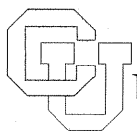
An Overview of Dino - - A New Language for Numerical Computation on Distributed Memory Multiprocessors *

Matthew Rosing
Robert B. Schnabel

CU-CS-385-88



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This research was supported by AFOSR grant AFOSR-85-0251, and NSF cooperative agreement DCR-8420944.

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR AND DO NOT NECESSARILY REFECT THE VIEWS OF THE NATIONAL SCIENCE FOUNDATION

An overview of Dino--
a new language for numerical computation on distributed memory multiprocessors

Matthew Rosing and Robert B. Schnabel

CU-CS-385-88 March 1988

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 DCR-8420944.

To appear in proceedings of The Third SIAM Conference on Parallel Processing for Scientific Computing, held in Los Angeles, CA, Dec 1987.

Abstract. We briefly discuss the design of a new language, called Dino, for programming parallel numerical algorithms on distributed memory multiprocessors. A significant difficulty with most current approaches to programming such computers is that interprocess communication and process control must be specified explicitly through messages, thereby making the parallel program difficult to write, debug, and understand. Our approach is to add several high level constructs to standard C that allow the programmer to describe the parallel algorithm to the computer in a natural way, similar to the way in which the algorithm designer might informally describe the algorithm. These constructs include the specification of a data structure of virtual processors that is appropriate for the problem, and the ability to map data and procedures to this virtual parallel machine. Parallelism is achieved through a concurrent procedure call that utilizes these data and procedure mappings. All the necessary interprocess communication and process control results implicitly through these constructs.

1. Introduction. Dino ("DIstributed Numerically Oriented language") is a new language for writing numerical programs for distributed memory multiprocessors. By distributed memory multiprocessor we mean any computer consisting of multiple processors with their own memories and no shared memory, which communicate by passing messages. Examples include hypercubes, and networks of computers used as multiprocessors.

The main goal of our work is to make parallel numerical programs for such computers easy to write and understand. The approach we take is to try to make the programs similar to the natural descriptions of parallel algorithms that algorithm designers often use in explaining their methods. Inherent in this approach is that interprocess communication and process control should be implicit in the language constructs.

The constructs of Dino use the fact that many numerical algorithms are highly structured. The main data structures used in these algorithms are arrays and possibly trees. The processes that execute in parallel usually are also highly structured; sometimes the algorithm consists of "single program, multiple data" segments where the same code executes on different processors and different parts of the data structure simultaneously. In conjunction, the distribution of data among processors, and the communication between them, typically follows regular patterns.

The Dino language allows such parallel numerical algorithms to be described in a natural,

Department of Computer Science, University of Colorado, Boulder CO 80309. Research supported by AFOSR grant AFOSR-85-0251, and NSF cooperative agreement DCR-8420944.

top down manner. It provides both a mechanism for efficiently distributing data over the processors of a distributed memory machine, and the ability to easily operate on that distributed data concurrently. Interprocess communications is implicit in the data mapping constructs, and is therefore less subject to programmer errors than sending and receiving messages. The key to these capabilities is the ability of the user to define a data structure of virtual processes (called environments) that fits the algorithm and data structures.

Dino consists of extensions to standard C. We have chosen C for several reasons. It is available on all the target parallel machines we have considered; it is a structured language, which complements the new, highly structured characteristics of DINO; and there are a wealth of compiler and other tools associated with C which considerably ease the task of implementing the new language. In addition, by choosing C we have been able to use C++ [6] for our initial, prototype implementation.

To our knowledge, relatively little high level language design has been done for distributed memory multiprocessors. Languages such as Linda [1,2] and Pisces [5] support distributed numerical programming, but have a more low level orientation to issues such as communication. We were partially motivated by languages such as the Force [3,4] which have greatly facilitated parallel numerical programming on shared memory multiprocessors; we would like to bring a similar level of ease to programming distributed memory multiprocessors.

The rest of this paper briefly and informally introduces the main ideas in Dino, gives one simple example of a Dino program, and gives a very brief synopsis of the status and directions of this work. Subsequent papers will describe Dino in more detail and discuss our experience with using it more thoroughly.

2. Dino Overview. The goal of Dino is to allow the programmer to communicate a distributed parallel algorithm to the computer in a way that is similar to the natural way that we often observe algorithm designers informally describing their methods. To facilitate this, Dino provides two important new capabilities, distributed data structures and composite procedures. Both are in turn based upon an underlying data structure of virtual concurrent processors (environments) that is provided by the user. We now briefly describe the characteristics of each of these three fundamental aspects of Dino.

2.1. Environments. The key construct that allows Dino to provide a natural, high level description of a parallel algorithm is a user defined structure of environments. An environment consists of data and procedures. It may contain multiple procedures, but only one procedure in an environment may be active at a time. Thus each environment can correspond to a process, and this is how environments are implemented in our Dino prototype. A more general possibility is mentioned in Section 3.

To create a parallel algorithm, the user declares a structure of environments which best fits the number of processes, and the communication pattern between processes, in the parallel algorithm. This structure can be viewed as a virtual parallel machine constructed for the particular algorithm. In our experience, the most common structures of environments are one, two, and higher dimensional arrays. This is because the parallelism in numerical algorithms often derives either from the partitioning of physical space into neighboring regions, or from the partitioning of arrays, both of which result in parallel algorithms whose data mappings and procedural parallelism are naturally described in terms of arrays of processors. It is possible, however, to use any data structure in defining a structure of environments.

2.2. Distributed data. Dino allows the user to specify mappings, either one to one or one to many, of data structures to the underlying virtual machine structure given by the user-defined structure of environments. These mappings, which are specified as part of the declaration of the data structure, are selected according to how the processors will access and share the data. They are the key to making interprocess communication natural and implicit.

An example of distributed variables is illustrated in Fig. 1 and in the Dino program in the Appendix. Suppose we wish to solve Poisson's equation with zero right hand side,

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0 \ ,$$

on a square domain with some given boundary condition by a simple finite difference method. In this method we discretize the variable space into $U_{i,j}$, $i=0,\cdots,N$, $j=0,\cdots,N$, and then iteratively apply the formula

$$U_{i,j} = (U_{i+1,j} + U_{i-1,j} + U_{i,j+1} + U_{i,j-1}) / 4 \qquad (2.1)$$

to calculate the new value at each grid point except the border points, until the values converge. The natural topology for this problem is a two dimensional grid. If we assume for simplicity that we create a unique environment $e_{i,j}$ for each grid point except the border points, then the natural structure of environments is also a two dimensional array, $e_{i,j}$, $i=0,\cdots,N-1$, $j=0,\cdots,N-1$. Now from equation (2.1), each variable $U_{i,j}$ (except border variables) is used in the environment where it resides, and the environments directly to the north, south, east, and west. Thus the distributed data mapping function is to map each $U_{i,j}$ to $e_{i,j}$, $e_{i+1,j}$, $e_{i-1,j}$, $e_{i,j+1}$, and $e_{i,j-1}$.

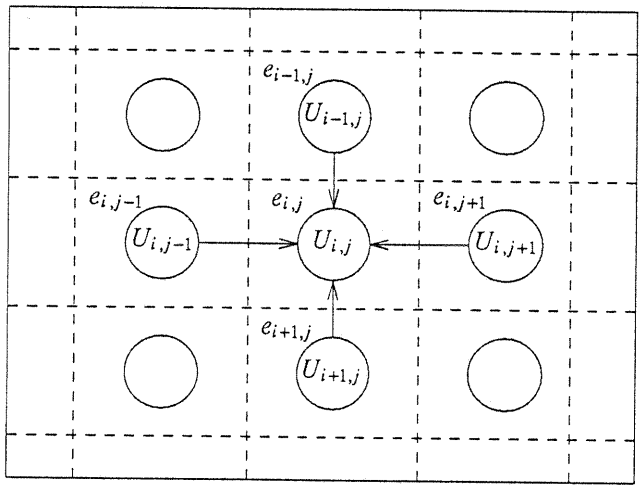


Fig. 1. Distributing a partial differential equation calculation

The variable $U_{i,j}$ is an example of a distributed variable that is mapped to multiple environments. In this case a local copy of that variable will exist in each of the environments to which it is mapped. A procedure in any of these environments can then access the variable either locally or remotely. A local access, which uses standard syntax, affects just the local copy and is the same as any standard reference to a variable. A remote access, which uses the syntax *variable name#*, is used to generate interprocess communication. A remote assignment to a distributed variable generates a message that is sent to every other processor to which that variable is mapped, while a remote read of a distributed variable will receive such a message and update the variable's value, in one of two ways that are described below. Thus one to many mapping functions can be thought of as defining locally shared variables, where certain data elements are shared between certain subsets of the processors. This mapping provides the information required to automatically generate the necessary sends and receives when these variables are accessed remotely.

Dino distributed variables may be either *synchronous* and *asynchronous*. The default is synchronous, but a distributed variable may be made asynchronous by placing "asynchronous" before "distributed" in its declaration. In either case, a remote write causes a message, with a new value of the distributed variable, to be sent to a buffer in each other environment to which

that variable is mapped. A remote read of a synchronous variable causes it to overwrite its local copy with the *first* value that has been received since the last remote read; if no new value is present, it blocks until one is received. A remote read of an ansynchronous variable causes it to overwrite its local copy with the *last* value that has been received since the last remote read; if no value has been received, it retains its current local value, and does not block. A nice consequence of this construction is that by changing a distributed variable's declaration from synchronous to asynchronous, a program can be changed from a data synchronous parallel program to an asynchronous ("chaotic") program.

Dino provides an extensive collection of standard mapping functions, such as the *NSEWoverlap* mapping used in the example. The user can also create arbitrary mapping functions.

2.3. Composite procedures. A composite procedure is a set of identical procedures, one residing within each environment of a structure of environments, which is called concurrently. Its parameters typically include distributed variables. A composite procedure call causes each procedure to execute, utilizing the portion of the distributed parameters, and possibly other distributed data structures, that are mapped to its environment. This results in a "single program, multiple data" form of parallelism. There is no need to explicitly send code to each processor, to initiate execution on individual processors, or to explicitly distribute or collect data among processors.

Distributed variables that are parameters to composite procedures can be declared as input, output, or input/output parameters. Upon invocation of the composite procedure, a input parameter is distributed based upon its mapping function. That is, the value of each of its elements is sent from the calling procedure to each environment to which that element is mapped. Upon terminatation of the composite procedure, the value of each output parameter is sent from the environment to which it is mapped back to the calling procedure. If the mapping is one to many, the mapping function specifies from which environment the value should be returned.

In the Poisson solver example, the composite procedure *Poisson* performs *itns* iterations of equations (2.1) at grid point $U_{i,j}$. The distributed variable U is a one to many, input and output parameter. The mapping function specifies that upon process termination, the value of each $U_{i,j}$ should be retrieved from the environment $e_{i,j}$.

In summary, the main advantage of composite procedures, together with distributed data and environments, is that they permit a natural, high-level description of many parallel algorithms, while making the details of interprocess communication implicit in the language.

3. Current Status and Future Directions. We have implemented a prototype of Dino using C++ [6]. It runs on the Intel hypercube, the hypercube simulator, and on our network of Sun workstations. We have written parallel programs for a variety of numerical algorithms in Dino/C++, as well as paper programs in standard Dino. Our opinion is that these programs are usually considerably easier to write and understand than the same parallel programs in existing languages for distributed memory multiprocessors; subsequent papers will give a larger number of examples that help support this claim. Another benefit of the prototype has been to enable us to understand better the low level issues in implementing Dino.

The Dino research is leading in a number of interesting directions, which we are beginning to pursue. First, we continue to consider new features for Dino. Mainly these are enrichments which would allow Dino to express a broader class of parallel algorithms easily. Examples include multiple distributed mapping functions for a single data structure, multiple environments in a single program, and facilities for supporting dynamic process structures and dynamic distributed data structures. A second area is tools for optimizing parallel programs. The information about interaction between environments that is readily available in a Dino program provides the opportunity to make good mappings of environments to processes, and of processes to the processor topology of the target computer. The latter is already done in a simple way in the C++ prototype. The former might allow the programmer to specify environments at a finer grain than there are processors, when this is the natural way to describe the algorithm, and then have a tool make

a good decision about how best to bundle the environments into processes. Finally, the graphical nature of the basic Dino constructs, such as the structure of environments and the data mapping functions, makes in natural to consider a graphical interface to Dino.

REFERENCES

(1) N. CARRIERO and D. GELERNTER, *The S/Net's Linda kernel,* ACM Transactions on Computer Systems 4, 1986, pp. 110-129.

(2) D. GELERNTER, N.CARRIERO, S. CHANDRAN, and S.CHANG, *Parallel programming in Linda,* in Proceedings of the 1985 International Conference on Parallel Processing, IEEE Press, 1985, pp. 255-263.

(3) H. F. JORDAN, *The Force,* in The Characteristics of Parallel Algorithms, L. H. Jamieson, D. B. Gannon and R. J. Douglass, Eds., MIT Press, 1987, pp. 395-436.

(4) H. F. JORDAN, *Structuring parallel algorithms in an MIMD, shared memory environment,* Parallel Computing 3, 1986, pp. 93-110.

(5) T. PRATT, *The Pisces 2 parallel programming environment,* in Proceedings of the 1987 International Conference on Parallel Processing, IEEE Press, 1987, pp. 439-445.

(6) B. STROUSTRUP, *The C++ Programming Language,* Addison-Wesley, Reading, Massachusetts, 1986.

Appendix -- Dino Program for Parallel Solution of Poisson's Equation

```
#define N 128
environment node[N:xid][N:yid]{
        composite Poisson(U, in itns)
                distributed float U[N+2][N+2] : NSEWoverlap;
                int itns; /*number of iterations*/
                {
                int i,x,y;
                x = xid + 1;
                y = yid + 1;
                        /*calculate using local values to start cycle*/
                U[x][y]# = (U[x+1][y] + U[x-1][y] + U[x][y+1] + U[x][y-1])/4;
                        /*calculate remaining iterations using remote values*/
                for (i=1; i<itns; i++)
                        U[x][y]#  = (U[x+1][y]# + U[x-1][y]# + U[x][y+1]# +
                                U[x][y-1]#)/4 ;
                } /*poisson*/
        } /*node*/
environment host{
        float G[N+2][N+2];
        main(){
                initPoisson(G);
                Poisson(G[][], 250)#;
                display(G);
                } /*main*/
        }/*host*/
```