Efficient Language Constructs for Large Parallel Programs -- An Overview of Dino2

Matthew Rosing
and
Robert B. Schnabel

CU-CS-578-92 January 1992

Efficient Language Constructs for Large Parallel Programs -- An Overview of Dino2

Matthew Rosing[1]
and
Robert B. Schnabel[2]

CU-CS-578-92 January 1992

This research was supported by NSF Grant ASC-9015577, NSF Grant CDA-8922510, and AFOSR Grant AFOSR-90-0109.

[1]Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Virginia 23665-5225 USA

[2]Department of Computer Science, Campus Box 430, University of Colorado, Boulder, Colorado 80309-0430 USA

Efficient Language Constructs for Large Parallel Programs -- An Overview of Dino2

Abstract

The goal of the research described in this paper is to develop easy-to-use, efficiently implementable language constructs for writing large data parallel numerical programs for distributed memory (MIMD) multiprocessors. Although distributed memory machines show great promise, they are difficult to program. This arises from the mismatch between the target machine, the language used, and the underlying model of the data parallel algorithm. Previously, several models, including explicitly parallel SIMD (Single Instruction Multiple Data) models, explicitly parallel SPMD (Single Program Multiple Data) models, and sequential programs annotated with data distribution statements and possibly parallel loops, have been developed to express programs with simple data parallelism. In this paper, we give an overview of a new language that combines the two explicitly parallel models, SIMD and SPMD, in a structured and modular manner such that large, efficient parallel programs can be written. Communications in this language are also flexibly and modularly defined utilizing a new attribute of distributed variables, a communications type.

1. Introduction

The goal of the research described in this paper is to develop easy-to-use, efficiently implementable language constructs for writing large data parallel numerical programs for distributed memory (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. Examples of distributed memory machines include the Intel and NCUBE hypercubes, Thinking Machine's CM5, and networks of workstations used as multiprocessors.

Although distributed memory machines show great promise, they are difficult to program. This arises from the low level details the programmer 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. Another way of saying this is that the programming model used must be changed to more closely match the types of algorithms implemented. 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 between the target machine, the languages used, and the underlying model of typical data parallel algorithms. It does this by developing a language model that on the one hand provides the user an intuitive interface to the target machine, and on the other hand still provides the compiler with enough information to generate efficient code. A key aspect of the language model is that it supports the expression of large, modular parallel programs, which may use different models of parallelism in different portions.

The next section motivates this research more thoroughly, and then describes the organization of the rest of the paper. This paper provides an overview of the new language model; more detail is provided in [16] and will be provided in forthcoming papers.

2. Motivation

Recently, several languages, including the DINO language [14], have been proposed for writing numerical programs on distributed memory machines [3-6, 8, 9, 13, 17]. These languages appear to be converging in terms of the underlying model used [15]. This model is primarily a data parallel one with a little support for functional parallelism in some cases. It has three main parts. First, arrays of virtual processors may be declared [1, 7, 8, 14] in a shape that best fits the algorithm. Second, arrays of data may be distributed across these virtual processors [5, 8, 14]. This distributed data is usually treated as a single global object and all accesses are made with respect to the global name space. Third, some model is used for specifying the computation. Here there appear to be two classes of approaches, either a annotated sequential program approach or an explicitly parallel approach. Within each of these classes there appear to be two main options.

Within the annotated sequential program approach, one option is to write entirely sequential execution statements (along with data distribution annotations) and have the compiler and run-time system automatically generate all of the communications based upon data dependency analysis and the data distribution annotations [2, 5, 7, 11, 12]. A second option is to also augment the sequential program with parallel execution annotations, generally parallel loops such as forall statements. These parallel loops generally follow a "loosely synchronous" or "copy-in copy-out" semantics that allows communication only at the beginning and end of the loop [8]. The communications are usually implicitly specified. The compiler may or may not attempt to extract additional parallelism beyond what is specified in the parallel statements.

Within the explicitly parallel approach, one option is to use a general SPMD (Single Program Multiple Data) 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 [14]. A second option is to use an SIMD (Single Instruction Multiple Data) synchronization model in which virtual processors effectively synchronize at all communications [11]. 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 Modula2* [10], combine some aspects of both the SIMD and SPMD models.

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 to express large parallel programs is the restrictiveness of the programming model. Almost every language follows one of the models described above. These are the sequential program with data distribution annotations, sequential program with parallel execution annotations, explicitly parallel SPMD, and explicitly parallel SIMD models. Each of these models has tradeoffs between ease of use, expressiveness, and efficiency.

If the language follows the explicitly parallel SPMD model, with parallelism specified at a per-task level, then there are two problems the user must face when writing large programs. First the user usually must explicitly put in synchronization and communications. Another, more subtle problem, is that each section of code must be written with the knowledge of how many processors will be available to execute the code. The number of processes should equal this number. Otherwise, it is difficult for the compiler to generate efficient code; the compiler may need to multiprogram the processes or use a similar technique. This will not result in as efficient code as if one explicitly wrote one process per processor. This constraint makes writing modularly defined, large programs especially difficult because the context of the module within the entire program must be understood in order that the number of processors available is known. Therefore, different parallel programs, based on different contexts, may have to be written for the same algorithm.

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 enough information 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 inappropriate for expressing many parallel algorithms.

The sequential model using parallel loops with loosely synchronous communication semantics has advantages and disadvantages similar to those of the SIMD model. The difference between these two models is primarily the granularity of the concurrent computation that is done between possible communications; in the SIMD model it is one operation, whereas in the sequential model with parallel loops it is the body of the loop. The larger granularity of the latter model makes it more expressive than the SIMD model in some regards, such as the non-uniformity of parallel tasks, but more restrictive in others, such as the placement of communications.

Finally, the sequential model using only distributed data annotations has the advantage that the programmer does not specify any communications or synchronizations, 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 some pipelined algorithms, that appear to be especially difficult to express in or derive from a sequential program. The latter disadvantage may apply to the sequential model with parallel loop annotations as well.

Thus, limiting a language to one of the models described above makes it difficult for the language to be satisfactory with respect to ease of use, expressiveness, and efficiency for a broad range of large computations. Furthermore, many large numerical programs have modules that fit each of these models. For example, the kernels of numerical programs tend to be highly structured, fine grained computations that fit the explicitly parallel SIMD model or either of the annotated sequential models, while 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 general 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. It must also provide an easy method of switching between these models. In addition, it must allow the programmer to write parallel modules where the actual available parallelism is unknown to the programmer. These needs form the main motivation for this research.

This paper gives an overview of a new language, called Dino2, that addresses these issues. Dino2 is a successor to the DINO language [14], 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 for expressing parallel computations, communications, and synchronizations are very different.

The conflict regarding what type of model of computation to use is resolved in Dino2 by using a modular approach to designing parallel programs. To readily accommodate both coarse and fine grained parallelism in a way that fits well together, the language is based upon an explicitly parallel computation model. Each module in Dino2 consists of a virtual parallel machine that operates using either a SIMD or SPMD synchronization model, and contains data and code. It is possible to mix the models of computation as needed. For example, a SIMD model could be used to do lower level matrix operations while a SPMD model could be used to handle load balancing at the higher levels of a program. Dino2 develops a framework which combines these two models in a structured fashion, and derives many of the advantages of both. In either model, all communications are implicitly derived from reads and writes of data that can be distributed across the virtual machine.

Each module in a Dino2 program is consists of a virtual machine, a synchronization model, and distributed data structures that form the basis of communication between virtual processors. Section 3 describes virtual machines, section 4 describes the SIMD and SPMD synchronization models, section 5 describes how modules can be combined to form complex parallel programs, and section 6 describes how virtual processors communicate with each other. More details of the language and potential implementation issues are provided in [16] and will be available in forthcoming papers.

3. Virtual Machines.

A Dino2 module is built around a parallel virtual machine defined by the user. A virtual machine consists of a single virtual processor or a structured set of virtual processors, and defines the parallelism of the module. Conceptually, each virtual processor executes in parallel. The virtual machine is used as a framework onto which data, communications, and code are placed. All virtual processors within a module contain the same code.

A module consisting of a single virtual processor is described by a normal procedure. In the more interesting case, where a module consists of a structured set of virtual processors, a construct called a *composite procedure* is used to describe the module. A composite procedure is essentially an array of similar procedures that are called and executed concurrently.

Figure 1 is an example program that contains a composite procedure which increments every element in a matrix by a parameterized amount. Execution starts in procedure *main* which contains one virtual processor. The call to *brighten* creates N^2 virtual processors each of which executes the body of the procedure. 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 found in DINO and in FortranD and are not described in detail in this paper. 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*. At the end of the call to *brighten* the N^2 virtual processors are destroyed and execution continues on the virtual processor running *main*. This process is described more fully in Section 5.

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

synch composite brighten(image, intensity)[N:idx][N:idy]
      float remote image[N][N] map element(); /*distributed array*/
      int remote intensity; /*mapped to all virtual processors*/
      {
      image[idx][idy] += intensity;
      }

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

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

A two-dimensional composite procedure used for image processing
Figure 1

An important aspect of Dino2 is that the virtual machine and the target machine are independent of each other. In data parallel programs the parallelism generally matches the size and shape of the major data structures. In Dino2, therefore, the virtual machine is generally related to the size and shape of the data as opposed to the size and shape of the target machine. As an example, in the *brighten* procedure, there are 1024^2 virtual processors declared whereas there are probably many fewer processors available to execute the code. The pattern of contraction of the virtual processors to the target machine can be specified by the programmer as well, although if it is omitted as in the *brighten* example, the default is to use a block mapping along each axis of the virtual parallel machine. Thus if the actual parallel machine had 64 processors, each actual processor would contain a 128×128 block of virtual processors in this example.

One advantage of specifying the full data parallelism of a program is that it is easier to write many numerical applications based on one virtual processor per data element instead of one processor per block of data elements. In the blocked version the user must explicitly specify how inherently parallel tasks are sequentialized at the processor level. In the fully parallel description the user does not need to specify how the parallel code should be transformed into sequential code. The compiler can handle this transformation automatically.

Another advantage to specifying the full parallelism of a program is that it will be easier to write modular programs. By describing a virtual machine for each module of a program, the user does not need to be concerned with how many processors will be available to execute the module. Without this machine independence each module must be written based on the context within which it will be used. In this case more than one module may have to be written for the same parallel algorithm. With a fully data parallel virtual machine, the compiler will generate the correct module. Generally this requires the compiler to contract the virtual machine into a smaller number of processes such that there is one process per processor. Hatcher and Quinn's compiler, used to translate SIMD languages to the NCUBE [11], is an example of how this transformation is done.

It should be noted, however, that the programmer can choose to specify that there is one virtual processor per physical processor. This might be desirable in cases, such as pipelined computations, where to specify the parallel algorithm correctly one may need to express the algorithm in terms of the actual parallelism of the machine. This is generally done in Dino2 in conjunction with the SPMD synchronization model, whereas the SIMD synchronization model is generally used in conjunction with a fully data parallel virtual machine.

4. Synchronization Model

The next important aspect of describing a module, after describing the virtual machine, is to describe how the virtual processors synchronize with each other. In Dino2, this is done via the synchronization and communication models. A synchronization involves all virtual processors and is a useful construct for implicitly coordinating the progress of virtual processors and implicitly generating communications. By communications we mean transferring data between two or more virtual processors.

Dino2 supports two synchronization models, SIMD and SPMD. In a purely SIMD model, virtual processors synchronize at every operation. Two major advantages of this model are that the user does not need to explicitly control 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: the synchronization is already handled at every operator and therefore the distributed data structures, which are used for communication, can be viewed as shared memory. Another advantage of the SIMD model is related to the parallelism of the virtual machine. This model makes it easy for the compiler to take a program written at the maximum level of data parallelism, which is usually the easiest way to write the program, and contract it to an equivalent, efficient program for the available processors.

The way that the SIMD model is used in Dino2 differs from what is typically thought of as pure SIMD in two important ways. Both are the result of the fact that the program will be executed on an MIMD machine. First, we do not assume that the processors actually synchronize after each operation or even after each communication point, only that the communications is consistent with this pure SIMD model. Second, a call to a standard C function or another module within a 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 module which can be either SIMD, SPMD, or totally independent. This flexibility applies to all Dino2 modules and is a critical aspect of the language.

As an example, Figure 2 shows a SIMD procedure *solve_nlinear* calling a normal procedure *eigenvalue*. This is a shell of a program that computes eigenvalues. For the most part, this algorithm consists of linear 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. This is most appropriately done with a normal procedure.

The second, more loosely synchronous model that Dino2 supports is the SPMD model. In this model, tasks only synchronize at the start and end of a module. It is possible to synchronize at points in between but in these cases the synchronization must be added via communication constructs as described in section 6. Communications at the start and end of the SPMD module are supported implicitly by distributing or collecting data structures at the start or end of 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.

```
double eigenvalue(left, right, A)
    double left, right, A[N][N];
    {
    ...
    }
synch composite solve_nlinear(A, values)[N:id]
    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);
    }
```

Calling a normal procedure from a SIMD procedure
Figure 2

5. Combining Modules to Form Complex Parallel Programs

A Dino2 module, as described above, consists of a virtual machine executing with a SIMD or SPMD synchronization model, distributed data, and code to operate on that data. A module is encapsulated by a composite procedure in the case of an array of virtual processors, or a normal procedure in the case where only a single virtual processor executes.

From this basis, a more complex parallel program can be created through various combinations of calling SIMD composite procedures, SPMD composite procedures, and normal procedures. Conceptually, this creates a more complex parallel virtual machine whose size, shape, and synchronization characteristics describe the parallel nature of the program.

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 representing the single procedure 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, 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 is 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). An example of this is solving block bordered systems of equations, where the natural degree of parallelism changes between the phase of the algorithm that operates on the main 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 a SIMD model for the LU phase to a SPMD model for the backsolve, and the number of virtual processors may change from the number of equations to the number of actual processors.

Phased parallelism, liked 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 language that uses a barrier statement in this manner for numerical programs is found in 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 the original composite procedure to that of the called composite procedure, and then back again after the barrier is exited.

An example of phased parallelism is shown in figure 3, a very simplified version of a procedure used to solve a block bordered system of equations. In this example, the main data structures consist of $Q+1$ diagonal blocks in the system of equations. Q of them are contained in A and each are of size NxN, and the lower right block, of size MxM, is represented by P. There are also border blocks that are not shown in the exam-

```
synch composite lu_decomp (B, NP) [NP:id]  . . .

synch composite modify_P (A, P) [M:id]  . . .

synch composite compose_x (A, P) [Q:id]  . . .

composite block_solve(A, P)[Q:id]
    double private A[Q][N][N] map slice();
    double private P[M][M] map wrapcol();
    {
    lu_decomp(A[id], N);
    barrier
        {
        modify_P(A, P);
        lu_decomp(P, M);
        }
    compose_x(A,P);
    }
```

An Example of Phased Parallelism
Figure 3

ple. The first part of the computation consists of factoring each block in A. This implies parallelism of degree Q, the number of virtual processors in *block_solve*. (The call to lu_decomp invokes nested parallelism and increases the parallelism to QN.) After the completion of this step, the results are used to modify P (using the border blocks), and then P is factored. Both of these steps have parallelism of degree M, and therefore the Q virtual processors used to factor A are temporarily transformed into M virtual processors. This is done with the barrier statement. (Note that lu_decomp is called with a different number of virtual processors in the second call.) After the barrier statement, the algorithm returns to another phase that has parallelism of degree Q at the high level and QN at the lower level.

There are other mechanisms for generating complex virtual machines in Dino2 that we only mention briefly. If a statement inside an SIMD or SPMD composite procedure is a call to a normal C procedure, then the degree of parallelism is unchanged, but while the normal procedure is executing on each virtual processor there is no synchronization between the processes. 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. A second mechanism consists of taking two virtual machines and combining them into a single virtual machine. This is implemented with the '::' statement and consists of executing two modules (composite or normal procedure) concurrently. This construct allows for functional parallelism, as opposed to data parallelism. Generally, when this is used in numerical computation it is at a high level within a program. An example is a program that uses a master/slave model to service a set of independent tasks.

6. Communication Model

Another important aspect of Dino2 is the method in which virtual processors communicate with each other. Because the language is designed for data parallel computation, there is strong support for distributing data structures across virtual processor structures. This distribution provides a natural mechanism for specifying communications. Communications are based upon reading and writing these distributed data structures.

The distribution of a data structure onto the virtual processors is specified by a mapping function. Mapping functions in Dino2 map arrays onto arrays and consist of a small language describing how elements of one array are mapped onto the other [16]. These mapping functions are similar to those found in [14] and [5] and are not described here.

An important aspect of making the Dino2 communications model simple is that the semantics of the communications are entirely embedded in the data type of the distributed structures being read from or written to. Our experience has been that applying special functions or operators to data structures to generate communications (such as the

DINO # operator) is confusing and error prone. Libraries containing send and receive functions also tend to be difficult to use because the semantics of issues such as when the data will be sent, when it will be available, or where it will go are usually not very simple. This is compounded by the fact that there are usually multiple versions of each of the basic send and receive functions, such as those for synchronous or asynchronous operations. By limiting the number of operations that can be applied to the data structures via their data type, the user has less chance of making an error because the semantics of the operation were misunderstood. In order to match the flexibility and power of the virtual machine modules, the communications model in Dino2 has also been designed with a large degree of flexibility and modularity.

The requirement for flexibility and the requirement that there not be any special operators or functions associated with communications suggest 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*. 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. The semantics of these communication types varies slightly with the synchronization model that is used. Within the SIMD model, remote variables can be accessed by any virtual process and are non-buffered, meaning that the most recently assigned or communicated value is used when the variable is read. Private variables can only be read or written by the virtual process that contains it. Buffered remote variables are not allowed within SIMD virtual processors. Within SPMD virtual processors all three communication types are allowed. Private and remote variables have the same semantics as those in the SIMD model. Note that remote variables, used in the context of the SPMD model, permit a "chaotic" communication model because, in contrast to the SIMD model, there is no synchronization between virtual processors. Buffered remote variables are similar to remote variables except that they have a buffered implementation. This means writes to a variable are buffered in the order they arrive and reads block until a value is present in the buffer, at which point that value is used and removed from the buffer. Within normal C procedures, only private variables can be read or written.

The concept of communication types for variables in parallel programs 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. The communication types provide the same flexibility as a communications library with respect to the types of communication paradigms that can be employed. Just as different communication models are described in the communication Intel library for the iPSC series of hypercubes, for example, different communication types are supported in Dino2. The fundamental difference is that in Dino2, the communication model associated with a specific variable is static, whereas with a library, multiple communication models can be used dynamically with the same variable. The static nature of a communication type in Dino2 should be much less error prone than using a library because it is not possible to accidentally mix communication models.

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. 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. 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.

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 wide range of techniques that are not possible to combine in other languages without introducing unmanageable complexity into some portion of the code. This modularity should also allow large programs to be written because of the independence between modules. Finally, the characteristics of the modules have been designed to permit efficient execution. Many implementation considerations associated with the language are discussed in [16], but a full implementation of the language has not yet been performed.

REFERENCES

[1] F. Andre, J. Pazat and H. Thomas, "Pandore: A System to Manage Data Distribution", *Proceedings of ACM ICS*, June. 1990.

[2] D. Callahan and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors", *J. Supercomputing* 2, 2 (Oct.. 1988), 151-169.

[3] W. Griswold, G. Harrison, D. Notkin and L. Snyder, "Scalable Abstractions for Parallel Programming", *Proceedings of the Fifth Distributed Memory Computing Conference*, Apr.. 1990.

[4] L. Hamey, J. Webb and I. Wu, "Apply, A Programming Language for Low-Level Vision on Diverse Parallel Architectures", in *Parallel Computation and Computers for Artificial Intellegence*, J. Kowalik (editor), Kluwer Academic Publishers, 1987.

[5] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer and C. Tseng, "An Overview of the Fortran D Programming System", Technical Report CRPC-TR91121, Rice University, Mar. 1991.

[6] H. Jordan, "The Force", in *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon and R. J. Douglass (editor), MIT Press, 1987, ch 16.

[7] K. Kennedy and H. Zima, "Virtual Shared Memory for Distributed-Memory Machines", *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, Mar.. 1989, 361-366.

[8] C. Koelbel, P. Mehrotra and J. V. Rosendale, "Supporting Shared Data Structures on Distributed Memory Architectures", *Conf. on Principles and Practice of Parallel Processing*, March, 1990.

[9] R. J. Littlefield, "Efficient Iteration in Data-Parallel Programs with Irregular and Dynamically Distributed Data Structures", Technical Report 90-02-06, University of Washington, 1990.

[10] M. Philippsen, W. Tichy and C. Herter, "Modula-2* and its Compilation", *Proceedings of the First International Conference of the Austrian Center for Parallel Computation*, Salzburg, Austria, Sep 1991.

[11] M. J. Quinn and P. J. Hatcher, "Data Parallel Programming on Multicomputers", *IEEE Software*, Sep. 1990, 69-76.

[12] A. Rogers and K. Pingali, "Process Decomposition Through Locality of Reference", *Proceedings of the SIGPLAN Notices Conference on Programming Languge Design and Implementation*, June. 1989, 69-80.

[13] J. R. Rose and G. L. S. Jr., "C*: An Extended C Language for Data Parallel Programming", PL87-5, Thinking Machines Corp., 1987.

[14] M. Rosing, R. B. Schnabel and R. P. Weaver, "The DINO Parallel Programming Language", *Journal of Parallel and Distributed Computing*, Sep 1991, 30-42.

[15] M. Rosing, R. B. Schnabel and R. P. Weaver, "Scientific Programming Languages for Distributed Memory Multiprocessors: Paradigms and Research Issues", Tech Report CU-CS-537-91, Univ. of Colorado, Dept. of Computer Science, 1991.

[16] M. Rosing, *Efficient Language Constructs for Complex Parallelism on Distributed Memory Multiprocessors*, PhD Thesis, University of Colorado, Boulder, Aug 1991.

[17] P. Tseng, *A Parallelizing Compiler for Distributed Memory Parallel Computers*, PhD Thesis, Carnegie Mellon, May 1989.