains a pointer to the task body code, space for the task parameters, and an initially empty access specification. The implementation executes the `access specification` section to generate the new task's initial access specification and copies the parameters into the task data structure. It then inserts the task's entries into the object queues according to the algorithm described in Section 3.2. If the task's access specification is enabled, it enters the enabled state. Otherwise, it enters the waiting state.

3.3.1.2 The Waiting State

A task in the waiting state cannot execute because its initial access specification has yet to be enabled. When the access specification is enabled the task enters the enabled state.

3.3.1.3 The Enabled State

A task in the enabled state is ready to execute, but is waiting to be assigned to a processor for execution. See Section 3.3.4 for a description of the scheduling algorithm.

3.3.1.4 The Executing State

The implementation executes a task by allocating a stack for the task and starting a thread that runs the task body on this stack. The task may change its access specification, causing the implementation to update the object queue information. An executing task may suspend because of excessive task creation, because of a conflicting child task access declaration, or because it executes a `with` construct. In these cases the implementation saves the state of thread executing the task and switches to another enabled or ready task if one exists.

3.3.1.5 The Suspended State

A task in the suspended state will eventually become able to execute again, at which point it enters the ready state.

3.3.1.6 The Ready State

A task in the ready state is ready to resume its execution. In the current implementation the task will always resume on the processor that first executed it.

3.3.1.7 The Finished State

Eventually the task finishes. The implementation removes the task's declarations from the object and exclusion queues. These removals may enable other tasks' declarations, and some of the tasks may enter the enabled state. The implementation then deallocates the task data structure and stack for use by subsequently created tasks.

3.3.2 Locality and Load Balancing

The shared-memory scheduler mediates a trade-off between locality and load balancing. This trade-off occurs when the scheduler finds itself with an idle processor and an enabled task that the locality heuristic prefers to execute on a busy processor. In this case it may not be clear whether it is better to execute the task on the idle processor (where the task may take longer to run) or wait for the busy processor to finish its current task (and waste the computational power of the idle processor).

The implementation always chooses load balance over locality. It never refuses to execute a task on an idle processor because of locality concerns. The scheduler has no precise information about how much faster a task will execute on a processor that has its objects available locally, nor does it know how long it will be until a busy processor goes idle. The benefits of leaving a processor idle are vague and uncertain, while the drawback is clear: the loss of the processor for the period of time it is idle.

3.3.3 Shared-Memory Systems

Different shared-memory machines have different memory system characteristics. The implementation adjusts to these different memory system characteristics by adjusting the locality heuristic to the characteristics of the machine at hand. We therefore motivate the discussion of the shared-memory locality heuristics by summarizing the locality issues associated with each of the following kinds of memory systems: 1) bus-based systems with a single memory module and per-processor caches (like the Silicon Graphics 4D/340), 2) distributed-memory systems with multiple memory modules and caches (like the Stanford DASH machine), 3) distributed-memory systems without caches (like the BBN Butterfly [13]) and 4) cache-only-memory systems (like the KSR1 [73]).

3.3.3.1 Bus-Based Systems

Bus-based systems have a roughly uniform access time from any processor to the main memory. The access time for cached data is typically significantly lower than the access time to main memory. Executing tasks on processors with cached copies of the accessed data enhances the locality of the computation.

3.3.3.2 Distributed-Memory Systems with Caches

Distributed-memory systems may have non-uniform access times from processors to memory modules. Typically, each processor has a local memory module, and it takes less time for that processor to access its local memory module than to access remote memory modules. There are two possible locality effects. As for the bus-based systems described in Section 3.3.3.1 above, executing tasks on processors with cached copies of the accessed data enhances the locality of the computation. But it may also enhance the locality of the computation to execute tasks on processors whose local memory modules contain the accessed data. In this case accesses that are not satisfied in the cache will be satisfied in local rather than remote memory, driving down the access time.

3.3.3.3 Distributed-Memory Systems without Caches

For machines that do not cache remote data the only way to enhance the locality of the computation is to execute tasks on processors whose local memory modules contain the accessed data.

3.3.3.4 Cache-Only-Memory Systems

The final class of systems we consider is the cache-only-memory systems. These systems differ from the distributed-memory systems discussed above in that they treat all of memory as a cache. In effect, each piece of data moves to the memory module closest to the last processor that accessed it. For these systems an appropriate way to enhance locality is to execute each task on the processor that last accessed the data that the task will access. The previous access will have fetched the data to that processor, and the data should be stored at least as close to that processor as to any other.

3.3.4 The Shared-Memory Scheduler

In this section we present the shared-memory scheduling algorithm in detail. The scheduler can use one of two locality heuristics: the cache locality heuristic and the memory locality heuristic. The cache locality heuristic is designed for systems in which data tends to be

most quickly accessible from the last processor to access it. This heuristic is therefore appropriate for bus-based systems and cache-only-memory systems. The memory locality heuristic is designed for systems in which data tends to be most quickly accessible from the processor closest to the memory module in which the data is allocated. This heuristic is therefore appropriate for distributed-memory systems without caches. On distributed-memory systems with caches data may be most quickly accessible either from the last processor to access it (if the data is still cached) or from the processor in whose memory module the data is allocated (if the data is not remotely cached). For reasons explained below in Section 3.3.5, the current implementation for distributed-memory systems with caches uses the memory locality heuristic.

The scheduler assigns tasks to processors using a distributed task queue algorithm. There is one queue associated with each processor. When a task can legally execute, it is inserted into one of the task queues. The implementation structures the task queue associated with each processor (called the processor task queue) as a queue of object task queues. There is one object task queue associated with each object; the object task queue is in turn a queue of tasks. Figure 3.2 contains a picture of these data structures. There is one version of these data structures for each processor.

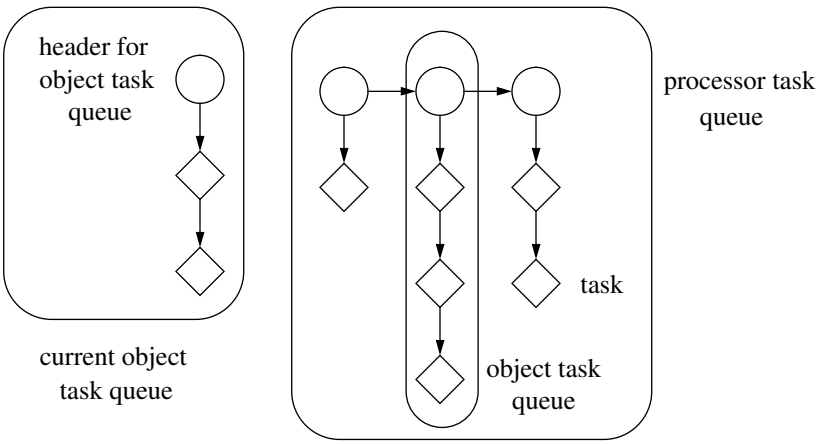

Figure 3.2: Task Queue Data Structures

Each object task queue is initially owned by the processor that owns the corresponding object (i.e. the processor in whose memory module the object was allocated). The main difference between the cache and memory locality heuristics is that the cache locality

heuristic may occasionally change the ownership relation by moving object task queues from one processor to another. The memory locality heuristic never changes the ownership relation.

Each task has a locality object; in the current implementation the locality object is the first object that the task declared it would access. When the implementation determines that a task can legally execute, it inserts the task into the object task queue associated with that task's locality object. If the object task queue was empty before the task was inserted, it inserts the object task queue into the processor task queue of the processor that owns that object task queue.

Each processor maintains a current object task queue. When a processor goes idle, it needs to choose a new task to execute. It first tries to execute the first task in the current object task queue. If the current object task queue is empty, the processor tries to make the first object task queue in the processor task queue the current object task queue. It then executes the first task in the new current object task queue. If the processor task queue is empty, it cyclically searches the task queues of other processors. Both heuristics first search the processor task queues. If the cache locality heuristic finds a non-empty processor task queue it removes the last object task queue and makes that object task queue the idle processor's current object task queue. The idle processor then executes the first task from the moved object task queue. If the memory locality heuristic finds a non-empty processor task queue it removes the last task from the last object task queue; the idle processor will execute this task. In this case the object task queue remains associated with its original processor.

If all processor task queues are empty the idle processor cyclically searches the current object task queues of other processors. In both heuristics, if the idle processor finds a non-empty current object task queue it removes the last task from the current object task queue and executes it. The idle processor continues cyclically searching both the processor task queues and the current object task queues until it finds an executable task or the computation completes.

There is one delicate implementation detail about this dynamic load balancing algorithm. A processor never removes a task from another processor's task queue if the other processor is idle. Parallel programs occasionally go through concurrency-poor regions of

the computation in which most of the processors are idle. In this situation most of the processors are cyclically searching other processors' task queues for tasks to execute. At the end of the concurrency-poor region new tasks start appearing in task queues. If each processor takes the first task it finds, the tasks will get assigned to processors arbitrarily and the locality heuristic will have no effect. The current algorithm eliminates this effect if the locality heuristic evenly distributes the new tasks across the processors. In this case the idle processors will either refuse to take a remote task (because the other processor is idle) or find the task queue of the other processor empty (because the processor has removed its task and started executing it). The processor will eventually check its own task queue and remove and execute the task that the locality heuristic assigned to it. This strategy makes the locality heuristic work well during the transition from concurrency-poor to concurrency-rich regions of the computation. It also eliminates contention on the lock that controls access to the task queue.

3.3.5 Discussion of the Locality Heuristics

Both heuristics attempt to enhance the locality of the computation by identifying a locality object for each task and executing the task on the processor that can most quickly satisfy references to the locality object. Both heuristics also attempt to execute tasks with the same locality object consecutively on the same processor. For machines that cache shared objects, this consecutive execution enhances the probability that references to the locality object will be satisfied in the cache. The idea is that the first task will fetch the locality object into the cache. When successive tasks access the locality object the accesses will then hit in the cache, eliminating expensive memory accesses. If the heuristic allowed other tasks to execute between tasks with the same locality object, the other tasks would fetch other objects. These other objects might then eject the locality object from the cache, destroying the locality of the task execution sequence.

For distributed-memory systems with caches, it is not clear which locality heuristic will work best. It may be better to assign a task either to the last processor that accessed its locality object (in hopes that the locality object will still be resident in its cache), or to a processor whose memory module contains the locality object (so that references that do not

hit in the cache can be satisfied out of the local memory rather than a remote memory). For such systems the current Jade implementation uses the memory locality heuristic. Remote processors will only steal individual tasks, and there is a permanent bias towards executing tasks on the processor whose memory contains the locality object. The rationale is that the coarse-grain computations for which Jade was designed may tend to access large objects that never take up lasting residence in the cache.

All of the architectures considered above except distributed-memory machines without caches replicate data for concurrent read access. The current locality heuristics do not take this replication into account. They attempt to execute each task on the processor that owns the locality object's task queue, rather than attempting to execute the task on any of the processors with locally available copies of the locality object's data. Ignoring the fact that a replicated object's data may be close to multiple processors could hurt the computation in the presence of an unbalanced assignment of tasks to processors. In this case the current dynamic load balancing algorithm will balance the load by transferring tasks arbitrarily to idle processors, and not attempt to place tasks on processors close to replicated copies of the locality object.

3.3.6 Extensions for Incoherent Caches

In this section we have assumed that the hardware fully implements the abstraction of a single shared address space. There are machines, however, that only partially implement this abstraction. These machines automatically fetch and cache remote memory, but rely on software to keep the caches consistent. While no Jade implementation currently exists for such machines, using Jade could substantially improve the programming environment.

The most difficult programming problem with using these machines is determining when to generate the cache flushes required to preserve consistency. Because the Jade implementation knows how tasks access data, it can automatically generate these cache flush operations. The programmer would simply use the Jade abstraction of a coherent shared address space, and be oblivious to the complexities introduced by the lack of hardware support for coherent caches.

The Jade implementation could use the following cache flush algorithm to guarantee

the consistency. When a task finished writing an object, the implementation would flush the cache containing local copies of that object's data. Before executing a task that reads an object, the implementation would determine if the object was written since the processor last read it or flushed its cache. If so, the implementation would flush the processor's cache before executing the task.

3.3.7 Summary

The shared-memory implementation extracts the concurrency and schedules the parallel tasks onto the processors for execution. The scheduling algorithm applies a locality heuristic that attempts to improve the locality of the computation. Each task has a locality object; the locality heuristic attempts to schedule the task onto the processor that owns the object. There are two variants of the locality heuristic. In the cache variant the processor that most recently wrote the object is the owner. In the memory variant the owner is always the processor in whose memory the object was allocated. The cache locality heuristic is suitable for bus-based multiprocessors and cache-only-memory machines, while the memory locality heuristic is suitable for distributed-memory machines.

3.4 The Message-Passing Implementation

The responsibilities of the message-passing implementation are a superset of the responsibilities of the shared-memory implementation. Like the shared-memory implementation, the message-passing implementation must discover the concurrency, synchronize the computation and map the tasks efficiently onto the processors. The message-passing implementation has the additional responsibility of using low-level message-passing operations to implement the Jade abstraction of a single address space. The fact that Jade runs in heterogeneous environments complicates the implementation of this abstraction because the implementation must perform the data format translation required to maintain a coherent representation of the data.

3.4.1 A Task Lifetime

In this section we summarize the functionality of the message-passing implementation by tracing the lifetime of a task. Figure 3.3 gives a state-transition diagram of a task's lifetime.

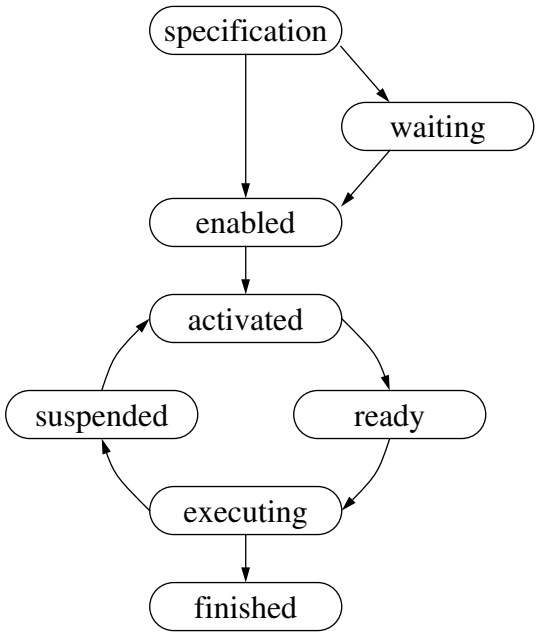

Figure 3.3: Message-Passing State-Transition Diagram

3.4.1.1 The Specification State

When a task is created, it starts out in the specification state. The implementation executes its access specification and copies the parameters into the allocated task data structure. It then inserts the task's entries into the object queues. See Section 3.4.3 for a description of the algorithm that inserts access specifications into remote object queues. If the task's access specification is enabled, it enters the enabled state. If the task must wait to access some of the objects, it enters the waiting state.

3.4.1.2 The Waiting State

A task is in the waiting state if its initial access specification has yet to be enabled. Whenever the object queue enables a task's access declaration it informs the task by sending it a message (if the task and object queue are on different processors), or by performing the operation locally (if the task and object queue are on the same processor). Eventually all of the task's access declarations become enabled, at which point the task enters the enabled state.

3.4.1.3 The Enabled State

A task is in the enabled state if it has yet to execute, but its initial access specification has been enabled. The scheduler will eventually assign the task to a processor for execution. See Section 3.4.2 for a description of the scheduling algorithm. When the implementation assigns a task to a processor for execution, it packs the task data structure into a message and transfers the task to the executing processor. At this point the task enters the activated state.

3.4.1.4 The Activated State

An activated task's synchronization constraints have been enabled and the scheduler has assigned the task to a processor for execution. The implementation cannot yet execute the task, however, because the task may need to access some objects that are not locally available. The implementation therefore fetches the non-local objects by sending request messages to processors that have locally available copies of the objects. The processors respond by sending a copy of the object to the requesting processors. When all of the messages containing remote objects for the task arrive back at the processor, the task enters the executable state. Section 3.4.7 describes how the implementation determines, for each object, which processor has a locally available copy of that object. Because the implementation replicates mutable objects for concurrent read access, the Jade implementation must solve the consistency problem. Section 3.4.7 presents the details of the Jade consistency mechanism.

3.4.1.5 The Executable State

Each processor maintains a local queue of executable tasks. When the processor goes idle it fetches a task from this queue and executes it. The task then enters the executing state.

3.4.1.6 The Executing State

An executing task may change its access specification, causing the implementation to generate messages that update remote object queue information. An executing task may suspend because of excessive task creation, because it creates a child with a conflicting access specification, or because it executes a `with` construct.

3.4.1.7 The Suspended State

A task in the suspended state will eventually become able to execute again, at which point it enters the activated state. In the current implementation the task will always resume on the processor that first executed it.

3.4.1.8 The Finished State

When a task finishes, it must remove all of its access declarations from the object queues. The implementation sends the completed task back to the processor that created it and issues the queue operations there. The hope is that the object queues will still be on the creating processor (the creating processor fetched the queues when it created the task), and the queue removals will take place locally.

3.4.2 The Message-Passing Scheduler

The message-passing implementation uses a centralized scheduling algorithm. This algorithm dynamically balances the load and employs a locality heuristic which attempts to minimize the amount of object traffic. It also assigns multiple enabled tasks to each processor so that the processor can use excess concurrency present in the computation to hide the latency of remote access. This section presents the scheduling algorithm in detail.

The scheduling algorithm is optimized for the case when the main processor (the processor running the main thread of control) creates all of the tasks in the computation. Although the computation will execute correctly if other processors create some of the tasks, the algorithm may fail to generate a reasonable load balance. We chose to implement a centralized algorithm rather than a distributed algorithm because it reduced the implementation time and because in the current set of Jade applications all tasks are created on the main processor.

Each task has a locality object; in the current implementation the locality object is the first object that the task declared it would access. Each object has an owner (the last processor to write the object). The owner is guaranteed to have a copy of the latest version of the object. The dynamic load balancing algorithm attempts to execute each task on the owner of its locality object. This processor is called the task's preferred processor. If the task executes on this processor, it can access the local version of the locality object and will not have to fetch the object from a remote processor.

Like the locality heuristics in the shared-memory implementation, the locality heuristic in the message-passing implementation ignores replication. When the implementation is forced to assign a task to a processor that does not own its locality object, it makes no attempt to assign the task to a processor that has a copy of the locality object.

The main processor (the processor running the main thread of the computation) attempts to keep each processor supplied with a target number of executable tasks. When a task on the main processor enters the enabled state, it first checks if any processor wants a task (i.e. has fewer tasks than the target number). If not, the implementation puts the task into the pool of unassigned tasks at the main processor. The implementation will eventually send the task to a processor for execution when the load drops. If any processor has fewer than the target number of tasks, the implementation will assign the enabled task to one of the least-loaded processors (i.e. the processors with the fewest tasks) and send the task to the processor for execution. If the preferred processor is one of the least-loaded processors, the implementation assigns the task to the preferred processor. Otherwise, the implementation chooses one of the least-loaded processors arbitrarily. This algorithm eagerly sends newly enabled tasks to processors for execution until all processors have the target number of tasks.

When a remote processor finishes a task it got from the main processor, it informs the main processor. The main processor then checks its pool of unassigned tasks. If the pool is not empty the implementation will assign one of the tasks in the pool to the remote processor for execution. If some of the tasks prefer to execute on the remote processor, the implementation sends one of them to the remote processor for execution. Otherwise, it gives the processor a task that prefers to execute on another processor.

When the remote processor receives the message containing the enabled task, it sends out messages requesting the task's remote objects. The processors that own the objects will then send back messages containing the requested objects. In the best case the processor will be executing another task when the new task arrives. The processor can then continue to execute the old task, hiding the latency of fetching the new task's remote objects. When the old task finishes, the remote objects may have arrived and the new task can immediately execute.

There are several trade-offs between the dynamic load balancing, locality and latency hiding aspects of this algorithm. There is the standard locality versus load balancing trade-off. If a task's locality processor is busy and another processor is idle, it may be faster to transfer the locality object to the idle processor and execute the task there. On the other hand, it may be faster to execute the task on its locality processor and waste the idle processor. The current algorithm always chooses load balance over locality.

There is also a trade-off between latency hiding and dynamic load balancing. Once a task has been assigned to a processor for execution it will execute on that processor. For dynamic load balancing purposes, the implementation should delay its assignment of tasks to processors as much as possible while ensuring that each processor has at least one task to execute. Giving processors multiple enabled tasks depletes the pool of unassigned tasks, which limits the implementation's ability to balance the computational load by assigning tasks to newly idle processors. The implementation may find itself in the uncomfortable position of having given away all of its enabled tasks to processors with several enabled tasks, only to have no enabled tasks for a newly idle processor.

Assigning multiple enabled tasks to each processor directly conflicts with the need to delay task assignment to enhance the effectiveness of the load balancing algorithm. In some programs the locality heuristic eliminates this conflict by evenly distributing tasks

to processors. The ultimate solution to this problem, however, would be to implement a distributed dynamic load balancing algorithm that could redo the task assignment in the face of an unbalanced load. The implementation could then promote the use of concurrency to hide latency by aggressively assigning tasks to processors for execution. If the load became unbalanced the implementation could reassign tasks from loaded processors to idle processors. The main cost would be the unnecessary communication caused by moving data to one processor on behalf of a task, only to have the dynamic load balancer reassign the task to another processor.

The most severe potential drawback of the centralized algorithm is that it does not take remotely created tasks into account. If a task is created on any processor other than the main processor, it will execute there regardless of how unbalanced the load may be. The load balancing algorithm may also assign tasks created on the main processor to a remote processor even though the remote processor already has an excess of locally created tasks.

3.4.3 The Object Queue Protocol

In the current Jade implementation each object queue resides completely on one processor. When a processor must perform an operation on a remote queue, the implementation can either move the queue to the processor and perform the operation locally, or forward the operation to the processor that owns the queue and perform the operation there. The implementation currently moves the object queue only when it inserts a new entry into the queue. The implementation performs the other queue operations remotely. Section 3.4.4 describes the precise algorithm the implementation uses to find a remote queue.

The implementation replicates access declaration information in both the task data structure and the object queue data structure. The implementation keeps this information coherent by passing messages between the processors that own the task data structure and the object queue data structure. Conceptually, the task and object queue send messages to each other when the information changes. We define the specific protocol below.

When a task eliminates an access declaration, it sends a message to the object queue informing it of the elimination. The task informs the object queue of no other access declaration modifications.

When an object queue enables a declaration, it sends a message to the task. The object queue also sends such a message when the task declares a deferred access and the object queue enables the corresponding immediate or child access declaration. The object queue must send these messages because the task does not inform the object queue when it changes deferred access declarations to immediate or child access declarations. In effect, the implementation takes advantage of the access declaration semantics to use a relaxed consistency protocol for the replicated access declaration information.

The object queue protocol must work for networks that reorder object queue messages. This reordering does not affect messages from tasks to object queues. Once a task has declared that it will access a given object, its access declaration monotonically decreases for that object. The effects of messages from tasks to object queues therefore commute. Because a child task's access specification can conflict with its parent task's access specification, the parent task can lose an enabled access declaration. This lack of monotonicity forces the implementation to recognize and compensate for reordered operations or messages from object queues to tasks.

The potential problem arises with combinations of delayed messages that give a task the right to perform an access and child task creations that revoke the task's right to perform the same access. If the system does not recognize and discard out-of-date messages from object queues to tasks, it may prematurely execute a task. For example, a task may declare a deferred write access to an object. A remote object queue may give the task the right to write the object, so it sends the task a message informing it of the change. The task may then create a child object which declares that it will write the object. The creation of this child task revokes the parent task's right to write the object. The parent task may next convert its deferred access declaration to an immediate access declaration and suspend waiting for the child task to finish its write. Sometime later the network may finally deliver the message granting the task the right to write the object. The implementation must realize that the message is out of date, and not enable the parent task's access declaration.

The implementation recognizes out-of-date messages using sequence numbers. Each object queue contains a counter. The implementation increments this counter every time it performs an operation on the queue or sends a message from the queue to a task. Each message from the queue to a task contains the current value of the counter. For each

declaration the task stores the value of the queue counter when the queue last updated that declaration's information. When the declaration gets a message from a queue, it compares the counter in the message to its stored counter value. If the message counter is less than the declaration counter, the message is out of date and is discarded. In the example above, the insertion of the child task's declaration into the object queue would increment the queue counter. The revocation of the right to write the object caused by the creation of the child task would store the new counter value into the parent task's access declaration. The implementation would recognize the message's out-of-date counter value and discard the message.

3.4.4 Locating Remote Entities

The Jade implementation deals with several entities (object queues and tasks) that can move from processor to processor. When the implementation needs to perform an operation on a given entity, the entity may reside on a remote processor. In this case the implementation must locate the remote entity and perform the operation. There are two kinds of operations: potentially remote operations that the implementation can perform on any processor holding the entity, and local operations that the implementation must perform on the processor that issued the operation. When a processor issues an operation on an object that it holds, it just performs the operation locally. For a potentially remote operation on an entity held by another processor, the implementation packs the operation into a message and sends the message to the processor holding the entity. For local operations on an entity held by another processor, the implementation sends out a request message for the entity. The processor holding the entity will eventually receive the request and move the entity to the processor that issued the request. When the entity arrives at the requesting processor it performs the local operation.

The implementation locates objects using a forwarding pointer scheme. At each processor the implementation maintains a forwarding pointer for each entity that the processor has ever held. This forwarding pointer points to the last processor known to have requested the entity. The implementation locates an entity by following these forwarding pointers until it finds the processor holding the entity. If a processor with no forwarding pointer needs to

locate an entity, the implementation extracts the number of the processor that created the entity (this number is encoded in the entity identifier) and forwards the operation or request to this processor (the entity is initially located on that processor).

We first discuss the request protocol, which is designed to minimize the number of hops required to locate an object. When a processor receives or issues a request for an object held by another processor, it checks its forwarding pointer. If the forwarding pointer points to another processor, it forwards the request to that processor and changes its forwarding pointer to point to the requesting processor. If the forwarding pointer points to itself, it has already requested the entity but the entity has yet to arrive. In this case the implementation appends the request to a local queue of requests for that entity.

When the request arrives at the processor holding the object, several things may happen. If the processor is performing an operation on the object, the implementation appends the request to the entity's request queue. If the processor has no pending operations, it resets its forwarding pointer to point to the requester and sends the entity to the requester.

When a processor finishes all of the operations that it can perform on an entity (this includes potentially remote operations and its own local operations), it checks the entity's request queue. If it is not empty, it sends the entity and its request queue to the first processor in the queue and resets its forwarding pointer to point to that processor. If the request queue is empty the processor continues to hold the object.

When an entity arrives at a processor that requested it, the processor performs all of its pending potentially remote and local operations. It then appends its request queue to the entity's request queue, and either forwards or continues to hold the entity as described above.

When a processor receives or issues a potentially remote operation on an entity that it holds, it performs the operation. If it does not hold the entity, it checks its forwarding pointer. If the forwarding pointer points to another processor, it forwards the operation to that processor. If the forwarding pointer points to itself, it has already requested the entity but the entity has yet to arrive. In this case the implementation appends the operation to a local queue of potentially remote operations. The implementation will perform these operations when the entity arrives.

The algorithm outlined above is a general-purpose mechanism for locating entities in

a message-passing system. It could be used in other systems any time the system must locate any piece of migrating state or information in the system. The algorithm above has the drawback that once a processor has requested an entity, it must maintain a forwarding pointer for that entity for the rest of the computation. In the worst case each processor may have one forwarding pointer for each object in the computation and the forwarding pointers may consume too much memory. It is possible to adjust the algorithm so that processors may discard forwarding pointers. The system could then discard pointers when memory becomes tight, or it could devote a fixed amount of space to the forwarding pointer table.

Here is the adjustment. Each entity maintains a counter. Every time the entity moves from one processor to another, it increments the counter. Each forwarding pointer contains a copy of the counter value when it forwarded the entity. Whenever a processor wishes to discard a forwarding pointer, it sends a message to the processor whose number is encoded in the entity identifier. This message informs the processor that its forwarding pointer should point to the processor in the discarded forwarding pointer. If the counter value of the home processor's forwarding pointer is less than counter value of the forwarding pointer in the message, the home processor changes its forwarding pointer to the forwarding pointer in the message and updates its counter. Otherwise, the implementation discards the message. If a processor receives a request and has no forwarding pointer for the request, it forwards the message to the home processor. When a processor forwards a request and resets its pointer to point to the requesting processor, the counter on the pointer stays the same. The only restriction is that the home processor must always maintain its forwarding pointer. While this algorithm may lead to transient cycles if the network reorders messages, eventually the cycles will resolve and all requests will eventually locate the object queue.

3.4.5 Communication and Synchronization

Because the Jade implementation knows ahead of time which remote objects each task will access, it can fetch the objects concurrently. Fetching objects concurrently is another example of how the Jade implementation can exploit the data usage information present in Jade programs to optimize the communication. Many parallel languages provide no mechanism that programmers can use to express ahead of time which pieces of data a task

will access. The implementations of these languages must therefore serially fetch remote data as the tasks access that data.

The Jade implementation also separates the satisfaction of synchronization constraints from the satisfaction of object access requirements. The satisfaction of a task's synchronization constraints causes the transition from the waiting state or suspended state to the enabled or activated state. The satisfaction of a task's object access requirements causes the transition to the executable state.

Some systems combine the satisfaction of synchronization constraints with satisfaction of data access requirements. In these systems the message containing the data also gives the task the right to access the data. This is a potentially more efficient mechanism because it may generate fewer messages. But using such a system in the current Jade implementation could easily generate useless and/or premature object movement.

With the current scheduling algorithm, a task typically changes processors when it takes the transition from the enabled state to the active state. All of the synchronization messages must go to the processor that owns the task in the enabled state; all of the object messages must go to the processor that owns the task in the active state. The object messages are usually much larger than the synchronization messages. If the implementation combined the data movement and synchronization messages, the large object messages would go to the processor that owned the task in the enabled state. If the task changed processors on the transition to the active state, the bandwidth taken up by the large object messages would be wasted. The current implementation sends object messages only to processors that will execute tasks declaring an access to the object.

There is a second, more subtle aspect of separating synchronization and object messages. Because copies of objects occupy memory, the implementation may be able to use memory more effectively by timing the arrival of objects close to the execution of the task that needs the objects. A task's synchronization constraints may be satisfied at temporally distant points. If the system combines synchronization and data messages, it may be forced to store objects whose messages satisfy early synchronization constraints. These objects may be resident for a long time as the task waits for the other synchronization constraints to be enabled. By separating synchronization and data movement messages, the implementation does not waste memory on object storage unless it knows that a task will access the object

in the near future.

3.4.6 The Consistency Problem

Any system that replicates mutable data must solve the consistency problem. The consistency problem arises when a processor writes a copy of a replicated object, generating a new version of the object. The implementation must then ensure that no processor subsequently reads one of the obsolete copies of the object.[1] Systems traditionally solve the consistency problem using either an invalidate or an update protocol. Systems using invalidate protocols keep track of all outstanding copies of an object. When a write occurs the system sends out messages that eliminate all of the obsolete copies. Update protocols work in a similar way, except that the writing processor generates messages that contain the new version of the object. These messages then overwrite the obsolete copies. At some point the writing processor must stall until it knows that all of the invalidates or updates have been performed. The exact stall point depends on the strength of the consistency protocol. For a more detailed treatment of consistency protocols see [45].

Update and invalidate protocols impair the performance of the system in several ways. First, there is the bandwidth cost of the update or invalidate messages. Second, there is the acknowledgment latency associated with stalling the writing processor until it knows that all of the invalidates and updates have been performed.

3.4.7 The Jade Consistency Mechanism

The Jade implementation uses an efficient consistency mechanism that eliminates some of the performance overhead of invalidate and update protocols. It first tags each copy of an object with a version number. The version number counts the number of times the program wrote the object before it generated that version of the object. For every task the implementation keeps track of which version of each object it must access, and the owner of that version (the processor that generated that version). The first owner of an object is the processor in whose memory the implementation initially allocated the object.

[1] More precisely, the system must ensure that no processor first observes that the writing processor has proceeded past the write, then reads the obsolete copy of the object.

Before the implementation executes or resumes a task, it checks the objects that the task will access. If there is no locally available copy of an object, or if the locally available copy has an obsolete version number, the implementation fetches the correct version from the owner. If the task will only read the object, the owner sends a copy to the reading processor. If the task will write the object, the owner moves the object. The writing processor then becomes the owner of the next version of the object.

The implementation generates no messages if the task will only read the object and the correct version is available locally. If the task will write the object and the correct version is available locally, the implementation writes the local copy and sends a message to the old owner telling it to deallocate its copy. Obviously, this message is not sent if the owner and executor are the same.

If a system replicates objects, it must be able to deallocate obsolete or unnecessary copies for good memory utilization. The Jade implementation may deallocate any copy of an object except the primary copy at the owner. The current implementation has a target amount of memory dedicated to objects, and deallocates object replicas (using a least-recently-accessed policy) when the amount of memory dedicated to objects rises above the target.

The implementation computes which version of an object each task should access using the object queue mechanism. If a task accesses an object its access declaration must go through the object queue. The object queue can therefore compute the version numbers and identities of owner processors by keeping track of both how many tasks declared they would write the object and which processor executed the last writing task. When the object queue gives a task the right to access an object, it tells the task both which version of the object it should access and the owner of that version.

There is a delicate implementation detail associated with fetching remote objects. It is possible for one processor to create an object whose initial owner is another processor. If a task on a third processor accesses the object, it will send a message to the initial owner requesting the object. It is possible for the message requesting the object to arrive at the owner before the message that tells the owner to allocate the object. In this case the owner knows nothing about the object. The implementation handles this race condition by maintaining a queue at each processor of requests for unknown objects. When the object

creation message arrives at the owner, the implementation checks this queue and forwards the object to any processors that need it.

3.4.8 Evaluation of Consistency Mechanism

The first advantage of this consistency mechanism is that it requires no update or invalidate messages. The implementation is therefore spared the overhead of processing these messages. The second advantage of this consistency mechanism is that it avoids the latency associated with ensuring that updates or invalidates have been performed at remote processors. The implementation never has to stall writing processors to ensure that the program executes correctly. The third advantage of this mechanism is that the implementation can move objects to the processors that access them without having to use a complicated protocol to locate the latest version of each object.

The Jade consistency mechanism can also take advantage of the potential performance advantages of other consistency protocols. Consider, for example, update protocols. Update protocols can hide read latency by eagerly replicating data at the processors that will access it. The Jade implementation can also hide read latency by eagerly sending objects to processors that will read the objects. But because the Jade implementation uses consistency mechanism based on version numbers, it has much more flexibility than systems which use strict update protocols. The Jade implementation can, for example, update only a subset of the processors which have obsolete copies of the object.

3.4.9 Adaptive Broadcast and Object Piggybacking

The Jade implementation takes advantage of the coherence mechanism's flexibility to implement an adaptive broadcast algorithm. The Jade implementation keeps track of which processors have read the latest version of each object. If all processors ever read the same version of an object, the implementation eagerly broadcasts all subsequent versions of the object to all processors. This optimization eliminates read latency and (on machines with hardware support for broadcast) drives down both the communication overhead at the owner processor and the total amount of bandwidth consumed to send the object to all of the processors.

The adaptive broadcast optimization may waste bandwidth and generate useless message handling overhead if processors do not actually access their copies of broadcasted objects. It would be possible to ameliorate this effect by eliminating the broadcast if all processors failed to access a given version of a broadcasted object. Each processor would keep track of whether it actually accessed its version of each object. If a processor received a new version of a broadcasted object and it did not access the previous version, it would send a message to the owner to turn off the broadcast for that object. The implementation would reinstate the broadcast only when all processors accessed the same version of the object. It is possible to generalize this approach to develop arbitrarily sophisticated schemes that attempt to discover and optimize a program's data transfer pattern.

The implementation can also exploit the data usage information to piggyback objects onto task messages. When the implementation sends a task to a remote processor for execution, it also sends (in the same message) any locally available objects that the task will access that are not replicated on the remote processor. Object piggybacking eliminates both remote access latency and the number of messages required to perform the computation.

3.4.10 Memory Management

The message-passing implementation has several data structures (object queues, task data structures and shared objects) that move between processors. The implementation must allocate memory for these data structures when the data structure arrives at a processor; the implementation must deallocate this memory for reuse when the data structure leaves the processor. The implementation uses the same memory management strategy for all data structures that move between processors.

When a data structure arrives at a processor it is stored in a message buffer. The implementation allocates memory for the data from the local memory management package, then copies the data out of the message buffer into the allocated memory. The address of the memory holding the data structure can therefore be different on different processors.

The implementation or the user program accesses each of these data structures using globally valid identifiers. The implementation keeps track of the correspondence between globally valid identifiers and the local addresses of the data structures using a table. There

is one such table for each processor and each kind of data structure. Each table maps the globally valid identifier of each locally resident data structure to the address of the memory holding that data structure. When the implementation needs to access a locally resident data structure, it uses the table to find the data structure.

An alternative implementation strategy for homogeneous systems would allocate each piece of data at the same address in each of the processors. The advantage of this strategy is that the implementation could use the address of each piece of data as its global identifier, and could eliminate the global to local translation. This strategy has several drawbacks. First, the implementation would have to partition the address space among processors and have each processor allocate memory from a different part of the address space. This would waste physical memory on systems with no support for sparse address spaces. Even for systems with such support, this allocation strategy could result in poor memory utilization caused by internal page fragmentation if the allocated objects were significantly smaller than the page size. Finally, the implementation would have to come up with a mechanism for determining if a given data structure was available locally.

For heterogeneous systems the strategy of allocating each data structure at the same virtual address would require the compilers on all the machines to allocate the same amount of space for each data type. This is totally impractical, because if one vendor introduced a new machine that required more space per data type, someone would have to change the memory layout strategy of the compilers on all of the other machines. This allocation strategy would also waste memory on machines that represented data more compactly than other machines.

3.4.11 Summary

Like the shared-memory implementation, the message-passing implementation extracts the concurrency and schedules the tasks onto processors for execution using a locality heuristic. The message-passing implementation must also implement the communication required to execute the program. The implementation exploits its control over the communication to implement several optimizations. These optimizations include the concurrent fetching of objects that tasks will access, the overlapping of computation and communication, and the

use of an adaptive broadcast protocol. The implementation also includes a general-purpose entity location algorithm based on a forwarding pointer scheme.

3.5 Common Aspects

Many of the issues associated with executing Jade programs are the same for both the shared-memory and the message-passing implementations. The two implementations often deal with these issues in the same way. In this section we discuss several algorithms that the two implementations share.

3.5.1 The Front End

Both front ends translate Jade code into C code containing calls to the Jade run-time library. For each `withonly` construct, the front ends replace the `withonly` construct with calls to Jade library routines, emit code that transfers the parameters from the parent task to the child task, and generate a separate function containing the task body. For each `with` construct the front ends have only to replace the `with` construct with calls to Jade library routines. The front ends also insert the dynamic access checks and convert the Jade variable declaration syntax to legal C declarations. To perform these actions they do a complete parse of the Jade code, including a complete type analysis.

There are two factors that complicate the construction of the message-passing front end. First, all communication uses message-passing constructs. The message-passing front end must therefore generate routines that interface with the message-passing system. Specifically, it generates routines to pack and unpack objects and task data from message buffers. The Jade run-time system calls these routines when it transfers data between machines. Second, Jade programs run in heterogeneous environments. This means that the implementation must represent all data and programming language constructs that cross machine boundaries in a machine-independent way. In particular, the front end must perform program transformations that support the implementation's use of globally valid identifiers for pointers to shared objects and shared functions. The routines that the front end generates must also perform the data format translations required to correctly transfer data

between machines with different data formats. For a detailed, example-driven discussion of what both front ends do, see Appendix A.

3.5.2 Access Checking

The implementation performs the dynamic access checks using an access declaration table. To prepare for a task's execution, the processor inserts each of the task's access declarations into a table indexed by the identifiers of the objects that the task declared it would access. There is one table per processor, and the task's declarations are inserted into the table associated with the processor that will execute the task. When the task executes, it performs the access checks by looking up declarations in the access declaration table and checking the accesses against the declarations. When programmers use the local pointer mechanism discussed in section 2.2.4, the implementation amortizes the lookup cost over many accesses via the local pointer.

3.5.3 Suppressing Excessive Task Creation

An important issue associated with managing dynamic concurrency is the suppression of excessive task creation [33, 96]. Jade programs may be able to generate an extremely large number of tasks. Because unexecuted tasks consume memory, excessive task creation stresses the machine's memory system, degrading performance. In extreme cases the machine may be unable to successfully execute the program. To successfully execute programs that can generate huge numbers of tasks, the Jade implementations contain mechanisms that suppress task creation when the number of outstanding tasks grows too large.

The shared-memory implementation maintains an approximation to the total number of outstanding tasks, and suppresses task creation when the total number of outstanding tasks rises above a maximum threshold. The message-passing implementation uses a similar mechanism locally on each processor. It counts the number of outstanding tasks stored on each processor, and suppresses task creation on that processor when the number rises above a maximum threshold.

When the implementation decides the program is generating too many tasks, it starts

suspending any task that creates a child task. The task does not resume execution until either the number of outstanding tasks drops below the mimimum threshold (there is a hysteresis built into the thresholds to prevent excessive suspension and resumption) or all of the task's child tasks finish executing.

This mechanism works well for programs with relatively flat task creation hierarchies. The idea is that there are two kinds of tasks: workers and spawners. Workers perform most of the useful computation, and usually execute to completion once started. The primary function of spawners is to create workers. When the program generates too many tasks, the algorithm attempts to identify and suspend spawners. It can then reduce the amount of memory dedicated to task data structures by executing workers to completion and deallocating their task data structures. When the amount of memory dedicated to task data structures drops below an acceptable amount, the implementation restarts the spawners to find more concurrency.

For programs with worker-spawner style task creation hierarchies the major drawback of this algorithm is that it has the potential to waste concurrency. A spawner may need to create many tasks in a concurrency-poor region before it can reach a concurrency-rich region. If the suspension threshold is set too low the implementation will suspend the spawner before it reaches the concurrency-rich region and the computation will unnecessarily suffer from a lack of concurrency. Given the small size of the task data structure, it is possible to set the suspension threshold fairly high.

The algorithm described above does not work as well for programs with deep task creation hierarchies. The problem is that the algorithm will always execute a task if it has no outstanding child tasks. Consider what happens if the program has a deep task creation hierarchy. The implementation will repeatedly choose an existing task and execute it until it creates a new child task. Because the program has a deep task creation hierarchy, almost every task will create a child task. Because the implementation will give each new task a new stack, the amount of space allocated to stacks will skyrocket. The amount of space allocated to task data structures will also go up, but this is not as much of a problem because task data structures are so much smaller than stacks.

The solution to this problem is to adjust the task execution algorithm when stack space gets tight. The implementation would run each new child task on the parent task's stack,

suspending the parent task until the child task finishes. It would be fairly straightforward to add such an algorithm to the current implementation of Jade.

Potentially excessive task creation is a problem for many parallel programming languages. The implementations of these languages therefore often attempt to suppress excessive task creation, using the standard mechanism of finding and suspending tasks that create child tasks. But the semantics of explicitly parallel languages means that a straightforward task suspension algorithm can cause the program to deadlock. In these languages it is possible for all of the existing tasks to require data produced by some suspended task. The proposed solution to this problem is to build in a deadlock elimination algorithm that unsuspends and executes tasks in the presence of deadlock searching for the task that will generate the piece of data required to get the computation going again. But the deadlock elimination algorithm may have to unsuspend and execute many tasks before it finds the task that unlocks the rest of the computation. Because the unsuspended tasks may create an unbounded number of child tasks before generating the required value, there are programs for which the algorithm fails to limit the excess task creation.

Jade's serial semantics, on the other hand, allow the Jade implementation to suspend any task without risking deadlock as long as that task is not the first (in the sequential execution order) outstanding task. In particular, it can suspend any task with outstanding child tasks and not risk deadlock. The Jade implementation can therefore use the straightforward task suspension algorithms to eliminate excess task creation and can impose a hard limit on the number of outstanding tasks without introducing the possibility of deadlock.

3.6 Basic Jade Overheads

In this section we present the basic time and space overhead of the Jade constructs.

3.6.1 `withonly` Time Overhead

To a first approximation, the time to create and execute a task depends on the number of objects the task declared that it would access and whether a task is executed locally on the same processor that created it or remotely on a different processor. The implementation

may take longer to execute a task if there is contention for the internal data structures or if the internal data structures are migrating between processors.

We measured the overhead of task creation using a benchmark program that creates and executes null tasks. By timing phases of these programs we can measure how long it takes to execute the different kinds of tasks. In reality the precise overhead in a given application can depend on complex interactions of different parts of the system. The figures presented below should therefore be taken as a rough indication of how large the overhead will usually be in Jade applications.

The benchmark program serially creates and serially executes many null tasks. We divide the total execution time by the number of tasks to calculate the per-task overhead. Appendix B contains the benchmark program used to measure the task creation and execution overhead. On DASH we measure three cases: 1) the tasks execute on the same processor as the creator, 2) the tasks execute on a different processor but within the same cluster as the creator, and 3) the tasks execute on a different cluster from the creator. The benchmark program generates each case by explicitly placing each tasks on the target processor. We plot the running times in microseconds for these three different cases in Figure 3.4. These curves graph the overhead as a function of the number of objects the task declared that it would access. The differences in the running times are caused by memory system effects.

For the iPSC/860 we measure two cases: 1) the tasks execute on the same processor as the creator and 2) the tasks execute on a remote processor. Figure 3.5 plots the running times in microseconds for these two cases. The remote overhead is substantially larger than the local overhead. We attribute this difference to the message composition and transfer overhead on the iPSC/860.

To facilitate a comparison of the relative efficiency of the DASH and iPSC/860 implementations, we also present the overhead in processor cycles. Figure 3.6 presents the overhead for DASH in processor cycles. This figure plots the execution times in Figure 3.4 multiplied by 33 MHz, the cycle time of the DASH processors. Figure 3.7 presents the overhead for the iPSC/860 in processor cycles. This figure plots the execution times in Figure 3.5 multiplied by 40 MHz, the cycle time of the iPSC/860 processors.

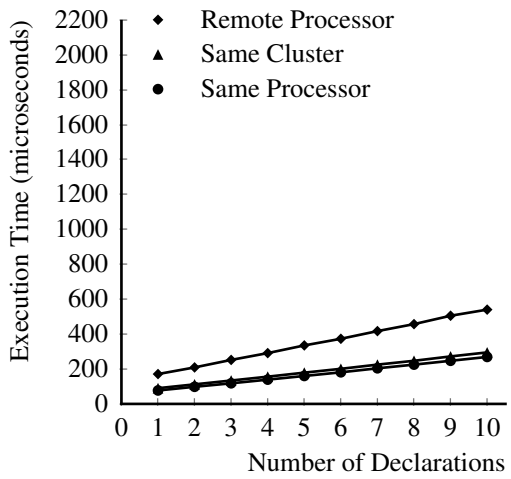

Figure 3.4: Task Overhead on DASH in Microseconds

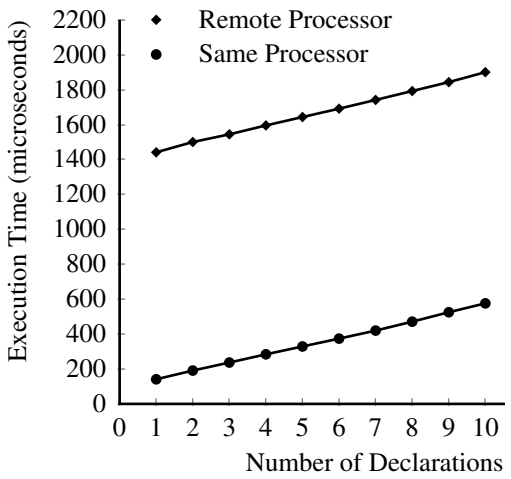

Figure 3.5: Task Overhead on the iPSC/860 in Microseconds

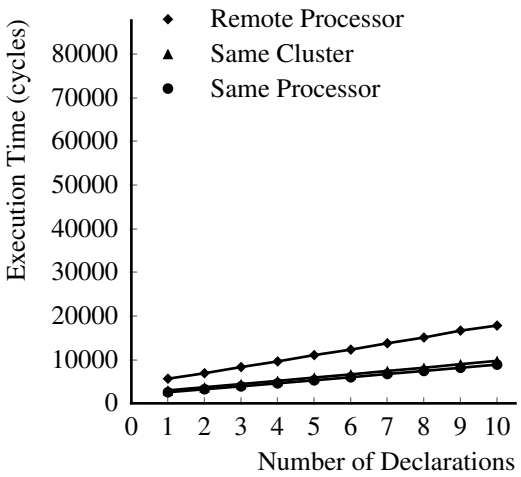

Figure 3.6: Task Overhead on DASH in Processor Cycles

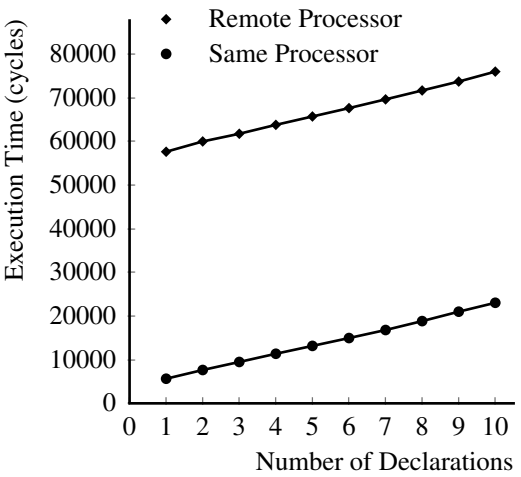

Figure 3.7: Task Overhead on the iPSC/860 in Processor Cycles

3.6.2 Speedup Benchmarks

The task overhead limits the grain size that Jade implementation can support. We created a benchmark program to measure how the speedup varies with the task size. The program has a sequence of phases; each phase serially creates and in parallel executes tasks of a given size. The sequence of phases varies the task size. The program devotes one processor to creating tasks and the other processors to executing tasks. Figure 3.8 presents the results of the program running with 32 processors on DASH; Figure 3.9 presents the results of the program running with 32 processors on the iPSC/860. Each figure plots the measured speedup as a function of the task size in microseconds. For calibration purposes we also present the speedups with the task size measured in processor cycles. The DASH speedup numbers are in Figure 3.10. This figure presents the same data as Figure 3.8 except that the task sizes have been multiplied by the 33 MHz cycle time of the DASH processors. Figure 3.11 presents the speedup numbers for the iPSC/860 with the task size measured in processor cycles. This figure presents the same data as Figure 3.9 except that the task sizes have been multiplied by the 40 MHz cycle time of the iPSC/860 processors.

In the benchmark program each task declares that it will access three objects. Each phase serially creates $31 * 256$ tasks that execute in parallel. The measured speedup is the task size times $31 * 256$ divided by the measured execution time to create and execute the tasks. Appendix B contains the benchmark program used to measure the speedups for various task sizes, and presents an analytic model that explains the shape of the speedup curves.

3.6.3 `with` Time Overhead

There is also time overhead associated with executing a `with` construct; to a first approximation the overhead is a function of the number of access specification statements that the `with` construct's `access specification` section executes. Figures 3.12 and 3.13 present the overhead in microseconds on DASH and the iPSC/860, respectively. Figures 3.14 and 3.15 present the corresponding graphs with the overhead measured in processor cycles. Appendix B contains the benchmark program used to measure this overhead.

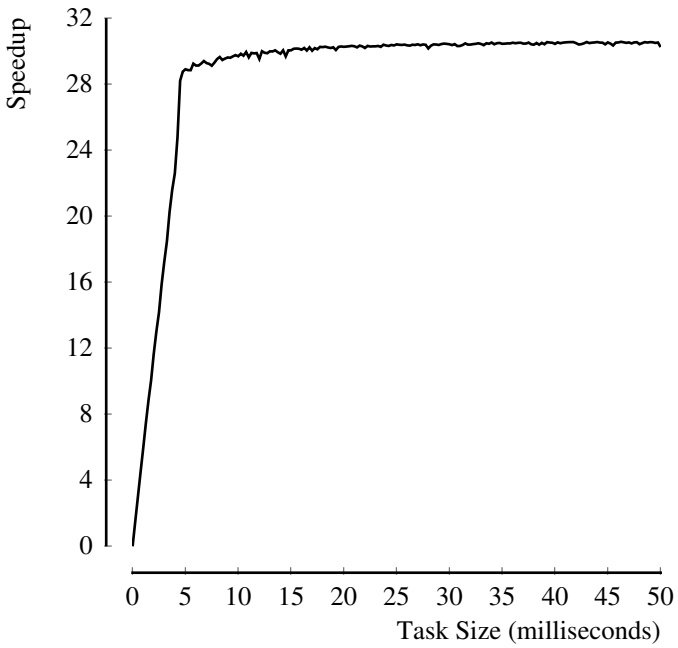

Figure 3.8: Speedup on DASH for 32 Processors (Task Size in Milliseconds)

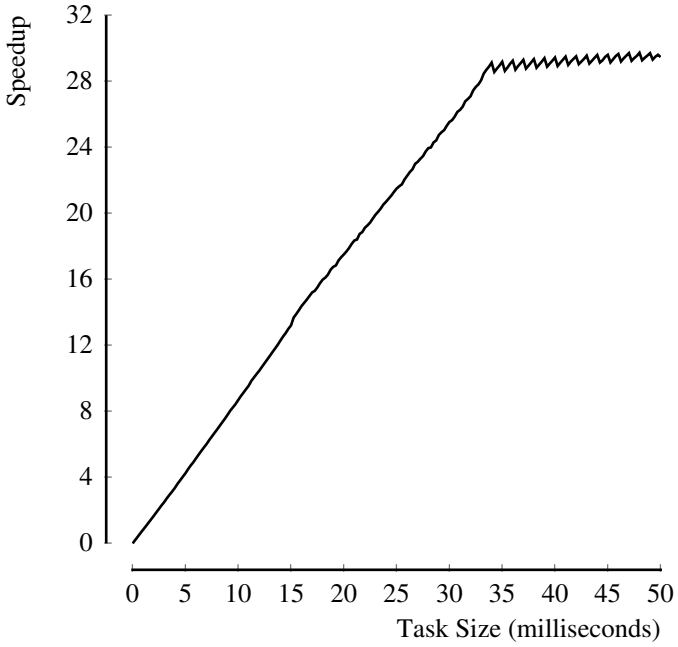

Figure 3.9: Speedup on the iPSC/860 for 32 Processors (Task Size in Milliseconds)

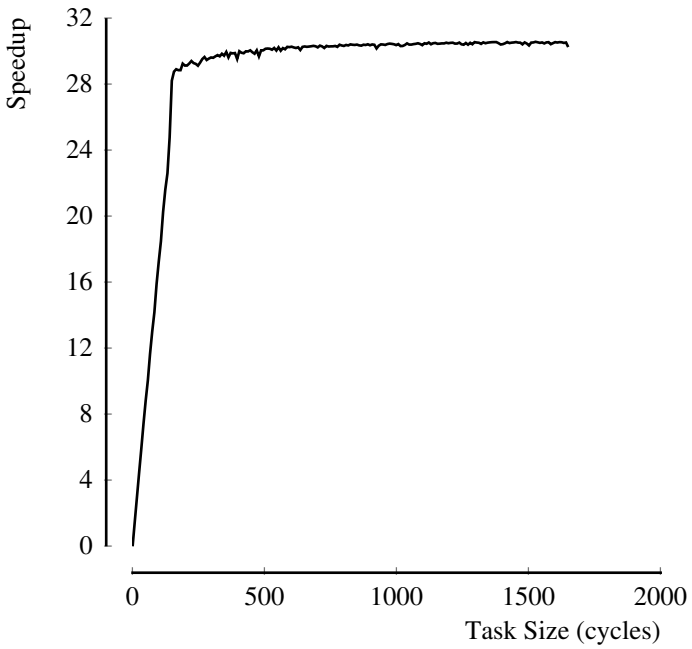

Figure 3.10: Speedup on DASH for 32 Processors (Task Size in Processor Cycles)

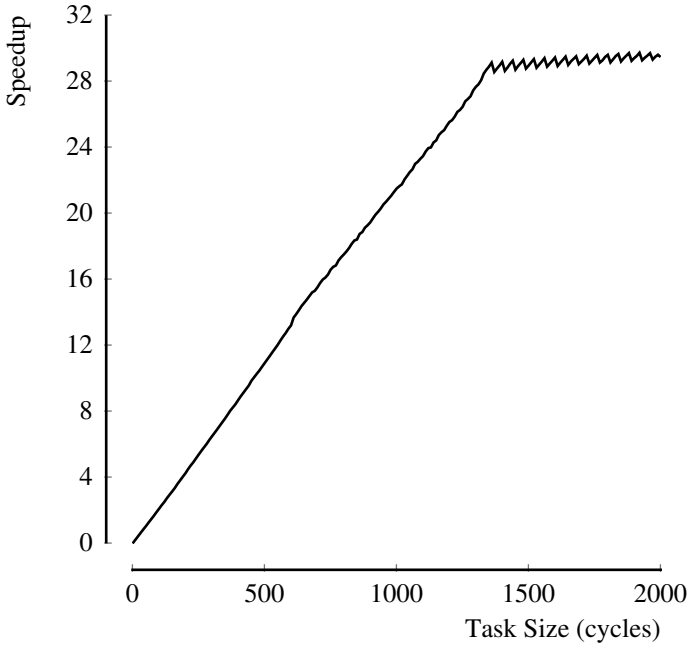

Figure 3.11: Speedup on the iPSC/860 for 32 Processors (Task Size in Processor Cycles)

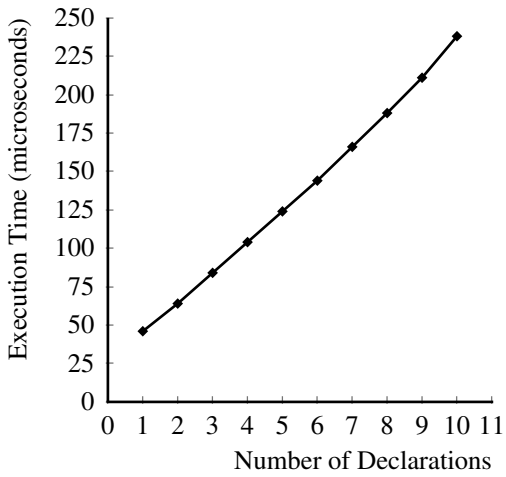

Figure 3.12: `with` Overhead on DASH in Microseconds

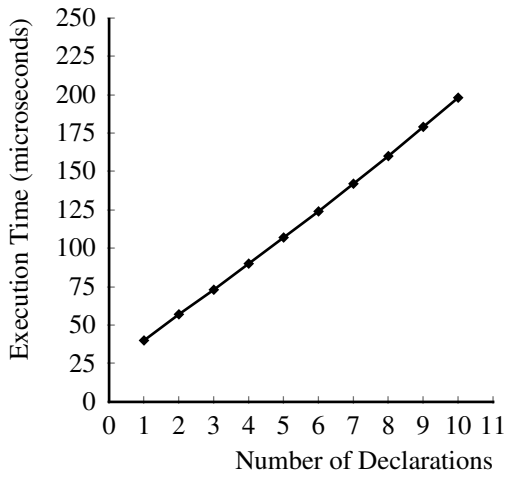

Figure 3.13: `with` Overhead on the iPSC/860 in Microseconds

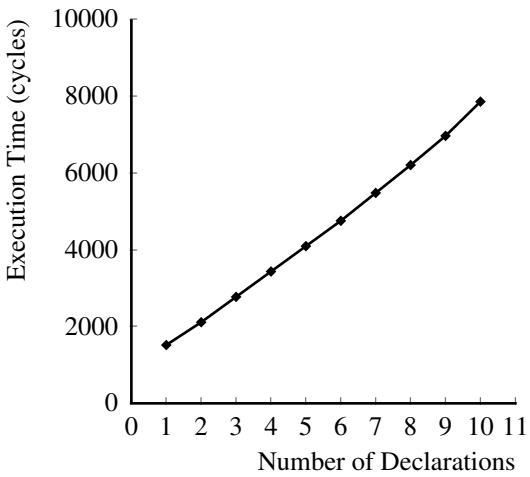

Figure 3.14: `with` Overhead on DASH in Processor Cycles

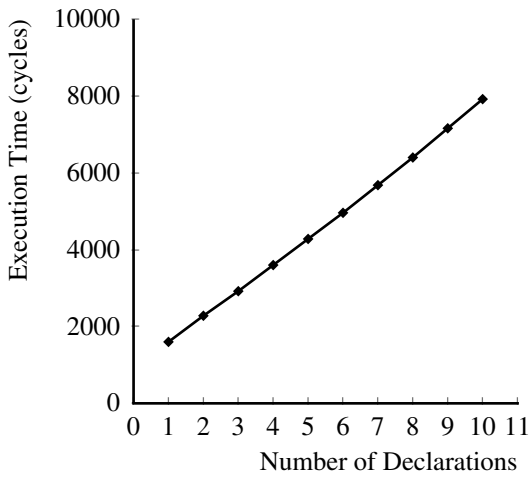

Figure 3.15: `with` Overhead on the iPSC/860 in Processor Cycles

3.6.4 Space Overheads

Both tasks and objects incur space overhead. In the shared-memory implementation, each task incurs an overhead of 552 bytes. Each object incurs an overhead of 84 bytes. Each access declaration incurs an overhead of 28 bytes associated with the task data structure. The task data structure contains space for 10 initial access declarations, so declarations do not start taking up additional space until a task declares that it will access more than 10 objects.

In the message-passing implementation each task incurs an overhead of 800 bytes. Each object incurs a total overhead of 400 bytes, with the object queue taking up 344 bytes and an object header taking up 56 bytes. There is one object header for each replica of the object. Each access declaration takes up 48 bytes associated with the task data structure and 28 bytes associated with the object queue data structure. Each task and object queue contains space for 10 initial declarations, so declarations do not start taking up space until a task declares that it will access more than 10 objects or until more than 10 tasks at a given point in time declare that they will access one object.

The task space overheads usually have very little impact on Jade computations. Because the implementation can suppress excess task creation (see Section 3.5.3), it can control the amount of memory devoted to task data structures. The object space overheads, on the other hand, can cause poor memory utilization for applications that create many small objects. In the worst case this poor memory utilization may artificially limit the problem size.

3.7 Summary

The Jade implementation bears the responsibility for bridging the gap between Jade's ostensibly sequential model of computation and the desired parallel execution. The implementation extracts the concurrency and maps the computation onto the parallel machine, using a heuristic designed to increase the locality of the computation. The message-passing implementation also exploits its control over the communication to apply several communication optimizations.

The Jade implementation encapsulates a set of algorithms that automatically manage

much of the process of exploiting concurrency. Encapsulating these algorithms allows Jade to deliver a high-level interface that simplifies the process of developing parallel software. Because the implementation also encapsulates the machine-specific code required to execute a program on the parallel machine at hand, Jade programs port without modification to a wide variety of computational environments.

Chapter 4

Applications Experience

As part of our evaluation of Jade, we obtained several complete scientific and engineering applications and parallelized them using Jade. We then executed these applications on several computational platforms. This experience gave us insight into the Jade programming process and provided an indication of what may happen when programmers use languages like Jade to parallelize complete applications.

We use our applications experience to evaluate Jade with respect to two properties: how well Jade supports the process of writing parallel programs and how well the resulting programs perform. Given the imprecise and often ambiguous nature of human-computer interaction, it is difficult to adequately evaluate the Jade programming process. While we present some numbers that measure application properties such as the number of Jade constructs in each application, our evaluation tends to be largely qualitative and usually focuses on phenomena visible in the final Jade version of each application.

Our performance evaluation, on the other hand, is based on hard performance data collected on parallel machines. While we often attempt to give the reader an understanding of the qualitative issues underlying the performance, our discussion of these issues is usually backed by numbers collected during the execution of the application.

As part of our evaluation we also consider the impact of the different Jade optimizations. We typically evaluate these optimizations by running the same application both with and without the optimization. The collected performance data often enable us to precisely characterize how each optimization affects the behavior of the parallel execution.

When possible we also present performance numbers for existing explicitly parallel versions of the applications. Because these versions typically control the computation at a low level for efficiency, they yield insight into the level of performance obtainable on the platform under consideration.

The rest of the chapter is structured as follows. We first describe how we acquired and developed each of the Jade applications. We then present the experimental methodology and collected performance data. The majority of the chapter is devoted to a deeper analysis of the programming and performance implications of using Jade for each application.

4.1 The Application Set

Choosing a set of benchmark applications to evaluate a language design and implementation is a tricky business. On the one hand it is important to choose applications that are within the target application domain. The benchmark set is therefore inevitably filtered as applications viewed as outside the application domain are rejected. But filtering the applications too stringently, on the other hand, can yield a sterile evaluation. How well the language deals with unforeseen application characteristics will be an important factor in its overall success, and a balanced benchmark set should include some programs that stretch the capabilities of the language and its implementation.

Several factors influenced our choice of applications. One major factor was availability. Existing benchmark sets were one source of applications, and three of our applications originally came from the SPLASH benchmark suite [128]. We also acquired three applications directly from the research groups that initially developed them. Another factor was the engineering effort involved in manipulating the application. The engineering effort required to deal with large programs restricted us to fairly small applications, but we did invest a substantial amount of time and effort to be sure that we at least developed complete applications. A final factor was our assessment of how well the application fit the target Jade application domain. When possible we performed an initial assessment by analyzing the characteristics of an existing parallel version before deciding to develop a Jade version. For two of our applications, however, the Jade parallelization was the first (and so far the only) parallelization to exist.

We next describe each of our applications. For each application we briefly summarize the computation that it performs, then describe how we acquired and parallelized the application.

- **Water** A program that evaluates forces and potentials in a system of water molecules in the liquid state. Water is derived from the Perfect Club benchmark MDG [21] and performs the same computation. The SPLASH benchmark suite [128] also contains a version of Water. I developed the Jade version of Water starting with serial C version that had been translated from the original Fortran.

- **String** A program that computes a velocity model of the geology between two oil wells. I originally obtained a serial version of String [58] written in a combination of C and Fortran from Jerry Harris (a professor in the Stanford Geophysics Department), Mark van Schaack (a graduate student in the Stanford Geophysics Department) and Caroline Lambert (a programmer in the Stanford Geophysics Department) and, with the help of Brian Schmidt (a graduate student in the Stanford Computer Science Department), translated the Fortran parts of the application to C. I then developed the Jade version starting from that serial C version. The Jade version is the only parallel version of this application that exists.

- **Search** A program that simulates the interaction of electron beams with solids [25, 26]. Jun Ye (a graduate student in the Stanford Electrical Engineering Department) developed an initial serial version of Search in C. I developed the initial Jade version, then handed the program off to Ray Browning (a researcher in the Stanford Electrical Engineering Department), who substantially modified the program to generate the final version of Search. The Jade version is the only parallel version of this application that exists.

- **Volume Rendering** A program that renders a three-dimensional volume data set for graphical display. Jason Nieh (a graduate student in the Stanford Computer Science Department) developed the Jade version of this application starting from an existing parallel version that he had developed in C using the ANL macro package. He used compile-time flags to disable the ANL constructs, then inserted the Jade constructs

required to parallelize the application. The SPLASH application suite contains the ANL version of the Volume Rendering application.

- **Panel Cholesky** A program that factors a sparse positive-definite matrix. For Panel Cholesky I obtained a parallel version written in C by Ed Rothberg (a graduate student in the Stanford Computer Science Department) using the ANL macro package. I reconstructed a serial version by removing the ANL constructs and developed the Jade version starting from this serial C version.

- **Ocean** A program that simulates the role of eddy and boundary currents in influencing large-scale ocean movements. Jennifer Anderson (a graduate student in the Stanford Computer Science Department) parallelized the Ocean application. She started with the parallel Fortran program from the SPLASH benchmark suite written using the ANL macro package and reconstructed a serial version by removing ANL constructs. She then translated the serial Fortran into C and parallelized the serial C version using Jade.

In any application-driven study the application set has a critical impact on the experimental results and on the assessment of the language and its implementation. We next describe several applications that did not make it into the application set. This description provides additional insight into the application selection process and helps to define the scope of the presented experimental results.

There are two Jade applications on which we did not run complete performance tests. The first application was a parallel version of the Unix *make* utility. Dan Scales (a graduate student in the Stanford Computer Science Department) developed the Jade version starting from a serial version written in C. For parallel compilation the limiting factor on the performance is the disk bandwidth. Dan Scales also developed a Jade version of the Barnes-Hut application from the SPLASH benchmark suite as a preliminary step towards developing a version in SAM [124]. This program did not scale well in the current Jade implementation because of because of a synchronization bottleneck associated with the object queue for a key shared object. All of the tasks read this object, and the current implementation serializes all of the resulting object queue operations. As described in Section 3.2.2, this artificial serialization generated a bottleneck.

During the course of the project we considered and rejected several other applications. For a variety of reasons the rest of the applications in the SPLASH benchmark suite fall well outside the intended Jade application domain. The parallel tasks in LocusRoute and MP3D asynchronously read and write potentially overlapping sections of a central data structure. PTHOR exploits concurrency at a granularity much finer than the targeted Jade granularity.

We also decided not to implement several potential applications from the Stanford scientific and engineering community. The most notable was an application from the Stanford Operations Research Department for solving stochastic linear programs. We rejected this application because of the engineering effort involved – we would have had to translate more than 100,000 lines of code from Fortran into C. The other application was from the Stanford Mechanical Engineering Department. This application simulated chaotic fluid-flow problems. We rejected this application because we believed its communication to computation ratio was too high for the current generation of multiprocessors.

4.2 Application Characteristics

We next present some basic application characteristics and use these characteristics to discuss several aspects of the Jade programming process. Table 4.1 presents some of their static characteristics. A comparison of the number of lines of code in the serial version (when available) with the number of lines of code in the parallel version indicates that using Jade usually involves a modest increase in the number of lines of code in the application. The number of Jade constructs required to parallelize the application (and especially the number of `withonly` constructs) is usually quite small.

As these numbers suggest, using Jade did not usually impose an onerous programming burden. For all of our applications the key to a successful parallelization was determining an appropriate structure for the shared objects. Such a structure was always fairly obvious given a high-level understanding of the basic source of exploited concurrency. Once the structure was in place inserting the Jade constructs required to specify the task granularity and data usage information was a straightforward process with no complications.

For all of the applications implementing the correct object structure for the Jade version

Application	Lines of Code Serial Version	Lines of Code Jade Version	`withonly` Sites	`with` Sites	Object Creation Sites
Water	1219	1471	2	20	7
String	2587	2941	3	37	19
Search	-	716	1	9	3
Volume Rendering	-	5419	2	8	15
Panel Cholesky	2047	2484	2	15	18
Ocean	1274	3262	27	28	20

Table 4.1: Static Application Characteristics

involved some modification of the original data structures. Except for Ocean, these modifications were performed without disturbing the vast majority of the code and generated minimal programming overhead. For Ocean the programmer had to decompose many of the arrays in the program. This decomposition forced the programmer to change the indexing algorithm for the arrays over large parts of the program. These changes imposed substantial programming overhead and dramatically increased the size of the Jade program relative to the original serial program.

We found that several aspects of the Jade language design supported the development of these parallel applications. Programmers came to rely on the fact that the Jade implementation verified the access specification information. They typically developed a working serial implementation with the data structured appropriately for the Jade version, then inserted the Jade constructs. Programmers became quite cavalier about this process, typically making changes quickly and relying on the implementation to catch any bugs in the parallelization. This stands in stark contrast to the situation with explicitly parallel languages. The possibility of nondeterministic execution masking errors usually makes programmers paranoid about changing a working program, and the parallelization proceeds much more slowly.

We next discuss the performance of the resulting parallel computations. We collected extensive performance measurements for each application on a shared-memory platform (the Stanford DASH machine) and on a message-passing platform (the Intel iPSC/860). Appendix **??** describes the basic hardware parameters of the two machines. Table 4.2 presents some basic performance numbers for the iPSC/860 runs, while Table 4.3 presents

Application	Sequential Execution Time (seconds)	Speedup on 32 Processors	Mean Task Size on 32 Processors (seconds)
Water	2406.72	26.29	4.75
String	19629.42	28.93	74.30
Search	1284.07	27.90	42.61
Panel Cholesky	28.53	0.74	.0020
Ocean	60.99	1.16	.0033

Table 4.2: Dynamic Application Characteristics for the iPSC/860

Application	Sequential Execution Time (seconds)	Speedup on 32 Processors	Mean Task Size on 32 Processors (seconds)
Water	3285.90	27.50	6.53
String	19314.80	27.36	81.63
Search	1652.91	31.16	51.52
Volume Rendering	32.44	17.16	0.63
Panel Cholesky	28.91	5.02	0.0024
Ocean	100.03	9.34	0.0047

Table 4.3: Dynamic Application Characteristics for DASH

the corresponding results for the DASH runs. For reasons described below in Section 4.5, Volume Rendering did not run on the iPSC/860.

To a first approximation there are two kinds of applications: coarse-grain applications with mean task sizes ranging from several seconds to well over a minute and finer-grain applications with a mean task size measured in milliseconds. The coarse-grain applications scale almost linearly to 32 processors while the finer-grain applications do not scale as well. The scaling problem is especially severe on the iPSC/860. On the iPSC/860 the relatively large message-passing overhead makes it impossible to implement the basic Jade primitives as efficiently as on DASH, which supports much finer-grain communication.

4.3 Performance Measurements

We next describe how we collected the performance measurements. Both implementations are heavily instrumented. The instrumentation can be turned on and off under the control

of several compile-time flags. It is possible to automatically collect the following kinds of data.

- **Timers** For each processor the implementation records the amount of time spent in different segments of the program. There are timers for the total time, the time spent executing application code, the time spent in the Jade implementation and the idle time. The implementation measures the time by reading a clock when the program enters and exits the different segments, adding the difference to a running sum upon exit.

  On DASH the implementation uses the 32 bit 60 nanosecond counter on the DASH performance monitor chip [82] to implement a high-resolution 64 bit clock. The 60 nanosecond counter is the lower half of this clock; the software increments the 32 bit upper half every time the counter wraps. The timers may slightly overestimate the amount of time spent in the different segments because they include time spent in the operating system.

  There is a discrepancy on DASH between times measured using the 60 nanosecond counter and times measured using the standard Unix *gettimeofday* system call. Using a test program and comparing the measured running times of several Jade applications using the two different clocks we determined that multiplying the 60 nanosecond counter times by a correction factor of 1.0417 brings the times back into agreement. All times reported in this thesis that were derived from 60 nanosecond counter measurements have been multiplied by this correction factor.

  On the iPSC/860 the implementation uses the `mclock` routine in the iPSC/860 programming interface as the clock. The timers may overestimate the amount of time spent in each section because they fail to take into account the time spent in the operating system handling messages. This message handling takes place asynchronously with respect to the Jade program, and parts of this time may be incorrectly attributed to the parts of the execution described above.

- **Event Counts** For each processor the implementation counts a variety of events that dynamically occur. These include task creation, execution and suspension events,

access declaration events, object creation events and object replication or movement events.

- **Message Data** In message-passing environments the implementation keeps a running sum, for each processor, of the number and size of each message that the processor sent and received. The implementation keeps separate counts for each message type. The primary use of this segregation is to separate messages carrying shared objects from control messages.

- **Event Logs** For each processor the implementation generates a log that records the time when specific events occurred during the application's execution. Each log is stored in memory and written out at the end of the measured phase.

  Both the shared-memory and message-passing implementations record events that deal with the execution of tasks. The implementation records an event when a task starts to execute, creates a child task, executes a `with` construct, suspends, resumes and/or terminates. Each suspension event records the cause of the suspension. It is therefore possible, for example, to distinguish the suspension events recorded when a task suspends at a `with` construct and when a task suspends at the end of a `block` construct.

  The message-passing implementation also records events that deal with the communication of shared objects between processors. The sending processor records an event when a task invokes a message-passing primitive to send or broadcast an object to another processor. It also records an event when the primitive returns. Each such event records why the object was transferred. It is therefore possible, for example, to distinguish messages sent in response to object request messages and messages containing piggybacked objects (see Section 3.4.9 for a description of how the Jade implementation piggybacks objects onto task messages). The receiving processor records an event when it requests an object and when it receives an object.

  We use the event logs to analyze the behavior of several applications. We extract and process the data in a way that is appropriate for each application, introducing the presentation formats as they are used.

4.3.1 Instrumentation Levels

We run each application at several instrumentation levels. The lower instrumentation levels perturb the computation less than the higher instrumentation levels, but also provide less information about the execution. We collect data at the following instrumentation levels.

- **Minimum Instrumentation (mi)** Measure total time and idle time only.
- **Full Instrumentation (fi)** Collect all data except event logs.
- **Full Log Instrumentation (fl)** Collect all data.

4.3.2 Optimization Levels

We next consider communication optimizations. The message-passing implementation always applies the optimization that replicates data for concurrent read access. Except for the adaptive broadcast optimization, it also always applies the other communication optimizations discussed in Chapter 3, although turning them off would affect the performance of none of the applications. We evaluate the impact of the locality and adaptive broadcast optimizations by running versions of the applications with these optimizations turned on and off. The shared-memory and message-passing versions provide the following three different locality optimization levels.

- **Explicit Task Placement (at)** In Ocean and Panel Cholesky the programmer can improve the locality of the computation by explicitly controlling the placement of tasks on processors. We describe the specific placement strategy when we describe each application.
- **Locality Heuristic (lo)** The shared-memory implementation uses the locality heuristic described in Section 3.3.4; the message-passing implementation uses the locality heuristic described in Section 3.4.2.
- **No Locality (nl)** The implementation distributes enabled tasks to idle processors in a first-come, first-served manner. There is no attempt to generate an assignment of tasks to processors that has good locality properties. The shared-memory implementation

uses a single shared task queue to perform the distribution of tasks to processors. In the message-passing implementation the main processor (the processor that starts the execution of the program) maintains a queue of idle processors and distributes enabled tasks to processors in the order in which they appear in the queue.

The standard locality optimization level is to use the locality heuristic; we run the other levels for comparison purposes.

The message-passing implementation provides two levels of adaptive broadcast optimizations.

- **Adaptive Broadcast (ab)** The implementation uses the adaptive broadcast algorithm described in Section 3.4.9.
- **No Adaptive Broadcast (nb)** The implementation does not use the adaptive broadcast algorithm.

The standard locality optimization level is to use the adaptive broadcast optimization; we run applications without it for comparison purposes.

4.3.3 Version Names

We name each version of an application by its instrumentation and optimization levels. Shared memory version names consist of two components separated by periods. The first component identifies the instrumentation level; the second identifies the locality optimization level. For example, the version name mi.lo specifies minimum instrumentation and the use of the locality heuristic. Message passing version names consist of three components separated by periods. The first component identifies the instrumentation level, the second identifies the locality optimization level and the third identifies the adaptive broadcast optimization level. For example, the version name mi.lo.ab specifies minimum instrumentation, the use of the locality heuristic and the adaptive broadcast optimization.

4.3.4 Collected Data

We collect data for several versions of each Jade application running on 1, 2, 4, 8, 12, 16, 20, 24, 28 and 32 processors. In all timing runs the machine was idle except for the Jade

application. We also run the Jade version with all of the Jade constructs stripped out. The resulting serial program contains the data structure modifications required to parallelize the program, but contains no overhead from the Jade implementation. We call this version the stripped version. We also run the original serial version if it is available. On DASH we measure the running time for the original serial version using the Unix *gettimeofday* system call. For all other versions on DASH we use the synthesized 64 bit 60 nanosecond counter described in Section 4.3. On the iPSC/860 we use the `mclock` routine for all versions.

For all applications we provide a table containing the running times for different versions of the applications. We then present data that explains the performance of application under the different instrumentation and optimization levels. The goal is to give the reader a clear, often qualitative understanding of the dynamic behavior of the computation and to isolate any properties of the Jade implementation that affect the performance. There is no standard format for this presentation – for each application we select the data and presentation format that we believe best explains the observed behavior.

The performance measurements characterize the behavior of Jade applications under the current implementation. In some cases an enhanced implementation would eliminate certain performance problems. When this is the case we describe the enhancement and the expected effect on the performance.

4.4 The Water, String and Search Applications

We combine the discussion of the Water, String and Search applications because their computations share the same basic concurrency structure. We first discuss each specific application, then describe the general form of concurrency that they all exhibit. We then discuss how to exploit this kind of concurrency in Jade.

4.4.1 Water

The Water computation consists of a sequence of iterations. Each iteration consists of several $O(n)$ phases (where n is the number of molecules) and two $O(n^2)$ phases. One $O(n^2)$ phase computes the intermolecular forces acting on each molecule; the other computes the

potential energy of the system of water molecules.

In the force-calculation phase the application computes the total force acting on each molecule as the sum of the individual forces between that molecule and all other molecules in the system. The result is stored in a force array indexed by molecule number. This phase also computes the total energy, storing the result in a single scalar variable. The potential-energy phase is similar, except that it sums the total potential energy into a single scalar potential-energy variable. Both phases depend on the positions and momentum of the molecules, which are computed in the $O(n)$ phases.

It is easy to see that all of the interactions can be computed concurrently. The only issue is the accumulation of the results into the final data structure. Because the basic accumulation operation is addition, the accumulations commute, associate and have an identity.

4.4.2 String

String [58] uses seismic travel-time inversion to construct a two-dimensional discrete velocity model of the geological medium between two oil wells. Each element of the velocity model records how fast sound waves travel through the corresponding part of the medium. The seismic data are collected by firing non-destructive seismic sources in one well and recording the seismic waves digitally as they arrive at the other well. The travel times of the waves can be measured from the resulting seismic traces. The application uses the travel-time data to iteratively compute the velocity model.

The computationally intensive phase of the application traces rays from one well to the other. The velocity model determines both the path and the simulated travel time of each ray. The computation records the difference between the simulated and measured travel times and backprojects the difference linearly along the path of the ray. At the end of the phase the computation uses the backprojected differences to construct an improved velocity model. The process continues for a specified number of iterations.

It is easy to see that all of the rays can be traced in parallel. The issue is the accumulation of the backprojected differences. The serial computation stores the velocity model and the backprojected differences in two-dimensional arrays. Each element of the difference array

stores the running sum of the backprojected differences for the corresponding element of the velocity model. As in the Water application, the accumulations commute, associate and have an identity.

4.4.3 Search

Search [25, 26] is a program from the Stanford Electrical Engineering department. It simulates the interaction of several electron beams at different energy levels with a variety of solids. It uses a Monte-Carlo technique to simulate the elastic scattering of each electron from the electron beam into the solid. The result of this simulation is used to measure how closely an empirical equation for electron scattering matches a full quantum-mechanical expansion of the wave equation stored in tables.

For each pair of solids and electron-beam energies the computation counts the number of electrons that emerge back out of the solid and (implicitly) the number that remain trapped inside. It is easy to see that the electron paths can be simulated concurrently. The only issue is managing the array that stores the counts of emerged electrons. Each element of this array stores the total count for one pair of solids and electron-beam energies. Because the result of each electron computation is a potential increment to an element of the array, the accumulations commute, associate and have an identity.

4.4.4 The Concurrency Stucture

Water and String alternate serial and parallel phases. Each phase reads data produced in the previous phase and generates data read in the next phase. Search consists of an initialization phase, a parallel phase and a termination phase that collects and prints out the results.

The parallel phases consist of many small pieces of computation. For convenience we call each piece of computation a *tasklet*. In Water each tasklet computes the interaction of two molecules, in String each tasklet traces one ray through the velocity model and in Search each tasklet simulates the path of one electron. Each tasklet generates a contribution; the contributions are combined to produce the final result of the parallel phase. In Water the final result is either the total force acting on each molecule and the total-energy or the potential-energy contribution of the interactions, in String the final result is the mean

velocity difference at each element of the velocity model and in Search the final result is the number of emerged electrons for each pair of solids and electron-beam energies.

The first issue we consider is the management of the data structure used to store the contributions. The most straightforward management strategy is to accumulate the contributions into the final result data structure as they are generated. Each accumulation would then execute with exclusive access to the result data structure. The problem is that using a single result data structure could cause a sequential bottleneck.

For these applications an effective way to eliminate this bottleneck is to create multiple copies of the result data structure and distribute the accumulations across the copies. Each copy is initialized to the identity for the accumulation operation. At the end of the parallel phase the computation combines all of the copies to generate the final result. There is a trade-off between the amount of exposed concurrency and the amount of space required for copies of the result data structure. Each of the applications takes a granularity parameter that determines the number of copies. This granularity parameter is typically set to the number of processors executing the parallel application.

The next issue is the aggregation of tasklets into Jade tasks. One strategy exposes all of the concurrency by generating one task per tasklet. This strategy maximizes both the amount of task management overhead and the effectiveness of the implementation's dynamic load balancing algorithm. Another strategy generates one task per copy of the result data structure, assigning tasklets to tasks in a round robin fashion. This strategy generates the least task management overhead but leaves the execution vulnerable to poor load balance. Intermediate strategies assign different numbers of tasklets to tasks.

Water, String and Search all use the one task per copy of the result data structure strategy. For these applications the round robin mapping of tasklets to tasks generates an acceptable load balance.

4.4.5 Parallel Reductions

All three applications combine intermediate copies of the result data structure to generate the final result. One way to perform this computation is to use a parallel reduction. This section shows how to code a parallel reduction in Jade. We introduce parallel reductions by

showing how to compute the sum of a set of `doubles`. In the eventual Jade version each `double` will be a shared object, so we represent the set as an array of pointers to `doubles`. The parallel reduction is computed in place, so the shared objects hold intermediate values during the computation. The final value is stored into the shared object that the first array element points to. Figure 4.1 shows the operations performed in a parallel reduction of a set of four `doubles`. Each operation adds two elements then stores the result back into the first element. Each array element participates in a sequence of such operations.

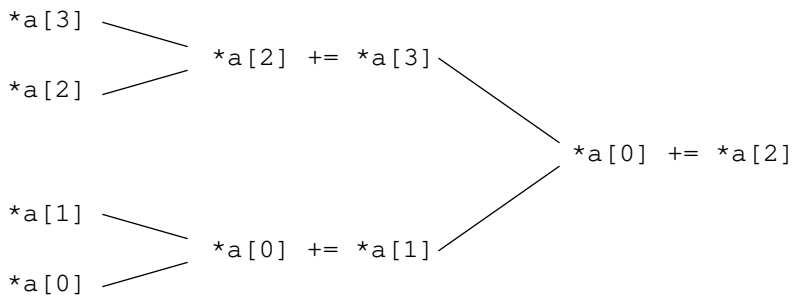

Figure 4.1: Parallel Reduction

To express the reduction in Jade, we first develop a serial procedure to perform the operations in the reduction. Figure 4.2 contains this serial procedure. It takes two parameters: `a`, which is the array pointing to the set of elements to reduce, and `g`, which tells how many elements there are in the set. The procedure processes the elements, starting with the element that the last array entry points to and ending with the first. The procedure brings each element to its final value before proceeding on to the next. The `for` loop in line `3` handles each element `*a[e]` in turn. The computation from lines `4` to `8` generates the offsets `o` of the elements to add to element `*a[e]`, with line `6` performing the addition. Every offset is twice as far away as the last offset, so the next offset `o` (computed in line `7` of Figure 4.2) is always the next power of two.

The condition in the `while` loop in line `5` exits when the current element `*a[e]` has reached its final value. The first clause in the condition (`(e + o) < g`) exits when the offset of the next element to add to `*a[e]` is outside the range of the array. All succeeding offsets will be outside the range and `*a[e]` has reached its final value.

The second clause in the condition (`!(e & o)`) exits when the bit in the binary representation of `e` that corresponds to the current value of `o` is `1`. To provide insight into

```
 1: reduce(double shared * shared *a, int g) {
 2:   int e, o;
 3:   for (e = g-1; e >= 0; e--) {
 4:     o = 1;
 5:     while (((e + o) < g) && !(e & o)) {
 6:       *a[e] += *a[e+o];
 7:       o = o*2;
 8:     }
 9:   }
10: }
```

Figure 4.2: Serial Code for Parallel Reduction

why the element has reached its final value when the bit is `1`, we label the branches of the reduction tree with either a `0` or a `1` as demonstrated in Figure 4.3. If you read off the labels along any path from the root to a leaf element, you get the binary representation of the index of the element at the leaf. At every internal node of the tree the reduction adds the two elements that contain the sums of the left and right subtrees to generate the total sum of the subtree rooted at that internal node. It must then write the total sum back into one of the elements for use in the next level of the reduction. The reduction always uses the element that corresponds to the branch labeled `0`. The code can therefore determine when the current element will not be used in the next level of the reduction (and has therefore reached its final value) by detecting the first `1` in its binary representation.

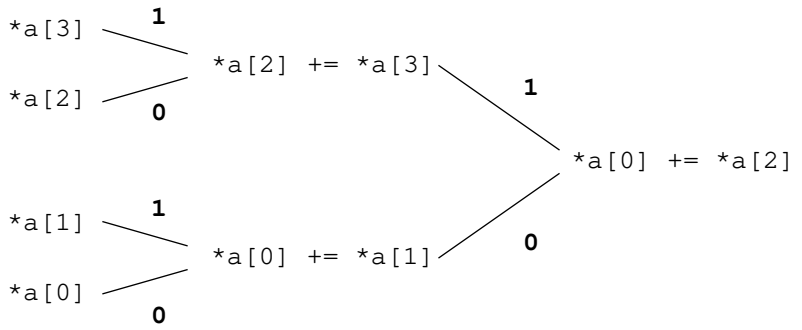

Figure 4.3: Labeled Parallel Reduction

To clarify the presentation of later algorithms we define `combine(e,g)` to be the

sequence of values that `e+o` takes on in line `5` of Figure 4.2. We will use `combine(e,g)` as an abstraction to hide the details of computing which elements to add with each other in the reduction. Figure 4.4, which performs the same computation as Figure 4.2, illustrates how we will use this notation.

```
1: reduce(double shared * shared *a, int g) {
2:   int e, o;
3:   for (e = g-1; e >= 0; e--) {
4:     for (j in combine(e, g)) {
5:       *a[e] += *a[j];
6:     }
7:   }
8: }
```

Figure 4.4: Serial Code for Parallel Reduction using `combine(e,g)`

To parallelize the reduction in Jade, we generate a task for each element. Each task adds the correct sequence of other elements to its element. It declares an immediate read and write access on its element and a deferred read access on all the elements it will add to its element. As the task reads the elements it uses a `with` construct to change the deferred declaration to an immediate declaration. Figure 4.5 contains the Jade code for the reduction.

4.4.6 Expressing the Concurrency in Jade

We next show how to express the three applications in Jade. We emphasize the similarity between the three applications by presenting a general concurrency schema that covers the specific concurrency pattern in each application. This schema uses a general abstraction (called the accumulator abstraction) that encapsulates the replication of the result data structures, the accumulation of the tasklets' contributions and the parallel reduction. While it would be possible (and even beneficial) to code the applications using this general abstraction, each application currently uses less general data structures.

The accumulator abstraction exports several operations. These operations include

```
 1: reduce(double shared * shared *a, int g) {
 2:   int e, j;
 3:   for (e = g-1; e >= 0; e--) {
 4:     withonly {
 5:       rd(a);
 6:       rd_wr(a[e]);
 7:       for (j in combine(e, g)) df_rd(a[j]);
 8:     } do (a, e, g) {
 9:       for (j in combine(e, g)) {
10:         with { rd(a[j]; } cont;
11:         *a[e] += *a[j];
12:       }
13:     }
14:   }
15: }
```

Figure 4.5: Jade Code for Parallel Reduction

`create_accumulator`, which creates an accumulator, the `init_accumulator` operation, which is invoked at the beginning of every parallel phase to initialize the accumulator, the `update_accumulator` operation, which each task invokes to combine a newly generated contribution into the accumulator, and `result_accumulator`, which each task invokes at the end of its computation to generate the final result. The `declare_update_accumulator` operation encapsulates the access specification statements required to declare how the `update_accumulator` and `result_accumulator` operations access data.

Figure 4.6 contains the Jade psuedo code that illustrates how to use the accumulator. The code first initializes the accumulator, then creates the tasks that perform the actual computation. The granularity parameter `g` controls the granularity of the computation by determining how many tasks are created. As described in Section 4.4.4, the tasklets are distributed to tasks in a round-robin fashion.

Each task's `access specification` section first invokes the operation that declares how the accumulator will access data, then declares how it will access the global shared objects and shared objects generated in the previous serial phase to compute its contributions.

```
accumulator shared *a;

init_accumulator(a);
for (i = g - 1; i >= 0; i--) {
  withonly {
    declare_update_accumulator(a, i);
    /*
      The programmer inserts here the application-specific
      access specification statements required to declare
      how the task reads shared objects. The task typically
      reads two kinds of shared objects:
        1) Global objects written once in an initialization
           phase and read by the rest of the program.
        2) Objects computed in the preceding serial phase.
    */
  } do (i, g) {
    for (every g'th tasklet t starting with tasklet i) {
      compute contribution c from tasklet t;
      update_accumulator(a, c, i);
    }
    result_accumulator(a, i);
  }
}
```

Figure 4.6: Parallel Schema for Water, String and Search

In Water each task declares that it will access the molecule positions and momentum data computed in the previous section and an object that holds some global variables. In String each task declares that it will access the new velocity model and an object that holds some data initialized at the start of the computation and read thereafter. In Search each task declares that it will access several objects that hold input parameters.

When the task runs, it computes its set of tasklets, updating the accumulator with each contribution as it is generated. When the task finishes with its set of tasklets it invokes the `result_accumulator` operation to perform its part of the parallel reduction.

We next describe the implementation of the accumulator abstraction. Figure 4.7 contains the data structure definition for the accumulator object. This object points to the copies of

```
typedef struct {
  /*
     result_copy points to the copies of the result data
     structure. It must be big enough to hold pointers to
     all of the copies.
  */
  void shared* result_copy[MAX_NUM_PROC];
  /*
    These point to the functions that implement the basic
    operations of the result data structure.
  */
  void shared (*copy)();
  void shared (*zero)();
  void shared (*add)();
  void shared (*update)();
  /*
     The number of actually allocated copies of the result
     data structure is g.
  */
  int g;
} accumulator;
```

Figure 4.7: Accumulator Data Structure

the result data structure and stores the granularity parameter `g`. The granularity parameter determines the number of copies of the result data structure. The accumulator data structure also stores the function pointers that implement the operations of the specific result data structure. The client of the abstraction provides these operations when the accumulator is created. The accumulator requires operations to create a copy of the result data structure, copy one result data structure into another result data structure, zero a result data structure, add one result data structure to another and store the result back into the first data structure, and an operation to update a result data structure with a tasklet's contribution.

Actually performing the modifications to integrate the replicated data structures into the application involved relatively little programming overhead. The vast majority of the modifications took place on the boundaries of the code that performs the computation of each

```
create_accumulator(g, create, copy, zero, add, update) {
  a = create_object(accumulator);
  with { rd_wr(a); } cont;
  a->g = g;
  a->copy = copy;
  a->zero = zero;
  a->add  = add;
  a->update = update;
  for (i = 0; i < g; i++) {
    a->result_copy[i] = (*create)();
  }
  with { df_wr(a); } cont;
  return a;
}

init_accumulator(a) {
  zero = a->zero;
  for (i = 0; i < a->g; i++) {
    copy = a->result_copy[i];
    withonly { rd_wr(copy); } do (copy, zero) {
      (*zero)(copy);
    }
  }
}
```

Figure 4.8: Accumulator Creation and Initialization

tasklet. The core parts of the program remained unchanged. The only other modifications involved the aggregation of global variables into a single global data structure using an approach similar to that illustrated in Figure 2.16. The globals are typically written once in an initialization phase of the computation and only read in the rest of the computation. The aggregation simplifies the access declarations and has minimal programming impact.

```
declare_update_accumulator(a, i) {
  rd(a);
  rd_wr(a->result_copy[i]);
  for (all j in combine(i, g)) {
    df_rd(a->result_copy[j]);
  }
}

update_accumulator(a, c, i) {
   (*(a->update))(a->result_copy[i], c);
}

result_accumulator(a, i) {
  for (all j in combine(i, a->g)) {
    with { rd(a->result_copy[i]); } cont;
    (*(a->add))(a->result_copy[j], a->result_copy[i]);
  }
}
```

Figure 4.9: Accumulator Operations

4.4.7 Water on the iPSC/860

Table 4.4 contains the execution times for several versions of Water running on the iPSC/860. As described in Section 4.3.3, the name of each version specifies the instrumentation and optimization levels. For example, the mi.lo.ab version specifies the minimum level of instrumentation (mi), the locality heuristic (lo) and the adaptive broadcast optimization (ab). The serial version is the original serial program; the stripped version is the Jade version with all Jade constructs (and therefore all dynamic Jade overhead) stripped out by the Jade front end. This data is collected using the timers described in Section 4.3. Each number in the table comes from a single run of the program.

For many of the applications we graphically display the performance data in the form of speedup curves similar to those presented in Figure 4.10. Each speedup curve plots, as a function of the number of processors executing the computation, the running time of the stripped computation divided by the running time of the parallel computation. We chose

the stripped version as the baseline because it is usually has the fastest serial execution time of any of the versions and because for some applications an original serial program does not exist. Ideally, the speedup for the computation running on p processors is p. Any deviation from this ideal indicates sources of inefficiency in the parallelization.

Figure 4.10 contains the speedup curves for four versions of Water. The rows of this figure vary the locality optimization level and keep the adaptive broadcast level constant. The top row contains the speedup curves with the adaptive broadcast optimization turned on. The columns of this figure keep the locality optimization level constant and vary the adaptive broadcast optimization level. The leftmost column contains the speedup curves with the locality optimization turned on.

The data set for these timing runs consists of 1728 molecules distributed randomly in a rectangular volume. It executes 8 iterations, with two parallel phases per iteration. These performance numbers omit an initial I/O and compution phase. In practice the computation would run for many iterations and the amortized cost of the initial phase would be negligible.

	1	2	4	8	16	24	32
serial	2482.91	-	-	-	-	-	-
stripped	2406.72	-	-	-	-	-	-
mi.lo.ab	2452.24	1231.70	622.14	318.00	166.43	118.96	91.86
fl.lo.ab	2435.16	1219.71	617.28	315.69	165.64	118.09	91.53
mi.nl.ab	2357.74	1215.61	615.15	314.93	166.01	118.91	92.39
fl.nl.ab	2454.78	1231.91	623.34	318.34	167.77	119.72	93.11
mi.lo.nb	2418.35	1214.55	615.34	319.26	177.74	139.22	121.87
fl.lo.nb	2459.87	1233.98	625.27	323.84	180.15	140.59	122.74
mi.nl.nb	2454.65	1230.76	623.87	323.82	181.74	143.65	126.90
fl.nl.nb	2442.38	1225.26	621.29	322.00	179.54	140.30	122.64

Table 4.4: Execution Times for Water on the iPSC/860 (seconds)

The performance numbers reveal several properties of this application. The adaptive broadcast optimization has by far the largest impact on the performance. With this optimization the application scales almost linearly to 32 processors. Without it the performance starts to tail off after 16 processors. The locality heuristic, on the other hand, has a negligible performance impact. We explore the reasons for these performance effects using data from the event logs. We next describe how we analyze and present this data.

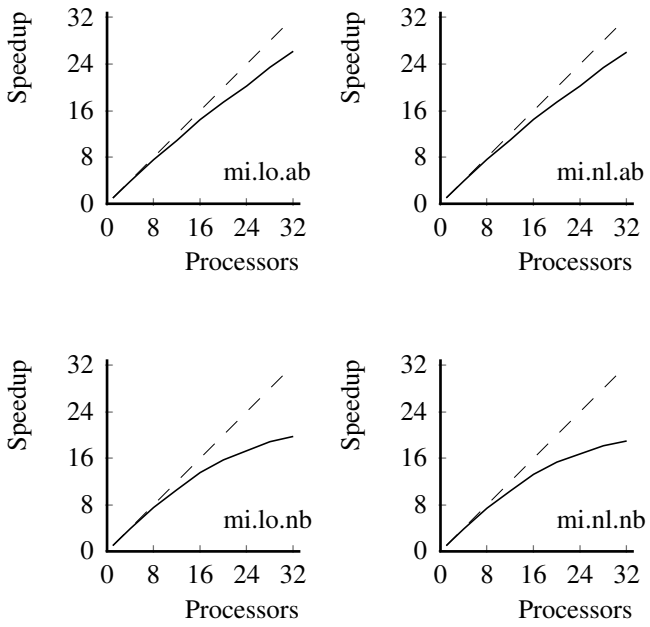

Figure 4.10: Speedups for Water on the iPSC/860

4.4.7.1 Activity Traces

For Water, as for several other applications, we present the event log information using a kind of time series graph called an activity trace. Each activity trace displays, as a function of time, the number of processors that are performing a given activity. For example, Figure 4.11 contains an activity trace that graphically displays the number of processors executing application code in a 32 processor run of the fl.lo.ab version of Water. This particular activity trace highlights the structure of the Water computation, clearly displaying the division into an interleaved sequence of large parallel phases and small serial phases.

We next discuss the format of the activity trace graph. The X dimension corresponds to increasing time, and the numbers on the X axis give the time range and scale for the trace. The Y dimension corresponds to the number of processors. The Y axis contains a bar whose height is the range of the graph. The bottom of the bar is located at the level corresponding to 0 processors. The top of the bar indicates the upper limit of the graph. The number in the upper left hand corner gives the number of processors to which the upper

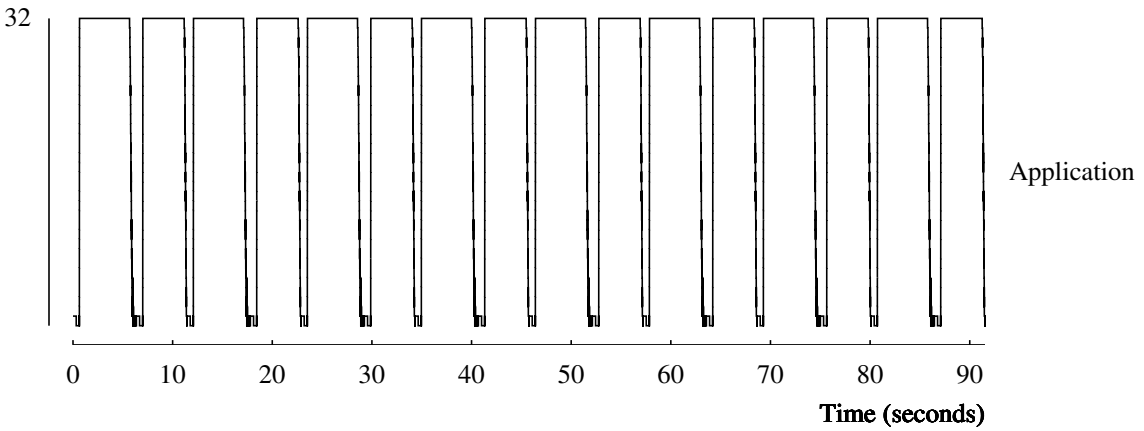

Figure 4.11: Activity Trace for Water (fl.lo.ab) on the iPSC/860

limit corresponds. So, in Figure 4.11 the Y dimension ranges from 0 to 32 processors, with the bar identifying the upper and lower limits. The label on the right hand side identifies the specific activity that the graph displays. The label for the graph in Figure 4.11 is Application, which indicates that the graph displays the number of processors executing application code. For the iPSC/860 runs we generate traces for the following activities.

- **Sending** A processor is counted as sending data at a given point in time if the number of start send events in the log up to that point is larger than the number of stop send events.

- **Broadcasting** A processor is counted as broadcasting data at a given point in time if the number of start broadcast events in the log up to that point is larger than the number of stop broadcast events. In our application set only the main processor ever broadcasts an object.

- **Creating** A processor is counted as creating child tasks at a given point in time if it meets one of two conditions. The first condition is that the point in time is between two child task creation events in the log and there are no events between the two child task creation events.[1] The second condition is that the point in time is between

[1] In general this condition overestimates the amount of time spent creating child tasks. If task creates a

a child task creation event and a task execution event, a task termination event, a task resumption event, a task suspension event, a stop broadcasting data event or a stop sending-piggybacked-object event, there are no events between the two events in question and the two events in question are within 0.001 seconds of each other. These conditions convert a sequence of child task creation events into child task creation intervals. In our application the only tasks that create child objects always execute on the main processor.

- **Application** For a processor to be counted as executing application code at a given point in time the number of task execution and resumption events in the event log up to that point in time must exceed the number of task termination and suspension events and the processor must not meet any of the conditions for the Sending, Broadcasting and Creating activities at that point in time.

- **Waiting** A processor is counted as waiting for data at a given point in time if the number of request object events in the log up to that point in time is larger than the number of receive object events and the processor does not meet the conditions for any other activity.

To avoid short spurious changes in the recorded activity, the implementation coalesces adjacent intervals during which a processor engages in the same activity if the difference between the two intervals is less than 0.001 seconds. So, for example, if a processor finishes sending one message and immediately starts sending another, it is counted as sending a message for the short time between the stop event from the first message send and the start event from the second message send. The exceptions are that adjacent Waiting intervals are not coalesced and adjacent Application intervals are not coalesced if they come from the execution of the same task.

We often stack several activity traces on top of each other to show correlations between the different activities. Figure 4.12 shows such a set of stacked activity traces from a 32 processor Water execution on the iPSC/860. We eliminate the X axes between adjacent

child task, executes some application code and then creates another child task, the condition will incorrectly count the time spent in application code as task creation time. None of our applications, however, executes any significant amount of application code between adjacent task creation events. The condition is therefore accurate for our set of applications.

activity traces to reduce visual clutter. We also reduce the size of the Y axis for an activity if the maximum number of processors engaging in that activity at any one time is substantially less than the number of processors executing the computation. The Y axis for the Application trace always corresponds to the number of processors executing the computation.

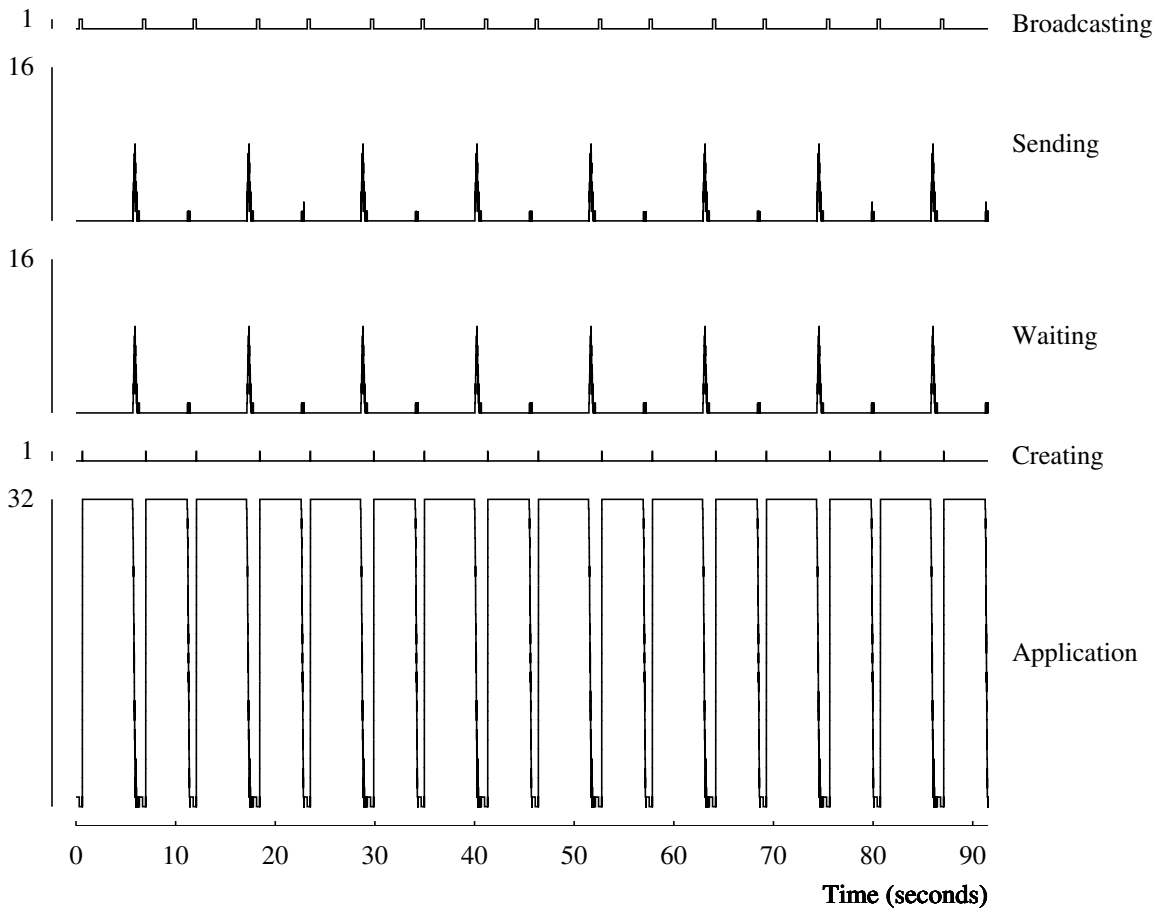

Figure 4.12: Stacked Activity Traces for Water (fl.lo.ab) on the iPSC/860

4.4.7.2 Water Behavior at the Standard Optimization Level

As described in Section 4.3.2, the standard optimization level uses the locality heuristic and adaptive broadcast optimizations. We describe the behavior of Water at this optimization

level with the aid of the activity traces in Figure 4.12. Each of the eight iterations consists of several phases. The iteration starts on the main processor with the main task executing a serial phase. The rest of the processors are idle. The serial phases show up in the activity trace as the small sections of the Application trace when only one processor is executing.

When the main processor finishes its serial phase it broadcasts the new molecule positions and momentum data to all of the other processors. Each broadcast generates a spike in the Broadcasting activity trace. The application then creates all of the tasks to perform the parallel phase; this task creation generates a spike in the Creating activity trace.

In the resulting parallel phase each processor computes its contributions to the total force acting on each molecule. The Application trace in Figure 4.12 shows that all 32 processors are busy executing application code for the majority of the parallel phase. The parallel phases in turn dominate the execution time, and most of the processors are busy executing application code for most of the computation.

As the tasks finish their force computations they participate in a parallel reduction of the replicated force array and total-energy scalar as described in Section 4.4.5. The parallel reductions for the force-calculation phases show up in the activity traces as the large spikes in the Sending and Waiting traces. At the end of the force-calculation reduction, the main processor executes the serial phase for the second half of the iteration. The second half has the same basic structure as the first half, but the parallel phase computes the potential energy, not the intermolecular forces. The spikes at the end of the potential-energy phases are smaller than the spikes at the end of the force-calculation phases because the messages in the force-calculation reduction carry the relatively large force arrays while the messages in the potential-energy reduction only carry the small potential-energy scalars.

4.4.7.3 Effect of the Adaptive Broadcast Optimization

As described in Section 3.4.9, the Jade implementation keeps track of which processors access each version of each object. When all processors access the same version of each object, the implementation switches to a broadcast protocol for subsequent versions. For Water this optimization results in the broadcast, after each serial phase, of the objects containing the new molecule positions and momentum data.

Without the adaptive broadcast optimization the implementation distributes the molecule

positions and momentum data to the processors using the object piggybacking optimization. During the task creation phase the objects containing the new molecule positions and momentum data are piggybacked onto the messages that distribute the tasks from the main processor to the other processors (see Section 3.4.9 for a description of the object piggybacking optimization). It therefore takes longer to send each task message from the main processor to the processor that will execute the task, which delays the creation of the next task. Figure 4.13, which displays part of the activity traces from a run without the adaptive broadcast optimization, shows how these delays stagger the initiation and completion of the parallel phases. The stagger in turn lengthens the time required to perform the parallel phases and degrades the performance. For comparison purposes we show, in Figure 4.14, the corresponding activity traces from the fl.lo.ab version, which uses the adaptive broadcast optimization. These traces show how the adaptive broadcast optimization eliminates the stagger in the initiation and completion of the parallel phases.

4.4.7.4 Effect of the Locality Heuristic

We next discuss the impact of the locality heuristic on the parallel execution. As described in Section 3.4.2, the locality heuristic chooses a locality object for each task and attempts to execute each task on the processor that owns the latest version of the locality object. This processor is called the target processor for the task. In the Water application the locality object for each task is the replica of the force array or potential-energy scalar into which the task will accumulate its contributions.

Our first measure of the effectiveness of the locality heuristic is how well it succeeds in placing each task on its target processor. We display this data using graphs, like those in Figure 4.15, that plot task locality percentages. Each graph plots, for a given version of the application, the percentage of tasks that execute on their target processors. More precisely, each point on the curve plots, as a function of the number of processors executing the computation, the number of tasks that executed on their target processor divided by the total number of executed tasks times 100. We generate this data using the event counts described in Section 4.3.

Figure 4.15 displays the task locality percentages for the fl.lo.ab and fl.nl.ab versions of Water. These graphs show that, for the Water computation, the locality heuristic is

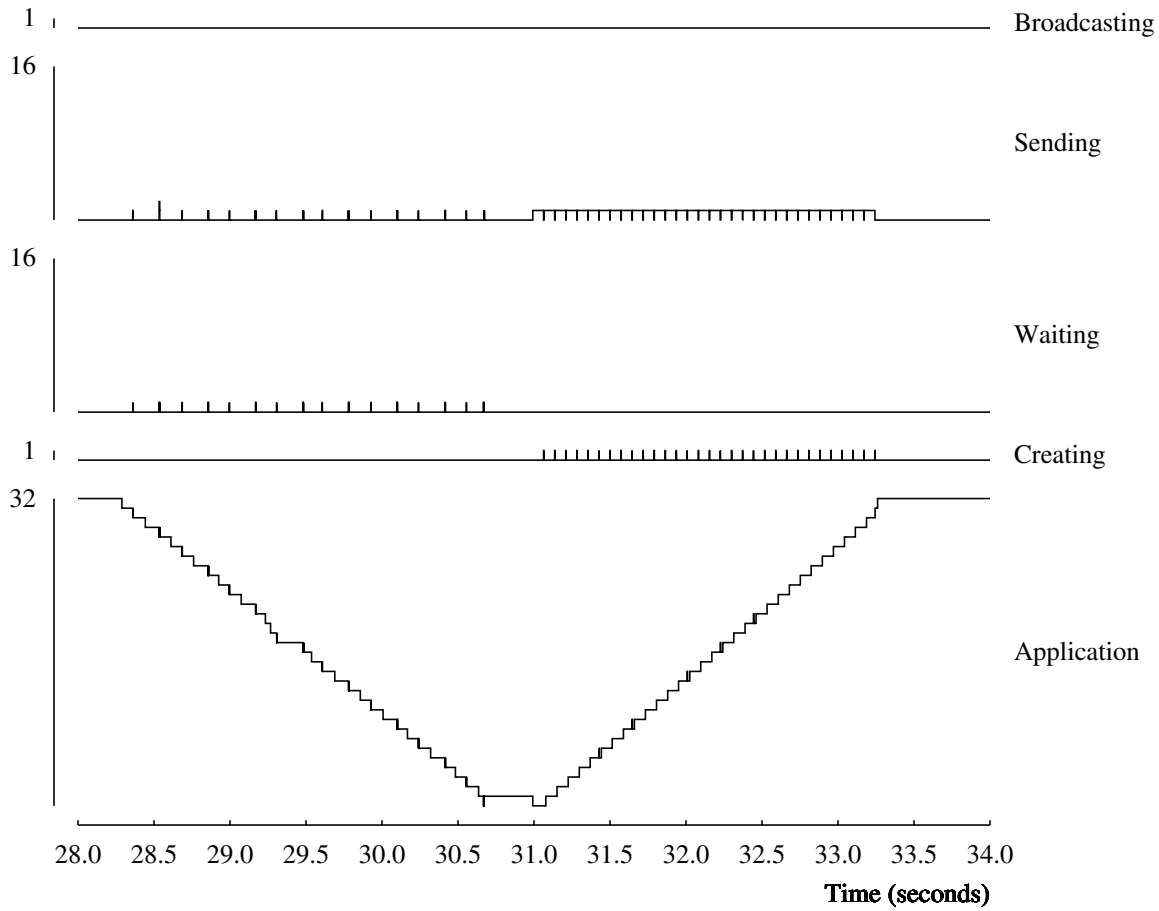

Figure 4.13: 32 Processor Activity Traces for Water (fl.lo.nb) on the iPSC/860

very effective at executing tasks on the target processor – every task executes on its target processor. Without the locality heuristic the percentage of tasks executed on their target processors drops quickly as the number of processors increases.

We continue our evaluation of the locality heuristic by considering its effect on the communication. If a task executes on a processor that does not contain the latest version of the replica of the force array, total-energy scalar or or potential-energy scalar that it will access, it fetches the latest version from the owner processor before it executes. Without the locality heuristic many tasks must fetch replicas from remote processors before executing. These fetches generate a flurry of communication at the beginning of each parallel phase

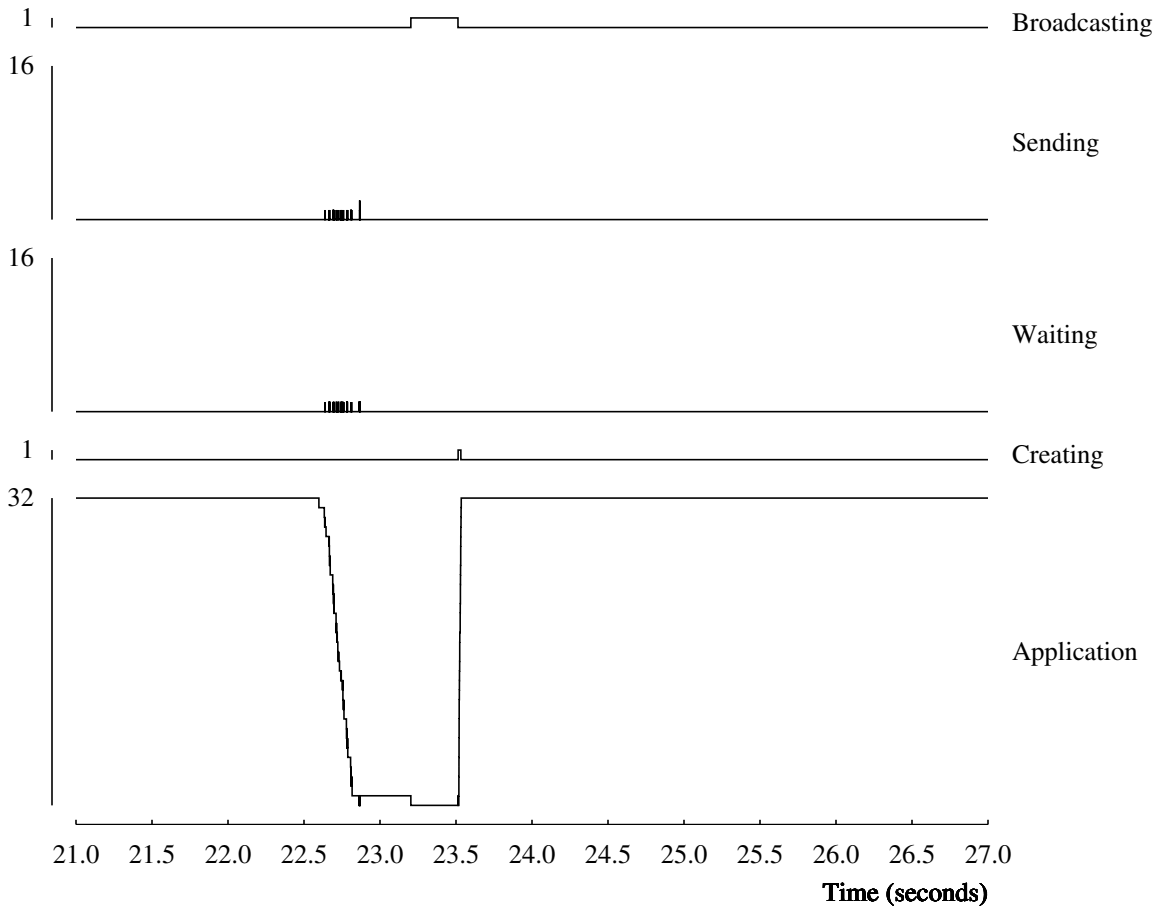

Figure 4.14: 32 Processor Activity Traces for Water (fl.lo.ab) on the iPSC/860

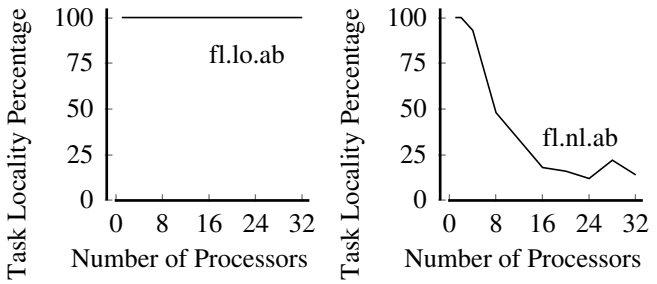

Figure 4.15: Task Locality Percentages for Water on the iPSC/860

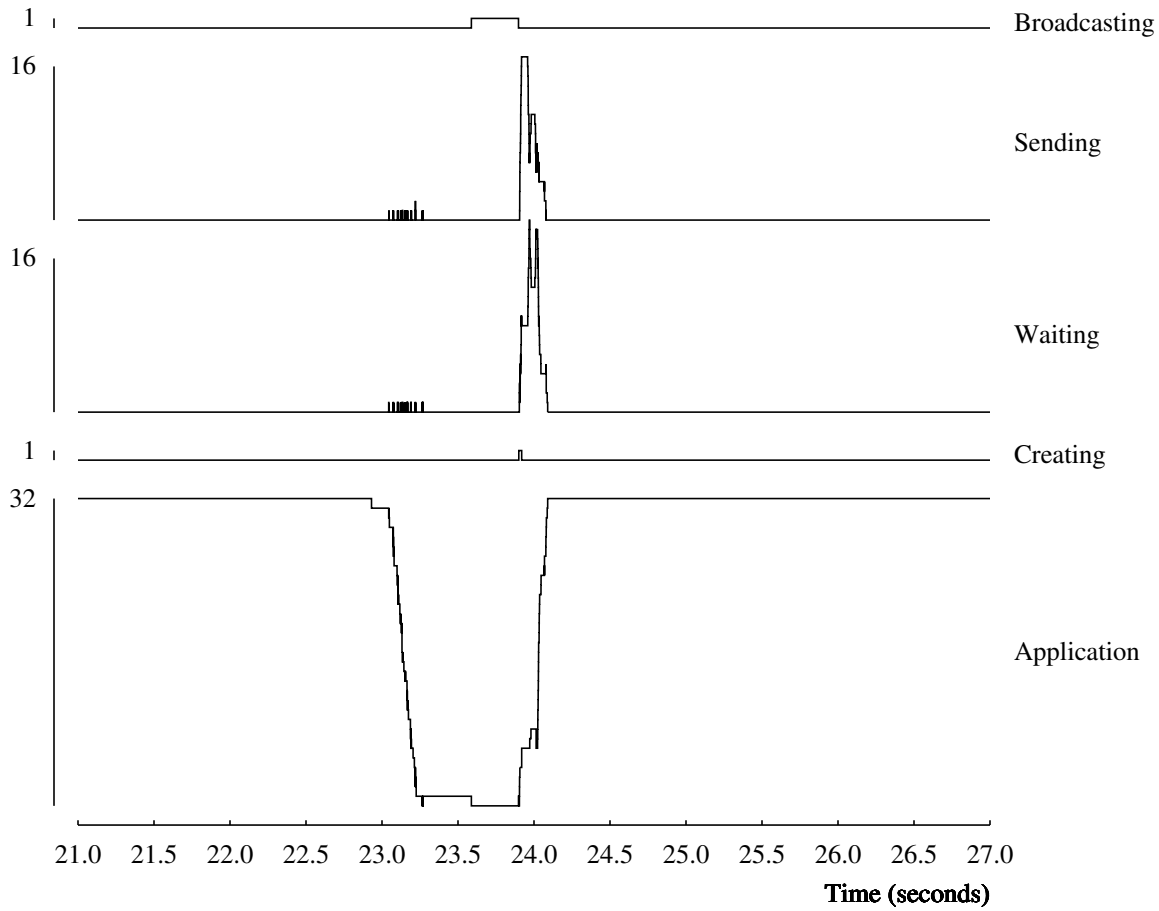

Figure 4.16: 32 Processor Activity Traces for Water (fl.nl.ab) on the iPSC/860

as the replicas migrate to the accessing processors. The activity traces in Figure 4.16 highlight the effect by displaying the interval between a parallel potential-energy phase and parallel force-calculation phase. The Sending and Waiting traces clearly show the burst of communication at the start of the force-calculation phase. The size of this communication burst depends on the size of the transferred replicas. The burst is much smaller at the start of the potential-energy phases because the transferred potential-energy scalars are much smaller than the force arrays. Figure 4.14, which presents the corresponding activity traces for a version that uses the locality heuristic, shows how the use of the locality heuristic eliminates the communication burst at the the beginning of each parallel phase.

Each task that accesses a remote replica of the force array or potential-energy scalar must delay its execution until the replica arrives. The maximum size of the delay is the time required to request, transfer and receive the replica. The performance numbers show that this delay has a negligible impact on the performance. Part of the reason for this lack of impact is that the extra communication tends not to occur along the critical path. For this computation the critical path goes through the task and reduction operations executed on the main processor. Even without the locality heuristic the main processor always owns the locality object for the parallel task that it executes.[2] So, this task is never delayed by the need to fetch a remote piece of data. The only other way the extra communication can extend the critical path is by delaying a reduction operation on the main processor. The fact that the task on the main processor is the last to be created and start executing minimizes the impact of delaying the other tasks.

4.4.8 Water on DASH

Table 4.5 contains the execution times for several versions of Water running on DASH. As for the iPSC/860 performance numbers, the name of each version specifies the instrumentation and optimization levels. For example, the mi.lo version specifies the minimum level of instrumentation and the use of the locality heuristic. The ANL version is the explicitly parallel version from the SPLASH benchmark suite written using the ANL macro package [88]. Figure 4.17 contains the corresponding speedup curves for two versions of the application. As for the iPSC/860 speedup curves, the baseline for these speedup curves is the stripped version. The data set is the same as for the iPSC/860 runs. The computation scales almost linearly to 32 processors.

One anomaly in the performance numbers is the difference in execution time between the serial, stripped and ANL versions. Further investigation reveals that the difference between the three versions is not caused by significant differences in the computation that they perform. We used the pixie instrumentation utility [129] to generate cycle counts for the three versions on DASH assuming a perfect memory system. The cycle counts for the serial and stripped versions were within 1 percent of each other, with four of the top five

[2]This task is the last task that the main processor creates, so it is likely to be the task left when the main processor finishes creating tasks and looks for another task to execute.

procedures in the profile consuming identical numbers of cycles. The cycle count for the ANL version was less than 4 percent more than the cycle counts for the other two versions. We therefore attribute the running time differences to memory system effects.

Another anomaly is that the Jade versions perform better than the ANL version. The major functionality difference between the two versions is that the ANL version only uses one copy of the final result data structure. The parallel tasks update this data structure every time they generate a contribution to the final result, using a lock to ensure the atomicity of the update. This strategy may generate more locking overhead and contention for access to the final result data structure.

	1	2	4	8	16	24	32
serial	3628.29	-	-	-	-	-	-
stripped	3285.90	-	-	-	-	-	-
mi.lo	3283.92	1655.66	835.88	425.40	223.13	152.83	118.54
fl.lo	3270.71	1648.96	833.19	423.14	220.63	153.03	119.48
mi.nl	3283.82	1651.12	844.58	434.32	228.81	168.04	125.16
fl.nl	3290.47	1648.60	832.91	434.36	229.84	160.82	124.74
ANL	3677.57	1834.83	927.96	501.00	254.04	175.85	144.10

Table 4.5: Execution Times for Water on DASH (seconds)

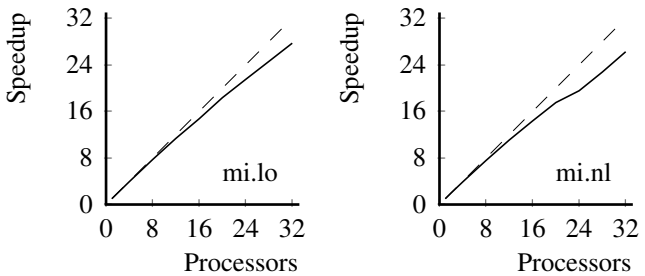

Figure 4.17: Speedups for Water on DASH

As for the iPSC/860 versions, we explore the dynamic behavior of the Jade DASH versions using event log data presented in the form of activity traces. Because all communication on DASH takes place implicitly as the program accesses remote data, the event

logs from the DASH runs contain no events that deal with communication. There are therefore no Sending, Waiting and Broadcasting traces for DASH activity traces. Figure 4.18 contains the complete Creating and Application activity traces for a 32 processor fl.lo (full log instrumentation with locality heuristic) DASH run. Like the iPSC/860 runs, the parallel phases dominate the execution and all 32 processors are busy executing application code for the majority of the computation.

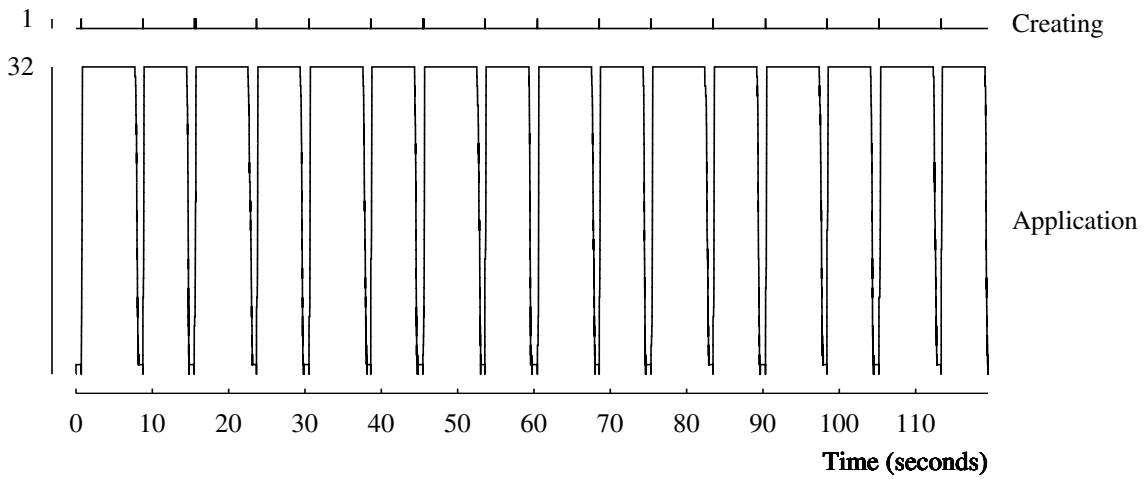

Figure 4.18: 32 Processor Activity Traces for Water (fl.lo) on DASH

4.4.8.1 Effect of the Locality Heuristic

The DASH locality heuristic, like the iPSC/860 locality heuristic, attempts to increase the locality of the computation by executing each task on the processor that owns its locality object. Figure 4.19, which displays the task locality percentages for two DASH runs, demonstrates that the locality heuristic is very effective at executing tasks on their target processors. Without the locality heuristic the percentage of tasks executed on their target processors drops quickly as the number of processors increases. Despite this dramatic difference in the assignment of tasks to their target processors, the use of the locality heuristic has very small impact on the overall performance. At 32 processors, for example, the fl.lo version, which uses the locality heuristic, runs only $124.75 - 119.48 = 5.27$

seconds faster than the fl.nl version, which does not use the heuristic.

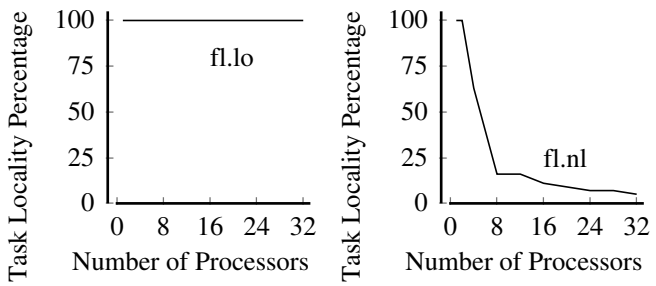

Figure 4.19: Task Locality Percentages for Water on DASH

The goal of the locality heuristic is to optimize the execution by reducing the number of remote accesses to shared objects. On DASH the time required to perform the remote accesses is counted as part of the task execution time. An improvement in the locality will therefore show up as a reduction in the task execution times. For this application reducing the task execution times may in turn improve the overall performance of the application by reducing the length of the parallel phases. The length of each parallel phase is determined by the length of the longest task. We therefore measure the effect of the locality heuristic by comparing the mean length of the longest task in each parallel phase for the two versions. For the fl.lo version running on 32 processors the mean longest task length is 6.73 seconds; for the fl.nl version it is 6.89 seconds.[3] The similar task execution times directly translate into similar overall execution times for the complete computation.

4.4.8.2 Differences Between the DASH and the iPSC/860 Versions

The major performance difference between the iPSC/860 and the DASH versions comes from the different communication mechanisms for shared objects. Without the adaptive broadcast optimization the serial transfer of the molecule positions and momentum data from the main processor to all of the other processors significantly degrades the performance of the iPSC/860 versions. On DASH every processor fetches all of the molecule positions

[3] We generate these numbers by processing the event log data.

and momentum data from the main processor, so there is at least as much data transferred out of the main processor for the DASH runs as for the iPSC/860 runs without the adaptive broadcast optimization. The relative performance of the DASH versions, however, is much better.

There are several reasons for the lack of performance degradation on DASH. First, on the iPSC/860 the main processor must manage the communication, so communication preempts any computation that would otherwise execute. In the iPSC/860 runs the execution of the task on the main processor is therefore delayed by the time required to transfer the molecule positions and momentum data to all of the other processors. On DASH, on the other hand, the remote requests for the molecule positions and momentum data are handled by the shared-memory hardware, not the main processor. The main processor can start its task's execution immediately and continue the execution while other processors concurrently fetch the molecule positions and momentum data from its memory.

Second, on DASH the communication is interleaved with the computation. This has the effect of spreading the communication out over a larger period of time, which tends to reduce contention at the main processor. On the iPSC/860 no task can start executing until its processor has received all of the remote data from the main processor. All of the communication therefore takes place at the very beginning of the parallel phase. One of the processors must wait until all of the other processors have received the new molecule positions and momentum data before its transfer begins. Its execution is therefore delayed both by the time required to transfer the data from the main processor to it and by the time required transfer the data from the main processor to all the other processors. A final difference is that the DASH communication hardware is somewhat more efficient than the iPSC/860 communication hardware.

4.4.9 String on the iPSC/860

Table 4.6 contains the execution times for String on the iPSC/860. Section 4.3.3 tells how the different version names identify the instrumentation and optimization levels for each run. Figure 4.20 contains the corresponding speedup curves for four versions of the application.

	1	2	4	8	16	24	32
serial	20270.45	-	-	-	-	-	-
stripped	19629.42	-	-	-	-	-	-
mi.lo.ab	18876.92	9496.70	4764.54	2415.14	1248.62	865.97	676.48
fl.lo.ab	17382.07	9473.24	4773.02	2418.75	1249.69	873.14	678.55
mi.nl.ab	18870.01	9504.86	4762.60	2431.85	1261.22	875.46	687.36
fl.nl.ab	18873.86	9529.52	4765.96	2424.12	-	869.27	680.94
mi.lo.nb	18876.97	9490.92	4758.89	2418.84	1253.26	874.64	690.72
fl.lo.nb	18877.42	9469.36	4765.68	2425.82	1255.29	874.18	689.57
mi.nl.nb	17397.49	9508.54	4764.26	2420.29	1272.67	887.83	709.57
fl.nl.nb	18852.93	9542.78	4774.49	2428.16	1258.48	878.88	699.33

Table 4.6: Execution Times for String on the iPSC/860 (seconds)

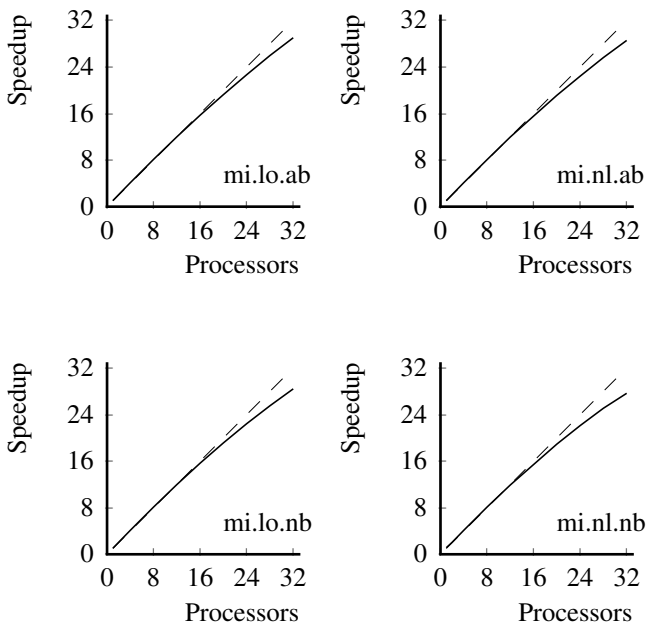

Figure 4.20: Speedups for String on the iPSC/860

The data set is from an oil field in West Texas and discretizes the 185 foot by 450 foot velocity image at a 1 foot by 1 foot resolution. It executes six iterations, with one parallel phase per iteration. The performance numbers are for the entire computation, including initial and final I/O.

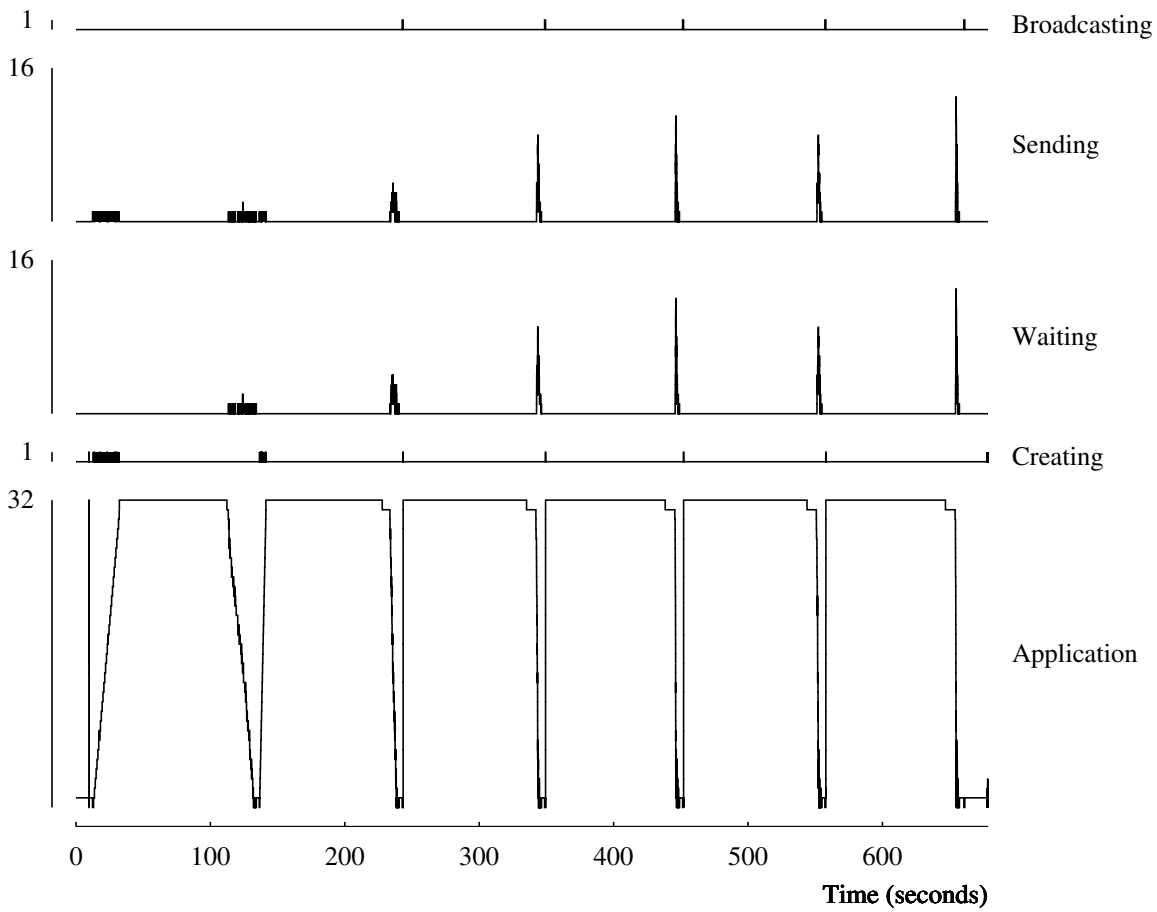

Figure 4.21: 32 Processor Activity Traces for String (fl.lo.ab) on the iPSC/860

For this data set the computation scales almost linearly to 32 processors, with none of the instrumentation or optimization levels making a substantial difference in the overall performance. Figure 4.12, which contains the complete activity trace for this application, illustrates the division into parallel and serial sections and shows that the computation spends the majority of its time in the parallel sections. We next describe the behavior of the

parallel computation with the aid of several sets of activity traces.

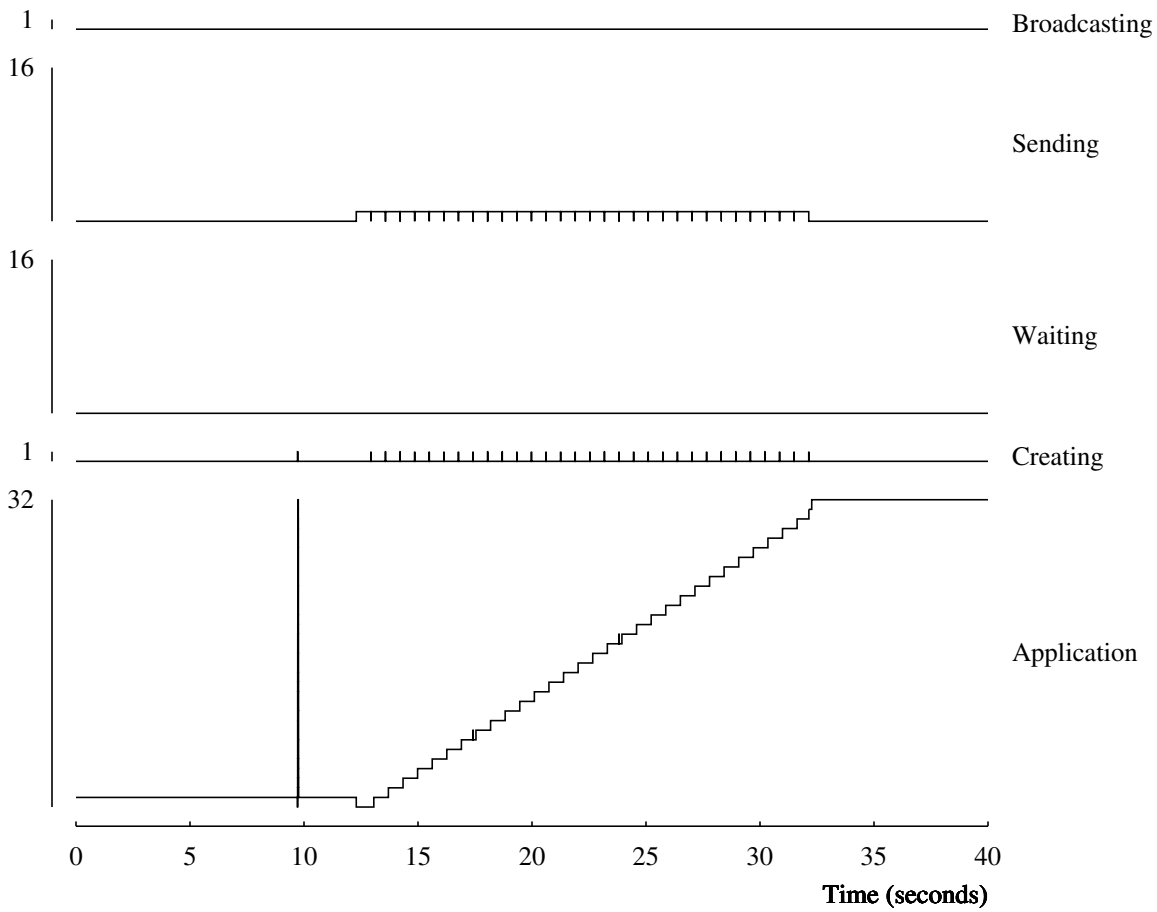

Figure 4.22: First Parallel Phase for String (fl.lo.ab) on the iPSC/860

We start our discussion with the first part of the computation. Figure 4.22 contains the corresponding activity traces. The program first executes an initialization phase, allocating shared objects and reading in input data. As part of this initialization phase it creates a set of tasks to perform some of the shared object allocations. These tasks generate the first spike in the Application trace. The program then proceeds on to the first parallel phase. Each task in this phase reads the initial velocity model and a set of global shared objects. These objects are written once in the initialization phase then read in the rest of the computation. When the child tasks are sent to processors for execution, the global objects

and initial velocity model are piggybacked onto the task messages. The global variables are together 1,192,264 bytes and the initial velocity model is 383,528 bytes, so each message is 1,575,792 bytes long plus a few hundred bytes for the task data. The main processor sends one such message to each of the other processors; when the processor receives the message it starts executing the task. The Application trace in Figure 4.22 shows how the time required to transfer the large messages staggers the task startups.

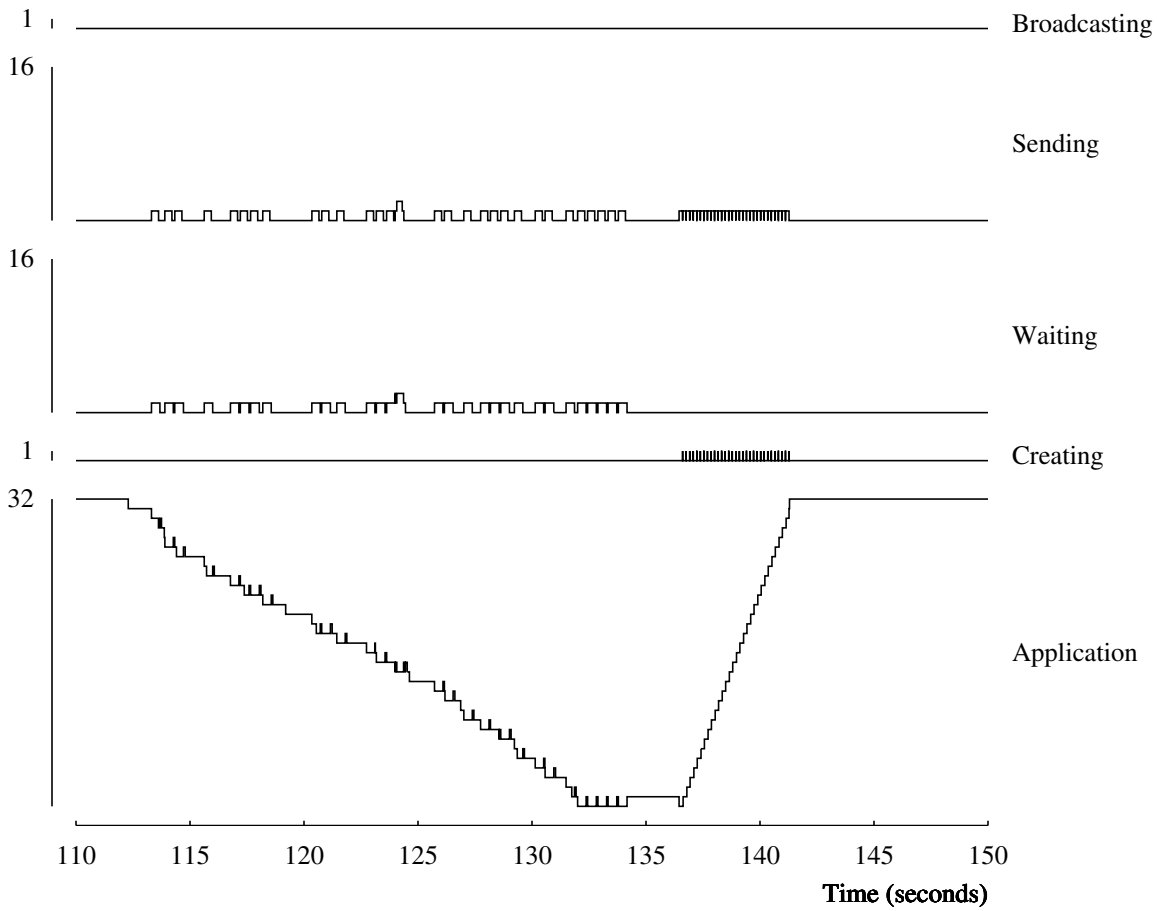

Figure 4.23: Second Parallel Phase for String (fl.lo.ab) on the iPSC/860

We next discuss the end of the first parallel phase, the next serial phase, and the start of the second parallel phase. Figure 4.23 contains the activity traces for this part of the computation. At the end of the first parallel phase the computation performs the parallel

reduction of the arrays used to generate the new velocity model. The staggered terminations of the tasks in the first parallel phase also stagger the execution of the parallel reduction. The computation next performs a serial phase to generate the new velocity model, then creates the tasks for the next parallel phase. Each task reads both the new velocity model and the global objects. Because copies of the global objects already exist on all of the processors, they generate no communication after the first parallel phase. Copies of the new velocity model are piggybacked on the task messages that start the second parallel phase, so the second parallel phase also has a staggered startup. As the Application trace in Figure 4.23 illustrates, the stagger is much less pronounced for the second phase than for the first phase because the task messages in the second phase do not carry the large global shared objects.

After the second phase the program performs four more parallel phases. In the second phase every processor reads the same version of the object containing the velocity model, so in subsequent phases the implementation broadcasts the new velocity model to all processors rather than piggybacking it on the task messages. Because the computation stores the velocity model for the first phase in a different object than for subsequent phases, the implementation does not apply the adaptive broadcast optimization until the third parallel phase. As Figure 4.24 illustrates, eliminating data from the task messages also eliminates almost all of the task startup stagger.

4.4.9.1 Effect of the Adaptive Broadcast Optimization

Without the adaptive broadcast optimization all of the parallel phases have a staggered startup similar to that in Figure 4.23. There are two reasons why the adaptive broadcast optimization has less effect on the overall performance for String than for Water. First, it is only used for four of the six iterations. Second, the size of the broadcasted data is smaller in comparison to the size of the tasks in the parallel phases for String than for Water. For example, the mean size of the longest task in each parallel phase for the fl.lo.ab version of String on 32 processors is 98.75 seconds. The size of the broadcasted object is 383,528 bytes long. When the implementation starts a parallel phase by first broadcasting the object then creating the parallel tasks, the broadcast takes approximately 0.70 seconds while the creation and distribution take approximately 0.018 seconds. When the implementation piggybacks

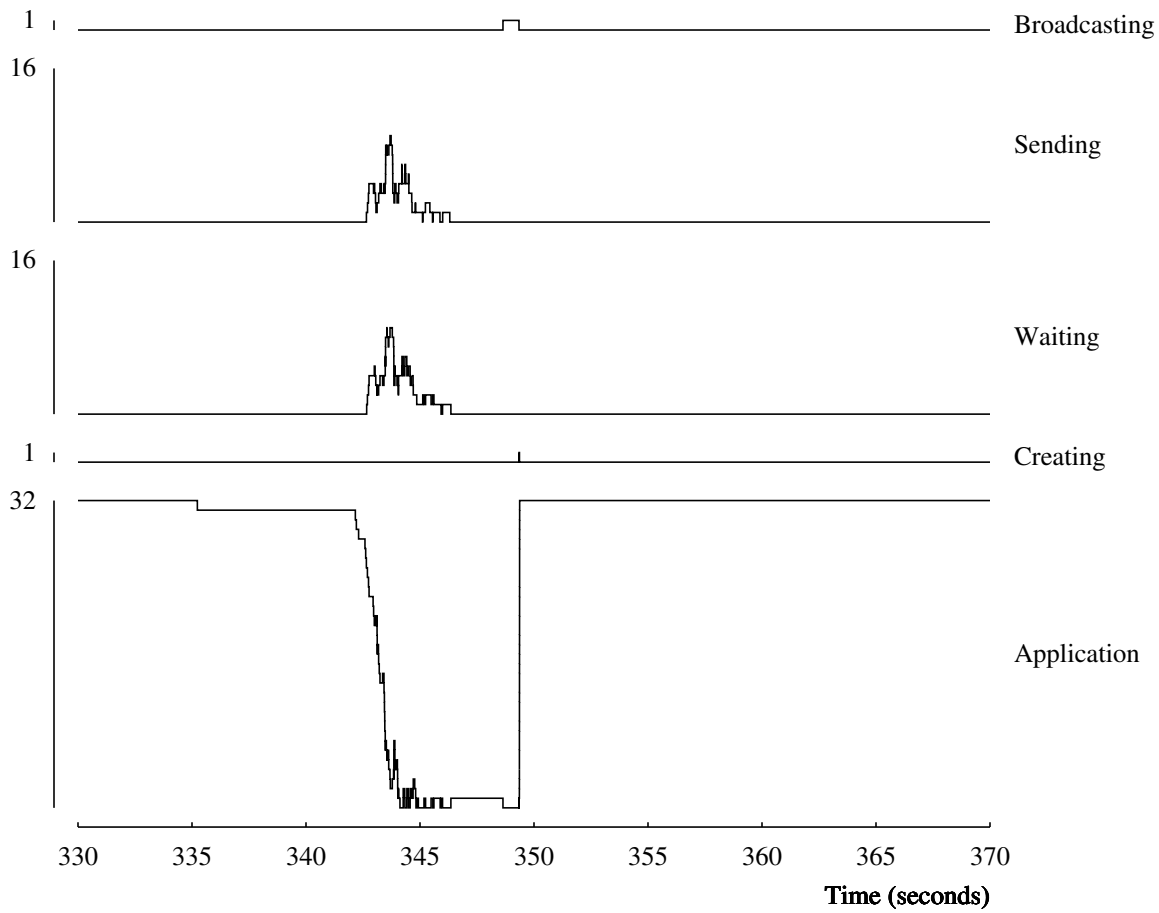

Figure 4.24: Fourth Parallel Phase for String (fl.lo.ab) on the iPSC/860

the object on to the task messages the time required to create and distribute the tasks goes up to approximately 4.82 seconds. For String the length of the parallel phase with the adaptive broadcast optimization is therefore approximately $0.70 + 0.018 + 98.75 = 99.46$ seconds; without the optimization it is $4.82 + 98.75 = 103.57$ seconds. Each parallel phase in the version with adaptive broadcast is therefore only about 4 percent faster. For Water on 32 processors the numbers are 4.87 seconds for the mean longest task in each parallel phase, 165,888 bytes for the broadcasted object, 0.31 seconds to broadcast the object, 0.018 seconds to create and distribute tasks without the piggybacked object and 2.25 seconds with the piggybacked object. The parallel phases are about $0.31 + 0.018 + 4.87 = 5.20$ seconds

with adaptive broadcast and $2.25 + 4.87 = 7.12$ seconds without adaptive broadcast. The parallel phases with adaptive broadcast are therefore about 37 percent faster.

4.4.9.2 Effect of the Locality Heuristic

The basic issues associated with the presence or absence of the locality heuristic are the same for String as for Water (see Section 4.4.7.4 for a discussion of the locality heuristic effect for Water). As Figure 4.25 illustrates, the locality heuristic always executes tasks on their target processors. Without the locality heuristic the task locality percentage drops as the number of processors increases.

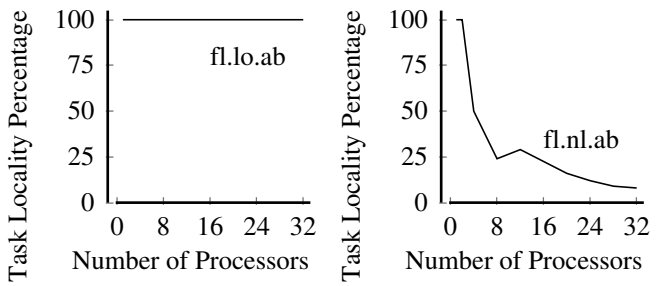

Figure 4.25: Task Locality Percentages for String on the iPSC/860

Failing to apply the locality heuristic generates a brief flurry at the beginning of each parallel phase as the processors exchange replicas of the data structures used to hold the backprojected time differences. As for Water on the iPSC/860, this communication has very little impact on the overall performance of the application.

4.4.10 String on DASH

Table 4.7 contains the execution times for several versions of String running on DASH. Figure 4.26 contains the corresponding speedup curves for two versions of the application. As for the iPSC/860 speedup curves, the baseline for these speedup curves is the stripped version. The data set is the same as for the iPSC/860 runs. The computation scales almost linearly to 32 processors. The locality optimization level has a significant impact on the

performance – the versions with the locality heuristic run faster than the versions without it.

	1	2	4	8	16	24	32
serial	20594.50	-	-	-	-	-	-
stripped	19314.80	-	-	-	-	-	-
mi.lo	19280.41	9794.00	4926.22	2502.83	1287.95	886.70	693.26
fl.lo	19621.15	9774.07	5003.69	2534.62	1320.00	903.95	705.84
mi.nl	19458.52	10325.28	4903.36	2538.85	1325.31	937.14	758.76
fl.nl	19396.12	9756.71	5017.82	2559.44	1350.06	948.73	769.21

Table 4.7: Execution Times for String on DASH (seconds)

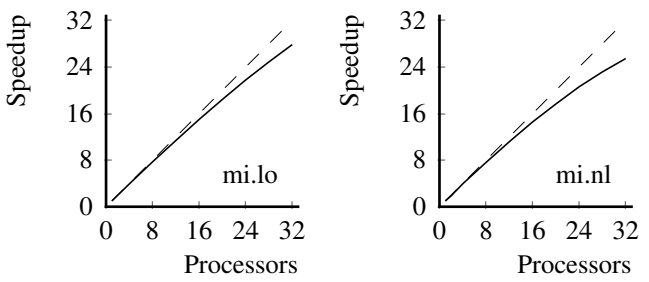

Figure 4.26: Speedups for String on DASH

We start our exploration of the locality effects by considering the task locality percentages as presented in Figure 4.27. This figure shows that with the locality heuristic, every task executes on its target processor (the processor that owns its locality object). The locality objects in this computation are the replicas of the data structure used to hold the backprojected differences between the simulated and measured ray travel times. Without the locality heuristic the number of tasks that execute on their target processor drops rapidly. In the absence of the locality heuristic one would expect the tasks to generate more remote references as they update the backprojected difference data structures.

We explore the effect of the locality heuristic on the tasks in the parallel phases using an approach similar to that for the Water application presented in Section 4.4.8.1. We

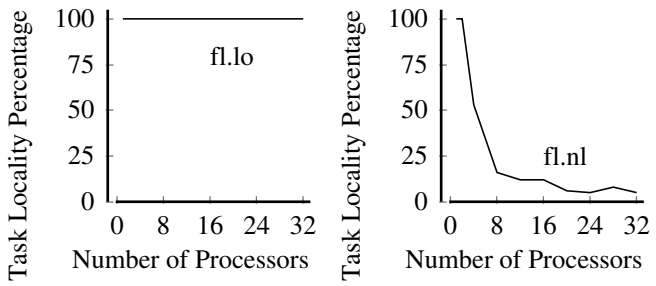

Figure 4.27: Task Locality Percentages for String on DASH

first calculate the mean length of the longest task in the six parallel phases that perform significant amounts of computation. For the fl.lo version on 32 processors the mean longest task time is 106.86 seconds; for the fl.nl version it is 107.82 seconds. This translates into a small difference in the amount of time the reduction steps on the main processor spend waiting for other reduction steps to finish. The fl.lo version spends 7.78 seconds waiting, while the fl.nl version spends 9.24 seconds waiting. There are also small differences caused by the single task queue in the fl.lo version.

None of these factors comes close to accounting for the performance difference between the two versions. Further investigation reveals that almost all of the difference is caused by interactions with the operating system's memory management system. During the initialization phase the application allocates shared objects for the replicated data structures used to hold the backprojected differences. Each such data structure is composed of a small normal shared object that points to two large part objects. In our timing runs each of the large part objects is 383,528 bytes long.

The application first creates the small shared objects in the main thread, then creates child tasks that allocate the two part objects for each replica. The shared-memory implementation of Jade always allocates part objects in the same memory module as the corresponding normal object. With the locality heuristic the new memory is allocated out of the local memory module of the processor executing the child task. Without the locality heuristic many part objects are allocated out of remote memory modules.

Each of the allocations first invokes the Unix *sbrk* routine to acquire the memory for

the allocated part object. For each of the allocated pages it first accesses the page to fault it in. It then performs a system call to migrate the page to the desired memory module, then accesses the page one more time. For multiple parallel memory allocations the memory management system executes these operations serially.

It turns out that the memory allocation process is much faster for local than for remote memory modules. In the 32 processor fl.lo run, for example, processors 1 through 32 spend 6.87 seconds in the memory allocation process, for a mean time of 0.11 seconds per 383,528 byte memory allocation. For the corresponding fl.nl run, the processors spend 58.41 seconds allocating memory, with the mean time per allocation increasing to 0.94 seconds.

Without the locality heuristic, the extra time required for the serialized memory allocations delays the execution of the rest of the program by approximately $58.41 - 6.87 = 51.54$ seconds. This is by far the largest contributing factor to the difference in performance between the versions with and without the locality heuristic.

It is difficult to see these interactions with the memory management system as a fundamental performance issue – it should be possible to make the operating system interactions run much faster than they currently do. It is important to realize, however, that the performance difference is not caused by memory system effects in the parallel computation phases.

4.4.11 Search on the iPSC/860

Table 4.8 contains the execution times for Search on the iPSC/860. Figure 4.28 contains the corresponding speedup curves for four versions of the application. The data set simulates six different solids at 10 initial beam energies. Each solid/energy data point requires the Monte-Carlo simulation of 5000 electron trajectories. The performance numbers are for the entire computation. For this data set the computation scales almost linearly to 32 processors.

The activity traces for a 32 processor run in Figure 4.29 show why the application performs as well as it does. The computation first executes a short initialization phase, then one long parallel phase. There is a short communication phase at the end of the parallel phase as the computation combines the results of the parallel tasks, then a short phase to

	1	2	4	8	16	24	32
stripped	1284.07	-	-	-	-	-	-
mi.lo.ab	1420.05	715.49	371.87	181.66	94.33	62.75	47.63
fl.lo.ab	1368.55	689.98	349.26	174.66	89.58	60.50	46.03
mi.nl.ab	1316.17	661.25	350.68	168.37	86.05	58.24	44.30
fl.nl.ab	1402.41	709.91	366.15	179.62	91.46	62.00	47.09
mi.lo.nb	1331.28	665.83	345.08	171.36	92.97	60.35	47.68
fl.lo.nb	1458.11	737.29	383.31	190.50	96.82	64.41	48.88
mi.nl.nb	1486.80	750.25	395.34	191.78	98.33	65.79	49.98
fl.nl.nb	1444.19	731.41	364.41	186.50	95.67	63.68	48.65

Table 4.8: Execution Times for Search on the iPSC/860 (seconds)

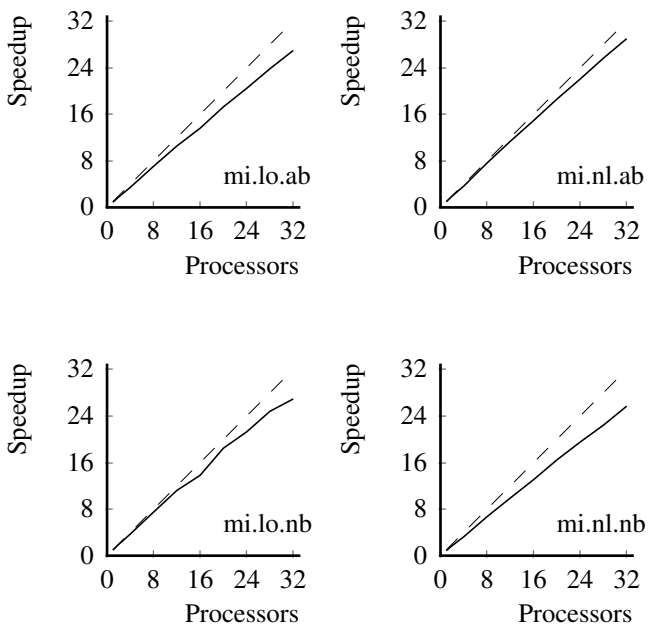

Figure 4.28: Speedups for Search on the iPSC/860

print out the results. The load is well balanced during the parallel phase and the parallel phase is large enough to amortize the very small relative serial overhead.

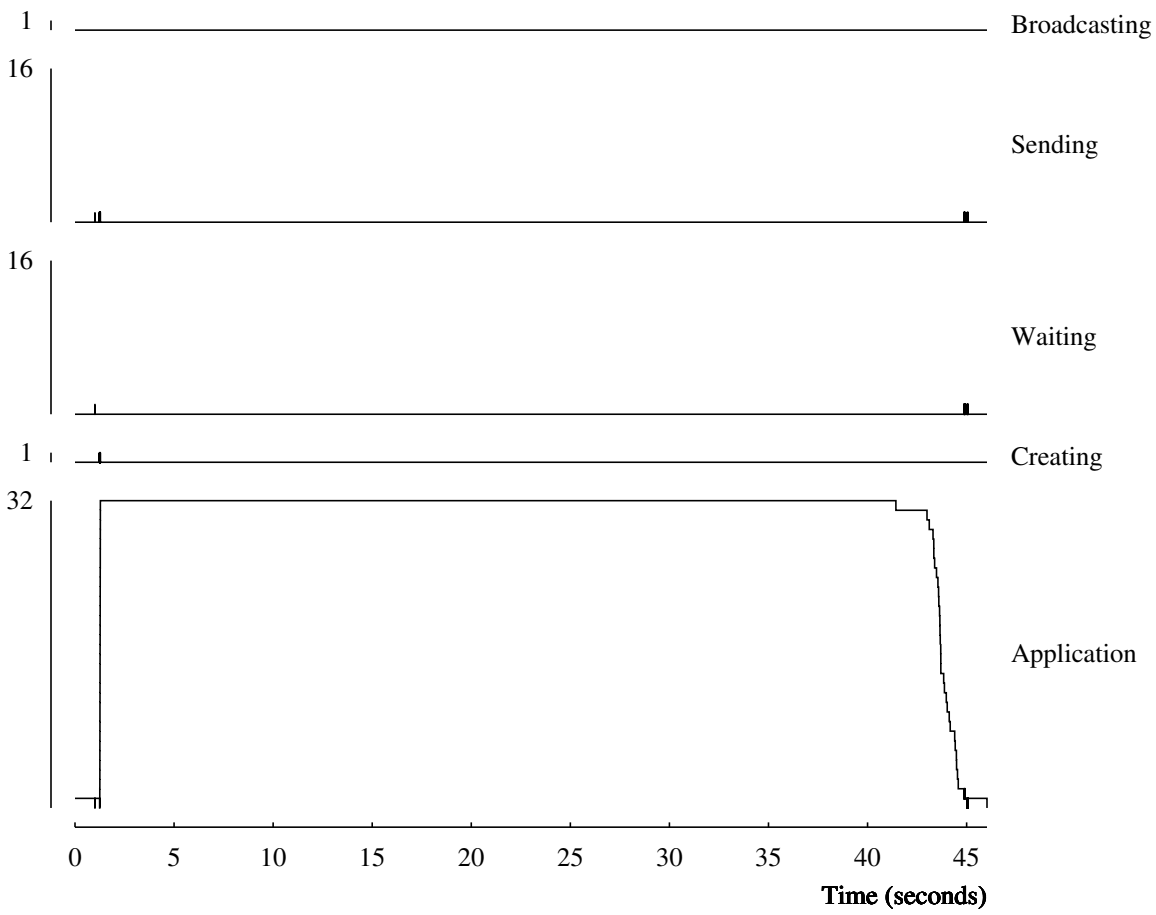

Figure 4.29: 32 Processor Activity Traces for Search (fl.lo.ab) on the iPSC/860

4.4.12 Search on DASH

Table 4.9 contains the execution times for several versions of Search running on DASH. Figure 4.30 contains the corresponding speedup curves for two versions of the application. There are measurable performance differences between the versions with and without the locality heuristic. This is surprising because the parallel tasks execute with almost no

accesses to shared objects. All of the versions should therefore have almost identical execution times.

	1	2	4	8	16	24	32
stripped	1652.91	-	-	-	-	-	-
mi.lo	1813.67	910.70	459.16	242.34	130.46	77.77	61.06
fl.lo	1654.42	829.15	417.51	208.24	116.30	81.88	62.60
mi.nl	1655.87	829.02	417.13	208.40	132.57	77.64	53.52
fl.nl	1657.96	829.15	416.93	209.66	127.63	70.66	53.05

Table 4.9: Execution Times for Search on DASH (seconds)

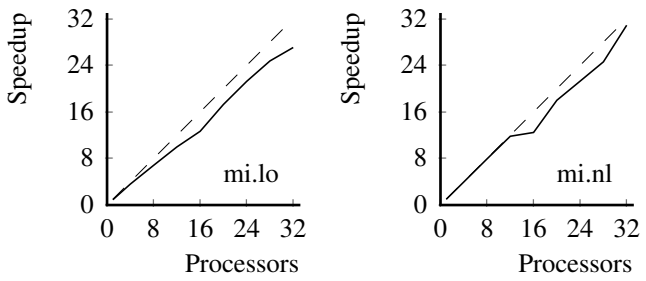

Figure 4.30: Speedups for Search on DASH

We start our investigation of the performance differences with the activity traces for two of the runs. Figure 4.31 shows the activity trace for the 32 processor fl.lo run, while Figure 4.32 shows the activity trace for the 32 processor fl.nl run. The traces reveal a load imbalance in the fl.lo version – two of the tasks take about 9 seconds longer to finish than the other tasks. The fl.nl version suffers from no such load imbalance.

It turns out that, for the fl.lo run, the tasks on processors 16 and 1 are the tasks that take significantly longer to execute. Further investigation yields data consistent with the hypothesis that the performance variation is caused by memory system effects.

We used the hardware performance monitor on DASH [82] to count the number of cache misses on each processor. The monitor counts local cache misses (misses satisfied from within the cluster issuing the reference) and remote cache misses (misses satisfied

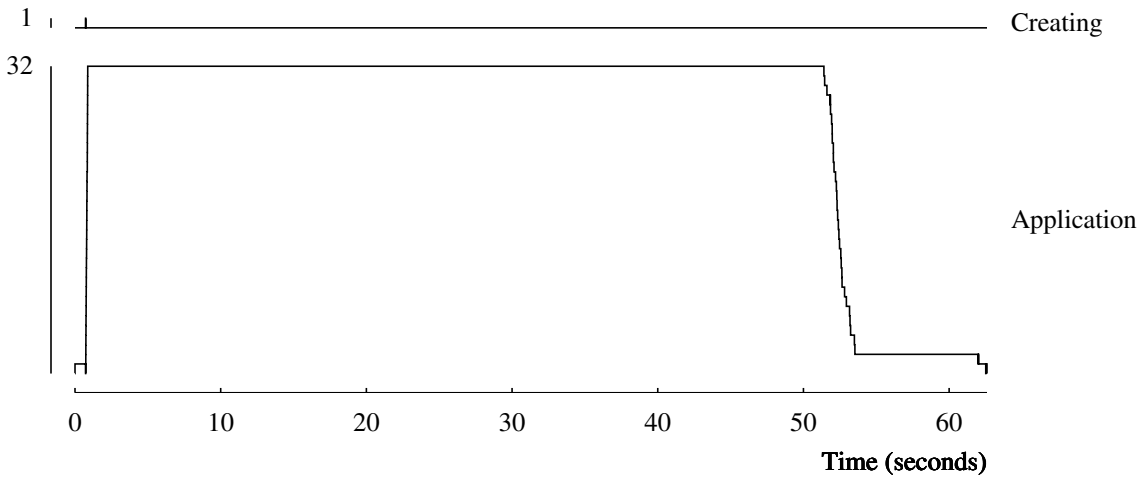

Figure 4.31: 32 Processor Activity Traces for Search (fl.lo) on DASH

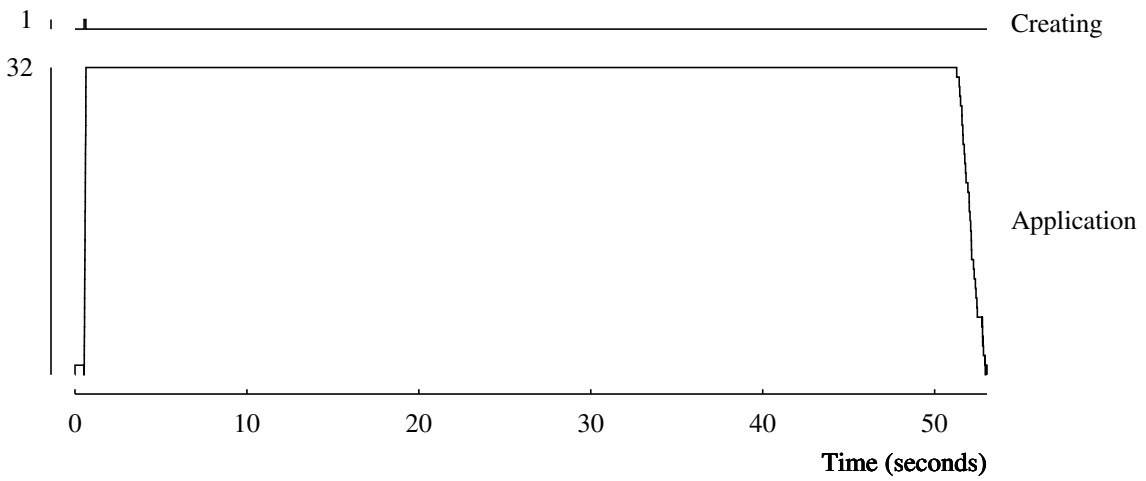

Figure 4.32: 32 Processor Activity Traces for Search (fl.nl) on DASH

from of a remote cluster) separately. The performance monitor reveals that the task on processor 16 has approximately 12.3 million local read misses and the task on processor 1 has approximately 9.8 million local read misses. The next two highest processors have .78 and .056 million local read misses. The number of remote misses is substantially smaller. For the fl.nl version running on 32 processors (which executes the same instructions in the application program and exhibits almost perfect speedup) the two highest processors have .040 and .028 million local read misses.

4.5 Volume Rendering

The Volume Rendering application generates a sequence of images, each corresponding to a given view of a set of volume data. The application manipulates three major data structures: the voxel array, which holds the volume data, the image array, which is an array of pixels holding the rendered image, and a shading-table array. For each view the program first computes the shading table then renders the image onto the image array.

The computationally intensive section of the Volume Rendering application shoots a ray from each pixel into the voxel array. The computation associated with this ray determines the value of the pixel. It is easy to see that the computation can trace all of the rays in parallel. Each ray only reads part of the voxel array and writes its pixel in the image array. The basic computational issues are maximizing locality, minimizing overhead and balancing the load. Because rays from adjacent pixels tend to access similar regions of the voxel array, shooting rays from contiguous sets of pixels on the same processor enhances locality. The original developers [100] determined that distributing pixels to processors in 8 by 8 chunks yields acceptable locality.

Because the rays generate different amounts of computation, a static assignment of 8 by 8 chunks to processors has the potential to generate an unbalanced load. Ideally, the programmer would create one task per chunk and rely on the implementation to dynamically balance the load by moving chunks to idle processors. With the current implementation this strategy generates more concurrency management overhead than strategies that aggregate chunks into larger-grain tasks. Static analysis would eliminate this problem by allowing the implementation to employ a task management strategy efficient enough to profitably

exploit concurrency at the granularity of the 8 by 8 pixel chunks. In the current version the programmer minimizes the overhead by maximizing the task size. The application matches the number of tasks to the number of processors, mapping the 8 by 8 pixel chunks onto tasks in a round robin fashion.

The image data structure is allocated in one contiguous chunk. Each image task computes its part of the image into a private data structure, then copies the result into the image data structure. This allows all of the tasks to execute concurrently with a serial copyback phase at the end of the image computation. Because the tasks write disjoint pieces of the image, providing more precise access declarations would allow the tasks to concurrently write the image data structure, eliminating the serial copyback phase. Because the shading table is also allocated in one chunk, the shading-table computation has a similar copyback phase.

For the timing runs the application rendered the *head* dataset described in [100]. This dataset has a 256 by 256 by 226 voxel array and a 416 by 416 image array. Each element of the voxel array takes up 4 bytes; each element of the image array takes up 1 byte. The voxel array dominates the storage requirements; it is approximately 58 megabytes long. The size of the voxel array interacts poorly with the communication mechanism of the message-passing implementation. In this implementation each object is a unit of communication and is transferred as a unit between processors. If a processor accesses part of an object the entire object must fit into the memory associated with the processor. Because the voxel array is larger than the 16 megabyte per-node memory on the iPSC/860, this application will not run on this platform. A communication mechanism that allowed the implementation to store large objects in a distributed fashion across all of the memories would eliminate this problem.

Table 4.10 presents the execution times for Volume Rendering; Figure 4.33 presents the corresponding speedup curves. We were able to obtain performance numbers for the explicitly parallel version described in [100]. This version was written using the ANL macro package [88]. The Jade performance tails off quickly after 16 processors, while the explicitly parallel version scales almost linearly to 32 processors. We first explore the behavior of the Jade versions, then discuss the reasons for the performance difference between the explicitly parallel version and the Jade versions.

	1	2	4	8	16	24	32
stripped	32.44	-	-	-	-	-	-
mi.lo	33.06	16.49	8.70	4.58	2.84	2.10	2.01
fl.lo	33.72	18.46	8.89	4.68	2.73	2.21	1.89
mi.nl	33.07	16.52	8.65	4.71	2.67	2.12	2.08
fl.nl	33.23	16.37	9.12	4.62	2.71	2.12	1.96
ANL	31.60	15.00	7.67	3.88	1.97	-	1.01

Table 4.10: Execution Times for Volume Rendering on DASH (seconds)

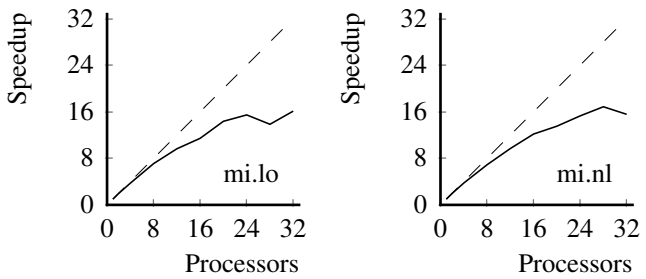

Figure 4.33: Speedups for Volume Rendering on DASH

Figure 4.34, which contains the activity traces for the 32 processor fl.lo run, reveals several sources of inefficiency. These sources include slow task creation and startup, a load imbalance, and the serial copyback phase. The serial copyback phase lasts about .45 seconds and is visible at the end of the Application trace. The staggered task completions in the Application trace at the end of the second parallel phase reveal the load imbalance.

We next explore the slow task creation and startup effects. One can see the slow task startup in the staggered task creation times for the two parallel phases. The first parallel phase computes the shading table, and the task startups are so staggered that this phase never comes close to executing the 32 tasks concurrently. Even though the startups for the rendering phase are less staggered, the effect is still clearly visible in the Application trace. The Creating trace highlights the slow task creation effect – it takes almost 0.2 seconds to create the tasks for the two phases.

We attribute the slow task startup for the first phase to memory system effects as the

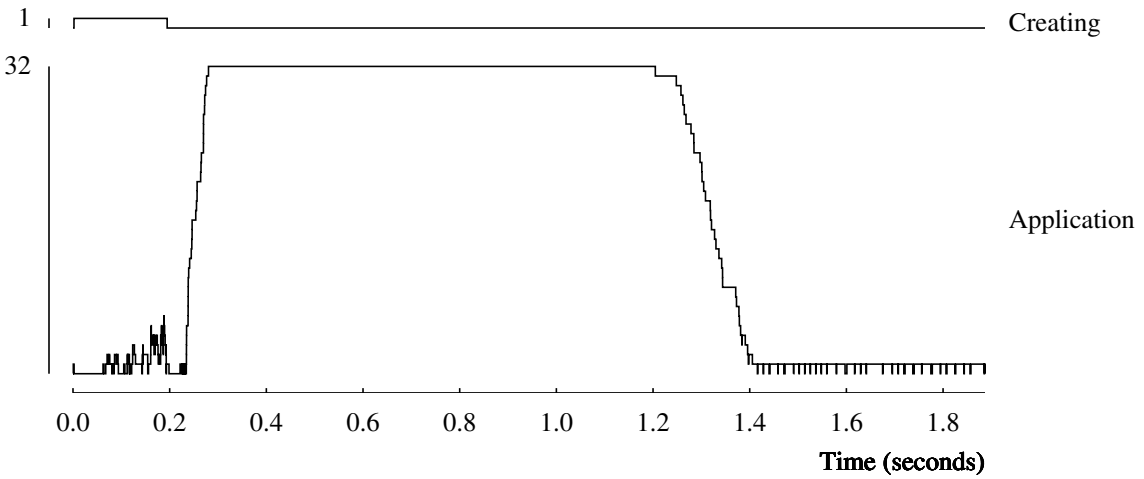

Figure 4.34: 32 Processor Activity Traces for Volume Rendering (fl.lo) on DASH

processors access data and execute code that they have not recently accessed or executed. Measurements of a small benchmark program designed to explore this behavior indicate that processors suffer substantial delays in very short instruction sequences that remove tasks from task queues. These delays in turn delay the execution of the tasks. Interactions with the operating system triggered by memory system effects as the processor executes stale code and accesses stale data may cause these delays.

The task queue removal delays interact with the fact that all of the tasks have the same locality object (the voxel array) to generate the slow task creation effect. Because all of the tasks have the same locality object, they are all inserted into the same task queue. When the main processor attempts to insert a newly created task into the task queue, it may find a delayed processor holding the task queue's lock as it removes a previously created child task. The main processor must then wait to acquire the lock, which delays the creation of subsequent child tasks.

We believe that the staggered task startups in the second phase are caused primarily by contention on the lock that controls access to the task queue. When the second phase starts all of the processors have recently executed the task queue removal code during the startup of the first phase. The second phase startup therefore suffers no memory system effects, leaving lock contention as the only source of task startup delay. The fact that the second

phase starts up much faster than the first phase supports this hypothesis.

To test the hypothesis that the delays in the task queue removal code are caused by memory system effects, we developed a version of the application that explicitly places the tasks on the different processors for execution. All of the tasks are therefore inserted into different task queues, which eliminates any contention for task queue locks. Figure 4.35 contains the activity trace for this version. The Creating trace shows that there is no slow task creation problem – all of the tasks for the entire execution have been created shortly after the program begins execution. There is, however, still a slow task startup problem in the first parallel phase. But the second parallel phase shows absolutely no sign of staggered task startups. This data is consistent with the hypothesis that each processor suffered from memory system effects during the startup of the first parallel phase but not during the startup of the second parallel phase.

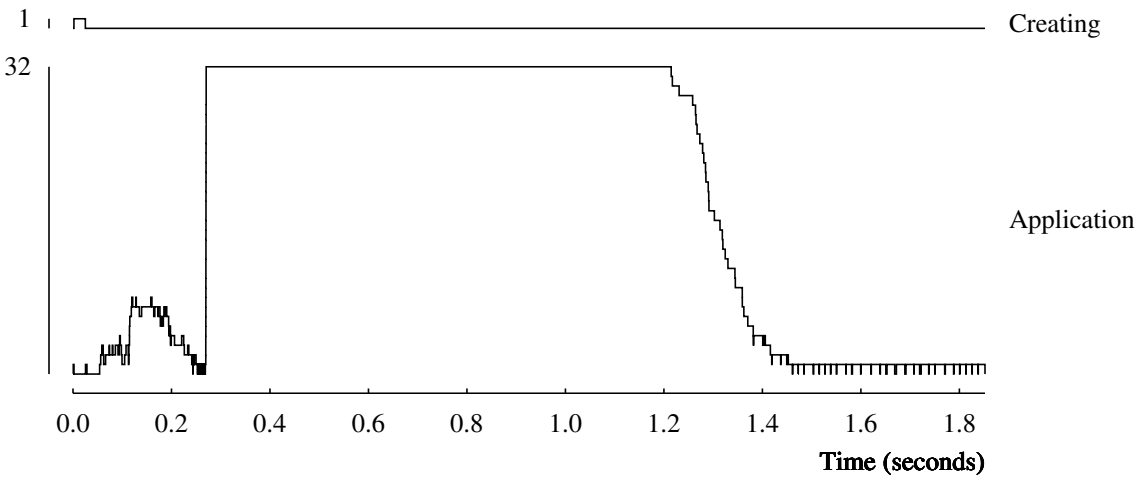

Figure 4.35: 32 Processor Activity Traces for Volume Rendering (fl.at) on DASH

Unlike the Jade version, the explicitly parallel version scales almost linearly to 32 processors. We attribute this superior performance to several factors. First, the ANL version suffers from neither the slow task creation nor the slow task startup effects. All of the tasks are created once at the beginning of the computation and synchronize using barriers after every parallel phase. The tasks concurrently write the image data structure, which eliminates the serial copyback phase. Finally, the program initially distributes the 8 by 8

pixel chunks to tasks in a round robin fashion, but uses an application-specific dynamic load balancing algorithm to redistribute the chunks to idle tasks if the load becomes unbalanced. This algorithm eliminates the load imbalance associated with a static assignment of chunks to tasks.

4.6 Panel Cholesky

Panel Cholesky is a version of the sparse Cholesky factorization algorithm presented in Section 2.3.2. In this version the columns of the matrix have been aggregated into larger-grain objects called panels. This aggregation increases both the data and task grain sizes. See Rothberg's thesis [113] for a comprehensive discussion of parallel sparse Cholesky factorization algorithms, including this one.

The serial program stores all of the panels contiguously in one long array. In the Jade version each panel is an object, so the programmer had to change the parts of the code that allocated and accessed the panels. Because the serial program logically accessed the matrix at the granularity of panels (even though the panels were allocated contiguously), this data structure modification did not significantly complicate the development. The programmer used the `create_at_object` construct to explicitly allocate panels to memories in a round-robin fashion omitting the memory associated with the first processor.[4] For the version with explicit task placement the programmer placed each task on the processor whose memory contains the updated panel. See Section 2.3.2 for a description of how tasks access panels.

There was one other data structure modification besides the decomposition of the matrix into panels. The programmer aggregated multiple global variables into a single shared object using a technique similar to that illustrated in Figure 2.16. As for other Jade applications, the global variables are written once at the start of the computation and read by the rest of the computation. The aggregation simplified the access declarations and drove down the access declaration overhead.

[4]Panel Cholesky has a small grain size and creates tasks sequentially. For such applications the best performance is obtained by devoting one processor to creating tasks. Allocating no panels on the first processor biases the locality heuristic away from executing worker tasks on the processor that creates the tasks.

4.6.1 Panel Cholesky on the iPSC/860

Table 4.12 contains the execution times for the Panel Cholesky factorization on the iPSC/860. The timing runs factor the BCSSTK15 matrix from the Harwell-Boeing sparse matrix benchmark set[37]. The performance numbers only measure the actual numerical factorization, omitting an initial symbolic factorization phase. To a first approximation the execution times are roughly comparable for all of the runs.

	1	2	4	8	16	24	32
serial	27.60	-	-	-	-	-	-
stripped	28.53	-	-	-	-	-	-
mi.at.ab	53.30	53.10	31.53	31.81	33.87	35.52	38.43
fi.at.ab	54.56	50.18	31.56	32.50	34.41	36.38	38.17
mi.lo.ab	53.62	34.11	33.91	35.54	43.84	47.77	49.73
fi.lo.ab	54.54	34.17	33.65	35.97	43.73	47.63	50.83
mi.nl.ab	53.35	103.90	100.60	75.82	59.35	55.42	58.97
fi.nl.ab	54.43	107.43	99.39	75.84	59.02	56.41	59.45
mi.at.nb	36.48	52.80	31.41	31.76	33.88	35.78	38.34
fi.at.nb	37.25	49.76	31.29	32.01	34.92	35.87	38.16
mi.lo.nb	36.70	33.65	31.98	35.48	42.87	46.15	49.49
fi.lo.nb	37.28	34.05	33.51	35.35	43.23	46.90	49.88
mi.nl.nb	36.53	102.64	100.94	75.21	59.48	56.13	58.71
fi.nl.nb	37.29	110.00	99.81	75.56	58.65	55.46	59.20

Table 4.11: Execution Times for Panel Cholesky on the iPSC/860 (seconds)

The execution times do not vary dramatically as the number of processors changes because the mean task execution time is close to the overhead required to execute a remote task. For all of the executions the mean task execution time is about 2 milliseconds and the majority of the tasks declare that they will access three objects. As Figure 3.5 shows, the total overhead on the iPSC/860 for a task that executes remotely and declares that it will access three objects is about 1.6 milliseconds. From the numbers in Section B.1.1 we can calculate that about 1.2 milliseconds of this overhead is incurred on the creating processor. The serial task management overhead significantly limits the performance. The extra communication incurred by the parallel versions further reduces the attractiveness of parallel execution relative to serial execution for this application.

We next explore several of the variations in the running times. The versions with explicit task placement run somewhat faster than the versions with the locality heuristic, which in turn run faster than the versions without the locality heuristic. Figure 4.36 displays the task locality percentages for these three versions. The version with explicit task placement has a higher task locality percentage[5] than the version with the locality heuristic, which in turn has a higher percentage than the version without the locality heuristic. The scheduling algorithm in the version with the locality heuristic executes tasks on non-target processors in an attempt to balance the load.

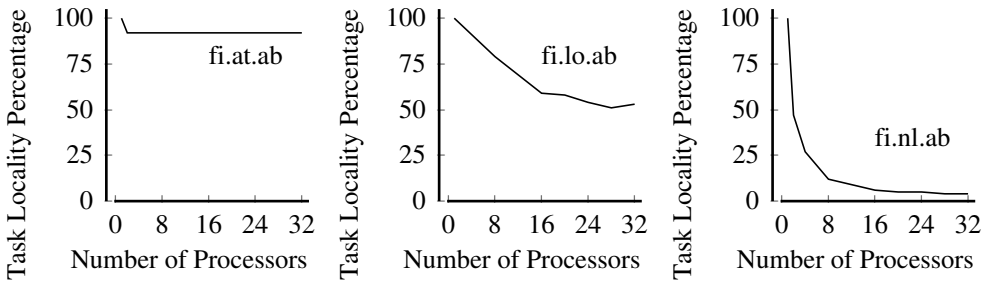

Figure 4.36: Task Locality Percentages for Panel Cholesky on the iPSC/860

We next consider the effect of the locality optimization level on the communication behavior. Figure 4.37 presents the mean amount of data each processor sent and received during the course of the computation. More precisely, the number plotted for each point on the curve is the total amount of data sent plus the total amount of data received (in megabytes) divided by the number of processors in the computation.[6] These curves illustrate that the amount of communication is correlated with the overall execution time of the program, which supports the hypothesis that communication overhead is the source of the performance differences between the versions with different locality optimization levels.

[5]The task locality percentage for the version with explicit task placement is not 100 percent. This is because the main processor initializes all of the panel objects, so they are all owned by the main processor at the start of the parallel phase. The first time a panel is accessed by a task in the parallel phase, the main processor owns the panel, not the processor executing the task.

[6]These numbers are generated using the message data described in Section 4.3. The mean communication volume for the 1 processor run is not plotted.

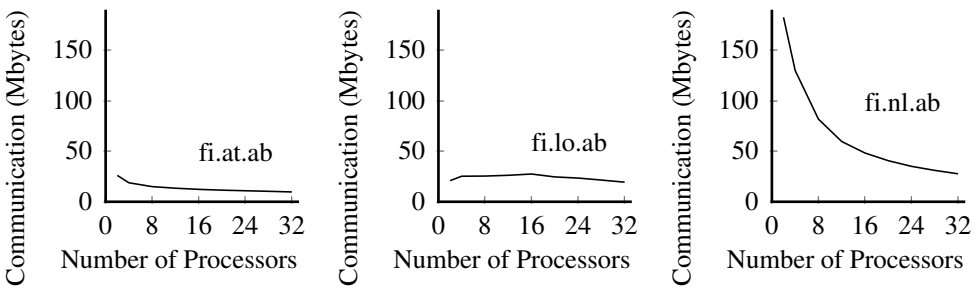

Figure 4.37: Mean Communication Volume for Panel Cholesky on the iPSC/860

4.6.2 Panel Cholesky on DASH

Table 4.12 contains the execution times for Panel Cholesky on DASH. Figure 4.38 contains the corresponding speedup curves. The input data set is the same as for the iPSC/860 runs. While the computation scales much better on DASH than on the iPSC/860, it still does not scale very well – the maximum speedup for the standard configuration of the Jade implementation is 5.4 for the mi.at version on 32 processors. We were also able to obtain performance numbers for two explicitly parallel versions of this computation: a version written in COOL [30] and a version written in the ANL macro package [113]. Although both of these versions perform better than the Jade versions, their performance is not spectacular. The COOL version only achieves a speedup of 8 out of 24 processors, while the ANL version only achieves a speedup of 9.9 out of 24 processors. The fastest Jade version achieves a speedup of 4.8 on 24 processors.

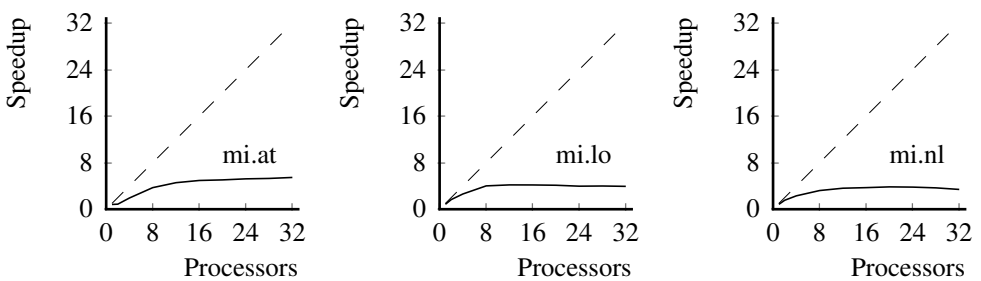

Figure 4.38: Speedups for Panel Cholesky on DASH

	1	2	4	8	16	24	32
serial	26.67	-	-	-	-	-	-
stripped	28.91	-	-	-	-	-	-
mi.at	35.15	33.33	14.96	7.79	5.85	5.54	5.30
fi.at	35.71	33.64	15.24	7.82	5.95	5.61	5.76
mi.lo	33.84	17.47	11.03	7.21	6.93	7.29	7.36
fi.lo	34.94	17.99	11.77	7.53	7.30	7.43	7.86
mi.nl	34.55	18.76	12.57	9.01	7.80	7.60	8.50
fi.nl	35.09	18.99	12.97	9.29	7.88	8.00	8.48
COOL	34.97	18.63	10.36	6.82	4.10	3.33	-
ANL	29.90	15.27	8.20	4.67	2.98	2.70	-

Table 4.12: Execution Times for Panel Cholesky on DASH (seconds)

The ANL macro package and COOL versions of this application differ substantially from the Jade version. The ANL macro package and COOL are explicitly parallel programming systems that allow the programmer to control the computation at a low level. Both the ANL macro package and COOL versions exploit this control to use a highly tuned, application-specific concurrency exploitation algorithm. This algorithm first analyzes the structure of the matrix in a preprocessing phase, then uses the extracted information to drive an application-specific task queue algorithm that generates the parallel computation. The Jade version, on the other hand, contains neither a preprocessing phase to extract scheduling information about the computation nor an application-specific task queue algorithm. It instead uses the general-purpose concurrency exploitation algorithms embedded in the Jade implementation. While the Jade approach places less of a burden on the programmer, for this application it also generated poorer performance than the explicitly parallel approach.

We next analyze the factors that contribute to the poor performance of the Jade versions. The first factor is an inherent lack of concurrency in the application. Even for the ANL version the maximum speedup is less than 10 for a 24 processor run, with much of the performance loss caused by an inherent lack of concurrency [113].

A second performance factor is that the general-purpose object queue and task queue algorithms in the Jade implementation are less efficient than the special-purpose algorithms used in hand-coded implementations. The overhead associated with the Jade synchronization and scheduling algorithms is especially severe for this application because much of

it is serialized – the Jade version of Panel Cholesky creates tasks sequentially. While the sequential task creation is not itself the critical path (even for the fastest runs the sequential task creation time takes only about one half of the total execution time), it may delay the creation of enabled tasks. This delayed enabled task creation wastes concurrency, potentially hurting the performance if an idle processor could have executed the task sooner. The mechanism in the Jade implementation that suppresses excess task creation (see Section 3.5.3) may exacerbate this effect. This mechanism suspends any task that creates a child task when the number of outstanding tasks is above a suspension threshold, preventing the implementation from searching for parallel tasks beyond a limit imposed by the suspension threshold. This wastes concurrency when, in fact, executable tasks exist beyond the limit.

In the current Jade implementation the task suspension threshold is set to 800 tasks, with the execution of suspended tasks resuming when the number of outstanding tasks drops below 700. At this threshold the implementation often suspends the main thread (which creates all of the child tasks) to avoid excess task creation. For example, on 32 processors the main thread in the fi.at version suspended 27 times to avoid excess task creation. We explore the effect of the suspension mechanism by raising the task suspension threshold. We first decrease the default number of declarations per task from 10 to 3. This reduces the amount of memory per task from 552 bytes to 352 bytes, which in turn reduces the task data structure impact on the memory system. Reducing the default number of declarations to 3 while keeping the task suspension threshold at 800 reduces the running time on 32 processors from 5.76 seconds to 4.98 seconds for the fi.at version. Raising the threshold to 3200 tasks (resuming when the number of outstanding tasks drops below 3100 tasks) eliminates any task suspension effects. At this threshold no task is ever suspended to avoid excess task creation, and the running time on 32 processors drops to 4.86 seconds. Our conclusion is that raising the task suspension threshold above 800 has little effect on the overall performance of the application.

We next consider locality effects. We start our analysis by presenting the task locality percentages for the versions with explicit task placement, with the locality heuristic and without the locality heuristic. As Figure 4.39 illustrates, the version with explicit task placement always executes tasks on their target processors, while the version with the locality heuristic sometimes moves tasks off their target processors in an attempt to balance

the load. The version with the locality heuristic still, however, has a much higher task locality percentage than the version without the locality heuristic.

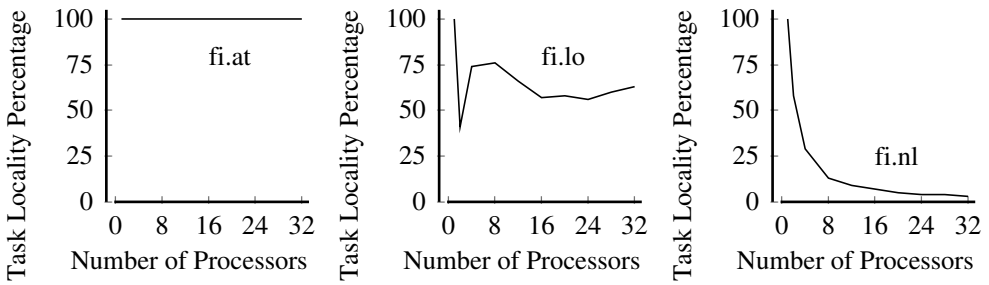

Figure 4.39: Task Locality Percentages for Panel Cholesky on DASH

On DASH, a decrease in the locality of the computation will show up as an increase in the time spent in the application code – the tasks will spend more time accessing remote data. We present the impact of the locality optimization level using graphs of the normalized application time like those in Figure 4.40. Each curve on the graph plots, as a function of the number of processors executing the computation, the total amount of time spent executing application code divided by the stripped execution time.[7] For example, a normalized application time of two means that the parallel execution spent twice as long executing application code as did the stripped version. Because the stripped and parallel versions execute the same instructions in the application code, any differences between the stripped time and the total time spent executing application code are caused only by memory system effects.

As Figure 4.40 illustrates, decreases in the task locality percentage are correlated with increases in the corresponding normalized application times, which in turn are correlated with increases in the overall execution times. We therefore attribute the differences in overall execution times between the versions that use different locality optimization levels to memory system effects that lead to differences in the amount of time spent executing application code.

[7]These graphs are generated using the timers described in Section 4.3.

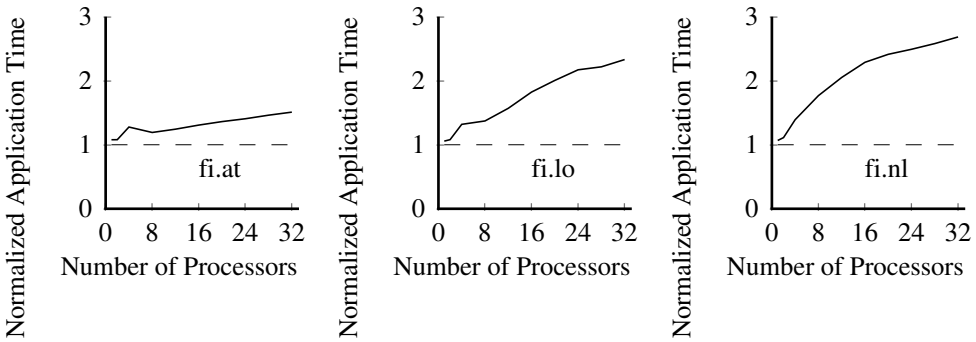

Figure 4.40: Normalized Application Time for Panel Cholesky on DASH

4.7 Ocean

The computationally intensive part of Ocean uses an iterative method to solve a set of spatial partial differential equations. It stores the state of the system in several two-dimensional arrays. On every iteration the solver recomputes each element of the array using a standard five-point stencil algorithm. The new value of the element depends on its old value and on the values of the four elements above it, below it, to the left of it and to the right of it. The solve terminates when the differences between the old values and the new values drop below a given threshold.

The programmer parallelized this application by assigning contiguous blocks of columns to different tasks. The tasks recompute the values of the elements in their blocks, concurrently writing the new values back into the blocks.

To express this computation in Jade the programmer explicitly decomposed the arrays so that the concurrent writes go to different shared objects. This decomposition is complicated by the fact that two tasks must access the boundary elements. The programmer handles this situation by decomposing the arrays along one dimension into block objects and border objects as illustrated in Figure 4.41. The decomposition takes place under the control of a granularity parameter. The number of blocks is typically the same as the number of processors executing the program minus one. The programmer uses the `create_at_object` construct to explicitly allocate each block and an adjacent border into the same memory module. Adjacent blocks are allocated in the memories associated with adjacent processors

starting at the second processor.[8]

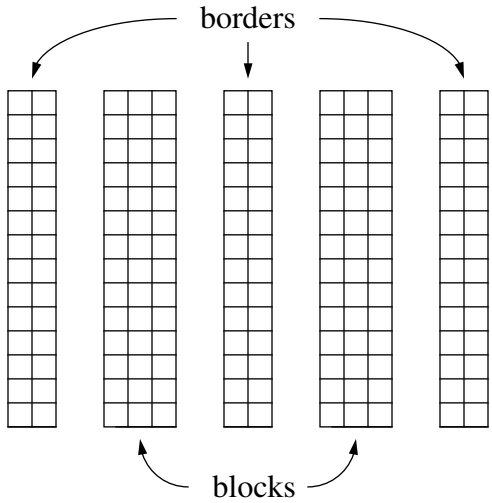

Figure 4.41: Ocean Data Decomposition

The resulting Jade computation behaves as follows. In each iteration of the solve phase, the main thread generates a new task for each block of the decomposed array. Each task processes its data from left to right, first accessing the left border, then the interior, then the right border. The tasks are created from right to left, which exposes the concurrency in the computation. For the version with explicit task placement the programmer places each task on the processor in whose memory the task's block was allocated.

The solve converges when the change in every element in the array is less than a given tolerance. Each task has its own convergence flag, which is set when the change in all of the task's elements is less than the tolerance. The main thread tests for convergence by testing each convergence flag in turn. If all of the flags are set, the main thread proceeds on the next phase of the computation. As soon as it finds a flag that is not set, it creates the set of tasks for the next iteration of the solve.

[8]Ocean has a small grain size and creates tasks sequentially. For such applications the best performance is obtained by devoting one processor to creating tasks. Allocating no blocks or boundaries on the first processor biases the locality heuristic away from executing worker tasks on the processor that creates the tasks.

4.7.1 Ocean on the iPSC/860

Table 4.13 contains the running times for Ocean on the iPSC/860. Figure 4.42 contains the corresponding speedup curves. The timing runs are for a 192 by 192 grid and omit an initialization phase. The maximum performance occurs for the mi.at version executing 12 processors with a speedup of 3.4.

	1	2	4	8	16	24	32
serial	54.19	-	-	-	-	-	-
stripped	60.99	-	-	-	-	-	-
mi.at.ab	79.46	67.20	29.32	19.01	24.10	37.62	52.80
fi.at.ab	77.84	67.20	28.98	19.57	24.21	37.06	52.37
mi.lo.ab	77.75	92.49	96.38	60.35	39.15	45.21	56.95
fi.lo.ab	77.71	93.74	95.95	57.28	39.50	44.48	55.96
mi.nl.ab	78.29	101.64	115.50	88.11	58.02	56.94	63.65
fi.nl.ab	78.03	101.33	117.17	84.92	58.76	56.98	63.94
mi.at.nb	63.56	66.20	28.85	18.85	24.35	38.07	53.20
fi.at.nb	63.09	117.22	29.10	19.26	24.67	37.44	52.82
mi.lo.nb	63.13	85.49	94.31	58.79	36.35	45.76	57.60
fi.lo.nb	63.26	88.35	86.07	57.09	39.87	44.07	56.04
mi.nl.nb	65.02	88.44	115.50	84.80	58.64	57.47	64.14
fi.nl.nb	63.67	88.05	117.35	83.65	57.64	57.10	64.09

Table 4.13: Execution Times for Ocean on the iPSC/860 (seconds)

On this platform the limiting factor is the management overhead on the main processor. We show that this is the limiting factor by measuring the amount of time that the main processor spends waiting for the right to access the convergence flags of tasks executing on other processors in the version with explicit task placement. In this version none of the tasks that actually perform useful work from the application program execute on the main processor. If the main processor spends a lot of time waiting for the right to access convergence flags, the limiting factor is the task execution time or communication time on other processors. If it spends no time waiting for the right to access convergence flags, the rest of the machine can execute tasks faster than the main processor can generate them, and the management time on the main processor is the limiting factor. Figure 4.43, which graphs the amount of time that the main processor spends waiting for the right to access

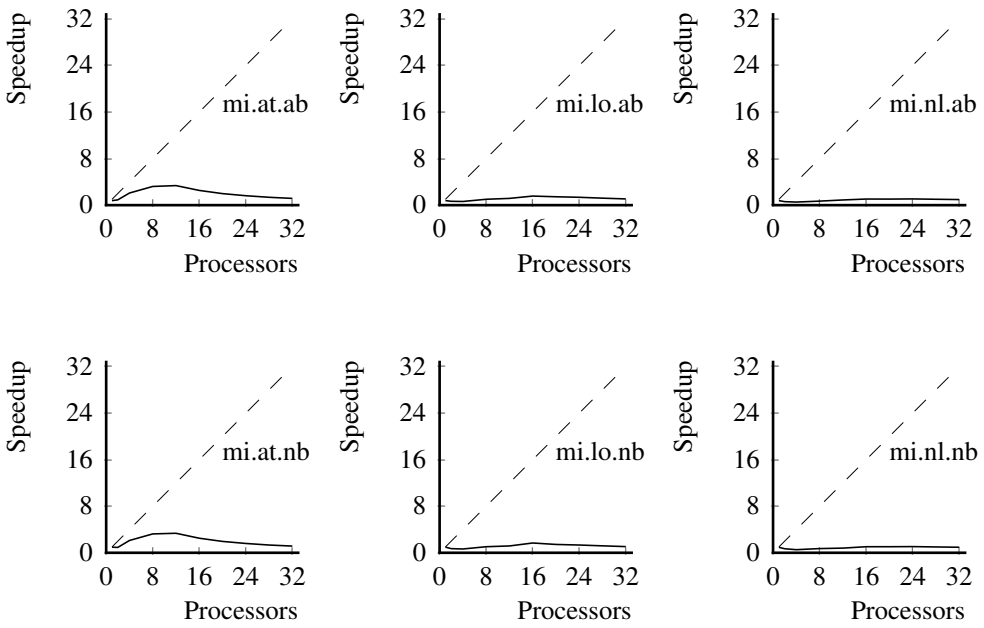

Figure 4.42: Speedups for Ocean on the iPSC/860

convergence flags as a function of the number of processors executing the application, shows that above 12 processors the main processor spends no time waiting for the right to access convergence flags.[9] The limiting factor above 12 processors is therefore the management time on the main processor.

We next consider the effect of the locality optimization level, which substantially affects the performance for small numbers of processors. The version with explicit task placement performs better than the version with the locality heuristic, which in turn performs better than the version without the locality heuristic. For larger numbers of processors the locality optimization level has less of an effect, and at 32 processors the times are almost identical for all of the versions. We first explore the locality effect by presenting, in Figure 4.44, the task locality percentages. These percentages show that the locality heuristic has trouble placing tasks on the target processor for small numbers of processors, but does much better

[9]This graph is generated by extracting the waiting time on the main processor from the event log. The waiting time for the 1 processor run is not plotted.

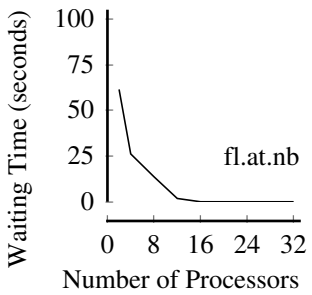

Figure 4.43: Waiting Times for Ocean on the iPSC/860

for larger numbers of processors. The graphs of the mean communication volume in Figure 4.45 show the effect of the locality optimization level on the communication. Each graph plots the sum of the sizes of all messages carrying shared objects divided by the number of processors performing the computation. The differences in the amount of communication per processor are correlated with the number of tasks executed on their target processor. The versions without explicit task placement generate significantly more communication than the version with explicit task placement.

Our analysis of the amount of time each processor spends waiting for the right to access convergence flags helps to explain the locality data. At small numbers of processors the limiting factor in the performance is how fast the tasks complete their execution. If the execution is delayed by poor locality (which forces the implementation to fetch more remote objects before executing the task), it shows up in the overall performance. At larger numbers of processors the limiting factor is the overhead at the main processor. If the execution of the child tasks is delayed by poor locality, it will have no effect on the overall performance – even if the task is delayed it will still finish before the main thread needs to access its convergence flag.

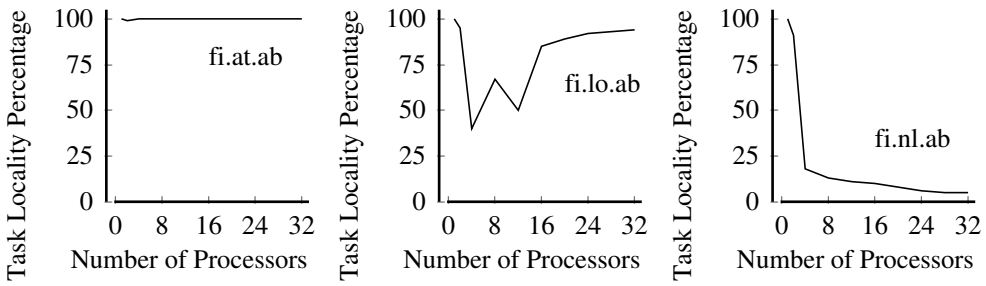

Figure 4.44: Task Locality Percentages for Ocean on the iPSC/860

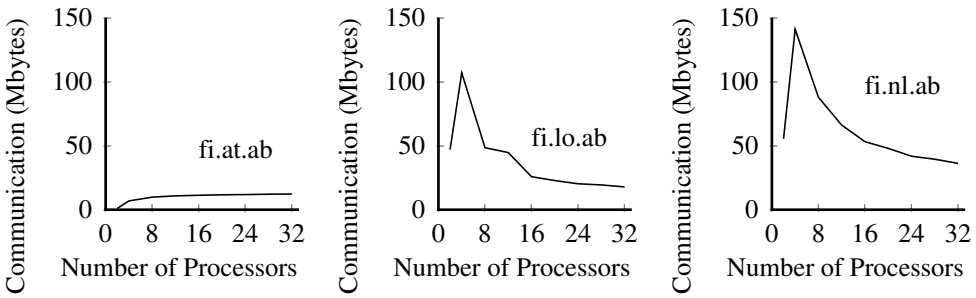

Figure 4.45: Mean Communication Volume for Ocean on the iPSC/860

4.7.2 Ocean on DASH

Table 4.14 contains the execution times for Ocean on DASH. Figure 4.46 contains the corresponding speedup curves. The maximum performance occurs in the mi.at version on 24 processors with a speedup of 12.7. We were also able to obtain performance results for an explicitly parallel version written in COOL [30]. The COOL version performs significantly better than the Jade versions for larger numbers of processors. We attribute the performance difference to serialized task management overhead in the Jade versions.

We next explore the performance of the Jade versions. Like on the iPSC/860, the limiting factor on the performance of the DASH version with explicit task placement is the task management overhead on the main processor. But since DASH supports a more efficient Jade implementation than does the iPSC/860, the DASH performance is substantially better

	1	2	4	8	16	24	32
serial	102.99	-	-	-	-	-	-
stripped	100.03	-	-	-	-	-	-
mi.at	104.27	105.44	35.98	16.21	9.18	7.88	9.87
fi.at	105.21	105.36	36.36	16.14	9.24	8.39	10.71
mi.lo	104.13	99.58	37.24	25.00	17.69	14.13	13.21
fi.lo	105.33	99.22	37.79	25.30	17.58	14.52	13.26
mi.nl	104.20	99.41	38.65	31.06	22.31	18.69	16.89
fi.nl	104.51	99.20	38.97	31.21	22.31	18.88	17.31
COOL	104.99	53.56	28.36	14.57	7.54	5.40	4.75

Table 4.14: Execution Times for Ocean on DASH (seconds)

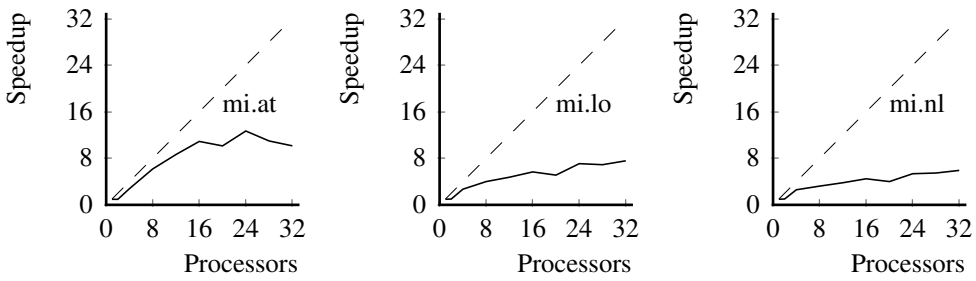

Figure 4.46: Speedups for Ocean on DASH

than the iPSC/860 performance. We can determine how the serialized task management overhead on the main processor affects the performance by computing the percentage of time that the main processor spends executing code from the Jade implementation.[10] Figure 4.47 graphs these percentages as a function of the number of processors for several versions of Ocean. More precisely, each point in this graph is the amount of time spent in the Jade implementation for a specific execution divided by the total running time for that execution times 100. The larger the percentage, the closer the task creation time comes to being the limiting factor on the performance. In fact, the serialized task management on the main processor is just barely the limiting factor for the version with explicit task placement. For the versions with and without the locality heuristic the main processor does not spend the

[10]This data comes from the timers described in Section 4.3.

entire execution time creating tasks, so the longer task execution times hurt the performance.

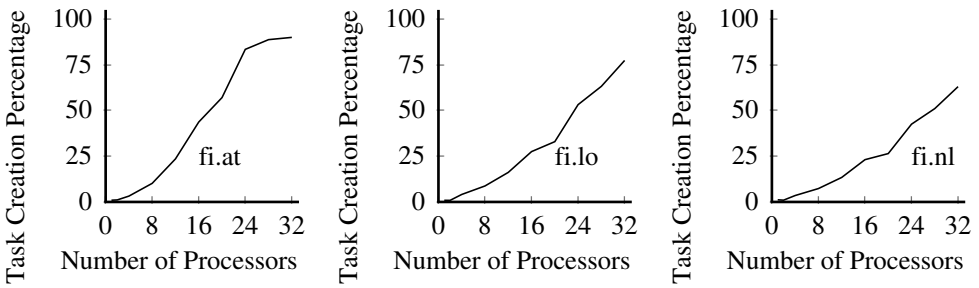

Figure 4.47: Task Creation Percentages for the Main Processor – Ocean on DASH

We next explore the effect of the locality optimization level, which has a significant impact on the performance. The versions with explicit task placement run significantly faster than the versions that use the locality heuristic, which in turn run significantly faster than the versions without the locality heuristic. The task locality percentages presented in Figure 4.48 show that the overall performance is correlated with the task locality percentages; the normalized application times presented in Figure 4.49 show that the total amount of time spent executing application code goes up dramatically as the locality decreases. We therefore attribute the differences in running times to the differing time spent in the user application due to memory system effects.

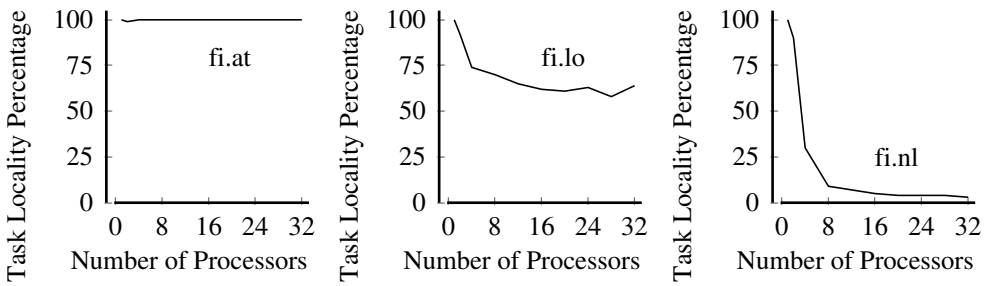

Figure 4.48: Task Locality Percentages for Ocean on DASH

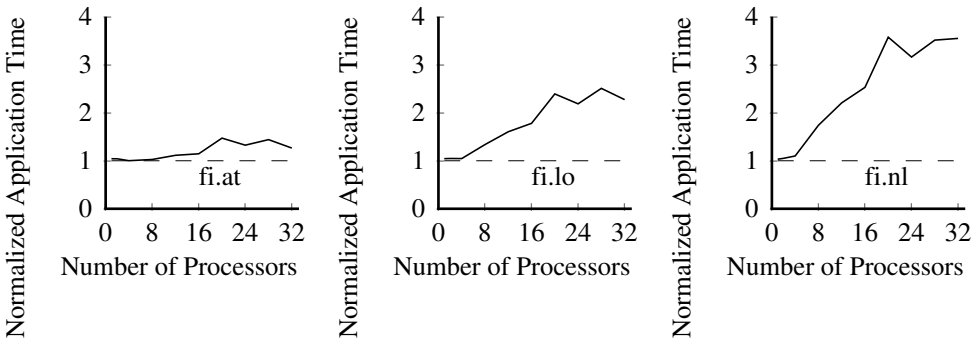

Figure 4.49: Normalized Application Time for Ocean on DASH

4.8 Programming Evaluation

Our applications experience shows that the key to developing successful Jade applications is devising and implementing an appropriate structure for the shared objects. Given such a structure, inserting the `with` and `withonly` constructs and the access specifications is a straightforward process that involves little coding effort.

In all of our applications the major data structure modifications were designed to enable concurrent writes. In Water, String and Search the programmer explicitly replicated the result data structure to enable parallel tasks to concurrently write the replicas instead of serially writing the original. In Volume Rendering each task allocated private temporary storage to hold the results of the image pixel calculation. This modification allowed the computation to replace serialized writes to the original image data structure with concurrent writes to the private data structures. In Panel Cholesky and Ocean the programmer decomposed a central data structure so that the tasks could concurrently write different parts.

Even though all of the applications substantially modified some of the original data structures, the programming overhead associated with performing the modifications varied widely from application to application. For all of the applications except Ocean the modifications were confined to small, peripheral sections of the code, and there was little programming overhead associated with the use of Jade. The key to the success of these applications was the programmer's ability to preserve the original data indexing algorithm

for the core of the computation.

For Ocean, on the other hand, the array decompositions generated wholesale changes throughout the entire program. The programmer had to change the basic indexing algorithm for almost all of the program's data and the program almost tripled in size. It is interesting that decomposition does not always generate such a large amount of programming overhead. In Panel Cholesky, for example, the decomposition of the array used to hold the factored matrix had very little effect on the overall structure of the program. Again, the key determining factor is whether or not the programmer has to change the indexing algorithm in the core of the computation.

The preceding discussion focuses on the specific properties of the parallelization visible in the final Jade version of each application. While we believe such an analysis can provide useful insight into how the language structures the program, it misses the dynamic nature of the interaction between the language, the programmer and the application during the program development process. Although we only have anecdotal data about this process, our experience using Jade leads us to believe that several aspects of Jade help programmers successfully navigate the program parallelization process.

Parallel program development proceeds most smoothly when the development can proceed via a sequence of small, incremental modifications to a working program, with the programmer checking the correctness of each modification before proceeding on to the next. Because the parallelization often requires a major restructuring of some part of the program, the programmer often reaches a stage where he must make several major modifications without being able to test any modification until all are performed. If anything goes wrong with one of the modifications it can be difficult to isolate the resulting bug because it could have been caused by any one of the multiple changes.

Jade programmers typically develop a program in two stages. In the first stage they start with a serial program that performs the desired computation, then apply the data structure modifications required for the Jade parallelization. They then insert the Jade constructs required to parallelize the program. The major modification stage, if there is one, occurs when the programmer makes the data structure modifications. It is always possible to incrementally insert the Jade constructs with no fear of changing the program's behavior. Furthermore, deterministic execution ensures that a single run completely characterizes a

program's behavior on a given input, which supports the incremental development process by making it easier to verify the correctness of each modification.

An interesting aspect of this program development process is that the potentially troublesome phase takes place before the programmer ever deals with the complication of parallel execution. The programmer can therefore use all of the existing infrastructure for the development of serial programs and can count on deterministic execution to simplify the debugging process. Our experience developing Jade applications combined with our previous experience developing explicitly parallel applications showed us that this approach can make it much easier to develop working parallel programs.

4.9 Performance Evaluation

The coarse-grain Jade applications (Water, String and Search) all perform quite well. The computations are inherently well suited for parallel execution and the use of Jade entails no significant performance penalty.

The finer-grain computations (Panel Cholesky and Ocean) have significant performance problems, especially on the iPSC/860 platform. While some of these performance problems are caused by inherent limitations of the applications and computational platforms, the Jade overhead has a substantial negative impact on the performance of both applications on both platforms. In particular, the serialized overhead on the main processor is a Jade-specific problem that limits the performance of both applications.

For Ocean an improved implementation would eliminate virtually all of the dynamic Jade overhead. Static analysis would allow the implementation to discover the basic concurrency pattern of the computation and generate code that is substantially more efficient. The resulting optimized computation would generate one large task per block for each solve phase. This task would repeatedly recompute its new values, periodically synchronizing with the tasks to its left and right. This would eliminate the serial task management bottleneck and substantially improve the performance of the computation.

Given the dynamic, data-dependent nature of the Panel Cholesky computation, however, there is very little hope for improvement from static analysis. No existing or envisioned compiler technique could automatically extract enough information from the program to

substantially reduce the dynamic Jade overhead. A maximally efficient implementation therefore seems to require the knowledge, insight and programming effort that only a skilled programmer using an explicitly parallel language can provide.

The Volume Rendering application exposed several limitations of the current Jade implementation. The most severe limitation was the fact that the message-passing implementation requires each accessed shared object to fit into the memory of the accessing processor. Because the voxel array is too large to fit in the per-node memories of the iPSC/860, the application would not run on that platform. An implementation that distributed the storage for individual objects across the memories of the machine would eliminate this limitation. Volume Rendering also illustrated how the inability to express concurrent writes to the same object can limit the performance. This inability lead to the serial copyback phase, which substantially impaired the DASH performance.

We next consider the performance impact of the various optimizations. The only communication optimization that has a major impact on all the applications is replicating data for concurrent read access. Because all of the tasks in all of the applications concurrently read at least one shared object, eliminating this optimization would serialize the computation. On the iPSC/860 the adaptive broadcast optimization enhances the performance of the Water code. For the other applications the optimization has little effect, either positive or negative.

Only two of the applications (Water and Panel Cholesky) are sensitive to locality optimizations. While the locality heuristic enhances the performance of these applications, they perform best with explicit task placement. The performance difference is not caused by the programmer applying a different locality strategy: the implementation's locality heuristic attempts to place tasks on the same processors as the programmer. For these applications the dynamic load balancing algorithm adversely affects the performance by prematurely moving tasks away from the processor that owns the locality object. This suggests that the implementation might be better off using a less aggressive load balancing algorithm.

4.10 Conclusion

Our applications experience shows that the current version of Jade is a qualified success. For all but one application the use of Jade entails limited programming overhead. The coarse-grain computations perform very well, with the dynamic Jade overhead having no significant impact on the performance. The finer-grain computations suffer from some Jade-specific performance problems, but some of these could be eliminated with a more advanced Jade implementation.

Chapter 5

Parallel Programming Systems

In this chapter we broaden our focus to discuss the design space for parallel programming systems. We structure the presentation by developing a classification of these systems, illustrating the strengths and weaknesses inherent in the different design points by discussing existing representative systems. As the discussion progresses, several recurrent themes emerge. The first theme is the fundamental design conflict between control and programmer support.

As described in Chapter 1, parallel machines present a complex execution model. A parallel programming system simplifies this execution model by enforcing a given programming paradigm. The paradigm structures the way the programmer thinks about the computation and increases the safety of the environment by eliminating certain kinds of errors. It may enhance portability by restricting the programmer to constructs implementable on a wide range of machines. Finally, it can reduce the programming burden by enabling the implementation to automatically perform parts of the parallelization process. This programming support can make it much easier to develop parallel applications.

The simplified paradigm inevitably hides some aspect of the hardware functionality. The Jade programming paradigm, for example, ostensibly hides the fact that the machine has multiple processors. Functional languages hide the fact that the machine has mutable memory. Such restrictions prevent the programmer from directly expressing computations that control the hidden aspects of the machine. The programmer must therefore rely on

the system to automatically exploit the hidden functionality. But the general-purpose algorithms characteristic of automatic systems can be much less efficient than the highly optimized, application-specific algorithms that motivated programmers generate. The supportive programming paradigm can quickly become a straightjacket if the needs of the application do not match the capabilities of the system. Many of the distinctions in our classification separate systems that take different positions on the division of responsibility between the programmer and the system.

A second recurring theme is the diverse set of implementation approaches that systems can apply to a given programming model. For example, it is possible to parallelize a serial program statically using a parallelizing compiler, dynamically using a system like Jade, or even optimistically using a speculative system like TimeWarp [70, 69]. The different approaches often reflect perturbations back into the programming model as the systems either impose additional restrictions that enable specific implementation techniques or require the programmer to help by providing extra information about the program. The finer distinctions in our classification often separate systems that implement the same basic programming model in different ways.

Before we present our classification, we should inform the reader about the scope of our discussion. There are many reasons to include parallelism in a programming model. For example, the programmer may wish to execute parts of a program concurrently for performance, manage the interaction of autonomous components in a physically distributed system, or express an event-driven computation in a language with explicit support for multiple agents that interact via events. These different motivations generate radically different design trade-offs. An analysis of how well a system works for one purpose usually has very little relevance for other purposes. In this chapter we only discuss parallel programming systems that are designed to exploit concurrency for performance, and only analyze the systems from this perspective.

5.1 The Major Groups

We classify parallel programming systems into three major groups: systems with serial semantics, monotonic systems and explicitly parallel systems. Systems with serial semantics start with a program written in a serial language. Parallel execution comes either from executing independent parts of the program concurrently or from within the basic operations of the language. This group includes serial languages such as Fortran and C implemented by a parallelizing compiler, Fortran D [42], languages such as Jade and FX-87 [87, 47] that provide information about how a serial program accesses data, data-parallel languages like C* [111, 112] and Fortran 90 [92], and speculative systems like Time Warp [70, 69] and ParaTran [132].

The monotonic group consists of systems whose basic computational model is the monotonic accumulation of information over the course of the computation. Each operation may wait for some information to become available, then generate additional information. Id [101], functional languages such as Haskell [67] and Sisal [38] and concurrent logic programming languages such as Strand [40] and Parlog [50] are in this group.

The explicitly parallel group includes systems in which the program explicitly generates parallel tasks. The tasks then synchronize and communicate using mechanisms such as locks, barriers, message-passing operations and/or shared memory. Packages like PVM [130], Presto [17] and the ANL macro package [88] focus on providing very basic, low-level primitives used to create and coordinate parallel execution. Languages like Occam [86, 27], CSP [64, 63], CML [106] and Modula-3 [98] integrate concurrency generation and synchronization primitives into the language. Concurrent object-oriented languages like COOL [30], Orca [9], POOL-T [5] and ABCL/1 [136] hide such primitives behind higher-level constructs.

5.2 Serial Semantics

Much of the complexity of parallel execution comes from the fact that parallel tasks can generate many different interleavings of the basic operations, with each interleaving generating a potentially different behavior. To understand the program, one must take all of

the different behaviors into account. Expressing the computation in a serial language, however, imposes a total order on the operations and the program only generates one behavior. From the programmer's perspective this dramatic simplification makes it much easier to reason about what the program may do. From the system's perspective it imposes the often considerable challenge of automatically generating parallel execution from a serial program.

We divide the serial semantics group into two major groups: the data-parallel group and the imperative group. The data-parallel group contains systems that execute programs written in data-parallel languages. The imperative group contains systems that automatically parallelize programs written in standard imperative languages such as Fortran or C. The system may support language extensions that programmers use to guide the parallelization process.

Within the imperative group we classify systems based on their implementation approaches. The first division separates this group into the speculative group and the conservative group. Speculative systems optimistically generate parallel execution, preserving the serial semantics by undoing the effect of prematurely executed tasks. Conservative systems, on the other hand, delay each task's execution until they know the execution will violate no dependences. The conservative group includes both static and dynamic systems. Static systems analyze the program before it runs and generate a parallel program that exploits the concurrency available at compile time. Dynamic systems analyze the program at run time and exploit concurrency as the program runs.

5.2.1 Data-Parallel Systems

Data-parallel languages such as Fortran 90 [92] and C* [111] provide a useful paradigm for programs with regular, data-parallel forms of concurrency. Programmers using these languages view their program as a sequence of operations on large aggregate data structures such as sets or arrays. The system can execute each aggregate operation in parallel by performing the operation on the individual elements concurrently. This approach preserves the advantages of the sequential programming paradigm while exposing the concurrency available within operations.

5.2.2 Static Systems

Static systems have been built for a wide range of languages. The most ambitious systems are parallelizing compilers, which attempt to automatically parallelize programs written in standard serial languages. More pragmatic systems expect the programmer to guide parts of the parallelization process. The programmer may help the system analyze the program by providing information about how the program structures and accesses data, or make policy decisions about the distribution of tasks and data to processors and memories.

5.2.2.1 Parallelizing Compilers

Parallelizing compilers [10, 78, 79, 4] statically analyze programs to find independent pieces of code. The compiler then generates a parallel program that executes the independent pieces of code concurrently. In message-passing environments the compiler also maps the data onto the processors and generates the communication operations required to transfer remote data to accessing processors.

Automatically parallelizing serial programs is an extremely challenging task [21]. Despite years of research in this area, several important problems remain to be solved before it is practical to use a compiler to automatically exploit the concurrency in parallelizable computations. For regular applications the fundamental problem is performing the global analysis required to generate tasks large enough to amortize the concurrency exploitation overhead. Experience manually applying automatable techniques across multiple procedure boundaries suggests that it is possible for a compiler to successfully parallelize the computation if it increases its scope to include large sections of the program [53, 21].

For irregular programs there are problems performing dependence analysis that is precise enough to expose the concurrency. While compilers can often successfully analyze the data access patterns of loop nests that manipulate dense matrices [91, 90], automatically analyzing the access patterns of programs that manipulate dynamically linked data structures or use indirect array indices remains an open research problem.

Another unsolved problem is determining how to coordinate the mapping of data and computation to memory modules and processors [6]. For good performance, the compiler

must evenly balance the computational load and avoid uncoordinated mappings that generate excessive communication as processors repeatedly fetch remote data. The problem becomes especially complex when different parts of the program access data in different ways. Fortran D [42] and High Performance Fortan [60] allow programmers to guide the mapping process by providing constructs that specify how to distribute arrays across multiple memory modules. The compiler can then use the data distribution to generate a mapping of computation to processors via the owner computes rule.

Finally, a fundamental limitation on parallelizing compilers is that there may not be enough information available at compile time to generate parallel code. If, for example, the concurrency depends on the input data the compiler will be unable to statically parallelize the program.

5.2.2.2 Data Usage Extensions

Researchers have also developed languages that allow programmers to provide extra information about how the program structures and accesses data. The goal is to expose more concurrency by improving the precision of the compiler's dependence analysis.

Refined C [76] and Refined Fortran [75, 36] allow programmers to create sets of variables that refer to disjoint regions of memory. When pieces of code access disjoint subsets of such variables, the compiler can statically verify that they can execute concurrently. Typical operations are creating a set of names that refer to disjoint regions of an array and creating an array of pointers that point to distinct data structures.

ADDS [59] declarations for data structures containing pointers to dynamically allocated data allow programmers to describe the set of data structures that can be reached by following different pointer chains. The compiler combines this information with an analysis of the pointer-chain paths that different parts of the computation follow to derive a precise estimate of how the computation will access data. The improved precision of the dependence analysis can expose additional opportunities for parallel execution.

FX-87 [87, 47] contains constructs that programmers use to specify how procedures access data. The system statically analyzes the program, using this information to determine which procedure calls can execute concurrently without violating the serial semantics. FX-87 programmers partition the program's data into a finite, statically determined set

of regions. The access specification and concurrency detection take place statically at the granularity of regions. The fact that regions are a static concept allows the FX-87 implementation to check the correctness of the access specifications at compile time. But regions also limit the precision of the data usage information. In general, many dynamic objects must be mapped to the same region, preventing the programmer from expressing concurrency available between parts of the program that access disjoint sets of such objects. An analysis of the concurrency in FX-87 programs [56] found that they failed to exploit important sources of concurrency because they mapped many pieces of data to the same region. The problem was especially severe for programs in which the main potential source of concurrency scaled with the size of the input problem. Because the number of regions was determined by the programs, not the input data, they could not exploit this source of concurrency.

5.2.3 Dynamic Systems

Unlike the static systems described above, dynamic systems analyze the program as it runs, taking into account the specific values that occur in a given execution. Because dynamic systems have more information about the computation than static systems, they can often discover more concurrency. The trade-off is that dynamic analysis generates overhead, which may degrade the performance.

5.2.3.1 Partial Evaluation

A branch of compiler research has attempted to extend techniques from parallelizing compilers to programs with dynamic, data-dependent concurrency patterns [94, 117, 116, 115, 84]. These compilers use a two-phase partial evaluation approach. The first phase partially evaluates part of the program on its input data, then generates and schedules the resulting task graph and communication operations. The second phase actually executes the computation. In the best case the partial evaluation overhead can be amortized over many identically structured instances of a given computation. While this approach can often exploit sources of concurrency unavailable to traditional parallelizing compilers, it has several fundamental limitations. The partial evaluation overhead limits the available concurrency because

the partial evaluation phase must complete before any tasks execute. This overhead can become especially severe if the partial evaluation phase must perform a large part of the total computation to determine the task graph. Another limitation is that the computation may generate a task graph too large to represent explicitly, in which case it is impossible to parallelize the program using this technique.

5.2.3.2 Online Systems

Online systems, such as the current Jade implementation, overlap the generation, analysis and execution of parallel tasks. This overlap allows programs to effectively exploit forms of concurrency in which the structure of part of the computation depends on data generated in previous tasks. Online systems can also regulate the amount of storage devoted to outstanding tasks. If a program starts to consume an excessive amount of task storage, the implementation can temporarily suspend the part of the computation responsible for the excessive task generation. Section 3.5.3 describes the algorithm that the current Jade implementation uses.

Online systems relax an artificial limit that partial evaluation places on the exploited concurrency. There is no need to complete a partial evaluation phase before starting the execution of tasks from the computation phase. Even in online systems, however, the need to discover how the computation accesses data can limit the amount of exploited concurrency. Before an online system executes a task it dynamically analyzes the preceding computation to ensure that the task's execution will violate no dependence. The time required to perform this analysis unnecessarily delays the execution if there was, in fact, no dependence.

There is a final problem having to do with the precision of the data access information. Even at run time the system may be unable to determine exactly which pieces of data each task will access without actually executing the task. When the system performs the analysis it must therefore use a conservative approximation of the task's actual accesses. This lack of precision can force the system to unnecessarily delay a task's execution because of potential dependences that fail to materialize in the actual computation.

5.2.4 Speculative Systems

Speculative systems optimistically assume that tasks can execute concurrently, then use rollback mechanisms to undo the effect of tasks that violate dependence constraints. In effect, the dependences are discovered only as the tasks execute, and active enforcement of the dependences via rollback occurs only when a violation has been detected. Both the Time Warp system for distributed simulation [70, 69] and the ParaTran system for parallelizing serial Scheme code [132] use a speculative approach.

Speculative systems can exploit concurrency available in programs that conservative approaches can not effectively analyze. Speculative systems have no need to analyze the preceding computation before executing a task, and can execute tasks concurrently even if they have potential dependences. If most of the dependences fail to materialize in the actual computation, the speculative approach successfully parallelizes the computation. The major disadvantage of this approach is the overhead required to detect dependence violations, to maintain enough information to undo the effect of prematurely executed tasks and to actually perform the rollbacks. The rollback overhead can become especially severe if the parallel execution frequently violates the precedence constraints.

5.2.5 Discussion

Systems with serial semantics preserve many of the substantial programming advantages of serial programming languages. They present a simple, familiar programming model based on the abstraction of a single thread of control. Unlike programmers using explicitly parallel systems, programmers that use these systems neither generate concurrency management code nor struggle with complex phenomena such as deadlock or transient timing-dependent bugs.

Part of the price for this supportive programming environment is paid in efficiency. To safely generate parallel execution, the system must know how the different parts of the program access data. There are three ways to discover this information, each of which can impose substantial performance loss.

- **Static Analysis** The problem with static analysis is that there may not be enough information available at compile time to determine if is possible to parallelize the

program. If there is any uncertainty the system must conservatively generate serial code. This serialization wastes concurrency when the computation could actually execute in parallel.

- **Dynamic Analysis** Analyzing the program dynamically can dramatically improve the precision of the data usage information and enable the exploitation of much more concurrency. But dynamic analysis also generates overhead, which can artificially limit the performance of the resulting parallel computation.

- **Speculation** The system can optimistically assume that tasks can execute concurrently. The problem with this approach is the dynamic overhead required to detect and roll back illegal parallel executions.

For a given application a skilled programmer may be able to develop special-purpose synchronization and concurrency generation algorithms that alleviate the performance problems. But the system cannot give the programmer the control required to express such algorithms without destroying the ostensibly serial programming model and forfeiting all of the advantages that go with it. This situation illustrates the fundamental conflict between providing a safe programming environment and allowing the programmer to control the parallel computation at a low level for efficiency.

A final issue is that some parallel algorithms adjust their behavior to the relative execution times of different parts of the program. For example, the region of the search space explored by a parallel branch-and-bound algorithm depends on how fast the tasks improve the bound. As the precise timing of the bound improvement varies from run to run, the program explores different regions of the search space. Programmers cannot generate such computations using systems with serial semantics because it is impossible to express the algorithms in a serial language.

5.3 Monotonic Systems

Systems with serial semantics provide mutable data but simplify the programming model by eliminating the concept of parallel execution. Monotonic systems, on the other hand,

provide parallel execution but simplify the programming model by eliminating the concept of mutable data. More specifically, monotonic systems are built on the abstraction of information. Computation consists of reading information (suspending if it has yet to be generated) and generating additional information. Information therefore monotonically accumulates over the course of the computation. Each system has its own information domain; computations in each system derive part of their meaning from the underlying domain.

In monotonic systems a variable's value does not change over the course of the computation – it only becomes more defined. Computation in monotonic languages is therefore inherently parallelizable. All operations can execute concurrently without conflict. The only required synchronization is implicit in the way the operations read information. The computation proceeds in an information-driven fashion. When one operation generates information that satisfies a second operation's information request, the second operation executes, generating further information or computation. We next present a brief summary of several different monotonic languages, then summarize the strengths and weaknesses of the monotonic approach to parallel computation.

5.3.1 Functional Languages

Programmers using functional languages structure their computation as a set of recursively defined functions. Variables hold values returned from function invocations and are used to transfer values between invocations. Because each variable holds only one value during the course of a computation, the execution of a functional program can be seen as the progressive generation of information about the values of variables.

5.3.2 Id

One important expressiveness restriction associated with functional languages is that the value of each variable is generated at a single point in time and space: the function invocation that determines its value. It is often convenient to generate the individual values that together comprise a large aggregate at different points in the program's execution. Id [101] provides for this functionality via I-structures [8, 102]. I-structures contain write-once elements that

are initially undefined. The program can separately define each element; when an operation attempts to use an undefined element it suspends until it is defined. It is an error to attempt to define an element twice.

5.3.3 Concurrent Logic Programming Languages

Concurrent logic programming languages such as Strand [40] and Parlog [50] and their generalization to the family of concurrent constraint-based programming languages [120, 119] present a model of computation based on constraints. The computation consists of a set of parallel agents that incrementally impose constraints on the values of the variables. Agents can also create new parallel agents, suspend until a variable's value satisfies a given constraint and choose between different behaviors based on the imposed constraints.

5.3.4 Discussion

The strengths of monotonic systems are their clean semantics and inherently parallel execution model. They allow programmers to expose the concurrency in their computations without having to deal with the complexity of an explicitly parallel programming system. A major barrier to the widespread acceptance of monotonic languages, however, is the difficulty of implementing these languages efficiently. The main sources of overhead are scheduling, memory management and excessive copying. These sources of overhead impose an ever-present performance penalty on the basic form of execution. We discuss these performance problems in Sections 5.3.4.1, 5.3.4.2 and 5.3.4.3.

There are also expressiveness problems associated with monotonic languages. As described in [11], the lack of mutable data can force programmers to tediously thread state through multiple layers of function calls. Updating a single variable can force the explicit regeneration of large linked data structures. The expressiveness and efficiency problems have led to hybrid designs which integrate mutable data into monotonic programming languages. For example, the designers of Id augmented the language with M-structures (see Section 5.4.2.2). PCN and Compositional C++ (see Section 5.3.5) allow the program to use mutable data in restricted contexts.

5.3.4.1 Scheduling and Partitioning

Monotonic languages typically expose concurrency at the level of the individual operations in the language. The scheduling overhead associated with exploiting concurrency at such a fine-grain level has inspired compiler efforts to partition the operations into larger sequential tasks [134, 121, 122, 123]. For good performance the partition should generate sufficient concurrency, minimize communication and successfully amortize the scheduling overhead.

Programs written in lazy functional languages, Id and concurrent constraint languages generate an unordered set of operations that become enabled and execute in an information-driven way. The lack of a statically derivable sequential execution order complicates the process of generating tasks large enough to profitably amortize the scheduling overhead. Research performed in the context of Id [125, 35, 34] attacks this problem with a two-level scheduling strategy. The compiler first statically partitions the program into threads. The run-time system then dynamically schedules related threads into larger-grain units called quanta. Because executing related threads sequentially is more efficient than interleaving related and unrelated threads, this strategy drives down the scheduling overhead.

A standard depth-first evaluation strategy will correctly execute any program written in an eager functional language. It is therefore easy for the partitioning algorithm to generate tasks large enough to amortize any scheduling overhead. But if the partitioner is not careful, it may generate tasks that are so large that the resulting partition fails to expose enough concurrency. The partitioning algorithm must therefore negotiate a trade-off between concurrency and scheduling overhead. Most of the work in this area has taken place in the context of the eager functional language Sisal. In particular, Sarkar has developed a partitioning algorithm for Sisal that negotiates the resulting trade-off between concurrency and scheduling overhead [121].

MultiLisp [54, 55] relies on the programmer to specify a partitioning. The MultiLisp future construct allows the programmer to explicitly specify the task granularity by declaring that a given function invocation should be evaluated concurrently with the computation after the function. The return value is undefined, and subsequent uses of the return value suspend until the function completes and defines the value. In the absence of futures the program executes sequentially.

5.3.4.2 Memory Management

The need to manage memory comes from the fact that programs typically generate many more values than the memory of the machine can hold. The computation must therefore reuse memory to execute successfully. Programs written in languages that support mutable data reuse memory efficiently by overwriting old values with new values. Because monotonic languages eliminate the notion of side effects, the implementation is responsible for generating the memory reuse.

Garbage collection is one common technique for reusing memory. The lack of side effects in monotonic languages means that each piece of memory goes through the collector between writes. Many of the problems associated with garbage collection are the same for parallel monotonic languages and mostly functional languages such as ML [93, 57]. One potential performance problem is the instruction overhead of garbage collection. A more serious problem, however, is the interaction with the memory hierarchy. Because the computation must store each generated value into a new memory location, it consumes memory very quickly. Because the volume of data accessed between collections is typically larger than the processor cache, the program exhibits poor locality. In fact, the memory system performance can be so bad that paging, and not poor processor cache locality, is the dominant memory system effect [32]. We discuss other strategies for eliminating memory management overhead in Section 5.3.4.3.

5.3.4.3 The Copy Problem

The lack of mutable data means that programs written in monotonic languages typically generate more copying overhead than programs written in more conventional languages. To update a component of a data structure, the program cannot simply generate the new value and store it back into the original structure. It must instead allocate a new structure to hold the value and copy the unchanged values from the old structure to the new structure. If the structure is part of a large linked data structure the program must recursively regenerate all of the data that pointed to the original (and now obsolete) structure.

The copy problem has attracted the most attention in the context of functional languages with arrays. In many of these languages the basic array manipulation primitive takes an

old array and produces a copy of the array with one element replaced with a specified new element. For large arrays the use of such an operation can generate ruinous copying overhead. The prospect of incurring such overhead has inspired the development of many techniques for eliminating the copy.

If the original array is not used in the subsequent computation, the system can update the array in place instead of generating a copy. It is possible to identify the last use of each array in the computation by compile-time analysis [49], reference counting [46], a combination of compile-time analysis and reference counting [66], or by language-level constructs that ensure that every use of a given array is a last use [51]. Such optimizations also reduce the memory management overhead since they reuse memory without the trip through the garbage collector.

One interesting aspect of update-in-place optimizations is that their use reintroduces the key problem of parallelizing programs that use mutable data: correctly sequencing reads with respect to writes in the parallel execution. Systems that use this optimization can no longer blindly evaluate expressions in parallel. They must analyze the generated access patterns and insert synchronization operations that correctly sequence the evaluation of expressions that access the updated data structure.

5.3.5 Mutable Data in a Monotonic Context

The efficiency and expressiveness problems associated with a monotonic model of computation prompted the designers of Strand to support a multilingual model of computation. In this model programmers use Strand to introduce concurrency and provide synchronization and communication; languages like C and Fortran are used for the actual pieces of sequential computation. This approach has the advantage that the vast majority of the computation is performed in a language that suffers from none of the expressiveness or efficiency problems associated with monotonic languages. In many cases it also allows programmers to parallelize existing sequential applications without modifying large sections of code.

The multilingual approach works best when it is natural to structure each task as a functional module that accepts a set of inputs and generates a set of outputs. This restricted model of computation meshes well with Strand's monotonic paradigm. Because

the sequential tasks completely encapsulate their use of mutable data, Strand programmers can reason about their behavior without leaving the basic Strand computational model.

The approach seems less appropriate, however, when the sequential computations use mutable memory in an externally visible way – when, for example, two tasks access the same global variables. In this case there is a fundamental mismatch between the two models of computation. The Strand programmer is put in the position of synchronizing multiple reads and writes to mutable data with operations designed to coordinate the production and use of information in a monotonic context.

PCN [41, 39] and Compositional C++ [31] more fully integrate mutable data into the monotonic model of parallel computation. These languages provide both sequential and parallel composition operators and mutable and definitional (write once) data. They eliminate the complexity of correctly synchronizing multiple concurrent reads and writes to mutable data by imposing the restriction that parallel tasks interact only via definitional data. It is illegal for one parallel task to write a piece of mutable data and another parallel task to access the same piece of data, although this restriction is not currently enforced. A correct PCN or Compositional C++ program uses the sequential composition operator (in other words, a barrier) to serialize the executions of such tasks. From the perspective of parallel execution it is then possible to view each task as monotonically producing information, although it may use mutable data to do so.

While PCN and Compositional C++ integrate mutable data into a fundamentally monotonic language without compromising the basic model of computation, the barrier synchronization mechanism that enables this integration limits the range of programs that can effectively use mutable data. Because it is impossible to implement irregular concurrency patterns with barriers, the tasks in programs with irregular concurrency patterns cannot interact using mutable memory.

5.4 Explicitly Parallel Systems

Systems with serial semantics and monotonic systems attempt to eliminate the complexity of parallel programming by imposing radical restrictions on the programming model. Explicitly parallel systems, on the other hand, accept the basic complexity of an execution

model that includes parallel execution and mutable data. The provided constructs allow programmers to create parallel tasks and control their interaction. While systems may impose restrictions for portability or to structure the programming model, the basic thrust is to let the programmer, rather than the system, manage the parallel execution.

There are two broad categories of explicitly parallel systems: message-passing systems and shared-memory systems. Tasks in message-passing systems interact by sending and receiving messages. To send a message a task composes a message containing the appropriate data and invokes a send construct, which transfers the data to a specified receiver task. When the receiver task executes a receive construct the system transfers the data into the specified variables.

Shared memory systems separate the notions of communication and synchronization. The tasks communicate implicitly by reading and writing the shared memory. They synchronize using constructs such as locks, barriers and condition variables. Some systems provide a higher-level but less flexible synchronization interface by augmenting the semantics of data access to implicitly include synchronization.

5.4.1 Message-Passing Systems

Message-passing systems have been based on both synchronous and asynchronous send constructs. Asynchronous sends return immediately, with the system invisibly buffering the data until the corresponding receive comes along. Actor languages [1, 2, 3], PVM [130] and the NX/2 system from Intel [104] are based on asynchronous sends. Synchronous sends block until the corresponding receive executes and the data transfer is complete. CML [106], Occam [86, 27], CSP [64, 63] and Ada [97] are based on synchronous message-passing constructs.

An apparent weakness of synchronous message passing is the imposed serialization. Blocking until the message is received eliminates the possibility of overlapping computation with communication at the sending task and can impose long delays if the receiver executes the corresponding receive construct long after the sender sends the message. Systems like CML, however, reduce the impact of this serialization by allowing the programmer to create lightweight threads. The program can create a new thread to perform the blocking send

while the main thread continues with the computation.

The message-passing paradigm works best for programs in which each process can predict when every other process will send it a message. In this case the processes invoke a blocking receive construct whenever they expect a message, and the computation proceeds in an orderly, message-driven manner. But the paradigm starts to break down when tasks need to receive and process messages asynchronously with respect to an ongoing main computation. This can happen, for example, if a task encapsulates a shared data structure that other tasks access in an unpredictable way.

One option is to periodically poll for message arrival. The drawback is that inserting the polling calls imposes programming overhead and executing the polling operations can degrade the performance. Another option is to set up an interrupt message handler that gets invoked whenever a message arrives. The `hrecv` construct in the NX/2 message-passing package supports this functionality [104]. The drawback is the programming overhead required to synchronize the main computation with the message handlers. The synchronization is typically performed at a low level by periodically disabling and re-enabling the message arrival interrupt. A final alternative is to create a lightweight thread which immediately invokes a receive construct and blocks waiting for a message to arrive. When the message arrives the thread runs, processes the message, then blocks waiting for another message. The drawback is again the programming overhead required to synchronize the message processing code with the main computation. In this case the programmer performs the synchronization using shared-memory synchronization primitives designed to manage the interaction of multiple parallel threads.

5.4.1.1 Programming Implications

Message passing systems can impose substantial programming overhead. The application must explicitly manage the movement and replication of data. Every time a task accesses remotely produced data, the producer and consumer must explicitly interact to generate the data transfer. Either the producer must know all of the potential consumers and deliver the data eagerly to them upon production, or the programmer must develop naming algorithms that allow consumers to find the producer of each piece of data. The program must also manage the memory used to hold remotely generated data. Requiring the programmer

to generate code to perform these activities complicates the programming process. A comparison of a fairly complex application written for both a message-passing system [114] and a shared-memory system [126, 89] highlights the programming burden that message-passing systems can impose.

One clear advantage of message-passing systems is that they present a simple performance model. Every remote interaction is cleanly identified in the program and a simple model accurately predicts the cost of each message transfer [16]. It is possible to understand the performance of the remaining serial code using a standard uniprocessor performance model. This simple model makes it much easier to tune the performance of a parallel computation. It helps the programmer to both understand the current performance and to predict the performance impact of contemplated modifications.

5.4.1.2 Performance Implications

Superficially, the basic abstractions in asynchronous message-passing systems seem to match the low-level communication mechanism of the hardware. These systems would therefore appear to maximize the amount of control the programmer has over the communication, which in turn would maximize the achievable performance. But in practice message-passing systems have imposed unnecessary overheads at both the hardware and software levels. Eliminating these overheads requires a careful redesign of the interface that the message-passing package exports to the programmer. We next analyze the sources of these overheads, then discuss recent research that addresses some of the problems.

In traditional message-passing systems there is a protection boundary between the application and the network. The overhead required to cross this boundary imposes system call and interrupt overhead. When a program sends a message it must perform a system call to invoke the operating system code that manages the network interface. When the message arrives at the receiver it generates an interrupt to invoke the operating system code that takes the message out of the network.

There is also software overhead at the receiver. The receiver must determine which task should get the message and route the message to that task. If the task has yet to post the correspond receive the operating system must allocate buffers to temporarily store the message. When the task finally executes the receive the system copies the message out of

the buffers into the task's address space.

The first step towards efficient message passing is a hardware redesign to eliminate the need for a protection boundary between the sender and the network. The goal of such a redesign is to eliminate the system call overhead associated with sending a message. The main issue is supporting direct user-level access to the network without compromising security. No program should be able to use its network access to interfere with other programs. The Thinking Machines CM-5 [103] provides this functionality by imposing a strict gang scheduling regime. Different programs cannot interfere because they cannot concurrently access the network. More recent systems [65, 22] perform protection checks in hardware on an attached co-processor, which enables the operating system to schedule each compute processor independently.

The next step is a redesign of the interface that the message-passing system presents to the user program. A fundamental problem with current systems is that the sender does not specify where in the receiver to store the message. The implementation must therefore temporarily buffer messages, which generates copying and memory management overhead. The solution is to provide an interface based on remote reads and writes rather than on sending and receiving messages. Such an interface eliminates buffering by telling the system the final location of the transferred data before the system actually starts the transfer.

The final advantage of a remote read and write interface is that it eliminates unnecessary interrupts at the remote compute processor. With minimal hardware support [65] it is possible to perform all remote operations on an agile co-processor designed to quickly handle remote reads and writes. This approach eliminates the substantial overhead associated with interrupting the compute processor, leaving it free to continue with its main thread of computation.

5.4.2 Shared-Memory Systems

Shared-memory systems eliminate the explicit data management burden of message-passing systems. Since all communication takes place implicitly via the shared memory, the provided constructs focus on concurrency generation and synchronization. We first consider

how different shared-memory systems structure the programming model. In order of increasing structure, we first discuss basic synchronization packages, then systems that combine synchronization with data access and finally concurrent object-oriented programming languages, which augment the object-oriented programming paradigm to include concurrency and synchronization.

We then discuss the design of systems that implement the abstraction of shared memory in software on message-passing machines. A key theme that structures this discussion is how performance concerns from the underlying message-passing substrate have influenced the interface these systems present to the programmer. The resulting systems minimize the number of messages required for synchronized access to blocks of shared data and expose the inherent asynchrony of the communication in the programming model.

5.4.2.1 Basic Synchronization Packages

Packages such as Presto [17] and the ANL macro package [88] and languages such as Modula-3 [98] rely on the hardware to implement the shared memory. The software only provides basic concurrency generation and synchronization primitives such as locks, barriers and condition variables. Such a bare-bones interface minimizes the safety of the programming model. In shared-memory systems, synchronization operations usually mediate access to shared data structures. The systems discussed in this section provide no support for the resulting association of synchronization and data. The programmer is completely responsible for developing the association and explicitly inserting synchronization operations around each access to shared data. As discussed in succeeding sections, higher-level systems promote a safer programming model by supporting the association of synchronization with data.

The low-level nature of the programming model pays off in efficiency and flexibility, maximizing the amount of control the programmer has over the parallel execution. The programmer can dynamically change the synchronization algorithm for a given data structure to adapt to the different access patterns of different parts of the computation. If the machine supports a particularly efficient synchronization algorithm, these systems do not interfere with the programmer's ability to express that algorithm. It is therefore possible to tightly control the mapping of the application onto the machine for efficiency.

A final advantage of low-level systems is that they minimally perturb the hardware performance model. Because the provided constructs translate directly into simple hardware operations, the software imposes no mysterious sources of overhead. The resulting performance model supports the process of tuning a computation by making it easier to identify expensive operations and to understand the performance impact of potential modifications.

5.4.2.2 Combining Synchronization and Data Access

Several systems provide a higher-level programming model by augmenting the semantics of data access to include synchronization. Id's M-structures [11] allow programmers to build data structures that are implicitly synchronized at the level of the individual words of memory. An M-structure is a word of memory augmented with a bit indicating if the word is empty or full. The program reads an M-structure with the take operation. This operation waits until the word is full, then reads the contents out, leaving the word empty. The program writes an M-structure with the put operation. This operation writes a new value into the word, leaving it full. M-structures support the mutual exclusion constraints associated with atomically updating a piece of data.

Linda [44, 28] provides the abstraction of a shared memory consisting of a collection of tuples. The program can insert, read or remove tuples from the tuple space. The read and remove operations are associative in that the program provides a partial specification of the desired tuple and the system reads or removes a tuple that matches the specification. The synchronization is associated with tuple access; the basic mechanism is that the associative read and remove operations block until a matching tuple is available. The program can implement mutual exclusion constraints by inserting and removing a single control tuple; it is possible to implement precedence constraints by inserting a single tuple when the constraint is satisfied. Typically, the tuple used to implement the synchronization also carries the associated data.

Systems that combine synchronization and data access pay synchronization overhead every time they access shared data. This overhead can become especially severe when a single synchronization operation is all that is required to synchronize many accesses to shared data. It is therefore important to use a simple synchronization mechanism that does not impose excessive overhead. General purpose implementations of Linda's associative

tuple space operations have been inefficient. Linda compilers therefore use global static analysis that attempts to recognize common higher-level idioms. They then implement these idioms using specialized, more efficient algorithms tailored for the specific situation at hand. This approach leads to a model of parallel programming in which the programmer encodes the high-level concepts in low-level primitives only to have the compiler attempt to reconstruct the high-level structure.

In certain circumstances combining synchronization and data access can also eliminate synchronization overhead. When the program accesses remote data the system can generate a single remote operation that combines the synchronization and data transfer operation. Standard shared-memory systems, on the other hand, generate two remote operations for each synchronized access to remote data: a synchronization operation to acquire the right to access the data, then a communication operation to actually perform the access [74].

Systems that combine data access and synchronization make it less likely that the program will suffer from synchronization errors. The programmer cannot create a synchronization error by simply forgetting to insert synchronization operations around a given data access. While most programs require additional synchronization to execute correctly, augmenting the semantics of data access to include basic synchronization operations eliminates several simple error cases.

5.4.2.3 Concurrent Object-Oriented Languages

Concurrent object-oriented programming languages provide a more structured approach to parallel programming by integrating support for concurrency into the object-oriented programming model. These languages augment the semantics of the basic operations in object-oriented programming languages to include parallel execution and synchronization.

We first discuss how researchers have augmented the object-oriented model of computation to include the generation of parallel execution. In languages like POOL-T [5], objects are given threads of control which execute concurrently. Languages like ABCL/1 [136] and COOL [30] support the concept of asynchronous methods, which execute concurrently with the invoking thread. The computation can synchronize for the return value using a mechanism similar to MultiLisp futures.

The synchronization mechanisms are all oriented around how the program accesses

objects. A basic mechanism is that each method executes with exclusive access to the receiver object. This implicit form of mutual exclusion synchronization originally appeared in the context of monitors [62], and was used in monitor-based languages such as Concurrent Pascal [23, 24] and Mesa [81, 95]. Concurrent object-oriented languages also adopt this synchronization mechanism, in part because it meshes well with the concept of a method operating on an object. Many languages relax the exclusive execution constraint to allow the concurrent execution of methods that read the same object. Like monitor languages, many concurrent object-oriented languages also provide condition variables for the implementation of precedence constraints.

In a parallel computation an object may reach a state in which it should defer certain messages. For example, an empty queue should defer a dequeue message until it receives an enqueue message and can return the enqueued object. The standard way to implement these precedence constraints in monitor-based languages is for each method to examine the state of the object and suspend itself on a condition variable if the object should defer the message. Other methods signal the condition variable when the object becomes able to respond.

Some concurrent object-oriented languages provide higher-level support for this kind of synchronization by allowing an object to temporarily disable certain messages. If the object is sent a disabled message, it will defer the message until the object reenables it. Tasks that use the result of the message block until it is reenabled, processed and the actual result is returned. In POOL-T each object's behavior is determined by a script that the object executes. When the object is ready to accept a message, it executes a construct that specifies the set of messages that it will accept at that point in time. The construct blocks until the object is sent one of the specified messages. The Rosette system [133] allows programmers to identify a set of object states and specify the set of messages that an object in a given state will accept. When an object finishes executing a method it specifies its next state. Capsules [43] allow the programmer to declaratively specify conditions that the object's state must meet for it to accept each message.

Ideally, the task of synchronizing the entire computation could be effectively decomposed into the task of generating the synchronization required for each object. In practice, however, concurrent object-oriented programs often require additional synchronization that

cuts across both object and method boundaries. Such a situation can arise, for example, if a task needs to atomically perform operations on multiple objects or multiple operations on the same object. There is no way to make the operations atomic without additional synchronization. There may also be application-specific precedence constraints on the order in which objects process messages from different tasks. While it is usually possible to modify the objects to perform this kind of synchronization, it may be counterproductive to do so. The modifications may be inappropriate (if the objects are used in other contexts or applications), inefficient (if the constraints can be implemented with one global synchronization operation instead of multiple synchronization operations distributed across the objects) or cumbersome (if the objects must be augmented with additional state to perform the synchronization).

We next consider issues associated with enforcing a pure object model. We say that an object-oriented language has a pure object model if the only way to access an object is to invoke a method with the object as the receiver. Orca, for example, enforces a pure object model [9]. A pure object model increases the security and modularity of programs by enforcing object encapsulation. It also simplifies the implementation of the language on message-passing platforms. The implementation first distributes the objects across the machine. When a task invokes a method on a remote object, the implementation can either perform a remote procedure call [99] to execute the method remotely as in the Emerald system [20, 71] or transfer the object to the invoking processor and execute the method locally. The drawback is that a pure object model can interact with the language's synchronization mechanisms to limit its flexibility. Consider, for example, a language whose synchronization mechanism enforces the mutually exclusive execution of methods that write the same object. If the language also enforces a pure object model, the programmer cannot express algorithms that concurrently write pieces of the same object.

We highlight the trade-offs associated with not enforcing a pure object model by considering the design of COOL, which allows each method to directly access any piece of data it can address using the underlying C++ language constructs. This separation of the synchronization and data allocation granularities makes the programming model more flexible (it is, for example, possible to create a program that concurrently writes a given object). It also makes the programming model less secure by increasing the difficulty of identifying and

correctly synchronizing all accesses to a given object. Finally, allowing methods to access objects other than the receiver complicates potential message-passing implementations by making it difficult to implement the shared memory at the granularity of objects. COOL only runs on machines that implement the shared memory in hardware.

5.4.2.4 Software Shared-Memory Systems

We now shift our focus to consider the issues associated with implementing the abstraction of shared memory in software on message-passing machines. Software shared-memory systems typically optimize the implementation of the shared memory by automatically migrating data to accessing processors and replicating data for concurrent read access. These optimizations and the implementation of a global name space for shared data provide significant software support for programming message-passing machines. We next discuss three different approaches to implementing the shared memory in software: the page-based approach, the region-based approach and the object-based approach.

Page-based systems such as Ivy [85], Munin [15, 14, 29] and Treadmarks [72] use the virtual-to-physical address-translation hardware to implement a cache consistency protocol at the granularity of pages. The translation hardware is used to detect accesses to remote pages, and the fault handler generates messages that move or copy the required pages from remote processors. Because the hardware checks if potentially remote data is cached locally, accesses to cached potentially remote data are as efficient as accesses to guaranteed local data. Because page-based systems distribute data across the machine at the granularity of pages, they can use the entire aggregate memory of the machine to store a single large object. Each processor only needs to hold the pages that it actually accesses.

A drawback of page-based systems is that the relatively large size of the pages increases the probability of an application suffering from excessive communication caused by false sharing (when multiple processors repeatedly access disjoint regions of a single page in conflicting ways). More recent systems [72, 15] ameliorate this problem by allowing different processors to concurrently write disjoint regions of the same page. Another potential performance problem is the substantial exception handling overhead that operating systems typically impose [7].

Page-based systems also interact with the application at the level of the raw address

space. Each parallel program must have the same address space on all the different machines on which it executes. Furthermore, because the system has no knowledge of which types of data are stored on each page, it cannot automatically apply the data format translation required in heterogeneous environments. These restrictions have so far limited page-based systems to homogeneous collections of machines. The lone exception (a heterogeneous prototype described in [137]) requires all of the compilers for the different machines to lay out data the same way and to store only one kind of data on each page.

The entry consistency protocol of Midway implements the shared address space at the granularity of program-defined regions [18, 19]. The program can define synchronization objects and associate regions of the shared memory with these objects. When the program needs to access part of a region of memory, it acquires a read or write lock on the associated synchronization object. As part of the lock acquisition the implementation ensures that an up-to-date copy of the object resides on the accessing processor. To reduce redundant communication, the Midway implementation divides shared-memory regions into fixed-size pieces (each piece is, in effect, a software cache line) and maintains a time stamp for each piece. When the system needs to bring a locally resident version of a region up-to-date it uses the time stamps to detect the out-of-date pieces and only generates the communication required to update these pieces. This approach eliminates communication wasted on transferring up-to-date pieces. Of course, the system may still waste communication transferring updated pieces that the task does not actually access.

The main advantages of the Midway's region-based approach over the page-based approach are the elimination of traps to detect accessed remote data (Midway programs provide advance notice of which data they will access) and the potential elimination of false sharing (Midway programs can divide the shared memory into regions that are not concurrently written by different tasks). A drawback is the programming overhead required to declare how the program will access data. The fact that the Midway implementation does not check the access declarations exacerbates the overhead. Another drawback is the execution overhead required to maintain the time stamps. Like page-based shared-memory systems, Midway interacts with the application at the level of the raw address space and therefore does not run on heterogeneous systems.

SAM [124], like several of the concurrent object-oriented systems described in Section

5.4.2.3, implements the shared address space at the granularity of user defined objects. The program tells the system when it will access each object and the system automatically generates a message to fetch a remote copy of the object if it is not available locally. Unlike page-based systems and Midway, SAM supports heterogeneity. The SAM implementation uses a globally valid representation for pointers to objects and can relocate objects within each address space. Because SAM knows the type of each object, it can also automatically apply the data format translation required for heterogeneous computational environments. Part of the price of supporting heterogeneity is paid in translation overhead. Whenever the program accesses a cached object the system must perform a software translation of the globally valid object identifier to the local address of the cached object.

A common theme that shapes the design of all software shared-memory systems is the need to transfer significant amounts of data in each message. In Ivy and Munin the unit of transfer is a page; both Midway and SAM rely on the programmer to aggregate the words of memory into coarser-granularity communication units. The motivation for larger-granularity communication is the significant overhead associated with each message [16]. Sending a few large messages instead of many small messages reduces the performance impact of this overhead.

5.4.2.5 Synchronization Mechanisms

The performance characteristics of message-passing systems have also influenced the design of the provided synchronization mechanisms. While a standard shared-memory interface separates synchronization from data access, software shared-memory systems typically provide an interface that associates synchronization with potential communication. As described in Section 5.4.2.2, such an interface allows the system to generate fewer messages for synchronized access to shared data. Midway, for example, provides a programming model in which the program acquires a read or write lock on a region before accessing the region. The system combines the messages required to acquire the lock and transfer the data.

SAM provides a synchronization mechanism based on version numbers. The program associates a number with each version of an object. When a task writes an object it provides a number for the newly generated version of the object. When a task reads an object, it

specifies the number of the version it needs to access and synchronizes on the availability of that version. The arrival of the single message containing the correct version of the object satisfies both the communication and synchronization requirements of the computation.

The SAM version number mechanism is similar to the Jade version consistency mechanism described in Section 3.4.7, and in fact evolved from this mechanism as part of an effort designed to give the programmer and/or higher-level systems more control over the synchronization and communication. As described in Section 3.4.7, using version numbers also eliminates coherence traffic. There is no need to update or invalidate out-of-date versions of objects because each task names the precise version it needs to access.

Exposing the version numbers directly in the programming model imposes a potential programmability problem. Requiring the programmer to generate code that calculates the version numbers can impose an onerous programming burden. SAM addresses this problem by allowing the program to access the latest version of an object (at the cost of extra communication to find the number of the latest version) and by providing libraries that encapsulate the version number calculation for common data structures with common access patterns.

5.4.2.6 Explicit Communication Operations

When a standard shared-memory system accesses remote data, it blocks until the communication required to perform the access takes place. Given the increasing cost of communication relative to computation, the resulting idle time is becoming an increasingly important cause of poor performance. A standard technique for reducing the performance impact of the communication latency is to overlap the communication with computation. Several recent software shared-memory systems allow programmers to express such optimizations by augmenting the standard shared-memory programming model with asynchronous remote read and write operations. These systems provide the functionality required to allow the programmer to control the communication at a low level for efficiency.

We start our discussion of these systems by considering the design of Split-C [77]. Like the software shared-memory systems discussed in Section 5.4.2.4, Split-C distributes a single shared address space across the memories of a message-passing machine and layers a software implementation of shared memory on top of the message-passing substrate. At

every potentially remote memory access the Split-C front end inserts code that checks if the actual access is local or remote. For remote accesses the system automatically generates a message that fetches the data from the remote memory. Split-C provides no support for the migration or replication of data. If the program must migrate or replicate data for good performance, the programmer must generate code that performs the memory management explicitly by copying remote data to local memory.

The main goal of Split-C is to integrate asynchronous remote read and write operations into a shared-memory programming system. An asynchronous remote write operation sends a write message containing the new data to the processor that owns the remote memory. The operation then returns, allowing the computation to proceed while the write message travels through the network to the remote processor. To coordinate the completion of remote write operations Split-C provides barrier operations that block until all outstanding remote writes complete.

An asynchronous remote read operation generates a message requesting the data and immediately returns, allowing the issuing task to continue with its computation. When the task actually needs to access the data it performs a synchronization operation that blocks until the data from all outstanding asynchronous read operations has arrived.

SAM also provides constructs that allow the program to control the communication. Tasks can prefetch objects, fetch multiple objects in parallel, and eagerly transfer objects from producers to consumers. These constructs allow the programmer to apply his knowledge of the application's inherent communication pattern to optimize the execution.

Both SAM and Split-C expose the latency required to access remote memory directly in the programming model. Programs can apply communication optimizations such as parallelizing the communication required to access remote data, overlapping computation with this communication, and eliminating the fetch latency associated with accessing remotely produced data. A major difference between SAM and Split-C is that SAM integrates this functionality into a shared-memory implementation that automatically replicates and migrates data. In Split-C the data for a given global address is always stored in the same memory location on the same processor. Not supporting migration and replication simplifies the implementation and enables more efficient access to remote data. There is no need to perform a check to see if the data is replicated locally, and the system does not have to

maintain naming data structures that track the current location of each piece of memory. The drawback, of course, is that many programs generate excessive communication if they do not replicate and migrate data. In this case Split-C forces the program to perform the replication and migration explicitly.

5.4.3 Discussion

Explicitly parallel systems maximize the amount of control the programmer has over the parallel execution, enabling the use of highly optimized, application-specific synchronization and parallelization algorithms. They impose minimal performance overhead because the provided constructs translate more or less directly into low-level, efficient operations. Because the model of computation is so close to the hardware, they impose no artificial expressiveness limitations in that they typically allow the programmer to express the full range of parallel algorithms.

The potential performance gains come at the price of a significantly more complex programming model. Explicitly parallel systems leave the programmer directly exposed to the full range of problems outlined in Chapter 1. They place the responsibility for generating the synchronization and, in message-passing systems, the communication, directly on the programmer. Furthermore, the programmer must develop these algorithms in a hostile programming environment characterized by complicated failure modes such as deadlock and nondeterministic, timing dependent bugs. As described in Section 2.3.5, explicitly parallel systems can also destroy the modularity of the program.

5.5 Discussion

Researchers have explored many possible design points for parallel programming systems. Despite the diversity of the resulting set of systems, no system completely satisfies the needs of most parallel programmers. This is partly due to the fact that parallel programming is an inherently difficult activity, and there is little hope that any system will actually make it easy to develop every parallel application. We believe, however, that it is possible to dramatically improve the design of existing systems. The key insight is that the needs of

the programmer change in conflicting ways during the development process, and the system must adjust to these changes.

In the early part of the development process the focus is on exploring the basic sources of concurrency in the application. The goal is to quickly develop initial parallelizations that help the programmer understand the basic characteristics of the application and guide the search for a successful parallelization. In this phase the system can support the programmer either by eliminating the possibility of programming errors or by automatically performing parts of the parallelization process.

As the development progresses, however, the focus shifts to mapping the parallel application more efficiently onto the machine. This may involve either developing more efficient synchronization and concurrency exploitation algorithms or remapping the computation to more effectively balance the load or minimize communication. For this tuning phase to go well, the system must provide a tractable performance model and allow the programmer to control the computation at a fairly low level for performance.

This conflicting set of demands shows why it is difficult to build a system that effectively supports the complete development process. The enforced abstractions in high-level systems provide a safe programming environment that promotes the quick development of initial parallelizations, but deny the programmer the flexibility and control required to effectively tune the performance. Low-level systems deliver the required control, but complicate initial development by forcing the programmer to manage many low-level aspects of the parallel execution.

We believe the ultimate solution will prove to be a layered system. The components of this system will cooperate to provide an integrated programming model that gracefully adjusts to the programmer's changing needs. The higher-level components will enforce abstractions that provide a safe, supportive programming environment for the initial parallelizations. Successive layers will peel away the abstractions to provide increasing amounts of flexibility and control, allowing the programmer to effectively tune the performance. The lowest-level components will directly expose the full functionality of each hardware platform.

We expect programmers to use the system at whatever level they find appropriate

for the computation at hand. If the program performs acceptably using only the higher-level components, the programmer can stop there. If parts of the program require extra optimization, the system will support a gradual migration to lower levels for more control. The key is for all the components at different levels to work together harmoniously to provide a continuous development path from the initial computation to an appropriately optimized parallel program. We expect concepts from Jade to be useful in structuring and implementing the higher-level components of the system.

Chapter 6

Future Work

Our completed Jade research focuses on exploring how best to structure a language to support a restricted set of parallel applications. We believe the most fruitful directions for future research will explore ways to implement Jade more efficiently and to apply concepts from Jade in new application domains. Researchers may either extend Jade to support these new domains or transplant ideas and mechanisms from Jade into other software systems. In this chapter we outline several such research directions.

6.1 Static Optimizations

In the current implementation of Jade all concurrency analysis and task management take place dynamically using general-purpose algorithms. Static analysis to discover efficiently implementable special cases could drive down the dynamic overhead, extending the implementation to support finer-grain parallel computations. Because of Jade's serial semantics, it is possible to apply analysis techniques developed for compiling serial languages directly to Jade programs.

The resulting information would allow the implementation to employ more efficient synchronization and communication algorithms for analyzable pieces of code. In some cases the message-passing implementation could use a distributed, data-driven synchronization algorithm. All synchronization would be bundled into the object messages that satisfied tasks' data usage requirements. In other situations the implementation could

replace the general-purpose object queue algorithm with a lower-overhead, coarser-grain synchronization algorithm. For example, the generated parallel algorithm could implement the series/parallel concurrency patterns characteristic of many parallel algorithms directly using barriers. The current implementation, of course, implements such concurrency patterns indirectly (and less efficiently) by dynamically analyzing how tasks access individual shared objects.

Other optimizations would parallelize or even eliminate sequential task creation. The implementation could statically discover parallel tasks and generate code that efficiently parallelized the task creation overhead. The dynamic implementation would provide special-purpose task management algorithms for such cases. The implementation could also fuse a sequence of Jade tasks into one coarser-granularity task that periodically interacted with the rest of the computation. Again, such a transformation would require support from the dynamic part of the Jade implementation.

Jade programs should be much easier for compilers to parallelize than programs written in standard serial languages. A major problem for parallelizing compilers is effectively partitioning the program into coarse-grain tasks. Jade provides constructs that programmers can use to guide the partitioning process.

A second major problem for parallelizing compilers is unraveling complex data access patterns to determine which pieces of data a piece of code actually accesses. In many cases the compiler is unable to analyze the program with enough precision to exploit important sources of concurrency. Jade's access specifications could help the compiler solve this problem. Access specifications can cleanly summarize how a complex piece of code accesses data. Analyzing a program at the level of access specifications could therefore expose more concurrency than analyzing the code that actually performs the computation.

6.2 Communication Enhancements

The message-passing implementation of Jade would benefit from a more flexible communication mechanism. In the current implementation each object is a unit of communication. A hybrid protocol that transferred small objects atomically and large objects in fixed-size pieces upon access would expand the range of supported Jade applications. It would be

possible to implement the protocol for large objects using page-based techniques developed for software shared-memory systems.

Such a hybrid approach would combine the benefits of current page-based and object-based communication mechanisms. Using a different protocol for small objects would ameliorate the false sharing problems that afflict current page-based systems. Using a page-based communication mechanism for large objects would allow computations to manipulate objects larger than any one memory module, drive down wasted bandwidth for computations that only accessed part of each object and allow the implementation to use the entire memory of the computing environment to store large objects.

Current page-based systems interact with the computation at the level of the virtual address space. For such systems to correctly implement the abstraction of a single coherent address space, each machine must use the same virtual address space for the computation. In heterogeneous systems this restriction imposes unreasonable constraints on the compilers [137].

The extra structure inherent in the Jade object model would allow the implementation to extend techniques from page-based systems for use in heterogeneous computing environments. The only restriction is that the operating system must support user-level paging operations similar to those described in [7].

The basic idea is that the Jade object model gives the implementation a machine-independent index space for each shared object. This index space consists of the object's global identifier and the indices of object's elements. Each processor would map shared objects into its address space, using the page protection mechanism to identify valid and invalid pieces of the object.

When a task accessed an invalid part of the object, the implementation would translate the faulting virtual address into the global index space of the object. It would then use this index space to request the required remotely stored elements of the object. Individual processors could map each object into different parts of their address spaces, storing the object in the appropriate native data format.

6.3 Extensions for Performance and Control

The basic problem with any high-level programming language is that it prevents the programmer from accessing the full functionality of the hardware. The restrictions may reduce the achievable performance or limit the range of expressible computations. For some applications programmers may need to use a lower-level programming system to achieve their computational goals.

In the long run, if the ideas from Jade are to survive they must become integrated into a general-purpose programming system that allows programmers to access the complete hardware functionality of the machine. As described in Section 5.5, such a system could be structured in layers. The highest layer would provide a safe, portable computational environment. Successive layers would progressively expose more of the hardware functionality. The challenge is to design a coherent sequence of layers that work together to provide an effective programming environment for the full range of parallel applications.

6.4 Nondeterministic Applications

Jade is designed to support deterministic computations. Even the commuting access declaration, which can generate nondeterministic execution, is intended for deterministic use. We next describe how to extend Jade to support both synchronous and asynchronous nondeterministic parallel computations. In each case we describe new access declarations that would allow programmers to describe the data usage patterns characteristic of each kind of computation. This approach leads to an integrated paradigm that would support many different kinds of task-level parallel computation in a single coherent language.

Synchronous computations contain parallel tasks that periodically interact through synchronized access to shared data structures. The shared data structures typically require concurrent-read/exclusive-write synchronization. Programmers cannot currently express these computations in Jade because the implementation preserves the original serial execution order for reads and writes to the same shared object. Jade could support these computations with commuting read and commuting write access declarations. Like normal read and write declarations, tasks that declared commuting reads would execute concurrently

while tasks that declared commuting writes would execute serially. But the implementation would also have the freedom to change the relative execution order between tasks that declared commuting reads and tasks that declared commuting writes, and could exploit that freedom to execute the program more efficiently.

Asynchronous computations contain parallel tasks that concurrently read and write shared data structures without synchronization. These computations typically evolve from synchronous computations via the opportunistic elimination of synchronization. The programmer realizes that the synchronization operations generate dynamic overhead and that the potential interference caused by their elimination will not cause the program to execute incorrectly. Jade could support these computations with new access declarations that allowed tasks to concurrently read and write overlapping regions of shared objects. A task that declared a concurrent read access could execute in parallel with a task that declared a concurrent write access, even if they accessed the same memory.

Chapter 7

Conclusion

Developing programming paradigms that allow programmers to effectively deal with the many different kinds of concurrency is a fundamental problem in computer science. The goal of the Jade project was to develop an effective paradigm for a specific purpose: the exploitation of task-level concurrency for performance. The concrete results of this project demonstrate that Jade, with its high-level abstractions of serial semantics and a single address space, satisfies this goal.

We have demonstrated Jade's portability by implementing it on a diverse set of hardware platforms. These machines span the range of computational platforms from tightly coupled shared-memory machines through dedicated homogeneous message-passing multiprocessors to loosely coupled heterogeneous collections of workstations. The implementations explored several key issues associated with supporting a single high-level model of computation on dramatically different hardware platforms. In particular, the implementations demonstrated how to exploit the programmer-provided data usage information to apply communication optimizations.

We evaluated the Jade language design by implementing several complete scientific and engineering applications in Jade. Our experience with these applications indicates that Jade works well for many applications that exploit task-level concurrency. We obtained excellent performance results for several applications on a variety of hardware platforms with minimal programming overhead. We also obtained less satisfactory results for programs that pushed the limits of the Jade language and implementation. Some applications would

work fine given improvements in the implementation; others would be best expressed in other languages.

Because Jade was designed to support a specific, targeted class of computations, it is, by itself, unsuitable as a general-purpose parallel programming language. Jade's enforced abstractions mean that programmers cannot express certain kinds of parallel algorithms in Jade and cannot control the machine at a low level for optimal efficiency. The ultimate impact of the Jade project will come from the integration of basic concepts and implementation techniques from Jade into other programming systems designed to support a wider range of applications. The advantage of developing a focused language like Jade is that it isolates a clear, conceptually elegant definition of the basic paradigm. Using the language therefore both allows and forces programmers to explore the advantages and disadvantages of the paradigm. With the Jade project behind us, we can identify what is missing in Jade and how the basic concepts of Jade are likely to live on in future languages and systems.

7.1 Viable Concepts from Jade

A fundamental idea behind Jade is to have programmers declaratively provide information about how the program accesses data. This is in harmony with a long-term trend in computer science to change the focus from control to data. In parallel computing the need to efficiently manage the memory hierarchy for performance will drive this change of focus. Future languages and systems will be increasingly organized around the interaction of data and computation, with various declarative mechanisms such as access specifications used to express the relevant information. COOL's locality hints[30], Midway's object usage declarations[19], shared region declarations [118] and the CHICO model of consistency[61] are all examples of this trend.

Access specifications give the implementation enough information to automatically generate the communication without forcing the implementation to use a specific communication mechanism. It is therefore possible to implement parallel languages based on access specifications on a wide variety of machines. Each implementation can use the native communication mechanism to implement the underlying abstraction of a single address space, and applications will efficiently port to all of the platforms.

Advance notice of how the program will access data gives the implementation the information it needs to apply locality and communication optimizations appropriate for the target hardware platform. In an explicitly parallel context the implementation can also use access specifications to automatically synchronize the computation.

Access specifications build on the programmer's high-level understanding of the program and mesh with the way the programmer thinks about its behavior. They allow the programmer to express complex parallel computations simply, concisely and in a way that places minimal demands on the programmer's cognitive abilities. Because access specifications provide so many concrete benefits, we expect them to appear increasingly often in future parallel language designs.

Jade supports the abstraction of a single shared address space with automatically cached data. The programming benefits of this abstraction ensure that many future languages and systems will support this approach (as many existing systems do). We expect that many such systems will use some form of access specifications to support the automatic generation of communication operations.

One of the unique features of Jade is its extreme portability. Jade currently runs on a wide range of hardware platforms and in principle could be implemented on almost any MIMD computing environment. We designed this portability into the language by scrupulously eliminating any dependences on specific architectural features. The speed with which specific computer systems become obsolete and the need to preserve software investment in parallel programs will drive a trend towards highly portable languages.

7.2 Final Remarks

There is a delicate interaction between the programmer, the computation and and the language used to express the computation. Jade takes a strong ideological position on how programmers should express, and ultimately, should think about parallel computation. Such a strong position will inevitably become diluted and absorbed into a more neutral mainstream. The value of taking such positions lies in the new perspectives they yield on the problems at hand, the new paths of research they may open up, and the intellectual joy of exploring different ways of thinking about a fascinating subject. In many areas of

inquiry only practical experience can generate satisfactory answers to the initial questions that inspired the research. One of the most satisfying aspects of Jade was experiencing its evolution from an abstract idea to an implemented system that enabled us to understand many of the practical implications of the basic paradigm.

Bibliography

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, MA, 1986.

[2] G. Agha. Concurrent object oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[3] G. Agha and C. Hewitt. Concurrent programming using Actors. In A. Yonezawa and M. Tokoro, editors, *Object Oriented Concurrent Programming*, pages 37–53. MIT Press, Cambridge MA, 1987.

[4] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[5] P. America. POOL-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object Oriented Concurrent Programming*, pages 199–220. MIT Press, Cambridge MA, 1987.

[6] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation*, Albuquerque, NM, June 1993.

[7] A. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991.

[8] Arvind and R. Thomas. I-structures: An efficient data type for functional languages. Technical Report MIT/LCS/TM-210, MIT, 1981.

[9] H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3), March 1992.

[10] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[11] P. Barth, R. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Proceedings of the Fifth ACM Conference on Functional Programming Languages and Computer Architecture*, pages 538–568. Springer-Verlag, August 1991.

[12] F. Baskett, T. Jermoluk, and D. Solomon. The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS + 100,000 lighted polygons per second. In *Proceedings of COMPCON Spring 88*, pages 468–471, 1988.

[13] BBN, Cambridge, MA. *Butterfly Parallel Processor Overview*, 1985.

[14] J. Bennett, J. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA, May 1990.

[15] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[16] R. Berrendorf and J. Helin. Evaluating the basic performance of the Intel iPSC/860 parallel computer. *Concurrency: Practice & Experience*, 4(3):223–240, May 1992.

[17] B. Bershad, E. Lazowska, and H. Levy. Presto: A system for object-oriented parallel programming. *Software – Practice and Experience*, 18(8):713–732, August 1988.

[18] B. Bershad and M. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[19] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared memory system. In *Proceedings of COMPCON'93*, pages 528–537, February 1993.

[20] A. Black, N. Hutchison, E. Jul, and H. Levy. Object structure in the Emerald system. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, 1986.

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

[22] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21th International Symposium on Computer Architecture*, Chicago, IL, April 1994.

[23] P. Brinch-Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–207, June 1975.

[24] P. Brinch-Hansen. *The Architecture of Concurrent Programs*. Prentice-Hall, Englewood Cliffs, NJ, 1977.

[25] R. Browning, T. Li, B. Chui, J. Ye, R. Pease, Z. Czyzewski, and D. Joy. Empirical forms for the electron/atom elastic scattering cross sections from 0.1-30keV. *Journal of Applied Physics*, 76(4), August 1994.

[26] R. Browning, T. Li, B. Chui, J. Ye, R. Pease, Z. Czyzewski, and D. Joy. Monte-carlo calculations of electron/atom elastic scattering from 0.1-30keV. *Scanning*, 1994. To appear.

[27] A. Burns. *Programming in Occam 2*. Addison-Wesley, Reading, MA, 1988.

[28] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[29] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[30] R. Chandra, A. Gupta, and J. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[31] K.M. Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. Technical Report Caltech-CS-TR-92-13, Computer Science Department, California Institute of Technology, 1992.

[32] E. Cooper, S. Nettles, and I. Subramanian. Improving the performance of SML garbage collection using application specific virtual memory management. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, San Francisco, CA, June 1992.

[33] D. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 141–150, May 1988.

[34] D. Culler, S. Goldstein, K. Schauser, and T. von Eicken. TAM – A compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, July 1993.

[35] D. Culler, K. Schauser, and T. von Eicken. Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, Santa Clara, CA, April 1991.

[36] H. Dietz and D. Klappholz. Refined Fortran: Another sequential language for parallel programming. In K. Hwang, S. M. Jacobs, and E. E. Swartzlander, editors,

Proceedings of the 1986 International Conference on Parallel Processing, pages 184–189, St. Charles, IL, August 1986.

[37] I. Duff, R. Grimes, and J. Lewis. Sparse matrix problems. *ACM Transactions on Mathematical Software*, 15(1):1–14, March 1989.

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

[39] I. Foster, R. Olson, and S. Tuecke. Productive parallel programming: The PCN approach. *Scientific Programming*, 1(1):51–66, Fall 1992.

[40] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[41] I. Foster and S. Tuecke. Parallel programming with PCN. Technical Report ANL-91/32, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, September 1991.

[42] 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.

[43] N. Gehani. Capsules: A shared memory access mechanism for concurrent C/C++. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):795–811, July 1993.

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

[45] K. Gharachorloo. *Memory Consistency Models for Shared Memory Multiprocessors*. PhD thesis, Stanford, CA, 1994.

[46] K. Gharachorloo, V. Sarkar, and J. Hennessy. A simple and efficient implementation approach for single assignment languages. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 259–268, Snowbird, UT, July 1988.

[47] D. Gifford, P. Jouvelot, J. Lucassen, and M. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, MIT, September 1987.

[48] G. Golub and C. Van Loan. *Matrix Computations, Second Edition*. The Johns Hopkins University Press, 1989.

[49] K. Gopinath. *Copy elimination in single assignment languages*. PhD thesis, Stanford, CA, April 1988.

[50] S. Gregory. *Parallel Logic Programming in PARLOG: the Language and its Implementation*. Addison-Wesley, Reading, MA, 1987.

[51] J. Guzman and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 333–343, Philadelphia, PA, June 1990.

[52] E. Hagersten, A. Landin, and S. Haridi. DDM – a cache-only memory architecture. *Computer*, 25(9):44–54, September 1992.

[53] M. Hall, K. Kennedy, and K. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[54] R. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[55] R. Halstead, Jr. An assessment of Multilisp: Lessons from experience. *International Journal of Parallel Programming*, 15(6):459–501, December 1986.

[56] R. Hammel and D. Gifford. FX-87 Performance Measurements: Dataflow Implementation. Technical Report MIT/LCS/TR-421, MIT, November 1988.

[57] R. Harper. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[58] J. Harris, S. Lazaratos, and R. Michelena. Tomographic string inversion. In *60th Annual International Meeting, Society of Exploration and Geophysics, Extended Abstracts*, pages 82–85, 1990.

[59] L. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[60] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, January 1993.

[61] M. Hill, J. Larus, K. Reinhardt, and D. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 262–273, Boston, MA, October 1992.

[62] C. A. R. Hoare. Monitors: An operating system concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[63] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[64] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, N.J., 1985.

[65] M. Homewood and M. McLaren. Meiko CS-2 interconnect elan-elite design. In *Proceedings of Hot Interconnects 93*, Stanford, CA, August 1993.

[66] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *Conference Record of the Twelfth Annual ACM Symposium on the Principles of Programming Languages*, pages 300–313, January 1985.

[67] P. Hudak, S. Peyton-Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and

J. Peterson. Report on the programming language Haskell: A non-strict, purely functional language (version 1.2). *SIGPLAN Notices*, 27(5):Ri–Rx, Rl–R163, May 1992.

[68] Intel Supercomputer Systems Division. *Paragon XP/S Product Overview*, 1991.

[69] D. Jefferson. Virtual time. *Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[70] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, and H. Younger. Distributed simulation and the Time Warp Operating System. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, pages 77–93, Austin, TX, November 1987.

[71] E. Jul, H. Levy, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.

[72] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–132, January 1994.

[73] Kendall Square Research Corporation, Cambridge, MA. *KSR-1 Technical Summary*, 1992.

[74] A. Klaiber and H. Levy. A comparison of message passing and shared memory architectures for data parallel programs. In *Proceedings of the 19th International Symposium on Computer Architecture*, Chicago, IL, April 1992.

[75] D. Klappholz. Refined Fortran: An update. In *Proceedings of Supercomputing '89*, Reno, NV, November 1989.

[76] D. Klappholz, A. Kallis, and X. Kong. Refined C – An update. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 331–357. MIT Press, Cambridge, MA, 1990.

[77] A. Krishnamurthy, D. Culler, A. Dusseau, S. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[78] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[79] D. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, December 1972.

[80] M. Lam and M. Rinard. Coarse-grain parallel programming in Jade. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 94–105, Williamsburg, VA, April 1991.

[81] B. Lampson and D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[82] D. Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor*. PhD thesis, Stanford, CA, February 1992.

[83] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[84] S. Leung and J. Zahorjan. Improving the performance of runtime parallelization. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–91, San Diego, CA, May 1993.

[85] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Dept. of Computer Science, Yale University, New Haven, CT, September 1986.

[86] INMOS Limited. *Occam Programming Manual*. Prentice-Hall, Englewood Cliffs, N.J., 1984.

[87] J. Lucassen. Types and effects: Towards the integration of functional and imperative programming. Technical Report MIT/LCS/TR-408, MIT, August 1987.

[88] E. Lusk, R. Overbeek, J. Boyle, R. Butler, T. Disz, B. Glickfield, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[89] M. Martonosi and A. Gupta. Tradeoffs in message passing and shared memory implementations of a standard cell router. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 88–96, August 1989.

[90] D. Maydan, S. Amarasinghe, and M. Lam. Data dependence and data-flow analysis of arrays. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[91] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[92] M. Metcalf and J. Reid. *Fortran 90 Explained*. Oxford Science Publications, 1990.

[93] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, 1990.

[94] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[95] J. Mitchell, W. Maybury, and R. Sweet. Mesa language manual, version 5.0. Technical Report CSL-79-3, Xerox Palo Alto Research Center, April 1979.

[96] E. Mohr, D. Kranz, and R. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197, June 1990.

[97] D. Mundie and D. Fisher. Parallel processing in Ada. *IEEE Computer*, 19(8):20–25, August 1986.

[98] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[99] J. Nelson. *Remote Procedure Call*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1981.

[100] J. Nieh and M. Levoy. Volume rendering on scalable shared-memory MIMD architectures. Technical Report CSL-TR-92-537, Computer Systems Laboratory, Stanford University, August 1992.

[101] R. Nikhil. Id version 90.0 reference manual. Technical Report 284-1, Computation Structures Group, MIT Laboratory for Computer Science, September 1990.

[102] R. Nikhil and K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[103] J. Palmer and G. Steele, Jr. Connection Machine model CM-5 system overview. In *Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[104] P. Pierce. The NX/2 operating system. In Geoffrey Fox, editor, *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 384–390, Pasadena, CA, January 1988.

[105] J.K. Reid. The Fortran 90 standard. In *IFIP Transactions A (Computer Science and Technology)*, volume A-2, pages 343–348, September 1991.

[106] J. Reppy. *Higher–order Concurrency*. PhD thesis, Dept. of Computer Science, Cornell University, June 1992.

[107] M. Rinard. Implicitly synchronized abstract data types: Data structures for modular parallel programming. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software and Applications*, Capri, Italy, October 1994.

[108] M. Rinard and M. Lam. Semantic foundations of Jade. In *Proceedings of the Nineteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 105–118, Albuquerque, NM, January 1992.

[109] M. Rinard, D. Scales, and M. Lam. Heterogeneous parallel programming in Jade. In *Proceedings of Supercomputing '92*, pages 245–256, November 1992.

[110] M. Rinard, D. Scales, and M. Lam. Jade: A high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, June 1993.

[111] J. Rose and G. Steele. C*: An extended C language for data parallel programming. Technical Report PL 87-5, Thinking Machines Corporation, Cambridge, MA, April 1987.

[112] J. Rose and G. Steele, Jr. C*: An extended C language for data parallel programming. In L. Kartashev and S. Kartashev, editors, *Proceedings of the Second International Conference on Supercomputing*, Santa Clara, CA, May 1987.

[113] E. Rothberg. *Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization*. PhD thesis, Stanford, CA, January 1993.

[114] J. Salmon. *Parallel Hierarchical N-body Methods*. PhD thesis, California Institute of Technology, December 1990.

[115] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and run-time compilation. *Concurrency: Practice & Experience*, 3(6):573–592, December 1991.

[116] 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(4):303–312, April 1990.

[117] J. Saltz, R. Mirchandaney, and K. Crowley. Runtime parallelization and scheduling of loops. In *Proceedings of the 1st Symposium on Parallel Algorithms and Architectures*, Santa Fe, NM, 1989.

[118] H. Sandu, B. Gamsa, and S. Zhou. The shared regions approach to software cache coherence on multiprocessors. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 229–238, San Diego, CA, May 1993.

[119] V. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the Seventeenth Annual ACM Symposium on the Principles of Programming Languages*, pages 232–245, San Francisco, CA, January 1990.

[120] V. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of the Eighteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 333–352, Orlando, FL, January 1991.

[121] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. PhD thesis, Stanford, CA, 1987.

[122] V. Sarkar. *Partition and Scheduling Parallel Programs for Multiprocessors*. The MIT Press, Cambridge, MA, 1989.

[123] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 17–26, July 1986.

[124] D. Scales and M. S. Lam. An efficient shared memory system for distributed memory machines. Technical Report CSL-TR-94-627, Computer Systems Laboratory, Stanford University, July 1994.

[125] K. Schauser, D. Culler, and T. von Eicken. Compiler-controlled multithreading for lenient parallel languages. In *Proceedings of the Fifth ACM Conference on Functional Programming Languages and Computer Architecture*, pages 50–72. Springer-Verlag, August 1991.

[126] J. Singh. *Parallel Hierarchical N-body Methods and their Implications for Multiprocessors*. PhD thesis, Stanford University, February 1993.

[127] J. Singh and J. Hennessy. Finding and exploiting parallelism in an ocean simulation program: Experiences, results, and implications. *Journal of Parallel and Distributed Computing*, 15(1):27–48, May 1992.

[128] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[129] M. Smith. Tracing with Pixie. Technical Report CSL-TR-91-497, Stanford University, November 1991.

[130] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315–339, December 1990.

[131] Thinking Machines Corporation, Cambridge, MA. *The Connection Machine CM-5 Technical Summary*, October 1991.

[132] P. Tinker and M. Katz. Parallel execution of sequential Scheme with Paratran. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 28–39, Snowbird, UT, July 1988.

[133] C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 103–112, October 1989.

[134] K. Traub. Sequential implementation of lenient programming languages. Technical Report MIT/LCS/TR-417, Massachussets Institute of Technology, 1988.

[135] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Stanford, CA, August 1992.

[136] A. Yonezawa, J. Briot, and E. Shibayama. Object oriented concurrent programming in ABCL/1. In *Proceedings of the OOPSLA-86 Conference*, pages 258–268, Portland OR, September 1986.

[137] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory. Technical Report CSRI-244, Computer Systems Research Institute, University of Toronto, September 1990.

Appendix A

The Jade Front End

A.1 The Shared-Memory Front End

The shared-memory front end performs some simple program transformations. These transformations translate the `withonly` and `with` constructs into calls to the Jade run-time system, generate the code to transfer the parameters between the parent and child tasks, insert the dynamic access checks and convert the Jade syntax to pure C code. To perform these actions the front end does a complete parse of the Jade code, including a complete type analysis. We demonstrate what the front end does by discussing the translation of several examples designed to present the functionality of the front end.

A.1.1 The `withonly` Construct

For a `withonly` construct, the preprocessor extracts the `task body` section, generates a separate function which contains the code from the `task body` section, generates code to perform the parameter transfer and generates several calls to the Jade implementation. The first example illustrates what code the front end emits when it encounters a `withonly` construct. Figure A.2 contains the C code that the front end generates for the function in Figure A.1.

The shared-memory implementation represents shared pointers with a normal C pointer to the allocated space that holds the object. In line `1` in Figure A.2 the implementation has

```
 1: add(double shared *a, double shared *b, int n)
 2: {
 3:   withonly { rd(a); wr(a); rd(b); } do (a, b, n) {
 4:     aux_add(a, b, n);
 5:   }
 6: }
```

Figure A.1: Jade Source for `withonly` Example

```
 1: add(double  *a, double  *b, int n)
 2: {
 3:   {
 4:     struct __test0 {
 5:       double *a;
 6:       double *b;
 7:       int n;
 8:     };
 9:     void __code_test0(struct __test0 *__p);
10:     struct __test0 *__p;
11:     __p = (struct __test0 *)
12:       begin_withonly(__code_test0,
13:                      sizeof(struct __test0));
14:     { rd(a); wr(a); rd(b); }
15:     __p->a = a;
16:     __p->b = b;
17:     __p->n = n;
18:     end_withonly();
19:   }
20: }
```

Figure A.2: Generated Code for `withonly` Example

converted the Jade syntax for declaring variables to C syntax for pointer declarations by removing the `shared` keyword from the declarations of `a` and `b`.

The next section of emitted code translates the `withonly` construct. The call to the `begin_withonly` routine in line `12` allocates the data structure used to hold the data associated with the task. The `begin_withonly` routine is passed a pointer to the function containing the task body, and a parameter indicating how much space is required for the task parameters. The `begin_withonly` routine returns a pointer to space inside the task data structure reserved for the task's parameters. The code in lines `14` through `16` copies the values of the parameters into this space. The code in line `13` is the `access specification` section of the `withonly` construct. The access specification statements (in this case the `rd(a)`, `wr(a)` and `rd(b)`) are calls to Jade library routines. The call to the `end_withonly` construct informs the Jade implementation that the task's `access specification` section has finished and that the parameters are in place.

Figure A.3 contains the generated task body function. This function takes a single parameter: the pointer to the space in the task data structure that holds the task's parameters. In lines `6` through `8` the front end generates code to extract the parameters from the task data structure. Line `10` contains the task body.

```
 1: void __code_test0(struct __test0 *__p)
 2: {
 3:   double *a;
 4:   double *b;
 5:   int n;
 6:   a = __p->a;
 7:   b = __p->b;
 8:   n = __p->n;
 9:   {
10:     aux_add(a, b, n);
11:   }
12: }
```

Figure A.3: Generated Task Body for `withonly` Example

A.1.2 Access Checks

We next present an example that illustrates how the front end inserts the dynamic checks required to verify that a task does not violate its access specification. The C code in Figure A.5 is what the Jade front end generates for the Jade code in Figure A.4.

```
1: aux_add(double shared *a, double shared *b, int n)
2: {
3:   int i;
4:   double local rd wr *la = a;
5:   for (i = 0; i < n; i++) {
6:     la[i] = la[i] + b[i];
7:   }
8: }
```

Figure A.4: Jade Source for Access Check Example

```
1: aux_add(double *a, double *b, int n)
2: {
3:   int i;
4:   double *la = ((double *)shared_to_local(a,0x060));
5:   for (i = 0; i < n; i++) {
6:     la[i] = la[i] + ((double *)
                        shared_to_local(b,0x020))[i];
7:   }
8: }
```

Figure A.5: Generated Code for Access Check Example

Every time the task body assigns to a local pointer or dereferences a shared pointer the implementation inserts a call to the `shared_to_local` routine in the Jade library. This routine performs the access check that enforces the preservation of the serial semantics. The first parameter to this routine is the object pointer, and the second specifies how the code will access the object. `0x040` specifies a write access, `0x020` specifies a read access, and `0x060` specifies both a read and a write access. The call to `shared_to_local` returns

```
1: double shared *make_double()
2: {
3:   return(create_object(double));
4: }
```

Figure A.6: Jade Source for Object Creation Example

```
1: double  *make_double()
2: {
3:   return((double *)create_object(sizeof(double), 1));
4: }
```

Figure A.7: Generated Code for `with` Example

a pointer to the object. In line `4` from Figure A.5 the front end has inserted a call to the `shared_to_local` routine because of the assignment to a local pointer. In line `6` the front end has inserted a call to the `shared_to_local` routine because of the dereference of a shared pointer.

A.1.3 Allocating Objects

The front end must translate the parameters of the `create_object` construct to fit the interface of the `create_object` routine in the Jade library. This routine allocates memory for each object, returning a pointer to the first byte of memory in the object. Figure A.7 presents the translation of the Jade code in Figure A.6. The front end has put a `sizeof` construct around the type, and inserted the default number of items (one) as the second parameter.

A.1.4 The `with` Construct

The next example illustrates what the front end does with a `with` construct. Figure A.9 is the translation of the Jade code in Figure A.8. The implementation inserts a call to the `begin_with` routine, then inserts the access specification from the `with` construct, then

inserts a call to the `end_with` routine. The only quirk is to ensure that the `with` construct gets treated as a single statement by putting braces around the generated code.

```
1: void zero(double shared *a, int n)
2: {
3:   int i;
4:   with { wr(a); } cont;
5:   for (i = 0; i < n; i++) {
6:     a[i] = 0.0;
7:   }
8:   with { df_wr(a); } cont;
9: }
```

Figure A.8: Jade Source for `with` Example

```
1: void zero(double  *a, int n)
2: {
3:   int i;
4:   { begin_with(); { wr(a); } end_with(); }
5:   for (i = 0; i < n; i++) {
6:     ((double *)shared_to_local(a,0x040))[i] = 0.0;
7:   }
8:   { begin_with(); { df_wr(a); } end_with(); }
9: }
```

Figure A.9: Generated Code for `with` Example

A.1.5 Global Variables

In the shared-memory implementation all pieces of code can access global variables directly. The front end therefore does nothing with global variable declarations except translate the declarations to legal C.

A.1.6 Shared Functions

The shared-memory front end represents shared function pointers as hard pointers to the address of the code. All it must do to translate the declarations and uses of shared functions is to remove the `shared` keyword to convert the Jade syntax to legal C syntax. Figure A.11 contains the C code generated from the example in Figure A.10.

```
1: int shared plus_one(int a)
2: {
3:   return(a + 1);
4: }
5:
6: int shared (*f()) (int)
7: {
8:   return(plus_one);
9: }
```

Figure A.10: Source Code for Shared Function Example

```
1: int plus_one(int a)
2: {
3:   return(a + 1);
4: }
5:
6: int (*f()) (int)
7: {
8:   return(plus_one);
9: }
```

Figure A.11: Generated Code for Shared Function Example

A.2 The Message-Passing Front End

Like the shared-memory front end, the message-passing front end translates the `with` and `withonly` constructs into calls to the Jade run-time system, generates the code to

transfer the parameters between the parent and child tasks, inserts the dynamic access checks and converts the Jade syntax to pure C code. The message-passing front end must also generate routines to pack and unpack objects, object queues and task data structures from message buffers, using a machine-independent representation for all data that crosses machine boundaries. These pack and unpack routines are built on a set of routines in the Jade implementation for manipulating message buffers. Each such routine manipulates data of a different type. The type information is present so that the implementation can apply the data format translation required to move data correctly between machines in a heterogeneous environment.

A.2.1 The `withonly` Construct

The message-passing implementation generates significantly more code for a `withonly` construct than the shared-memory implementation. We discuss what the front end does by presenting how it translates the `withonly` construct in Figure A.12.

```
1: add(double shared *a, double shared *b, int n)
2: {
3:   withonly { rd(a); wr(a); rd(b); } do (a, b, n) {
4:     aux_add(a, b, n);
5:   }
6: }
```

Figure A.12: Jade Source for `withonly` Example

Because data associated with tasks must cross processor boundaries, the implementation has to generate code that will pack and unpack data associated with tasks from message buffers. For each `withonly` construct the front end therefore generates a set of routines to pack and unpack the task's parameters from message buffers. The front end also generates a routine that computes how much space the parameter data will occupy in the message buffer. Figure A.13 contains these automatically generated routines. The routines with names like `int_put` are routines from the Jade implementation that pack typed data into message buffers; routines with names like `int_get` unpack the data.

```
 1: struct __test0 {
 2:   double *a;
 3:   double *b;
 4:   int n;
 5: };
 6: void __woput_test0(struct __test0 *p)
 7: {
 8:   object_id_put((int) p->a);
 9:   object_id_put((int) p->b);
10:   int_put(p->n);
11: }
12: void __woget_test0(struct __test0 *p)
13: {
14:   object_id_get((int *) &p->a);
15:   object_id_get((int *) &p->b);
16:   int_get(&p->n);
17: }
18: int __wosize_test0(struct __test0 *p)
19: {
20:   int size = 0;
21:   size +=  object_id_size((int) p->a);
22:   size +=  object_id_size((int) p->b);
23:   size +=  int_size(p->n);
24:   return size;
25: }
```

Figure A.13: Generated Packing and Unpacking Routines for `withonly` Example

At the actual `withonly` site the message-passing front end generates code that is very similar to the shared-memory front end. There is one subtle difference in line `14`. The message-passing front end and initialization code in the Jade message-passing library cooperate to number every `withonly` site in the program. The first parameter to the `begin_withonly` routine in the message-passing library is the global serial number of the `withonly` site. The message-passing front end generates a static variable for each file holding the serial number of the first `withonly` site in the file. In this example that variable is called `__wo_test`. This variable is declared in line `1`.

The front end computes the global serial number for each `withonly` site in the file by adding an offset to this variable. The reason the implementation uses a serial number instead of a raw function pointer is that the task data may have to cross machine boundaries, and the implementation needs a machine-independent representation of which `withonly` site generated the task. On each processor the serial number indexes tables which contain pointers to functions (such as the task body function and functions to pack and unpack task parameters from message buffers) that the implementation uses to manipulate the task. The translation of the function containing the task body is the same as in the shared-memory implementation.

```
 1: static int __wo_test;
 2:
 3: add (double *a , double *b , int n)
 4: {
 5:   {
 6:     struct __test0 {
 7:       double *a;
 8:       double *b;
 9:       int n;
10:     };
11:     void __code_test0();
12:     struct __test0 *__p;
13:     __p = (struct __test0 *)
14:       begin_withonly(__wo_test + 0,
15:                      sizeof(struct __test0));
16:     { rd(a); wr(a); rd(b); }
17:     __p->a = a;
18:     __p->b = b;
19:     __p->n = n;
20:     end_withonly();
21:   }
22: }
```

Figure A.14: Generated Code for `withonly` Example

For every file the Jade front end generates code to initialize the tables of data associated

with `withonly` constructs. The implementation calls this routine once on every processor before it transfers control to the user code. Figure A.15 contains the generated functions. The implementation uses the `__init_wo_test` function to compute how many `withonly` sites there are in the file. Each file contains an automatically generated routine that returns the number of `withonly` sites in that file. There is also an automatically generated routine that calls each of these functions in turn to compute the total number of `withonly` constructs in the whole program. The implementation then allocates tables of that size, and calls the functions that initialize the tables for each file.

```
 1: int __num_wo_test(int t)
 2: {
 3:   return(t+2);
 4: }
 5: int __init_wo_test(int t, void (*put_tbl[])(),
 6:                    void (*get_tbl[])(),
 7:                    void (*code_tbl[])(),
 8:                    int fsize_tbl[])
 9: {
10:   __wo_test = t;
11:   put_tbl[t + 0] = __woput_test0;
12:   get_tbl[t + 0] = __woget_test0;
13:   code_tbl[t + 0] = __code_test0;
14:   {
15:     struct __test0 p;
16:     fsize_tbl[t + 0] = __wosize_test0(&p);
17:   }
18:   return(t+1);
19: }
```

Figure A.15: Generated `withonly` Initialization Functions

A.2.2 Access Checks

The message-passing front end generates the exact same access check code as the shared-memory front end. The message-passing implementation implements shared pointers using

globally valid machine-independent identifiers, so the `shared_to_local` routine in the message-passing implementation must both translate the globally valid identifiers to the local version of the object and check that the task correctly declared the access. The generated code is the same as in the shared-memory implementation, however.

A.2.3 Allocating Objects

We next discuss the translation of the `create_object` construct. Because the message-passing implementation must transfer objects between processors to implement the abstraction of a single address space, the implementation must generate routines that pack and unpack objects from message buffers. The front end must also keep track of the types of objects so that the run-time system can correctly perform the data format translation required in heterogeneous environments. We describe what the front end does for objects by presenting what the front end generates for the code in Figure A.16.

```
1: double shared *make()
2: {
3:   return(create_object(double));
4: }
```

Figure A.16: Source Code for Object Creation Example

For each object creation site, the implementation generates a set of routines that pack and unpack objects created at that site. Figure A.17 contains the generated pack and unpack routines.

The implementation also generates a routine that computes how much space the object occupies in a message buffer, and a routine to deallocate the space holding any part objects to which the object points. The implementation allocates and deallocates the space associated with each object as the object moves between processors. Figure A.18 contains the generated size and deallocation routines.

Figure A.19 contains the translation of the object creation site in Figure A.16. It is similar to what the shared-memory implementation generates. There is a subtle difference in line `4`. Like `withonly` sites, the implementation numbers all of the object creation sites

```
 1: void __put_test0(double *p)
 2: {
 3:   {
 4:     int n;
 5:     if (p == 0) {
 6:       int_put(0);
 7:     }
 8:     else {
 9:       n = *((int *)(p) - 1);
10:       int_put(n);
11:       if (n != 0) double_array_put(p, n);
12:     }
13:   }
14: }
15: void __get_test0(double **pp)
16: {
17:   {
18:     int n;
19:     int_get(&n);
20:     double_array_get(*pp, n);
21:   }
22: }
```

Figure A.17: Pack and Unpack Routines for Object Creation Example

in the program. The first parameter to the `create_object` routine in the message-passing library is the serial number of the object creation site. The Jade implementation uses this number as a machine-independent representation of the object's type. On each processor it indexes a table containing local data (such as pointers to functions to pack and unpack the object's data into message buffers) that the implementation needs to manipulate objects of that type. The implementation determines the total number of object creation sites in the program and sets up the tables that point to object manipulation routines in much the same way as it performs the analogous functions for `withonly` sites.

```
 1: void __free_test0(double *p) {}
 2: int __size_test0(double *p)
 3: {
 4:   int size = 0;
 5:   {
 6:     int n;
 7:     size += int_size(n);
 8:     if (p != 0) {
 9:       n = *((int *)(p) - 1);
10:       if (n != 0) size += double_array_size(p, n);
11:     }
12:   }
13:   return size;
14: }
```

Figure A.18: Size and Deallocation Routines for Object Creation Example

```
1: static int __type_test;
2: double  *make ( )
3: {
4:   return((double *)create_object (__type_test + 0, 1));
5: }
```

Figure A.19: Generated Code for Object Creation Example

A.2.4 The `with` Construct

The message-passing Jade implementation generates the same code for `with` constructs as the shared-memory Jade implementation.

A.2.5 Global Objects

The message-passing Jade implementation generates a machine-independent identifier for every shared object. Shared pointers contain these identifiers rather than native pointers valid in only one address space. The Jade implementation therefore generates a structure for each global variable; one of the elements of this structure stores the identifier of the

global variable. There is also a dummy field that the implementation uses to store some information about the global variable. Figure A.21 contains the translated object declaration for the global variable in Figure A.20. The front end automatically adjusts all references to the global variable to take the new format into account.

```
int shared g;
```

Figure A.20: Global Object Declaration

```
struct { int *_id; double _dummy[11]; int _val[1]; } g;
```

Figure A.21: Generated Code for Global Object Declaration

When the implementation transfers the data associated with a global object between processors, it must store the data in the preallocated space – i.e. in the `val` field in the global variable declaration. The implementation must also ensure that the `_id` field contains the correct object identifier on all processors. The implementation therefore generates routines that initialize the `_id` field and a table containing pointers to the data associated with global objects. These tables are indexed by the machine-independent identifiers of the global objects.

In the described scheme a global object occupies space on all processors whether or not that processor actually references the global object. There is an option in the Jade preprocessor to simply treat global objects like other objects, and dynamically allocate and deallocate the space that holds the local version of the object as the object moves between processors. This is useful if there are very large global objects that do not need to be resident on all processors.

A.2.6 Shared Functions

Because shared function pointers can cross processor boundaries, the message-passing Jade implementation must represent them in a globally valid, machine-independent way. As for

`withonly` sites and object creation sites, the implementation numbers all of the sites in the program that use a shared function name as a value. Each shared function pointer is then represented by the serial number of the value site that generated it. When the program calls a shared function using a shared function pointer, it indirects through a table indexed by shared function numbers that contains pointers to the local versions of shared functions. This makes it possible to transfer pointers to shared functions between machines in a heterogeneous environment. We illustrate what the front end does with shared function pointers using the example in Figure A.22.

```
1: int shared plus_one(int a)
2: {
3:   return(a + 1);
4: }
5:
6: int shared (*f()) (int)
7: {
8:   return(plus_one);
9: }
```

Figure A.22: Source Code for Shared Function Example

Figure A.23 contains the generated code for the example in Figure A.22. The implementation generates a variable called `_fn_test` that contains the serial number of the first shared function name use site in the file. In line `9` the implementation uses this variable to determine the global serial number of the shared function name use site. The implementation also automatically generates routines to initialize the shared function name use sites and to generate the tables pointing to local versions of shared functions.

```
 1: static int __fn_test;
 2: int plus_one (int a)
 3: {
 4:   return (a+1);
 5: }
 6:
 7: int (*f()) (int)
 8: {
 9:   return ((int (*)()) (__fn_test + 0)) ;
10: }
```

Figure A.23: Generated Code for Shared Function Example

Appendix B

Benchmark Programs

B.1 Benchmark Programs

This appendix contains the benchmark programs that measure the basic time overhead of the Jade constructs. There are three benchmark programs: a program designed to measure the task creation and execution overhead, a program designed to measure the `with` overhead, and a program designed to determine the grain size that the Jade implementations can profitably exploit. The results in section 3.6 come from these benchmark programs.

B.1.1 `withonly` Overhead

The benchmark program in Figure B.1 measures the task creation and execution time for Jade tasks. The routine serially creates and serially executes `NUM_TASK` tasks. It batches the creation and execution in that it creates `BLOCK_TASK` tasks, then executes the tasks before it goes on to create and execute the next batch of tasks. To ensure that there is no concurrency in the creation and execution, for each batch the routine first creates one task (the blocking task) that must execute on the processor that creates the tasks. The blocking task prevents all of the subsequently created tasks in the batch from executing. The execution of each batch therefore proceeds as follows:

- The creator creates the blocking task.
- The creator creates all of the other tasks in the batch.

- The creator executes the blocking task.
- The rest of the tasks execute sequentially.

Because all of the tasks in the batch execute sequentially, there is no concurrency in the execution of the tasks. The running time accurately measures the total time to create and execute the tasks. The program assumes that the time to create and execute the blocking task is negligible compared to the time to create and execute the rest of the tasks in the batch.

B.1.2 `with` Overhead

The benchmark program in Figure B.2 measures the time to execute `with` constructs. It simply measures the time to execute a group of `with` constructs, then divides by the number of `with` constructs to get the overhead per `with`.

B.1.3 Speedup Benchmark

The program in Figures B.3 and B.4 measures the speedup for various task sizes. For each task size it serially creates 31 * 256 tasks and assigns the tasks to processors for execution in a round-robin fashion, omitting the main processor (processor 0), which creates all of the tasks. The tasks themselves execute in parallel. For a 32 processor run processors 1 through 32 (the worker processors) each execute 256 tasks.

We next develop a simple analytical model of the performance of this benchmark. Specifically, we define a function $m(o, t)$ which gives the running time of the program assuming the Jade overhead per task at the creating processor is o seconds and the tasks execute for t seconds. Because there is some concurrency in the execution of a task, o will be less than the total task creation overhead presented in Figures 3.4 and 3.5. On a real machine the overhead may vary slightly with the task time because of the interleaving of the processor interactions.

If the task size is smaller than the time required to create 31 tasks (i.e. if $31 * o < t$), every worker processor will be idle when it gets a task. In this case the critical path is the

```
#define NUM_TASK   100000
#define BLOCK_TASK 250
#define NUM_SPEC   10

do_withonly(a, proc)
int shared *a[NUM_SPEC];
int proc;
{
  int start_time, stop_time, run_time;
  int s, cs, i, num_stat = 0;
  for (cs = 1; cs <= NUM_SPEC; cs++) {
    get_time(&start_time);
    i = 0;
    while (i < NUM_TASK) {
      block {
        withonly {
          for (s = 0; s < cs; s++) wr(a[s]);
        } @ 0 do () {}
        while (i < NUM_TASK) {
          withonly {
            for (s = 0; s < cs; s++) wr(a[s]);
          } @ proc do () {}
          i++;
          if (i % BLOCK_TASK == 0) break;
        }
      }
    }
    get_time(&stop_time);
    run_time = stop_time-start_time;
    printf("proc = %d, num stmts = %d, overhead = %lf\n",
      proc, cs, ticks_to_seconds_time(run_time) / NUM_TASK);
  }
}
```

Figure B.1: `withonly` Overhead Benchmark

```
#define NUM_WITH 10000

int do_with(a, proc)
int shared *a[NUM_SPEC];
int proc;
{
  int num_stat = 0;
  int start_time, stop_time;
  int s, cs, bl;
  for (cs = 1; cs <= NUM_SPEC; cs++) {
    get_time(&start_time);
    for (bl = 0; bl < NUM_WITH; bl++) {
      with {
        for (s = 0; s < cs; s++) rd(a[s]);
      } cont;
      with {
        for (s = 0; s < cs; s++) df_rd(a[s]);
      } cont;
    }
    get_time(&stop_time);
    printf(f, "num specs = %d, overhead = %lf\n",
      cs, ticks_to_seconds_time(stop_time-start_time) /
          (2 * NUM_WITH));
  }
}
```

Figure B.2: `with` Overhead Benchmark

creation of all of the tasks followed by the execution of the last task. The running time for this task size is therefore $31 * 256 * o + t$.

If the task size is greater than or equal to the time required to create 31 tasks (i.e. if $31 * o \geq t$), each worker processor will be idle until it gets its first task, and will then be busy until it finishes executing its last task. The critical path is the creation of the first round of tasks, followed by the execution of the 256 tasks on the last processor to finish executing its tasks. The running time for this task size is therefore $31 * o + 256 * t$. We therefore

```
#define START_TIME 0.050
#define STOP_TIME  -0.0001
#define INC_TIME 0.00025
#define NUM_TASKS (31 * 256)
#define NUM_OBJECTS (31)

speedup(read, write, glb)
int   shared *read[NUM_OBJECTS];
int   shared *write[NUM_OBJECTS];
int   shared *glb;
{
  double t;
  int start_time, stop_time, n, run_time;
  for (t = START_TIME; t >= STOP_TIME; t -= INC_TIME) {
    get_time(&start_time);
    block {
      for (n = 0; n < NUM_TASKS; n++) {
        withonly {
          wr(write[n % NUM_OBJECTS]);
          rd(read[(n + 1) % NUM_OBJECTS]);
          rd(glb);
        } @ (n % NUM_OBJECTS) + 1 do (glb, n, t) {
          wait(t);
        }
      }
    }
    get_time(&stop_time);
    run_time = stop_time - start_time;
    printf("task time= %lf speedup= %lf run time= %lf\n",
      t,
      (t * NUM_TASKS) / ticks_to_seconds_time(run_time),
      ticks_to_seconds_time(run_time));
  }
}
```

Figure B.3: Speedup Benchmark

```
wait(t)
double t;
{
  int st, ct;
  get_time(&st);
  while (1) {
    get_time(&ct);
    if (ticks_to_seconds_time(ct - st) >= t) {
      break;
    }
  }
}
```

Figure B.4: `wait` Procedure for Speedup Benchmark

define $m(o,t)$ as follows:

$$m(o,t) = \begin{cases} 31 * 256 * o + t & \text{if } 31 * o \geq t \\ 31 * o + 256 * t & \text{if } 31 * o < t \end{cases}$$

We next experimentally measure the running time of benchmark program. Figures B.5 and B.6 graph the actual measured execution time $e(t)$ divided by 256 for a variety of task times t between 0 and 50 milliseconds. If there were no Jade overhead $e(t)/256$ would equal t. The deviation from this ideal displays the relative effect of the Jade overhead. Both curves have a knee at the point when $31 * o = t$. This knee explains the corresponding knee in the curves in Figures 3.8 and 3.9, reproduced here as Figures B.7 and B.8 for the convenience of the reader.

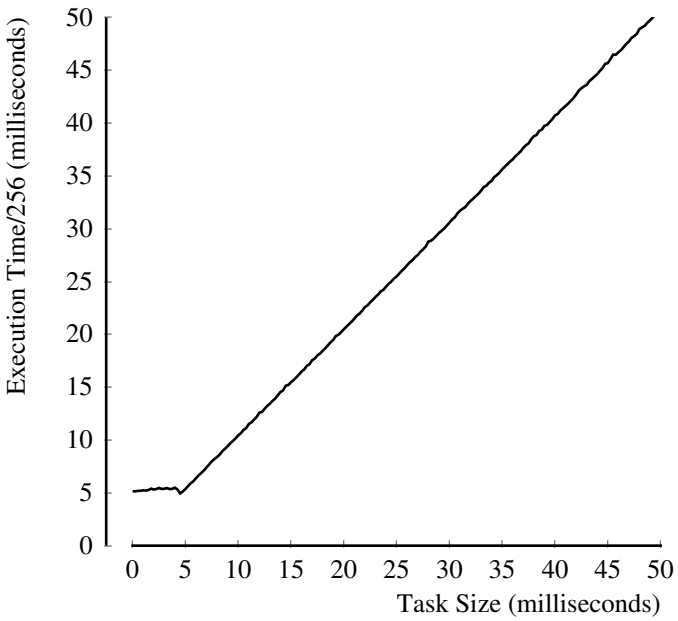

Figure B.5: Measured $e(t)/256$ on DASH

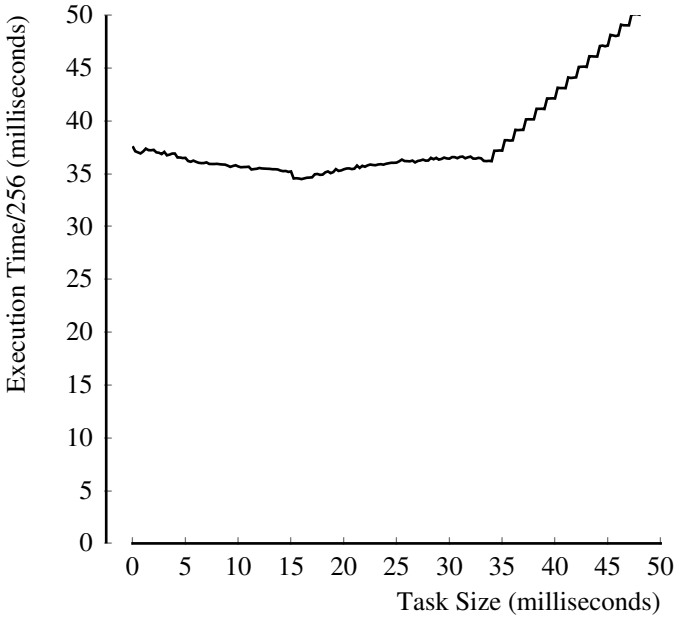

Figure B.6: Measured $e(t)/256$ on iPSC/860

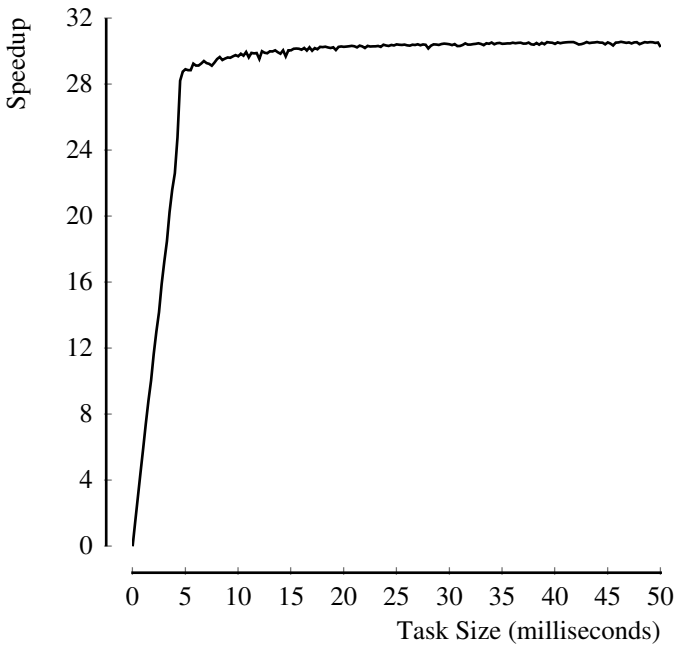

Figure B.7: Measured Speedup on DASH

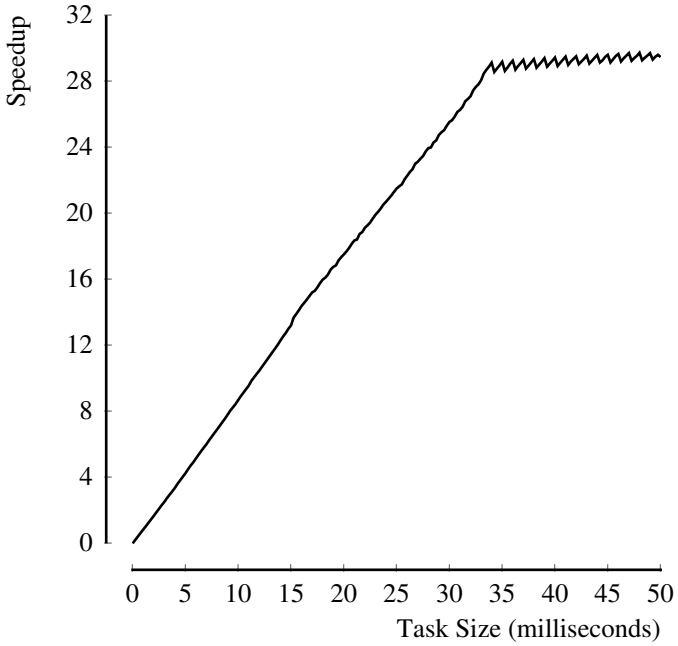

Figure B.8: Measured Speedup on iPSC/860

Appendix C

Machine Characteristics

In this Appendix we describe some basic hardware characteristics of the iPSC/860 and Stanford DASH machines. [1]

C.1 The iPSC/860

The iPSC/860 is a distributed-memory machine consisting of i860 processing nodes interconnected by a hypercube network. Each processing node consists of a 40MHz i860 XR processor with a 4KByte instruction cache and an 8KByte write-back data cache, each with 32-byte lines. Each node can deliver a maximum of 40 MIPS and 30 double-precision MFLOPS. (There are special dual-operation instructions that allow a floating-point add and multiply to execution in parallel, thus raising the peak rate to 60 MFLOPS, but these instructions are not used in our compiled code.)

The processing nodes are connected by a hypercube network that scales from 8 to 128 processors in powers of 2. The network is circuit-switched and provides a bandwidth of 2.8 megabytes per second between neighboring nodes.

Jade uses the NX/2 message-passing library on the iPSC/860, which provides primitives for sending arbitrary-sized messages between any two nodes. The NX/2 library automatically does buffering on the send and receive sides. We have measured a minimum time to

[1] We would like to thank Dan Scales for compiling the information in this Appendix.

send a short message as 47 microseconds, and the minimum round-trip for a request/reply as 154 microseconds.

C.2 The Stanford DASH Machine

The Stanford DASH machine[82] is a cache-coherent shared-memory multiprocessor. It uses a distributed directory-based protocol to provide cache coherence. It is organized as a group of processing clusters connected by a mesh interconnection network. Each of the clusters is a Silicon Graphics 4D/340 bus-based multiprocessor. The 4D/340 system has four processing nodes, each of which contains a 33MHz R3000 processor, a R3010 floating-point co-processor, a 64KByte instruction cache, 64KByte first-level write-through data cache, and a 256KByte second-level write-back data cache. Each node has a peak performance of 25 VAX MIPS and 10 double-precision MFLOPS. Cache coherence within a cluster is maintained at the level of 16-byte lines via a bus-based snoopy protocol. Each cluster also includes a directory processor that snoops on the bus and handles references to and from other clusters. The directory processor maintains directory information on the cacheable main memory within that cluster that indicates which clusters, if any, currently cache each line.

The interconnection network consists of a pair of wormhole routed meshes, one for request messages and one for replies. The total bandwidth in and out of each cluster is 120 megabytes per second. The DASH prototype supports up to 16 clusters for a total of 64 processors.

The latencies for read accesses to shared data in DASH vary depending on the current state of the data in the caches of the local processor, the processors in the local processor, and the processors in the remote clusters. A processor takes 1, 15, and 29 cycles to access data that is in its first-level cache, in its second-level cache, or in the cache of another processor in the cluster, respectively. If the data is not available in the local cluster, a request must be made to the home cluster of the data, causing a latency of 101 cycles if there is no contention. If the data is dirty in a third cluster, the request must be forwarded to that cluster and the latency is 132 cycles. The bandwidth at which the processor can access data therefore varies on the location of the data; for an average latency of 50 cycles

and assuming that the processor uses every byte in each line that it accesses, the usable bandwidth is about 10 megabytes per second.