Heterogeneous Parallel Programming in Jade

Martin C. Rinard, Daniel J. Scales and Monica S. Lam
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

This paper presents Jade, a high-level parallel programming language for managing coarse-grain concurrency. Jade simplifies programming by providing the programmer with the abstractions of sequential execution and a shared address space. Jade programmers augment sequential, imperative programs with constructs that declare how parts of the program access data; the Jade implementation dynamically interprets this information to execute the program in parallel. This parallel execution preserves the serial semantics of the original program. Jade has been implemented as an extension to C on shared-memory multiprocessors, a message-passing machine, networks of heterogeneous workstations, and systems with special-purpose functional units. Programs written in Jade run on all of these platforms without modification.

1 Introduction

This paper addresses the problem of providing programming support for coarse-grain concurrency. It is often possible to increase the performance of regular, data-parallel computations by aggregating operations into large tasks. Coarse-grain concurrency is also available in many irregular computations. This form of concurrency is available, for example, between independent sub-computations that access different data structures. These computations typically generate complex concurrency patterns with sophisticated synchronization requirements. The parallelism is often data dependent, requiring dynamic detection and management.

This research was supported in part by DARPA contract DABT63-91-K-0003.

Exploiting coarse-grain concurrency is often the most cost-effective way to increase the performance of an application. The large task size and infrequent communication that characterize such computations allow them to execute efficiently in such cheap, widely available computational environments as loosely-coupled collections of workstations. Applications with specialized computational requirements may exploit coarse-grain concurrency in heterogeneous environments. For example, one part of a computation may require the computational power of a dedicated signal processor, while another part may visually present results on a sophisticated graphics workstation.

The design of a language for expressing coarse-grain concurrency should address the important but often conflicting goals of programmability, portability, and efficiency. Programs written in the language should be easy to develop, debug and maintain. These programs should execute efficiently and without modification on the wide variety of computational environments that can effectively execute coarse-grain parallel computations. These environments range from shared-memory multiprocessors through networks of workstations to heterogeneous systems with special-purpose accelerators connected by high-speed networks.

This paper presents Jade, a high-level parallel programming language for developing coarse-grain applications. Jade simplifies programming because it allows the programmer to use the familiar shared-memory, sequential, imperative programming paradigm. Instead of using low-level constructs to create, synchronize, and communicate between parallel tasks, Jade programmers provide information about how a serial program uses data. The Jade implementation then interprets this information to automatically detect the concurrency in the application, distribute the tasks across the parallel machines, and manage the data movement required to implement the shared address space abstraction. Jade provides high-level support for heterogeneous parallel programming by allowing programmers to run sequential, imperative programs in parallel on a heterogeneous collection of machines.

Because Jade preserves the serial, imperative program-

ming paradigm, all parallel executions of a Jade program deterministically generate the same result as a serial execution of the program. This property of Jade considerably simplifies the process of debugging parallel applications. Rather than struggling to reproduce subtle timing-dependent bugs, Jade programmers can employ the same standard techniques used to debug serial programs.

Jade programmers supply application-specific information about how tasks access data, not low-level commands that directly manage the hardware. This high-level approach gives implementors the flexibility they need to generate optimized implementations of Jade on a wide variety of platforms. On shared memory machines, the implementation only needs to synchronize the computation and can rely on the hardware to implement the shared address space. In a message-passing environment, the implementation must also map the shared address space onto the local address spaces and manage the data movement between machines. If the heterogeneous machines in the system have different data formats, the implementation must perform the format translation required to maintain a correct representation of the data on each machine.

We have implemented Jade as an extension to C on shared memory multiprocessors, a homogeneous message passing machine (the Intel iPSC/860), a network of heterogeneous workstations, and a heterogeneous system containing special-purpose graphics accelerators. A previous paper [9] describes a preliminary design of Jade that applied only to shared-memory machines. [16] contains a formal treatment of the equivalence between the serial and parallel executions of Jade programs. This paper describes the Jade language and provides a detailed example of how to use Jade to parallelize a sparse Cholesky factorization algorithm. We also present some preliminary applications experience and performance numbers for Jade applications executing in a variety of computational environments.

2 Language Overview

Jade programmers provide the high-level knowledge required for effective parallel execution of a Jade program. The programmer must specify three things: 1) a decomposition of the data into the atomic units that the program will access, 2) a decomposition of a sequential program into tasks, and 3) a description of how each task will access data when it runs.

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

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

The first declaration defines a statically allocated vector of `doubles`, while the second declaration defines a reference (pointer) to a dynamically allocated vector of `doubles`.

Jade programmers use the `withonly-do` 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` (read) statement declares that the task may read the given object, while the `wr` (write) statement declares that the task may write the given object. Each access specification statement takes one parameter: a reference to the object that the task may access. 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.

If one task declares that it will write a shared object and another task declares that it will access that object, we say that the two tasks declare conflicting accesses. Tasks that declare no conflicting accesses (because they access disjoint sets of objects or read the same objects) can execute concurrently. To preserve the serial semantics, the Jade implementation executes tasks that declare conflicting accesses to the same object serially and in the same order as they would execute in the underlying sequential program.

Since the parallelization and data movement is based on access specifications, it is important that the access specifications be accurate. If the programmer fails to declare an access, tasks might execute in the wrong order or attempt to access an object that is not available at the current processor. Such undeclared accesses could cause

nondeterministic or erroneous parallel execution. The Jade implementation precludes this possibility by dynamically checking 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

We now provide an example which illustrates both how to use Jade and how Jade programs execute on a message-passing machine. In this example we show how Jade is used to parallelize a sparse Cholesky factorization algorithm.

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. This data structure represents the sparsity pattern of the matrix using one contiguous vector of integers that contains the row indices of the matrix's nonzero elements. The matrix itself is represented as a vector of columns. Each column consists of a packed vector representing that column's nonzeros, and an offset into the row index vector that tells where that row indices for that column begin. Figure 2 provides the C data structure declarations for the sparse matrix data structure.

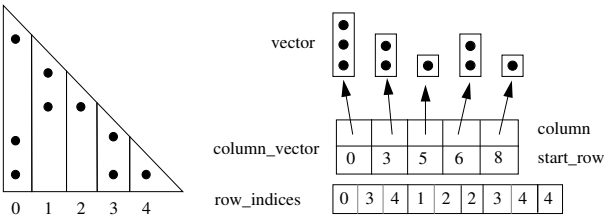

Figure 1: Sparse Matrix Data Structure

```
typedef int *row_indices;
typedef double *vector;
typedef struct {
  int start_row;
  vector column;
} column_data;
typedef column_data *column_vector;
```

Figure 2: C Data Structure Declarations

The serial factorization algorithm processes the columns of the matrix from left to right. The algorithm 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. Figure 3 contains the serial code for this algorithm.

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

```
factor(c, r, n)
column_vector c;
row_indices r;
int n;
{
  int i, j;
  for (i = 0; i < n; i++) {
    /* update column i */
    InternalUpdate(c, r, i);
    for (j = c[i].start_row;
         j < c[i+1].start_row; j++) {
      /* update col. r[j] with col. i */
      ExternalUpdate(c, r, i, r[j]);
    }
  }
}
```

Figure 3: Sparse Cholesky Factorization Algorithm

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.

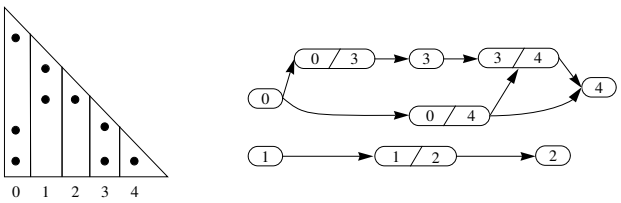

Figure 4: Dynamic Task Graph

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 next augments the sequential code with `withonly-do` constructs to identify the tasks and specify how they access shared objects. To parallelize

```
typedef double shared *vector;
typedef int shared *row_indices;
typedef struct {
  int start_row;
  vector column;
} column_data;
typedef column_data shared *column_vector;
```

Figure 5: Jade Data Structure Declarations

the sparse Cholesky factorization code, the programmer adds two `withonly-do` 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-do` construct uses the `rd_wr` and `rd` 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-do` construct uses the same access specification statements to declare 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 on a heterogeneous collection of machines.

This example highlights the data-oriented, implicitly parallel aspects of Jade. The Jade programmer only provides information about how parts of the program access data. The programmer does not explicitly specify which tasks can execute in parallel. The Jade implementation, not the programmer, detects the available concurrency. Because the access specifications are dynamically determined, the programmer can express the dynamic, data-dependent concurrency available in the sparse Cholesky factorization.

This example also illustrates how Jade programs can exploit forms of concurrency that are well beyond the reach of current parallelizing compilers. The dynamically resolved data accesses characteristic of this and other sparse matrix computations defeat the dependence-testing algorithms of such compilers. We know of no automatic aggregation algorithm that can generate this program's coarse-grained tasks. In fact, this example is actually a simplification of the sparse Cholesky algorithm that we have parallelized in Jade; in the more complex algorithm, the task grain size is increased further by aggregating adjacent columns into groups called "supernodes".

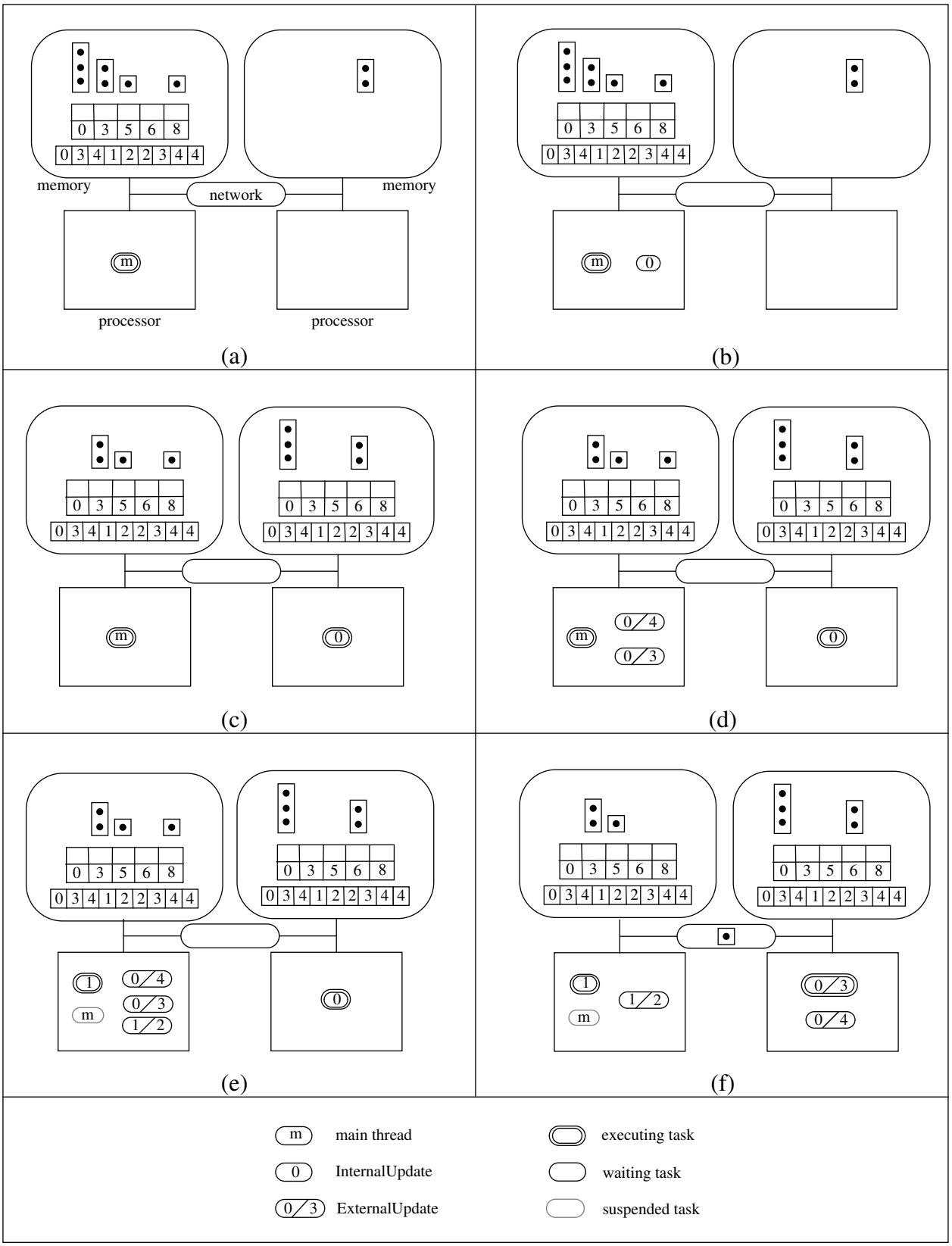

Figure 7: Executing A Jade Program

```
factor(c, r, n)
column_vector c;
row_indices r;
int n;
{
  int i, j;
  for (i = 0; i < n; i++) {
    withonly {
      rd_wr(c[i].column); rd(c); rd(r);
    } do (c, r, i) {
      InternalUpdate(c, r, i);
    }
    for (j = c[i].start_row;
         j < c[i+1].start_row; j++) {
      withonly {
        rd_wr(c[r[j]].column);
        rd(c[i].column); rd(c); rd(r);
      } do (c, r, i, j) {
        ExternalUpdate(c, r, i, r[j]);
      }
    }
  }
}
```

Figure 6: Jade Sparse Cholesky Factorization

3.3 Executing a Jade Program

We now trace the execution of the sparse Cholesky factorization algorithm on a collection of message-passing machines. This will demonstrate how the Jade implementation manages the parallel execution of such dynamic computations on, for example, a network of workstations.

Figure 7 shows the system state at several points in the execution of the Jade sparse Cholesky factorization algorithm. In this figure, the algorithm is executing concurrently on two machines connected by a network. Each machine has its own private memory; the two machines communicate with messages. Figure 7(a) labels the various parts of system and graphically represents the initial state of the system. In this initial state, one machine is executing the original task that starts the program execution. This main task will create all of the tasks in the factorization. We assume that the columns are initially distributed across the system.

In Figure 7(b), the original thread has created the first task, the `InternalUpdate` to column 0. This task is ready to execute, but is suspended because the first machine is still running the original thread. Because the second machine is idle, the Jade implementation will move the `InternalUpdate` task to that machine to be executed.

In Figure 7(c), the implementation has moved the `InternalUpdate` task to the second machine. Based on the task's access specification, the Jade implementation generates the messages required to move or copy the required objects to the machine that will access them. The `InternalUpdate` will write the first column, so the implementation has moved column 0 to the second machine and deallocated the version on the first machine (which will be obsolete after the `InternalUpdate`). The `InternalUpdate` task will only read the column vector and row index data structures. Consequently, the Jade implementation copies these data structures to the second machine without invalidating the version on the first machine, thus allowing the machines to read these data structures concurrently. In moving or copying objects between machines, the implementation (or the transport protocol it uses) also performs any data format conversion required because of different representations of data items on the two machines.

As the `InternalUpdate` executes, it will use references to access various shared objects. Because objects can migrate across machines, each reference to a shared object is in reality a globally valid identifier for that object. The Jade front end generates code that dynamically translates the globally valid identifier to a pointer to the local version of the object. The access checking that guarantees the serial semantics takes place at the same time as the global to local translation. As described earlier, the task can amortize the cost of one translation/check over many accesses to the same shared object.

In Figure 7(d), the original task has generated the two `ExternalUpdate` tasks from column 0 to columns 3 and 4. In this case there is a dynamic data dependence conflict between the `InternalUpdate` task and the two `ExternalUpdate` tasks: the `InternalUpdate` task writes column 0, while the two `ExternalUpdate` tasks read column 0. The implementation therefore suspends the two `ExternalUpdate` tasks until the `InternalUpdate` task completes.

In Figure 7(e), the implementation has generated the `InternalUpdate` to column 1 and the `ExternalUpdate` from column 1 to column 2. At this point the implementation may decide that the original task is creating tasks faster than they are being consumed, and that there is a danger that excessive task generation may overwhelm the machine. In this case, the implementation can suspend the original task and execute the `InternalUpdate` to column 1. In many explicitly parallel languages, suppressing excessive task creation may create deadlock [8, 13]. In Jade, on the other hand, the implementation can suppress task creation in the face of

excess concurrency without risking deadlock. The serial semantics guarantees that a task never waits for a subsequent task (in the original serial order). Consequently, the implementation can suspend any task with at least one outstanding child task or legally inline any task without risking deadlock.

In Figure 7(f), the `InternalUpdate` to column 0 has completed, and the implementation has moved the two `ExternalUpdate` tasks from the first machine to the second machine. Because columns 0 and 3 are already located on the second machine, the implementation can execute the `ExternalUpdate` from column 0 to column 3 immediately. Column 4, on the other hand, is still located on the first machine. The Jade implementation will therefore issue a message to the first machine requesting column 4. While column 4 is in transit from the first machine to the second machine, the implementation executes the `ExternalUpdate` from column 0 to column 3. The Jade implementation uses the excess concurrency present in the computation to hide the latency of fetching remote data.

The execution of a Jade program can be viewed as a process of creating and executing a parallel task graph. Some systems [3, 12] separate the task generation and execution phases of a computation. However, the Jade implementation overlaps the creation and execution of the task graph. Overlapping task generation and execution allows the machine to get an early start on the computation. More importantly, it allows the Jade implementation to execute programs whose dynamic task graphs depend on the results of previously executed tasks, or whose task graphs are too large to represent explicitly in their entirety.

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-do` constructs. The task body of a `withonly-do` construct may execute any number of `withonly-do` 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-do` construct 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 extends 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-cont`, and the additional access specification statements `df_rd`, `df_wr`, `no_rd`, and `no_wr`. The general syntactic form of the `with-cont` construct is:

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

As in the `withonly-do` 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` and `df_wr` statements declare a *deferred* access to the shared object. 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-cont` construct that uses the `rd` or `wr` 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-cont` construct until preceding tasks with conflicting accesses to the object have completed.

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

4.3 Higher-level Access Specifications

The Jade access specifications introduced so far only allow the programmer to declare that a task will read or write certain objects. The programmer, however, may have higher-level knowledge about the way tasks access objects. For example, the programmer may know that even though two tasks update the same object, the updates can happen in either order. The Jade programming model generalizes to include access specifications that allow the programmer to express such higher-level program knowledge.

4.4 Low-Level Control

Jade provides several low-level constructs that programmers can use to guide the default management algorithms in the Jade implementation. Programmers can explicitly specify the machine on which a task will execute or an object will be allocated. The programmer also has access to the number of machines involved in the computation for use in adjusting the task granularity.

5 Summary of Jade Language and Implementation

Jade implements several high-level abstractions that support the effective development of coarse-grain parallel applications:

- Shared Address Space. Jade programs reference shared objects via a single uniform address space that is accessible by all tasks.
- Implicit Concurrency. Jade programmers only provide information about how tasks access shared data, and tasks execute in parallel if they have no conflicting accesses to data. Jade programmers indirectly express the possibility of concurrent execution by creating tasks whose access specifications do not conflict.
- Serial, Imperative Paradigm. Every parallel execution of a Jade program deterministically preserves the semantics of the serial, imperative program upon which it is based.

These concepts together support a model of programming that is familiar to programmers and convenient for them to use. We have implemented this programming model on shared-memory multiprocessors, a homogeneous message-passing machine, and on a heterogeneous collection of workstations. Jade promotes the portability of parallel applications by providing a high-level programming model which can be supported on widely different parallel architectures.

Given this high-level programming model, the Jade implementation, rather than the programmer, handles much of the complexity involved in running a parallel application. Several aspects of the Jade implementation involve executing the application *correctly* according to the Jade programming model:

- Parallel Execution. The Jade implementation analyzes tasks' access declarations to determine which tasks can execute concurrently without violating the serial semantics. It also generates the synchronization required to execute conflicting tasks in the correct order.
- Access Checking. The Jade implementation dynamically checks each task's accesses to ensure that its access specification is correct. If a task attempts to perform an undeclared access, the implementation generates an error.
- Object Management. In a message-passing environment, the Jade implementation moves or copies objects between machines as necessary to implement the shared address space abstraction. The implementation also translates global object references to pointers to the local version of objects.
- Data Format Conversion. In a computing environment with different representations for the same data items, the Jade implementation performs the data format conversion required to maintain a coherent representation of the data on different machines.

Jade's high level programming model gives the Jade implementation the flexibility it needs to optimize the parallel execution of Jade programs. The current Jade implementation uses the following optimization algorithms in an attempt to execute applications *efficiently*:

- Dynamic Load Balancing. The Jade implementation keeps track of which processors may be idle and dynamically assigns executable tasks to processors which may become idle. Dynamic load balancing is especially important in a heterogeneous environment in which some machines execute faster than others.
- Matching Exploited Concurrency with Available Concurrency. The Jade implementation suppresses excess task creation as necessary in order to prevent the excessive generation of concurrency from overwhelming the parallel machine.

- Enhancing Locality. The Jade implementation uses a heuristic that attempts to execute tasks on the same processor if they access some of the same objects. Such a task assignment may improve locality, because tasks can reuse objects fetched by other tasks.
- Hiding Latency with Concurrency. If there is excess concurrency in the application, the Jade implementation hides the latency associated with accessing remote objects by executing one task while fetching the shared objects that another task will access.
- Object Replication. In a message-passing environment, the Jade implementation replicates shared objects for concurrent access.

The list of activities of the implementation illustrates the utility of Jade. Without Jade, programmers would have to manage the associated problems themselves. In a large or complicated parallel program, this management software could dominate the application development process. Jade allows programmers to concentrate on the algorithmic aspects of the particular application at hand rather than on the systems issues associated with mapping that application onto the current parallel machine.

6 Comparison with Other Work

In this section we compare Jade with other systems designed to provide the abstraction of a shared address space on a collections of machines that interact via message passing. We consider three alternatives in turn: distributed shared memory, Linda, and object-oriented parallel languages.

6.1 Distributed Shared Memory

The difficulties associated with managing the movement and replication of data have prompted researchers to look for ways to provide a shared-memory interface to a collection of message passing machines [2, 11, 15]. The typical approach is to use a cache-coherence algorithm implemented in software, with pages taking the place of cache lines and the page fault hardware detecting attempts to access remote or invalid pages. The page-fault handler can then copy the remote page on a read access or move the page on a write access.

Although such systems free the programmer from having to explicitly manage the data movement, the programmer still has to generate a synchronization algorithm for each application. The inherently nondeterministic nature of the distributed shared memory paradigm can make it difficult to debug and maintain these synchronization algorithms. Jade programs execute deterministically in accordance with the serial program on which they are based.

Another potential problem is that all sharing takes place at the granularity of pages. If the program accesses an object that is smaller than a page, the page coherence system will fetch the entire page. The comparatively large size of pages also 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). This problem does not occur in Jade because all data sharing takes place at the level of individual objects.

Distributed shared memory systems deal with the raw address space of the application and have no knowledge of the types of the data stored in various locations in memory. Each parallel program must therefore have the same address space on all the different machines on which it executes. This restriction has so far limited distributed shared memory systems to homogeneous collections of machines (except for a prototype described in [19]). The Jade implementation can do the necessary conversions in a heterogeneous environment because it knows the types of all shared objects.

One advantage of the distributed shared memory approach over Jade is that the shared memory abstraction is provided completely transparently to the programmer. In Jade, the programmer must specify the granularity of shared objects and declare which variables and data structures reference shared objects.

6.2 Linda

Linda [5] is an explicitly parallel language that supports the concept of a global tuple space. Linda programmers create parallel processes that interact via asynchronous operations that insert, read and remove tuples from tuple space. One major difference between Jade and Linda is that Jade supports an object model that is integrated into the serial imperative language on which Jade is based. In Jade programs, shared objects are normal data structures which programmers manipulate using the standard operators provided by the underlying programming language. In Linda, on the other hand, shared objects must be located in a separate area of storage (tuple space). Tasks which manipulate shared data must explicitly extract the data from tuple space, manipulate it locally, then reinsert it.

The other major difference between Jade and Linda is that Linda is an explicitly parallel, inherently nondeterministic programming language. Each parallel application written in Linda contains an application-specific synchronization algorithm built using the low-level tuple-space

primitives.

6.3 Parallel Object-Oriented Languages

There are a variety of distributed programming languages that take an object-oriented approach to programming message-passing machines or networks of workstations [4, 6, 1]. These languages typically make each object a unit of communication and each method invocation a unit of synchronization. Each method that modifies an object obtains exclusive access to that object; in some systems objects may be replicated to allow concurrent read access. Programmers using these languages exploit concurrency by using built-in constructs to explicitly create parallel tasks. Programmers must then manually synchronize these tasks using the built-in primitive of mutual exclusion on object access. Some systems also support a signalling mechanism similar to that used on monitor-based systems.

Jade differs from these languages in that it provides an implicitly parallel, inherently deterministic programming paradigm. Another difference is that in the parallel object-oriented languages, each method invocation can access only one object. These languages do not support the concept of an atomic task that accesses multiple objects. If a programmer has an application (such as the sparse Cholesky factorization example presented earlier) that requires tasks to access multiple objects atomically, the programmer must implement a distributed locking protocol that correctly synchronizes the multiple accesses. Jade correctly synchronizes tasks that access any number of objects.

7 Preliminary Applications Experience

We have implemented Jade on shared-memory parallel processors and on both heterogeneous and homogeneous message-passing environments. There are no source code modifications required to port Jade applications between these platforms. Our shared-memory implementation of Jade runs on the Silicon Graphics 4D/240S multiprocessor and on the Stanford DASH multiprocessor [10]. The workstation implementation of Jade uses PVM [18] as a reliable, typed transport protocol. Currently, this implementation of Jade runs on the SPARC-based Sun workstations and on MIPS-based systems, including the DECStation 3100 and 5000 machines, the Silicon Graphics workstations, and the Stanford DASH multiprocessor. Jade applications can use any combination of these kinds of workstations in the course of a single execution. We have also implemented Jade on the Intel iPSC/860 using the native message passing system. Jade also runs on the High Resolution Video (HRV) Workstation [14] from Sun Microsystems Laboratories. The machine consists of several SPARC and Intel i860 processors; its functional units are capable of digitizing and displaying video at up to full HDTV rates.

To test the Jade paradigm, we have implemented several computational kernels, including the sparse Cholesky factorization algorithm described earlier and the Barnes-Hut algorithm for solving the N-body problem. We have also implemented several complete applications in Jade. The following describes our experience in developing several applications chosen to illustrate different aspects of Jade.

7.1 Make

make is a UNIX program that incrementally recompiles a program based on which of its dependent files have changed. *make* reads as input a "makefile" which describes, for each file involved in the compilation process, the command to be executed to rebuild that file from its dependent files. The serial *make* program contains a loop that sequentially executes the commands required to rebuild out-of-date files. In the Jade version of this program, the body of this loop is enclosed in a `withonly-do` construct that declares which files each recompilation command will access. As the loop executes, it generates a task to recompile each out-of-date file. The Jade implementation executes these tasks concurrently unless one command depends on the result of another command. The dynamic parallelism available in the recompilation process defeats static analysis: it depends on the makefile and on the modification dates of the files it accesses. It is easy to express this form of concurrency in Jade, however. The performance of the *make* program is limited by the amount of parallelism in the recompilation process and the available disk bandwidth.

7.2 Digital Image Processing

An application that we have developed on the HRV workstation illustrates how programmers can use Jade to control the execution of a coarse-grain application on a heterogeneous machine. In this application a SPARC-based workstation uses a camera to capture and compress in hardware a sequence of video frames. It passes each frame to one of the i860-based graphics accelerators, which decompresses the frames in software, applies a simple digital transformation, and displays the frame on the HDTV monitor. The Jade version of this program consists of a loop with two `withonly-do` constructs. The first construct's task body acquires a camera frame; the second construct's task body applies the digital transformation.

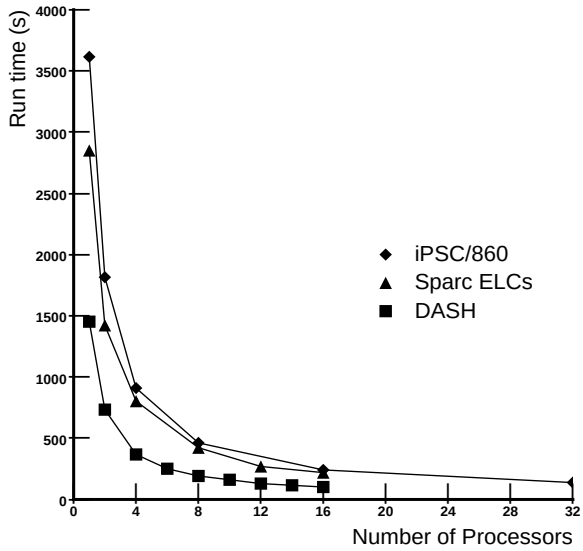

Figure 8: Running Times for Liquid Water Simulation

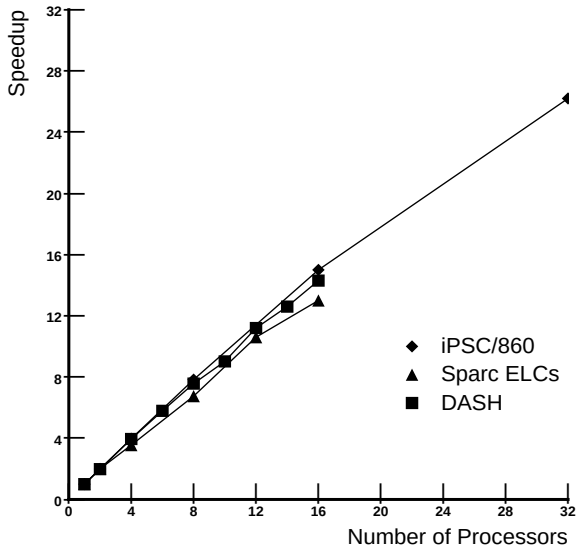

Figure 9: Speedups for Liquid Water Simulation

Using Jade simplifies the development of this application because the programmer does not have to write complex message-passing code to initiate the communication between the workstation and the graphics accelerators and to manage the movement of frames through the machine.

7.3 Liquid Water Simulation

LWS is a program derived from the Perfect Club benchmark MDG that evaluates forces and potentials in a system of water molecules in the liquid state. For the problem sizes that we are running, almost all of the computation takes place inside the $O(n^2)$ phase that determines the pairwise interactions of the n molecules. We therefore execute only that phase in parallel and run the $O(n)$ phases serially. To parallelize this program in Jade we first restructured several of the program's data structures and then added 23 Jade constructs. These modifications increased the size of the program from 1216 to 1358 lines of C code.

Figure 8 shows a plot of the running times of the same Jade LWS program in three different parallel environments: an Intel iPSC/860, a Mica multiprocessor (an array of Sparc ELC boards connected by Ethernet from Sun Microsystems Laboratories), and the Stanford DASH shared-memory multiprocessor. All times are for a simulation involving 2197 particles. Figure 9 shows a plot of the speedups for the same runs. There is ample coarse-grain parallelism in the LWS application; the figures confirm that Jade can give good performance for such an application over a range of architectures.

8 Conclusions

Jade is a high-level language for developing portable coarse-grain parallel programs. Jade supports the development of such programs by providing programmers with the abstractions of sequential execution and a shared address space. Jade programmers apply their application knowledge to define suitable task and data granularity and describe how each task accesses data. The Jade implementation uses this information to automatically detect the concurrency in the application, map the tasks onto the various parallel machines, and, in message-passing environments, manage the data movement required to implement the shared address space abstraction.

Because the Jade implementation dynamically resolves tasks' access specifications, Jade programs can exploit dynamic, data-dependent concurrency. The run-time overhead associated with detecting and managing dynamic concurrency limits the grain size that Jade programs can efficiently use. Jade's serial semantics, however, enables the direct application of parallelizing compiler techniques to Jade programs. Jade therefore promotes a hierarchical model of parallel computation in which a high-level programming language allows programmers to exploit coarse-grain, data-dependent concurrency while the compiler exploits fine-grain, concurrency statically available within tasks. The difficulty of applying compiler optimizations to explicitly parallel code [7, 17] limits the amount of concurrency that programmers can exploit using explicitly parallel languages.

We have implemented Jade in a wide variety of computational environments, from tightly-coupled shared mem-

ory multiprocessors through networks of workstations to heterogeneous systems with special-purpose accelerators connected by high-speed networks. Jade programs execute without modification on all of these computational platforms. Our initial applications experience demonstrates that Jade effectively supports the development of efficient coarse-grain parallel programs.

Most portable parallel programming languages give the programmer only the low-level functionality present on all of the targeted platforms. With a sequential semantics and a single address space model, Jade provides portability and efficiency as it maintains a high level of programming abstraction.

Acknowledgments

Jennifer Anderson participated in the initial design and implementation of Jade. We thank Edward Rothberg for his help with the sparse Cholesky factorization code and Jason Nieh for advice on porting Jade to the iPSC/860. We also would like to thank Jim Hanko, Eugene Kuerner, and Duane Northcutt for helping us port Jade to the HRV workstation, and Dave Ditzel for use of the Mica network of Suns for evaluating Jade's performance.

References

[1] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3), March 1992.

[2] John Bennett, John B. Carter, and Willy 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*, pages 168–176, March 1990.

[3] A. A. Berlin. Partial Evaluation Applied to Numerical Computation. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 139–150, June 1990.

[4] 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.

[5] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[6] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems*, pages 147–158, December 1989.

[7] Jyh-Herng Chow and William Ludwell Harrison III. Compile-Time Analysis of Parallel Programs that Share Memory. In *Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 130–141, January 1992.

[8] D. E. Culler and Arvind. Resource Requirements of Dataflow Programs. In *15th Annual International Symposium on Computer Architecture*, pages 141–150, May 1988.

[9] M. S. Lam and M. C. 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, April 1991.

[10] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *Computer*, 25(3):63–79, March 1992.

[11] Kai Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages II 94–101, August 1988.

[12] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and K. Crowley. Principles of Runtime Support for Parallel Processors. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 140–152, 1988.

[13] E. Mohr, D. A. Kranz, and R. H. 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.

[14] J. Duane Northcutt, Gerard A. Wall, James G. Hanko, and Eugene M. Kuerner. A High Resolution Video Workstation. *Signal Processing: Image Communication*, 4(4 & 5):445–455, 1992.

[15] U. Ramachandran, M. Yousef, and A. Khalidi. An Implementation of Distributed Shared Memory. *Software - Practice and Experience*, 21(5):443–464, May 1991.

[16] M. C. Rinard and M. S. Lam. Semantic foundations of Jade. In *Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 105–118, January 1992.

[17] H. Srinivasan and M. Wolfe. Analyzing Programs with Explicit Parallelism. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 405–419. 1992.

[18] V.S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[19] Songnian Zhou, Michael Stumm, Kai Li, and David Wortman. Heterogeneous Distributed Shared Memory. Technical Report CSRI-244, Computer Systems Research Institute, University of Toronto, September 1990.