Massive Parallelism and Process Contraction in Dino

Matthew Rosing, Robert B. Schnabel, and Robert P. Weaver

CU-CS-467-90 March 1990

Department of Computer Science
Campus Box 430
University of Colorado,
Boulder, Colorado, 80309 USA

This research was supported by AFOSR grant AFOSR-85-0251, and NSF Cooperative Agreement CDA-8420944.

To appear in proceedings of The Fourth SIAM Conference on Parallel Processing for Scientific Computing, held in Chicago, Illinois, December 1989.

Abstract. Dino is a language, built upon C, for expressing parallel numerical programs on MIMD distributed memory multiprocessors. We describe new capabilities that we are designing for the Dino language and compiler that will make it possible to specify massively parallel, SIMD numerical computations in a natural way, and still have them run efficiently on distributed memory multiprocessors that may only have a moderate number of actual processors and relatively slow interprocessor communication. This is accomplished by writing programs with a large number of virtual processes, and having the Dino compiler automatically contract them into efficient programs with a smaller number of actual processes.

1. Introduction. We describe language constructs that allow for the natural and efficient implementation of massively parallel, SIMD-like computations in a language intended for distributed memory, MIMD computation. This is accomplished by writing programs with a large number of virtual processes/processors, and having the compiler automatically contract these to one process per physical processor. In a general MIMD programming language, it is extremely difficult to contract processes because of the uncertainty at compile time of the communication dependencies between the processes. To address this problem, we propose adding the ability to declare that certain program portions will execute in a pseudo-SIMD fashion. This provides enough information for the compiler to generate efficient contracted code and, in the case of many numerical applications, also allows for a more natural description of the parallel algorithm.

The concept of a virtual parallel machine has become useful in parallel languages because it provides a clean abstraction of the actual machine while also reflecting the amount of parallelism in the parallel algorithm. However, where it would be most natural to declare a virtual machine having the full parallelism of the algorithm, efficiency considerations often force the declaration of a machine having the parallelism equivalent to the number of available processors. For example, a program operating on a 128 x 128 grid might naturally be written with 16384 virtual processors that each handle one grid point (see Appendix). But if only 32 processors are available then the programmer often has to block the code into 32 processes that each handle 512 grid points. There may be the alternative of multiprogramming the nodes, i.e. having multiple processes time-slice on each physical processor, but operating systems overhead generally makes this option inefficient.

*Research supported by NSF Cooperative Agreement CDA-8420944 and AFOSR grant AFOSR-85-0251

The reason that there are no compilers that contract fully MIMD parallel programs into programs with smaller numbers of processes has to do with the great difficulty of determining at compile time when interprocess communications will be required, which in turn makes it very difficult to determine the order of execution. The solution we propose is to allow the user to specify, where appropriate, that sections of the program execute in a pseudo-SIMD mode. By pseudo-SIMD we mean that the communication patterns are those that would result if the program were executed in a fully SIMD manner. We expect that this restriction will provide the compiler with enough information to efficiently contract the code. At the same time, it should provide sufficient expressiveness for many numerical algorithms, as is evidenced by the appropriateness of vector and fully SIMD computers for many numerical computations.

The only other research projects of which we are aware that have addressed these issues have done so by using a more restricted programming model. Kali [KMR90] allows the programmer to specify a fully parallel virtual machine, but the programming model is restricted to a "copy-in, copy-out" model where all communication between concurrently executing processes occurs at the beginning and end of "forall" loops. Spot [Soc90] utilizes a similar paradigm. C* [RoJ87] is intended for an SIMD computer and thus uses a fully SIMD model. We believe that the model proposed here is less restrictive and allows natural specification of a considerably wider range of numerical computations.

In Section 2 we describe the DINO language [RSW90], which serves as the basis for our new language constructs. Section 3 discusses problems with contracting processes in a fully MIMD language. Section 4 describes a change to the MIMD model that permits contraction. In Section 5 we discuss some of the more important compiler issues and optimizations required to implement the proposed changes.

2. DINO Language Overview. When programming in DINO, the programmer first defines a virtual parallel machine that best fits the major data structures and communication patterns of the algorithm. This arrangement of processors is called a structure of environments. Second, the programmer specifies the way that data structures will be distributed among the virtual processors. This is called distributed data. Last, the programmer provides algorithms that will run on each processor (each environment in the same structure of environments contains the same algorithms). These are called composite procedures. Parallelism results from simultaneous execution of all the copies of an algorithm in a structure of environments. We now discuss each of these features briefly.

The programmer defined structure of environments is the underlying parallel model in DINO. Typically, the structure is a single or multiple dimensional array, thus defining a virtual parallel machine with this topology. Each environment in a given structure consists of (identical) data structures and procedures, and is similar to a process. An environment may contain multiple procedures, but only one procedure in an environment may be active at a time.

The key to constructing a parallel DINO program is the mapping of data structures to the structure(s) of environments. DINO encourages the programmer to take a global view of data structures that are distributed among multiple processors and operated on concurrently. The programmer does this by declaring a data structure as distributed data and providing its mapping onto the structure of environments. Individual data elements may be mapped to one or multiple environments. These mappings determine how the environments will access and share the data. DINO provides an extensive library of standard mapping functions, and the user can create additional mapping functions.

Distributed variables are the key to making interprocess communication in DINO natural and implicit. One way this occurs is by using distributed variables as parameters to composite procedures; the parameter is distributed to or collected from the appropriate environments according to its mapping function. The second way is to make a remote access to a distributed variable within the body of composite procedure. Such accesses, which are any standard references to the variable followed by a # sign, generate sends (for writes) or receives (for reads) of the distributed variable according its mapping function. The sends and receives are synchronized using a produce-consume paradigm. Examples of these constructs are contained in Appendix A. We will not discuss them further as the language constructs we propose cause them to be replaced with a more natural programming style.

A composite procedure is a set of identical procedures, one residing within each environment of a structure of environments, that are executed concurrently. The parameters of a composite procedure typically include distributed variables. Composite procedures are called from the main, *host* environment that is part of every DINO program. A composite procedure call causes an instance of the procedure to execute in each environment of the structure, utilizing the portion of the distributed parameters that are mapped to its environment. This results in a SPMD or single program multiple data form of parallelism. There is some support for functional parallelism in DINO, but this is not discussed in this paper.

An example DINO program is given in Appendix A. This program takes a digital image, applies a smoothing operation over it, and then computes the maximum second derivative over the x and y axes of the image. It uses a virtual machine, named *grid*, with one processor per grid point; each grid point $U[x][y]$ is mapped to $grid[x][y]$ and its four nearest neighbors.

3. Problems With the MIMD Model. DINO and languages using similar programming models do not always allow the user to specify a virtual machine whose size equals the amount of parallelism in a program. For example, the program in appendix A would not be suitable for a machine containing 32 processors because the degree of multiprogramming required would be too high. Instead, without the ability to contract virtual machines, programs must often be written using the exact number of available processors. This causes two problems. First, such programs are longer and more complicated than one element per processor versions. This is because each environment must handle multiple elements, and separate, vectorized communication statements must be inserted for efficiency. The second problem is that this programming style makes it difficult to write modular code. A programmer may want to use the routine at the highest level of the computation, where it can use all the processors in the machine, or at a low level where it is executed concurrently with other routines and has only one or a few processors available to it. The programmer might also like to use The routine on different machines with different numbers of processors. Without the ability to contract code there will need to be multiple versions of the routine. A contractable language, however, will allow the routine to be written in a way which is independent of the number of processors available to it at run time.

This discussion motivates one language capability we would like to have, the ability to specify a virtual machine having the *maximum parallelism* of the algorithm. There are two other related capabilities, pertaining to the way virtual machines change over time, that we would like to have available. The first, called *nested parallelism*, occurs when each of the n elements of a virtual machine expands to become an m element virtual machine, resulting in degree nm parallelism overall. An example is a matrix vector multiply which, at one level, has parallelism equal to the number of rows but than changes to the number of matrix elements as each row does a dot product with the vector. The second, called *phased parallelism*, is when an entire virtual machine of n elements is replaced by a virtual machine of m elements.

In order for each of these three types of virtual machine constructs, maximum, nested, and phased, to be efficiently implementable, the compiler must be able to combine multiple virtual processes into a single process. The reason that DINO, along with other MIMD languages, does not permit this is the difficulty in contracting processes. The problem is that communication dependencies, which prescribe the order of execution, may be extremely difficult to compute at compile time. The only other alternative is using multiprogramming, coroutines, or something similar to implement the full parallelism of the virtual machine, but this is too inefficient.

4. A Modification to the MIMD Model. To provide the ability to contract processes, as well as the three virtual machine constructs discussed above, we propose to add the ability to declare, where appropriate, that composite procedures run in a pseudo-SIMD mode. By pseudo-SIMD we don't suggest that the processes synchronize after each instruction, but rather that their results are equivalent to the results that would be obtained in running them in an SIMD mode. That is to say, the data dependencies involving communications between processes that are implied by the SIMD model are retained, and the remaining synchronizations due to the SIMD model are removed. In conjunction, # operators are no longer used following some accesses to distributed variables, although communication still occurs only through accesses to distributed variables. This programming paradigm ensures that there is enough information at compile time to determine the points of communication, and hence the order of execution of the contracted program.

The pseudo-SIMD mechanism in DINO is implemented by allowing the user to specify that the composite procedure is "synchronous" as opposed to the MIMD default which we refer to as asynchronous. For example, a pseudo-SIMD version of the program in Appendix A is obtained by inserting "synchronous" before "composite smooth" and removing the # signs that were used with the array U. Specifying the synchronization mode at the procedure level makes it easy to separate the two models. This mechanism also allows flexible mixing of the two models. It is possible for asynchronous procedures to call synchronous procedures, and vise versa.

The three types of constructs for building complex virtual machines, which we called maximum, nested, and phased parallelism, can be implemented easily using synchronous composite procedures. The first is trivially expressed by specifying a large enough virtual machine. The second is implemented by having a call to an m element composite procedure Y as a statement within an n element synchronous composite procedure X. This results in nm elements of the procedure Y executing concurrently. As in the current definition of DINO, each element of the higher level composite procedure X suspends execution until all m elements of the lower level composite procedure Y that it called have completed. A difference is that current DINO would only allow one copy of the lower level composite procedure (Y) to be instantiated at once, whereas the new version will allow all n copies to be instantiated concurrently. The third construct, phased parallelism, where the computation changes parallelism from n processes to m (and possibly back again) is closely related to nested parallelism and is expressed very similarly. The only difference is that the keyword *phase* precedes the call to the second composite procedure within the first one, and signifies that only one copy of it, rather than n, are to be created.

The pseudo-SIMD model of parallelism is appropriate for many numerical algorithms, as is exemplified by the fact that SIMD machines and vector computers are widely used. By having both MIMD and pseudo-SIMD models in the language, we permit algorithms that do not fit the SIMD model, so that the language can be fully expressive. An important capability in this regard is that pseudo-SIMD procedures can include calls to standard C functions; the pseudo-SIMD semantics are not inherited by these functions.

A beneficial side effect of specifying pseudo-SIMD procedures is that communication is more implicit than in fully MIMD languages. The reason is that the programmer does not need to distinguish syntactically between standard accesses to distributed variables and accesses that entail communication. Rather, the compiler or run-time system can generate the receives and sends based on the reads and writes in the program. This is possible because the points of communication are quite easily determined. This frees the programmer from having to consider many lower level details that can make parallel programming more difficult than sequential programming. It may also permit simple programming solutions in common situations that are not as naturally addressed in a MIMD language; an example would be the statement "U[i][j] = U[(i-1)%N][j]" which would rotate U by rows in the example in the appendix. Note that such conveniences are also possible in languages like Kali [KMR90] and Spot [Soc90] that use even more restrictive models than the pseudo-SIMD model discussed here.

5. Compilation Issues. Several tasks are required of the compiler to efficiently contract virtual machines. The first major task is to determine the number of processors allocated to each procedure and than map the virtual processors to the actual processors. The issues involved in constructing this mapping are very similar to those we currently consider in mapping distributed data to environment structures. In particular, the mapping may be made primarily to achieve communication efficiency or load balancing. In the former case, one might place consecutive blocks of virtual processors on consecutive real processors (if communication is with nearest neighbors) while in the latter case one might use a wrap mapping of processes to processors. It may prove useful to introduce a new construct for mapping environments to processors for this purpose.

The second major compiler task is actually contracting virtual processors. That is, given a synchronous composite procedure that defines the code to be executed by one virtual processor, transform it into a composite procedure that will execute the code of several of these virtual processors, while preserving the meaning of the program. In the case where there is no communication within the body of the composite procedure, this transformation simply consists of placing a for loop around this code body. Where there is communication the transformation is more complicated. The main purpose for adding a synchronized model is to provide the compiler with enough information to perform this transformation. The compiler first determines which statements may require communication, by identifying references to distributed variables that may not be mapped to that virtual processor. The simplest transformation is then to place each block of straight line code between consecutive possible communication points within a for loop, and generate communication between these segments.

Many optimizations can be made to this simple transformation. One is concerned with presending messages. In many synchronized composite procedures we expect that the compiler will be able to determine exactly where communicated data will be needed, and where it will be generated, by using the same data dependence analysis found in vectorizing compilers. Based on this analysis the compiler will be able to generate sends at the point where a needed value is produced, rather than waiting until the point at which it is required. This optimization will allow the overlapping of communication with computation.

A somewhat more complicated optimization that is also important is to aggregate communications. As an example, suppose a composite procedure written for N virtual processors is to be contracted down to P processors, and that its body is a for loop that is iterated M times, with each iteration consisting of a computation followed by a communication. If the dependence analysis shows that the M iterations are independent, then the contracted procedure could be written with the outer loop running over the N/P virtual processors and the inner loop running over the M iterations, or vise versa. Depending upon the communication statement, either order may be superior to the other with regard to aggregating communications to a single (group of) processor(s). This type of optimization is very similar to what vectorizing compilers do except that the communications are vectorized instead of the arithmetic operations.

A final optimization that we feel is important has to do with data storage. The simplest transformation of non-distributed data structures is to generate one copy of each variable for each virtual processor that the contracted procedure handles, i.e. N/P copies of each in the above example. In many cases, however, only one copy of the variable is needed. For example, consider a composite procedure containing a loop control variable. It is quite possible that only one copy of this variable is needed. This is guaranteed to be true if the times between the generation and use of the variable in the contracted procedure do not overlap. This optimization is currently performed in many serial compilers. One copy is also sufficient if all the values can be shown to be the same at compile time.

APPENDIX -- A Simple, Massively Parallel DINO Program

```
#define N 128
#include <dino.h>

environment grid [N:x][N:y] {
    composite smooth(in U, out maxder)/*preceded by "synchronous" with new constructs*/
    float distributed U[N][N] map FivePoint;/*maps U[x][y] to grid[x][y] and 4 neighbors*/
    float distributed maxder[2] map all;
    {
    float myderiv[2];

    if ((x != 0) && (y != 0) && (x != N-1) && (y != N-1))
        U[x][y]# = (U[x][y] + U[x-1][y] + U[x+1][y] + U[x][y-1] + U[x][y+1]) / 5;
      /* all # signs are deleted from distributed variables when using new constructs */
    else U[x][y]# = BorderSmooth (U, x, y); /*BorderSmooth omitted for brevity*/
    if ((x != 0) && (x != N-1))
        myderiv[0] = U[x][y] - (U[x-1][y]# + U[x+1][y]#)/2 ;
    else myderiv[0] = 0;
    if ((y != 0) && (y != N-1))
        myderiv[1] = U[x][y] - (U[x][y-1]# + U[x][y+1]#)/2 ;
    else myderiv[1] = 0;
    maxder[] = gmax(myderiv[])#;
    }
  }
environment host {
    float U [N][N];
    float max [2];

    main ()
        {
        InputData (U);  /* omitted for brevity */
        smooth (U[][], max[])# ;
        printf(" Max 2nd Derivative of Smoothed Data: x axis is %.3f", max[0]);
        printf(" Max 2nd Derivative of Smoothed Data: y axis is %.3f", max[1]);
        }
    }
```

REFERENCES

[KMR90] 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.

[RoJ87] J. R. Rose and G. L. S. Jr., "C*: An Extended C Language for Data Parallel Programming", PL87-5, Thinking Machines Corp., 1987.

[RSW90] M. Rosing, R. B. Schnabel and R. P. Weaver, "The DINO Parallel Programming Language", TR CU-CS-457-90, Univ. of Colorado Dept. of Computer Science, 1990.

[Soc90] D. G. Socha, "Spot: A data parallel language for iterative algorithms", TR 90-03-01, Univ. of Washington, Dept. of Computer Science, 1990.