Hierarchical Concurrency in Jade

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

Abstract
Jade is a data-oriented language for parallelizing programs written in a serial, imperative programming language such as C, FORTRAN, or C++. A Jade programmer simply augments the program with assertions specifying how the program accesses data. The Jade implementation dynamically interprets these assertions to execute the program concurrently while enforcing the program's data dependence constraints. In a previous paper [4], we show how programmers use Jade to express concurrency patterns such as dynamic task graphs and pipelining. In this paper, we show how Jade programmers can naturally express hierarchical concurrency patterns by specifying how a program uses hierarchically structured data.

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

1 Introduction

Jade [4] is a language for parallelizing programs written in a serial, imperative programming language such as C, FORTRAN, or C++. Jade is an implicitly parallel language designed to express deterministic concurrency patterns that preserve the serial semantics of these programs. Because we believe that the compiler should exploit both instruction-level and loop-level concurrency, programmers use Jade to express coarse-grain, task-level concurrency.

A Jade programmer first divides a sequential program up into tasks. Tasks interact through accesses to *shared objects*. The programmer summarizes each task's *effects* by specifying which externally visible shared objects the task will read or write. The Jade implementation uses this local data usage information to relax the program's serial execution order, permitting tasks without conflicting accesses to shared objects to execute concurrently. The programmer dynamically specifies each task's effects when it is created, thus allowing Jade to exploit data-dependent concurrency which can only be determined at run-time.

Jade's data-oriented approach to concurrency contrasts with the control-oriented approach provided by many parallel programming languages [1, 2, 3, 5]. A programmer using one of these languages directly implements the program's global concurrency structure using low-level constructs to create and coordinate parallel tasks. To synchronize these tasks correctly, the programmer must establish explicit connections between otherwise unrelated pieces of the program that access the same data. These connections make the parallel program difficult to create, modify and maintain. Jade programs, on the other hand, only contain local information about the pieces of data that tasks read and write. The Jade implementation, not the programmer, extracts and enforces the global concurrency pattern implicit in the program's data usage. By requiring only local data usage information, Jade promotes modularity and simplifies parallel programming.

In a previous paper [4], we show how a Jade programmer can easily produce concurrency patterns such as dynamic task graphs and pipelining by specifying how a program uses data. In this paper, we show how Jade programmers can naturally express hierarchical concurrency patterns by specifying how a program uses hierarchically structured data. For example, the Jade implementation can generate a parallel tree traversal from a specification of how a program accesses nodes and subtrees. Similarly, a programmer who views a matrix hierarchically as a collection of columns may create a task to perform a matrix operation, which in turn creates tasks to perform the operation on each of the matrix's columns.

The organization of this paper is as follows. We first briefly introduce the basic concepts of Jade: the shared objects and the language constructs. We then demonstrate how the use of hierarchical data structures in Jade programs leads to hierarchical concurrency patterns. We also illustrate how the structure of the data hierarchy can be used to refine gradually the specified effects of a task. Finally, we show that hierarchical concurrency makes the generation of concurrency in a Jade program more efficient.

2 Shared Objects

Each Jade task contains a section of code declaring which shared objects the task will read or write. The programmer expresses the effects in this section using a synchronization abstraction called *tokens*. The programmer first decides on the granularity at which tasks will access each piece of data, then associates a token with each piece of data at that level of granularity. To declare that a task will access a given piece of data, the programmer uses an *effect specification operation* applied to the corresponding token.

Each token has two effect specification operations: *rd* and *wr*, specifying, respectively, the read effect and write effect. By definition, two reads to the same shared object do not conflict; all other binary combinations of reads and writes to the same object conflict. Combinations of reads and writes to different shared objects do not conflict. To maintain the serial semantics of a program, tasks with conflicting effects execute in their original serial execution order.

Jade programmers usually encapsulate tokens and the data with which the tokens are associated as an object. (We use the C++ class notation below to make this encapsulation more apparent.) Each such object provides its own effect specification interface, implementing its effect specification operations internally using its encapsulated tokens. In the simplest such use of tokens, the programmer augments a class with a token and effect specification operations; the following SharedMatrix class illustrates such a use. In this case, the matrix's effect specification interface consists of the read_matrix and write_matrix operations.

```
class SharedMatrix {
        token _token;
        double elements[N][N];
    public:
        void read_matrix() { _token.rd(); };
        void write_matrix() { _token.wr(); };
        . . .
}
```

This practice of associating one token with each object works well for data that is created and accessed only as a single object. In section 4 we will show how a Jade programmer can create token hierarchies that correspond to the hierarchical structure of the data. These token hierarchies allow the programmer to specify naturally the effects of tasks that manipulate hierarchically structured shared objects.

3 Jade Constructs

Jade is a language for declaratively expressing data usage information; the names of the Jade constructs reflect this declarative perspective. The Jade implementation assigns an operational meaning to Jade programs by using this data usage information to create concurrency and synchronization. In this section we present the Jade constructs, giving both the declarative and operational meanings. For a more in-depth presentation, see [4].

3.1 With and Only With

Jade programmers use the **withth** (pronounced "with and only with") construct to declare that a piece of code will execute *with* and *only with* a specified set of effects to shared objects:

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

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.

The parameters section contains a list of variables from the enclosing environment. The values of these parameters are copied into the task's context when it is created. The task can reference these parameters when it runs.

The effect specification section declares how the task will access shared objects. The specification is an arbitrary piece of code containing access specification statements. The implementation executes this piece of code when the task is created to determine which shared objects the task will read and/or write.

The specification may contain dynamically resolved variable references and control flow constructs such as conditionals, loops and function calls. The programmer may therefore use information available only at run time when declaring a task's effects.

To preserve the serial semantics, the Jade implementation must carry out conflicting accesses to shared objects in the underlying sequential execution order. The implementation must therefore ensure that a task does not execute until all previous (in the underlying sequential execution order) tasks with conflicting effects have completed. For example, if a particular task will write a given shared object, the implementation must execute all preceding tasks that read or write the object before executing that task.

Withth supports a simple model of parallel computation in which synchronization takes place only at task boundaries. A task can run only when it acquires access all of the shared 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 two sections we present the Jade constructs that allow programmers to express these more complex synchronization patterns.

3.2 With and Withonly

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

To allow the programmer to express concurrency and synchronization with greater precision, Jade provides a construct (**with**) that specifies only positive effect information and a construct (**withonly**) that specifies only negative effect information. With these constructs, the programmer can create tasks that obtain access to shared objects incrementally as they are used in the task, rather than all at once when the task is created. The general forms of the **with** and **withonly** constructs are:

with { effect specification }

withonly { effect specification }
            (parameters for task body) {
    task body
}

The **with** construct specifies the shared objects that the code after the **with** will read or write. Because the code must have access to the shared objects in the effect specification section before it can proceed, we say that the **with** *demands* these objects.

When the task proceeds, we say that it holds a *demand* on each of the shared objects. The implementation suspends the task at a **with** statement until the completion of all preceding (in the underlying serial execution order) tasks with potentially conflicting accesses to the specified shared objects. A **with** construct declaring the access must precede each task's first use of each shared object. **With** constructs create synchronization, not concurrency.

The **withonly** construct declares that the task body has no effects besides the specified effects. It does not declare that the task will immediately carry out any of the specified effects; the task body must declare each effect as it occurs using nested **with** or withth constructs. Therefore, the **withonly** construct creates opportunities for concurrency, but does not imply synchronization.

To execute a withonly statement, the Jade implementation creates an immediately executable task containing the **withonly**'s code and continues to execute the code following the **withonly** construct. To ensure that the accesses to the shared objects proceed in the order determined by the original serial program, the implementation suspends any subsequent task with conflicting demands on the specified shared objects. We say that the task created by the **withonly** holds a *claim* on each of these objects. These claims ensure that any later task that demands conflicting effects on any one of the objects is suspended. Any subsequently created task whose demands do not conflict with the claims made by the **withonly** task need not wait.

The task body of a **withonly** may include any number of nested **withonly**s (or **withth**s) in a fully recursive manner. The effects of a task include the effects of all its sub-tasks, so a **withonly** must specify the effects not only of all serial code in the task body, but also of any nested **withonly**s that create sub-tasks. However, tasks that create sub-tasks often do not need to use any **with**s, because the actual work and therefore the effects only occur in the sub-tasks. Frequently, therefore, **with**'s only occur in the lowest-level tasks.

We now illustrate the use of these constructs with a simple example. The following Update routine creates a task to update a SharedMatrix. The created task can run concurrently with tasks accessing other shared objects.

```
void SharedMatrix :: Update() {
    withth { this->write_matrix(); } (SharedMatrix *this; ) {
        /* code to update the matrix */
    }
}
```

This code uses a **withth** to create a task accessing the SharedMatrix. This code's meaning is identical to that of the following code, which uses a **withonly** to create the task, then a **with** to demand the data.

```
void SharedMatrix :: Update() {
    withonly { this->write_matrix(); } (SharedMatrix *this; ) {
        with { this->write_matrix(); }
        /* code to update the matrix */
    }
}
```

3.3 Without

The final Jade construct, **without**, allows the programmer to specify that a task has completed a declared effect, and therefore no longer needs to access the corresponding shared object. We say that a **without** *releases* a task's claims or demands on the shared objects in its effect specification. Syntactically, this construct is similar to the **with** construct:

without { effect specification }

A **without** construct dynamically enclosed in a task specifies that the task body's remaining computation will perform none of the effects in the **without**'s effect specification. Programmers use **without** to reduce the enclosing task's specified set of 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** completes. In the absence of the **without** these tasks would have had to wait until the enclosing task terminated, when an implicit release of the task's claims and demands is done.

3.4 Summary of Use

The following general rules define the legal use of claims, demands, and releases:

- Each task's first access to a shared object must be (dynamically) preceded by a **with** construct demanding the token that corresponds to that object and properly specifying the type of effect that will be made.

- Every release of a shared object must occur only after the last access to that shared object by the closest (dynamically) enclosing task.

- Every **withonly** must claim all the tokens that any code in its task body demands (via **with**) or claims (via nested **withonly**s).

4 Hierarchical Shared Objects

Programmers frequently organize the data of an object in a hierarchical fashion, as a method of hiding complexity, structuring access to the data, and promoting modularity. Different parts of the program may access the data at different levels of the hierarchy. Because Jade adopts a data-oriented approach to concurrency, it is natural for the concurrency pattern of a Jade program to assume the same hierarchical structure as the data on which it operates. A Jade program may therefore preserve the complexity hiding and modularity advantages of the original serial program. In this section we show how the programmer uses hierarchical effect specifications to create concurrency patterns that match the hierarchical structure of the data.

Let us reconsider the matrix update routine described above. Suppose that the programmer can decompose the matrix update into a set of independent column updates. To exploit the concurrency available between column updates, the programmer must be able to express the Update operation's column-oriented data usage pattern. The programmer therefore creates a token for each column:

```
class SharedColumnMatrix {
        token _token;
        token _column_token[N];
        double elements[N][N];
    public:
        SharedColumnMatrix();
        void read_column(int i) { _column_token[i].rd(); };
        void write_column(int i) { _column_token[i].wr(); };
        . . .
}
```

Because the column tokens' data are part of matrix token's data, there is a hierarchical relationship between the column tokens and the matrix token. The programmer declares this hierarchical relationship when the matrix is created using the token's sub_token operation:

```
SharedColumnMatrix :: SharedColumnMatrix() {
    /* Other matrix initialization code */

    for (int i = 0; i < N; i++) {
        _column_token[i].sub_token(_token);
    }
}
```

We say that the column tokens are *sub-tokens* of the matrix token, and the matrix token is their *super-token*. Any token that is part of a token hierarchy is called a *hierarchical token*.

The matrix is now expressed as a hierarchical shared object which may be viewed either as a single shared object (the matrix), or as a collection of lower-level shared objects (the columns). Because the matrix has been decomposed in this way, the programmer can implement an update operation which exploits concurrency between column operations on the same matrix. The new update task can perform the column updates concurrently:

```
void SharedColumnMatrix :: Update() {
    withonly { this->write_matrix(); } (SharedColumnMatrix *this; ) {
        for (int i = 0; i < N; i++) {
            withth { this->write_column(i); }
                (SharedColumnMatrix *this; int i; ) {
                /* code to update column i */
            }
```

```
        }
    }
}
```

This example contains three hierarchies: a data hierarchy (i.e., the matrix is decomposed into columns), a token hierarchy (which is an abstraction of the data hierarchy), and a task hierarchy (i.e., the Update task is hierarchically decomposed into column update tasks). The match between the data and token hierarchies allows the programmer to express naturally the available concurrency of the Update operation. In addition, the use of the token hierarchy is a natural data hiding technique: it allows a high-level task to specify its accesses at a high level. The task's effect specification need not change if the lower-level decomposition of the data changes.

The Jade implementation correctly synchronizes accesses at different levels of the hierarchy. The programmer can demand data at different levels – the Jade implementation serializes a task demanding any of the column tokens with any task demanding the matrix token. Also, as the preceding example demonstrates, programmers can claim data (via **withonly**) at one level of the hierarchy and demand the data (via **withth** or **with**) at a lower level. The rule governing the use of the claims (the third rule in Section 3.4) thus must be adjusted slightly to account for the use of hierarchical tokens:

- For each token that its task body claims or demands, a **withonly** must claim either that token or any token above it in its token hierarchy.

We have shown how Jade programmers exploit concurrency within a shared object's operation. Jade's data-oriented approach to hierarchical concurrency, however, also allows the programmer to express the concurrency available across operations. The following example, which applies several Update operations to SharedColumnMatrixes, exploits concurrency between matrix operations (the Update operations on A and B run completely in parallel) and concurrency within a matrix operation (operations on different columns of the same matrix run in parallel).

```
void UpdateBoth()
{
    SharedColumnMatrix A;
    SharedColumnMatrix B;

    A.Update();
    B.Update();
    A.Update();
}
```

Figure 1: Concurrency Structure of UpdateBoth

This example expresses yet another form of concurrency between the two Updates of A. As each column update task of the first update of A completes, the corresponding column update task of the second update of A can execute. The second column update task does not depend on the progress of any of any other column update task.

It is difficult to express the synchronization required to exploit this form of concurrency in control-oriented parallel languages without destroying the structure of the program. A programmer using such a language would typically not attempt to exploit the concurrency available between separate matrix updates. Instead, the programmer would ensure that successive column updates occur serially by using a full barrier between successive matrix updates. The barrier would waste concurrency by unnecessarily serializing updates to different columns. In contrast, the Jade implementation only enforces the task serializations required to correctly preserve the serial semantics. Figure 1 illustrates the concurrency pattern of the preceding matrix update example.

5 Refining Effect Specifications

The preceding SharedColumnMatrix example demonstrates how programmers use token hierarchies to create task hierarchies. Jade programmers can also use hierarchical tokens to refine incrementally a single task's effects. It is sometimes impossible to determine the exact set of shared objects that a task will access at the start of the task. A token hierarchy allows a task to refine its effect specification as more information becomes available. When it is created, the task claims a hierarchical token representing all the possible objects that the task might access. As the task runs, the set of potentially accessed objects becomes smaller. The task can then refine its effect specification by claiming a token lower in the hierarchy and releasing the higher-level token. This refinement of an effect specification is most useful if a large amount of computation is required to determine exactly which shared objects the task will access.

As an example, suppose a program manipulates a SharedColumnMatrix, and the ProcessUpdates operation on the matrix executes a sequence of updates to individual columns of the matrix. ProcessUpdates is passed an array that provides information on the updates to be processed, but it still requires a lengthy computation (represented below by the function ComputeWhichColumn) to determine the exact column to update.

The simplest implementation of ProcessUpdates creates a task to do a column update only after determining the exact column to update. Such an implementation reduces the potential concurrency by serializing all of the calls to ComputeWhichColumn. On the other hand, if the programmer includes the ComputeWhichColumn computation in each task, then it is impossible to determine exactly which column the task will update at task creation time. The programmer therefore uses the hierarchical relationship between the matrix token and the column tokens to refine the effect specification of the task in two stages. The task first declares a write effect on the entire matrix using the matrix token. When the task determines the exact column to update, it refines its effect to indicate the exact column that it will updated:

```
void SharedColumnMatrix :: ProcessUpdates(UpdateList, UpdateLength)
UpdateInfo UpdateList[];
int UpdateLength;
{
    for (int i = 0; i < UpdateLength; i++) {
        UpdateInfo& update;

        update = UpdateList[i];
        withonly {
            this->write_matrix();
        } (SharedColumnMatrix *this; UpdateInfo& update; ) {
            int column;
            column = ComputeWhichColumn(update);
            with { this->write_column(column); }
            without { this->write_matrix(); }
            this->UpdateColumn(column, update);
        }
    }
```

```
}
```

This process of refining the a task's effects as information becomes available is useful for creating concurrent tasks as early as possible. In this case the initial effect specification of each task is highly imprecise, but still provides the information that the task will only touch columns in the indicated matrix. Therefore, later tasks that do not access the matrix can execute concurrently with the update task. However, a subsequent update task which modifies a column of the same matrix will be held up. This task can execute only after the first task demands its column and releases its claim on the entire matrix. If the tasks modify different columns, they can then execute concurrently. If the tasks modify the same column, they must execute serially.

6 Efficient Generation of Concurrency

In the preceding sections, we have described how programmers use hierarchical tokens to express the concurrency within and across operations on hierarchical data structures. In this section, we describe how Jade extracts the concurrency pattern dynamically during the program execution. In particular, we will show that not only does hierarchical concurrency help the programmer build modular programs, it also makes the parallelization more efficient.

As described above, the effect specifications of the Jade constructs are executable pieces of code which compute the effects of tasks at run-time. Thus, a Jade program can exploit concurrency which can only be discovered while the program is running. However, because the effects are computed at run-time, the Jade implementation must determine the dependences among the tasks incrementally as the program executes. The execution of a Jade program may therefore be viewed as a process of dynamically creating and executing a task graph.

Consider the simple case where only **withth**'s are used. The predecessors of a new task in the task graph are exactly the preceding tasks whose effects conflict with the effects of the new task. The fundamental rule for executing the task graph is that all of a task's predecessors in the graph must be completed before that task can execute. However, a task cannot possibly be executed until it has been created and added to the task graph. Therefore, the speed and the order in which the graph is built may limit the concurrency that can be exploited in a program.

Let us illustrate the issues related to Jade's dynamic generation of concurrency with an example. Below we give a "flat" version of the Update operation which creates the column update tasks directly, rather than creating a task which in turn creates the column tasks:

```
void SharedColumnMatrix :: Update() {
    for (int i = 0; i < N; i++) {
        withth { this->write_column(i); }
            (SharedColumnMatrix *this; int i; ) {
            /* code to update column i */
        }
    }
}
```

In this flat version, all of the tasks are spawned serially in the main thread of control that calls the Update operation. Thus, the use of the flat Update may delay the generation and exploitation of concurrency available in code following the call to Update.

Suppose the programmer wishes to Update two distinct matrices. First, all of the tasks are spawned serially in the main thread of control that calls the Update operations. Even if all the tasks can be executed in parallel, Amdahl's Law indicates that the serial spawning bottleneck could significantly limit the concurrency in the program. Second, the tasks on the first matrix must be spawned before those on the second can be created. Thus, the concurrency between operations on the two matrices cannot be exploited until after the spawning overhead associated with the first matrix.

Hierarchical concurrency provides a natural solution to reduce the serial spawning overhead. Consider the update of two distinct matrices using the hierarchical version of Update given in Section 4. Because the Update of the first matrix creates a task to spawn the sub-tasks, the main thread of control can proceed to the Update of the second task, which also immediately creates a task. The serialization of spawning the column update tasks is reduced by allowing the spawning to proceed in parallel. Not only can the creation of sub-tasks of the two matrices be overlapped, but also sub-tasks of the second update can in fact execute before all the sub-tasks of the first are spawned. This is possible because the side effects of all sub-tasks have been summarized in the effect specification of the parent tasks. Thus, hierarchical concurrency parallelizes the spawning overhead and relaxes the task spawning order to expose concurrency early.

Let us now consider the case where the parent tasks operate on the same data, as in the function UpdateBoth defined in Section 4. Figure 2 displays the process of generating the concurrency in UpdateBoth. The shaded area within each task box indicates the work required to create the task by its parent task. The shaded lines indicate serial execution within a task. For example, the main thread of control follows a path through the shaded areas of the three main Update tasks, indicating that its only work is to create these three tasks. Each of the main Update tasks, in turn, serially creates the column tasks for that update. The threads of sub-task creation can proceed in parallel.

However, unlike the updates to B, the sub-tasks of the second update to A cannot

Figure 2: Dynamic Generation of Concurrency in UpdateBoth

execute immediately after being created. Since the top-level task of the first update to A claims the entire matrix A, tasks of the second update to A must at least wait until the first parent task has completed. (We depict this dependence with edges between the top-level task of the first update of A and the sub-tasks of the second update of A.) The parent task completes when it has created all the sub-tasks; it need not wait for its children to complete. In this way, parallelism between sub-tasks of the first and second updates to A is exposed as soon as all the sub-tasks of the first update are spawned. This process may be thought of again in terms of refinement. While the sub-tasks have not yet all been created, the **withonly** task holds claims on the possible effects of sub-tasks. As the sub-tasks are created, they refine the effect information by indicating which task has effects on which data. When all the sub-tasks have been created and the effect information is fully refined, the claims held by the **withonly** are no longer needed. They are naturally released when the **withonly** task completes.

7 Conclusion

Jade is a language for parallelizing programs written in imperative programming languages. The key insight behind Jade is that concurrency is best expressed indirectly with data usage information. In this paper we have demonstrated the usefulness of Jade's data-oriented approach for expressing the concurrency available in programs that manipulate hierarchical data structures.

Jade programmers can create token hierarchies which match the hierarchical structure of the data. Such a token hierarchy allows the programmer to express tasks' effects at different levels, corresponding to how the tasks access the data. In this way, the programmer can express the effects of a task at the most appropriate level for that task. Token hierarchies can also be useful for refining the effect specification of a task which incrementally narrows down the set of data it can potentially access. When a programmer expresses tasks' effects in this hierarchical manner, the Jade implementation can fully exploit the natural concurrency available within and among operations on hierarchically structured data.

References

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

[2] J. J. Dongarra and D. C. Sorenson. A portable environment for developing parallel FORTRAN programs. *Parallel Computing*, 5(1 & 2):175–186, 1987.

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

[4] M. S. Lam and M. C. Rinard. Coarse-grain parallel programming in Jade. In *Proceedings of the Third ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.

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