Coarse-Grain Parallel Programming in Jade

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

Abstract

This paper presents Jade, a language which allows a programmer to easily express dynamic coarse-grain parallelism. Starting with a sequential program, a programmer augments those sections of code to be parallelized with abstract data usage information. The compiler and run-time system use this information to concurrently execute the program while respecting the program's data dependence constraints. Using Jade can significantly reduce the time and effort required to develop and maintain a parallel version of an imperative application with serial semantics. The paper introduces the basic principles of the language, compares Jade with other existing languages, and presents the performance of a sparse matrix Cholesky factorization algorithm implemented in Jade.

1 Introduction

The goal of our research is to provide programming language support for exploiting coarse-grain concurrency, or concurrency in which each unit of serial computation executes at least several thousand instructions. There are two major reasons why automatic techniques that extract static parallelism from sequential programs cannot fully exploit available coarse-grain concurrency. First, only the programmer has the high-level knowledge necessary to decompose his program into coarse-grain tasks. This information is lost once the program is encoded in a conventional programming language. Second, it is sometimes important to exploit irregular, data-dependent concurrency available only as the program runs. The large grain size often makes it possible to profitably amortize the dynamic overhead required to exploit this last source of concurrency.

This research was supported in part by DARPA contract N00014-87-K-0828.

There have been several attempts to provide programming language support for coarse-grain concurrency. Most of the proposed languages support an explicitly parallel programming paradigm. The programmer must therefore directly manage the concurrency using constructs that create and synchronize parallel tasks. This management burden complicates the programming process, making parallel programming a more time-consuming, error-prone activity than programming in a conventional sequential language.

This paper introduces a new programming language called Jade, which supports coarse-grain concurrency within the sequential imperative programming paradigm. Jade programmers augment a sequential program with high-level dynamic data usage information. The Jade implementation uses this information to determine which operations can be executed concurrently without violating the program's sequential semantics. While the compiler can sometimes use this information to extract statically available concurrency, the Jade run time system is capable of analyzing the data usage information and extracting dynamically available concurrency. Because the Jade implementation is responsible for managing the parallel activity over the physical hardware, machine dependent optimizations can be provided by tailoring the implementation to different architectures. Thus Jade not only simplifies programming by preserving the familiar sequential imperative model of programming, but also enhances portability by providing machine-specific optimizations within the Jade implementation.

Because the power to abstract away from low-level details is critical in a language designed to support coarse-grain concurrency, Jade allows the programmer to express the data usage information at a level of abstraction appropriate to the granularity of parallelism. The programmer groups the units of serial execution as *tasks* and structures the data shared by multiple tasks as *shared data objects*. The programmer can express each task's side effects in terms of high-level side effect specification operations on shared data objects. The design of these objects therefore determines the parallelization and synchronization granularity.

The organization of the paper is as follows. We first introduce the basic programming paradigm, illustrating both how a programmer attaches simple data usage infor-

mation to a program's tasks and how the Jade implementation uses that information to run the program in parallel. In Section 3, we show that by providing more detailed information about a task's data usage, a Jade programmer can achieve more sophisticated concurrency patterns. Section 4 describes how programmers build shared data objects with high-level side effect specification operations. We then present a complete Jade programming example, and close with a discussion and comparison with other languages designed to express coarse-grain concurrency.

2 Basic Programming Paradigm

In Jade, the programmer provides the *program* knowledge required for efficient parallelization; the implementation combines its *machine* knowledge with this information to map the computation efficiently onto the underlying hardware. Here are the Jade programmer's responsibilities:

- *Task Decomposition:* The programmer starts with a serial program and uses Jade constructs to identify the program's task decomposition.
- *Side Effect Specification:* The programmer provides a dynamically determined specification of the side effects each task performs on the shared data objects it accesses.

The Jade implementation performs the following activities:

- *Constraint Extraction:* The implementation uses the program's serial execution order and the tasks' side effect specifications to extract the dynamic inter-task dependence constraints that the parallel execution must obey.
- *Synchronized Parallel Execution:* The implementation maps the tasks efficiently onto the hardware while enforcing the extracted dependence constraints.

The programmer expresses his program's task decomposition and side effect specifications using extensions to existing sequential languages. The Jade extensions include a data type used to define shared data objects, and several additional language constructs. These extensions have been implemented for C++, C and FORTRAN.

2.1 Shared Data Objects

All data accessed by multiple tasks must be identified as shared data objects. Programmers declare tasks' side effects by applying side effect specification operations to these shared data objects. For example, the *rd* (read) operation specifies that the task will read the given object, the *wr* (write) operation specifies that the task will write the given object, and the *rw* (read-write) operation specifies that the task will first read then write the given object. It is the programmer's responsibility to ensure that the declared side effect specification operations correspond to the way the task accesses the data. The Jade implementation provides several common shared data types used to create shared data objects; Section 4 describes how Jade programmers can define their own shared data types.

When executing a program in parallel, the Jade implementation preserves the program's semantics by maintaining the serial execution order for tasks with conflicting side effect specifications. For example, two tasks that write the same shared data object have conflicting side effects, and must execute sequentially in the program's serial execution order. The Jade implementation must also preserve the serial execution order between a task that writes a shared data object and another task that reads the same object. Of course, tasks accessing disjoint sets of objects or reading the same object can execute concurrently.

2.2 With and Only With

We now illustrate the basic Jade programming paradigm by presenting the *withth* (pronounced "with and only with") construct. Jade programmers use this construct to declare that a piece of code will execute *with* and *only with* a specified set of side effects to shared data objects.

```
withth   { side effect specification }
         ( parameters for task body ) {
     task body
}
```

Operationally, the Jade implementation creates a task when it executes a *withth* construct; the task body section contains the serial code executed when the task runs. When such a task executes it may reference certain variables from the enclosing environment which parameterize its behavior. The programmer gives a list of these variables in the parameters section. At task creation time the Jade implementation preserves the values of these variables by copying them into the task's context.

The programmer uses the side effect specification section to declare the side effects that the task will perform on shared data objects. The specification itself is an arbitrary piece of code containing side effect specification operations on these shared objects. Conceptually, the Jade implementation determines the task's side effects at task creation time by executing this specification. Because the specification can contain control flow constructs such as conditionals, loops and function calls, the programmer may use information available only at run time when declaring a task's side effects.

A Jade program's concurrency pattern is completely orthogonal to its procedural decomposition. As the following simple example illustrates, there is no requirement that concurrently executable tasks come from the same procedure invocation.

```
Double v[n];
int index[m];
p()
{
   for (int i = 0; i < m; i++) {
      q(v, index[i]);
   }
}

q(v, j)
Double v[];
int j;
{
      /* modify v[j] */
      ...
}
```

This example repeatedly applies *q* to elements of *v* accessed indirectly through the index array *index*. We assume that *q* modifies *v[j]* and has no other side effect. Invocations of *q* modifying different elements of the array can execute concurrently; conversely, invocations modifying the same element must obey the code's original sequential order.

A programmer can code the parallel version of our example in Jade as follows:

```
SharedDouble v[n];
int index[m];
p()
{
   for (int i = 0; i < m; i++) {
      q(v, index[i]);
   }
}

q(v, j)
SharedDouble v[];
int j;
{
   withth   { v[j].rw(); }
            ( SharedDouble v[]; int j; ) {
      /* modify v[j]; */
      ...
   }
}
```

Here the programmer identifies each invocation of *q* as a separate task. He therefore converts the objects that *q* modifies into shared data objects, in this case *SharedDouble*s, and uses the *rw* operation to declare that the task will read then write its parameter.

We now describe this program's operational interpretation. As the program executes the loop sequentially, the Jade implementation creates a new task for every invocation of *q*, copying *j* and the address of the array *v* into the new task's context. The implementation then analyzes the task's side effect specification and infers that the new task must not execute until all tasks from preceding iterations that include *v[j]* in their side effect specification have finished.

This example illustrates how simple Jade programs execute: one processor runs the program serially, periodically creating tasks at *withth* statements that the other processors pick up and execute. The Jade implementation uses the serial task creation order to determine the relative execution order of tasks with conflicting side effects.

In this simple model of parallel computation, synchronization takes place only at task boundaries. A task can run only when it acquires all of the shared data objects it will access; it releases the acquired objects only upon termination. Although it is possible to express the concurrency patterns of many parallel applications using just *withth*, some parallel applications have more complex concurrency patterns requiring periodic inter-task synchronization. In the next section we present the Jade constructs that allow programmers to express these more complex synchronization patterns.

3 Decoupling Parallelism and Synchronization

The restricted form of synchronization that *withth* supports can unnecessarily serialize computation in two cases: when a task's first access to a shared data object occurs long after the task starts running, and when a task's last access to a shared data object occurs long before the task terminates. The following procedure provides a concrete example of both forms of unnecessary serialization.

```
SharedDouble x, y;
p()
{
   withth { x.wr(); } () {
      x = f(1);
   }
   withth { y.rd(); x.rw(); } () {
      Double s;
      s = g(y);
      x = h(x, s);
   }
   withth { y.wr(); } () {
      y = f(2);
   }
}
```

This procedure generates three tasks. The tasks must execute sequentially to preserve the serial semantics. The

first unnecessary serialization comes from the fact that the second task does not access *x* until it finishes the statement *s = g(y).* Therefore, the statement *x = f(1)* from the first task should be able to execute concurrently with the statement *s = g(y)* from the second task.

The second unnecessary serialization comes from the fact that the second task never accesses *y* after the statement *s = g(y)* finishes. Therefore, the statement *x = h(x, s)* from the second task should be able to execute concurrently with the statement *y = f(2)* from the third task. In the next two sections we show how to eliminate both sources of unnecessary serialization.

One way to achieve full concurrency is to break the second task up into two tasks. This solution is inferior because the modification is not motivated by examining the code of the second task itself. Moreover, this solution requires that *s* be made into a shared object. The need to manage the two new serial tasks may also cause extra overhead. The solution presented below bypasses these problems by allowing tasks to synchronize as they execute.

3.1 With and With Only

Analyzing the *withth* construct, we observe that it simultaneously specifies two kinds of side effect information: *positive* side effect information and *negative* side effect information. The *withth* construct specifies positive side effect information by stating that the task body will carry out the declared side effects. Therefore, the task must not execute until it can perform these side effects without violating the program's serial semantics. *Withth* specifies negative side effect information by stating that the task body has no side effects except the declared side effects. The task can therefore run concurrently with any other piece of code as long as their side effects do not conflict. Operationally, positive side effect information causes *synchronization*, while negative side effect information creates opportunities for *concurrency*.

By providing a construct (*with*) that specifies only positive side effect information and a construct (*withonly*) that specifies only negative side effect information, we allow the programmer to create tasks that incrementally acquire shared data objects as they are accessed. The general forms of the *with* and *withonly* constructs are:

```
with        { side effect specification } {
   code body
}
withonly    { side effect specification }
            ( parameters for task body ) {
   task body
}
```

The *with* construct specifies side effects to shared data objects that the code body will immediately perform. Because the code body must have access to the shared data objects in the side effect specification section before it can proceed, we say that the *with* demands these objects. Since *with* provides only positive side effect information, the code body may declare additional side effects using nested Jade constructs.

Operationally, the Jade implementation suspends execution at a *with* statement until all previously created tasks that have a dependence conflict with the *with* statement have completed. *With* constructs therefore create synchronization, not concurrency.

The *withonly* construct only specifies that the code has no side effects besides the declared side effects. Before performing any side effects to shared data, the task body must declare the side effects using nested *with* or *withth* constructs. Because the task body does not immediately access the shared data objects in the side effect specification section, we say that the *withonly* claims these objects.

Operationally, when the Jade implementation executes a *withonly* statement, it creates an immediately executable task containing the *withonly*'s code and continues to execute the code following the *withonly* construct. Any subsequently created task need not wait for the *withonly* to finish unless its side effect specification conflicts with that of the *withonly*.

Returning to our example, the programmer can use *with* and *withonly* to make the second task demand *x* and *y* as they are accessed, instead of at the beginning of the task.

```
SharedDouble x, y;
p()
{
   withth { x.wr(); } () {
      x = f(1);
   }
   withonly { y.rd(); x.rw(); } () {
      Double s;
      with { y.rd(); } {
         s = g(y);
      }
      with { x.rw(); } {
         x = h(x, s);
      }
   }
   withth { y.wr(); } () {
      y = f(2);
   }
}
```

The statements *x = f(1)* and *s = g(y)* can now execute concurrently. However, the second and third tasks will still execute sequentially, because the second task will hold the claim to read *y* until it terminates.

3.2 Without

To eliminate this last source of unnecessary serialization, the programmer must be able to specify that a task has completed a declared side effect, and therefore no longer needs to access the corresponding shared data object. Jade provides the negative side effect specification construct *without* for just this purpose. Here is the general form of the *without* construct.

```
without    { side effect specification }
```

A *without* construct dynamically enclosed in a task specifies that the task body's remaining computation will perform none of the side effects in the *without*'s side effect specification. Programmers use *without* to reduce the enclosing task's specified set of side effects. This reduction may eliminate conflicts between the enclosing task and tasks occurring later in the sequential execution order. The later tasks may therefore be able to execute as soon as the *without* executes. In the absence of the *without* these tasks would have had to wait until the enclosing task terminated.

In our example, the programmer can use a *without* to allow the statements *x = h(x, s)* from the second task and *y = f(2)* from the third task to execute concurrently.

```
SharedDouble x, y;
p()
{
   withth { x.wr(); } () {
      x = f(1);
   }
   withonly { y.rd(); x.rw(); } () {
      Double s;
      with { y.rd(); } {
         s = g(y);
      }
      without { y.rd(); }
      with { x.rw(); } {
         x = h(x, s);
      }
   }
   withth { y.wr(); } () {
      y = f(2);
   }
}
```

3.3 Hierarchical Concurrency

We have presented *withonly* as a way to delay a task's demands for shared data objects while maintaining the underlying serial execution order on side effects to those objects. *Withonly* can also be used to express hierarchically structured concurrency patterns. By enclosing subtasks in *withonly* constructs, a programmer can easily create, execute and synchronize multiple parallel computations simultaneously. Consider the following example:

```
SharedDouble *x;
p()
{
   withonly { x->wr(); } () {
      SharedDouble *y, *z;
      y = new SharedDouble;
      z = new SharedDouble;
      withth { y->wr(); } (SharedDouble *y;) {
         *y = f(1);
      }
      withth { z->wr(); } (SharedDouble *z;) {
         *z = f(2);
      }
      with { x->rw(); y->rd(); z->rd(); } {
         *x = h(x,y,z);
      }
   }
}
```

Here the *withonly* encloses a group of subtasks that produce *x*; the entire procedure itself can run concurrently with other parts of the program that do not need the value of *x*. This example illustrates that a task need not specify all of its side effects, but just the externally visible ones. Here the *withonly* need not claim *y* and *z* because they are not visible outside its task body. The following two general rules define the legal use of claims and demands:

- Every access to a shared data object must be (dynamically) enclosed in a *with* or *withth* construct that declares that access, and
- Every *withonly* or *withth* must declare all of its task body's accesses to externally visible shared data objects.

4 Applying Data Abstraction to Synchronization

Jade tasks synchronize on the pieces of data that they access. In the preceding examples, tasks accessed and synchronized on fine-grain objects (i.e. *Double*s). Coarse grain tasks, however, access coarse pieces of data. For example, many parallel matrix algorithms access a matrix by rows or by columns. Because the unit of synchronization should match the granularity of data access, the shared data objects that these tasks use should support synchronization on the coarse pieces of data that the tasks access.

Jade programmers build shared data objects using a synchronization type called tokens. Each token functions as a synchronization data abstraction by carrying the dependence for a conceptual unit of data. We use the C++ class notation to make this abstraction more apparent. FORTRAN and C do not have the syntactic sugar

to bundle up the tokens with the data but the basic programming ideas are the same.

Each token has three side effect specification operations: *rd*, *wr*, and *rw* specifying, respectively, the read side effect, write side effect and read then write side effect. In the simplest case, the programmer augments a data type with a token and side effect specification operations; the *SharedDouble* used above is such an example:

```
class SharedDouble : public Double {
private:
        token _token;
public:
        void rd() { _token.rd(); };
        void wr() { _token.wr(); };
        void rw() { _token.rw(); };
}
```

We now present an example which demonstrates how Jade programmers use tokens to build complex shared data objects. The following sparse matrix data structure stores the compressed columns in a linear array. An index array gives each column's starting location in the linear array; another array gives each element's row index. If the computation synchronizes on the matrix's columns, the programmer simply associates a token with each column.

```
class Sparse {
public:
    double Data[MAX_ELEMENTS];
    int RowIndex[MAX_ELEMENTS];
    int ColumnIndex[MAX_COLUMNS];
    int NumColumns;
}
class SharedSparse : public Sparse {
private:
    token _token[MAX_COLUMNS];
public:
    void col_rd(int col) { _token[col].rd(); }
    void col_wr(int col) { _token[col].wr(); }
    void col_rw(int col) { _token[col].rw(); }
    void col_cm(int col) { _token[col].cm(); }
}

SharedSparse *M;
ColMod() {
    for (int i = 0; i < M->NumColumns; i++) {
        withth { M->col_rw(i); } (int i;) {
            /* modify column i of M */
            ...
        }
    }
}
```

Here, each token carries the dependence for a column in the sparse matrix, which consists of an arbitrary number of contiguous data elements. In general, a shared data object's synchronization granularity need not correspond to any syntactic data declaration unit. Using the token and its side effect operations as primitives, a Jade programmer can define the object's own side effect specification operations to match the way the program uses the data. In this example the *ColMod* routine accesses the sparse matrix by columns, which matches the *SharedSparse* type's side effect specification interface. Expressing side effects in terms of tokens can clarify the dependence structure. In this example, the clarified structure makes it possible for a compiler to discover the static independence of *ColMod*'s loop iterations.

We have described how Jade enforces the serial semantics by maintaining the sequential order between writes and other accesses to the same data. It is possible to further relax the execution order by exploiting the higher level semantics of user-defined operations. Consider the histogram example. Because addition is commutative and associative, the histogram increments commute with each other. Therefore, the implementation need not enforce the individual read and write dependence constraints as long as the increments execute with mutual exclusion. For another example of commuting, mutually exclusive updates, see the sparse Cholesky factorization algorithm presented in Section 5. Because many programs contain commuting updates, Jade tokens support the commutative update (*cm*) side effect specification operation in addition to the basic *rd*, *wr* and *rw* operations.

This view of synchronization smoothly generalizes from individual memory locations with read and write operations to abstract data types with associated side effect specification operations, each with its own synchronization rules. The shared data object concept provides an effective synchronization framework for concurrent object-oriented programming.

5 A Programming Example

The current Jade implementation consists of a run-time system and a preprocessor that translates Jade code to C, C++ or FORTRAN code containing calls to this run-time system. This implementation runs on an Encore Multimax and a Silicon Graphics IRIS 4D/240S. Implemented applications include a sparse Cholesky factorization algorithm due to Rothberg and Gupta [19], the Perfect Club benchmark MDG [2], LocusRoute, a VLSI routing system due to Rose [18], a parallel Make program, and cyclic reduction, a column-oriented matrix algorithm.

To illustrate Jade with a more realistic example, we now show how Rothberg and Gupta's sparse Cholesky factorization algorithm is implemented in Jade. The factorization algorithm is based on supernodes[19], or groups of adjacent columns with identical nonzero structure. For example, the supernodes of the matrix in Figure 1 are (1,2), (3), (4,5,6), (7,8,9) and (10, 11).

The serial computation processes the supernodes from left to right. Each supernode generates one internal up-

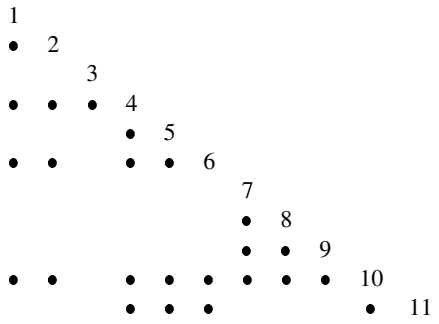

Figure 1: A Sparse Matrix

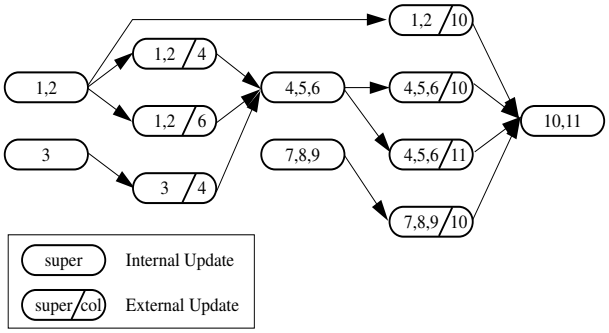

Figure 2: A Sparse Matrix Task Graph

date and a set of external updates. The internal update performs the internal supernode column cancellations and divides each column by its diagonal. Each internal update writes all of the columns in the supernode. There is one external update for each column to the right of the supernode whose column index is the same as the row index of one of the supernode's nonzeros. An external update computes the contributions of the supernode's columns to the external column. Each external update reads all of the supernode's columns and writes the external column. Two external updates to the same column commute.

In a coarse-grain parallel implementation of this algorithm, each update corresponds to a task. The dependence pattern of the computation is as follows: all of the external updates to a supernode's columns must complete before the supernode's internal update starts. As soon as the internal update finishes, the supernode's external updates to other columns can start. Finally, updates to the same columns can commute but must execute with mutual exclusion.

This computation's concurrency pattern depends on the nonzero structure of the matrix. The task graph for the matrix in Figure 1 is shown in Figure 2. To keep the diagram simple, mutual exclusion constraints between external updates to the same column are not shown.

The serial code for this algorithm contains a routine that takes a supernode as a parameter and performs the supernode's internal and external updates. The Jade version merely encloses each update in a *withth* statement. Therefore, the Jade programmer need only insert two *withth*'s to go from the serial version to the parallel version, as shown in Figure 3.

The algorithm accesses a supernode first by columns (when performing the external updates to the supernode's columns), then as a unit (when performing the supernode's external updates to other columns). To capture both granularities of synchronization, we add a token for each supernode to the column-oriented *SharedSparse*

```
Factor (M)
SharedSparse *M;
{
  for (all supernodes super in the matrix M
        from left to right) {
    CompleteSuperNode(M,super);
  }
}

CompleteSuperNode(M,super)
SharedSparse *M;
int super;
{
  withth {
    for (all columns col in super) {
      M->col_rw(col);
    }
    M->super_rw(super);
  } (SharedSparse *M; int super;) {
    InternalUpdate(M,super);
  }

  for (all columns col that super updates) {
    withth {
      M->super_rd(super);
      M->col_cm(col);
    } (SharedSparse *M; int super; int col;) {
      ExternalUpdate(M,super,col);
    }
  }
}
```

Figure 3: Jade Sparse Cholesky Factorization

class definition from Section 4. The side effects to a supernode are specified as column accesses before the internal update, and as supernode accesses after the update. To interface between the two, the internal update claims the data in both column and supernode granularities.

We compare the Jade version with Rothberg and Gupta's parallel version [20] implemented in the ANL Macro package[16]. Each internal or external update is also a task in Rothberg and Gupta's program. Their program explicitly spawns a thread for every processor. External updates are statically partitioned among the threads, and internal updates are managed using a task queue. Before the actual factorization begins, the program precomputes the number of external updates to each supernode. Every time an external update completes, the code decrements the external column's supernode count and checks if it is zero. If so, the code explicitly enqueues that supernode's internal update onto the task queue. All threads are notified when the internal update is completed. This code has been highly optimized, and has a minimal run-time overhead.

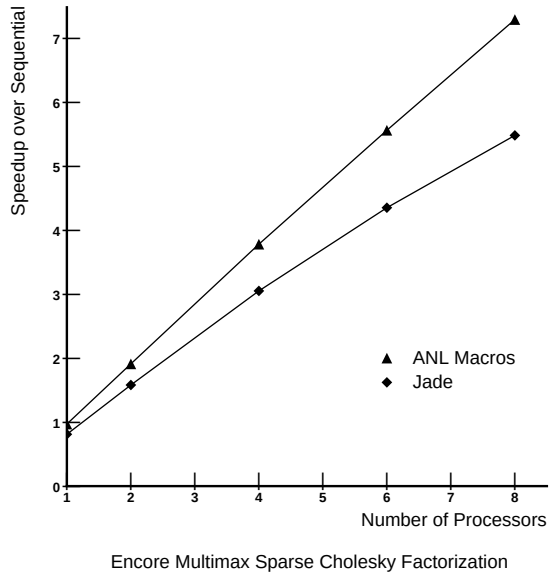

Figure 4: Jade and ANL Macro Package Speedups

We compare the performance of both versions of the algorithm in Figure 4. The performance is measured as the speedup factor relative to the extremely efficient serial sparse Cholesky factorization algorithm presented in [19]. These speedup figures are for the factorization phase of the computation only; the matrix is BCSSTK15 (the module of an offshore platform) from the Harwell-Boeing Sparse Matrix Collection [7]. The factored matrix has 3,948 columns, 647,274 nonzeros, 1,633 supernodes and generated 161,060 Jade tasks, with an average of 1,024 floating point operations per task. These performance numbers are collected from the Encore Multimax Jade and ANL Macro package implementations.

We first observe that the performance of the optimized ANL program running on a single processor is comparable to that of the sequential program, indicating that the ANL program has low overhead. On the other hand, Jade's general run-time system has a much higher overhead. Fortunately, the Jade program scales reasonably, with the 8-processor implementation running about 7 times faster than the uniprocessor Jade program. We are currently working on optimizations that will improve the performance of the Jade run-time system.

6 Discussion and Comparison with Other Work

Jade is designed to support the parallel execution of computations expressible as a sequential program. To successfully parallelize a program using Jade, the programmer must ensure that it has enough inherent concurrency to keep the target machine busy. In some cases, the programmer will need to privatize some of the global variables to eliminate unnecessary sequencing constraints caused by data reuse. In other cases the programmer may need to use different algorithms with more inherent concurrency.

Jade was designed for machines with a single address space, such as the large-scale DASH multiprocessor under development at Stanford [13]. In such machines, the long latency associated with remote data accesses makes it important to reuse cached data whenever possible. The current Jade implementation identifies tasks which access the same tokens, and schedules these tasks on the same processor. Tasks will therefore be able to reuse data brought into the cache by the previously executed task.

The current implementation requires an underlying shared address space. For Jade to run on a machine with separate address spaces, the implementation must generate the communication required to transfer shared data between processors. The current language, however, does not explicitly associate tokens with the data whose dependence they carry. Therefore, a Jade implementation cannot generate the communication because it cannot know which actual pieces of data a task will touch. We plan to extend the language so that tokens are explicitly associated with the data they represent. This association will make it possible to implement Jade on machines with separate address spaces.

Jade has two design principles which together set it apart from other programming languages. The first principle is that Jade provides implicit concurrency and synchronization by relaxing a sequential program's execution order. Since the Jade implementation enforces the data dependence constraints, the programmer can preserve both the structure and the semantics of the serial

program in the parallel version. The second principle is that Jade supports data abstraction in that Jade programmers specify side effect information using high-level operations on shared data objects.

In the following section we examine the ramifications of the first principle by comparing Jade to explicitly parallel programming languages. We then compare Jade with other languages designed to express the concurrency available in serial programs.

6.1 Explicit Concurrency and Synchronization

A major issue in parallel programming language design is the question of how to correctly synchronize coarse-grain tasks. In this section we compare Jade with approaches that provide constructs to create and explicitly synchronize parallel tasks.

6.1.1 Task Queue Model

One common way to synchronize coarse-grain parallel computation is to use a threads package with thread creation and low-level synchronization primitives to implement an explicit task queue. The programmer first breaks his program up into a set of tasks; a task is enabled (i.e., put onto the task queue) when all of its predecessors in the dependence graph have terminated. Free processors grab and run these enabled tasks.

The programmer must enforce the inter-task data dependence constraints by inserting synchronization primitives into tasks that touch the same data. This direct management code becomes distributed throughout the program text, encoding the global synchronization pattern in terms of the provided low-level synchronization primitives. This synchronization code creates new explicit connections between parts of the program that access the same data, making the program harder to create and modify. If the program's concurrency pattern changes, the programmer must go through the program modifying the distributed pieces of synchronization code.

Programmers can use Jade as a high-level interface to the task queue model of computation. Jade programmers provide local data usage information which the Jade implementation uses to extract and implement the global task dependence graph. Because Jade programmers do not manage the synchronization, they add no new explicit connections between pieces of code. Therefore, Jade programs are easier to modify and maintain than the corresponding task queue versions. The major advantage of a direct task queue implementation is efficiency: the programmer can control the machine at a fairly low level and use special-purpose synchronization strategies tailored to the application at hand. Jade's general purpose synchronization strategy may therefore be less efficient, although the difference will be negligible for computations with a large enough grain size.

6.1.2 Explicit Communication Operations

Many proposed parallel programming languages provide explicit communication operations to move data between parallel tasks. Programmers insert these operations into their tasks at data production and consumption points to synchronize the computation. For example, languages such as CSP [11], Ada [17] and Occam [14] provide synchronous message passing operations. One major problem with this approach is that producers and consumers must agree on the order and relative time of data transfer.

Linda supports a less tightly coupled programming style by providing a global tuple space with asynchronous operations to insert, read and remove data [4]. Tuple spaces support mutual exclusion and asynchronous producer/consumer synchronization based on the presence or absence of data. Tuple spaces also support some less frequently used synchronization mechanisms such as counting semaphores. Although these mechanisms easily synchronize some restricted dependence patterns, they do not support the synchronization patterns required to enforce general dependence constraints. As in the task queue model, programmers implementing applications with such general dependence constraints must directly encode the program's global synchronization pattern using the provided synchronization mechanisms as low-level primitives. For example, the Linda sparse Cholesky factorization application directly implements the task graph's synchronization pattern using counting semaphores [3].

6.1.3 Global Control Languages

Another approach is to use a control language to directly express an application's global concurrency pattern. For reasons of efficiency and programming convenience, the actual pieces of code that the control language invokes to carry out the computation are written in a serial, imperative language such as FORTRAN or C. Here we present a brief list of some of the approaches.

SCHEDULE allows programmers to give the system a set of tasks and an explicit specification of the task dependence graph [6]. SCHEDULE then executes the tasks while obeying the given dependence constraints. Programmers using coarse-grain dataflow languages such as LGDF [1] and TDFL [21] express concurrency and synchronization with dataflow graphs. Execution of the dataflow graph provides synchronized concurrency. A Strand programmer expresses his program's global concurrency structure in the committed-choice concurrent logic programming paradigm [8]. Suspension on unbound logic variables provides synchronization, while simultaneous goal satisfaction provides concurrency.

While these languages centralize the synchronization instead of distributing it throughout the program, the programmer must still directly implement his program's global synchronization structure. These approaches also burden the programmer with an additional programming paradigm, and force the programmer to use the alternative paradigm down to the lowest level of granularity.

6.1.4 Function and Method Modifiers

Some languages augment the semantics of function invocation to provide concurrency and synchronization. For example, Multilisp futures enforce the producer/consumer sequence constraint between a function creating data and its caller consuming the return value [9]. Because this mechanism works well for synchronizing returns from asynchronously invoked functions or methods, concurrent object-oriented languages such as COOL [5] and ConcurrentSmalltalk [22] also provide the future synchronization mechanism.

Futures, however, are not designed to synchronize the multiple updates to mutable shared data that are a central feature of object-oriented programming. Therefore, concurrent object-oriented languages also let a programmer specify that a method must have mutually exclusive access to the receiver before it can run.

It is sometimes possible to parallelize a COOL or ConcurrentSmalltalk application by adding a few future or mutual exclusion modifiers to a sequential program. But to implement applications with general dependence constraints, programmers resort to using futures and mutual exclusion as low-level concurrency and synchronization primitives. For example, the COOL sparse Cholesky factorization algorithm is synchronized by counting completed column updates.

6.1.5 Jade

In all of the languages presented above, programmers must directly manage the program's global concurrency structure to implement applications with general dependence constraints. A Jade programmer, however, can simply express these parallel applications within an implicitly parallel paradigm. Both the structure and the semantics of the original sequential program are preserved in the parallel version. In Jade, a programmer need only provide local data usage information; the Jade implementation is responsible for directly managing the program's global concurrency structure.

In Jade programs synchronization and data flow unidirectionally from tasks occurring earlier in the sequential execution order to tasks occurring later. This unidirectional flow allows the Jade implementation to suppress spawning in the face of excess concurrency without risking deadlock. But, this also means that Jade cannot express parallel algorithms requiring bidirectional task communication. Our Jade design therefore sacrifices generality in order to fully support the sequential imperative programming paradigm.

In the absence of static optimizations or hierarchically structured concurrency, the Jade implementation creates tasks sequentially. This serial task creation may cause a significant performance loss if the grain size is small. To drive down the minimum grain size for which Jade is applicable, we are currently investigating the use of static analysis to detect simple, common parallel structures and substitute the general parallelization and synchronization approaches with specialized solutions.

6.2 Parallelizing Serial Programs

We now compare Jade with other approaches designed to parallelize programs written in sequential programming languages.

6.2.1 Control Concurrency

One direct way to endow a sequential language with synchronized concurrency is to augment the language with explicit control constructs such as fork/join, par_begin/par_end, or doall statements. These constructs allow the programmer to spawn several independent processes; the program then blocks until all the spawned processes terminate. It is the programmer's responsibility to ensure that there are no race conditions, so that the serial and parallel semantics are identical.

There are two major drawbacks to this control-oriented approach. First, these parallel constructs can express neither irregular task dependence patterns nor common parallel idioms requiring periodic intertask synchronization. Second, they force the programmer to destroy the program structure by moving concurrently executable pieces of code to the same artificial spawn point. Consequently, the program may be harder to understand, and the need for code transformations may discourage the programmer from exploiting all possible sources of parallelism within the program.

Jade, on the other hand, allows programmers to preserve the structure of the original program in the parallel version. Jade programmers need not move concurrently executable pieces of code to a common invocation point; the Jade constructs make it easy and even natural to exploit synchronized concurrency across module and procedure boundaries. Jade's dynamic side effect specification capabilities support the creation of irregular data-dependent concurrency.

6.2.2 Data Usage Concurrency

There are several languages which, like Jade, allow the programmer to express concurrency with side effect specification constructs. In FX-87 [15], memory locations are

partitioned into a finite, statically determined set of regions. The programmer declares the regions of memory that a function touches as part of the function's type. The FX-87 implementation then uses a static type checking algorithm to verify the correspondence between a procedure's declared and actual side effects. The implementation can use this information to execute parts of the program with no conflicting side effects concurrently.

While the finite set of regions determined at compile-time enables the FX-87 type checker to verify the correctness of the specification, it also severely limits the scope of the supported concurrent behavior. First, it means that some of the dynamically created variables must be mapped to the same static region; this reduces opportunities for concurrency. More importantly, each aggregate, such as an array, must be in a single region. This side effect specification imprecision dramatically reduces the amount of expressible concurrency, especially for programs whose main source of concurrency is tasks that access disjoint regions of an array [10].

Refined C's *disjoint* statement allows the programmer to create a set of access-restricted aliases that break an array up into disjoint pieces [12]. A programmer can then refer to the data via these aliases to indicate the lack of dependence between accesses using different aliases. Compiler analysis may be used to disambiguate between references of the same alias names. The disjoint statements are dynamic; that is, different views can be adopted at different times of the computation.

6.2.3 Jade

Jade differs from FX-87 and Refined C in its unique support for abstraction. First, the programmer need only describe the side effects of entire tasks, not individual functions. This not only simplifies programming but also gives the system valuable information on a suitable decomposition of the computation into parallel tasks. Second, the side effect specification is abstract, in terms of user-defined functions on user-defined objects. The Jade implementation therefore sees the computation performed at the same conceptual level of abstraction as the programmer. Furthermore, instead of enforcing the individual read and write ordering of individual memory locations, higher semantic knowledge can be used to relax the sequential execution order; commutative updates are a common example. The usage information is in a form such that both static and dynamic forms of parallelism can be detected and exploited.

Unlike FX-87 and Refined C, the programmer, not the implementation, is responsible for ensuring that a task's side effect specification correctly summarizes its actual side effects. In the presence of an explicit association between tokens and the data they represent, Jade can statically check the correctness whenever possible, and, if necessary, insert dynamic checks to ensure that the specification is correct. If the overhead of such safety checks is intolerable in production mode, the programmer can use them only during the program debugging stage. Since Jade only needs to check that every reference is included in the side effect specification of the task, each debugging run checks that the program is correct with respect to a set of input data. This correctness is independent of the timing of the parallel execution.

7 Conclusion

Jade supports the exploitation of coarse-grain concurrency within the sequential imperative programming paradigm. Jade programmers augment a sequential program with high-level dynamic data usage information; the Jade implementation then uses this information to concurrently execute the program while respecting the data dependence constraints. Jade therefore provides implicit concurrency, freeing the programmer from the burden of explicit concurrency management.

Jade programmers use the *withth* construct to express simple concurrency patterns in which synchronization takes place only at task boundaries. *Withth* can be used as a high-level interface to the task queue model of computation. The *with*, *withonly* and *without* constructs support more complicated concurrency structures requiring periodic inter-task synchronization.

Jade supports the expression of the full range of coarse-grain concurrency, including irregular data-dependent concurrency available only as the program runs. Jade's support for the expression of concurrency available across procedure boundaries allows Jade programmers to retain the structure of the original program in the parallel version. Therefore, Jade programmers can preserve program structure decisions made for reasons of good design.

Jade's token data type supports the creation of shared data objects with high-level side effect specification operations. Jade programmers can therefore express their tasks' side effect information at the same level of abstraction as the tasks access the shared data objects.

In the future we plan to extend Jade so that the tokens are explicitly associated with the data they represent. This association will allow us to implement Jade on machines with separate address spaces. We will also investigate how to improve the performance of Jade using compiler technology.

Acknowledgments

We would like to thank Jennifer Anderson and Dan Scales for participating in many fruitful discussions on Jade. Jennifer also wrote the Jade preprocessors; Dan implemented the Jade versions of LocusRoute and Make.

We would also like to thank Edward Rothberg for his help with the sparse Cholesky factorization code.

References

[1] R. G. Babb II, L. Storc, and W. C. Ragsdale. A large-grain data flow scheduler for parallel processing on cyberplus. In *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986.

[2] M. Berry and et al. The perfect club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[3] N. Carriero and D. Gelernter. Applications Experience with Linda. In *Proceedings of the ACM Symposium on Parallel Programming*, pages 173–187, New Haven, Conn., July 1988.

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

[5] R. Chandra, A. Gupta, and J. L. Hennessy. COOL: A Language for Parallel Programming. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 126–148. MIT Press, Cambridge, MA, 1990.

[6] J. J. Dongarra and D. C. Sorenson. A portable environment for developing parallel FORTRAN programs. *Parallel Computing*, (5):175–186, 1987.

[7] I. Duff, R. Grimes, and J. Lewis. Sparse Matrix Test Problems. *ACM Transactions on Mathematical Software*, 15(1):1–14, 1989.

[8] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1990.

[9] R. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[10] R. T. Hammel and D. K. Gifford. FX-87 Performance Measurements: Dataflow Implementation. Technical Report MIT/LCS/TR-421, MIT, November 1988.

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

[12] D. Klappholz, A. Kallis, and X. Kong. Refined C - An Update. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 331–357. MIT Press, Cambridge, MA, 1990.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[14] Inmos Ltd. *Occam Programming Manual*. Prentice-Hall, Englewood Cliffs, N.J., 1984.

[15] J. M. Lucassen. Types and Effects: Towards the Integration of Functional and Imperative Programming. Technical Report MIT/LCS/TR-408, MIT, August 1987.

[16] E. Lusk, R. Overbeek, et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[17] United States Department of Defense. *Reference Manual for the Ada programming language*. DoD, Washington, D.C., January 1983. ANSI/MIL-STD-1815A.

[18] J. S. Rose. LocusRoute: A Parallel Global Router for Standard Cells. In *Proceedings of the 25th Design Automation Conference*, pages 189–195, June 1988.

[19] E. Rothberg and A. Gupta. Efficient sparse matrix factorization on high-performance workstations - exploiting the memory hierarchy. To appear in ACM Transactions on Mathematical Software.

[20] E. Rothberg and A. Gupta. Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations. In *Proceedings of Supercomputing '90*, pages 232–241, November 1990.

[21] P. A. Suhler, J. Biswas, K. M. Korner, and J. C. Browne. TDFL: A task-level dataflow language. *Journal of Parallel and Distributed Computing*, 9:103–115, 1990.

[22] Y. Yokote and M. Tokoro. Concurrent Programming in ConcurrentSmalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. MIT Press, Cambridge, MA, 1987.