The Design, Implementation, and Evaluation of Jade

MARTIN C. RINARD
Massachusetts Institute of Technology
and
MONICA S. LAM
Stanford University

Jade is a portable, implicitly parallel 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 declare how parts of the program access data. The Jade implementation uses this data access information to automatically extract the concurrency and map the application onto the machine at hand. The resulting parallel execution preserves the semantics of the original serial program. We have implemented Jade as an extension to C, and Jade implementations exist for shared-memory multiprocessors, homogeneous message-passing machines, and heterogeneous networks of workstations. In this article we discuss the design goals and decisions that determined the final form of Jade and present an overview of the Jade implementation. We also present our experience using Jade to implement several complete scientific and engineering applications. We use this experience to evaluate how the different Jade language features were used in practice and how well Jade as a whole supports the process of developing parallel applications. We find that the basic idea of preserving the serial semantics simplifies the program development process, and that the concept of using data access specifications to guide the parallelization offers significant advantages over more traditional control-based approaches. We also find that the Jade data model can interact poorly with concurrency patterns that write disjoint pieces of a single aggregate data structure, although this problem arises in only one of the applications.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.3.2 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*

General Terms: Languages, Performance

Additional Key Words and Phrases: Parallel computing, parallel programming languages

The research reported in this article was conducted while the first author was a graduate student in the Computer Science Department at Stanford University. The research was supported in part by DARPA contract N00039-91-C-0138.
Authors' addresses: M. Rinard, MIT Laboratory for Computer Science, 545 Technology Square NE43-620A, Cambridge, MA 02139; email: martin@lcs.mit.edu; M. Lam, Computer Science Department, Stanford University, Stanford, CA 94305; email: lam@cs.stanford.edu.

1. INTRODUCTION

Programmers have traditionally developed software for parallel machines using explicitly parallel systems [Lusk et al. 1987; Sunderam 1990]. These systems provide constructs that programmers use to create parallel tasks. On shared-memory machines the programmer synchronizes the tasks using low-level primitives such as locks, condition variables, and barriers. On message-passing machines the programmer must also manage the communication using explicit message-passing operations such as send and receive. Explicitly parallel systems are only one step removed from the hardware, giving programmers maximum control over the parallel execution. Programmers can exploit this control to generate extremely efficient computations. But explicitly parallel systems also directly expose the programmer to a host of program development and maintainance problems.

Existing parallel machines present two fundamentally different programming models: the shared-memory model [Hagersten et al. 1992; Kendall Square Research Corporation 1992; Lenoski et al. 1992] and the message-passing model [Intel Supercomputer Systems Division 1991; Thinking Machines Corporation 1991]. Even machines that support the same basic model of computation may present interfaces with significantly different functionality and performance characteristics. Developing the same computation on different machines may therefore lead to radically different programs [Salmon 1990; Singh 1993], and it can be difficult to port a program written for one machine to a machine with a substantially different programming interface [Martonosi and Gupta 1989].

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

For some parallel programs with simple concurrency patterns, the programmer can generate this management code without too much difficulty, and its direct incorporation into the source code does not significantly damage the structure of the program. In general, however, an explicitly parallel programming environment complicates the programming process and can impair the structure and maintainability of the resulting program. To generate correct synchronization code, the programmer must develop a global mental model of how all the parallel tasks interact and keep that model in mind when coding each task. The result is a decentralized concurrency management algorithm scattered throughout the program. To function effectively, a new programmer attempting to maintain the program must first reconstruct, then understand, both the global synchronization algorithm and the underlying mental model behind the algorithm. Explicitly parallel environments therefore destroy modularity because they force programmers to understand the dynamic behavior of the entire program, not just the module at hand [Rinard 1994b].

Finally, nondeterministic execution exacerbates all of the problems outlined above.

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

Jade [Rinard and Lam 1992; Rinard et al. 1992; Rinard et al. 1993] is a high-level, implicitly parallel language designed for exploiting task-level concurrency. Jade programmers direct the parallelization process by augmenting a serial program with granularity and data usage information.[1] The Jade implementation uses this information to automatically extract the concurrency and map the resulting parallel computation onto the machine at hand. This approach eliminates many of the problems that complicate the development of parallel software. Jade programs port without modification between shared-memory machines, message-passing machines, and even heterogeneous collections of workstations. The Jade implementation, and not the programmer, is responsible for extracting the concurrency, correctly synchronizing the parallel execution, and, on message-passing machines, generating the communication required to execute the program. Finally, because Jade preserves the semantics of the original serial program, Jade programs execute deterministically.

The remainder of the article is structured as follows. Section 2 presents the concepts and constructs of Jade. Section 3 discusses the advantages and limitations of Jade. Section 4 presents the reasoning behind the Jade language design and discusses how the concrete Jade constructs relate to the language design rationale. Section 5 describes the Jade implementation. Section 6 describes our experience with Jade applications. We discuss related work in Section 7 and conclude in Section 8.

2. THE JADE LANGUAGE

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

2.1 Fundamental Concepts

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

[1]Jade is currently implemented as an extension to C. Identical implementations are available at the first author's home page, http://www.cag.lcs.mit.edu/~rinard and at the SUIF home page, http://suif.stanford.edu; another implementation is available at the SAM home page, http://suif.stanford.edu/~scales/sam.html.

2.1.1 *Shared Objects.* Jade supports the abstraction of a single mutable memory that all parts of the computation can access. Each piece of data allocated in this memory is called a shared object. The programmer therefore implicitly aggregates the individual words of memory into larger granularity objects by allocating data at a certain granularity. No piece of memory can be part of more than one object.[2]

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

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

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

This execution strategy preserves the relative order of reads and writes to each shared object, which guarantees that the program preserves the semantics of the original program. One important consequence of this property is that Jade programs execute deterministically.

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

When a task can legally execute, the Jade implementation assigns the task to a

[2]The question may arise how this concept interacts with Fortran constructs such as COMMON or EQUIVALENCE. COMMON variables are a natural candidate for shared objects. EQUIVALENCE declarations would interact poorly with the concept of shared objects because different variables from different equivalence statements would be allocated in the same memory.

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

double shared *p;
```

Fig. 1. Jade shared object declarations.

processor for execution. In the message-passing implementation, the processor on which the task will execute issues messages requesting the remote shared objects that the task will access. The implementation then moves (on a write access) or copies (on a read access) the objects to that processor. When all of the remote shared objects arrive, the implementation executes the task. When the task finishes, the implementation may enable other tasks for execution.

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

2.2 Basic Jade Constructs

In this section we describe the basic constructs Jade programmers use to create and manipulate tasks, shared objects, and access specifications.

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

Jade uses the `shared` keyword to identify shared objects. Figure 1 shows how to declare a shared double, a shared array of doubles, and a shared structure. Programmers access objects using pointers. Figure 1 also shows how to use the `shared` keyword to declare a pointer to a shared object; the `shared` is inserted before the `*` in the pointer declaration.

It is possible for shared objects to, in turn, contain pointers to other shared objects. To declare pointers to such objects, the programmer may need to use several instances of the `shared` keyword in a given declaration. Figure 2 contains the declaration of a pointer to a shared pointer to a shared double and the declaration of a shared structure which contains an array of pointers to shared vectors of doubles. Figure 3 presents a picture of these data structures. Even though `q` points to a shared object, `q` itself is a private object in the sense that only the task that declares `q` can access `q`.

Programmers can allocate shared objects dynamically using the `create_object` construct, which takes as parameters the type of data in the object and the number of elements of that type to allocate. If there is only one element in the object, the programmer can omit the parameter. The `create_object` construct returns a

```
double shared * shared *q;

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

Fig. 2. More Jade shared object declarations.

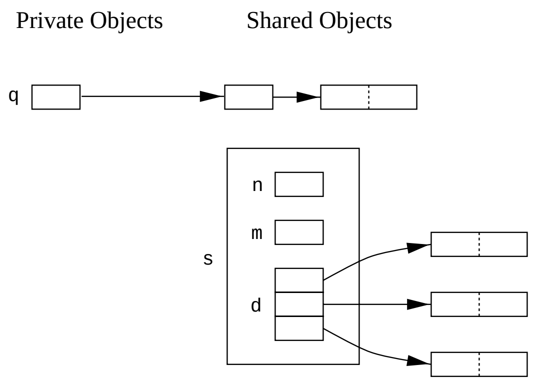


Fig. 3. Data structures for shared object declarations.

pointer to the allocated object. Figure 4 contains a simple example.

Programmers can also implicitly allocate a shared object by declaring a global variable to be shared. While procedure parameters or local variables may point to shared objects, it is illegal for the parameter or variable itself to be shared. It is therefore impossible to allocate shared objects on the procedure invocation stack. In Figure 1, `x`, `A`, and `s` must be global.

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

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

We did not consider making Jade garbage-collected because the underlying language, C, does not provide a data model that supports garbage collection.

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

Fig. 4. Jade shared object creation.

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

Fig. 5. Jade part object declarations.

2.3 Part Objects

In some situations the natural allocation granularity of the data may be finer than the desired shared object granularity in the parallel computation. For example, the Jade sparse Cholesky factorization algorithm in Section 2.4.2 manipulates a data structure that contains pointers to several dynamically allocated index arrays. In the parallel computation, the desired shared object granularity is the data structure plus the index arrays. In such situations Jade allows programmers to aggregate multiple allocation units into a single shared object. The programmer creates such objects by declaring that some of the objects to which a shared object points are part of that object. As Figure 5 illustrates, the programmer declares such pointers using the `part` keyword.

Programmers dynamically allocate part objects using the `create_part_object` construct. The first parameter is a pointer to the shared object that the part object is part of. The second and third parameters are the type and number of data items in the part object. Figure 6 contains an example that illustrates how to allocate part objects. Programmers are also responsible for deallocating part objects when they are done with them; Jade provides the `destroy_part_object` construct for this purpose.

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

The mechanism is that the programmer can declare a pointer to a shared object and restrict how the program will access data using that pointer. Such a pointer is called a local pointer; Figure 7 contains several examples which demonstrate how to declare local pointers.

In Figure 7, the program can only read shared objects via `rp`, write shared objects via `wp`, and read and/or write objects via `rwp`. The implementation statically

```
matrix.row_index = create_part_object(&matrix, int, N);
matrix.start_row_index = create_part_object(&matrix, int, 200);
destroy_part_object(matrix.row_index);
```

Fig. 6. Jade part object creation and destruction.

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

Fig. 7. Jade local pointer declarations.

enforces these access restrictions. It performs a dynamic check when the program sets the local pointer to point to a shared object. The code fragment in Figure 8 illustrates which accesses are checked statically and which are checked dynamically. Figure 9 presents a picture of the data structures.

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

It is impossible for one task to use another task's local pointer. The implementation enforces this restriction by forbidding shared objects to contain local pointers and forbidding a task to pass one of its local pointers as a parameter to a child task. (Section 2.3.3 explains how tasks pass parameters to child tasks.)

It is important to keep the concepts of shared and local pointers separate. Each execution of a `create_object` construct returns a shared pointer. The only other way to obtain a shared pointer is to apply the `&` operator to a shared global variable. When applied to shared objects (with the exception for global variables noted above), the `&` operation yields local pointers. When applied to shared pointers, the pointer arithmetic operations yield local pointers (the implementation uses the surrounding context to determine the appropriate access restriction). The code fragment in Figure 10 illustrates these concepts.

A shared pointer always points to the first element of a shared object, but a local pointer can point to any element of a shared object. Shared pointers can travel across task boundaries, but no task can access another task's local pointers. It is illegal to store local pointers in shared objects. Figure 11 presents several declarations that are illegal because of this restriction.

2.3.2 *Private Objects.* Jade programs may also manipulate pieces of data that only one task may access. Such pieces of data are called private objects. All pieces of data allocated on the procedure call stack are private objects. Jade also provides a memory allocation construct, `new_mem`, that programmers can use to dynamically allocate private objects. `new_mem` has the same calling interface as the C `malloc`

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

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

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

Fig. 8. Local pointer usage.

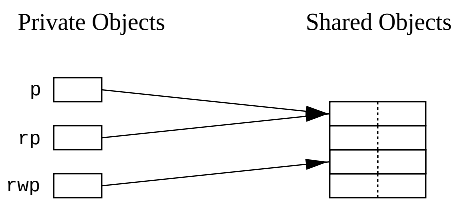


Fig. 9. Data structures for local pointer usage.

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

Fig. 10. Shared and local pointers.

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

Fig. 11. Illegal declarations.

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

Fig. 12. More illegal declarations.

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

Fig. 13. Legal declarations.

routine, taking as a parameter the size of the allocated private object. There is also the `old_mem` routine for deallocating private data; it has the same calling interface as the C `free` routine.[3] Programmers declare variables that deal with private objects using the normal C variable declaration syntax.

The Jade implementation enforces the restriction that no task may access another task's private objects. The implementation enforces this restriction in part by requiring that no shared object contain a pointer to a private object. The declarations in Figure 12 are therefore illegal, because they declare shared objects that contain pointers to private objects.

Of course, it is possible for private objects to contain pointers to shared objects. The declarations in Figure 13 are legal, because they declare private objects that contain pointers to shared objects.

2.3.3 *The* `withonly` *Construct.*

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

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

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

The `task body` section executes in a naming environment separate from the enclosing naming environment. The `parameters` section exists to transfer values from the enclosing environment to the task. The `parameters` section itself is a

[3]We chose the names `new_mem` and `old_mem` rather than `malloc` and `free` to emphasize the semantic difference between private objects and shared objects. `malloc` and `free` allocate and free data that any part of the program can access. `new_mem` and `old_mem` allocate and free data that only the allocating task can access.

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

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

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

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

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

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

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

2.4 A Programming Example

In this section we show how to use Jade to parallelize a sparse Cholesky factorization algorithm. This algorithm performs the numerical factorization of a sparse,

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

Fig. 14. Data structure declarations for serial sparse Cholesky factorization.

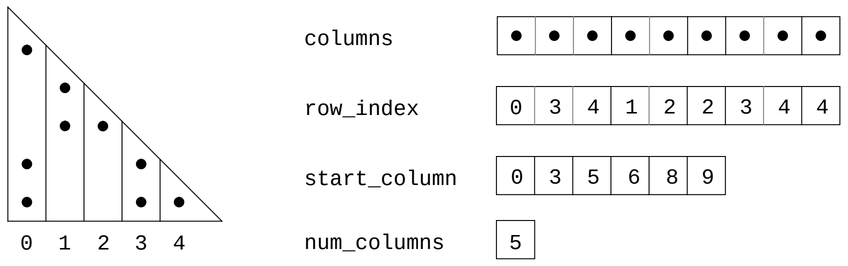


Fig. 15. Data structures for serial sparse Cholesky factorization.

symmetric, positive-definite matrix. The algorithm runs after a symbolic factorization phase has determined the structure of the final factored matrix [Rothberg 1993]. The example illustrates how to use Jade's object and `withonly` constructs, and it demonstrates how Jade programs can exploit dynamic concurrency.

2.4.1 *The Serial Algorithm.* The serial algorithm stores the matrix using the data structures declared in Figure 14. Figure 15 shows a sample sparse matrix and the corresponding data structures. Because the matrix is symmetric, the algorithm only needs to store its lower triangle. The factorization algorithm repeatedly updates the data structures that represent this lower triangle.

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

Figure 16 contains the serial code for this algorithm. The algorithm processes the columns of the matrix from left to right. It first performs an internal update on the current column. This update reads and writes the current column, bringing it to its final value in the computation. The algorithm then uses the current column to update some subset of the columns to its right. For a dense matrix, the algorithm would update all of the columns to the right of the current column. For sparse matrices, the algorithm omits some of these updates because they would not change the updated column.

2.4.2 *The Jade Algorithm.* The first step in parallelizing a program using Jade is to determine the appropriate data granularity. In this case the programmer decides that the parallel computation will access the matrix at the granularity of the individual columns. The programmer must therefore decompose the `columns`

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

Fig. 16. Serial sparse Cholesky factorization algorithm.

array so that each column is stored in a different shared object. The new matrix is structured as an array of column objects. The programmer also decides that the parallel computation will access the structuring data (the `num_columns`, `row_index`, and `start_column` data structures) as a unit. The programmer allocates these data structures as part objects of a single matrix object. Figure 17 gives the new data structure declarations, while Figure 18 shows the sample matrix and the new data structures.

The programmer next modifies the `InternalUpdate` and `ExternalUpdate` routines to use the new matrix data structure and then inserts the `withonly` constructs that identify each update as a task and specify how each task accesses data. Figure 19 contains the new `factor` routine.

2.4.3 *Dynamic Behavior.* Conceptually, the execution of the `factor` routine on our sample matrix generates the task graph in Figure 20. When the program executes, the main task creates the internal and external update tasks as it executes the body of the `factor` procedure. When the implementation creates each task, it first executes the task's `access specification` section to determine how the task will access data. It is this dynamic determination of tasks' access specifications that allows programmers to express dynamic, data-dependent concurrency patterns. Given the access specification, the implementation next determines if the task can legally execute or if the task must wait for other tasks to complete. The implementation maintains a pool of executable tasks, and dynamically load balances the computation by assigning these tasks to processors as they become idle. In a message-passing environment the implementation also generates the messages required to move or copy the columns between processors so that each task accesses the correct version of each column. As tasks complete, other tasks become legally executable and join the pool of executable tasks. In effect, the Jade implementation dynamically interprets the high-level task structure of the program to detect and exploit the concurrency.

2.4.4 *Modularity.* The sparse Cholesky factorization example illustrates how Jade supports the development of modular programs that execute concurrently. Each access specification only contains local information about how its task accesses

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

Fig. 17. Data structure declarations for Jade sparse Cholesky factorization.

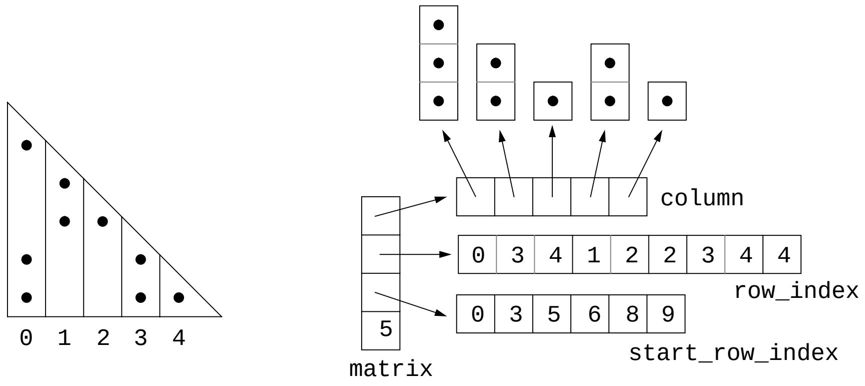


Fig. 18. Data structures for Jade sparse Cholesky factorization.

data—each task is independent of all other tasks in the program. Even though the tasks must interact with each other to correctly synchronize the computation, the Jade implementation, and not the programmer, automatically generates the synchronization using the access specifications and the original serial execution order.

2.4.5 *Comparison with Task-Scheduling Systems.* It is worth comparing the Jade execution model to the execution model in other systems that have been used to solve the sparse Cholesky factorization problem. One aspect of this problem is that it is possible to construct the task graph for the entire factorization before the execution of the factorization begins. In fact, the symbolic factorization must derive information equivalent to this task graph before it can determine the structure of the final factored matrix.

Several researchers have built two-phase systems that schedule a programmer-provided task graph, then execute the scheduled task computation [Dongarra and Sorensen 1987; Fu and Yang 1997]. Because these systems have the entire task graph available when they schedule the computation, they may be able to generate very efficient schedules. They also eliminate scheduling overhead during the execution of the computation. The primary limitation is that these systems are not suitable for problems whose task graph structure depends on the results of computation performed in previous tasks. They also force the programmer to extract the

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

Fig. 19. Jade sparse Cholesky factorization algorithm.

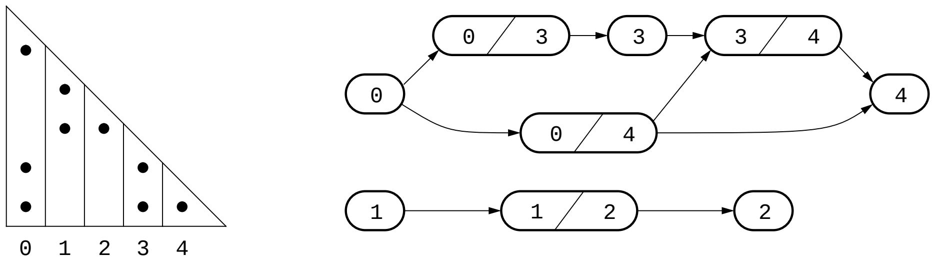


Fig. 20. Task graph for Jade sparse Cholesky factorization.

parallelism and present the parallelism to the scheduling system in the form of a task graph. Jade's online approach provides more programmer support (the Jade implementation, not the programmer, extracts the concurrency) and is suitable for a more general class of problems. The cost of this generality is a potentially less efficient schedule and scheduling overhead during the execution of the task graph.

2.5 Programmer Responsibilities

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

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

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

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

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

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

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

2.6 Advanced Constructs

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

2.6.1 *Task Boundary Synchronization.* In the model of parallel computation described so far, all synchronization takes place at task boundaries. A task does not start its execution until it can legally perform all of the accesses that it will ever perform. It does not give up the right to perform any of these accesses until it completes. This form of synchronization wastes concurrency in two cases: when a task's first access to an object occurs long after it starts, and when a task's last access to an object occurs long before it finishes. Figure 21 contains an example of both kinds of unnecessary synchronization (assuming that `p`, `q`, `r`, and `s` all point to different objects).

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

To support this kind of pipelining concurrency, Jade provides a more elaborate notion of access specification, several new access specification statements, and a construct, the `with` construct, that programmers can use to update a task's access specification as the task executes. Together, these enhancements allow programmers to more precisely specify when tasks perform their individual accesses to shared objects. This additional timing information may expose additional concurrency by enabling independent parts of tasks to execute concurrently even if dynamic data dependence constraints exist between other parts of the tasks. In

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

Fig. 21. Task boundary synchronization example.

the example in Figure 21, the programmer can use the new access specification statements and the `with` construct to expose the available pipelining concurrency.

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

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

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

2.6.3 *Advanced Access Specifications.* An access specification is a set of access declarations; each declaration declares how a task will access a given object. Declarations come in two flavors: immediate declarations and deferred declarations. An immediate declaration gives the task the right to access the object. The basic access specification statements described in Section 2.3.5 generate immediate declarations. A deferred declaration does not give the task the right to access the object. Rather, it gives the task the right to change the deferred declaration to an immediate declaration, and to then access the object. The deferred access specification statements described in Section 2.6.4 generate deferred access declarations.

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

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

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

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

—`df_rd(o)`: Specifies a deferred read declaration.

—`df_wr(o)`: Specifies a deferred write declaration.

—`df_de(o)`: Specifies a deferred deallocate declaration.

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

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

—`no_rd(o)`: Eliminates read declarations.

—`no_wr(o)`: Eliminates write declarations.

—`no_de(o)`: Eliminates deallocate declarations.

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

2.6.6 *Pipelined Concurrency.* The programmer can use the `with` construct and the advanced access specification statements to exploit the pipelining concurrency

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

Fig. 22. Pipelined concurrency example.

available in the example in Figure 21. The programmer first uses the `df_rd(p)` access specification statement to inform the implementation that the second task may eventually read `p`, but that it will not do so immediately. This gives the implementation the information it needs to overlap the execution of the first task with the statement `*q = g(d)` from the second task. When the second task needs to read `p`, it uses a `with` construct and the `rd(p)` access specification statement to convert the deferred declaration to an immediate declaration. At the same time, the `with` construct uses the `no_wr(q)` statement to declare that the second task will no longer write `q`. This gives the implementation the information it needs to overlap the execution of the third task and the statement `*r = h(*q, *p)` from the second task. Figure 22 contains the final version of the program.

3. DISCUSSION

In any programming language design there is a trade-off between the range of computations that the language can express and how well the language supports its target programming paradigm. Jade enforces high-level abstractions that provide a safe, portable programming model for a focused set of applications. Tailoring the language for this application set limits the range of applications that Jade supports. In this section we discuss the scope of Jade, the advantages of using Jade in its target application domain, and the limitations that supporting this domain imposes.

3.1 Scope

We designed Jade as a focused language tailored for a specific kind of concurrency. Within its intended domain it provides a safe, effective programming environment that harmonizes with the programmer's needs and abilities. We next describe the kind of concurrency that we designed Jade to exploit.

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

Jade's high-level abstractions hide certain aspects of the underlying parallel computing environment. While hiding these aspects protects the programmer from much of the complexity of explicit concurrency, it also narrows the scope of the language by denying the programmer control over the corresponding aspects of the parallel computation. It is therefore inappropriate to use Jade for programs that demand highly optimized, application-specific synchronization and communication algorithms.

Jade is designed to express deterministic parallel computations. There are, however, task-level parallel computations (for example, many parallel search and branch-and-bound algorithms) that are inherently nondeterministic. It would be possible to extend Jade for such applications by providing access declarations that allowed programmers to express how these nondeterministic computations accessed data.

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

Because Jade is designed as an extension to an existing sequential language, the question may arise as to whether Jade is intended primarily as a tool for porting existing sequential programs to new parallel machines. While Jade may be useful as a porting tool for existing programs, we believe that, within its application domain, Jade's fundamental advantages over explicitly parallel languages make it entirely suitable for the development of new parallel programs. In fact, we designed Jade primarily for this purpose.

3.2 Advantages

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

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

Serial programming languages promote modularity because they allow the programmer to focus on the dynamic behavior of the current piece of code. Jade preserves the modularity advantages of serial languages because programmers only provide local information about how each task accesses data. There is no code in a Jade program that manages the interactions between tasks. Programmers can function effectively with a detailed understanding of only a subset of the program; changes are confined to the modified tasks. Jade's locality properties also allow programmers to develop the abstractions required for modular parallel programming. For example, Jade programmers can build abstract data types that completely encapsulate the Jade constructs required to guide the parallelization process [Rinard 1994b].

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

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

Finally, Jade programs are portable. We have implemented Jade as an extension to C on a wide variety of computational environments: uniprocessors, shared-memory multiprocessors, distributed-memory multiprocessors, message-passing machines and heterogeneous collections of workstations. Jade programs port without modification between all of these platforms.

3.3 Limitations

Jade's enforced abstractions maximize the safety and portability of Jade programs but prevent the programmer from accessing the full functionality of the parallel machine. This lack of control limits the programmer's ability to optimize the parallel execution. All Jade programs must use the general-purpose concurrency management algorithms encapsulated inside the Jade implementation. Because Jade denies the programmer control over many aspects of the process of exploiting concurrency, programmers cannot use highly optimized, application-specific synchronization, communication or task management algorithms. [4]

[4]It would be straightforward to allow programmers to specify a customized synchronization protocol for each object or class of objects—the Jade implementation uses a table-driven synchronization algorithm that would be trivial to extend for objects with different synchronization requirements. In the message-passing implementation it would also be straightforward to allow programmers to provide a customized communication package for each object or class of objects. The programmer could use such a package, for example, to transfer only those parts of an object that a specific task will access. With the current Jade implementation it would be straightforward to provide an

In some cases the programmer may need to use a lower-level programming system to make the application perform acceptably.

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

Consider, for example, a standard iterative grid relaxation algorithm such as Successive Over Relaxation (SOR) [Golub and Loan 1989]. A programmer using an explicitly parallel language could parallelize the algorithm by subdividing the grid into blocks and assigning one task to each block. At every iteration each task would communicate with its neighbor tasks to acquire the latest values of the boundary elements, then recompute the values of the elements in its block. The tasks would continue their computation, interacting until the algorithm converged [Singh and Hennessy 1992].

In this parallel computation data flows cyclically through the tasks—over the course of the computation each task both generates data that its neighbor tasks read and reads data generated by its neighbor tasks. Jade's serial semantics, however, means that if one Jade task produces data that another Jade task reads, the first task cannot also read data produced by the other task. To express computations such as the parallel SOR computation described above in Jade, the programmer must create one Jade task per block at every iteration, instead of one task per block for the entire computation as in the explicitly parallel program. While it may be more convenient to express the program in Jade (the Jade program is closer to the original serial program), the additional task management overhead may impair the performance of the Jade program if the task size is small relative to the task management overhead. The task model of Cilk [Blumofe et al. 1995] also precludes cyclic data flow between tasks, so similar issues would arise in a Cilk version of SOR and similar applications.

Some parallel algorithms dynamically adjust their behavior to adapt to the varying relative execution times characteristic of parallel computation. The tasks in a parallel branch and bound search algorithm, for example, may visit different parts of the search tree in different executions depending on how fast the bound is updated. Such algorithms nondeterministically access different pieces of data in different executions. In some cases the program itself may generate different results. Because Jade's abstractions are designed to support deterministic computations, it may be impossible to express such algorithms in the current version of Jade. In

application program interface (API) that would allow programmers to customize both the synchronization and communication protocols. Programmers could use this API without accessing the source code of the Jade implementation. It would, of course, be the responsibility of the implementor of the customized synchronization or communication protocol to ensure the preservation of the serial semantics. It would be much more difficult to customize the task management algorithms. These are imbedded deeply into the Jade implementation, with no obvious way for the programmer to extend the functionality without changing the Jade implementation.

many cases, however, the nondeterminism arises from the way the different parts of the computation access data. It would be possible to support many of these computations by extending Jade to support the expression of their nondeterministic data access patterns.

4. LANGUAGE DESIGN RATIONALE

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

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

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

—**Safety:** Jade programs must not exhibit the complex failure modes (such as deadlock and livelock) that characterize explicitly parallel programming. Jade must preserve the serial semantics and provide guaranteed deterministic execution.

—**Modularity:** Jade must preserve the modularity benefits of the sequential programming paradigm.

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

—**Efficiency:** Because Jade was designed to exploit coarse-grain, task-level concurrency, we expected the dynamic task management overhead to be profitably amortized by the large task size. The important design goal was to minimize (and hopefully eliminate) any performance overhead imposed on the sequential computation of each task. The resulting design imposes almost no such overhead.

—**Programmability:** We wished to minimize the amount of programmer effort required to express a computation in Jade.

—**Expressive Power:** We wished to maximize the range of supported parallel applications. In particular, the programmer had to be able to express dynamic concurrency.

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

4.1 Implicit Concurrency

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

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

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

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

4.2 Task Model

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

4.3 Access Specifications

Jade requires the programmer to provide a specification of how each task will access data. The alternative would be to generate the `access specification` sections of the tasks automatically. This would have required the construction of sophisticated program analysis software, which would have significantly complicated the implementation of Jade. Given the programmer's high-level, application-specific knowledge, it is reasonable to assume that in general the programmer could generate more efficient and precise access specifications than an automatic program analysis tool. Experience with Jade applications demonstrates that, given an appropriate shared object structure, the programmer can easily generate the access specifications.

Access specifications give the implementation advance notice of precisely which objects a task will access. The implementation can exploit this information to apply communication optimizations such as concurrently fetching multiple remote objects for each task. Many other parallel systems [Bennett et al. 1990; Bershad et al. 1993; Gelernter 1985; Li 1986] only discover when tasks will access data as the tasks actually perform the access, and lack the advance information required to apply sophisticated communication optimizations.

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

4.4 Local Pointers

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

4.5 Pointer Restrictions

In Jade, shared objects cannot contain local or private pointers. The restriction on private pointers exists to prevent one task from accessing another task's private objects. The restriction on local pointers exists to prevent one task from using another task's dynamic access checks. Recall that all references to local pointers are statically checked. Consider the following scenario. One task acquires a local pointer (generating a dynamic access check) and stores it in a shared object. Another task comes along, reads the shared object to get the local pointer and dereferences the local pointer. The access via the local pointer goes unchecked, and there would be a hole in the access checking mechanism. The restriction prevents this scenario.

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

4.6 Shared and Private Data

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

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

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

4.7 Allocation Units and Access Specifications

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

An early version of Jade [Lam and Rinard 1991] avoided this problem by decoupling the units of allocation and synchronization. The units of synchronization (called tokens) were abstract, and their correspondence with the actual data was completely conceptual. The lack of an enforced correspondence dramatically increased the flexibility of the language. Programmers started dynamically modifying the access declaration granularity of particular pieces of data, using tokens to represent abstract concepts like the right to change the access declaration granularity.

In some respects using the old version of Jade imposed less of a programming burden than using the current version because the programmer never had to change the way the program allocated and accessed data. A major drawback of the old version was that it did not enforce the access specifications. Because the implementation was not aware of the correspondence between data and tokens, it could not check the correctness of the data usage information. Another restriction associated with the lack of an explicit correspondence between data and tokens was that the implementation could not determine which pieces of data each task would access just by looking at its access specification. The implementation could not automatically implement the abstraction of a single address space on message-passing machines by transferring the data at the access declaration granularity in response to each task's access specification.

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

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

It is worth noting that standard flat shared memory systems decouple the units of allocation and synchronization [Amza et al. 1996; Lenoski 1992; Shoinas et al. 1994]. Communication takes place using a flat shared address space, and synchronization takes place using locks and barriers. The locks and barriers are not explicitly coupled to any memory location, although of course such a coupling exists implicitly if the program is correctly synchronized. These systems provide the flexibility that comes from decoupling the data from the units of synchronization, but leave the program open to errors that come from unsynchronized accesses to shared data.

4.8 Allocation Units and Communication

In the message-passing implementation each object is a unit of communication. Transferring the entire object wastes bandwidth when a task actually accesses only a small part of the transferred object. Making each object a unit of communication also requires that each object be fully resident in the accessing processor's memory module. Machines without virtual memory cannot execute programs that access objects bigger than the physical memory associated with each processor. One of the Jade applications (the Volume Rendering application) actually fails to run on one platform because of this restriction. These problems highlight the advantages of systems that separate the units of allocation from the units of communication [Scales et al. 1994; Shoinas et al. 1994].

One alternative is to distribute fixed-size pieces of objects across the memory modules and transfer pieces of objects on demand as processors access them. The shared-memory implementation does this implicitly by using the shared-memory hardware, which distributes pieces of objects across the caches at the granularity of cache lines. Page-based software shared-memory systems apply this principle at the granularity of pages, using the page fault mechanism to detect references to nonresident pieces of objects. Another strategy would have the front end augment every access to a shared object with a software check to see if the accessed piece is available locally. If not, the implementation would automatically fetch the accessed piece from a remote memory module. These strategies would allow the system to use the whole memory of the computing environment to store large objects. They might also drive down the amount of wasted communication bandwidth.

While each of these strategies addresses a fundamental shortcoming of the current Jade communication strategy, they all have drawbacks. Page-based approaches require the implementation to interact with the paging system of the resident operating system. In many operating systems the implementations of the user-level fault handling primitives are inefficient [Appel and Li 1991], and some operating systems do not provide these primitives at all. On the other hand, using a strategy that dynamically checked each access would impose substantial overhead on each access to a shared object.

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

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

The disadvantage of this approach is that it wastes parts of the address space. On machines with no support for sparse address spaces the unaccessed pieces of the object could occupy a large part of physical memory, causing poor utilization of physical memory. On systems that support sparse address spaces this is less of a concern because the pages holding the unaccessed section of the object would remain unmapped and not occupy physical memory. Even these systems could suffer from internal page fragmentation if the communication units were significantly smaller than the page size or if the communication units did not occupy contiguous parts of the object. In a block decomposition of a matrix, for example, each block occupies a noncontiguous part of the matrix's index space.

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

4.9 Language Evolution

We implemented two versions of Jade: Version 1 [Lam and Rinard 1991] and Version 2 [Rinard and Lam 1992]. As described in Section 4.7, in Version 1 the connection between the data and the units of synchronization was completely conceptual. This design made the language very flexible and minimized the effort required to develop Jade programs starting from existing serial programs. But it made it difficult (if not impossible) to implement Jade on message-passing machines and to verify that Jade programs did not violate their access specifications.

The primary design goal of Version 2 was to eliminate these two limitations of Version 1. We therefore developed the Jade object model as described in Section 2.2.1 and redesigned the constructs of the language so that access specifications are given in terms of shared objects. This design change in turn drove a development of the type system to better support shared objects. We developed the concept of part objects (see Section 2.3) to give the programmer more flexibility when allocating shared objects. We developed the concept of local pointers (see Section 2.3.1) to allow the programmer to avoid the access checking overhead on every access.

The other change between Version 1 and Version 2 was a movement of functionality from the Jade constructs to the access declaration statements. Version 1 had two task creation constructs, `withth` and `withonly`, two access declaration statements, `rd(o)` and `wr(o)`, and two constructs for updating a task's access specification, `with` and `without`. The Version 1 `withth` had the same semantics as the Version 2 `withonly`, with all of the access declaration statements interpreted as immediate access declaration statements. The Version 1 `withonly` had the same semantics as the Version 2 `withonly`, with all of the access declaration statements interpreted as deferred access declaration statements. The Version 1 `with` had the same semantics as the Version 2 `with`, with all of the access declaration statements interpreted as immediate access declaration statements. The Version 1 `without` had the same semantics as the Version 2 `with`, with all of the access declaration statements interpreted as negative access declaration statements.

The construct change from Version 1 to Version 2 eliminated several limitations of the Version 1 constructs. In Version 1, all tasks started out with either all immediate access declarations or all deferred access declarations. The new access declaration statements in Version 2 allowed programmers to create tasks with a mixture of immediate and deferred access declarations. Several of the benchmark applications described in Section 6 (String, Water, Search, and Volume Rendering) create these kinds of tasks. In Version 1, it was impossible for one task to obtain immediate access to an object, convert the immediate access to deferred access, then convert the deferred access back to an immediate access. All of our benchmark applications contain tasks that use this functionality.

Version 1 was inspired largely by our previous experience with explicitly parallel programs, notably the applications in the SPLASH benchmark suite [Singh et al. 1992]. Version 1 already had all of the key concepts of Jade present. In particular, we anticipated the need to exploit pipelined concurrency very early in the Version 1 design process. The constructs required to express pipelined concurrency were therefore part of Version 1 from the beginning.

We had developed several Version 1 Jade applications by the time we designed Version 2, but the design changes were, by and large, not made in response to specific limitations of Version 1 that we experienced while developing Version 1 applications.[5] They were instead driven by a desire to make it possible to execute Jade programs on message-passing machines, to verify that programs did not violate their access specifications, and to improve the conceptual basis of the language.

5. IMPLEMENTATION

The Jade programmer and the Jade implementation each have particular strengths that are best suited for performing different parts of the process of parallelizing the

[5]The Version 1 applications were developed by several Ph.D. students in the Stanford Computer Science Department, including the first author of this article.

computation. The programmer's strength is providing the high-level, application-specific knowledge required to determine an effective data and computation granularity. The implementation's strength is performing the analysis required to discover parallel tasks, executing the detailed bookkeeping operations required to correctly synchronize the resulting parallel computation, and providing the low-level, machine-specific knowledge required to efficiently map the computation onto the particular parallel machine at hand.

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

We have demonstrated the viability and applicability of these algorithms by implementing Jade on many different computational platforms. Jade implementations currently exist for shared-memory machines such as the Stanford DASH machine [Lenoski 1992] and the Silicon Graphics 4D/340 [Baskett et al. 1988], for message-passing machines such as the Intel iPSC/860 [Berrendorf and Helin 1992], and for heterogeneous networks of workstations. While no implementation currently exists for shared-memory machines with incoherent caches such as the Cray T3D [Arpaci et al. 1995; Karamcheti and Chien 1995], it would be possible to implement Jade on such machines.

5.1 Overview

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

—**Concurrency Detection:** The implementation analyzes access specifications to determine which tasks can execute concurrently without violating the serial semantics.

—**Synchronization:** The implementation synchronizes the parallel computation.

—**Scheduling:** The implementation assigns tasks to processors for execution.

—**Access Checking:** The implementation dynamically checks each task's accesses to ensure that it respects its access specification. If a task violates its access specification, the implementation generates an error.

—**Controlling Excess Concurrency:** The implementation suppresses excessive task creation to avoid overwhelming the parallel machine with tasks.

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

—**Object Management:** The message-passing implementation moves or copies objects between machines as necessary to implement the abstraction of a single address space.

—**Naming:** The message-passing implementation maintains a global name space for shared objects, including an algorithm that locates remote objects.

—**Format Translation:** In heterogeneous environments the implementation performs the format conversion required to correctly transfer data between machines with different data formats.

The scheduling algorithms for both the shared-memory and message-passing implementations contain a dynamic load balancing algorithm [Rinard 1994a].

5.2 Object Queues

We next discuss the mechanism that the Jade implementation uses to extract concurrency and synchronize the computation. We first describe the mechanism that the implementation uses for read and write declarations, then generalize to include deallocate declarations.

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

Immediate write entries are enabled when they reach the front of the queue. Immediate read entries are enabled when there are only read entries before them in the queue. Deferred entries are always enabled. The purpose of deferred entries is to prevent later tasks from executing prematurely. A task is enabled (and can legally execute) when all of its object queue entries are enabled.

When a task is created, it inserts an entry into the queue of each object that it declares it will read or write. The new task inserts its entry just before the parent task's entry. This insertion strategy ensures that tasks' entries appear in the object queues in the sequential execution order.

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

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

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

It is possible to break the artificial object queue serialization using a mechanism that hierarchically numbers object versions. Each task could compute which version of each object it would access given only the version numbers of the objects in the parent task's access declaration. A task would run only when the correct versions of the objects it would access became available. This mechanism would allow multiple parent tasks to create child tasks that accessed the same object with no additional synchronization.

5.2.3 *Correctness of Object Queue Mechanism.* In this section we discuss why the object queue mechanism correctly synchronizes the computation. There are two aspects to the correctness of a synchronization mechanism for Jade programs: (1) at every point of the computation there is at least one task that can execute (i.e., the synchronization mechanism is deadlock free) and (2) no task reads the wrong value from a shared object.

Tasks' entries appear in the object queues in the same order that the tasks would execute in a serial program (the serial entry order property). This is true of the original state of an object queue, which contains one entry from the task that created the object. Subsequent object queue operations preserve the serial entry order property. If a task completes or eliminates an access declaration, the implementation removes its entries from the object queue. The new sequence of entries is then a subsequence of the original sequence of entries, and the property holds.

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

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

To establish that no task reads the wrong value from a shared object, we establish that all parallel executions preserve the relative order of reads and writes to the same object. We establish that if a write occurs before a read to the same object in a sequential execution, the write will occur before the read in all parallel executions. A similar argument establishes that the read will take place before any subsequent (in the sequential execution order) writes. We present a more formal treatment of this argument elsewhere [Rinard and Lam 1992].

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

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

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

All of the ancestors of the writing task between the writing task and the first task must insert write entries into the queue. By the time the reading task is created and inserts its entry into the queue, either the write will have occurred, or there is and will continue to be a write entry before it in the queue until the write occurs. This write entry will prevent the read task from performing the read until the write task performs the write.

5.2.4 *Extensions for Deallocate Declarations.* Deallocate declarations also insert entries into the object queue. A deallocate entry is enabled when it is the first entry in the queue. The correctness condition for deallocations is that all previous tasks that access the object complete their accesses before the object is deallocated. An argument similar to the one in Section 5.2.3 establishes that the object queue mechanism preserves the serial execution order for deallocations relative to other accesses to the same object.

5.3 The Shared-Memory Implementation

In this section we discuss the implementation of Jade for multiprocessors with hardware support for shared memory. Because the hardware implements the Jade abstraction of a single address space, the implementation is only responsible for finding the concurrency, synchronizing the computation, and mapping the tasks onto the processors.

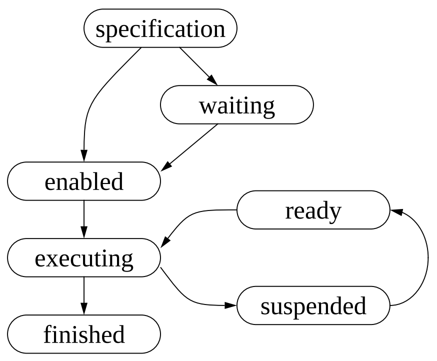


Fig. 23. Shared-memory state-transition diagram.

5.3.1 *A Task Lifetime.* In this section we summarize the dynamic behavior of the shared-memory implementation by tracing the lifetime of a task. Figure 23 gives a state-transition diagram of a task's lifetime. We describe the activities that take place in each state in turn.

—**The Specification State:** To execute a `withonly` construct, the implementation first allocates a task data structure. This data structure contains a pointer to the task body code, space for the task parameters, and an initially empty access specification. The implementation executes the `access specification` section to generate the new task's initial access specification and copies the parameters into the task data structure. It then inserts the task's entries into the object queues according to the algorithm described in Section 5.2. If the task's access specification is enabled, it enters the enabled state. Otherwise, it enters the waiting state.

—**The Waiting State:** A task in the waiting state cannot execute because its initial access specification has yet to be enabled. When the access specification is enabled, the task enters the enabled state.

—**The Enabled State:** A task in the enabled state is ready to execute, but is waiting to be assigned to a processor for execution.

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

—**The Suspended State:** A task in the suspended state will eventually become able to execute again, at which point it enters the ready state.

—**The Ready State:** A task in the ready state is ready to resume its execution. In the current implementation the task will always resume on the processor that first executed it.

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

5.3.2 *Extensions for Incoherent Caches.* In this section we have assumed that the hardware fully implements the abstraction of a single shared address space. Machines with incoherent caches, however, only partially implement this abstraction [Arpaci et al. 1995; Karamcheti and Chien 1995]. These machines automatically fetch and cache remote memory, but rely on software to keep the caches consistent. While no Jade implementation currently exists for machines with incoherent caches, we believe that Jade could be a useful programming language for such machines.

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

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

5.4 The Message-Passing Implementation

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

5.4.1 *A Task Lifetime.* In this section we summarize the functionality of the message-passing implementation by tracing the lifetime of a task. Figure 24 gives a state-transition diagram of a task's lifetime.

—**The Specification State:** When a task is created, it starts out in the specification state. The implementation executes its access specification and copies the parameters into the allocated task data structure. It then inserts the task's entries into the object queues. See Section 5.4.2 for a description of the algorithm that inserts access specifications into remote object queues. If the task's access specification is enabled, it enters the enabled state. If the task must wait to access some of the objects, it enters the waiting state.

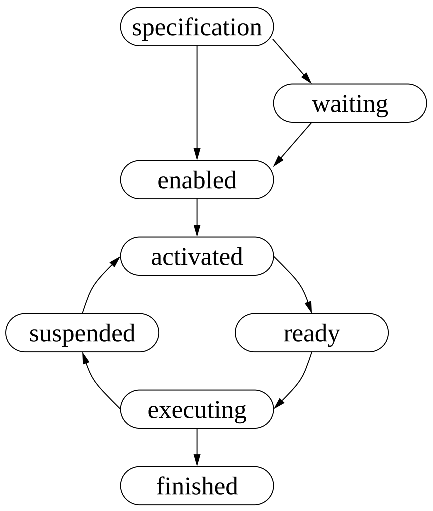


Fig. 24. Message-passing state-transition diagram.

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

—**The Enabled State:** A task is in the enabled state if it has yet to execute, but its initial access specification has been enabled. The scheduler will eventually assign the task to a processor for execution. When the implementation assigns a task to a processor for execution, it packs the task data structure into a message and transfers the task to the executing processor. At this point the task enters the activated state.

—**The Activated State:** An activated task's synchronization constraints have been enabled and the scheduler has assigned the task to a processor for execution. The implementation cannot yet execute the task, however, because the task may need to access some objects that are not locally available. The implementation therefore fetches the nonlocal objects by sending request messages to processors that have locally available copies of the objects. The processors respond by sending a copy of the object to the requesting processors. When all of the messages containing remote objects for the task arrive back at the processor, the task enters the ready state. Section 5.4.5 describes how the implementation determines, for each object, which processor has a locally available copy of that object. Because the implementation replicates mutable objects for concurrent read access, the Jade implementation must solve the consistency problem. Section 5.4.5 presents the details of the Jade consistency mechanism.

—**The Ready State:** Each processor maintains a local queue of tasks that are ready to execute. When the processor goes idle it fetches a task from this queue and executes it. The task then enters the executing state.

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

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

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

5.4.2 *The Object Queue Protocol.* In the current Jade implementation each object queue resides completely on one processor. When a processor must perform an operation on a remote queue, the implementation can either move the queue to the processor and perform the operation locally, or forward the operation to the processor that owns the queue and perform the operation there. The implementation currently moves the object queue only when it inserts a new entry into the queue. The implementation performs the other queue operations remotely. Section 5.4.3 describes the precise algorithm the implementation uses to find a remote queue.

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

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

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

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

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

The implementation recognizes out-of-date messages using sequence numbers. Each object queue contains a counter. The implementation increments this counter every time it performs an operation on the queue or sends a message from the queue to a task. Each message from the queue to a task contains the current value of the counter. For each declaration the task stores the value of the queue counter when the queue last updated that declaration's information. When the declaration gets a message from a queue, it compares the counter in the message to its stored counter value. If the message counter is less than the declaration counter, the message is out of date and is discarded. In the example above, the insertion of the child task's declaration into the object queue would increment the queue counter. The revocation of the right to write the object caused by the creation of the child task would store the new counter value into the parent task's access declaration. The implementation would recognize the message's out-of-date counter value and discard the message.

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

The implementation locates objects using a forwarding pointer scheme. At each processor the implementation maintains a forwarding pointer for each entity that the processor has ever held. This forwarding pointer points to the last processor known to have requested the entity. The implementation locates an entity by following these forwarding pointers until it finds the processor holding the entity. If a processor with no forwarding pointer needs to locate an entity, the implementation extracts the number of the processor that created the entity (this number is encoded in the entity identifier) and forwards the operation or request to this processor (the entity is initially located on that processor).

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

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

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

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

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

The algorithm outlined above is a general-purpose mechanism for locating entities in a message-passing system. It could be used in other systems any time the system must locate any piece of migrating state or information in the system. The algorithm above has the drawback that once a processor has requested an entity, it must maintain a forwarding pointer for that entity for the rest of the computation. In the worst case each processor may have one forwarding pointer for each object in the computation and the forwarding pointers may consume too much memory.

It is possible to adjust the algorithm so that processors may discard forwarding pointers. The system could then discard pointers when memory becomes tight, or it could devote a fixed amount of space to the forwarding pointer table.

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

5.4.4 *The Consistency Problem.* Any system that replicates mutable data must solve the consistency problem. The consistency problem arises when a processor writes a copy of a replicated object, generating a new version of the object. The implementation must then ensure that no processor subsequently reads one of the obsolete copies of the object.[6] Systems traditionally solve the consistency problem using either an invalidate or an update protocol. Systems using invalidate protocols keep track of all outstanding copies of an object. When a write occurs the system sends out messages that eliminate all of the obsolete copies. Update protocols work in a similar way, except that the writing processor generates messages that contain the new version of the object. These messages then overwrite the obsolete copies. At some point the writing processor must stall until it knows that all of the invalidates or updates have been performed. The exact stall point depends on the strength of the consistency protocol. Gharachorloo's thesis contains a more detailed treatment of consistency protocols [Gharachorloo 1996].

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

5.4.5 *The Jade Consistency Mechanism.* The Jade consistency mechanism eliminates some of the performance overhead of invalidate and update protocols. It first tags each copy of an object with a version number. The version number counts the number of times the program wrote the object before it generated that version of the object. For every task the implementation keeps track of which version of each

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

object it must access, and the owner of that version (the processor that generated that version). The first owner of an object is the processor in whose memory the implementation initially allocated the object.

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

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

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

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

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

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

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

The implementation or the user program accesses each of these data structures using globally valid identifiers. The implementation keeps track of the correspondence between globally valid identifiers and the local addresses of the data structures using a table. There is one such table for each processor and each kind of data structure. Each table maps the globally valid identifier of each locally resident data structure to the address of the memory holding that data structure. When the implementation needs to access a locally resident data structure, it uses the table to find the data structure.

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

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

5.5 Common Aspects

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

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

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

5.5.2 *Access Checking.* The implementation performs the dynamic access checks using an access declaration table. To prepare for a task's execution, the processor inserts each of the task's access declarations into a table indexed by the identifiers of the objects that the task declared it would access. There is one table per processor, and the task's declarations are inserted into the table associated with the processor that will execute the task. When the task executes, it performs the access checks by looking up declarations in the access declaration table and checking the accesses against the declarations. When programmers use the local pointer mechanism discussed in Section 2.3.1, the implementation amortizes the lookup cost over many accesses via the local pointer.

5.6 Basic Jade Overheads

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

5.6.1 `withonly` *Time Overhead.* To a first approximation, the time to create and execute a task depends on the number of objects the task declared that it would access and whether a task is executed locally on the same processor that created it or remotely on a different processor. The implementation may take longer to execute a task if there is contention for the internal data structures or if the internal data structures are migrating between processors.

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

The benchmark program serially creates and serially executes many null tasks. We divide the total execution time by the number of tasks to calculate the per-task overhead. On DASH we measure three cases: (1) the tasks execute on the same processor as the creator, (2) the tasks execute on a different processor but within the same cluster as the creator, and (3) the tasks execute on a different cluster from the creator. The benchmark program generates each case by explicitly placing each task on the target processor. We plot the running times in microseconds for these

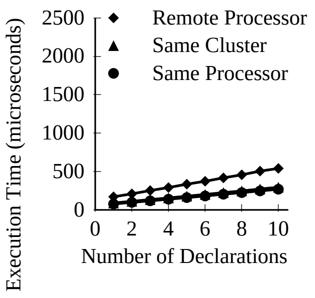


Fig. 25. Task overhead on DASH.

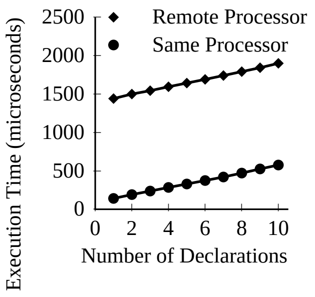


Fig. 26. Task overhead on the iPSC/860.

three different cases in Figure 25. These curves graph the overhead as a function of the number of objects the task declared that it would access. The differences in the running times are caused by memory system effects.

For the iPSC/860 we measure two cases: (1) the tasks execute on the same processor as the creator and (2) the tasks execute on a remote processor. Figure 26 plots the running times in microseconds for these two cases. The remote overhead is substantially larger than the local overhead. We attribute this difference to the message composition and transfer overhead on the iPSC/860.

5.6.2 *Speedup Benchmarks.* The task overhead limits the grain size that Jade implementation can support. We created a benchmark program to measure how the speedup varies with the task size. The program has a sequence of phases; each phase serially creates and in parallel executes tasks of a given size. The sequence of phases varies the task size. The program devotes one processor to creating tasks and the other processors to executing tasks. Figure 27 presents the results of the program running with 32 processors on DASH; Figure 28 presents the results of the program running with 32 processors on the iPSC/860. Each figure plots the measured speedup as a function of the task size in milliseconds.

In the benchmark program each task declares that it will access three objects. Each phase serially creates 31×256 tasks that execute in parallel. The measured

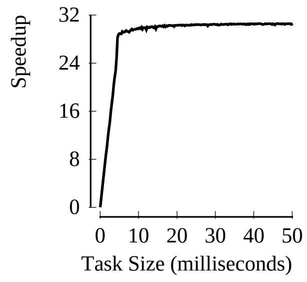


Fig. 27. Speedup on DASH for 32 processors.

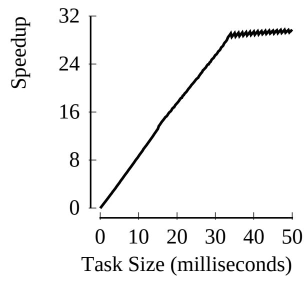


Fig. 28. Speedup on the iPSC/860 for 32 processors.

speedup is the task size times 31×256 divided by the measured execution time to create and execute the tasks.

5.6.3 `with` *Time Overhead.* There is also time overhead associated with executing a `with` construct; to a first approximation the overhead is a function of the number of access specification statements that the `with` construct's `access specification` section executes. Figures 29 and 30 present the overhead in microseconds on DASH and the iPSC/860, respectively.

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

In the message-passing implementation each task incurs an overhead of 800 bytes. Each object incurs a total overhead of 400 bytes, with the object queue taking up 344 bytes and an object header taking up 56 bytes. There is one object header for each replica of the object. Each access declaration takes up 48 bytes associated with the task data structure and 28 bytes associated with the object queue data

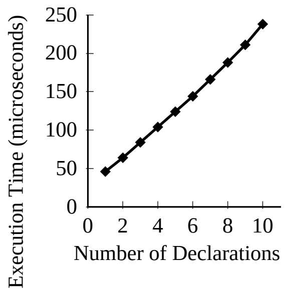


Fig. 29. `with` overhead on DASH.

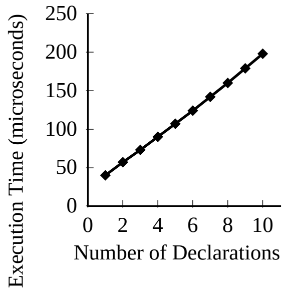


Fig. 30. `with` overhead on the iPSC/860.

structure. Each task and object queue contains space for 10 initial declarations, so declarations do not start taking up space until a task declares that it will access more than 10 objects or until more than 10 tasks at a given point in time declare that they will access one object.

6. APPLICATIONS EXPERIENCE

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

6.1 The Application Set

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

Several factors influenced our choice of applications. One major factor was availability. Existing benchmark sets were one source of applications, and three of our applications originally came from the SPLASH benchmark suite [Singh et al. 1992]. 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 properties of an existing parallel version before deciding to develop a Jade version. For two of our applications, however, the Jade parallelization was the first (and so far the only) parallelization to exist. We next describe each of our applications.

—**Water**: A program that evaluates forces and potentials in a system of water molecules in the liquid state [Woo et al. 1995].

—**String**: A program that computes a velocity model of the geology between two oil wells [Harris et al. 1990].

—**Search**: A program that simulates the interaction of electron beams with solids [Browning et al. 1994; Browning et al. 1995].

—**Volume Rendering**: A program that renders a three-dimensional volume data set for graphical display [Nieh and Levoy 1992].

—**Panel Cholesky**: A program that factors a sparse positive-definite matrix [Rothberg 1993].

—**Ocean**: A program that simulates the role of eddy and boundary currents in influencing large-scale ocean movements [Singh and Hennessy 1992].

To the best of our knowledge, six people have participated in the development of complete Jade applications: the first author of this article, four other Ph.D. students in the Stanford Computer Science Department, and one research scientist in the Stanford Electrical Engineering Department. All of the benchmark applications except Search were developed either by the first author or one of two other Ph.D. students; Search was developed jointly by the first author and the research scientist from the Stanford Electrical Engineering Department.

6.2 Application Characteristics

We next present some basic application properties and use these properties to discuss several aspects of the Jade programming process. Table 1 presents some static properties. A comparison of the number of lines of code in the serial version (when available) with the number of lines of code in the Jade 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.

Table 1. Static Application Characteristics

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 2. Language Feature Usage

Application	Uses Part Objects	Uses Private Objects	Uses Local Pointers	Uses Nested Concurrency	Uses Pipelined Concurrency
Water	no	yes	yes	no	yes
String	yes	yes	yes	no	yes
Search	no	yes	yes	no	no
Volume Rendering	yes	yes	yes	no	yes
Panel Cholesky	yes	yes	yes	no	no
Ocean	no	yes	yes	no	yes

Table 2 shows which language features the applications use. Jade supports a rich object model with a variety of different kinds of objects and pointers. The language feature usage data shows that this richness is justified in practice—all of the different kinds of objects and pointers are used by multiple applications. The applications use less of the control functionality, however. In every Jade application there is a single main thread of control that creates all of the parallel tasks. None of the applications uses nested task creation to generate a tree-like pattern of concurrency. This is an interesting result, especially in light of the fact that other researchers have found it useful to exploit this form of concurrency [Blelloch et al. 1993; Blumofe et al. 1995]. On the other hand, four of the six applications use the pipelined form of concurrency discussed in Section 2.6.6. Three of the applications (Water, String and Volume Rendering) use this form of concurrency in a stylized way to perform a parallel reduction of values generated by a set of parallel tasks.

6.3 Programming Evaluation

As the data presented in the previous section suggest, using Jade did not usually impose an onerous programming burden. For all of our applications, the key to a successful parallelization was determining an appropriate structure for the shared objects. Such a structure was always fairly obvious, given a high-level understanding of the basic source of exploited concurrency. Once the structure was in place, using Jade constructs 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. But 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.

The one exception to this pattern was Ocean. The parallel tasks in Ocean concurrently write disjoint pieces of several key data structures. It is natural to make each of these data structures a single shared object. But because in Jade all synchronization takes place at the granularity of objects, this allocation strategy would serialize the computation. The programmer was therefore forced to explicitly decompose the data structures into multiple objects. This decomposition enabled the concurrent writes necessary to parallelize the application.

Unfortunately, the decomposition forced the programmer to change the data structure indexing algorithm 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. Note that decomposing a data structure to enable concurrent writes does not always generate such a large amount of programming overhead. In Panel Cholesky, for example, the programmer decomposed the data structure used to hold the factored matrix. This decomposition, however, 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.

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.

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 or she 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 important 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.

6.4 Performance Evaluation

We next discuss the performance of the Jade applications. We ran each application on a shared-memory platform (the Stanford DASH machine [Lenoski 1992]) and on a message-passing platform (the Intel iPSC/860 [Berrendorf and Helin 1992]). Table 3 presents some basic performance numbers for the iPSC/860 runs, while Table 4 presents the corresponding results for the DASH runs. Volume Rendering did not run on the iPSC/860—one of its shared objects was too large to fit in the physical memory of any one node of the iPSC/860.

Figures 31 through 36 present the speedup curves for each application. These curves plot the running time of the serial version of the application[7] divided by the running time of the Jade version as a function of the number of processors executing the Jade version.

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 problems for the fine-grained applications are caused in large part by the serial task creation overhead discussed in Section 2.5.2. The scaling problem is especially severe on the iPSC/860. On the iPSC/860 the relatively large message-passing overhead makes it impossible to im-

[7]This serial version was derived from the Jade version by automatically removing all of the Jade constructs to obtain a serial C program. Obviously, this serial version executes with no parallelization overhead. We chose this version rather than the original serial version as the baseline for the speedup curves because it was available for all applications (there is no serial version for Search or Volume Rendering) and because for some of the applications we made performance improvements to the application as part of the Jade conversion process. We have reported the running times of the available serial versions elsewhere [Rinard 1994a]. For all combinations of applications and machines, except Panel Cholesky on the iPSC/860 and DASH and Ocean running on the iPSC/860, the original serial version is slower than the Jade version with all the constructs automatically removed. For Panel Cholesky, the ratio of the running time of the Jade version with all of the constructs automatically removed to the running time of the original serial version is always less than 1.08. For Ocean on the iPSC/860, the ratio of the running time of the Jade version with all of the constructs automatically removed to the running time of the original serial version is less than 1.13.

Table 3. Dynamic Application Characteristics for the iPSC/860

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	0.0020
Ocean	60.99	1.16	0.0033

Table 4. Dynamic Application Characteristics for DASH

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

plement the basic Jade primitives as efficiently as on DASH, which supports much finer-grain communication.

We next discuss the performance of each application in turn. For Panel Cholesky and Ocean running on DASH, we quantify the performance cost of Jade's high-level, portable programming model by comparing the performance of the Jade versions with versions written in lower level languages that run only on shared memory machines.

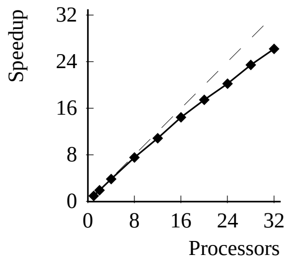
6.5 Water, String, and Search

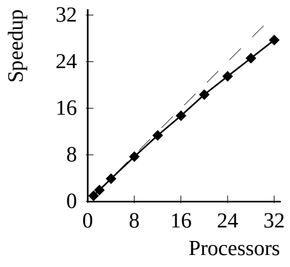
Water, String, and Search all exhibit very good performance, scaling almost perfectly up to 32 processors. The task sizes are large relative to the task creation and communication overhead, which allows the implementation to profitably amortize the overhead to negligible levels, and the load is evenly balanced.

6.6 Volume Rendering

As mentioned above, Volume Rendering does not run on the iPSC/860. The current message-passing implementation of Jade requires an object to be completely resident on a processor if that processor accesses the object. Volume Rendering manipulates a large (approximately 58 Mbytes) object, which does not fit into the 16 Mbyte memory on each node of the iPSC/860. The Jade implementation is therefore unable to allocate space for the 58 Mbyte object that Volume Rendering needs to execute.

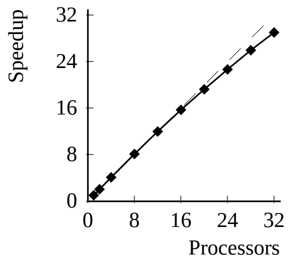
Although Volume Rendering scales reasonably well on DASH, it does not achieve the performance of String, Water, or Search. We explored the performance of the application using *event logs*. We instrumented the Jade implementation to generate a log that records the time when specific events occur during the execution of the application. The logs are stored in memory and written out at the end of the execution. For Volume Rendering, recording the log information does not impose

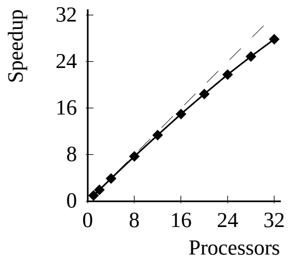




Speedup for Water on the iPSC/860   Speedup for Water on DASH

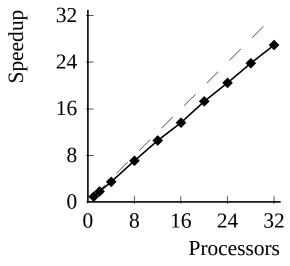
Fig. 31. Speedups for Water.

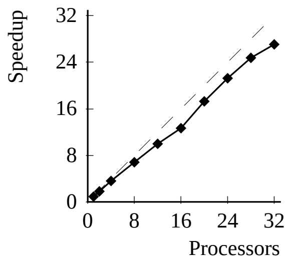




Speedup for String on the iPSC/860   Speedup for String on DASH

Fig. 32. Speedups for String.





Speedup for Search on the iPSC/860   Speedup for Search on DASH

Fig. 33. Speedups for Search.

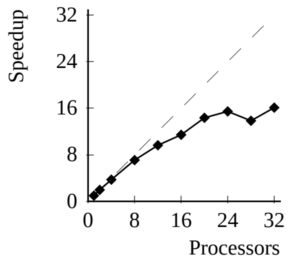
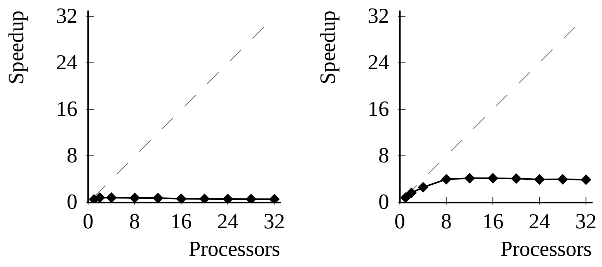
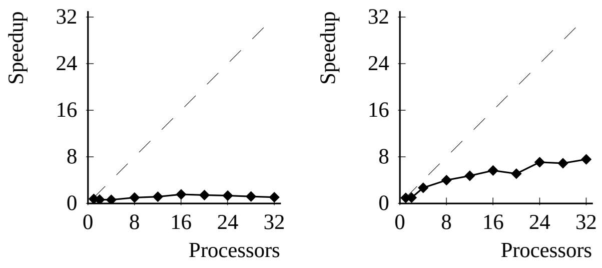


Fig. 34. Speedup for Volume Rendering on DASH.



Speedup for Panel Cholesky on the iPSC/860 Speedup for Panel Cholesky on DASH

Fig. 35. Speedups for Panel Cholesky.



Speedup for Ocean on the iPSC/860 Speedup for Ocean on DASH

Fig. 36. Speedups for Ocean.

a significant performance overhead.

The logs primarily record information that deals with the execution of tasks. For Volume Rendering the relevant events are the creation, execution, and termination of tasks. A correlation of the logs with the structure of the application demonstrates the sources of performance loss. The vast majority of the computation in the application occurs during the primary parallel phase when the application renders the dataset for a new view. At the end of the parallel phase, each task has computed a set of contributions to the final view. The application then executes a serial phase during which each task copies its contributions back into the data structure that stores the view. The tasks write disjoint sections of the view data structure. A separation of allocation and synchronization units as discussed in Section 4.7 would allow the serial copy back phase to execute in parallel. An examination of the event logs for this application running on DASH demonstrates that there are two primary sources of performance loss: idle time during a serial copy back phase and poor load balancing during the primary parallel phase [Rinard 1994a].

6.7 Panel Cholesky

Panel Cholesky performs poorly on the iPSC/860. We attribute the poor performance to the fact that, on this platform, the mean task size is comparable to the task overhead. In this situation parallel execution offers little or no performance benefit.

Panel Cholesky performs much better on DASH than on the iPSC/860, although the computation still does not scale very well—the maximum speedup is 3.8 on 16 processors. For DASH, we can compare the performance of the Jade version of Panel Cholesky with the performance of the highly optimized, explicitly parallel, hand-tuned version in the SPLASH benchmark set [Singh et al. 1992]. The SPLASH version achieves a speedup of 9.8 out of 24 processors, with much of the performance loss coming from an inherent lack of concurrency in the computation [Rothberg 1993]. We attribute the difference in performance between the Jade version and the SPLASH version to the task management overhead in the Jade version and the general scheduling algorithm in the Jade implementation, which generates a less efficient assignment of tasks to processors than does the SPLASH version [Rinard 1994a].

6.8 Ocean

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 solve terminates when the differences between the old values and the new values drop below a given threshold.

Ocean executes a sequence of parallel phases. At the start of each parallel phase, the main thread creates a set of parallel tasks, each of which updates part of the state of the system. When it finishes, each task writes a flag that indicates whether its part of the system has converged. At the end of the parallel phase, the main thread reads all of the flags to determine if the entire solve has converged. For both the iPSC/860 and DASH, the main processor, which executes the main thread, is devoted to task management. On the iPSC/860, we measured the amount of time the main processor spends on task management, and found that, above 8 processors, it spent almost all of its time managing tasks [Rinard 1994a]. We therefore attribute the poor performance on the iPSC/860 to serialized task management on the main processor.

On DASH, serialized task management also has a negative impact on the performance. But there is another factor that has a significant impact—the scheduler generates a suboptimal placement of tasks on processors. The computation suffers from poor locality, and the parallel version takes significantly more time to perform the stencil computation in the application than does the serial version [Rinard 1994a]. On DASH, these two factors make the Jade version perform significantly worse than a version of the same computation written in the explicitly parallel language COOL [Chandra et al. 1993]. The Jade version achieves a maximum speedup of 7.57 on 28 processors, while the COOL version achieves a maximum speedup of 21.06 on 28 processors.

7. RELATED WORK

Researchers have developed an enormous number of parallel programming languages [America 1987; Bal et al. 1992; Burns 1988; Carriero and Gelernter 1989; Chandra et al. 1993; Foster and Taylor 1990; Gregory 1987; Hoare 1985; INMOS Limited 1984; Krishnamurthy et al. 1992; Reppy 1992; Yonezawa et al. 1986]. There is a fundamental difference between Jade and almost all of these languages: Jade is a declarative language used to provide information about how a serial program accesses data, while the vast majority of parallel programming languages are control-oriented languages with constructs that programmers use to directly control the parallel execution. The primary advantage of these languages is that they allow programmers to directly control the parallel execution for maximum efficiency. We have discussed the disadvantages of the control-oriented approach elsewhere in the article (see Section 1).

7.1 Languages for Imperative Programs with Serial Semantics

We next focus on the relatively few languages that are designed to help a compiler or runtime system effectively parallelize a serial, imperative program. Data-parallel languages such as Fortran 90 [Metcalf and Reid 1990] and C* [Rose and Steele 1987] provide a useful paradigm for programs with regular, data-parallel forms of concurrency. Programmers using these languages view their program as a sequence of operations on large aggregate data structures such as sets or arrays. The system can execute each aggregate operation in parallel by performing the operation on the individual elements concurrently. The more general data parallel language NESL supports nested data parallel computations [Blelloch et al. 1993]. NESL preserves the basic spirit of the data parallel paradigm (regular operations on large aggregate data structures) while supporting a more general class of computations on aggregate data structures. The data parallel approach preserves the advantages of the sequential programming paradigm while exposing the concurrency available within operations. It provides a simple interface that is ideal for expressing regular forms of concurrency. Jade is designed to exploit an orthogonal source of concurrency: the task-level concurrency available between operations on different data structures.

Other researchers have also developed languages that allow programmers to provide extra information about how the program structures and accesses data. The goal is to expose more concurrency by improving the precision of the compiler's dependence analysis.

Refined C [Klappholz et al. 1990] and Refined Fortran [Dietz and Klappholz 1986; Klappholz 1989] allow programmers to create sets of variables that refer to disjoint regions of memory. When pieces of code access disjoint subsets of such variables, the compiler can statically verify that they can execute concurrently. Typical operations are creating a set of names that refer to disjoint regions of an array and creating an array of pointers that point to distinct data structures.

ADDS [Hendren et al. 1992] declarations for data structures containing pointers to dynamically allocated data allow programmers to describe the set of data structures that can be reached by following different pointer chains. The compiler combines this information with an analysis of the pointer-chain paths that different parts of the computation follow to derive a precise estimate of how the computation will access data. The improved precision of the dependence analysis can expose additional opportunities for parallel execution.

FX-87 [Gifford et al. 1987; Hammel and Gifford 1988; Lucassen 1987] contains constructs that programmers use to specify how procedures access data. The system statically analyzes the program, using this information to determine which procedure calls can execute concurrently without violating the serial semantics. FX-87 programmers partition the program's data into a finite, statically determined set of regions. The access specification and concurrency detection take place statically at the granularity of regions. The fact that regions are a static concept allows the FX-87 implementation to check the correctness of the access specifications at compile time. But regions also limit the precision of the data usage information. In general, many dynamic objects may be mapped to the same region, limiting the ability of the FX-87 implementation to exploit concurrency available between parts of the program that access disjoint sets of such objects.

The Fx compiler (distinct from FX language discussed in the previous paragraph) exploits programmer annotations to extract task-level parallelism from High Performance Fortran Programs [Gross et al. 1994]. Each task corresponds to a subroutine call; the programmer annotates the subroutine calls with information specifying how the subroutine will read and write its parameters. All communication takes place at subroutine boundaries. To ensure that it can extract the task graph at compile time, the Fx compiler enforces a restricted model of computation. In particular, there can be no conditionals within parallel phases. Jade presents a much more general model of computation. The programmer can create tasks at arbitrary points in the program (not just within conditional-free parallel regions), tasks can access any shared object (not just the parameters), tasks can pipeline their accesses to shared objects, and the task boundaries are decoupled from the subroutine boundaries. The cost for this generality is that the concurrency extraction and task and data movement scheduling all take place at run time and generate run-time overhead. The Fx compiler exploits its restricted model of computation to perform the concurrency extraction and scheduling statically instead of dynamically.

7.2 Parallel Libraries

Specialized libraries that implement a specific set of parallel algorithms provide a simple, easy way to use parallel machines. The numerical analysis community has developed fast parallel implementations of common linear algebra routines [Dayde and Duff 1990]. Other researchers have developed a framework for implementing common data usage patterns [Scales and Lam 1994]. In the best case these systems provide the same advantages as data parallel languages. They preserve the abstraction of sequential execution by encapsulating the parallel computation inside routines invoked from a serial program.

7.3 Functional Languages

Functional languages such as Id [Arvind and Thomas 1981] and Sisal [Feo et al. 1990] support a functional model of computation. From the perspective of someone implementing Id or Sisal on a parallel machine, the important feature of these languages is that they allow programs to define the value of a variable at most once. Programs written in Id, Sisal, and other functional languages can therefore execute in an inherently parallel data-driven fashion in which each computation can execute as soon as the values it uses have been produced. Multilisp futures also support a similar model of computation, although programs that use futures may not execute deterministically if the computations encapsulated in futures imperatively update externally visible data [Halstead 1985].

Despite the fact that Id and Multilisp programs execute deterministically, we do not view them as having a serial semantics. Even on uniprocessors these languages may require conceptually parallel execution [Mohr et al. 1990; Traub 1991].

The primary difference between Jade and functional languages is that Jade supports mutable data—Jade programs can update shared objects multiple times, while programs written in functional languages can define each variable at most once. The elimination of mutual data poses challenging storage management problems for functional languages, and the performance of programs written in these languages has tended to suffer from copying and storage management overhead.

8. CONCLUSION

Developing programming paradigms that allow programmers to effectively deal with the many different kinds of concurrency is a fundamental problem in computer science. The goal of the Jade project was to develop an effective paradigm for a specific purpose: the exploitation of task-level concurrency for performance. The concrete results of this project demonstrate that Jade, with its high-level abstractions of serial semantics and a single address space, satisfies this goal.

We have demonstrated Jade's portability by implementing it on a diverse set of hardware platforms. These machines span the range of computational platforms from tightly coupled shared-memory machines through dedicated homogeneous message-passing multiprocessors to loosely coupled heterogeneous collections of workstations.

We evaluated the Jade language design by implementing several complete scientific and engineering applications in Jade. We obtained excellent performance results for several applications on a variety of hardware platforms with minimal programming overhead. We also obtained less satisfactory results for programs that pushed the limits of the Jade language and implementation. Some applications would work well, given improvements in the implementation; others would be best expressed in other languages.

Because Jade was designed to support a specific, targeted class of computations, it is, by itself, unsuitable as a general-purpose parallel programming language. Jade's enforced abstractions mean that programmers cannot express certain kinds of parallel algorithms in Jade and cannot control the machine at a low level for optimal efficiency. The ultimate impact of the Jade project will come from the integration of basic concepts and implementation techniques from Jade into other programming systems designed to support a wider range of applications. The advantage of developing a focused language like Jade is that it isolates a clear, conceptually elegant definition of the basic paradigm. Using the language therefore both allows and forces programmers to explore the advantages and disadvantages of the paradigm. With the Jade project behind us, we can identify how the basic concepts of Jade are likely to live on in future languages and systems.

A fundamental idea behind Jade is to have programmers declaratively provide information about how the program accesses data. This is in harmony with a long-term trend in computer science to change the focus from control to data. In parallel computing the need to efficiently manage the memory hierarchy for performance will drive this change of focus. Future languages and systems will be increasingly organized around the interaction of data and computation, with various declarative mechanisms, such as access specifications, used to express the relevant information. COOL's locality hints [Chandra et al. 1993], Midway's object usage declarations [Bershad et al. 1993], shared region declarations [Sandu et al. 1993], and the CHICO model of consistency [Hill et al. 1992] are all examples of this trend.

Access specifications give the implementation enough information to automatically generate the communication without forcing the implementation to use a specific communication mechanism. It is therefore possible to implement parallel languages based on access specifications on a wide variety of machines. Each implementation can use the native communication mechanism to implement the underlying abstraction of a single address space, and applications will efficiently port to all of the platforms.

Advance notice of how the program will access data gives the implementation the information it needs to apply locality and communication optimizations appropriate for the target hardware platform. In an explicitly parallel context the implementation can also use access specifications to automatically synchronize the computation.

Access specifications build on the programmer's high-level understanding of the program and mesh with the way the programmer thinks about its behavior. They allow the programmer to express complex parallel computations simply, concisely, and in a way that places minimal demands on the programmer's cognitive abilities. Because access specifications provide so many concrete benefits, we expect them to appear increasingly often in future parallel language designs.

Jade supports the abstraction of a single shared address space with automatically cached data. The programming benefits of this abstraction ensure that many future languages and systems will support this approach (as many existing systems do).

We expect that many such systems will use some form of access specifications to support the automatic generation of communication operations.

One of the unique features of Jade is its extreme portability. Jade currently runs on a wide range of hardware platforms and in principle could be implemented on almost any MIMD computing environment. We designed this portability into the language by scrupulously eliminating any dependences on specific architectural features. The speed with which specific computer systems become obsolete and the need to preserve software investment in parallel programs will drive a trend towards highly portable languages.

ACKNOWLEDGMENTS

We would like to thank Dan Scales, Jennifer Anderson, and Morry Katz for their participation in many Jade language design discussions. Jennifer Anderson implemented the first Jade front end; Dan Scales implemented the second (and final) Jade front end. We would also like to thank the anonymous reviewers for their helpful comments and suggestions.

REFERENCES

America, P. 1987. POOL-T: A parallel object-oriented language. In *Object Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, Eds. MIT Press, Cambridge, Mass., 199–220.

Amza, C., Cox, A., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., and Zwaenepoel, W. 1996. TreadMarks: Shared memory computing on networks of workstations. *IEEE Comput. 29,* 2 (June), 18–28.

Appel, A. and Li, K. 1991. Virtual memory primitives for user programs. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York.

Arpaci, R., Culler, D., Krishnamurthy, A., Steinberg, S., and Yelick, K. 1995. Empirical evaluation of the CRAY-T3D: a compiler perspective. In *Proceedings of the 22nd International Symposium on Computer Architecture*. ACM, New York.

Arvind and Thomas, R. 1981. I-structures: An efficient data type for functional languages. Tech. Rep. MIT/LCS/TM-210, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

Bal, H., Kaashoek, M., and Tanenbaum, A. 1992. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng. 18,* 3 (Mar.).

Baskett, F., Jermoluk, T., and Solomon, D. 1988. The 4D-MP graphics superworkstation: Computing + graphics = 40 mips + 40 mflops + 100,000 lighted polygons per second. In *Proceedings of COMPCON Spring 88*. 468–471.

Bennett, J., Carter, J., and Zwaenepoel, W. 1990. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York.

Berrendorf, R. and Helin, J. 1992. Evaluating the basic performance of the Intel iPSC/860 parallel computer. *Concur. Pract. Exper. 4,* 3 (May), 223–240.

Bershad, B., Zekauskas, M., and Sawdon, W. 1993. The Midway distributed shared memory system. In *Proceedings of COMPCON'93*. 528–537.

Blelloch, G., Chatterjee, S., Hardwick, J., Sipelstein, J., and Zagha, M. 1993. Implementation of a portable nested data-parallel language. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, New York.

Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., and Zhou, Y. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN*

Symposium on Principles and Practice of Parallel Programming. ACM, New York.

BROWNING, R., LI, T., CHUI, B., YE, J., PEASE, R., CZYZEWSKI, Z., AND JOY, D. 1994. Empirical forms for the electron/atom elastic scattering cross sections from 0.1-30keV. *Appl. Phys. 76,* 4 (Aug.), 2016–2022.

BROWNING, R., LI, T., CHUI, B., YE, J., PEASE, R., CZYZEWSKI, Z., AND JOY, D. 1995. Low-energy electron/atom elastic scattering cross sections for 0.1-30keV. *Scanning 17,* 4 (July/August), 250–253.

BURNS, A. 1988. *Programming in Occam 2.* Addison-Wesley, Reading, Mass.

CARRIERO, N. AND GELERNTER, D. 1989. Linda in context. *Commun. ACM 32,* 4 (Apr.), 444–458.

CHANDRA, R., GUPTA, A., AND HENNESSY, J. 1993. Data locality and load balancing in COOL. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, New York.

DAYDE, M. AND DUFF, I. 1990. Use of parallel Level 3 BLAS in LU factorization on three vector multiprocessors; the Alliant FX/80, the Cray-2, and the IBM 3090 VF. In *Proceedings of the 1990 ACM International Conference on Supercomputing.* ACM, New York.

DIETZ, H. AND KLAPPHOLZ, D. 1986. Refined Fortran: Another sequential language for parallel programming. In *Proceedings of the 1986 International Conference on Parallel Processing*, K. Hwang, S. M. Jacobs, and E. E. Swartzlander, Eds. 184–189.

DONGARRA, J. AND SORENSEN, D. 1987. SCHEDULE: Tools for developing and analyzing parallel Fortran programs. In *The Characteristics of Parallel Algorithms*, D. Gannon, L. Jamieson, and R. Douglass, Eds. The MIT Press, Cambridge, Mass.

FEO, J., CANN, D., AND OLDEHOEFT, R. 1990. A report on the Sisal language project. *J. Parallel Distrib. Comput. 10,* 4 (Dec.), 349–366.

FOSTER, I. AND TAYLOR, S. 1990. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, N.J.

FU, C. AND YANG, T. 1997. Space and time efficient execution of parallel irregular computations. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, New York.

GELERNTER, D. 1985. Generative communication in Linda. *ACM Trans. Program. Lang. Syst. 7,* 1 (Jan.), 80–112.

GHARACHORLOO, K. 1996. Memory consistency models for shared memory multiprocessors. Ph.D. thesis, Dept. of Electrical Engineering, Stanford Univ., Stanford, Calif.

GIFFORD, D., JOUVELOT, P., LUCASSEN, J., AND SHELDON, M. 1987. FX-87 reference manual. Tech. Rep. MIT/LCS/TR-407, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass. Sept.

GOLUB, G. AND LOAN, C. V. 1989. *Matrix Computations*, 2nd ed. The Johns Hopkins Univ. Press, Baltimore, Md.

GREGORY, S. 1987. *Parallel Logic Programming in PARLOG: The Language and Its Implementation.* Addison-Wesley, Reading, Mass.

GROSS, T., O'HALLORAN, D., AND SUBHLOK, J. 1994. Task parallelism in a high performance Fortran framework. *IEEE Parallel Distrib. Tech. 2,* 3 (Fall), 16–26.

HAGERSTEN, E., LANDIN, A., AND HARIDI, S. 1992. DDM—A cache-only memory architecture. *Computer 25,* 9 (Sept.), 44–54.

HALSTEAD, JR., R. 1985. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7,* 4 (Oct.), 501–538.

HAMMEL, R. AND GIFFORD, D. 1988. FX-87 performance measurements: Dataflow implementation. Tech. Rep. MIT/LCS/TR-421, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass. Nov.

HARRIS, J., LAZARATOS, S., AND MICHELENA, R. 1990. Tomographic string inversion. In *Proceedings of the 60th Annual International Meeting, Society of Exploration and Geophysics, Extended Abstracts.* 82–85.

HENDREN, L., HUMMEL, J., AND NICOLAU, A. 1992. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceed-*

ings of the SIGPLAN '92 Conference on Program Language Design and Implementation. ACM, New York.

Hill, M., Larus, J., Reinhardt, K., and Wood, D. 1992. Cooperative shared memory: Software and hardware for scalable multiprocessors. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, New York, 262–273.

Hoare, C. A. R. 1985. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, N.J.

INMOS Limited. 1984. *Occam Programming Manual.* Prentice-Hall, Englewood Cliffs, N.J.

Intel Supercomputer Systems Division. 1991. *Paragon XP/S Product Overview.* Intel Supercomputer Systems Division.

Karamcheti, V. and Chien, A. 1995. A comparison of architectural support for messaging in the TMC CM-5 and the Cray T3D. In *Proceedings of the 22nd International Symposium on Computer Architecture.* ACM, New York.

Kendall Square Research Corporation. 1992. *KSR-1 Technical Summary.* Kendall Square Research Corp., Cambridge, Mass.

Klappholz, D. 1989. Refined Fortran: An update. In *Proceedings of Supercomputing '89.* IEEE Computer Society Press, Los Alamitos, Calif.

Klappholz, D., Kallis, A., and Kong, X. 1990. Refined C—An Update. In *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau, and D. Padua, Eds. The MIT Press, Cambridge, Mass., 331–357.

Krishnamurthy, A., Culler, D., Dusseau, A., Goldstein, S., Lumetta, S., von Eicken, T., and Yelick, K. 1992. Parallel programming in Split-C. In *Proceedings of Supercomputing '92.* IEEE Computer Society Press, Los Alamitos, Calif., 262–273.

Lam, M. and Rinard, M. 1991. Coarse-grain parallel programming in Jade. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, New York, 94–105.

Lenoski, D. 1992. The design and analysis of DASH: A scalable directory-based multiprocessor. Ph.D. thesis, Dept. of Electrical Engineering, Stanford Univ., Stanford, Calif.

Lenoski, D., Laudon, J., Joe, T., Nakahira, D., Stevens, L., Gupta, A., and Hennessy, J. 1992. The DASH prototype: Implementation and performance. In *Proceedings of the 19th International Symposium on Computer Architecture.* ACM, New York.

Li, K. 1986. Shared virtual memory on loosely coupled multiprocessors. Ph.D. thesis, Dept. of Computer Science, Yale Univ., New Haven, Conn.

Lucassen, J. 1987. Types and effects: Towards the integration of functional and imperative programming. Tech. Rep. MIT/LCS/TR-408, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass. Aug.

Lusk, E., Overbeek, R., Boyle, J., Butler, R., Disz, T., Glickfield, B., Patterson, J., and Stevens, R. 1987. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc.

Martonosi, M. and Gupta, A. 1989. Tradeoffs in message passing and shared memory implementations of a standard cell router. In *Proceedings of the 1989 International Conference on Parallel Processing.* 88–96.

Metcalf, M. and Reid, J. 1990. *Fortran 90 Explained.* Oxford Science Publications.

Mohr, E., Kranz, D., and Halstead, R. 1990. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming.* ACM, New York, 185–197.

Nieh, J. and Levoy, M. 1992. Volume rendering on scalable shared-memory MIMD architectures. Tech. Rep. CSL-TR-92-537, Computer Systems Laboratory, Stanford Univ., Stanford, Calif. Aug.

Reppy, J. 1992. Higher–order concurrency. Ph.D. thesis, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y.

Rinard, M. 1994a. The design, implementation and evaluation of Jade, a portable, implicitly parallel programming language. Ph.D. thesis, Dept. of Computer Science, Stanford Univ., Stanford, Calif.

Rinard, M. 1994b. Implicitly synchronized abstract data types: Data structures for modular parallel programming. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software and Applications*, M. Furnari, Ed. World Scientific Publishing, 259–274.

Rinard, M. and Lam, M. 1992. Semantic foundations of Jade. In *Proceedings of the 19th Annual ACM Symposium on the Principles of Programming Languages.* ACM, New York, 105–118.

Rinard, M., Scales, D., and Lam, M. 1992. Heterogeneous Parallel Programming in Jade. In *Proceedings of Supercomputing '92.* IEEE Computer Society Press, Los Alamitos, Calif., 245–256.

Rinard, M., Scales, D., and Lam, M. 1993. Jade: A high-level, machine-independent language for parallel programming. *IEEE Comput. 26,* 6 (June), 28–38.

Rose, J. and Steele, G. 1987. C*: An extended C language for data parallel programming. Tech. Rep. PL 87-5, Thinking Machines Corp., Cambridge, Mass. Apr.

Rothberg, E. 1993. Exploiting the memory hierarchy in sequential and parallel sparse cholesky factorization. Ph.D. thesis, Dept. of Computer Science, Stanford Univ., Stanford, Calif.

Salmon, J. K. 1990. Parallel hierarchical N-body methods. Ph.D. thesis, California Institute of Technology.

Sandu, H., Gamsa, B., and Zhou, S. 1993. The shared regions approach to software cache coherence on multiprocessors. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, New York, 229–238.

Scales, D., Gharachorloo, K., and Thekkath, C. 1994. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, New York.

Scales, D. and Lam, M. S. 1994. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation.* ACM, New York.

Shoinas, I., Falsafi, B., Lebeck, A., Reinhardt, S., Larus, J., and Wood, D. 1994. Fine-grain access control for distributed shared memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, New York.

Singh, J. 1993. Parallel hierarchical N-body methods and their implications for multiprocessors. Ph.D. thesis, Dept. of Electrical Engineering, Stanford Univ., Stanford, Calif.

Singh, J. and Hennessy, J. 1992. Finding and exploiting parallelism in an ocean simulation program: Experiences, results, and implications. *J. Parallel Distrib. Comput. 15,* 1 (May), 27–48.

Singh, J., Weber, W., and Gupta, A. 1992. SPLASH: Stanford parallel applications for shared memory. *Comput. Arch. News 20,* 1 (Mar.), 5–44.

Sunderam, V. 1990. PVM: A framework for parallel distributed computing. *Concur. Pract. Exper. 2,* 4 (Dec.), 315–339.

Thinking Machines Corporation. 1991. *The Connection Machine CM-5 Technical Summary.* Thinking Machines Corp., Cambridge, Mass.

Traub, K. 1991. *Implementation of Non-strict Functional Programming Languages.* The MIT Press, Cambridge, Mass.

Woo, S., Ohara, M., Torrie, E., Singh, J., and Gupta, A. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture.* ACM, New York.

Yonezawa, A., Briot, J.-P., and Shibayama, E. 1986. Object oriented concurrent programming in ABCL/1. In *Proceedings of the 1st Annual Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM, New York, 258–268.

Received July 1996; revised September 1997; accepted November 1997