Applications Experience in Jade

Martin C. Rinard

Department of Computer Science

University of California, Santa Barbara

Santa Barbara, California 93106

martin@cs.ucsb.edu

(805) 893-2777

Abstract

This paper presents our experience developing applications in Jade, a portable, implicitly parallel programming language designed for exploiting task-level concurrency. Jade programmers start with a program written in a standard serial, imperative language, then use Jade constructs to describe how parts of the program access data. The Jade implementation analyzes this information to automatically extract the concurrency and execute the program in parallel. The resulting parallel execution is guaranteed to preserve the semantics of the serial program. We have implemented Jade as an extension to C on shared memory multiprocessors, a homogeneous message-passing machine and networks of heterogeneous workstations.

To evaluate Jade, we obtained several complete scientific and engineering applications and parallelized them using Jade. We then executed these applications on several computational platforms. We use this 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.

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.

1 Introduction

Practitioners have implemented a wide range of languages and systems for developing programs that execute on parallel machines. Existing systems range from low-level macro packages[14] that allow the programmer to directly control the parallel execution to compilers that automatically parallelize serial computations[1]. Despite this wide range of systems, there is little consensus on how the characteristics of a given language or system support or hinder the process of developing parallel programs. We believe that an adequate understanding of this issue can only emerge from experience using the languages and systems in question to develop parallel applications.

This paper presents our experience developing applications in Jade [17, 18], a portable, implicitly parallel programming language designed for exploiting task-level concurrency. Jade programmers start with a program written in a standard serial, imperative language, then use Jade constructs to describe how parts of the program access data. The Jade implementation analyzes this information to automatically extract the concurrency and execute the program in parallel. The resulting parallel execution is guaranteed to preserve the semantics of the serial program. We have implemented Jade as an extension to C on shared memory multiprocessors, a homogeneous message-passing machine and networks of heterogeneous workstations.

To evaluate Jade, we obtained several complete scientific and engineering applications and parallelized them using Jade. We then executed these applications on several computational platforms. We use this 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. This paper presents the results of this experience.

The rest of the paper is structured as follows. In Section 2 we describe the Jade language. Section 3 presents a programming example from one of the applications; Section 4 describes some of the more advanced Jade constructs. In Section 5 we describe the application set and how it was acquired. In Section 6 we describe some of the basic characteristics of the application set. In Sections 7 through 10 we present the individual applications in detail. In Section 11 we discuss the implications of our experience as they relate to programming Jade applications and in Section 12 we discuss the performance implications of using Jade. We conclude in Section 14.

2 Language Overview

Because of Jade's sequential semantics, it is possible to implement Jade as an extension to a sequential programming language. This approach allows programmers to learn how to use Jade quickly, and preserves much of the language-specific investment in programmer training and software tools. We have currently implemented Jade as an extension to C. We first present the basic Jade constructs and then present a detailed example that illustrates their use.

2.1 Object Model

Jade supports the abstraction of a single shared memory that all tasks can access; each piece of data allocated (statically or dynamically) in this memory is called a *shared object*. Pointers to shared objects are identified in a Jade program using the `shared` type qualifier. For example:

```
double shared A[10];
double shared *B;
```

The first declaration defines a statically allocated shared vector of `doubles`, while the second declaration defines a reference (pointer) to a dynamically allocated shared vector of `doubles`. Programmers use the `create_object` construct to dynamically allocate shared objects; this construct takes as parameters the type and number of allocated items in the shared object. Given the declaration of `B` presented above, the programmer could dynamically allocate a shared vector of 100 `doubles` as follows:

```
B = create_object(double, 100);
```

2.2 Basic Construct

Jade programmers use the `withonly` construct to identify a task and to specify how that task will access data. Here is the general syntactic form of the construct:

```
withonly { access declaration } do
        (parameters for task body) {
  task body
}
```

The `task body` section contains the serial code executed when the task runs. The `parameters` section declares a list of variables from the enclosing environment that the task body may access. The `access declaration` section summarizes how the task body will access the shared objects.

The access declaration is an arbitrary piece of code containing *access specification statements*. Each such statement declares how the task will access a given shared object. For example, the `rd(o)` (read) statement declares that the task may read the shared object `o`. The `wr(o)` (write) statement declares that the task may write `o`. The task's access declaration section may contain dynamically resolved variable references and control flow constructs such as conditionals, loops and function calls. The programmer may therefore use information available only at run time when generating a task's access specification. Consequently, Jade programs can exploit dynamic concurrency that can only be determined at run time.

Since the parallelization and data movement is based on access specifications, it is important that the access specifications be accurate. The Jade implementation therefore dynamically checks each task's accesses to shared

objects. If a task attempts to perform an undeclared access, the implementation will generate a run-time error. The Jade type system has a mechanism that allows each task to perform the dynamic access check once for each object. Because the overhead is then amortized over many object accesses, in practice the total checking overhead is negligible.

Each shared object represents a unit of synchronization. If one task writes part of a shared object and another task accesses that object, the implementation executes the tasks serially. Tasks that access different shared objects or read the same objects can execute concurrently. The programmer must therefore allocate objects at a fine enough granularity to expose the desired amount of concurrency.

In a message-passing environment, each shared object is also a unit of communication. If a task will access an object, the implementation will either move (on a write access) or copy (on a read access) the entire object to the machine on which the task will execute. If the task only accesses a small part of the object, the communication overhead may negate any performance gains from concurrency. The programmer must therefore allocate objects at a fine enough granularity to minimize superfluous communication.

3 A Jade Programming Example

In this section we illustrate the use of Jade by showing how a sparse Cholesky factorization algorithm is parallelized in Jade.

3.1 Sparse Cholesky Factorization

The sparse Cholesky factorization algorithm factors a sparse, symmetric, positive definite matrix A into the form LL^T, where L is lower triangular. Because the matrix is symmetric, the algorithm only needs to store the lower triangle of the matrix. The factorization algorithm then repeatedly updates the data structures that represent this lower triangle. We assume that the matrix is stored using the data structure in Figure 1. In this data structure, the nonzeros for each column are stored in a separate packed array. Because the zeros are omitted, the data structure must represent the sparsity pattern explicitly using separate row index and start row index arrays. There is one element of the row index array for every nonzero in the packed arrays; that element provides the row index for the nonzero. The start row index data structure specifies where the row indices for a given column start. In the example in Figure 2, the row indices for column 0 occur in elements 0 through 2 of the row index array.[1] An examination of the row index array shows that element 0 of the packed array for column 0 contains the nonzero for column 0 and row 0, element 1 contains the nonzero for column 0 and row 3, and element 2 contains the nonzero for column 0 and row 4. Figure 2 provides the C data structure declarations for the sparse matrix data structure.

Figure 3 contains the code for the serial factorization 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 divides the column by the square root of its diagonal. After this internal update the current column reaches its final value. 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 right of the current column. For sparse matrices, the algorithm omits some of these updates because they would not change the updated column.

One way to parallelize this algorithm is to turn each update into a task. In this case the parallel computation consists of a coarse-grain task graph of update tasks. Each update task performs either an `InternalUpdate` or an `ExternalUpdate`; there are precedence constraints between tasks that enforce the dynamic data dependence constraints of the underlying computation. Figure 4 shows a sparse matrix and its corresponding parallel task graph. The nodes of the task graph represent update tasks, while the edges represent precedence constraints. Each `InternalUpdate` task is labelled with the index of the column that it updates, while each `ExternalUpdate` task is labelled with the indices of the two columns that it accesses. The structure of this task graph depends on the sparsity pattern of the matrix to be factored. Because the sparsity pattern of the matrix depends on the input, a programmer parallelizing this algorithm must be able to express dynamic, data-dependent concurrency.

---

[1] Following the standard C convention, the first element of each array is element 0.

0 1 2 3 4

column

| 0 | 3 | 4 | 1 | 2 | 2 | 3 | 4 | 4 |

row_index

| 0 | 3 | 5 | 6 | 8 | 9 |

start_row_index

matrix

5
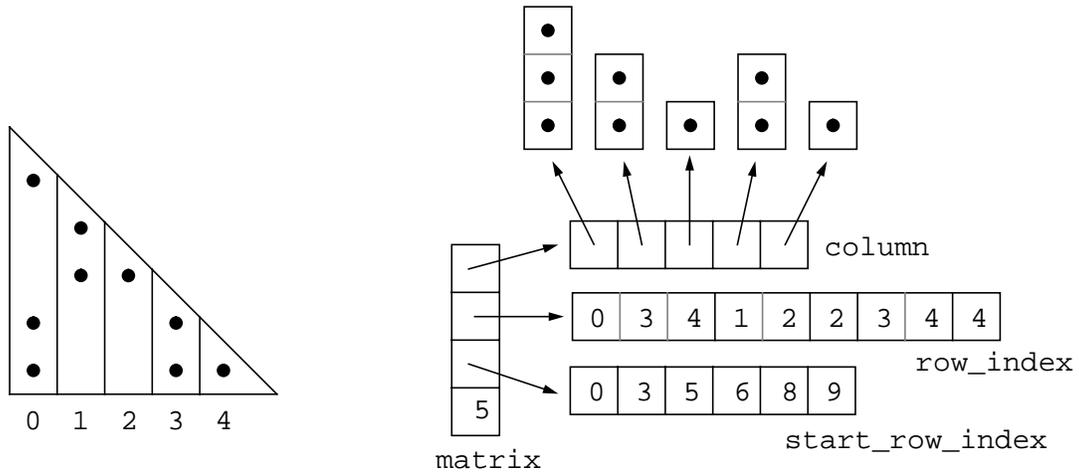
Figure 1: Data Structures for Sparse Cholesky Factorization

```
typedef double *vector;
struct {
  vector *column;
  int *row_index;
  int *start_row_index;
  int num_columns;
} matrix;
```

Figure 2: Data Structure Declarations for Sparse Cholesky Factorization

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

Figure 3: Sparse Cholesky Factorization Algorithm
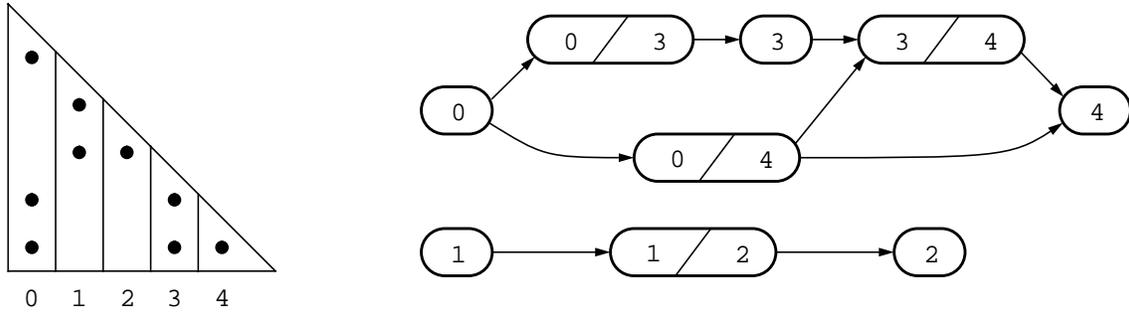
Figure 4: Task Graph for Jade Sparse Cholesky Factorization

3.2 Using Jade to Parallelize Sparse Cholesky Factorization

The first step in parallelizing a program using Jade is to determine the design of the shared objects. For some applications, the programmer may need to explicitly decompose objects into smaller allocation units to allow the application to concurrently write disjoint parts of the object. In the sparse Cholesky example the objects are already allocated at the appropriate granularity. The programmer only needs to modify the matrix declaration to identify the references to shared objects, as shown in Figure 5. The programmer also uses the `part` type qualifier to indentify several of the objects that the `matrix` points to as part of the matrix shared object. The matrix shared object is therefore a composite shared object that includes the `column`, `row_index` and `start_row_index` part objects.

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

Figure 5: Data Structure Declarations for Jade Sparse Cholesky Factorization

The programmer next augments the sequential code with `withonly` constructs to identify the tasks and specify how they access shared objects. To parallelize the sparse Cholesky factorization code, the programmer adds two `withonly` constructs to the original serial code; each construct identifies an update as a task. Figure 6 contains the Jade version of the sparse Cholesky factorization algorithm. The first `withonly` construct uses the `rd_wr(m->column[i])` and `rd(m)` access specification statements to declare that the `InternalUpdate` will execute *with only* reads and writes to the `i`'th column of the matrix and reads to the column array and row index data structures. The second `withonly` construct declares that the `ExternalUpdate` will execute *with only* reads and writes to the `r[j]`'th column of the matrix, reads to the `i`'th column of the matrix, and reads to the column vector and row index data structures.

At this point the programmer is done: the Jade implementation has all the information it needs to execute the factorization in parallel. When the program executes, the main task sequentially 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 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 farming these tasks out to processors as

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

Figure 6: Jade Sparse Cholesky Factorization

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.

4 Advanced Language Features

We have, so far, described the basic programming model in Jade. Here we describe briefly some of the advanced constructs of the Jade language that allow a Jade programmer to achieve more sophisticated parallel behavior.

4.1 Hierarchical Concurrency

Programmers may create hierarchical forms of concurrency in a Jade program by dynamically nesting `withonly` constructs. The task body of a `withonly` construct may execute any number of `withonly` constructs, in a fully recursive manner. The access specification of a task that hierarchically creates child tasks must declare both its own accesses and the accesses performed by all of its child tasks.

4.2 More Precise Access Specifications

As described so far, Jade constructs support only a limited model of parallel computation in which all synchronization takes place at task boundaries. Two tasks may either execute concurrently (if none of their accesses conflict) or sequentially (if any of their accesses conflict). The `withonly` construct by itself allows no partial

overlap in the executions of tasks with data dependence conflicts. This limitation makes it impossible to exploit pipelining concurrency available between tasks that access the same data.

Jade relaxes this restricted model by allowing the programmer to provide more precise information about when a task actually accesses data. Jade provides this functionality with an additional construct, `with`, and the additional access specification statements `df_rd(o)`, `df_wr(o)`, `no_rd(o)`, and `no_wr(o)`. The general syntactic form of the `with` construct is:

```
with { access declaration} cont;
```

As in the `withonly` construct, the `access declaration` section is an arbitrary piece of code containing access declaration statements. These statements update the task's access specification, allowing the specification to reflect more precisely how the remainder (or continuation, as the `cont` keyword suggests) of the task will access shared objects.

The `df_rd(o)` and `df_wr(o)` statements declare a *deferred* access to o. That is, they specify that the task may eventually read or write the object, but that it will not do so immediately. Because the task cannot immediately access the object, it can execute concurrently with earlier tasks that do access the object. When the task reaches the point where it will access the object, it must execute a `with` construct that uses the `rd(o)` or `wr(o)` access declaration statements to convert the deferred declaration to an *immediate* declaration. This immediate declaration then gives the task the right to access the object. In order to preserve the serial semantics, the Jade implementation may suspend the task at the `with` construct until preceding tasks with conflicting accesses to the object have completed. It is also possible to use the `df_rd(o)` and `df_wr(o)` statements inside a `with` construct to convert immediate declarations to deferred declarations. This may allow the task and any future child tasks that declare immediate accesses to the objects in question to execute concurrently.

The `no_rd(o)` (no future read) and `no_wr(o)` (no future write) statements allow the programmer to declare that a task has finished its access to o and will no longer read or write o. This declaration dynamically reduces a task's access specification, which may potentially eliminate a conflict between the task executing the `with` and later tasks. In this case the later tasks may execute as soon as the `with` executes rather than waiting until the first task completes.

5 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 the applications originally came from the SPLASH benchmark suite [20]. 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 the applications, however, the Jade parallelization was the first (and so far the only) parallelization to exist.

We next describe each of the 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 [4] and performs the same computation. The SPLASH benchmark suite [20] also contains a version of Water. The author 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. The author originally obtained a serial version of String [12] 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. The author 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 [6, 7]. Jun Ye (a graduate student in the Stanford Electrical Engineering Department) developed an initial serial version of Search in C. The author 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, the author obtained a parallel version written in C by Ed Rothberg (a graduate student in the Stanford Computer Science Department) using the ANL macro package. The author 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.

6 Application Characteristics

We next present some basic application characteristics and use these characteristics to discuss several aspects of the Jade programming process. Table 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 the 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 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.

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 1: Static Application Characteristics

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 2: Dynamic Application Characteristics for the iPSC/860

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). Table 2 presents some basic performance numbers for the iPSC/860 runs, while Table 3 presents the corresponding results for the DASH runs. For reasons described below in Section 8, 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.

7   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.

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 3: Dynamic Application Characteristics for DASH

7.1 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.

7.2 String

String [12] 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.

7.3 Search

Search [6, 7] 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.

7.4   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. We note that this general approach is the classic, correct way to parallelize the applications, and there is nothing specific to Jade about this particular parallelization.

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.

7.5   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 7 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.

To express the reduction in Jade, we first develop a serial procedure to perform the operations in the reduction. Figure 8 contains this serial procedure. It takes two parameters: `a`, which is the array pointing to the set of

```
 *a[3] ─────
              ╲
                ╲  *a[2] += *a[3] ╲
              ╱                     ╲
 *a[2] ─────                          ╲
                                        *a[0] += *a[2]
 *a[1] ─────                          ╱
              ╲                     ╱
                ╲  *a[0] += *a[1] ╱
              ╱
 *a[0] ─────
```
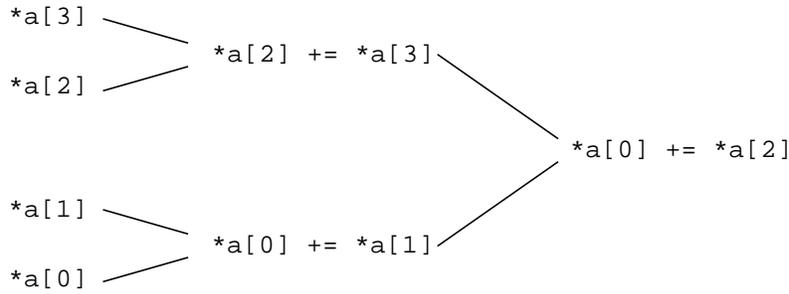
Figure 7: Parallel Reduction

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 8) is always the next power of two.

```
 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 8: Serial Code for Parallel Reduction

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 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 9. 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.

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 8. 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 10, which performs the same computation as Figure 8, illustrates how we will use this notation.

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
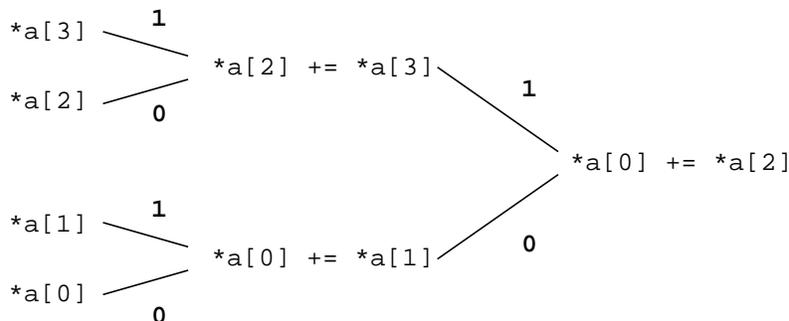
Figure 9: Labeled Parallel Reduction

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

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 11 contains the Jade code for the reduction.

7.6   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 `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 12 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 7.4, 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. 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

```
 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 11: Jade Code for Parallel Reduction

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 13 contains the data structure definition for the accumulator object. This object points to the copies of 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 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. 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.

7.7   Water Performance Results

Table 4 presents the running times for Water on the iPSC/860; Table 5 presents the running times on DASH. The ANL figures are from the explicitly parallel version of this application from the SPLASH benchmark suite. Figures 16 and 17 present the corresponding speedup curves. The speedups are given with respect to the execution times of the sequential version from Tables 2 and 3. The sequential version is the serial program generated by stripping all of the Jade constructs out of the Jade version of the program. It executes with no Jade overhead.

```
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 12: Parallel Schema for Water, String and Search

```
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 13: Accumulator Data Structure

```
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 14: Accumulator Creation and Initialization

```
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 15: Accumulator Operations

	1	2	4	8	16	24	32
Jade	2452.24	1231.70	622.14	318.00	166.43	118.96	91.86

Table 4: Execution Times for Water on the iPSC/860 (seconds)

	1	2	4	8	16	24	32
Jade	3283.92	1655.66	835.88	425.40	223.13	152.83	118.54
ANL	3677.57	1834.83	927.96	501.00	254.04	175.85	144.10

Table 5: Execution Times for Water on DASH (seconds)
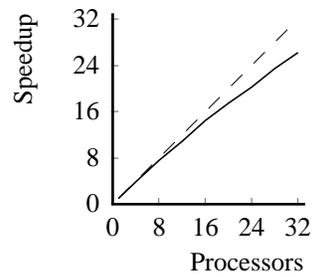
Figure 16: Speedups for Water on the iPSC/860
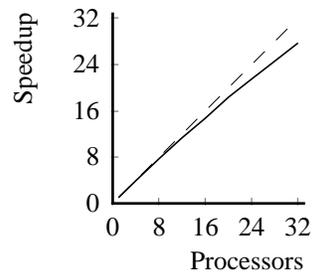
Figure 17: Speedups for Water on DASH

7.8 String Performance Results

Table 6 presents the running times for String on the iPSC/860, Table 7 presents the running times on DASH. Figures 18 and 19 present the corresponding speedup curves.

	1	2	4	8	16	24	32
Jade	18876.92	9496.70	4764.54	2415.14	1248.62	865.97	676.48

Table 6: Execution Times for String on the iPSC/860 (seconds)

	1	2	4	8	16	24	32
Jade	19280.41	9794.00	4926.22	2502.83	1287.95	886.70	693.26

Table 7: Execution Times for String on DASH (seconds)
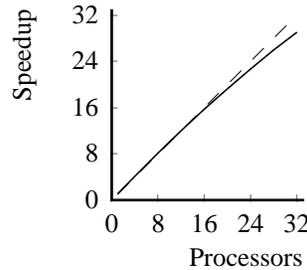
Figure 18: Speedups for String on the iPSC/860

7.9 Search Performance Results

Table 8 presents the running times for Search on the iPSC/860, Table 9 presents the running times on DASH. Figures 20 and 21 present the corresponding speedup curves.

8 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 [15] determined that distributing pixels to processors in 8 by 8 chunks yields acceptable locality.
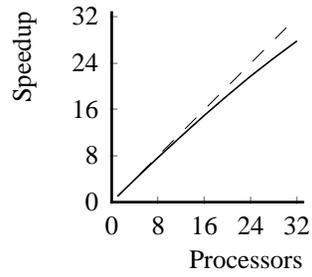
Figure 19: Speedups for String on DASH

	1	2	4	8	16	24	32
Jade	1420.05	715.49	371.87	181.66	94.33	62.75	47.63

Table 8: Execution Times for Search on the iPSC/860 (seconds)

	1	2	4	8	16	24	32
Jade	1813.67	910.70	459.16	242.34	130.46	77.77	61.06

Table 9: Execution Times for Search on DASH (seconds)
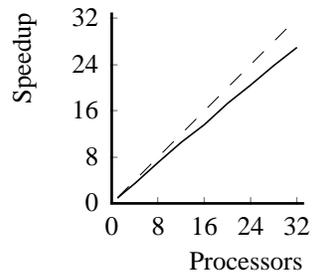
Figure 20: Speedups for Search on the iPSC/860
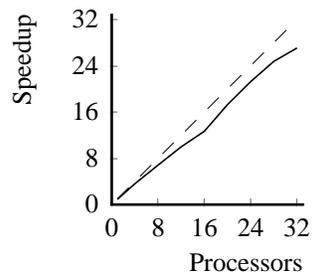
Figure 21: Speedups for Search on DASH

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 [15]. 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.

	1	2	4	8	16	24	32
Jade	33.06	16.49	8.70	4.58	2.84	2.10	2.01
ANL	31.60	15.00	7.67	3.88	1.97	-	1.01

Table 10: Execution Times for Volume Rendering on DASH (seconds)
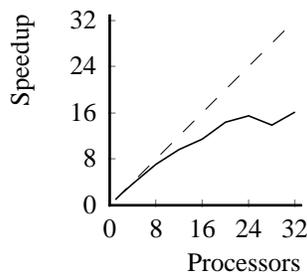
Figure 22: Speedups for Volume Rendering on DASH

Table 10 presents the execution times for Volume Rendering; Figure 22 presents the corresponding speedup curves. We were able to obtain performance numbers for the explicitly parallel version described in [15]. This version was written using the ANL macro package [14]. The Jade performance tails off quickly after 16 processors, while the explicitly parallel version scales almost linearly to 32 processors. The sources of inefficiency for the Jade version include slow task creation and startup, a load imbalance, and the serial copyback phase. See [16] for a

full analysis of each source of inefficiency.

9 Panel Cholesky

Panel Cholesky factors a sparse positive-definite matrix. The computation is basically the same as in the Cholesky factorization algorithm presented in Section 3 except that 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 [19] 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.

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. 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.

Table 11 presents the running times for Panel Cholesky on the iPSC/860, Table 12 presents the running times on DASH. The ANL version is an explicitly parallel version of the computation written in the ANL macro package; the COOL version is an explicitly parallel version written in the concurrent object-oriented language COOL[8]. Figures 23 and 24 present the corresponding speedup curves.

	1	2	4	8	16	24	32
Jade	53.62	34.11	33.91	35.54	43.84	47.77	49.73

Table 11: Execution Times for Panel Cholesky on the iPSC/860 (seconds)

	1	2	4	8	16	24	32
Jade	33.84	17.47	11.03	7.21	6.93	7.29	7.36
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 12: Execution Times for Panel Cholesky on DASH (seconds)
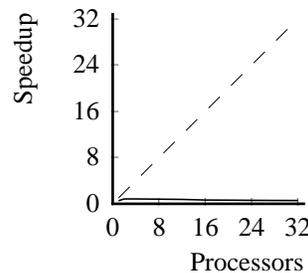

Figure 23: Speedups for Panel Cholesky on the iPSC/860

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
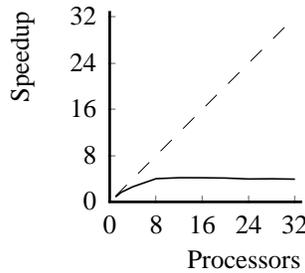
Figure 24: Speedups for Panel Cholesky on DASH

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 [19]. 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.

10 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 25. 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.[2]
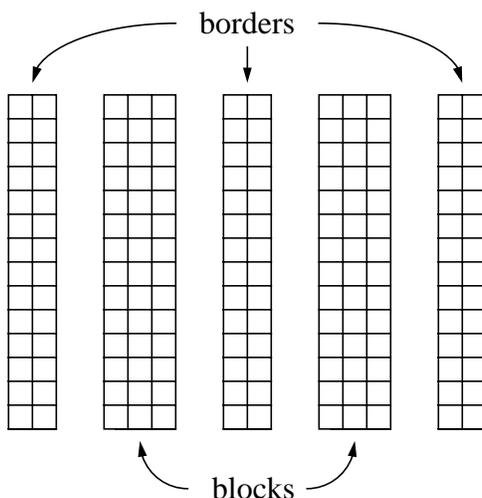
Figure 25: Ocean Data Decomposition

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.

Table 13 presents the running times for Ocean on the iPSC/860, Table 14 presents the running times on DASH. The COOL version is an explicitly parallel version written in the concurrent object-oriented language COOL[8]. Figures 26 and 27 present the corresponding speedup curves.

	1	2	4	8	16	24	32
Jade	77.75	92.49	96.38	60.35	39.15	45.21	56.95

Table 13: Execution Times for Ocean on the iPSC/860 (seconds)

	1	2	4	8	16	24	32
Jade	104.13	99.58	37.24	25.00	17.69	14.13	13.21
COOL	104.99	53.56	28.36	14.57	7.54	5.40	4.75

Table 14: Execution Times for Ocean on DASH (seconds)

On the iPSC/860 the limiting factor is the task management overhead, more specifically the large latency required for the task management messages. On DASH the task management overhead is again the limiting factor

---

[2]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.
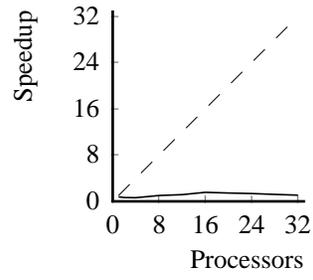
Figure 26: Speedups for Ocean on the iPSC/860
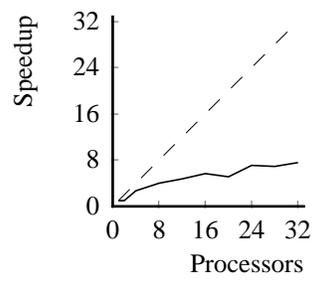
Figure 27: Speedups for Ocean on DASH

on the performance. But since DASH supports a more efficient Jade implementation than does the iPSC/860, the DASH performance is substantially better than the iPSC/860 performance. See [16] for a complete analysis of these effects.

11   Programming Discussion

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 the 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.

12   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 led to the serial copyback phase, which substantially impaired the DASH performance.

We note that the three applications that exhibited the best overall performance (Water, String and Search) share a common concurrency pattern: independent computations linked by parallel reductions. We believe, however, that Jade is suitable for a much more general class of applications. The primary determining factors are grain size and amount of concurrency, not the specific concurrency pattern. Water, String and Search perform well because they have a large grain size and because almost all of the computation takes place inside parallel phases. We expect Jade to work well for any computation that exhibits these characteristics, regardless of its specific concurrency pattern.

13   Related Work

Many researchers have built systems that are designed to exploit task-level parallelism. We briefly survey several of these systems and compare them to Jade. PVM [21] and MPI [11] are message passing systems. They provide standard message passing constructs such as `send` and `receive` that allow parallel programs to communicate by sending messages. The major functionality that these systems provide over the standard Unix system calls is support for heterogeneity and a clean interface that eliminates much of the complexity of process creation and low-level communication details. Programmers who use such systems, however, are still responsible for explicitly extracting the concurrency and generating all of the communication. Jade is designed to provide a much higher level interface: the Jade implementation, not the programmer, automatically exploits the concurrency and, on message-passing machines, generates the communication. We have found PVM useful, however, as a portable communication platform — one implementation of Jade uses PVM as its communication library [16].

Fortran M supports a more abstract message passing model of computation [10]. Programmers structure the computation as a set of modules that interact by passing messages through channels. The model of communication is much more flexible than in PVM or MPI, which require the program to specify the specific process to which every message will be sent. Fortran M is also has a clean theoretical foundation and supports language restrictions that guarantee deterministic execution. Fortran M programmers, however, are still responsible for extracting the concurrency and generating the communication.

CODE and HeNCE [2, 5] provide a graphical interface for creating modules and channels and linking them together into a communicating system. The modules are typically written in a sequential language such as C or Fortran. Although programmers that use either of these systems must still extract the concurrency and generate the communication manually, the graphical interface does make it much easier to link the modules together. The graphical representation can also make it easier to understand the parallel structure of the final program.

CC++ supports a shared-memory model of computation [9]. Instead of sending messages, processes communicate by reading and writing a shared memory. The CC++ implementation is responsible for implementing this abstraction on message passing machines. CC++ provides definitional, or write-once variables, which support a channel-based programming style similar to Fortran M. Munin [3] and TreadMarks [13] are two other systems that provide the abstraction of shared memory in a message passing environment. They support a more conventional shared memory programming model based solely on mutable data and synchronization constructs such as mutual exclusion locks. Shared memory systems are one step closer to Jade than message passing systems — both Jade and shared memory systems provide the abstraction of a single mutable memory. The major distinction is that the shared memory systems discussed here are explicitly parallel systems that rely on the programmer to express the concurrency explicitly. In Jade, the implementation automatically extracts the concurrency, which frees the programmer having to write detailed code to control the parallel execution.

14   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.

References

[1] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[2] A. Beguelin, J. Dongarra, G. Geist, R. Manchek, and V. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[3] 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.

[4] 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.

[5] J. Browne, S. Hyder, J. Dongarra, K. Moore, and P. Newton. Visual programming and debugging for parallel computing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(1):75–83, Spring 1995.

[6] 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.

[7] 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.

[8] 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.

[9] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[10] I. Foster and K. M. Chandy. FORTRAN M : a language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, April 1995.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[12] 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.

[13] 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*, January 1994.

[14] 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.

[15] 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.

[16] M. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Stanford, CA, 1994.

[17] M. Rinard, D. Scales, and M. Lam. Heterogeneous Parallel Programming in Jade. In *Proceedings of Supercomputing '92*, pages 245–256, November 1992.

[18] M. Rinard, D. Scales, and M. Lam. Jade: a high-level, machine-independent language for parallel programming. *Computer*, 26(6):28–38, June 1993.

[19] E. Rothberg. *Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization*. PhD thesis, Stanford, CA, January 1993.

[20] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[21] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315–339, December 1990.