Abstract
Jade is a data-oriented language for exploiting coarse-grain parallelism. A Jade programmer simply augments a serial program with assertions specifying how the program accesses data. The Jade implementation dynamically interprets these assertions, using them to execute the program concurrently while enforcing the program's data dependence constraints. Jade has been implemented as extensions to C, FORTRAN, and C++, and currently runs on the Encore Multimax, Silicon Graphics IRIS 4D/240S, and the Stanford DASH multiprocessors. In this paper, we show how Jade programmers can naturally express hierarchical concurrency patterns by specifying how a program uses hierarchically structured data.

1 Introduction

Jade is a data-oriented language for expressing coarse-grain concurrency. Instead of using explicit control constructs to express the concurrency available in a program, a Jade programmer augments a serial program with Jade constructs that declare how the various sections of the program access data. The Jade implementation uses this information to execute the program concurrently while enforcing the program's underlying data dependence constraints. Jade programmers therefore create coarse-grain parallel programs that preserve the semantics of the original serial programs.

A Jade programmer first divides a sequential program up into tasks. Tasks interact

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

through accesses to *shared objects*. The programmer summarizes each task's accesses by specifying which shared objects the task will read or write. The Jade implementation uses this information to relax the program's serial execution order; tasks without conflicting accesses can execute concurrently. Because each task's access specification is determined dynamically, Jade programs can exploit data-dependent concurrency available only at run time.

We may contrast Jade's data-oriented approach to concurrency with the control-oriented approach provided by many parallel programming languages [1, 2, 3, 5]. Control-oriented languages typically provide low-level constructs for creating and synchronizing parallel tasks. These constructs provide fine control over the concurrent behavior of a program. However, it can be difficult to create and maintain parallel programs which contain such a low-level specification of the concurrency structure. Programmers using these low-level constructs must often establish explicit synchronization connections between logically unrelated modules which access the same data. These connections violate the modular structure of the original serial program, making the parallel program much harder to understand and modify.

Jade, with its data-oriented constructs, provides a less familiar but conceptually higher-level approach to concurrency. Jade programs 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 dependence constraints. By requiring only local data usage information, Jade promotes modularity in parallel programs.

Jade's data-oriented approach to concurrency means that the design of a program's data structures determines its concurrency pattern. Programs with simple data structures often have simple concurrency patterns such as dynamic task graphs and pipelining [4]. This paper shows how programs which use hierarchical data structures have more sophisticated concurrency patterns with nested levels of parallelism. For example, the Jade implementation can generate a parallel tree traversal from a specification of how a program accesses nodes and subtrees. A Jade program that accesses 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 columns in the matrix.

The organization of this paper is as follows. We first briefly review 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 incrementally refine tasks' specified accesses. 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 uses a synchronization abstraction called *tokens* to express how the task will access data. 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 *access specification operation* applied to the corresponding token.

Each token has the *rd* and *wr* access specification operations, specifying, respectively, a read access and a write access. Two access specifications *conflict* if they refer to the same token and at least one of them specifies a write access. Tasks with conflicting access specifications must execute in the original serial execution order to maintain the serial semantics of a program.

Jade programmers usually encapsulate data and the tokens which represent the data as an object. (We use the C++ class notation below to make this encapsulation more apparent.) Each such object provides its own access specification interface, implementing its access specification operations internally using its private tokens. In the simplest such use of tokens, the programmer associates one token with each object and augments the object's interface with the appropriate access specification operations. The following `SharedMatrix` class illustrates such a use. This class directly translates its access specification operations to access operations on its private token. (We describe the `df_rd`, `df_wr`, `no_rd` and `no_wr` access operations in sections 4.1 and 4.2.)

```
class SharedMatrix {
       token _token;
       double elements[N][N];
    public:
       void read_matrix() { _token.rd(); };
       void write_matrix() { _token.wr(); };
       void df_read_matrix() { _token.df_rd(); };
       void df_write_matrix() { _token.df_wr(); };
       void no_read_matrix() { _token.no_rd(); };
       void no_write_matrix() { _token.no_wr(); };
       .  .  .
}
```

This practice of associating one token with each object works well for data that is created and accessed as a unit. In section 5 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 how tasks 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.

3.1 Basic Constructs

Jade programmers use the **withonly** construct to declare how a section of code will access shared objects:

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

The Jade implementation creates a task when it executes a **withonly** construct. The `task body` contains the serial code to be executed when the task runs. The `access specification` section declares how the task will access shared objects. This section is an arbitrary piece of code containing access specification operations. The implementation executes this piece of code when the task is created to determine which shared objects the task will read and/or write. This section may contain dynamically resolved variable references and control flow constructs such as conditionals, loops and function calls. The programmer may therefore use information available only at run time when declaring how a task will access data.

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

The **withonly** construct indicates that the task body will execute *with only* the accesses declared in the access specification section. The Jade implementation uses the access specification to determine when two tasks can execute concurrently. Tasks whose access specifications do not conflict are free to execute concurrently. Conversely, tasks with conflicting access specifications must execute in the original serial execution order. To enforce this restriction, the Jade implementation does not allow a task to execute until all earlier tasks (in the underlying sequential execution order) with conflicting access specifications have completed.

We illustrate the use of the **withonly** construct with a simple example. The following `Update` routine creates a task to update a `SharedMatrix`:

```
void Update(SharedMatrix *M) {
```

```
    withonly { M->write_matrix(); } do (SharedMatrix *M; ) {
        /* code to update the matrix */
    }
}
```

The created task can run concurrently with tasks accessing other shared objects, but must execute serially with respect to other tasks which access `M`. For example, the following code fragment takes pointers to two `SharedMatrix`es `A` and `B`, and either updates `A` twice or updates both `A` and `B`:

```
Update(A);
if (flag > 0)
    Update(B);
else   Update(A);
```

When `flag` is positive, the two `Update` tasks can execute concurrently because they modify different matrices. If `flag` is not positive, then both `Updates` operate on the matrix `A`. The Jade implementation will therefore serialize the tasks because their access specifications conflict.

The **withonly** construct and the `rd` and `wr` access operations described so far support a simple model of parallel computation in which 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 many parallel applications only need to synchronize at task boundaries, some applications have more complex concurrency patterns requiring periodic inter-task synchronization. To express these more complex synchronization patterns, Jade provides constructs that allow the programmer to precisely specify when a task will actually access shared objects. We next show how Jade programmers can use these advanced constructs to express concurrency patterns with periodic inter-task synchronization.

4 Advanced Constructs

Allowing tasks to synchronize only at task boundaries can unnecessarily serialize computation in two cases: when a task's last access to a shared object occurs long before the task completes, and and when a task's first access to a shared object occurs long after the task starts running. The following code fragment provides a concrete example of both forms of unnecessary serialization:

```
void Combine(SharedMatrix *L, *M, *N) {
    withonly {
```

```
        L->read_matrix(); M->read_matrix();
        N->read_matrix(); N->write_matrix();
    } do (L, M, N) {
        /* code initializing N to L */
        /* code combining N with M */
    }
}

Combine(A,B,C); Update(A);
Combine(B,A,D);
```

This code fragment generates three tasks. The tasks must execute sequentially to preserve the serial semantics. The first unnecessary serialization comes from the fact that `Combine(A,B,C)` never accesses `A` after it uses `A` to initialize `C`. Therefore, the code that combines `C` and `B` should be able to execute concurrently with the `Update(A)` task.

The second unnecessary serialization comes from the fact that `Combine(B,A,D)` does not access `A` until it finishes the initialization of `D` to `B`. Therefore, this initialization should be able to execute concurrently with the `Update(A)` task.

One way to achieve full concurrency is to make the `Combine` routine generate two tasks. This solution is inferior because the modification is not motivated by examining the code of the second task itself. The need to manage the two new serial tasks may also cause extra overhead. The solution presented below bypasses these problems by allowing the `Combine` tasks to more precisely specify when they will actually access `A` and `B`.

4.1 Completed Accesses

To eliminate the first unnecessary serialization, the programmer must be able to express the fact that once the `Combine(A,B,C)` task has initialized `C` to `A`, the rest of the task will no longer access `A`. A programmer can express this fact using the **with** construct, which allows a programmer to dynamically modify a task's access specification so that it more accurately reflects the task's future behavior:

with { `access specification` } **continue**;

As in the **withonly** construct, the `access specification` section is an arbitrary piece of code containing access specification operations. Programmers use the `no_rd` (no future read) and `no_wr` (no future write) access operations in this section to declare that a task will no longer access the corresponding shared object:

```
void Combine(SharedMatrix *L, *M, *N) {
    withonly {
        L->read_matrix(); M->read_matrix();
```

```
        N->read_matrix(); N->write_matrix();
    } do (L, M, N) {
        /* code initializing N to L */
        if (L != M) with { L->no_read_matrix(); } continue;
        /* code combining N with M */
    }
}
```

The combination of a **with** construct and `no_rd` and/or `no_wr` access operations allows the programmer to dynamically reduce a task's access specification. This reduction may eliminate conflicts between the task executing the **with** and other tasks occurring later in the sequential execution order. The later tasks may therefore be able to execute as soon as the `with` completes. In the absence of the `with` these tasks would have had to wait until the first task finished. In our example the **with** construct allows the combination of `C` with `B` to execute concurrently with the `Update(A)` task.

4.2 Deferred Accesses

To eliminate the second unnecessary serialization, the programmer must be able to express the fact that the `Combine(B,A,D)` task will not access `A` until it has finished the initialization of `D` to `B`. In this case we say that the task's access to `A` is *deferred* because the task does not access `A` until long after it starts running. The programmer can express such deferred accesses at task creation time using the `df_rd` (deferred read) and `df_wr` (deferred write) access specification operations. The programmer indicates when the task will actually access the shared object using the **with** construct. We say that such a **with** construct converts the deferred access to an *immediate* access.

```
void Combine(SharedMatrix *L, *M, *N) {
    withonly {
        L->read_matrix(); M->df_read_matrix();
        N->read_matrix(); N->write_matrix();
    } do (L, M, N) {
        /* code initializing N to L */
        if (L != M) with {
            L->no_read_matrix();
            M->read_matrix();
            } continue;
        /* code combining N with M */
    }
}
```

A task with a deferred access to a shared object can run even if previous tasks have yet to complete conflicting accesses to that object. The task will wait for the previous tasks to complete their accesses when it executes a **with** statement which converts the deferred access to an immediate access. Deferred accesses allow the Jade implementation to concurrently execute the non-conflicting parts of tasks with access conflicts. In our example deferred accesses allow the code that combines `D` with `A` to execute concurrently with the `Update(A)` task.

5 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 access 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 Update(SharedColumnMatrix *M) {
    withonly { M->df_write_matrix(); } do (SharedColumnMatrix *M; ) {
        for (int i = 0; i < N; i++) {
            withonly { M->write_column(i); }
                (SharedColumnMatrix *M; 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 access 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. For example, the Jade implementation will serialize two tasks if one declares a column access and the other declares a conflicting matrix access. As the preceding example demonstrates, programmers can specify a deferred access at one level of the hierarchy and then specify an immediate access at a lower level.

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 `SharedColumnMatrix`es, 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).

```
Update(A);
Update(B);
Update(A);
```

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

The synchronization between the individual column update tasks necessary to exploit this concurrency is difficult to express in control-oriented parallel languages. A programmer using such a language would typically not attempt to exploit all of 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. This barrier would waste available concurrency by unnecessarily serializing updates to different columns. In contrast, the Jade implementation only enforces the serializations required to correctly preserve the serial semantics. Figure 1 illustrates the concurrency pattern of the preceding matrix update example.

6 Refining Access 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 access specification. 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 access specification as more information becomes available. When it is created, the task declares a deferred access to 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 access specification by specifying an access to a token lower in the hierarchy and cancelling the access to the higher-level token. This refinement of an access specification is most useful

Figure 1: Concurrency Structure of `UpdateBoth`

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 access specification of the task in two stages. The task first declares a deferred write access to the entire matrix using the matrix token. When the task determines the exact column to update, it refines the task's access specification to indicate the exact column that it will updated:

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

```
        update = UpdateList[i];
        withonly {
            M->write_matrix();
        } do (SharedColumnMatrix *M; UpdateInfo& update; ) {
            int column;
            column = ComputeWhichColumn(update);
            with { M->write_column(column);
                   nv M->write_matrix(); } cont;
            M->UpdateColumn(column, update);
        }
    }
}
```

This process of refining a task's access specification as information becomes available is useful for creating concurrent tasks as early as possible. In this case each task's initial access specification 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 declares the column access and cancels its access to 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.

7 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 how hierarchical concurrency makes the parallelization more efficient.

As described above, the `access specification` sections of Jade constructs are executable pieces of code which dynamically compute how tasks will access shared objects. Thus, a Jade program can exploit concurrency which can only be discovered while the program is running. However, because the access specifications are computed at runtime, the Jade implementation must dynamically determine the inter-task dependences 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 in which the program serially creates tasks which only declare immediate accesses. Each new task's predecessors in the task graph are exactly those

earlier tasks whose access specifications conflict with the access specification of the new task. To correctly execute this task graph, all of a task's predecessors must complete before the task itself can execute. However, a task cannot possibly execute until it has been created and added to the task graph. Therefore, the speed with which the implementation builds the graph may limit the amount of expoitable concurrency.

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 Update(SharedColumnMatrix *M;) {
    for (int i = 0; i < N; i++) {
        withonly { M->write_column(i); }
            do (SharedColumnMatrix *M; int i; ) {
            /* code to update column i */
        }
    }
}
```

In this flat version, all of the tasks are created 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 created serially in the main thread of control that calls the `Update` operations. Even if all the tasks generated by a single invocation of the `Update` Amdahl's Law indicates that a serial task creation bottleneck could significantly limit the exploitable concurrency in the program. Second, the tasks which update the first matrix are all created before the tasks which update the second matrix. Thus, the concurrency available between operations on the two matrices cannot be exploited until the implementation creates all of the tasks which update the first matrix.

Hierarchical concurrency provides a natural solution to reduce the serial task creation overhead. Consider the update of two distinct matrices using the hierarchical version of `Update` given in Section 5. Because the `Update` of the first matrix immediately creates a task to create the individual column update tasks, the main thread of control immediately proceeds to the `Update` of the second task. This enables the concurrent creation of the column update tasks from *both* `Update` operations. In fact, the column update tasks from the second `Update` operation can execute before the implementation creates all of the tasks from the first `Update` operation. This is possible because the access specification of each parent `Update` task accurately summarizes the access specifications of its column update tasks. Thus, hierarchical concurrency parallelizes the task creation overhead and relaxes the task creation 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 5. Figure 2 displays the process of generating the concurrency in `UpdateBoth`. The shaded area within each task box indicates the work performed by the parent task to create that 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 update tasks for that invocation of `Update`. The sub-task creation threads can all execute concurrently.

However, unlike the updates to `B`, the column update tasks of the second update to `A` cannot execute immediately after being created. Since the top-level task of the first update to `A` has declared a deferred access to the entire matrix `A` and may potentially access `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 created. This process may be thought of again in terms of refinement. While the sub-tasks have not yet all been created, the `withonly-do` task declares a deferred access that includes all the possible accesses of sub-tasks. As the sub-tasks are created, they refine the access specification by indicating which task has accesses which data. When all the sub-tasks have been created and the access specifications have been fully refined, the deferred access to the entire matrix declared by the **withonly** is no longer needed, and naturally goes away when the `withonly` task completes.

8 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 token hierarchies allow the programmer to express tasks' accesses at different levels, corresponding to how the tasks access the data. In this way, the programmer can express a task's accesses at the most appropriate level for that task. Token hierarchies can also be useful for refining the access specification of a task which incre-

Figure 2: Dynamic Generation of Concurrency in `UpdateBoth`

mentally narrows down the set of data it can potentially access. When a programmer expresses how tasks access data in this hierarchical manner, the Jade implementation can fully exploit the natural concurrency available within and among operations on hierarchically structured data. The resulting hierarchical concurrency patterns also naturally make more efficient the generation of the dynamic task graph in a Jade program.

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.