Jade: A High-Level, Machine-Independent Language for Parallel Programming

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

1 Introduction

The past decade has seen tremendous progress in computer architecture and a proliferation of parallel machines. The advent of high-speed RISC processors has led to the development of both low-cost high-performance engineering workstations and tightly-coupled massively parallel supercomputers. Advances in VLSI technology have enabled the construction of cost-effective engines dedicated to special functions such as signal processing and high-resolution graphics. The imminent widespread availability of high-speed networks will theoretically enable practitioners to bring the combined power of all these machines to bear on a single application.

The software for heterogeneous machines can be quite complex because of the need to manage the low-level aspects of the computation. For a program to execute correctly, the software must decompose an application into parallel tasks and correctly synchronize these tasks. Typically, the software must also use message-passing constructs to manage the transfer of data between machines; if the machines have different data formats, the software must perform the format translation required to maintain a correct representation of the data on each machine. The nondeterministic nature of parallel program execution complicates the development of such software: programmers often spend their time struggling to reproduce and fix subtle, nondeterministic bugs.

The previous paragraph only addresses the difficulties associated with constructing a correct parallel program. While correctness is of paramount importance, parallel programs must also be efficient and portable. Simultaneously achieving these last two goals is difficult because they often conflict. Efficient execution often requires complicated software that controls the parallel execution at a very low level. This software must take into account such machine-specific details as the degree of parallelism available in the hardware, the processor speed, the communication latency and bandwidth, and the amount of memory on each processor. If machine-specific code appears in the application program, it will not port efficiently to new hardware platforms. Portability is crucial if applications software is to keep up with the rapid development of parallel hardware.

1.1 High-Level Languages

Heterogeneous machines demand high-level programming paradigms that insulate the programmer from the complexities associated with explicitly parallel programming. A programming language is the usual way to present the

This research was supported in part by DARPA contracts DABT63-91-K-0003 and N00039-91-C-0138.

paradigm to the programmer. Each language is built around a set of abstractions that simplify and structure the way the programmer thinks about and expresses a parallel algorithm. These abstractions typically offer the programmer a safer, more specialized model of parallel computation that can eliminate such anomalies as nondeterministic execution and deadlock.

The implementation of such a high-level language encapsulates a set of algorithms for managing concurrency. These algorithms automate the process of exploiting the kind of concurrency that the paradigm supports. In effect, each implementation is a reusable package containing the knowledge of how to efficiently use a given machine. The programmer can therefore express the computation at a higher, more abstract level and rely on the implementation to deal with the low-level aspects of mapping the computation onto the concrete machine at hand. Programs written in the language should therefore efficiently port to new generations of hardware. The wide application of parallel processing requires the leverage that high-level programming languages provide.

Languages such as Fortran 90 and C* provide a useful paradigm for programs with regular, data parallel forms of concurrency. Programmers using these languages view their program as a sequence of operations on large aggregate data structures such as sets or arrays. The implementation can execute each aggregate operation in parallel by performing the operation on the individual elements concurrently. This approach works well for programs that fit the data parallel paradigm. Its success illustrates the utility of having the language implementation, rather than the programmer, control the parallel execution and data movement.

Parallelizing compilers can also liberate programmers from the onerous task of writing low-level, explicitly parallel code. In this case, the programmer sees the abstraction of a sequential execution semantics and is largely oblivious to the parallelization process. The compiler is responsible for extracting parallelism from the code and scheduling the computation efficiently onto the target parallel machine. Like the data parallel approach, this works well for a restricted kind of parallelism: the loop-level parallelism present in scientific programs that manipulate dense matrices. The complexity of the highly tuned, machine-specific code that parallelizing compilers generate [11] illustrates the need for high-level abstractions that shield programmers from the low-level details of their computations.

Both the parallelizing compiler and the data parallel approaches are designed to exploit regular concurrency available *within* a single operation on aggregate data structures. An orthogonal source of concurrency also exists *between* operations on different data structures. In contrast to data-parallel forms of concurrency, task-level concurrency is often irregular and sometimes depends on the input data or on the results of previous computation. The tasks' computations may be heterogeneous, with different tasks executing completely different computations.

Exploiting task-level concurrency is especially important in a heterogeneous environment because of the ability to match the execution requirements of different parts of the program with the computational capabilities of the different machines. Some tasks may require special-purpose hardware, either because the hardware can execute that task's computation efficiently, or because the hardware has some unique functionality that the task requires.

No single form of concurrency is sufficient for parallelizing all applications. In general, programs will need to exploit all kinds of concurrency. The challenge is to come up with a comprehensive model of parallel computing that encompasses all of the paradigms. The components of this model will cooperate to allow programmers to express all of the different kinds of concurrency available within a single application.

1.2 Jade

Jade is a high-level, implicitly parallel language designed for exploiting coarse-grain, task-level concurrency in both homogeneous and heterogeneous environments. Jade presents the programmer with the dual abstractions of a single address space and serial semantics. Instead of using explicitly parallel constructs to create and synchronize parallel tasks, Jade programmers guide the parallelization process by providing the high-level, application-specific information required to execute the program in parallel on a heterogeneous collection of machines. The programmer must specify three things: 1) a decomposition of the data into the atomic units that the program will access, 2) a decomposition of a sequential program into tasks, and 3) a description of how each task will access data when it runs. Given this information, the implementation automatically extracts and exploits the task-level concurrency present in the computation.

The Jade implementation parallelizes the computation by identifying parts of the serial program that can execute concurrently without changing the program's result. The basic principle is that each part of a serial program reads and/or writes certain pieces of data. Consider two adjacent parts of a serial program. If each part writes pieces of data that the other part does not access, it is possible to execute the parts concurrently without changing the result that the program generates. In this case we say that the parts are *independent*. If one part writes a piece of data that the other part reads or writes, executing the two parts concurrently may cause the program to generate an incorrect result. In this case we say that the parts are *dependent*. If a parallel execution executes dependent parts in the same order as the serial execution, it will generate the same result as the serial execution.

All attempts to parallelize serial programs have been oriented towards finding independent parts to execute concurrently. Traditional parallelizing compilers, for example, statically analyze the code to find loops whose iterations are independent. The compiler can then generate an explicitly parallel program that executes loops with independent iterations concurrently.

It is also possible to find independent program parts dynamically. In this case, the program execution consists of three components: 1) a component that runs ahead of the actual execution analyzing how the program accesses data, 2) a component that finds independent program parts and schedules the computation accordingly, and 3) a component that actually executes the scheduled computation. The IBM 360/91 [1] applied this dynamic approach at the instruction level. It fetched and decoded several instructions at a time, and dynamically determined which memory locations and registers each instruction read and wrote. The execution unit detected and concurrently executed independent instructions. This aggressive approach to dynamic instruction scheduling is the basis for current superscalar microprocessors such as the SuperSPARC, the Motorola 88110 and the IBM RIOS.

Jade promotes a combined model of parallel computing in which the implementation uses both the dynamic and static approaches to parallelize a Jade program. The Jade constructs provide the necessary dependence information at the level of the coarse-grain data structures (or *objects*) that tasks manipulates. Such summary information is highly useful to a compiler, since it is very difficult to determine statically all the accesses of a coarse-grain task. In sections of the program where the tasks' data usage information is statically analyzable, the compiler can generate highly optimized, explicitly parallel code with little dynamic overhead. The remaining data- and computation-dependent concurrency can be discovered and exploited using the dynamic approach. Our current Jade implementation only extracts concurrency dynamically and does not do any compile-time analysis to exploit static concurrency.

The coarse granularity of the Jade approach allows Jade programs to quickly discover and exploit concurrency available between distant parts of the program. Exploiting coarse-grain concurrency also means that Jade can profitably amortize the communication and synchronization overhead associated with sending data and tasks to remote processors for execution. Jade is therefore suited for programs that execute on loosely coupled multiprocessors and networks of machines. We do not attempt to exploit fine-grain concurrency using Jade, because of the dynamic overhead in the current implementation associated with extracting concurrency from the data usage information.

Every Jade program is a serial program augmented with Jade constructs providing information about how the serial program accesses data. Because Jade preserves the semantics of this serial program, Jade programs execute deterministically. This property considerably simplifies the process of debugging parallel applications. Jade programs access data using a single, global address space. When a Jade program executes in a message-passing environment, the implementation, not the programmer, analyzes each task's data usage requirements to generate the messages necessary to communicate the proper data between tasks.

It is possible to implement Jade on a wide variety of machines: uniprocessors, multiprocessors, distributed memory machines, networks of workstations, and heterogeneous systems of any size and topology. Because the programmer does not directly control the parallel execution, each Jade implementation has the freedom to apply machine-specific optimizations and implementation strategies.

Jade is designed to cooperate with approaches designed to express other forms of concurrency. For example, Jade's serial semantics enables the direct application of parallelizing compiler techniques to Jade programs. Jade therefore promotes an integrated model of parallel computation in which a high-level programming language allows programmers to exploit coarse-grain, data-dependent concurrency, while the compiler exploits fine-grain, concurrency statically available within tasks. The difficulty of applying compiler optimizations to explicitly parallel code [6] limits the amount of concurrency that compilers can extract from programs written in explicitly parallel languages.

2 Language Overview

Because of Jade's sequential semantics, it is possible to implement Jade as an extension to a sequential programming language. This approach allows programmers to learn how to use Jade quickly, and preserves much of the language-specific investment in programmer training and software tools. Currently, we have implemented a version of Jade as an extension to C. We first present the basic Jade constructs and then discuss a detailed example that illustrates their use.

2.1 Object Model

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

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

The first declaration defines a statically allocated shared vector of `doubles`, while the second declaration defines a reference (pointer) to a dynamically allocated shared vector of `doubles`. Programmers use the `create_object` construct to dynamically allocate shared objects; this construct takes as parameters the type and number of allocated items in the shared object. Given the declaration of `B` presented above, the programmer could dynamically allocate a shared vector of 100 `doubles` as follows:

```
B = create_object(double, 100);
```

2.2 Basic Construct

Jade programmers use the `withonly-do` construct to identify a task and to specify how that task will access data. Here is the general syntactic form of the construct:

```
withonly { access declaration } do
         (parameters for task body) {
  task body
}
```

The `task body` section contains the serial code executed when the task runs. The `parameters` section declares a list of variables from the enclosing environment that the task body may access. The `access declaration` section summarizes how the task body will access the shared objects.

The access declaration is an arbitrary piece of code containing *access specification statements*. Each such statement declares how the task will access a given shared object. For example, the `rd` (read) statement declares that the task may read the given object, while the `wr` (write) statement declares that the task may write the given object. Each access specification statement takes one parameter: a reference to the object that the task may access. The task's access declaration section may contain dynamically resolved variable references and control flow constructs such as conditionals and loops. Jade programmers may therefore exploit dynamic concurrency that depends either on the input data or on the result of previous computation. Because the access specification section can also contain procedure calls, the programmer can use procedures to build access specification abstractions. Each such abstraction could encapsulate, for example, the access specification operations that declare how a procedure invoked in the task body will access data.

Since the parallelization and data movement are based on access specifications, it is important that the access specifications be accurate. The Jade implementation checks the accuracy of the access specifications by dynamically checking each task's accesses to shared objects. If a task attempts to perform an undeclared access, the implementation will generate a run-time error identifying the statement that generated the access. The Jade type system has a mechanism that allows each task to perform the dynamic access check once for each object. Because the overhead is then amortized over many object accesses, in practice the total checking overhead is negligible.

Each shared object represents a unit of synchronization. If one task writes part of a shared object and another task accesses that object, the implementation executes the tasks serially. Furthermore, the implementation preserves the serial semantics but executing the tasks in the same order as in the original serial execution. Tasks that access different shared objects or read the same objects can execute concurrently. The programmer must therefore allocate objects at a fine enough granularity to expose the desired amount of concurrency.

In a message-passing environment, each shared object is also a unit of communication. If a task will access an object, the implementation will either move (on a write access) or copy (on a read access) the entire object to the machine on which the task will execute. If the task only accesses a small part of the object, the communication overhead may negate any performance gains from concurrency. The programmer must therefore allocate objects at a fine enough granularity to minimize superfluous communication.

3 An Example

In this section we present a programming example. This example introduces the basic Jade model of parallel execution. It illustrates both how Jade programmers express irregular, dynamically determined concurrency and how the implementation extracts this source of concurrency.

3.1 Sparse Cholesky Factorization

The sparse Cholesky factorization algorithm factors a sparse, symmetric, positive definite matrix A into the form LL^T, where L is lower triangular. Because the matrix is symmetric, the algorithm only needs to store the lower triangle of the matrix. The factorization algorithm then repeatedly updates the data structures that represent this lower triangle. We assume that the matrix is stored using the data structure in Figure 1. The matrix itself is represented as a vector of columns. The non-zero elements of each column of the matrix are stored in packed form in variable-length vectors. The sparsity pattern of the matrix is represented using one contiguous vector of integers that contains the row indices of each of the matrix's nonzero elements, as they appear in column-major order. Each column consists of a pointer to the packed vector containing its non-zero elements and an offset into the row index vector that tells where the row indices for that column begin. Figure 2 provides the C data structure declarations for the sparse matrix data structure.

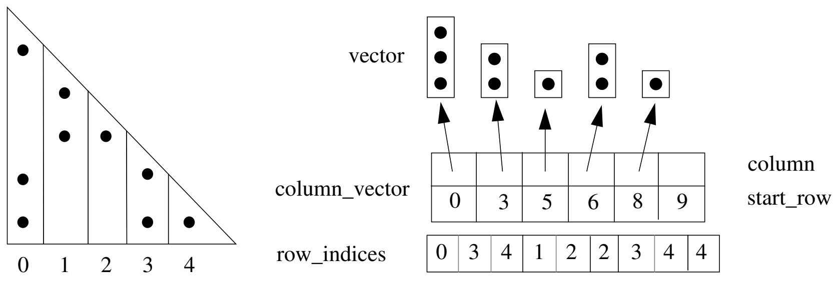

Figure 1: Sparse Matrix Data Structure

The serial factorization algorithm processes the columns of the matrix from left to right. The algorithm first performs an internal update on the current column; this update divides the column by the square root of its diagonal element. After this internal update the current column reaches its final value. The algorithm then uses the current column to update some subset of the columns to its right. For a dense matrix the algorithm would update all of the

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

Figure 2: C Data Structure Declarations

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

Figure 3: Sparse Cholesky Factorization Algorithm

columns to right of the current column. For sparse matrices the algorithm omits some of these updates because they would not change the updated column. Figure 3 contains the serial code for this algorithm.

One way to parallelize this algorithm is to turn each update into a task. In this case the parallel computation consists of a coarse-grain task graph of update tasks. Each update task performs either an `InternalUpdate` or an `ExternalUpdate`; there are precedence constraints between tasks that enforce the dynamic data dependence constraints of the underlying computation. Figure 4 shows a sparse matrix and its corresponding parallel task graph. The nodes of the task graph represent update tasks, while the edges represent precedence constraints. Each `InternalUpdate` task is labelled with the index of the column that it updates, while each `ExternalUpdate` task is labelled with the indices of the two columns that it accesses. The structure of this task graph depends on the sparsity pattern of the matrix to be factored. Because the sparsity pattern of the matrix depends on the input, a programmer parallelizing this algorithm must be able to express dynamic, data-dependent concurrency.

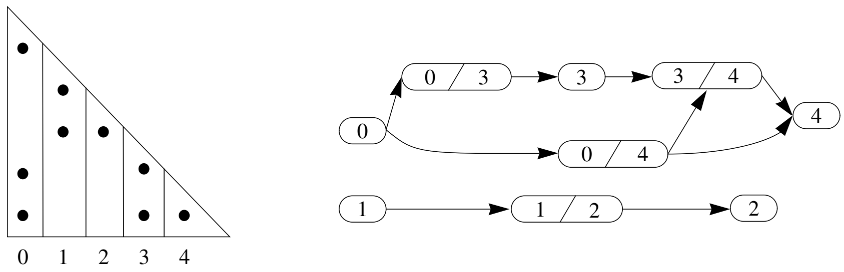

Figure 4: Dynamic Task Graph

3.2 Using Jade to Parallelize Sparse Cholesky Factorization

The first step in parallelizing a program using Jade is to determine the design of the shared objects. For some applications the programmer may need to explicitly decompose objects into smaller allocation units to allow the application to concurrently write disjoint parts of the object, or replicate certain objects to expose more concurrency. In the sparse Cholesky example the objects are already allocated at the appropriate granularity. The programmer only needs to modify the matrix declaration to identify the references to shared objects, as shown in Figure 5.

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

Figure 5: Jade Data Structure Declarations

The programmer next augments the sequential code with `withonly-do` constructs to identify the tasks and specify how they access shared objects. To parallelize the sparse Cholesky factorization code, the programmer adds two `withonly-do` constructs to the original serial code; each construct identifies an update as a task. Figure 6 contains the Jade version of the sparse Cholesky factorization algorithm. The first `withonly-do` construct uses the `rd_wr` and `rd` access specification statements to declare that the `InternalUpdate` will execute *with only* reads

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

Figure 6: Jade Sparse Cholesky Factorization

and writes to the `i`'th column of the matrix and reads to the column array and row index data structures. The second `withonly-do` construct uses the same access specification statements to declare that the `ExternalUpdate` will execute *with only* reads and writes to the `r[j]`'th column of the matrix, reads to the `i`'th column of the matrix, and reads to the column vector and row index data structures.

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

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

4 Advanced Language Features

We have, so far, described the basic programming model in Jade. Here we describe briefly some of the advanced constructs of the Jade language that allow a Jade programmer to achieve more sophisticated parallel behavior.

4.1 Higher-level Access Specifications

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

4.2 Hierarchical Concurrency

Programmers may create hierarchical forms of concurrency in a Jade program by dynamically nesting `withonly-do` constructs. The task body of a `withonly-do` construct may execute any number of `withonly-do` constructs, in a fully recursive manner. The access specification of a task that hierarchically creates child tasks must declare both its own accesses and the accesses performed by all of its child tasks.

4.3 More Precise Access Specifications

As described so far, Jade constructs support only a limited model of parallel computation in which all synchronization takes place at task boundaries. Two tasks may either execute concurrently (if none of their accesses conflict) or sequentially (if any of their accesses conflict). The `withonly-do` construct allows no partial overlap in the executions of tasks with data dependence conflicts. This limitation makes it impossible to exploit pipelining concurrency available between tasks that access the same data.

Jade relaxes this restricted model by allowing the programmer to provide more precise information about when a task actually accesses data. Jade provides this functionality with an additional construct, `with-cont`, and the additional access specification statements `df_rd`, `df_wr`, `no_rd`, and `no_wr`. The general syntactic form of the `with-cont` construct is:

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

As in the `withonly-do` construct, the `access declaration` section is an arbitrary piece of code containing access declaration statements. These statements update the task's access specification, allowing the specification to reflect more precisely how the remainder (or continuation, as the `cont` keyword suggests) of the task will access shared objects.

The `df_rd` and `df_wr` statements declare a *deferred* access to the shared object. That is, they specify that the task may eventually read or write the object, but that it will not do so immediately. Because the task cannot immediately access the object, it can execute concurrently with earlier tasks that do access the object. When the task reaches the point where it will access the object, it must execute a `with-cont` construct that uses the `rd` or `wr` access declaration statements to convert the deferred declaration to an *immediate* declaration. This immediate declaration then gives the task the right to access the object. In order to preserve the serial semantics, the Jade implementation may suspend the task at the `with-cont` construct until preceding tasks with conflicting accesses to the object have completed.

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

5 A Digital Video Application in Jade

We now provide an example which illustrates how to use some of the more advanced Jade features. In this section we show how to write a simple digital video imaging program that runs on the High-Resolution Video (HRV) machine at Sun Laboratories [9].

Figure 7 contains a simplified block diagram of the HRV machine. This machine contains four SPARC processors and five i860 processors as graphics accelerators that drive a high-definition television monitor. The SPARCs interface to several input devices; the imaging application uses a video camera. The HRV machine has both a control bus and a high-bandwidth internal bus over which the SPARC and i860 processors communicate. The SPARCs use an ATM network to communicate with the rest of the computational environment.

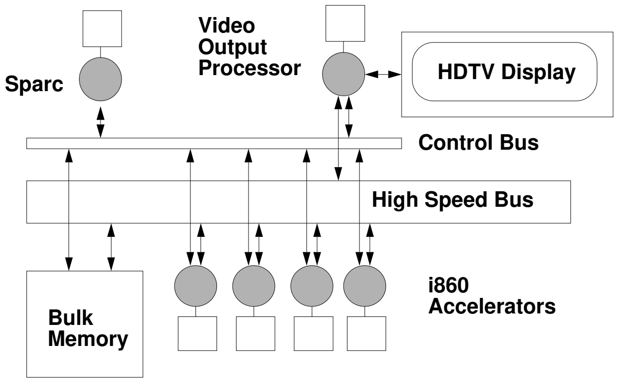

Figure 7: HRV Machine

The digital video processing program is a simple application. It applies an image transformation to the camera input and displays it on the HDTV display. The SPARCs repeatedly acquire compressed camera frames and ship them to the i860-based processors. These processors decompress the frames, apply the transformation, and display the transformed frames in windows on the HDTV monitor.

The explicitly parallel version of this application first uses operating-system primitives to start up a process on each processor and to set up communication channels between the processes. The programmer must sequence the process creation correctly, or the communication channel setup will fail. When the communication has been initiated, the processes use message-passing primitives to transfer image frames between the processes.

To write this application in Jade, the programmer first starts with a serial program that performs the acquisition, decompression, image transformation, and display. This program manipulates a set of buffers that store successive compressed camera frames. The program decompresses the images in these buffers to an uncompressed frame, then performs the image transformation on the uncompressed frame. Figure 8 presents the data structures used to store the buffers and uncompressed frames, while figure 9 presents a high-level description of the serial program. The program loops through the set of buffers that `b` points to; `cb` is the current buffer. The decompress routine uncompresses the frame in the current buffer and stores it in `f`. The transform routine operates on the uncompressed frame `f`. To display the image, the program writes it into the frame buffer `fb`.

The programmer next converts this serial program to a Jade program. As in the sparse Cholesky factorization example, the objects are already allocated at the correct granularity, so the programmer only needs to modify the buffer and frame declarations to identify the references to shared objects, as shown in Figure 10. The programmer next augments the sequential code with Jade constructs to identify the tasks and specify how they access shared objects. Figure 11 contains the Jade version of the program. To parallelize the digital video application, the programmer adds two `withonly-do` constructs to the original serial code. The first construct identifies each frame acquisition as a task, the second construct identifies the decompression, transformation and display of each frame as another task. Because the SPARCs control the camera, the frame acquisition tasks must run on a SPARC. Jade allows programmers

```
#define SIZE_BUFFER 1024
#define SIZE_FRAME 8192
#define NUM_BUFFERS 4
typedef char *buffer;
typedef buffer buffers[NUM_BUFFERS];
typedef char frame[SIZE_FRAME];
typedef char *frame_buffer;
```

Figure 8: Buffer Data Structure Declarations

```
view(b, fb)
buffers b;
frame_buffer fb;
{
  buffer cb;
  int icb;
  frame f;

  icb = 0;
  while (TRUE) {
    cb = b[icb];
    GetFrame(cb);
    DecompressFrame(cb, f);
    TransformFrame(f);
    DisplayFrame(f, fb);
    icb = (icb + 1) % NUM_BUFFERS;
  }
}
```

Figure 9: Code for Digital Video Processing

to declare that tasks require a specific resource associated with one of the processors. In this case the programmer uses the `resource` construct to declare that these tasks must execute on a processor that controls the camera. Each of these tasks writes the current buffer, which will hold the compressed frame that the task acquires.

```
#define SIZE_BUFFER 1024
#define SIZE_FRAME 8192
#define NUM_BUFFERS 4
typedef char shared *buffer;
typedef buffer buffers[NUM_BUFFERS];
typedef char frame[SIZE_FRAME];
typedef char shared *frame_buffer;
```

Figure 10: Jade Buffer Data Structure Declarations

```
view(b, fb)
buffers b;
frame_buffer fb;
{
  buffer cb;
  int icb;

  icb = 0;
  while (TRUE) {
    cb = b[icb];
    withonly { wr(cb); } resource (CAMERA) do (cb) {
      GetFrame(cb);
    }
    withonly { rd(cb); df_wr(fb); } resource (ACCELERATOR) do (cb, fb) {
      frame f;
      DecompressFrame(cb, f);
      TransformFrame(f);
      with { wr(fb); } cont;
      DisplayFrame(f, fb);
    }
    icb = (icb + 1) % NUM_BUFFERS;
  }
}
```

Figure 11: Jade Code for Digital Video Processing

The tasks created at the second `withonly-do` construct should run on the graphics accelerators. The programmer therefore declares that these tasks require the accelerator resource. Each of these tasks reads the current compressed frame, and eventually writes the transformed frame into the frame buffer. Because the write to the frame buffer only takes place after the decompress and transform phases, the `withonly-do` construct declares a deferred write access to the frame buffer instead of an immediate write access. The task body then uses a `with-cont` construct to convert this deferred access to an immediate access just before the frame is displayed. As written, the program serializes the tasks only as they display successive frames. The use of an immediate (rather than deferred) access declaration would have serialized the decompression and transformation of successive frames.

As part of the programming process, the programmer moved the declaration of the uncompressed frame `f` into the second task body. Each task therefore has its own private copy of `f`, which no other task can access. The programmer therefore does not declare that the task accesses `f` – programmers only declare tasks' externally visible accesses to shared objects. This privatization of `f`, along with the use of the deferred access declaration, enables the concurrent decompression and transformation of successive frames.

This application illustrates several aspects of Jade. First, because the program contains no low-level invocation and communication constructs, it will trivially port to new generations of the HRV machine with more hardware resources and generalized communication interfaces. Because the communication startup code is packaged inside the Jade implementation, the applications programmer does not have to deal with the low-level details of correctly sequencing startup and initiating communication. Because Jade preserves the abstraction of a single address space, programmers do not need to use low-level message-passing constructs to manage the movement of data through the machine – the Jade implementation manages this movement. For a coherent image to appear, the frames must be displayed in the same order as they came off the camera. Because the Jade implementation preserves the order of writes to the frame buffer, it automatically generates the synchronization required display the frames in this order. By encapsulating the synchronization, the Jade implementation frees programmers from having to deal with complexities such as deadlock and nondeterministic execution that are associated with explicitly parallel environments.

6 Discussion and Comparison

Systems that provide software support for parallel programming vary widely in the functionality that they provide for the communication of data and expression of parallelism. Jade addresses these problems by providing the high-level abstractions of a single shared address space and implicit concurrency derived from data usage information. In the following sections, we describe some of the alternate ways of addressing these issues and explain the implications of the approach taken in the Jade language.

6.1 Communication of Data

Existing systems provide a variety of levels of abstraction for the communication of data in heterogeneous parallel systems. At the lowest level of abstraction are tools that provide only basic message-passing capabilities for sending data between processors, but hide the heterogeneity from the programmer. Software packages such as PVM [10] manage the translation issues associated with heterogeneity. This allows the same message-passing code to run on machines with different internal data formats – for example, a heterogeneous set of workstations connected by a network. While this set of tools provides a low-level abstraction, it has the greatest degree of generality and is useful as a foundation for high-level tools.

Linda [3] provides a higher-level abstraction in the form of a shared tuple space. Rather than communicating via messages, tasks communicate via data that are added to and removed from the shared tuple space. Linda's tuple space operations port across a variety of parallel hardware environments. While the Linda tuple space provides the high-level abstraction of a common address space, its low-level operations leave the programmer with the difficult task of correctly synchronizing the program execution. General-purpose implementations of Linda's associative tuple space operations have been inefficient. Linda compilers therefore use global static analysis that attempts to recognize common higher-level idioms that can be more efficiently implemented. This approach leads to a model of parallel programming in which the programmer encodes the high-level concepts in low-level primitives only to have the compiler attempt to reconstruct the higher-level structure.

The next layer of software support implements the abstraction of shared memory on a set of message-passing machines. Shared Virtual Memory systems such as Ivy [8] and Munin [4] use the virtual memory hardware to implement a page-based coherence protocol. The page-fault hardware is used to detect accesses to remote or invalid pages, and the page-fault handler can be used to move or copy pages from remote processors. Shared virtual memory systems deal with the raw address space of the application and typically have no knowledge of the types of the data stored in various locations in memory. Each parallel program must therefore have the same address space on all the different machines on which it executes. This restriction has so far limited these systems to homogeneous collections of machines (except for a prototype described in [12]). The comparatively large size of the pages also increases the probability of an application suffering from excessive communication caused by false sharing (when multiple processors repeatedly access disjoint regions of a single page in conflicting ways).

Object-oriented parallel languages such as Amber [5] and Orca [2] provide shared memory at an object level. These systems support the shared memory abstraction by allowing methods to be invoked on any existing object from any processor. Some systems may implement this abstraction by running the method on the remote processor that owns the object; other systems bring a copy of the object to the local processor for execution of the method.

Jade also supports the shared memory abstraction at the level of user-defined objects. Object-based shared memory systems can deal effectively with heterogeneity, because the system knows the types of each data object and can do the necessary translations as objects are sent from one processor to another. The problem of false sharing does not arise, since data is communicated at the level of the user-defined object, rather than at the level of pages. A main difference between Jade and the object-based parallel languages discussed in the previous paragraph is that Jade supports the concept that a unit of computation can access arbitrarily many objects. This makes it easy for the programmer to express algorithms that need to atomically access multiple objects. The languages described above make each method invocation into a separate unit of computation; each method can access only the receiver directly. If a programmer using such a language needs to write a program that atomically accesses multiple objects, he must insert additional synchronization code.

6.2 Expression of Parallelism

Most parallel programming systems provide constructs that programmers use to explicitly create and control parallelism. For example, many systems provide a threads package that allows for the creation of threads and the synchronization of the threads via locks and barriers. The programmer breaks up the program into a set of tasks, each to be executed by a separate thread. The programmer enforces inter-task data dependence constraints by inserting synchronization primitives into tasks that access the same data. This synchronization code can create new explicit connections between parts of the program that access the same data, destroying the modularity of the progam and making it 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.

Some parallel programming languages augment a serial programming language with explicitly parallel constructs such as fork/join, par_begin/par_end, and doall statements. These constructs allow the programmer to spawn several independent tasks; the program then blocks until all the spawned processes terminate. Data parallel languages take a similar approach in which all parallelism is created in constructs that operate in parallel on data aggregates. These kinds of parallel constructs cannot express irregular, task-level parallelism. Additionally, they may require the programmer to restructure his original code in order to place concurrently executable pieces of code in the same parallel construct. It is the programmer's responsibility to ensure that there are no race conditions between the independent tasks created by these constructs.

Jade provides constructs for implicitly expressing parallelism and synchronizing tasks. Jade programmers only provide information about how tasks access shared data, and tasks execute in parallel if they have no conflicting accesses to data. This implicit, data-oriented approach to parallelism has a number of implications:

- Deterministic Execution - Because they execute identically to the serial program on which they are based, all Jade programs execute deterministically. Jade programs can therefore be debugged in the same way as serial programs, without the difficulties associated with reproducing timing-dependent bugs that can occur in explicitly parallel programs.

  The Jade serial semantics also restricts the expressive power of the language. Some parallel algorithms are inherently nondeterministic. It is impossible to write these algorithms in Jade. We have consciously limited the expressive power of Jade to provide a higher-level, more tractable programming model.

- Familiar Programming Model - The high-level abstraction of serial execution, in conjunction with the abstraction of a shared address space, together provide a model of programming that is familiar to programmers and convenient for them to use.

- Irregular and Dynamic Concurrency - Jade programs can naturally express both regular and irregular patterns of concurrency. There are no explicit control points where concurrency is created or ended. Instead, tasks execute concurrently based solely on whether their accesses conflict. Because the Jade access specifications are executed at run-time, Jade programs can express dynamic concurrency patterns that depend on the input data. However, because of the overhead of dynamically extracting concurrency in this way, Jade is not intended for expressing fine-grain concurrency.
- Modularity - The programmer does not disrupt the modularity of the code by inserting synchronization operations that control the interactions between between tasks that access the same data. Jade programs only contain local information about how each task accesses data. It is the responsibility of the Jade implementation, and not the programmer, to generate the synchronization required to correctly execute the program.
- Maintainability - When the programmer modifies a parallel program, the program's underlying concurrency pattern may change. If the programmer is using an explicitly parallel programming language, these changes to the concurrency pattern may entail many changes to the explicit synchronization code throughout the program. A Jade programmer simply updates the the data usage information to reflect any changes in the data accesses of tasks. The Jade implementation can then generate the new concurrency pattern using its encapsulated synchronization algorithm.
- Portability - Jade's high-level programming model can be implemented on a wide variety of parallel machines. Other languages expose enough of the low-level details of the targeted hardware platform to preclude the possibility of porting applications without source code modifications.

7 Jade Implementation

In this section we briefly discuss the Jade implementation. Because Jade adopts such a high-level programming model, the implementation handles much of the complexity involved in running a parallel application. Several aspects of the Jade implementation involve executing the application *correctly* according to the Jade programming model:

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

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

- Dynamic Load Balancing. The Jade implementation keeps track of which processors may be idle and dynamically assigns executable tasks to processors which may become idle. Dynamic load balancing is especially important in a heterogeneous environment in which some machines execute faster than others.

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

The list of activities of the implementation illustrates the utility of Jade. In effect, the Jade implementation encapsulates a set of algorithms for managing parallel computation. Programmers painlessly reuse these algorithms every time they write a Jade program. Without Jade, programmers would have to manage the associated problems themselves. In a large or complicated parallel program, this management software could dominate the application development process. Jade allows programmers to concentrate on the algorithmic aspects of the particular application at hand rather than on the systems issues associated with mapping that application onto the current parallel machine.

8 Applications Experience

We have implemented Jade on shared memory parallel processors and on both heterogeneous and homogeneous message-passing environments. There are no source code modifications required to port Jade applications between these platforms. Our shared memory implementation of Jade runs on the Silicon Graphics 4D/240S multiprocessor, and on the Stanford DASH multiprocessor [7]. The workstation implementation of Jade uses PVM as a reliable, typed transport protocol. Currently, this implementation of Jade runs on the SPARC-based SUN Microsystems workstations, and MIPS-based systems including the DECStation 3100 and 5000 machines, the Silicon Graphics workstations and the Stanford DASH multiprocessor. Jade applications can use any combination of these kinds of workstations in the course of a single execution. We have also implemented Jade on the Intel iPSC/860 using the NX/2 message passing system. Jade also runs on the High Resolution Video (HRV) Workstation from SUN Microsystems Laboratories described earlier.

To test the Jade paradigm, we have implemented several computational kernels, including a sparse Cholesky factorization algorithm and the Barnes-Hut algorithm for solving the N-body problem. The digital image processing application described earlier is an example of a program that uses the diverse resources of a heterogeneous machine.

We have also implemented several complete applications in Jade. The following describes our experience in developing several benchmarks chosen to illustrate different aspects of Jade. We ran the applications in three different parallel environments: the Stanford DASH shared-memory multiprocessor, an Intel iPSC/860 and a Mica multiprocessor. The Mica multiprocessor is an array of Sparc ELC boards connected by a private 10Mbit/sec Ethernet. This machine represents a standard, widely available environment consisting of a network of workstations. For each of the applications we used data sets that reflected the way the applications would be used in practice.

8.1 Make

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

8.2 Liquid Water Simulation

Water is a program derived from the Perfect Club benchmark MDG that evaluates forces and potentials in a system of water molecules in the liquid state. For the problem sizes that we are running, almost all of the computation takes place inside the $O(n^2)$ phase that determines the pairwise interactions of the n molecules. We therefore execute only that phase in parallel and run the $O(n)$ phases serially. To parallelize this program in Jade we first restructured several of the program's data structures and then added 24 Jade constructs – 22 `with-cont` constructs and 2 `withonly-do` constructs. These modifications increased the size of the program from 1216 to 1358 lines of code.

Figure 12 shows a plot of the speedup for the Jade Water program in the three different parallel environments. The speedups are given relative to the serial code with no Jade overhead. The data set for the Water program consists of 2197 molecules, and simulates these molecules for four timesteps. The presented speedup curve omits the time spent in an initialization phase. In practice the application would run for at least one hundred time steps, and the initialization time would be negligible compared to the rest of the compute time.

8.3 String

String is a program from the Stanford Geophysics department. It uses seismic travel time inversion to construct a velocity model of the geology between two oil wells. The seismic data are collected by firing non-destructive seismic sources in one well and recording the seismic waves digitally as they arrive at the other well. The inter-well travel times of the waves can be measured from the resulting seismic traces. The application iteratively computes the velocity of the seismic waves in the media between the two wells by inverting the travel time equation (velocity = distance/time).

The main part of the application shoots rays from one well to the other, using a discretized velocity model to compute the travel time. The application then calculates the difference between the computed travel time and the measured travel time for each ray. It backprojects the differences along the ray paths, and updates the velocity model to account for the differences. The application repeats this process of shooting rays and updating the velocity model until the differences are reduced to an acceptable level. This produces a velocity model which can be related to the underlying geology between the two wells.

To parallelize this application in Jade we replicated one of the main data structures to allow the program to trace the rays concurrently. After the rays are traced the program performs a parallel reduction of the replicated data structure to derive the improved velocity model. These modifications involved the insertion of 35 `with-cont` constructs and 3 `withonly-do` constructs, which increased the size of the program from 2591 lines of C to 2909 lines of C augmented with Jade constructs.

Figure 13 shows a plot of the speedup for the Jade String program in the three different parallel environments. The speedups are given relative to the serial code with no Jade overhead. The speedup curve presented is for the application's entire computation, including initial and final I/O. The data set for the String program is from an oil field in West Texas and discretizes the 185 foot by 450 foot velocity image at the resolution of 1 foot by 1 foot. It shoots approximately 37,000 rays per iteration and runs six iterations.

8.4 Search

Search is a program from the Stanford Electrical Engineering department. This program simulates the interaction of electron beams with various solids. In particular, the program uses a Monte-Carlo technique to simulate the elastic scattering of each electron from the electron beam into the solid. The result of this simulation is used to measure how closely an empirical equation for electron scattering matches a full quantum mechanical expansion of the wave equation

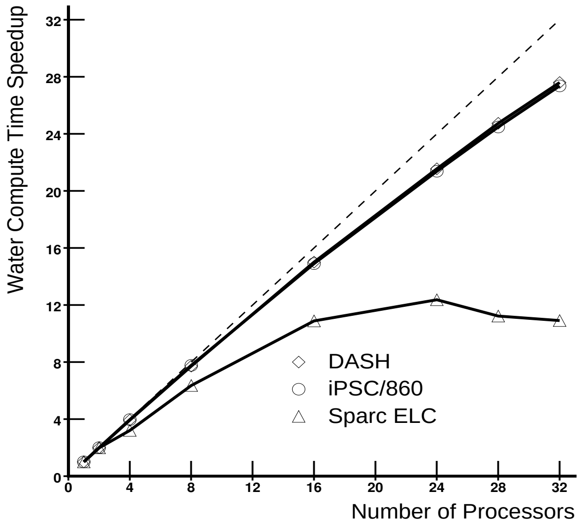

Figure 12: Speedups for Water

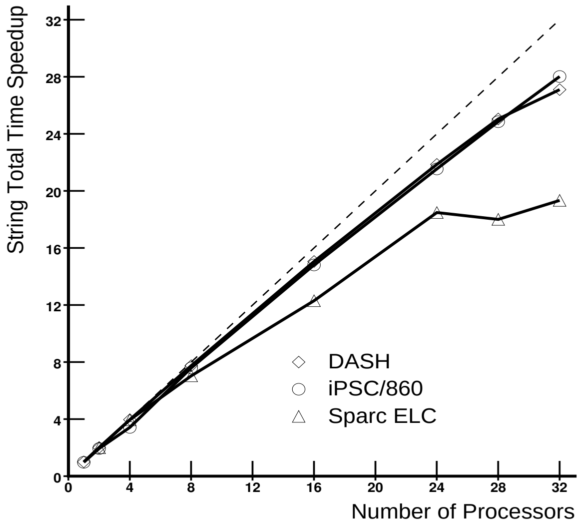

Figure 13: Speedups for String

stored in tables. The main computation simulates six different solids at 10 initial beam energies. Each solid-energy data point requires the Monte-Carlo simulation of 5,000 electron trajectories. All of the electron trajectories and all of the solid-energy points can execute concurrently. The program contains nine `with-cont` constructs and one `withonly-do` construct, and consists of 676 lines of code.

Figure 14 shows a plot of the speedup for the Jade Search program in the three different parallel environments. The speedups are given relative to the serial code with no Jade overhead. The speedup curve presented is for the application's entire computation.

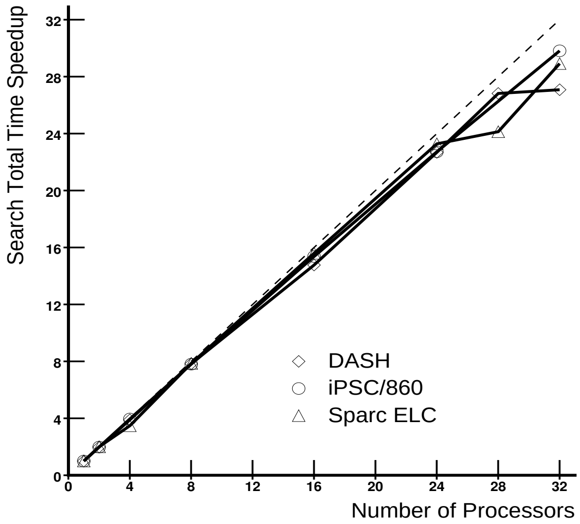

Figure 14: Speedups for Search

9 Conclusions

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

We have implemented Jade in a wide variety of computational environments, from tightly-coupled shared memory multiprocessors through networks of workstations to heterogeneous systems with special-purpose accelerators connected by high-speed networks. Jade programs execute without modification on all of these computational platforms. Our initial applications experience demonstrates that Jade effectively supports the development of efficient coarse-grain parallel programs.

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

Acknowledgments

Jennifer Anderson participated in the initial design and implementation of Jade. We thank Edward Rothberg for helping us with the sparse Cholesky factorization code and Jason Nieh for advising us on porting Jade to the iPSC/860. We acknowledge the support of Jim Hanko, Eugene Kuerner, and Duane Northcutt, who helped us port Jade to the HRV workstation. We thank Dave Ditzel for use of the Mica network of Suns and David Chenevert for helping us use the Mica environment. Ray Browning wrote the Search program. We thank Jerry Harris for letting us use the String program. We thank Caroline Lambert and Mark Van Schaack for helping us to understand the structure of the String program.

References

[1] D.W. Anderson, F.J. Sparacio, and F.M. Tomasulo. The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling. *IBM Journal of Research and Development*, 11(1):8–24, January 1967.

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

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

[4] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

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

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

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

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

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

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

[11] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, Oct. 1991.

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