Flexible Language Constructs for Large Parallel Programs

MATT ROSING[1] AND ROBERT SCHNABEL[2]

[1]*Pacific Northwest Laboratory, Richland, WA 99352*
[2]*University of Colorado, Boulder, CO 80309*

ABSTRACT

The goal of the research described in this article is to develop flexible language constructs for writing large data parallel numerical programs for distributed memory (multiple instruction multiple data [MIMD]) multiprocessors. Previously, several models have been developed to support synchronization and communication. Models for global synchronization include single instruction multiple data (SIMD), single program multiple data (SPMD), and sequential programs annotated with data distribution statements. The two primary models for communication include implicit communication based on shared memory and explicit communication based on messages. None of these models by themselves seem sufficient to permit the natural and efficient expression of the variety of algorithms that occur in large scientific computations. In this article, we give an overview of a new language that combines many of these programming models in a clean manner. This is done in a modular fashion such that different models can be combined to support large programs. Within a module, the selection of a model depends on the algorithm and its efficiency requirements. In this article, we give an overview of the language and discuss some of the critical implementation details. © 1994 John Wiley & Sons, Inc.

1 INTRODUCTION

The goal of the research described in this article is to develop easy-to-use, efficiently implementable language constructs for writing large data parallel numerical programs for distributed memory (multiple instruction multiple data [MIMD]) multiprocessors. By data parallel algorithms we mean those where identical or similar operations are performed concurrently on different sections of a typically large data structure. Such computations are typical in many scientific computations, such as computational fluid dynamics algorithms, although their data parallel structure does not necessarily imply that the algorithms are single instruction multiple data (SIMD), or that it is easy to parallelize them efficiently [1]. Distributed memory multiprocessors appear to be the main candidates for scalable, high performance parallel computers. Current examples of distributed memory machines include the Intel iPSC series of hypercubes and mesh-connected machines, Thinking Machine's CM5, and networks of workstations used as multiprocessors.

Although distributed memory machines show great promise for high performance computation, they are currently difficult to program. The difficulty arises from the low level details the program-

Received August 1992
Revised November 1993

mer must handle regarding communications, synchronization, and process control. Raising the level of these operations from the message level sends and receives found in many current systems to the point where most of these details are handled implicitly will make programming distributed memory machines much easier. However, the efficiency of the resulting code generated from the model must not be adversely affected or else few programmers will be interested in using the model. For example, simulating a uniform shared memory will likely be too inefficient.

The research described here attempts to reduce this mismatch among the target machine, the languages used, and the underlying model of typical data parallel algorithms. A key issue addressed in this research is the development of a language model that supports the expression of large, modular parallel programs. Such programs may consist of multiple levels and/or multiple phases of parallelism, and may use different models of parallelism in different portions of the program. Furthermore, as is typical in numerical programs, the overall efficiency of the program may depend on the efficiency of a small portion of the code. Thus, it is important that the user can easily switch between high level abstract models that are easy to program but may not compile as efficiently as desired, and low level models that give the user a lot of control over the hardware. As an example, many programs are fine tuned to make better use of message passing facilities. Such improvements can easily cut the execution time in half yet such improvements are only required on a small portion of the code. Thus, it is important for a language to support multiple models in a clean manner. The expression of such complex, parallel algorithms has received little consideration so far, with the exception of Oracle [2].

The language described here is based in part on many recently developed languages, including our DINO language [3], that have been proposed for writing numerical programs on distributed memory machines (for a few of the more relevant languages see references 2, 4–13). These languages appear to be converging in terms of the underlying parallel programming model used [14]. This model is primarily a data parallel one with a little support for functional parallelism in some cases. It has four main parts. First, single or multiple dimensional arrays of virtual processors may be declared in a shape that best fits the algorithm [3, 4, 15]. Second, single or multiple dimensional arrays of data may be distributed (mapped) across these virtual processors [3–5, 12]. This distributed data is usually treated as a single global object and all accesses are made with respect to the global name space. Third, communications are generated by accessing the distributed objects. The communications can be implicit, similar to shared memory, or explicit based on sends and receives. Finally, some model is used for specifying the computation. Here there appear to be two classes of approaches, either an annotated sequential program approach or an explicitly parallel approach. In this language, we use the explicitly parallel approach, as it appears to have greater flexibility in the models it can support on MIMD machines.

Within the explicitly parallel approach, one option is to use a general single program multiple data (SPMD) synchronization model. In this method, parallelism is usually specified at a per-task level, and communication is generally specified with explicit sends and receives but with the low level details of message typing, buffering, channels, and other aspects handled by the compiler [3]. A second option is to use an SIMD synchronization model in which virtual processors effectively synchronize at all communications [16]. In this model, parallelism is generally specified at a fully data parallel level, and all communication is implicitly generated by the compiler.

Some languages, such as ELP [17] and Modula2* [18], combine some aspects of both the SIMD and SPMD models. ELP is a language designed specifically to program the PASM parallel computer [19], an experimental machine that supports both SIMD and MIMD computational modes. ELP supports the ability to declare blocks of code to run in either an SPMD or SIMD mode and can change between the two in a single instruction, as this is supported in the hardware. Parallelism in Modula2* is specified using either SIMD or SPMD parallel loops. Modula2* supports virtual processors and the ability to nest parallel constructs to any depth. ELP does not support this because the hardware does not. Modula2* does not support any form of synchronization in the SPMD mode so communications must be done with libraries in this case. ELP supports a barrier synchronization when all processors write to a mono variable, but, based on the literature, there is no other form of synchronization in the SPMD mode.

One issue that the languages developed so far do not address is writing very large programs. This is the main issue addressed in this research. Although most of the above mentioned languages are suitable for expressing simple algorithms (up to a few hundred lines), they are less suitable for writing large, modular, multiple-phase parallel programs. This is partly due to their inability to define and tie together modules that are independent of the rest of the program.

A large factor that contributes to the inability of expressing large parallel programs is the restrictiveness of the programming model supported in each of these languages. Almost every language follows just one of the models described above. Each of these models has tradeoffs among ease of use, expressiveness, and efficiency.

For example, if the language follows the explicitly parallel SPMD model, with parallelism specified at a per-task level, then there are two tradeoffs the user must face when writing large programs. First the user usually must explicitly put in synchronization and communications. Second, the explicitly parallel nature of the SPMD model may cause difficulties in efficient compilation if there are many more tasks than physical processors. When there are more tasks than processors, the compiler and run-time system must emulate a large virtual machine. The overhead in doing this may become prohibitive as the number of tasks increases. To handle this, the programmer may have to write one task per physical processor. Although this gives the programmer more control of the machine, it tends to be more difficult to do.

On the other hand, if the language follows the explicitly parallel SIMD model, with parallelism specified at a fully data parallel level, then the user has the advantages of simple synchronization and of being able to efficiently specify large numbers of processes that match the data parallelism and are independent of the target machine. The SIMD model provides the compiler with more information than the SPMD model to efficiently contract many virtual processes into fewer real processes, thus overcoming the constraint of knowing the number of available processors. But this model is significantly limited in its expressiveness, due to the lock step execution enforced by the SIMD model, and is therefore insufficient for expressing many parallel algorithms.

Finally, the sequential model using only distributed data annotations has the advantage that the programmer does not specify any communications or synchronization, and the disadvantage that it sometimes may be hard to obtain an efficient parallel program from the sequential specification. First, it is still an open research question to determine how effectively and broadly one can derive efficient parallel programs from sequential specifications, using dependency analysis and data distribution annotations. Second, there are some efficient parallel algorithms, such as pipelined algorithms with nonunit block sizes, that appear to be especially difficult to express in or derive from a sequential program.

The tradeoffs between efficiency and ease of use described above are typical decisions that must be made in developing computationally intensive numerical programs. These tradeoffs are caused, in part, by the inability of optimizing compilers to generate code that is as efficient as that of the user. The typical solution to this type of problem is to use the high level model for the bulk of the computation, where the efficiency is not that critical and the quality of the optimized code is acceptable, and use a lower level model for the portion of code where the resulting efficiency is very important. Examples of this two-model techniques include using assembly code in Fortran, using Fortran within HPF, and using vector statements within Fortran vectorizing compilers. One of the goals of this research is to support multiple models having various ease of use and efficiency tradeoffs, and also support an easy transition between the various models.

Another reason for supporting multiple models in a single language is that many large numerical programs have modules that fit different models. For example, many kernels of numerical programs are highly structured, fine grained computations that fit the explicitly parallel SIMD model, whereas the overall computation structure as well as selected kernels may be less structured and fit the coarse grained, explicitly parallel SPMD model.

For these reasons, it appears to us that a language for specifying large, modular parallel numerical programs needs to support at least two types of models, an SPMD model for coarse grained parallel computations, and some model that efficiently expresses fine grained data parallel computations. Such a language should also support both implicit and explicit models of communication. Most importantly, though, it must also provide an easy method of switching between these models. This is a critical factor in making the language flexible enough to handle a wide range of programming models and to give the user control over the machine where a high degree of efficiency is required. These needs form the main motivation for this research.

This article describes a language that supports these models. The methodology used by the programmer is centered around the large data structures that are typically the source of computationally intensive scientific programs. The programmer first identifies the data structures to be operated on in parallel and the nature of this parallelism. This defines a virtual machine and how the major data structures are distributed onto it, and is described in Section 2. Next, based on the parallel operations that are to be applied to the data, the user decides which synchronization model to use. For operations that are very tightly synchronized, the SIMD model is used. For other, less synchronous models, the SPMD model is used. The SPMD model can also be further refined using different types of variables. This is described more fully in Section 4. Each module built using this methodology is encapsulated by a special type of composite procedure that includes the virtual machine, the distributed data structures, the synchronization model, and the code executed by each processor. The final task of the programmer is to combine composite procedures in a structured framework to support a complex virtual machine that may include the nesting of parallel modules, combining different modules to execute concurrently, and changing between parallel modules for different phases of the algorithm. This is described in Section 3.

This article gives an overview of a new language, called Dino2, that addresses these issues. Dino2 is a successor to the DINO language [3], and shares with it the fact that it is a superset of the C language. The two languages also have similar capabilities for expressing distributed data and arrays of virtual processors, but their methods for expressing parallel computations, communications, and synchronization are very different.

The remainder of the article is organized as follows. Section 2 describes the basic concepts of a Dino2 module, and Section 3 describes how modules can be combined to form large complex programs. Section 4 describes the different synchronization models that are supported for the modules, whereas Section 5 describes the language support for these constructs. Section 6 describes some implementation details for these synchronization models and for communication. Section 7 offers some brief conclusions. More details of the language and potential implementation issues are provided in Rosing [20].

2 VIRTUAL PARALLEL MACHINES

A Dino2 module is built around a virtual parallel machine defined by the user. A virtual machine consists of a single virtual processor or a single or multiple dimensional array of virtual processors, and defines the parallelism of the module. It is encapsulated in a construct called a composite procedure. The virtual machine is used as a framework onto which data, communications, and code are placed. All virtual processors within a composite procedure contain the same code, but generally, different portions of the distributed data structures. Conceptually, each virtual processor in a composite procedure executes in parallel when the composite procedure is invoked. Composite procedures are similar to parallel loops found in other languages but have important differences that will be described below. However, there is no inherent reason why parallel loops could not be adapted to have the same semantics as composite procedures.

Figure 1 is an example program that contains a composite procedure that increments every element in a matrix by a parameterized amount. Execution starts in procedure `main` that contains one virtual processor. The call to `brighten` creates N^2 virtual processors, each of which executes the body of the procedure. The value of `N` is passed into the module as a parameter. The actual parameter `A` is distributed across the new virtual machine using the mapping function `element`; this results in each virtual processor containing one element of the matrix. (Mapping functions in Dino2 are similar to those originally defined in DINO [21], and other languages, such as Fortran D, and are not described in detail in this article. The basic capability is the mapping of any axis of a distributed data structure to any axis of a virtual parallel machine, using block, cyclic, or overlapped mappings.) Within each virtual processor of `brighten`, the constants `idx` and `idy` denote the indices of that virtual processor in the structure of processors. These indices are used in the expression `image[idx] [idy]` to specify the local element of `image`. Each element of `image` is augmented by the value of the variable `intensity`. At the end of the call to `brighten` the N^2 virtual processors are terminated and execution continues on the virtual processor running `main`.

Implementing data parallelism using composite procedures has similarities and differences compared with using do loops. It is similar in that both describe the full parallelism of the algorithm and are independent of the machine. This naturally allows the user to define one task per data element instead of one per processor and is an important abstraction for developing modular, machine independent code.

```
#define N 1024
map element () = [block][block];

synch composite brighten (image, intensity, n)    [n:idx][n:idy]

  float remote image[n][n]  map element();     /*distributed array*/
  int remote intensity;     /*mapped to all virtual processors*/
  int n;     /*size of virtual machine and array*/
  {
  image[idx][idy] += intensity;
  }

main(){
  float A[N][N];

  read(A);      /*stub procedure*/
  brighten (A, 1, N);
  display(A);      /*stub procedure*/
  }
```

FIGURE 1 A simple composite procedure.

The differences are equally important in developing numerical programs. The first of these, locality, is an extremely important issue in developing computationally intensive programs. This language supports locality by allowing the user to specify how each composite procedure is mapped to the target machine. This is done using a mechanism similar to how data is mapped to a virtual machine. The mapping functions supported include all of those used to map data to virtual machines (block, cyclic, etc.) and also one that dynamically allocates virtual processors to real processors for imbalanced tasks. This ability to specify where tasks are executed is important for the user to control load balancing and minimize communication. The location of task execution in many other languages follows the "owner computes" rule [5, 12] and is less flexible than explicitly controlling the placement. Another option used is the `on clause` [4].

Another reason for using composite procedures instead of do loops is that we find that the composite procedure better encompasses all of the parts related to parallel computation that must be specified. This includes a set of tasks to be executed and how these are mapped to the target machine, the synchronization model the tasks will follow, the data to be operated on and how it is mapped to the tasks, and how the tasks can communicate with each other. Although this could easily be done using do loops with the appropriate syntactical changes, we find it easier to place this in a construct like a procedure where it is possible to use the scoping and parameter mechanisms to change from one model to another.

The independence between modules supported by composite procedures is important for isolating synchronization models. Each module executes within either a SIMD or SPMD synchronization model and the semantics of the synchronization model are independent of the procedure that called the module. The module will also not affect the synchronization model of any modules that it might invoke. This independence supports flexibility and structure when combining modules having widely varying types of communication and synchronization techniques. An example where this is used is in a nonlinear optimization algorithm where the outer algorithm is SIMD but includes a finite difference gradient evaluation where each virtual processor performs a nonlinear function evaluation independently and asynchronously. Synchronization models are described fully in Section 4, whereas methods to combine modules are described in the next section.

Modular independence also supports writing more machine dependent algorithms that must be efficient. For example, the programmer can choose to specify that there is one virtual processor per physical processor if this makes it easier to describe the parallel algorithm. This might be desirable in cases such as some block parallel computations, where to specify the parallel algorithm correctly one may need to express it in terms of the actual parallelism of the machine. Specifying one task per physical processor will also be important where extreme efficiency requirements prohibit the overhead associated with contracting many virtual processors onto a single processor. Although the contraction can be done quite well, the compiler will never be able to do it as well as the programmer. This is discussed more fully in Section 6.

3 COMBINING MODULES TO FORM COMPLEX PARALLEL PROGRAMS

As mentioned above, there are several types of modules in Dino2. These include SIMD composite procedures, SPMD composite procedures, and normal procedures. From these basic modules, a more complex parallel program can be created through various combinations of calls to composite procedures and/or normal procedures. Conceptually, this creates a more complex parallel virtual machine whose size, shape, and synchronization characteristics describe the parallel nature of the program. The methods for combining modules are described in this section.

In the simplest case, a normal procedure can call a SIMD or SPMD composite procedure. This is the basic mechanism for generating parallelism, and results in changing the virtual machine from a single virtual processor into a set of virtual processors, one for each element of the composite procedure. An example of this was previously shown in Figure 1.

A similar transformation occurs when one composite procedure calls another. That is, suppose each element of a composite procedure with n virtual processors calls another composite procedure with m virtual processors. This is called nested parallelism, and results in a parallel virtual machine with nm virtual processors. Nested parallelism may be used to refine parallel operations on complex data structures. A simple example of this is solving a block diagonal system of equations. At the highest level there is a virtual machine consisting of a virtual processor for each block. At a finer level there may be a virtual processor for each row of each block. As in all the combinations, it is permissible for the two composite procedures to have the same or different synchronization models.

Another combination is called phased parallelism. This occurs when an entire virtual machine of n elements is replaced by a virtual machine with either a different number of elements, or a different synchronization model, or both (and then back again). This is analogous, in a sequential language, to having one procedure call another. In a sequential language, when one procedure calls another, the calling procedure is temporarily halted and saved on a stack while the new procedure is executed. In phased parallelism, the entire composite procedure is temporarily halted while the new procedure is called. This supports structured programming techniques for parallel constructs, much like procedures support structure in sequential programs.

An example of phased parallelism occurs in solving block bordered systems of equations, where the natural degree of parallelism changes between the phase of the algorithm that operates on the diagonal blocks and the phase that operates on the bottom block. Another example is in solving a system of linear equations by using a parallel LU decomposition followed by a pipelined backsolve; here the virtual machine changes from an SIMD model for the LU phase to an SPMD model for the pipelined backsolve, and the number of virtual processors may change from the number of equations to the number of actual processors. Finally, a temporary change in modules may be required to replace an abstract model with a highly efficient, machine dependent model.

Phased parallelism, like nested parallelism, is implemented in Dino2 by having one composite procedure call another, but with the second composite procedure call placed within a "barrier statement." A barrier statement consists of the keyword `barrier` and a C compound statement. When executed within the context of a composite procedure, a barrier synchronizes all the virtual processors associated with the composite procedure and temporarily replaces them by a single virtual processor that executes the compound statement. An example of a parallel language that uses a barrier statement in this manner is the Force [6]. If the statement within the barrier is a call to a composite procedure, then the net effect is a change in parallelism from that of the original composite procedure to that of the called composite procedure, and then back again after the barrier is exited.

A final combination for generating complex virtual machines consists of taking two virtual machines and combining them into a single virtual machine. This is implemented with the '::' statement and is similar to a "cobegin." This construct allows for a functional type of parallelism, as opposed to data parallelism. Generally, when this is used in numerical computation it is at a high level within a program. For example, functional parallelism allows a program to operate concurrently on two different data structures. This construct also can be used to create more irregular parallel programs, such as programs that use a master/slave model to service a set of independent tasks.

An important aspect in developing efficient programs using these constructs is to minimize communication when changing the parallelism of the virtual machine. The virtual machine constructs imply that data will be remapped from one virtual machine to another when a program moves between modules. However, when implementing the different forms of parallelism, the compiler will only move data if the mapping to the physical processors changes. That is, although the data will be moved based on a change in the virtual machine, due to the mapping of the virtual machine to the physical machine, there often will be no change in the mapping to the physical machine. Detecting that the physical mapping does not change is fairly straightforward at composite procedure boundaries but may be more difficult for statements within a barrier statement, depending on the mapping of the data used. In the following example, although there are a number of changes in the virtual machine, the data never needs to be physically moved.

An example of a numerical algorithm that involves both nested and phased parallelism is shown in Figures 2 and 3. This is a procedure used to solve a block bordered system of linear equations. Such systems, which are common in numerical computation, involve a block diagonal matrix augmented by a relatively small, possibly dense final set of rows and columns. In our example, we assume there are Q diagonal blocks, each N×N, followed by bottom and right borders of M possibly dense rows and columns. Thus, the main data structures consist of the Q N×N diagonal blocks, contained in A, the lower right diagonal block, which is of size M×M and is represented by P, and the remainders of the row and column borders, which are of size N×M and M×N, respectively, and are represented by B and C. The diagonal blocks in A are distributed so that one block is on each of the Q virtual processors, and the borders B and C are distributed correspondingly. (This is done using the user-defined "Slice" mapping, which partitions along the first index.) The final diagonal block P is distributed by columns using cyclic mapping.

The first part of the computation consists of factoring each of the Q diagonal blocks, contained in A, and making some corresponding calculations involving the border B and the right hand side f. This implies parallelism of degree Q, the number of diagonal blocks, and of virtual processors in `block_solve`. Within this procedure, however, two of the main steps involve each of the Q virtual processors themselves calling composite procedures `lu` and `solve`, that have parallelism N, to perform computations on the individual N×N matrices. This is an example of nested parallelism, and increases the parallelism to QN during these parts of the computation. After the completion of this portion of the algorithm, the results are used to modify P (using the border blocks), P is factored, and the final M components of the solution,

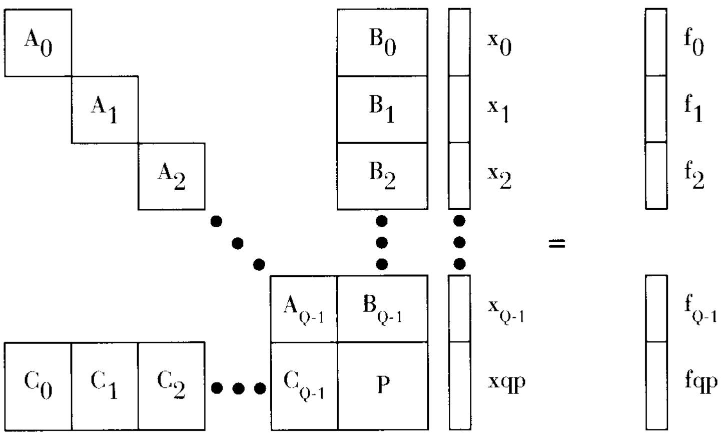

FIGURE 2 Block bordered linear equations.

```
composite block_solve
  (in A, in B, in C, in P, in f, in fqp, out x, out xqp) [Q:id]  map Block()
  double private A[Q][N][N] map Slice();/*Slice distributes data structures */
  double private B[Q][N][M] map Slice();/*  along their first index  */
  double private C[Q][M][N] map Slice();
  double private P[M][M]    map WrapCol();
  double private f[Q*N]     map Block();
  double private fqp[M]     map Block();
  double private x[Q*N]     map Block();
  double private xqp[M];
  {
  double private p[M] map Wrap();
  double p1[N];
  double private b[M] map Block();
  double private sumCW[M][M] map WrapCol();
  double private sumCZ[M] map Block();
  double private W[Q][N][M] map Slice();  /* = A^-1 B*/
  double private z[Q*N]  map Block();
  int private i;
  double private tempB[N];

  lu(A[id], p1,N); /* nested parallelism QN, Q from block_solve, N from lu */
  solve(A[id], &z[id*N], &f[id*N], p1,N);   /* z = A^-1 f */
                            /* nested parallelism QN */

  for (i=0; i<M; i++) {  /*parallelism is Q*/  /* W = A^-1 B */
    tempB[] = B[id][][i];
    solve(A[id], tempW, tempB, p1, N);
    W[id][][i] = tempW[];
    }
  barrier {  /* phased parallelism */
    form_sums( C, sumCW, sumCz);  /*parallelism is Q*/
    sub_vec(P[id], P[id], sumCW[id], M); /* P = P - sum CW*/
    sub_vec(b, fqp, sumCz,M); /* b = fqp - sum Cz*/
    lu( P, p, M);  /*factor P*/
    solve( P, xqp, b, p, M);        /* xqp = P^-1 b*/ /*Parallelism M*/
    }

  /* x = z - w*xqp */ /*parallelism is Q, parallelism of block_solve*/
  for( r=0; r<=N-1; r++ ){
    x[id*N+r] = 0;
    for( c=0; c<=M-1; c++ )
      x[id*N+r] += W[id][r][c]*xqp[c];
    }
  neg_vec( &x[id*N], N );   /*nested parallelism QN*/
  add_vec( &x[id*N], &z[id*N], N );   /*nested parallelism QN*/
  }
```

FIGURE 3 Block bordered linear equations—nested and phased parallelism.

xqp, are computed. These steps all have parallelism M, and therefore the Q virtual processors used in the remainder of the computation are temporarily transformed into M virtual processors. This is done using the barrier statement, and is an example of phased parallelism. (Note that lu and solve are called with a different number of virtual processors, M, in the second call than in the first.) After the barrier statement, the algorithm returns to a third phase that calculates the remainder of the solution, x, and reverts to parallelism of degree Q. It again involves calls that use nested parallelism to expand the parallelism to degree QN. Although this example may seem complex, it is not artificial [22].

4 SYNCHRONIZATION MODELS

The other important high level aspect of describing a Dino2 module, after specifying the virtual machine upon which it is based, is to describe how the virtual processors synchronize and communicate with each other. There are four synchronization models that are supported in the language. These include the SIMD and SPMD models that were mentioned briefly earlier, a chaotic version of the SPMD model in which there is communication between virtual processors but no synchronization, and an independent model in which there is no communication or synchronization between virtual processors. In this section we describe these four synchronization models. The syntax of how to build these synchronization models is described in the next section.

In a purely SIMD model virtual processors would synchronize at every operation. Two major advantages of this model are that the user does not need to explicitly specify synchronization, and that communication can easily be made implicit. Therefore, it is probably the easiest method of programming multiprocessor computers. It is not necessary to use send or receive primitives to transmit data because global synchronization is specified at every operator, and therefore the distributed data structures, which are used for communication, can be viewed as shared memory, and the communication can be deduced by the compiler and run time system.

The SIMD model that is used in Dino2 differs from this pure SIMD model in two important ways and is similar to how other languages implement SIMD on a MIMD machine [18, 23]. Both are the result of the fact that the language is designed for, and executed on, an MIMD machine. First, we do not require that the processors actually synchronize after each operation or even after each communication point, only that communications are inserted that cause the execution of the module to be consistent with what it would be using a pure SIMD model. Second, as was mentioned in Section 3, a call to a standard C function or another module within an SIMD module does not force the SIMD semantics onto the execution of the called function or module. Instead, the called function or module operates under its own synchronization model, which can be either SIMD, SPMD, or totally independent. This flexibility applies to all calls of Dino2 modules, and is a critical aspect of the language.

The second, more loosely synchronous model that Dino2 supports is the SPMD model. In this model, the only specified global synchronization is at the start and end of the module. It is possible to synchronize at points in between, but in these cases the synchronization is produce-consume synchronization that is added by using communication constructs based on distributed data structures, as described in the next section. (As with all composite procedures, communication also occurs at the start and end of the module by distributing or collecting the distributed data structures that are used as input or output parameters to the module, respectively.) The SPMD model is particularly useful for expressing irregular or coarse grained parallel algorithms. In practice it is often most naturally used with a virtual machine whose degree of parallelism corresponds to the actual machine, but sometimes with a virtual parallel machine whose degree of parallelism corresponds to a main data structure (see Fig. 5).

The third model supported is an SPMD model where there is no synchronization associated with communication within the body of a module. We call this model "chaotic SPMD." It differs from the normal SPMD model in what synchronization is implied when two virtual processors communicate data between them. In the normal SPMD model, a produce-consume synchronization is implied: The receiving process blocks until a value arrives. In this manner, messages are consumed in a deterministic order. In the chaotic SPMD model, the receiving process uses the most recently received value if there is a new one, and the current value otherwise, and does not block. This model is similar to shared memory without any synchronization mechanisms, and allows for nondeterminism. It has been used in a variety of chaotic

```
double eigenvalue(left, right, A) /* normal C function called by SIMD
procedure */
                                   /* results in independent parallel model */
  double left, right, A[N][N];
  {
  . . .
    }

synch composite solve_nlinear(A, values)[N:id]  /* SIMD Procedures */
  double remote A[N][N] ;
  double remote value[N] map Block();
  {
  double remote left[N], right[N];
  /*compute interval*/
  . . .
  /*compute eigenvalues*/
  value[id] = eigenvalue(left[id], right[id], A);
  }
```

FIGURE 4 An SIMD program calling a normal C procedure.

iterative numerical algorithms, and can also be useful in non-numerical simulation. The SPMD model in Modula2* is a chaotic model [18].

The final model is an independent model where there is no communication or synchronization between processors during their parallel execution. Often this form of execution is appropriate at low levels of parallel algorithms (as illustrated in Fig. 4).

To illustrate how some of these models are used in Dino2, Figure 4 shows a SIMD procedure calling a normal procedure. This is a shell of a program that computes the eigenvalues of a matrix A. Much of this algorithm consists of algebra to find the intervals containing each eigenvalue. This is best modeled with the SIMD model. From this point, an independent computation is used to find each eigenvalue. These computations proceed independently. This is accomplished using the independent model by calling a normal C function from each process in the SIMD procedure.

5 LANGUAGE SUPPORT FOR SYNCHRONIZATION AND COMMUNICATION

The synchronization models described above are composed by using one of two different types of composite procedures (SIMD or SPMD), and constructs for generating communication between virtual processors. The composite procedure types define the global synchronization characteristics of a module. A SPMD procedure is declared with the keyword `composite` and an SIMD procedure is declared with the keywords `synch composite`.

Communication between virtual processors is implemented by reading and writing variables in distributed data structures that have been mapped to the virtual parallel machine. An important feature of Dino2 is that the semantics of communication is entirely embedded in the data type of the distributed data structures being read from or written to. This means that each access to a given distributed data structure has the same communication semantics. One alternative is to apply special functions or operators to data structures to generate communications (such as the DINO # operator). This means that an access to a given element may or may not specify communication, depending on whether the operator is used. Our experience has been this is confusing and error prone. A second alternative is to use libraries containing send and receive functions to specify communications. These also tend to be difficult to use but more importantly, by making communication part of the language, it is possible for the compiler to take advantage of certain hardware characteristics while implementing the remote reads and writes. Potentially, as will be described in Section 6, this could lead to code that would run faster than if the programmer used libraries of standard send and receive procedures.

The requirement for flexibility and the requirement that there not be any special operators or functions associated with communications sug-

Table 1. Synchronization Models in Dino2

	Composite Procedure Type	
Communication Type	SIMD	SPMD
Private only	Independent	Independent
Remote + private	SIMD	Chaotic SPMD
Buffered + private	(illegal)	SPMD

gest that there be different types of distributed data that have different semantics with respect to communication. In response to this we have developed what we call a communication type that is associated with each data structure. A communication type describes the communication semantics of a variable. This is similar to data types that are found in all languages and are associated with every variable. The usual data type describes the semantics of operations on a variable. For example, the divide operator has different semantics depending on whether the operands are integers or floats. Similarly, the communication type describes the semantics of the communications associated with reads and writes of a variable.

A variable in Dino2 may have one of three communication types: private, remote, or buffered remote (specified by the keywords `private`, `remote`, and `buffered`). Table 1 summarizes how these combine with the two types of composite procedures to form the synchronization models discussed in Section 4. This is explained in the next three paragraphs.

Private variables can be used in any procedure (SIMD, SPMD, or normal), and can only be read or written by the virtual process that contains the variable. That is, no communication is associated with these variables. When only private variables are used in a composite procedure, or in a C function called from a composite procedure, the independent model results.

The role of remote and buffered remote variables varies with the type of procedure in which the variable is used, although their communication semantics are unchanged. Within an SIMD composite procedure, only remote (i.e., not buffered remote) variables can be used. Remote variables can be accessed by any virtual process in the composite procedure where they are declared and the accesses are nonbuffered, meaning that the value most recently assigned to a variable by any virtual process is used when the variable is read. In conjunction with the implicit global synchronization semantics specified by the SIMD composite procedure type, this defines the SIMD synchronization model described in the previous section.

Within SPMD composite procedures, either remote or buffered remote variables can be used. Remote variables have the same semantics in SPMD as in SIMD composite procedures. When used in the context of an SPMD composite procedure, however, these variables lead to the chaotic SPMD synchronization model discussed in Section 4 because, in contrast to the SIMD model, there is no global synchronization between virtual processors. Buffered remote variables are similar to remote variables except that they have a buffered implementation. That is, all writes to the variable are buffered by each virtual process that may require them in the order in which they arrive, and reads to the variable block until a value is present in the buffer, at which point that value is used and removed from the buffer. Using buffered remote variables in conjunction with SPMD composite procedures leads to the SPMD synchronization model described in Section 4.

Figure 5 is an example of an SPMD procedure. This procedure does a pipelined solve of a linear system of equations involving a banded, lower triangular matrix of the type that arises in some differential equation algorithms on two-dimensional N×N grids. The matrix consists of the main diagonal of N^2 ones (which are not stored); a diagonal immediately below the main diagonal of N^2-1 elements, stored in `al`, of which each Nthe element is zero due to the border affects of the grid; and a diagonal N rows below the main diagonal, with N^2-N elements, stored in `an`. (The vectors `al` and `an` are padded with 1 and N leading zeroes, respectively, so that their element with index `i` corresponds to row `i` of the matrix.) Due to the zeroes in `al`, a pipelined type of parallelism can be used to perform the solve. At step 1, y_0 is computed. At step 2, y_1 and y_N are computed. At step 3, y_2, y_{N+1}, and y_{2N} are computed. At step 4, y_3, y_{N+2}, y_{2N+1}, and y_{3N} are computed, and so on. These dependencies suggest that N virtual processors should be used, and the two vectors `al` and `an` should be mapped cyclically onto the virtual machine. Each y_i is computed by $rhs_i - y_{i-1} * al_{i-1} - y_{i-N} * an_{i-N}$. The first and third terms are computed locally because the processor that contains y_i also contains rhs_i, y_{i-N}, and an_{i-N}. The product $y_{i-1} * al_{i-1}$ is received from the neighboring processor by reading `pipe[id]`. This read will block until the value has arrived because `pipe` has a buffered communication type. This product is sent to the neighboring processor in the write to

```
composite pipe_solve(y,rhs,al,an) [N:id]
  private float y[N*N] map wrap();
  private float rhs[N*N] map wrap();
  private float al[N*N] map wrap(); /*the off diagonal*/
  private float an[N*N] map wrap(); /*the far off, by N, diagonal*/
  {
  int i;
  buffered float pipe[N] map block(); /*used to implement pipe*/

  for(i=id; i<id+N*(N-1); i+=N)
    y[i] = rhs[i];
  for(i=id; i<id+N*(N-1); i+=N){
    if(id>1)                    /*if not the first stage in pipe*/
      y[i] -= pipe[id];          /*wait for y[i-1]*al[i-1]*/
    /*y[i] is now evaluated*/
    if(id<N-1)                  /*if not the last stage in pipe*/
      pipe[id+1] = y[i]*al[i];   /*compute value and send to next stage*/
    if(i<N*(N-1))               /*if not in last wave*/
      y[i+N] -= y[i]*an[i];       /*compute last term for y[i+N]*/
    }
}
```

FIGURE 5 A pipelined solve—an example of an SPMD procedure.

pipe[id+1]. Each variable pipe[i] is used repeatedly to send messages from processor i−1 to processor i, but the semantics of buffered remote variables assure that the correct algorithm semantics are enforced. This algorithm is subtle to understand and perhaps to program, but parallel algorithms like this are important for efficient parallel numerical computation, and appear to require an SPMD model to express accurately and efficiently.

In keeping with the goal of supporting modularity for large parallel programs, the communication type of a variable in Dino2 may be changed in a structured fashion between modules. A data structure having one communication type may be passed as a parameter to a procedure where the corresponding formal parameter has a different communication type. As an example, assume that there is a distributed array of remote floats declared within the body of a composite procedure, and that it is desirable to temporarily turn off any communications associated with the data structure. This can be done by passing the array to a procedure where the formal parameter is a distributed array of private floats. Note that, as arrays are passed by reference in C, there is no communication generated from this. Within the body of the new procedure there will be no communications generated from reads or writes of the data structure. This ability provides the user with the flexibility to control the communication semantics of a variable, but in a manner that is structured through the use of scoping and procedure semantics.

The concept of communication types for variables in parallel languages appears to be a new contribution of this work. Communication types give the user a great deal of flexibility in selecting the type of communication semantics to use, and also adds structure to the communications in a program.

6 IMPLEMENTATION

In this section we discuss some of the more interesting implementation details of compiling Dino2 programs. Although a Dino2 compiler has not been built, the more critical components have been built in compilers for other languages. Based on our previous experience with writing the DINO compiler, the areas of compilation that will effect the efficiency of the resulting programs the most include contracting virtual processors into processes, communication, and the mapping of virtual processors to the target machine.

The contraction of virtual processors to one process per physical processor is probably the most crucial aspect of the compilation of Dino2, and must be done for each composite procedure. The reason that this step is crucial is that the contraction of composite procedures needs to minimize communications and the overhead of simulating parallel tasks. On current parallel machines, accomplishing this is a very important aspect of developing efficient code.

In general, it is expected that there will be more virtual processors than actual processors. Furthermore, the number of virtual processors and actual processors will not be known until runtime. To accommodate these variations, each composite procedure on entry will have to compute a set of integer offsets that are used to describe the virtual processors and data located on each physical processor. These values are then used within the body of the procedure.

The bulk of the compilation strategy used to compile composite procedures depends on the synchronization model and the type of communications used within the procedure. The compilation of SIMD composite procedures would probably be the hardest case because of the need to remove unnecessary synchronization points and to vectorize communication and computation where possible. This work has previously been done, however, in the C* compiler on the nCUBE [23, 24]. The C* compiler first identifies points of communication and then, based on this information, transforms the control constructs of the program so the code executes on an MIMD machine using sends and receives. From this transformation, several optimizations are performed to improve the efficiency of the communications. These include moving sends as far forward as possible in the program and moving receives as far back as possible. Preliminary results of the C* compiler for the nCUBE are fairly good. For example, in a parallel Gaussian elimination program, the translated code ran 30% slower than the hand coded version.

Further results of this compilation have been discussed [24]. This article describes measured speedups versus the number of processors on, among other machines, an Intel IPSC/2. All times are compared with the best sequential time on a single node of the processor. Although this does not compare the speed of hand coded programs to that of what the C* compiler can do, it does illustrate how well SIMD programs can be executed, excluding communication. The results range from a low of roughly 50% efficiency for numerical integration, 80% for the Gauss-Jordan algorithm, and a high of 98% for computing primes. Typical results were in the 79–80% range. These results are good and indicate that the SIMD model is viable for many programs. Where the user needs better efficiency, a more explicit model giving the user more control over the machine, such as SPMD, can be used.

The contraction of an SPMD composite procedure is dependent on whether it contains communications or not. If there is no communication then the contraction is simply a matter of adding a for loop around the body of the procedure. Detecting communication is simply a matter of looking for what type of variables are read or written. If the variables are all of type private then there will be no communication. Alternatively, if remote or buffered remote variables are declared and used to generate remote communications then the compiler must generate code that preserves the order implied by the communication.

SPMD procedures that use buffered remote variables could be implemented using lightweight processes or threads [25]. However, it should be possible for the compiler to generate code that simulates threads. Such code would not be interrupt driven and would probably be more efficient than a general purpose threads package. This is possible because the compiler can generate code specifically for a given composite procedure whereas a run time system must handle the most general case.

The general methodology used in the translation of an SPMD procedure with communications is as follows. The translated program is broken into a sequence of statements of re-entrant blocks of code. The blocks are delimited by reads to buffered variables as these are the only times that a process can block. Variables that are declared to be buffered remote in the Dino program will consist of a list of values that have been buffered but have not been read. In the case where virtual processes have blocked waiting for a value, this type of variable will point to a list of virtual processors that have blocked reading the variable. A C switch statement is used to simulate each block of code. Each `case` statement represents one block of code. At the end of each block, a virtual process will either block on a remote read and stop executing, or it will continue to the next block. A while loop around the switch statement cycles until all of the blocks for all of the virtual processors have completed executing. To generate blocks that are re-entrant the compiler must generate variables for any values that will be used in more than one

```
composite proc() [N: id]
{
float buff x[N] map Element();
float q;

S1;
x[f()] = id;
S2;
q = x[g()];
S3;
}
```

FIGURE 6 An SPMD composite procedure with buffered communications.

block and would normally be stored on the processor stack. Thus, a context switch is little more than changing an index that describes which virtual processor is being executed. The simulation of SPMD virtual processors in this manner should be more efficient than using a general purpose threads package.

To illustrate this approach, Figure 6 is an SPMD composite procedure that contains three statements separated by a remote write and a remote read. A possible translation of this example that could be performed by a compiler is given in Figure 7. In the translated program some of the declarations and expressions have been replaced by comments for clarity. In this example, the N virtual processors are contracted onto a single physical processor. The more general situation, where the virtual machine is contracted onto two or more real processors, is not shown because the example would be more complicated without adding much to the basic ideas. The main difference would be that communications between virtual processors are slightly different.

In the translated program the buffered remote variable `x` is translated into a struct of type `x_buff` that is a list of values or blocked virtual processes. The state of each virtual processor is contained in the array `proc_vp`. This includes the block of code that is to be executed next, `state`, and the value of the remote read. One additional data structure, `ready_que`, that is required is the queue of ready virtual processes. This is similar to the blocked queue associated with each buffered variable.

The code in Figure 6 is broken into two blocks when translated. These include everything up to and including the read to `x[g()]`, and everything after that. In this case the value of the remote read is the only value that needs to be explicitly stored. This value is stored in `x_tval`. In this example, each virtual process starts off executing the first block. In the first block, a remote writer buffers the value in the buffer associated with `x[f()]`. This is done in `write_x` and is not shown here. After the write, a virtual processor executes `S2` and then attempts to read `x[g()]`. If no value exists then the virtual processor is blocked on that value. This is done in `read_x`. As soon as a write is made to the variable that this process is blocking on, that value is assigned to the temporary variable `x_tval` associated with the blocked process and the process is placed back in the ready queue. After successfully reading the value, it is assigned to the local value of `q` and `S3` is then executed before the virtual processor is removed from the ready queue. Execution ends when the ready queue is empty.

Another case in which SPMD procedures are contracted is the chaotic SPMD model, or where SPMD procedures use remote variables as opposed to buffered remote variables. The compiler should make some attempt to keep this "fair" when executing the composite procedure on a single real processor. The compiler could execute all of the code for each virtual processor before executing the code of any other processor but this would probably not have the intended effect. A better method would consist of breaking the code into blocks, as if the remote variables were buffered remote, and always changing to another virtual process at each remote read.

The implementation of constructs that generate communication is another critical element in the compilation of Dino2 programs. The two types of constructs that create communication are the remapping of data between modules, and implementing remote reads and writes of distributed data. The remapping of data between modules is implemented similarly to remote writes, and could be handled by the compiler in a similar manner. Therefore we do not discuss it separately, because remote writes are discussed below. It would probably be advantageous to also have special libraries to handle common remappings very efficiently.

The two types of variables that can create communications are remote and buffered remote variables. Because the communication semantics are an explicit part of the language, it is fairly easy for the compiler to determine where in the program communication will be generated. In the SPMD model, it is always assumed that reading or writing a remote or buffered remote variable will always

```
struct x_buff{
/*contains list of buffered values*/
/*or a list of blocked vps*/
};

struct proc_state{
int state;
struct proc_next *next; /*used to keep vps in ques*/
float x_tval[id]; /*values associated with remote read for each vp*/
}

proc(){
struct x_buff x[N];
float q[N];
struct proc_state proc_vp[N];
struct proc_state *ready_que;

initialize_proc_state();
/*initialize the status of each virtual process
and place in the ready queue*/

while ( /*ready queue not empty*/){
id = /*id of first element in ready queue*/
switch(proc_vp[id].state){
case 1:
S1;
write_x(f(),id);
/*put id into the buffer for x[f()] or
  put a blocked vp in ready queue*/
S2;
proc_vp[id].x_tval = read_x(g(), id);
/*if a value is buffered for x[g()] take it and continue
  else block this vp on x[g()]*/
case 2:
q[id] = proc_vp[id].x_tval;
S3;
remove_vp(id); /*remove this vp from the ready queue*/
}
```

FIGURE 7 Translation of an SPMD procedure with buffered communications.

generate communication. In the SIMD model, optimizations are used to try to avoid putting in calls for communication where it is not needed. These optimizations are well understood [24, 26].

The primary concern in implementing these variables is the minimization of message startup times, or latency. Message latency has both a hardware and software component. The hardware component is decreasing with newer machines. As an example, the iPSC/860 has a hardware latency of roughly 25 μs and the Intel Paragon will have a roughly 1 μs hardware latency [27]. The software latency, when based on messages, is roughly in the range of 30–100 μs. This large time is dependent on the very general nature of the library underlying the message interface. In the general case, these libraries must handle messages of any length arriving at any time. There is a large opportunity to greatly reduce the software latency by taking advantage of knowledge about how communication is used in a program. For communication intensive programs, it is quite reasonable for a compiler to be able to generate communication that runs faster than if the code were hand coded using messages. Furthermore, as the latency is reduced, the need to aggregate messages will become less important.

One technique for generating efficient communications would be to use a system similar to Active Messages [28]. An active message is essentially an asynchronous remote procedure call (RPC) and consists of a procedure id and parameters to the procedure. The RPC mechanism is much more efficient than a message based system because the system does not handle any low level details of handling messages such as buffering or type checking. Instead, all the system does is invoke the correct procedure and let the procedure handle what needs to be done. In this manner, only the services that are required are implemented and used. On the CM5, the use of active messages has reduced the message latency from roughly 70 μs to less than 5 μs.

In this context, active messages can be used to generate remote reads and writes in a fairly obvious manner. There are RPCs corresponding to both remote reads and writes. Buffered variables can be implemented with a buffer associated with each variable. This will provide faster access than having one large buffer consisting of tagged messages that must be interpreted for each access.

The final aspect of compilation that is of interest is the placement of virtual processors onto the target machine. This mapping is partially specified by the programmer and can be either static or dynamic. In the static case it is assumed that each virtual processor in a composite procedure takes the same amount of time to execute. If this is not an appropriate model than the programmer should use a dynamic mapping or must use an explicit technique to do the mapping. The mapping of static composite procedures is based on a simple set of rules and a static analysis of the program. Each composite procedure call divides the number of virtual processors by the number of available processors and then maps the composite procedure based on the mapping function specified. This is done so that adjacent virtual processors in the virtual processor data structure are on the same physical processor as often as possible, and generally are on adjacent physical processors otherwise. Techniques for accomplishing this for common structures are well known and are similar to blocking and distributing data arrays onto arrays of virtual processors. It is important that the mapping does not change throughout the execution of the procedure. By doing this, it is guaranteed that the distributed data will not be moved and communication can always be sent directly to the correct processor. Furthermore, in the cases where there does not need to be any communication for remapping of parameters (the actual and formal parameters have the same type of mapping) there does not need to be any communication generated. In this case a composite procedure call can be implemented with a simple procedure call.

The second type of mapping, a dynamic one, could be supported using either a centralized or distributed task allocation scheme such as that found in Hsu and Liu [29] or Lin and Keller [30]. This ability to dynamically schedule tasks is similar to self-scheduled loops in Jordan [6]. A difference is that communication between processors is only allowed, in the form of parameters, at the start and end of the composite procedure call. This is done because of the difficulties in implementing communications in an environment where the placement of virtual processors is not known until they are executed. It would be difficult to implement communications because the location of a variable would be hard to find without going back through the mechanism that distributed the tasks.

7 CONCLUSION

The Dino2 language provides several new features for writing large, modular parallel programs. These include: (1) the provision of two synchronization models, SIMD and SPMD, that can be used in conjunction with parallel computation modules; (2) the ability to combine SIMD modules, SPMD modules, and normal C procedures using nested and phased parallelism to obtain complex parallel programs; and (3) the provision of communication types for distributed variables that define the communication semantics associated with reads and writes to these variables. These features provide the user with a flexible and expressive parallel programming language that still should be easy to use and result in efficient code. By modularizing the degree of parallelism, the synchronization model, and the communications, programs can be written using a range of techniques that are not possible to combine in other languages without introducing unmanageable complexity into some portion of the code. This flexibility to combine different parallel algorithm paradigms will be needed to write parallel programs for many large, complex scientific computations. The modularity should also help in writing large programs because parallel modules can be written independently. Finally, the characteristics of the modules have been designed to permit efficient execution. Many implementation considerations associated with the language are discussed in Rosing [20], but a full implementation of the language has not yet been performed.

ACKNOWLEDGMENTS

Both authors supported by NSF grant ASC-9015577, NSF grant CDA-8922510, and AFOSR-90-0109. Rosing was also supported by the National Aeronautics and Space Administration under NASA Contract Nos. NAS1-18605 and NAS1-19480 while in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, and currently by Pacific Northwest Laboratory, operated for the U.S. Dept. of Energy (DOE) by Battelle Memorial Institute under contract DE-AC06-76RLO 1830.

REFERENCES

[1] D. Olander and R. B. Schnabel, *Proceedings of the Scalable High Performance Computing Conference.* Williamsburg, VA: IEEE Computer Society Press, 1992, pp. 276–283.

[2] W. Griswold, G. Harrison, L. Snyder, and D. Notkin, *Proceedings of the Fifth Distributed Memory Computing Conference.* 1990.

[3] M. Rosing, R. B. Schnabel, and R. P. Weaver, "The DINO parallel programming language," *J. Parallel Distrib. Comput.*, Vol. 13, pp. 30–42, 1991.

[4] C. Koelbel, P. Mehrotra, and J. Van Rosendale, *Conference on Principles and Practice of Parallel Processing.* 1990.

[5] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, "FortranD language specification," Technical Report CRPC-TR90079, Department of Computer Science, Rice University, 1990.

[6] H. Jordan, *The Force.* Boston: MIT Press, 1987.

[7] J. R. Rose and G. L. Steele Jr. C, "An extended C language for data parallel programming," Technical Report PL-5, Thinking Machines Corp., 1987.

[8] R. J. Littlefield, "Efficient iteration in data-parallel programs with irregular and dynamically distributed data structures," Technical Report 90-02-06, Department of Computer Science, University of Washington, 1990.

[9] L. Hamey and I-C. Wu J. Webb, *Apply, a Programming Language for Low-Level Vision on Diverse Parallel Architectures.* Kluwer Academic Publishers, 1987.

[10] P.-S. Tseng, "A parallelizing compiler for distributed memory parallel computers," PhD thesis, Carnegie Mellon, May 1989.

[11] B. Chapman, P. Mehrotra, and H. Zima, Programming in Vienna Fortran, *Sci. Prog.*, vol. 1, pp. 31–50, 1992.

[12] C. Koelbel, "Hpff," Technical Report, Rice University, 1993.

[13] J. K. Lee and D. Gannon, "Object oriented parallel programming: Experiments and results," *Supercomputing,* 1991.

[14] M. Rosing, R. B. Schnabel, and R. P. Weaver, "Scientific programming languages for distributed memory multiprocessors: Paradigms and research issues," Technical Report CU-CS-537-91, Univ of Colorado, Department of Computer Science, 1991.

[15] F. Andre, J. Pazat, and H. Thomas, *Proceedings of ACM ICS.* 1990.

[16] M. J. Quinn and P. J. Hatcher, "Data parallel programming on multicomputer," *IEEE Software,* pp. 69–76, 1990.

[17] M. A. Nichols, H. J. Siegel, and H. G. Dietz, *Third Symposium on the Frontiers of Massively Parallel Computation.* 1990, pp. 397–406.

[18] M. Philippsen, W. Tichy, and C. Herter, *Proceedings of the First International Conference of the Austrian Center for Parallel Computation.* 1991.

[19] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, Jr., P. T. Mueller, Jr., H. E. Smalley, and S. D. Smith, "Pasm: A partitionable simd/mimd system for image processing and pattern recognition," *IEEE Transact. Comput.* vol. C-30, pp. 934–947, 1981.

[20] M. Rosing, "Efficient language constructs for complex parallelism on distributed memory multiprocessors," PhD thesis, University of Colorado, Boulder, August 1991.

[21] M. Rosing and R. B. Schnabel, *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing.* Los Angeles: SIAM, 1987, pp. 312–316.

[22] X. Zhang, R. H. Byrd, and R. B. Schnabel, "Parallel methods for solving nonlinear block bordered systems of equations," *SIAM J. Sci. Stat. Comput.*, vol. 13, pp. 841–859.

[23] M. J. Quinn, P. J. Hatcher, and K. C. Jourdenais, *Proceedings of ACM/SIGPLAN PPEALS, Parallel Programming: Experience with Applications, Languages, and Systems.* ACM Press, 1987, pp. 57–65.

[24] P. Hatcher, M. Quinn, A. Lapadula, B. Seevers, R. Anderson, and R. Jones, "Data-parallel pro-

gramming on MIMD computers," *IEEE Transac. Parallel Distrib. Systems*, vol. 3, 1991, pp. 377–383.

[25] D. Eager and J. Zahorjan, "Chores: Enhanced run-time support for shared-memory parallel computing," Technical Report 91-08-05, University of Washington, 1991.

[26] S. Hiranadani, K. Kennedy, and C. Tseng, *Proceedings Supercomputing 1991*. 1991, pp. 86–100.

[27] M. Rosing and J. Saltz, "Low latency messages on distributed memory multiprocessors," Technical Report 92-25, Institute for Computer Applications in Science and Engineering, 1992.

[28] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, *Proceedings of the 19th Annual International Symposium on Computer Architecture*. 1992, pp. 256–266.

[29] C. H. Hsu and J. Liu, *Proceedings of the 6th Internationl conference on Distributed Computing Systems*. 1986.

[30] F. C. H. Lin and R. M. Keller, *Proceedings of the 6th International conference on Distributed Computing Systems*. 1986.