The DINO User's Manual

T. M. Derby, E. Eskow, R. K. Neves, M. Rosing,
R. B. Schnabel, and R. P. Weaver

CU-CS-501-90 November 1990

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

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

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

Contents

1 Introduction

DINO (DIstributed Numerically Oriented language) is a language for writing parallel programs for distributed memory (MIMD) multiprocessors. It is oriented towards expressing data parallel algorithms, which predominate in parallel numerical computation. Its goal is to make programming such algorithms natural and easy, without hindering their run-time efficiency. DINO consists of C augmented by several high level parallel constructs that are intended to allow the parallel program to conform to the way an algorithm designer naturally thinks about parallel algorithms. The key constructs are the ability to declare a virtual parallel computer that is best suited to the parallel computation, the ability to map distributed data structures onto this virtual machine, and the ability to define procedures that will run on each processor of the virtual machine concurrently. Most of the remaining details of distributed parallel computation, including process management and interprocessor communication, result implicitly from these high level constructs and are handled automatically by the compiler.

Section 2 contains a short description of the DINO language, followed by a methodology for writing DINO programs, and two examples. The definitions of all DINO constructs are in Section 3. Section 4 has a number of examples, with a features index. Section 5 contains information about how to setup DINO and how to use the compiler itself. Section 6 contains a list of known compiler bugs, plus hints on run-time errors and performance. Information on how to install DINO is contained in Section 7. Finally, Appendix A contains a complete EBNF specification for the DINO extensions to the C language.

This manual is probably unlike any manual you have used before. In particular, Section 2 gives a conceptual overview of how a programmer should approach writting DINO programs, along with two examples of how to use this approach. Section 4 gives a large number of example programs that illustrate various DINO features. We suggest that new users should probably read Sections 2 and 5 in order to get started. Users can then proceed to the examples in Section 4, or read the reference material in Section 3.

2 Basic DINO

2.1 What DINO Is

DINO is a language for writing parallel numerical programs for distributed memory multiprocessors. By a distributed memory multiprocessor we mean a computer with multiple, independent processors, each with its own memory, but with no shared memory or shared address space, so that communication between processors is accomplished by passing messages. Examples include MIMD hypercubes, and networks of computers used as multiprocessors.

It is generally harder to design an algorithm for a multiprocessor than for a serial machine, because the algorithm designer has more tasks to accomplish. First, the algorithm designer must divide the desired computation among the available processors. Second, the algorithm designer must decide how these processes will synchronize. In addition, on a distributed memory multiprocessor, the algorithm designer must consider how data should be distributed among the processors, and how the processors should communicate any shared information.

The goal of DINO is to make programming distributed memory parallel numerical algorithms as easy as possible, without hindering efficiency. We believe that the key to this goal is raising the task of specifying algorithm and data decomposition, interprocess communication, and process management to a higher level of abstraction than is provided in many current message passing systems. DINO accomplishes this by providing high level parallel constructs that conform to the way the algorithm designer naturally thinks about the parallel algorithm. This, in turn, transfers many of the low-level details associated with distributed parallel computation to the compiler. In particular, details regarding message passing, process management, and synchronization are no longer necessary in the code that the programmer writes, and associated efficiency considerations are addressed by the compiler.

This high level approach to distributed numerical computation is feasible because so many numerical algorithms are highly structured. The major data structures in these algorithms are usually arrays. In addition, the algorithms usually exhibit data parallelism, where at each stage of the computation, parallelism is achieved by dividing the data structures into pieces, and performing similar or identical computations on each piece concurrently. DINO is mainly intended to support such data parallel computation, although DINO also provides some support for functional parallelism.

The basic approach taken in DINO is to provide a top down description of the distributed parallel algorithm. The programmer first defines a virtual parallel machine that best fits the major data structures and communication patterns of the algorithm. Next, the programmer specifies the way that these major data structures will be distributed, and possibly replicated, among the virtual processors. Finally, the programmer provides procedures that will run on each virtual processor concurrently. Thus the basic model of parallelism is Single Program Multiple Data, although far more complexity is possible. Most of the remaining details of parallel computation and communication are handled implicitly by the compiler. A key component in

making this approach viable has been the development of a rich mechanism for specifying the mappings of data structures to virtual machines, along with the efficient implementation of these mappings and the resultant communication patterns in the compiler.

Sequential code in a DINO program is written in standard C; DINO is a superset of C. Only the parallel constructs in DINO, and a few related additions, are new.

2.2 Basic Programming Model

Rather than just jumping in and confusing yourself with all the syntax of DINO, we believe it is more useful to start by looking at a conceptual overview of how DINO programs are written. We have followed this by two examples which show how to proceed from algorithm to finished code. The first of these examples is extremely simple, the second more complex.

We urge you to read the overview section, and at least the first example. This should give you enough of the flavor of DINO that you will understand the formal language definition and examples (in Sections 3 and 4).

2.2.1 DINO Overview

When writing programs in DINO, we have found it useful to proceed in a specific sequence of conceptual "steps." Our particular way of looking at the process of solving a problem in DINO helps us to structure the solution so it best fits the DINO paradigm. We suggest that you try this method of organizing the process of turning your solutions into DINO code at least long enough to get an idea of how we feel you should think about parallel numerical problems in DINO. There are five specific "steps":

- Step 0 — Begin with a good understanding of the problem and a general approach to its solution.

  In order to use our method, you must begin with a good understanding of your problem and a general approach to its solution, including what you believe the main data structures should look like and an idea of how work can be divided among a number of processors.

- Step 1 — Choose a *structure of environments* appropriate to the problem.

  In DINO, environments are virtual processors. In the current version of DINO, there should be only one virtual processor for each real processor on the target machine, due to memory and speed limitations. A group of environments that is used for a particular computational task is structured (a vector, 2D array, etc.) to make it easier to visualize how the data and the work is divided up and to make it easier to name particular environments.

You choose the particular structure of environments by having some idea of how your data structures will be broken up. If, for example, your major data structure is a 2D matrix and you want to send each row to an environment, choose a vector of environments. If your major data structure is a 2D matrix and you want to send each element to an environment, choose a 2D array of environments. Obviously, if you have more than one data structure, the environment structure must be the same for each or you need to clarify how your solution will work.

- Step 2 — Determine how the principal data structures should be distributed among the environments in the structure you have chosen (specify the *mappings*).

  For each of your data structures, you should specify how it is mapped onto the structure of environments you have decided to use. Conceptually, this task can be divided into three parts:

  - Determine the basic *partitioning*.

    Most of your major data structures will be broken up and distributed among the environments in the structure of environments that you specified. For example, if you have an N by N array named A to be distributed to a structure of environments named "node" which is a vector of size N you might want to think of putting one row on each environment so that $A[0][]$ is on node[0]. It is useful to think of node[0] as the owner of $A[0][]$ or of node[0] as being the home of $A[0][]$.

  - Determine any *replication*.

    In some cases, you will want to have the same data structure copied (replicated) on all environments in a structure of environments. You will want to do these when every environment needs access to the same values. While you can think of this as "partitioning" the data structure so that the whole thing goes to the first environment, then placing complete "copies" on all the other environments (see the next point), it is confusing to view it this way (this is how it is actually implemented). Instead, think of these data structures as simply replicated everywhere.

  - Determine where any *copies* (used for communication) go.

    Finally, you may want to determine that copies of parts of the data structure should be on environments other than the "home" environment for that part of your data structure. These copies will be used to indicate that you will communicate values for this data from the "home" environments to the environments where the copies are. The environments in which this copied data is placed are referred to as "copy" environments. For example, if you have broken a 2D array up by columns, each environment might need the values from the neighboring columns to do its computations. In this

case, you would want a copy of each column to be on the environments to the "right" and the "left" of the "home" environment for that column (except for the end columns).

The compiler uses this copy information to simplify communications. When you tell DINO to "send" the "home" column, it knows to send it to the environments where copies exist. When you tell DINO to "receive" a "copy" column, it knows what environment to receive it from. (The compiler also creates permanent storage for any "copy" data on an environment, a fact you may find important if you have very large data structures.)

The mechanics of specifying this to DINO are usually simple. There are a fair number of predefined mappings in the include file "dino.h". For example, **Block** puts each element of a vector on an environment in a vector of environments (assuming the number of elements equals the number of environments). The mapping function **all** replicates a data structure on each environment. **Block-Overlap** puts each element of a vector on an environment in a vector of environments and a copy of each element on the two neighboring environments (except for the edges). You simply use the one of these that fits your situation. If you have to, you can build your own mappings (information on this process can be found in Section 3.7.2).

- Step 3 — Write a *composite procedure* to implement computation in the structure of environments.

Determine what computations will take place in an environment. The code should be written for one environment. This means that you must have some way of identifying the particular environment your code is executing on. That is the function of the identifiers in the declarations for each of the environment indices, referred to in this manual as *environment index identifiers*. For example, if the environment declaration is "`environment node[10:id] { ··· }`", "id" is a constant that contains the number of the current environment.

You can conceptually break writing a composite procedure down into three "steps":

- Write the code for the *computation*.

  Write the code for one environment assuming that all the necessary data is available there.

- Insert any necessary *receives*.

  If during the computation, you will need updated values in the data that is "copy" data on this environment, put in receives for that data. A good rule of thumb is to put the receives in as late in the computation as possible. A second rule of thumb is that data communicated between

environments should be sent and received in as large blocks as is possible. Obviously, these two rules sometimes conflict, and you must decided what the best trade-off is.

- Insert the corresponding *sends*.

  Finally, you have to put in sends corresponding to the receives you inserted in the last step. Again, a good rule of thumb is to put the sends in as early in the computation as possible, and to send in blocks where possible.

When doing these steps, we've found that it's best to write code for the "middle" first. Most of the complications in DINOcode occur at the "edges," so we suggest that you write code for the middle first. Then modify it to take into account the extra complications introduced by the edge conditions.

We mean "middle" in two senses — spatially and temporally. Spatially means the middle of your distributed data structures. Often, more complicated code has to be written for the edges, for one of the following reasons:

1. The actual edge of the global data structure may have different communication patterns than the central portions. Your code may need to reflect this.
2. If you block your distributed data (put more than one element on an axis in an environment,) then then edge of the block on an environment may have to be treated differently.

Temporally means that the first and last iterations of a loop are often different than the others. For example, DINO initializes copy data when parameters are distributed. This may mean that a first send/receive pair is unnecessary.

- Step 4 — Finally, write a *host environment*, containing a *main() function*, to implement computation on the host environment.

  Write the code that does any initialization, calls the composite procedure(s), and processes the results from the composite procedure(s). All the data used by composite procedures is passed in as parameters to these procedures. DINO will automatically distribute this data in the manner specified by your mappings.

2.2.2 Walkthrough

To illustrate writing a DINO program, we offer two examples. The first is a very simple program intended to show the basic application of the principles outlined in the overview. The second is a somewhat more complex example intended to show some of the subtleties involved in using DINO. We will walk through the steps outlined above for each example to illustrate the process.

These examples illustrate the thought process that a programmer should go through when writting a DINO program. To get much out of them, they have to be read carefully and not just skimmed. We recommend setting aside some time to work through them carefully.

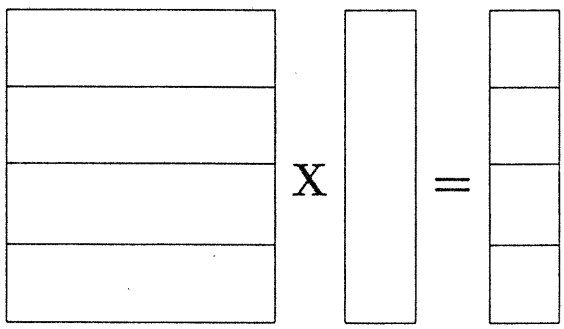
2.2.3 Matrix-Vector Multiply

This example takes a very simple problem, a matrix-vector multiplication where the number of rows in the matrix is the same as the number of processors available, and shows how to write a DINO program to solve this problem.

- Step 0 — Begin with a good understanding of the problem and a general approach to its solution.

  The algorithm is almost trivial — each element of the solution vector is obtained by taking the dot product of the corresponding row of the multiplicand matrix with the multiplicand vector. To parallelize this problem, we will choose to send each row of the multiplicand matrix to a processor, a copy of the multiplicand vector to each processor, and collect each element of the answer vector back from the processors.

  If we draw a picture of our algorithm (we usually resort to drawing pictures to understand all but the simplest algorithms), we get something like this:



- Step 1 — Choose a *structure of environments* appropriate to the problem.

  Given the way we have decided to break up the problem, a vector of environments seems appropriate. We will use a structure of environments the size of the machine (which we assume to be 16). So our code initially looks like this:

```
environment node[N]
{
}
```

  To make things easier to change, we will use a macro definition - N, which will represent both the size of the machine and the size of the array and vectors (16).

- Step 2 — Determine how the principal data structures should be distributed among the environments in the structure you have chosen (specify the *mappings*.)

  - Determine the basic *partitioning*.

    Lets call the multiplicand matrix A, the multiplicand vector B, and the solution vector C. We block A by rows and C by elements. (We don't partition B at all; see the next point.)

  - Determine any *replication*.

    We want a complete copy of B on each environment, so we replicate it.

  - Determine where any *copies* (used for communication) go.

    There is no inter-processor communication in this program (except, of course, for parameter distribution and collection, but that is totally automatic.) Thus, no copies are needed.

  The mappings that give the partitioning we want are **BlockRow**, **all**, and **Block**. (We looked at the predefined DINO mappings to find this — see Section 3.4.2.) Our declarations therefore look like:

```
float distributed A[N][N] map BlockRow;
float distributed B[N] map all;
float distributed C[N] map Block;
```

- Step 3 — Write a *composite procedure* to implement computation in the structure of environments.

  - Write the code for the *computation*.

    This is almost trivial. We need a single loop to do the dot product. The only complication is that we must know which environment we are on in order to know which row of A and element of C to use in the computation. There is a special constant we can declare that will have the value of our index in it, which is known as an environment index identifier (see Section 3.2). This makes our environment declaration look like this:

```
environment node[N:id]
   {
   }
```

    So here is our code on the node environments:

```
environment node[N:id]
   {
    composite mult(in A, in B, out C)
        float distributed A[N][N] map BlockRow;
        float distributed B[N] map all;
        float distributed C[N] map Block;
       {
        int I;

        C[id] = 0;
        for (I = 0; I < N; I++)
            C[id] += A[id][I] * B[I];
       }
   }
```

Notice that the distributed data, particularly the arrays A and C, is referenced using its global name, even though only part of either of these data structures is actually on each environment. DINO automatically handles this process. However, if you try to reference some part of a distributed data structure that is not on an environment, DINO isn't so nice. At this time, DINO does not check for these kind of mistakes (it would require run-time checking - see Section 6.2). At best, your program will crash. At worst, you will get incorrect results. So be careful of this.

- Insert any necessary *receives*.

  There aren't any.

- Insert the corresponding *sends*.

  There aren't any of these either.

- Step 4 — Finally, write a *host environment*, containing a *main() function* to implement computation on the host environment.

  This turns out to be very straight forward. All of this is ordinary C code except the composite procedure call

```
mult(a[][], b[], c[])#;
```

  Note that since we are sending entire arrays (and not just pointers to them), we use ranges (see section 3.6).

With this last part, our program is complete. Here is the DINO code for the whole thing:

```
/*
 *      This does a matrix-vector multiply.  The
 *      limitations are that the matrix and vector
 *      sizes must be equal to the number of processors.
 *
 *      The basic algorithm is that a row of the
 *      matrix is sent to each processor and a copy
 *      of the vector is sent to each processor.
 *      Each processor does a dot product with its
 *      data and returns one element of the result.
 *
 */

#define N 16

#include "dino.h"

environment node[N:id]
   {
    composite mult(in A, in B, out C)
        float distributed A[N][N] map BlockRow;
        float distributed B[N] map all;
        float distributed C[N] map Block;
       {
        int I;

        C[id] = 0;
        for (I = 0; I < N; I++)
            C[id] += A[id][I] * B[I];
       }
   }

environment host
   {
    main()
       {
        int I, J;
        float a[N][N], b[N], c[N];

            /* Initialize the array and
                     the multiplicand vector to something. */

        for (I = 0; I < N; I++)
           {
            b[I] = 0.1;
```

```
            for (J = 0; J < N; J++)
                a[I][J] = (I + J) / 10.0;
        }

        /* Call the composite procedure. */

    (void) printf("\nStarting the computation . . . .\n\n");
    mult(a[][], b[], c[])#;

        /* Print out the results. */

    (void) printf("Results:\n");
    for (I = 0; I < N; I++)
        {
        if (I % 8 == 0)
            (void) printf("\n");
        (void) printf("   %6.2f", c[I]);
        }
    (void) printf("\n");
    }
}
```

2.2.4 Matrix-Matrix Multiply

This example takes a somewhat more complex problem than the last one, namely a matrix-matrix multiply in which the sizes of the data structures can be a multiple of the number of processors available, and shows how to write the DINO program to compute the result. In addition to illustrating how to cope with the increased difficulty because (1) the basic algorithm is somewhat more complex than the one in the last example and (2) we are allowing multiple rows or columns in each processor, this problems illustrates how to cope with the difficulty caused by a problem that doesn't fit the DINO paradigm as well as other problems.

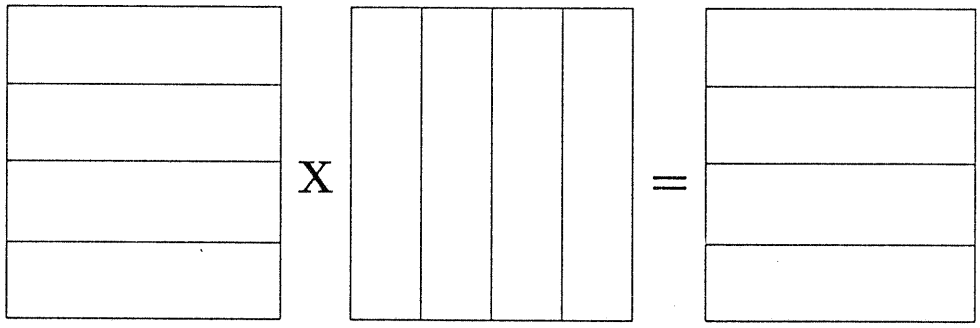
- Step 0 — Begin with a good understanding of the problem and a general approach to its solution.

  The general algorithm for a matrix-matrix multiply is simple – each element of the solution matrix is obtained by taking the dot product of the corresponding row and column of the two multiplicand matrices. Our concern here is to decide how to parallelize this problem. We start off with three square matrices – two multiplicands and the solution. Initially, we will try an obvious approach: block the first multiplicand by rows and send one block to each processor. If we block the answer in the same manner, we can just collect the answer at the end of the computation.

The second multiplicand is more of a problem. Initially, lets explore what happens if we block it by columns, and send one block to each processor. This allows us to compute part of the answer that is on each processor, but not all of it. A little thought and we realize that we will need all the columns to compute our part of the answer. Let us assume for now that we will block the second multiplicand by columns and use DINO's communications to make the blocks available to all processors.

Now we might notice that although we are supposed to be thinking about the parallel algorithm in general, we are influenced by the physical constraints of the machine. As we do more of these problems, we will learn that we are also influenced by the nature of DINO. In reality, the process is an iterative one; we may have to go through the four "steps" several times to obtain a good understanding of a particular problem.

If we draw a picture of our algorithm we get something like:



- Step 1 — Choose a *structure of environments* appropriate to the problem.

Given the way we have decided to break up our problem, a vector of environments seems appropriate. For efficiency, we will use a structure of environments the size of the machine (which we assume to be 16). So our code initially looks like this:

```
environment node[P]
{
}
```

To make things easier to change, we will use macro definitions — P for the size of the machine (16), and N for the size of the arrays (256).

If we think about this decision a little, it might seem to make more sense to use a torus instead of a vector. That way when we go to "shift" each block of columns to the left, we would not have to treat the ends of the vector specially. However, this particular feature is not implemented in this version of DINO.

- Step 2 — Determine how the principal data structures should be distributed among the environments in the structure you have chosen (specify the *mappings*.)

– Determine the basic *partitioning*.

This seems easy enough. Lets call our multiplicand matrices A and B and our solution matrix C. We block A and C by rows and B by columns.

– Determine any *replication*.

There isn't any.

– Determine where any *copies* (used for communication) go.

This may cause us problems. What we really want to do is to take each block of columns and shift it one environment to the left for each iteration of the computation until every block has been to all of the environments. But this is a different sort of communication than the DINO paradigm is designed to support (see Section 3.7.1); DINO assumes that a particular piece of data has one "home" environment and other environments may have "copies" of the data.

How could we get around this? Here are two ideas that might come to mind: (1) place a copy of each block on every processor, or (2) place a copy of each block on the processor to its left and use that to shift the blocks. Let us look at each of these ideas.

The first of these ideas is quite easy to implement and would be easy to write code for. But it has one serious drawback. The compiler must create storage for the whole B matrix on each environment if we do this. Since, in this example, we are concerned about storage, this is not acceptable. (Note, that this is equivalent to replicating B across all the environments.)

The second idea at first seems somewhat more workable. We put a copy of each block on the environment to the left of it. Then when we are ready to communicate, we assign the value of the copy data to the home data and send it. (We can do this with a simple DINO statement like

```
B[][<my_first, my_last>]# =
                B[][<left_first, left_last>];
```

assuming "`my_first`", "`my_last`", "`left_first`", and "`left_last`" are defined as we do later in this example to give the first and last columns in the current block and in the block to the left of us.)

However, this leads to several difficulties that causes us to reject this approach: (1) we would have to be very careful how we reference the elements of B since we are shifting the columns around but DINO is not aware of this so the global names we would ordinarily use will not be correct (see more discussion on this point later); (2) we would have to handle the communications at the ends of the environment structure specially since we are unable to declare it to be a torus; (3) this would complicate the communications structure since there are two blocks already on each environment when

we start up – no communication needs to be done until the third iteration; and (4) we would double the storage necessary for B on each environment.

Well, when all else fails, there is a form of send and receive that puts everything in the hands of the programmer. We will fall back on that. We will tell the compiler that we are just going to partition B and then handle the communications ourselves. That means we have to be more careful that we get everything right. A future version of DINO may handle this particular problem better.

The mappings that give the partitioning we want are **BlockRow** and **BlockCol**. (We looked at the predefined DINO mappings to find this information; see Section 3.4.2.) Our declarations therefore look like:

```
float distributed A[N][N] map BlockRow;
float distributed B[N][N] map BlockCol;
float distributed C[N][N] map BlockRow;
```

- Step 3 — Write a *composite procedure* to implement computation in the structure of environments.

  - Write the code for the *computation.*

    Initially, this seems pretty straight forward. We have four loops: (1) the outermost keeps track of which column block of B we have, (2) the next one is the row of A that we are working on, (3) then there is which column within the particular column block of B we have, and (4) finally there is a loop for doing the dot product itself. There are several points to note:

    * Right away we realize we need some way of identifying which environment we are on. Once again, we use an *environment index identifier.* This makes our environment declaration look like this:

      ```
      environment node[P:id]
      {
      }
      ```

    * The non-standard communications we are using will cause us problems in referencing data if we are not careful. Normally, distributed data is referenced by its global name. So, for example, on `node[0]` our first row is "`0`" and our last row is "`15`". Of course, we have to write an expression that is good on any environment so we would write the first row as "`id * N/P`" and the last row as "`(id + 1) * N/P - 1`".

However, we have done something funny with the columns in B. We initially distributed them with a standard mapping, but then we "shift" them around "by hand" instead of having DINO do it. Since the compiler doesn't know what we are doing with the communications, it will no longer keep track of the correct global names of the columns of B once we do the first shift. We have to do that. So elements of B will have to be referenced differently than elements of A or C.

What we need to realize is that the current block of B will be addressed in exactly the same manner on every iteration. This is because the compiler doesn't think that anything has changed. So, for example, when B is first distributed, the first column on node[1] is "`id * N/P`" ("`16`") and the last column on node[1] is "`(id + 1) * N/P - 1`" ("`31`") – the correct global names. After the first shift, the first column is still "`id * N/P`" and the last column is still "`(id + 1) * N/P - 1`", even though the correct global names are now "`32`" and "`47`".

* For two reasons, it will be better if we define some intermediate values at the top of our program and compute them there: (1) it moves invariant expressions out of the loops (we know, good compilers are supposed to do this; but on the machines we've seen they don't), and (2) it makes the code much easier to read. So, for example, we define:

```
my_first = id * N/P;
my_last = (id + 1) * N/P - 1;
```

to take care of identifying the first and last row (column) on our environment. (We use the last instead of the last + 1 because we need that value in some range specifications.) For this problem, in addition to the two values above, we defined "`left_first`" (the first column on the environment to the left of us), "`left_last`" (the last column on the environment to the left of us), "`left`" (the index of the environment to the left of us), and "`right`" (the index of the environment to the right of us). These last four we will use for communications.

Notice that it is easier to code some of these calculations if we first write them for the "middle," and then look at the edge cases. For example, if we just look at the middle, then we would define `left` as "`left = id - 1;`". Then we look to see if there is anything different at either end. It turns out that at node[0], we want `left` to be equal to `P - 1`, not `-1`, so the definition of `left` becomes "`left = (id == 0) ? (P - 1) : (id - 1);`".

So here is our code on the node environments (except for the communications):

```
environment node[P:id]
   {
```

```
composite mult(in A, in B, out C)
    float distributed A[N][N] map BlockRow;
    float distributed B[N][N] map BlockCol;
    float distributed C[N][N] map BlockRow;
   {
    int I, J, K, L;
    int my_first, my_last, right,
                left_first, left_last, left;

        /* Compute the environment indices of the
                    nodes on the right and left of me. */

    right = (id == P - 1) ? (0) : (id + 1);
    left = (id == 0) ? (P - 1) : (id - 1);

        /* Compute the starting and stopping
                     indices of the data in my block. */

    my_first = id * N/P;
    my_last = (id + 1) * N/P - 1;

        /* Compute the starting and stopping indices
                        of the data in the block of
                        the node to the left of me. */

    left_first =
        (id == 0) ? ((P - 1) * N/P) : ((id - 1) * N/P);
    left_last = (id == 0) ? (N - 1) : (id * N/P - 1);

    for (I = 0; I < P; I++)
       {
            /* Compute the values in the block of
               C[][] that I currently have data for. */

        for (J = my_first; J < my_last + 1; J++)
            for (K = 0; K < N/P; K++)
               {
                C[J][K + ((id + I) * N/P) % N] = 0;
                for (L = 0; L < N; L++)
                    C[J][K + ((id + I) * N/P) % N] +=
                           A[J][L] * B[L][K + my_first];
               }
       }
   }
}
```

– Insert any necessary *receives*.

We want to put the receive as late as possible in the computation. We could do this by putting it just before the second loop (the "J" loop), but then we would have to put an if statement around it to insure that it only received from the second iteration on (since DINO initially distributes the data on the composite procedure call. Instead we elect to put it at the end of the "J" loop. Our receive will look like this:

```
B[][<my_first, my_last>] =
        B[][<my_first, my_last>]# {node[right]};
```

We should note that we have to specify everything explicitly. In DINO, if we are using the default paradigm, the compiler does a lot of things for us. The statement "`B[][<my_first, my_last>]#;`" would cause the compiler to look for a piece of data with that name (in this context, "name" includes the bounds) and which came from the "home" environment of that particular piece of data. When it found it, it would automatically store it in the correct location. When we go outside the default paradigm (by specifying the destination or source of the message), the programmer must specify it all explicitly. "All" includes:

* Where the incoming data will be placed (the specification on the left hand side of the assignment).

* The name of the incoming data (the part of the right hand side of the assignment that has the data name in it).

* The environment that the data is coming from.

– Insert the corresponding *sends*.

We have two concerns here: (1) we want to assure that there are sends corresponding to every receive in the program (otherwise we either generate extra messages which can cause problems in a large program or we never get a message we are looking for and the program locks up), and (2) we want to put the send as early as possible in the computation.

The first of these can be complex, because there is not always a one-to-one relationship between the number of sends and the number of receives (in DINO you can send a range of data and the compiler will actually do a send to more than one environment; the same is true for receives). However, that is not the case here. So we just have to be sure there is a send for each receive. We insure that if the send and receive are in the same loop.

The second of these concerns is satisfied if we place the send at the beginning of the "J" loop. Then our send looks like this:

```
B[][<left_first, left_last>]#{node[left]} =
```

```
                                    B[][<my_first, my_last>];
```

Some points to notice on this:

* We have to explicitly include the same things in this kind of send that we did in the corresponding receive.

* The name of the data must be the same in the send as the receive. Because the expression for the range endpoints uses "id" (which is different on each environment), the expressions have to be different for the send and receive to give us the same name. This accounts for using "left_first" and "left_last" to send the data but "my_first" and "my_last" to receive it.

The computation loop (with all communication statements inserted) looks like:

```
for (I = 0; I < P; I++)
   {
        /* First, send out the data that the node on
           the right of me will need next iteration. */

    B[][<left_first, left_last>]#{node[left]} =
                                B[][<my_first, my_last>];

        /* Then, compute the values in the block of
               C[][] that I currently have data for. */

    for (J = my_first; J < my_last + 1; J++)
        for (K = 0; K < N/P; K++)
           {
            C[J][K + ((id + I) * N/P) % N] = 0;
            for (L = 0; L < N; L++)
                C[J][K + ((id + I) * N/P) % N] +=
                            A[J][L] * B[L][K + my_first];
           }

       /* Finally, receive the data from the node on
          the right that I need for the next iteration. */

    B[][<my_first, my_last>] =
                B[][<my_first, my_last>]#{node[right]};
  }
```

The astute observer will notice that there is an extra send/receive pair in this loop. We do not need to do the send/receive on the last iteration since there will not be a next iteration to use the new data. However, it would have required

two additional if statements to remove this, which for large matrices would slow the program down more than the additional send/receive pair.

- Step 4 — Finally, write a *host environment*, containing a *main() function* to implement computation on the host environment.

  This turns out to be very straight forward. All of this is ordinary C code except the composite procedure call

  ```
  mult(a[][], b[][], c[][])#;
  ```

  Note that since we are sending entire arrays (and not just pointers to them), we use ranges (see Section 3.6).

With this final step, our program is done. Here is the DINO code for the complete program:

```
/*
 *      This program does a matrix-matrix multiply.  The
 *      only limitations are that the matrices must be
 *      square and that their size must be a multiple of
 *      the number of processors.
 *
 *      The basic algorithm is to block the first matrix
 *      by rows and send a block to each processor, and
 *      block the second matrix by columns and send a block
 *      to each processor.  Each processor computes the
 *      results for that sub-matrix for which it has data
 *      and then the blocks of columns are all shifted to
 *      the right one processor.
 *
 */

#define N 256
#define P 16

#include "dino.h"

environment node[P:id]
   {
    composite mult(in A, in B, out C)
        float distributed A[N][N] map BlockRow;
        float distributed B[N][N] map BlockCol;
        float distributed C[N][N] map BlockRow;
       {
```

```
int I, J, K, L;
int my_first, my_last, right,
            left_first, left_last, left;

    /* Compute the environment indices of the
               nodes on the right and left of me. */

right = (id == P - 1) ? (0) : (id + 1);
left = (id == 0) ? (P - 1) : (id - 1);

    /* Compute the starting and stopping
                 indices of the data in my block. */

my_first = id * N/P;
my_last = (id + 1) * N/P - 1;

    /* Compute the starting and stopping indices
                    of the data in the block of
                    the node to the left of me. */

left_first =
    (id == 0) ? ((P - 1) * N/P) : ((id - 1) * N/P);
left_last = (id == 0) ? (N - 1) : (id * N/P - 1);

for (I = 0; I < P; I++)
   {
        /* First, send out the data that the node on
           the right of me will need next iteration. */

    B[][<left_first, left_last>]#{node[left]} =
                              B[][<my_first, my_last>];

        /* Then, compute the values in the block of
               C[][] that I currently have data for. */

    for (J = my_first; J < my_last + 1; J++)
        for (K = 0; K < N/P; K++)
           {
            C[J][K + ((id + I) * N/P) % N] = 0;
            for (L = 0; L < N; L++)
                C[J][K + ((id + I) * N/P) % N] +=
                            A[J][L] * B[L][K + my_first];
           }

       /* Finally, receive the data from the node on
```

```
                the right that I need for the next iteration. */

            B[][<my_first, my_last>] =
                        B[][<my_first, my_last>]#{node[right]};
          }
      }
   }

environment host
   {
    main()
       {
        int I, J;
        float a[N][N], b[N][N], c[N][N];

            /* Initialize the two
                        multiplicand arrays to something. */

        for (I = 0; I < N; I++)
            for (J = 0; J < N; J++)
               {
                a[I][J] = 0.1;
                b[I][J] = (I + J) / 10.0;
               }

            /* Call the composite procedure. */

        (void) printf("\nStarting the computation . . . .\n\n");
        mult(a[][], b[][], c[][])#;

            /* Print out the results -- because the data
                we use generates the same answer for each element
                in any column, we only print out the first row. */

        (void) printf("Results:\n");
        for (I = 0; I < N; I++)
           {
            if (I % 8 == 0)
                (void) printf("\n");
            (void) printf("   %6.2f", c[0][I]);
           }
        (void) printf("\n");
       }
   }
```

3 DINO — The Language

This section contains the formal definition of DINO. It is broken down into the major categories of DINO constructs: program structure, environments, composite procedures, distributed data, reduction functions, ranges, and user defined mappings. If the meanings of the EBNF specifications used here are unfamiliar, an explanation can be found in Appendix A.

3.1 Program Structure

```
PROGRAM ::= (ENVIRONMENT or MAPPING_FUNCTION or DATA_DEFINITION )+.
```

A program consists of one or more environment declarations, zero or more mapping function declarations, and zero or more (distributed) data declarations. One of the environment declarations must be defined as a scalar environment with the name "host". This environment must contain a function, named "main", where execution starts. The remaining environment declarations generally define structures of environments that contain composite procedures which are invoked from the host.

Distributed data structures that are declared at the program level are mapped to one or more environment structures in the program as specified by their mapping functions. Ordinary data declarations can also be made at this level. If they are, independent copies are instantiated on every environment in the program. Mapping functions that are declared at the program level are accessible to all parts of the program. The normal C rule that requires definition before use is followed throughout DINO.

3.2 Environments

```
ENVIRONMENT ::=
    'environment' IDENTIFIER DIMENSION*
    '{' EXTERNAL_DEFINITION+ '}'.

DIMENSION ::= '[' EXPRESSION [ ':' IDENTIFIER ] ']'.

EXTERNAL_DEFINITION ::=
    FUNCTION_DEFINITION | DATA_DEFINITION | MAPPING_FUNCTION.
```

A structure of environments provides a virtual parallel machine, and a mechanism for building parallel algorithms in a top down fashion. An environment may contain composite procedures, standard C functions, distributed data declarations, standard C data declarations, and mapping functions. Each environment within a given structure contains the same procedures, functions, and C data declarations, and shares the same distributed data declarations.

A structure of environments may be any single or multiple dimensional array. For example, the declaration

```
environment grid [N:xid] [M:yid]  {  . . .  }
```

specifies an N by M array of virtual processors named "grid". In order to give each environment within a structure an identity that can be used in calculations, the programmer can declare constants that will contain the subscripts identifying that particular environment. These are referred to as *environment index identifiers*. In the above example, these are "xid" and "yid". Most often these are used to refer to that environment's portion of a distributed data structure, for example a matrix element $A[xid][yid]$.

A DINO program may contain any number of environment declarations. Typically only one, the host, is scalar, and the remainder are arrays. Multiple structures of environments may be used when successive phases of a computation map naturally onto different parallel machines (in this case, generally only the environments in one structure become active at once) or in functionally parallel programs (in this case the environments in multiple structures become active at the same time; see Section 3.3).

An environment can contain an arbitrary number of composite procedures and standard C functions. However only one task, or thread of control, may be active in an environment at a time. We refer to this as *single task semantics*. In addition, the only way to start a task executing in an environment other than the host is to call a composite procedure which resides in that environment. These two facts imply that there can not be any composite procedures in the host environment because procedure "main" is always active.

A procedure executing in one environment can not directly reference data in another environment. Procedures in two environments can exchange information only if there is cooperation between the procedures, namely a remote read and write of a distributed variable, (Section 3.4.4) or the use of a reduction operator on a distributed variable (Section 3.5). These semantics and the single task semantics described above imply that there is is no hidden shared-memory emulation in DINO and also results in DINO programs being deterministic unless asynchronous distributed variables or explicit environment sets are used (see Section 3.4).

Structures of environments are treated as blocks with respect to scope. Ordinary data that is declared within an environment structure is copied to every environment in that structure. Copies of ordinary data on separate environments have no relation to each other and can only be accessed within their own environments. Distributed data is treated as described in Section 3.4. The one exception to this scoping rule is composite procedures, which are always visible in the topmost scope. Thus composite procedures may be called from any environment except their own, i.e. recursive calls are not allowed.

The DINO compiler creates one process for every environment in each structure and maps these processes statically onto the actual parallel machine. This is

done in a manner that attempts to optimize the distance on the parallel computer between contiguously indexed environments in each structure. The mapping also attempts to optimize load balance as follows: if any structure of environments contains as many or more environments than there are actual processors, then the environments are partitioned evenly over the entire parallel machine. When a composite procedure is invoked in a structure with more environments than processors, multiprogramming occurs. This style of programming does not lead to optimal efficiency and may not run due to memory limitations (see Section 6.3). If there are two or more structures of one dimensional environments that together contain no more environments than the total number of processors, then each environment is assigned to a unique processor.

3.3 Composite Procedures

```
declaration:

FUNCTION_DEFINITION ::=
    'composite' IDENTIFIER '(' [COMP_PARAMETER_LIST] ')'
    FUNCTION_BODY.

COMP_PARAMETER_LIST ::= ( ['in' | 'out'] IDENTIFIER ) || ','.

call:

STATEMENT ::=
    IDENTIFIER '(' [EXPRESSION_LIST] ')' '#' [ '{' ENV_EXP '}' ]
    [ '::' STATEMENT ].

ENV_EXP ::= EXPRESSION.
```

A composite procedure is a procedure which runs on each environment of a structure of environments (or a subset thereof) and thereby implements concurrency. A composite procedure consists of multiple copies of the same procedure, one residing within each environment of a structure of environments. Calling the composite procedure invokes all of these procedures at the same time. Typically, each procedure works on a different part of some distributed data structure(s), resulting in a single program, multiple data (SPMD) form of parallelism. These distributed data structures may either be defined in the structure of environments or globally or may be parameters to the composite procedure. Each procedure may also contain standard local data. We will first describe the various parts of composite procedure declaration and invocation statements, and then the order of events that occur when a composite procedure is invoked.

The formal parameters of a composite procedure may be distributed variables or standard variables. Formal parameters that are distributed variables may be preceded by the keywords "in" (call-by-value), "out" (call-by-result), or no keyword

(call-by-value/result). Formal parameters that are standard variables must be preceded by "in" and are input parameters that are replicated in all the environments on which the composite procedure is called.

A composite procedure call has the same syntax as ordinary C functions, except that arrays or subsections of arrays (Section 3.6) and remote references to distributed variables can be used as arguments, and the parameter list is followed by a "#" sign. All actual parameters corresponding to result and value/result parameters must refer to variables. Unlike C, DINO checks the number and types of actual parameters against the formal parameters in a composite procedure call. Composite procedures do not return a value.

When a composite procedure is called, the invocation can be limited to a subset of the environments in the environment structure on which it is defined. This is accomplished using the optional ENV_EXP following the "#" sign, which returns a subset of environments on which the procedure is to be invoked (the "active-set"). The form of ENV_EXP is one or more environment names (or ranges — see Section 3.6) connected by set union or difference operators "+" and "-". If this expression is not given, then the composite procedure is invoked on all of the environments in the structure.

Functional parallelism is achieved in DINO by the utilization of the optional ":: STATEMENT" construct following a composite procedure call (this definition is recursive; see the EBNF specification in Appendix A). This STATEMENT is executed concurrently with the composite procedures specified by the call. It can be either another composite procedure call utilizing a different structure of environments (or a disjoint subset of the same structure), or a standard C statement that is executed on the host. The program blocks until both the composite procedure and the concurrent statement complete execution.

The keyword "caller", when used within a composite procedure, is the environment which invoked that composite procedure. This may be used when a composite procedure is executing concurrently with code on the invoking environment, as described in the previous paragraph, and wants to communicate with the procedure that invoked it via explicit reads and writes of global distributed data (see Section 3.4.4).

Now we describe in detail the events that occur when a composite procedure is invoked. First, the ENV_EXP expression is evaluated if it is given. This defines an active-set of environments as described above. Only procedures and environments in the active-set are used in the rest of the sequence. Second, all of the actual parameters that correspond to value and value/result formal parameters are evaluated. If any of these actual parameters contains a "#" operator then a remote read is executed to get the value (see Section 3.4.4). Third, the values of the actual parameters are assigned to the formal parameters and sent to the appropriate environments. For formal parameters that are not defined as "distributed", the actual parameter value is sent to each environment in the active-set. If the formal parameter is defined as "distributed" then the actual parameter value is distributed to each environment that it is

mapped to (but limited to the active-set). Fourth, the procedure is actually called at each environment in the active set. At the same time, the concurrent STATEMENT, if any, is executed in the calling environment (possibly resulting in one or more additional composite procedure calls on different environments). Fifth, after all these procedures/STATEMENTs have completed executing, the result and value/result parameters are returned. Each element of a result parameter is returned from exactly one environment, its "home" environment (see Section 3.4.3). If any of these actual parameters contains a "#" operator then a remote write is be executed. Finally, the calling environment continues execution after the concurrent STATEMENT.

3.4 Distributed Data

Distributed data is used to map data structures onto the structures of environments in a DINO program. It provides the mechanism for communication between environments, which is accomplished by using distributed variables as formal parameters to composite procedures, and by remote references to distributed variables. It also allows the programmer to maintain a global view of these data structures (in most situations; see Section 2.2.4) and joins with composite procedures to produce an SPMD model of computation.

Currently, the types of data structures that can be distributed are arrays of any dimension. DINO provides a rich mechanism for specifying the mappings of such data structures onto structures of environments, including many different one-to-one and one-to-many mappings, and also provides efficient implementation of the communication that can result through the use of these variables. The declaration of distributed data is discussed in Section 3.4.1. Section 3.4.2 discusses the predefined mapping functions in the include file "dino.h". In Section 3.4.3, the concepts of home and copy data are discussed. The use of distributed data, primarily remote references for communication, is discussed in Section 3.4.4.

3.4.1 Declaring Distributed Data

```
distributed data declaration:

DECLARATOR ::=
    [ 'asynch' ] 'distributed' DECLARATOR
    ( '[' CONSTANT_EXPRESSION ']' )+ MAPPING.

MAPPING ::=
    'map' ('all' | IDENTIFIER | (( IDENTIFIER IDENTIFIER ) || 'map')).
```

Distributed data declarations follow standard C syntax, except that the keyword "distributed" precedes the name of the distributed variable, and the keyword "map" followed by the name of a mapping function follows it. An example is

```
float distributed A[N][N] map BlockRow;
```

Distributed data may be declared at any level of a DINO program, and follows standard C scoping rules. The `MAPPING` portion of a distributed data declaration that is not contained in an environment declaration must use the "`IDENTIFIER IDENTIFIER || 'map'`" syntax, where the second identifier in each pair is an environment name:

```
float distributed A[N][N] map grid BlockBlock map node BlockRow
```

The mapping function(s) are either taken from DINO's library of predefined mapping functions, as is the case with **BlockRow** and **BlockBlock** above, or is defined by a mapping function definition statement (see Section 3.7.2).

3.4.2 Predefined Mappings

DINO provides a set of predefined mapping functions for mapping one and two dimensional (hereafter "1D" and "2D") data structures onto 1D structures of environments, and 2D data structures onto 2D structures of environments. The predefined 1D to 1D mappings are

Block, **Wrap**, and **BlockOverlap**.

They can be used to map any array of N elements onto an array of P environments, and work in a fairly obvious way. If N is a multiple of P, then **Block** maps the first N/P data elements onto the first environment, and so on; if $N < P$ then the first N environments contain one element each and the last $N - P$ contain none, while if $N > P$ but N is not a multiple of P, then the first N mod P environments contain an extra element. **Wrap** maps each element i ($i = 0, \cdots, N-1$) to environment [i mod P]. **BlockOverlap** does a **Block** mapping and in addition, maps the first and last elements of each block (except elements 0 and $N-1$) to the next lower and higher environment, respectively. This extra mapping is referred to as an *overlap* (One-sided overlaps and wider overlaps are also possible using user-defined mappings; see Section 3.7).

The predefined mappings for mapping 2D (M x N) data structures to 1D (P) environment structures are

BlockRow, **BlockCol**, **WrapRow**, **WrapCol**, **BlockRowOverlap**, and **BlockColOverlap**.

They work identically to the above 1D to 1D mappings except that rows or columns are used in the place of individual elements. For mapping 2D (M x N) data structures onto 2D (P x Q) environment structures, the predefined mapping functions are

BlockBlock, **FivePt**, and **NinePt**.

BlockBlock divides the M x N array into sub-arrays of size M/P x N/Q on each environment. **FivePt** is a **BlockBlock** mapping with overlaps in the four directions; in the case when $M = P$ and $N = Q$ it reduces to the standard five point star (element $[i][j]$ mapped on to environments $[i-1][j]$, $[i+1][j]$, $[i][j]$, $[i][j-1]$, and $[i][j+1]$, and visa versa.) **NinePt** is a **FivePt** mapping where in addition the corner elements are included in the overlap (in the $M = P$, $N = Q$ case these are $[i-1][j-1]$, $[i-1][j+1]$, $[i+1][j-1]$, and $[i+1][j+1]$.) Finally, the mapping keyword **all** maps all the elements of any dimensional data structure onto each environment of any dimensional structure of environments.

3.4.3 "Home" and "Copy" Data

Implicitly associated with each mapping function is the specification of one "home" environment, and zero or more "copy" environments, for each element of the distributed data structure. These constructs have an effect when distributed data structures are parameters to composite procedures (see Section 3.3) and in remote references to distributed data (Section 3.4.4). If an element of a distributed data structure is mapped to only one environment, this is its home. If it is mapped to multiple environments, this must be done using either overlaps or the "all" construct. With overlaps, the home environment is the environment that the data element would be mapped to if the overlap term was omitted, while the additional environments that the overlap term causes it to be mapped to are its copy environments. If an element is mapped using "all", then we arbitrarily consider the lowest subscripted environment to be its home and the remainder to be copy environments, but this distinction is generally unimportant since such variables are usually either input-only parameters, or uniformly shared variables internal to the composite procedure. The semantics of home and copy environments assure that DINO programs are deterministic in the default case (see Section 3.4.4).

3.4.4 Using Distributed Data

```
EXPRESSION ::= PRIMARY '#' [ '{' ENV_EXP [ 'from' EXPRESSION ] '}' ].

ENV_EXP ::= 'caller' | EXPRESSION.
```

There are three different ways to use a distributed variable. The first is as a formal parameter to a composite procedure call, as discussed in Section 3.3. The other two are by local and remote references (either reads or writes), appearing as expressions in any standard C statement.

A local reference of a distributed variable uses standard C syntax and is the same as referencing any regular variable. The value of the variable is retrieved from, or stored into, the local copy. This implies that the variable being referenced must be mapped to the environment in which the reference is made.

A remote reference, either a read or a write, involves communication between environments. It is indicated by the distributed variable name (and subscripts,

if any), followed by a "#", optionally followed by an explicit specification of the set of environments to communicate with for this reference. If this set is not provided explicitly, it is defined implicitly by the mapping function to be the set of all environments to which the given subrange of the distributed variable is mapped. The implicit environment specification is usually sufficient, and is considered preferable because it is more structured and because there is less likelihood of coding error.

We first describe the semantics of remote references in the case where the environment set is specified implicitly (by the associated mapping function). These references basically follow the convention that the home environment produces new values and the copy environments consume them.

If the remote reference is a read and the current environment is a copy environment, then DINO looks for the first (least recent) value of that variable that has been received from the home environment but not yet utilized (a message buffer is maintained, and the oldest message from the home environment with a value of that variable is used, and then removed from the buffer.) If no new value of that variable has been received from the home environment, the reading process blocks until one is received. When the new value is available, it is used to update the local copy of the variable, and then the remainder of the expression in which the remote read is located is evaluated using the local copy. If the remote reference is a read and the current environment is a home environment, then the remote read is the same as a local read (with some performance degradation,) and may produce a run-time warning message (see Section 6.2.) If the current environment is neither a home nor a copy environment, then a remote read is an error.

A remote write to a distributed variable using an implicit environment set works as follows. If the current environment is the home environment of that variable and there are associated copy environments, then the value being assigned to that variable is used to update the local copy (since the write occurs in an assignment context), and also is sent to all the copy environments. If the current environment is the home environment and there are no associated copy environments, then the remote write is treated as a local write (again, there is some performance degradation). If the current environment is not the home environment, then an implicit remote write is an error.

As example of remote reads and a remote write, consider a 2-dimensional smoothing algorithm where an N x M distributed array of data $A[][]$ is mapped onto an N x M array of virtual processors "grid", with each $A[i][j]$ mapped onto $grid[i][j]$ (its home environment) and the 4 adjoining environments (this is the mapping function FivePt.) Then the statement

```
A[i][j]# = (A[i-1][j]# + A[i][j-1]# + A[i][j]
         + A[i][j+1]# + A[i+1][j]#)/5;
```

will receive values of the 4 adjoining elements of $A[][]$ from their home environments, calculate the average of these four values plus the local value, assign this new value to the local copy of $A[i][j]$, and send it to the 4 adjoining environments.

If the programmer wishes to send or receive distributed variables in a different manner than by these implicit rules, a set of environments — ENV_EXP in the EBNF specification above — can be specified after the "#" sign. ENV_EXP uses the same syntax as was discussed for it in Section 3.3. For example,

```
grid[][] - grid[xid][yid] - grid[1][2]
```

specifies all the environments in the grid environment structure except the environment that the statement is executed in and *grid*[1][2]. In contrast to remote references with implicit environment sets, the distributed variable does not need to be mapped to the environment where the statement is executed.

A remote read with an explicit environment set is processed as follows. The process looks for the oldest unutilized value of that distributed variable that has been received from any node in the environment set, including itself if it is in the set, and blocks if no new value is available. Once a value is received, it is used in evaluating the remainder of the expression. (Note that the execution may be non-deterministic if there is more than one environment in the set.)

For a remote write with an explicit environment set, the new value of the variable is sent to all of the environments in the environment set, including itself if it is in the set. Note that explicit references to variables that are mapped to the local environment have different semantics than in the implicit case: they do generate messages and do not automatically update the local copy or default to local reference.

After a remote read where the environment set was specified explicitly and contained more than one element, it may be unknown which environment the new value came from. In order to obtain this information, the "from" construct may be used. The keyword "from" is appended to the environment set specification, and is followed by a variable of type "envvar". For example, the statements

```
envvar who;
...
w = x # {grid[][] from who};
...
y # {who} = z;
```

receive the value of x from some environment in the structure "grid", assign the name of this environment to *who*, and later send the value of z to the environment that the value of x was received from.

All of the above discussion pertains to "synchronous" distributed variables, which are the default in DINO. Distributed variables can also be declared to be "asynchronous" by placing the word "asynch" before "distributed" in their declarations. The difference between synchronous and asynchronous variables is that a remote read to an asynchronous variable uses the most recent value and does not block. If several

values of the variable have been received since the last remote read, then the most recently received value is used to update the local copy and the remaining values are discarded. If no new value has arrived since the last remote read, then the local copy is used. In keeping with these rules, an explicit remote read of an asynchronous variable requires that the variable be mapped to the current environment. The remaining semantics of remote references to asynchronous variables, including all the semantics for a remote write, are the same as for synchronous distributed variables.

By making the distinction between synchronous and asynchronous communication part of the distributed variable declaration, as opposed to a property of the statement that invokes communication, it is possible to transform a DINO program from a synchronous to an asynchronous variant by simply changing one or a few declarations. Clearly, the execution of algorithms containing asynchronous variables may be non-deterministic.

3.5 Reduction Functions

```
PRIMARY ::=
    REDUCTION '(' EXPRESSION [ ',' EXPRESSION ] ')' '#'
    [ '{' ENV_EXP '}' ].

REDUCTION ::=
    'gsum' | 'gprod' | 'gmin' | 'gmindex' | 'gmax' | 'gmaxdex'.
```

One type of calculation that involves communication is so fundamental to parallel computation, and departs sufficiently from simple patterns of communication, that we have included it as a parallel language construct. This is a reduction operation, which involves performing a commutative operation (e.g. +, *, min, max) over one value from each environment and returning the result to all the environments. DINO provides the reduction operators "gsum", "gprod", "gmax", and "gmin" that return the sum, product, maximum, or minimum of the arguments specified on each environment. For example, if `error` and `maxerror` exist on each environment in the current active set, then the statement

```
maxerror = gmax(error);
```

assigns the maximum of all the local values of `error` to each local copy of `maxerror`. DINO also provides two reduction functions, "gmaxdex" and "gmindex", that take two parameters; the first is the value to be reduced and the second is (a pointer to) an integer index. This is a user-defined index and is not related to any environment index; The programmer is responsible for assigning a unique index number to each environment participating in the reduction. When these functions return, the second parameter returns (points to) the index of the maximum (or minimum) value.

The reduction functions can be called with an explicit "active set" (using the optional '{' ENV_EXP '}') in the same way that a composite procedure or a remote data reference can. The active set specifies which environments will participate in the reduction. The first argument of a one argument reduction may evaluate to a (sub)array, rather than a scalar, in which case the reduction operation is performed individually over each component of the (sub)array.

It is implicit in a reduction call that synchronization between all participating environments is involved, and that all participating environments must reach the call, otherwise execution is blocked. The environments do not necessarily have to execute identical lines of code, but simply the same operation with matching parameter types. Reductions are implemented using efficient, tree-based computation and communication.

3.6 Subarrays and Ranges

```
PRIMARY ::= PRIMARY ( '[]' | '[<' EXPRESSION ',' EXPRESSION '>]' ).
```

To use most distributed memory multiprocessors efficiently, it is important to send messages consisting of blocks of data, rather than multiple messages consisting of single data elements, whenever possible. In order to efficiently move blocks of data between environments, DINO provides the ability to specify subarrays, and to use arrays and subarrays in simple assignment statements, which may include remote reads and writes (C provides no subarray facilities.)

A rectangular subsection of an array is specified by giving the first and the last element for each axis, e.g. "A[<i,j>]". The index numbers are separated by a comma and enclosed in angle brackets. The default value, specified by "[]", is all elements of the array along the indicated axis.

Arrays or subarrays, including remote references to them, may be used in assignment statements of the form

$$\texttt{A[<}I_1\texttt{,}J_1\texttt{>]}\ldots\texttt{[<}I_n\texttt{,}J_n\texttt{>]} = \texttt{B[<}K_1\texttt{,}L_1\texttt{>]}\ldots\texttt{[<}K_n\texttt{,}L_n\texttt{>]};$$

The size and shape of the operands in the statement must be consistent, and the indices of the left and right operand are mapped to each other in the obvious manner. If the left operand is a remote reference to a distributed array, then the target environments of each element depend on the mapping function. In this way an entire array can be distributed over a set of environments in a single write, with different portions possibly going to different environments. Conversely, if the right operand is a remote reference to a distributed (sub)array, this (sub)array is gathered from one or more environments in a single read.

DINO also allows the use of arrays and subarrays as parameters for composite procedures and reduction functions, and allows ranges to be used in specifying environment sets.

3.7 User-Defined Mappings

3.7.1 Mapping Philosophy

There are two general categories of information that programmer specified mappings in DINO include: information about data distribution, and information about communication patterns. That is, a mapping function may serve two purposes. It allows the compiler to statically map the data to the processors. It also allows the compiler to do a static analysis of communications resulting in more efficient communication.

The obvious purpose for a mapping is to allow the compiler to partition the data among the processors. While this may seem to simply be a matter of dividing a data structure up into "pieces" and sending each "piece" to a processor, in DINO it is somewhat more complex. It is often useful to have the same data value available to many processors. Although this can be implemented by having the programmer communicate the needed value once the data has been partitioned, it may be more efficient and easier for the programmer to think about the problem if the mapping allows for replication of the same "piece" on many processors.

Related to the concept of replicating data so that more than one processor may make use of it is the fact that the programmer may wish to update that value during the program. For this purpose, DINO mappings may also be used to provide some high level information about communication patterns (using overlaps.) In order to have a communication, the compiler must know "what" piece of data is being communicated, "when" is it being communicated, "where" is it being communicated to (from), and "how" it is communicated. DINO allows the programmer to specify what data piece is to be communicated and when (lexically) it is to sent and received. The compiler can then determine, based on the mapping, where it is to go to (or come from) and how the communication will be handled (all the underlying detail of messages).

To do this, DINO uses the following paradigm: First, when a data structure is mapped onto an environment structure, a partition is implicitly defined. That is, each "piece" of data is assigned to one processor that "owns" the data. We call this the "home" processor for that piece of data. Second, any other processor that will need to use that piece of data is assigned the same piece, but this processor is designated as a "copy" processor for that piece of data. Then, all sends and receives for this piece of data are from the home processor to all its copy processors. With this paradigm a parallel program is guaranteed to be deterministic, and communication patterns can be described simply by specifying the copy processors, if any, for each piece of data. This default paradigm can be overridden by the programmer but it is sufficient for many numerical algorithms.

3.7.2 Mapping Definition

```
MAPPING_FUNCTION ::=
    'map' IDENTIFIER '=' ( '[' MAP_TYPE [ALIGN] ']' )+.

MAP_TYPE ::= 'all' | 'compress' | BLOCK_MAP | WRAP_MAP.

BLOCK_MAP ::=
    'block' [ 'overlap' [ EXPRESSION ]] [ 'cross' 'axis' EXPRESSION ].

WRAP_MAP ::= 'wrap' [ EXPRESSION ].

ALIGN ::= 'align' 'axis' EXPRESSION.
```

We have attempted to provide a general mapping mechanism that still allows efficient implementation at compile time. Our mapping specification describes how to map arrays of data onto arrays of processors and is based on describing how each axis of the data structure is mapped to the processor structure (a sub-specification). By having an (almost) orthogonal set of sub-specifications, a powerful mapping specification can be constructed out of a simple set of primitives.

A mapping specification is defined in three steps. First, the programmer specifies how each axis of the data structure is matched to one or more axes of the structure of processors. Second, the programmer specifies how the data on that axis is distributed among the processors defined in the first step. There are three basic choices for this step, complete replication, partition, or partition with copies. Third, if partition with copies is desired, the programmer specifies how the copies of data are distributed to other processors.

In the first step, matching axes of the data structure to axes of the structure of processors, there are only two primitives — "compress" and "align". With compress, the programmer specifies that this data axis will not be distributed. This primitive is used if the data structure has more axes than the structure of processors. With align, the programmer specifies that a particular axis in the data structure should be mapped to a particular axis in structure of processors. If this primitive is omitted for an axis in the data structure, the obvious default of mapping the next available data axis to the next processor axis is used. (A data axis might not be available because it has been designated as "compress".)

For example, if the programmer is mapping a two dimensional matrix to a vector of processors, a mapping specification of the form:

```
[ . . . ][compress]
```

will distribute rows to the processors and a specification of the form:

```
[compress][ . . . ]
```

will distribute columns. The specific mapping of the rows or columns will depend on what goes in the [. . .] sub-specification, something that the programmer determines in steps two and three.

If the programmer is mapping a two dimensional matrix to a two dimensional structure of processors, the mapping specification:

```
[ . . .  align 1][ . . .  align 0]
```

would cause the second axis of the data structure to map to the first axis of the structure of processors, and the first axis of the data structure to map to the second axis of the structure of processors. In effect this allows the matrix to be transposed and placed on the processors.

In the second step, the distribution of a data axis to a processor axis, there are three mapping primitives — "all", "block", and "wrap". Using the first of these, the programmer can specify that the data axis be completely replicated across the associated axis of the structure of processors. With the second or third, the programmer can specify that the data axis is distributed in either one of two ways — blocked or wrapped. Block mappings assign one (approximately) equal sized contiguous piece of the data structure to each processor. Wrap mappings assign every Pth position on the data axis to the same processor (assuming P processors). Wrap is essentially a variant on block that is used for improved load balancing in a wide variety of parallel numerical algorithms. The programmer may also specify the width of the wrap.

If, in our first example, the matrix is N by N and there are P processors with $N = 4P$, then a mapping specification with:

```
[block] [compress]
```

would distribute four contiguous rows to each processor. Alternatively,

```
[compress] [wrap]
```

would put column 0, P, $2P$, ... on processor 0, etc.

In the third step, the specification of a distribution for copies, there are two mapping primitives that the programmer can use — "overlap" and "cross". "overlap" specifies that copies of neighboring data points on that axis will be available on each processor. The programmer can specify the direction and depth of the overlap. "cross" allows the programmer to specify that copies will be available for data points that are found in the intersection of two or more overlaps — in effect making copies of neighboring data points along diagonals available on each processor. This might be used, for example, in an algorithm that requires a nine point stencil.

For example, if a vector $[\, x_0 \;\; x_1 \;\; x_2 \;\; x_3 \,]$ is distributed across a vector of four processors with the mapping specification

```
[block overlap 1,1]
```

the four processors will have the following elements:

$$\begin{array}{ll} \textit{processor } 0 & [\,x_0 \;\; x_1 \;\;\;\;\;\;\;\;] \\ \textit{processor } 1 & [\,x_0 \;\; x_1 \;\; x_2 \;\;\;\;] \\ \textit{processor } 2 & [\;\;\;\; x_1 \;\; x_2 \;\; x_3\,] \\ \textit{processor } 3 & [\;\;\;\;\;\;\;\; x_2 \;\; x_3\,] \end{array}$$

The notation "`1,1`" specifies overlaps of one element to the left and right respectively. More precisely, the first "`1`" means that each environment will have a single element for the environment with the next lower index. The second "`1`" means that each environment will have a single element from the environment with the next lower index. Note that the "leftmost" and the "rightmost" processors will receive 1 element that is a copy instead of two, i.e., there is no wrap around. The home processor of each data element is the processor it would be mapped to if the overlap were omitted, in this case processor i for x_i.

If we distribute an N by N matrix across an N by N structure of processors with the following mapping specification:

```
[block overlap 1,1 cross 1][block overlap 1, 1]
```

then the data on each processor (except the edge processors) will have the following pattern (nine point stencil):

$$\begin{bmatrix} c & N & c \\ W & D & E \\ c & S & c \end{bmatrix}$$

where D is the data for that processor, N, S, E, and W are copies of data from four neighboring processors due to the overlap primitive, and the c's are copies of data from the four diagonal neighbors due to the cross primitive.

Using this structured approach allows us to combine a rather small set of primitives for each data axis to generate a very large set of mappings from data structures to processors. For example, if an N by N matrix is to be distributed across a vector of N processors so that each processor received a column plus copies of the two columns to the left of "its" column, the mapping would be:

```
[compress][block overlap 2,0]
```

The "leftmost" and the "next leftmost" processors will receive, respectively, 0 and 1 columns that are copies.

As a final example, if an N by N matrix is to be distributed across a vector of P processors, where N is 16 and P is 4, so that each processor receives copies of the closest row from the two processors next to it, the mapping would be:

```
[block overlap 1,1][compress]
```

The resulting data structure on a processor (for an interior processor) would be:

$$\begin{bmatrix} u & u & u & u & u & u & u & u \\ D & D & D & D & D & D & D & D \\ D & D & D & D & D & D & D & D \\ D & D & D & D & D & D & D & D \\ D & D & D & D & D & D & D & D \\ d & d & d & d & d & d & d & d \end{bmatrix}$$

where D is the home data for that processor, and u and d are copies of data from the processors above and below respectively.

4 DINO Examples

This section contains a number of DINO examples, which illustrate most of the features of DINO. Each example is listed on its own page, with the path name of the file, relative to the examples directory, at the top of the page. An Index of Features is provided which allows you to find an example program containing the feature you want to look at. The Index lists only one example for each feature, even though some features may be illustrated in many examples.

4.1 Index of Features

4.2 "Hello World"

The programs in this section simply print out messages from the environments created in the programs. They illustrate how to declare environment structures and simple composite procedures.

Example 1.1: examples/helloworld/ex1.d

```
/* This program prints out messages from a one dimensional environment
 * structure.  */

#define P 16

environment node[P:id] {          /* ==> Declaration of a one dimensional
                                         environment structure of size P, with an
                                         environment index identifier id, which
                                         tells WHICH environment we are.  */

  composite go()                  /* ==> Declaration of a composite
                                         procedure with no parameters.  */
  {
    printf ("node[%d] says hello\n", id);
  }

}

environment host {                /* ==> Declaration of a scalar environment
                                         structure "host" is required in
                                         all DINO programs.  */

  void main ()                    /* ==> Execution starts at function
                                         "main" within environment "host"
                                         in all DINO programs.  */
  {
    printf ("host says hello\n");

    go()#;                        /* ==> Call of a composite procedure
                                         (indicated by the "#" sign).  */
    printf ("host says goodbye\n");
  }
}
```

Example 1.2: examples/helloworld/ex2.d

```
/* This program prints out messages from a two dimensional environment
 * structure.  */

#define P1 3
#define P2 4

environment node[P1:id1][P2:id2] {
                                  /* ==> Declaration of a two dimensional
                                         environment structure of size P1 by P2,
                                         declaring environment index indentifiers
                                         id1 and id2.  */
  composite go()

  {
    printf ("node[%d][%d] says hello\n", id1, id2);
                                  /* ==> Uses the environment index identifiers
                                         id1 and id2.  */
  }

}

environment host {

  void main ()
  {
    printf ("host says hello\n");
    go()#;
    printf ("host says goodbye\n");
  }
}
```

Example 1.3: examples/helloworld/ex3.d

```
/* This program prints out messages from two composite procedures, inside of a
 * single one dimensional environment structure.  */

#define P 14

environment node[P:id] {        /* ==> The size of an environment structure
                                       does not have to be a power of two */

  composite part1()

  {
    printf ("node[%d] says hello from part1()#\n", id);
  }

  composite part2()             /* ==> Multiple composite procedures may be
                                       declared within the same environment
                                       structure.    However, only one of these
                                       may be active at any one time.  */

  {
    printf ("node[%d] says hello from part2()#\n", id);
  }

}

environment host {

  void main ()

  {
    printf ("host says hello\n");

    part1()#;
    part2()#;                   /* ==> Part1()# must finish execution before
                                       Part2()# begins.  */

    printf ("host says goodbye\n");
  }

}
```

Example 1.4: examples/helloworld/ex4.d

```
/* This program prints out messages from two composite procedures, inside of
 * two different one dimensional environment structures.  */

#define P1 8
#define P2 6

environment node1[P1:id] {

  composite part1()
  {
    printf ("node1[%d] says hello from part1()#\n", id);
  }

}

environment node2[P2:id] {      /* ==> Multiple environment structures are
                                       allowed, and are simply declared one
                                       after another.  */

  composite part2()
  {
    printf ("node2[%d] says hello from part2()#\n", id);
  }

}

environment host {

  void main ()
  {
    printf ("host says hello\n");

    part1()#;
    part2()#;                   /*==> Part1()# must finish before Part2()#
                                       begins execution.  */

    printf ("host says goodbye\n");
  }
}
```

Example 1.5: examples/helloworld/ex5.d

```
/* This program prints out messages from two composite procedures which are
 * called in parallel, inside of two different one dimensional environment
 * structures.  */

#define P1 8
#define P2 6

environment node1[P1:id] {

  composite part1()
  {
    printf ("node1[%d] says hello from part1()#\n", id);
  }

}

environment node2[P2:id] {

  composite part2()
  {
    printf ("node2[%d] says hello from part2()#\n", id);
  }

}

environment host {

  void main ()
  {
    printf ("host says hello\n");

    part1()# ::  part2()#;       /* ==> Part1()# and part2()# run in parallel,
                                        through the use of the "::" operator.
                                        These two composite procedures must be
                                        in different environments in order for
                                        the parallelism to work.  */

    printf ("host says goodby\n");
  }
}
```

4.3 Matrix-Vector Multiply

These programs all solve the identical problem — a matrix-vector multiplication, but each program divides the data up among the processors in different ways.

Example 2.1: examples/matvec/ex1.d

```
/* This program computes a matrix - vector product, dividing the matrix into
 * rows, with one row per processor.  The matrix is of size M x N. */

#include "dino.h"                /* ==> Includes the predefined DINO mapping
                                        functions.  */

#define M 16
#define N 9

environment node[M:id] {

  composite matvec (in a, in x, out y)
                                 /* ==> Declaration of a composite procedure
                                        with three parameters, whose declara-
                                        tions follow immediately.    (Note the
                                        use of the keywords "in" and "out".)  */

  double distributed a[M][N] map BlockRow; /* Input matrix */
                                 /* ==> Declaration of distributed data requires
                                        use of the word "distributed" before
                                        the variable name, and must be followed
                                        by "map" and the name of the mapping
                                        function to be used.  */

                                 /* ==> This is an example of using a distributed
                                        data structure as a parameter to a
                                        composite procedure.  */

  double distributed x[N] map all;   /* Input vector */
                                 /* ==> Illustrates use of the built-in mapping
                                        function "all".  */

  double distributed y[M] map Block; /* Result vector */

  {
    int i;              /* Looping variable */

    /* Each environment computes y[id] */
    y[id] = 0;                   /* ==> Since y[] has as the same number of
```

```
                                        elements as the environment structure,
                                        each environment has y[id] in local
                                        memory.  */

                                    /* ==> The local copy of a distributed
                                        data structure is refferenced like
                                        an ordinary variable.  */

    for (i = 0; i < N; i++)
      y[id] += a[id][i] * x[i];
  }
}

environment host {

  double a[M][N];
  double x[N];
  double y[M];

  void main ()

  {
    int i, j;           /* Looping variables */

    /* Set up the initial data for a[][] and v[] */
    for (j = 0; j < N; j++) {
      x[j] = j;
      for (i = 0; i < M; i++)
        a[i][j] = i + j;
    }

    /* Print out the initial data */
    printf ("Initial data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%6.2f", a[i][j]);
      printf ("\n");
    }

    printf ("\nInitial data for x:\n");
    for (i = 0; i < N; i++)
      printf ("%6.2f\n", x[i]);

    /* Perform the computation */
    matvec (a[][], x[], y[])#;  /* ==> Call of a composite procedure with
                                        parameters.    Note the use of "[]"
                                        to signify use of an entire axis of
                                        the data.    Note that composite
```

```
                                        procedure calls do not use pointers
                                        to return results.  */

    /* Printout the resulting vector y */
    printf ("\nResult data for y:\n");
    for (i = 0; i < M; i++)
      printf ("%6.2f\n", y[i]);
  }
}
```

Example 2.2: examples/matvec/ex2.d

```
/* This program computes a matrix - vector product, dividing the matrix into
 * rows, with multiple rows per processor.  The matrix is of size M x N,
 * where M must be a multiple of P (meaning each processor has an equal
 * number of rows).  */

#include "dino.h"

#define P 4
#define M 16
#define N 9

environment node[P:id] {

  composite matvec (in a, in x, out y)
  double distributed a[M][N] map BlockRow;      /* Input matrix */
  double distributed x[N] map all;              /* Input vector */
  double distributed y[M] map Block;            /* Result vector */

  {
    int i, j;           /* Looping variables */

    /* Loop thru the rows of a[][] which are on this environment */
    for (i = id * M/P; i < (id + 1) * M/P; i++) {

                                /* ==> In this program, M is a multiple of P,
                                       so each environment has M/P rows of
                                       the marix a[][], and M/P elements of
                                       the vector y[].    The environment index
                                       identifier "id" is used to compute
                                       which elements of a distributed data
                                       structure an environment has.  */

      /* Compute y[i] */
      y[i] = 0;
      for (j = 0; j < N; j++)
        y[i] += a[i][j] * x[j];
    }
  }
}

environment host {

  double a[M][N];
  double x[N];
  double y[M];
```

```
  void main ()

  {
    int i, j;           /* Looping variables */

    /* Set up the initial data for a[][] and v[] */
    for (j = 0; j < N; j++) {
      x[j] = j;
      for (i = 0; i < M; i++)
        a[i][j] = i + j;
    }

    /* Print out the initial data */
    printf ("Initial data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%6.2f", a[i][j]);
      printf ("\n");
    }

    printf ("\nInitial data for x:\n");
    for (i = 0; i < N; i++)
      printf ("%6.2f\n", x[i]);

    /* Perform the computation */
    matvec (a[][], x[], y[])#;

    /* Printout the resulting vector y */
    printf ("\nResult data for y:\n");
    for (i = 0; i < M; i++)
      printf ("%6.2f\n", y[i]);
  }
}
```

Example 2.3: examples/matvec/ex3.d

```
/* This program computes a matrix - vector product, dividing the matrix into
 * rows, with multiple rows per processor.  The matrix is of size M x N,
 * where M must be greater than P, but not necessarily a multiple.    The
 * fairly complex calculations which are required are done by hand.  */

#include "dino.h"

#define P 3
#define M 16
#define N 9

environment node[P:id] {

  composite matvec (in a, in x, out y)
  double distributed a[M][N] map BlockRow;      /* Input matrix */
  double distributed x[N] map all;              /* Input vector */
  double distributed y[M] map Block;            /* Result vector */

  {
    int i, j;                   /* Looping variables */
    int firstrow, lastrow;      /* The first and last rows on this processor */

                                /* ==> Since the number of rows in the matrix
                                       a[][] is not a multiple of the size of
                                       the environment, a complicated
                                       calculation based on id must be done to
                                       compute what rows of a[][] are on this
                                       environment.  */

    /* Compute firstrow and lastrow */
    firstrow = id * M/P + (id < M%P ? id :  M%P);
    lastrow = (id + 1) * M/P + ((id + 1) < M%P ? id :  M%P - 1);

    /* Loop thru the rows of a[][] which are on this environment */
    for (i = firstrow; i <= lastrow; i++) {

      /* Compute y[i] */
      y[i] = 0;
      for (j = 0; j < N; j++)
        y[i] += a[i][j] * x[j];
    }
  }
}

environment host {
```

```
  double a[M][N];
  double x[N];
  double y[M];

  void main ()

  {
    int i, j;           /* Looping variables */

    /* Set up the initial data for a[][] and v[] */
    for (j = 0; j < N; j++) {
      x[j] = j;
      for (i = 0; i < M; i++)
        a[i][j] = i + j;
    }

    /* Print out the initial data */
    printf ("Initial data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%6.2f", a[i][j]);
      printf ("\n");
    }

    printf ("\nInitial data for x:\n");
    for (i = 0; i < N; i++)
      printf ("%6.2f\n", x[i]);

    /* Perform the computation */
    matvec (a[][], x[], y[])#;

    /* Printout the resulting vector y */
    printf ("\nResult data for y:\n");
    for (i = 0; i < M; i++)
      printf ("%6.2f\n", y[i]);
  }
}
```

Example 2.4: examples/matvec/ex4.d

```
/* This program computes a matrix - vector product, dividing the matrix into
 * rows, with multiple rows per processor.  The matrix is of size M x N,
 * where M must be greater than P, but not necessarily a multiple.    This
 * program uses methodology and functions provided by the map.c include
 * file.    Use of a package along these lines can be helpful when a large
 * number of distributed objects of sizes which are not multiples of the
 * environment structure size are used.  */

/* A listing of the map.c file appears immediately after this program.  */

#include "dino.h"

#define P 3
#define M 16
#define N 9

environment node[P:id] {

#include "../inc/map.c"

                                /* ==> Includes the map.c include file.    Since
                                       this file contains code, it must be
                                       included within an environment.  */

  composite matvec (in a, in x, out y)
  double distributed a[M][N] map BlockRow;      /* Input matrix */
  double distributed x[N] map all;              /* Input vector */
  double distributed y[M] map Block;            /* Result vector */

  {
    int i, j;           /* Looping variables */
    map_var a1;                 /* ==> Declares a variable to hold useful
                                       information about an axis of a
                                       distributed data structure which
                                       is [block] mapped.  */

    /* Setup map_var a1 */
    set_map_var (M,     /* Size of the axis */
                 P,     /* Size of the environment axis it's mapped to */
                 id,    /* Environment index identifier for that axis */
                 0,     /* Left overlap of the mapping function */
                 0,     /* Right overlap of the mapping function */
                 &a1);  /* Address of the map_var */

                                /* ==> This function, defined in map.c, sets
```

```
                                        up the map_var a1 to contain information
                                        about the first axis of the distributed
                                        variable a (which is the same as the
                                        only axis of x).  */

    /* Loop thru the rows of a[][] which are on this environment */
    for (i = a1.left; i <= a1.right; i++) {

                                    /* ==> This statement loops thru the correct
                                           rows of a[][] by using information from
                                           a1, the map_var set up earlier.
                                           "a1.left" and "a1.right" refer to the
                                           first and last elements of the "home"
                                           data on this environment */
      /* Compute y[i] */
      y[i] = 0;
      for (j = 0; j < N; j++)
        y[i] += a[i][j] * x[j];
    }
  }
}

environment host {

  double a[M][N];
  double x[N];
  double y[M];

  void main ()

  {
    int i, j;           /* Looping variables */

    /* Set up the initial data for a[][] and v[] */
    for (j = 0; j < N; j++) {
      x[j] = j;
      for (i = 0; i < M; i++)
        a[i][j] = i + j;
    }

    /* Print out the initial data */
    printf ("Initial data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%6.2f", a[i][j]);
      printf ("\n");
    }
```

```
    printf ("\nInitial data for x:\n");
    for (i = 0; i < N; i++)
      printf ("%6.2f\n", x[i]);

    /* Perform the computation */
    matvec (a[][], x[], y[])#;

    /* Printout the resulting vector y */
    printf ("\nResult data for y:\n");
    for (i = 0; i < M; i++)
      printf ("%6.2f\n", y[i]);
  }
}
```

Example : examples/inc/map.c

```
/* This file contains functions for easy manipulation of DINO distributed
 * arrays where each processor can have different sized pieces.  */

typedef struct {
  int left; /* First element of home data */
  int right; /* Last element of home data */
  int lover; /* First element of overlap data */
  int rover; /* Last element of overlap data */
} map_var;

/* This function computes a map variable for a given axis of a distributed
 * array, given the size of the axis, the size of the environment axis it
 * is mapped to, the index number for that axis, and the left and right
 * overlaps.  */

void set_map_var (n_, p_, i_, l_, r_, M_)
int n_; /* Size of the axis */
int p_; /* Size of the environment axis it's mapped to */
int i_; /* The index of this environment */
int l_; /* The left overlap of the mapping */
int r_; /* The right overlap of the mapping */
map_var *M_; /* The result variable */

{
  M_->left = i_ * (n_/p_) + (i_ < n_%p_ ?  i_ :  n_%p_);
  M_->right = (i_ + 1) * (n_/p_) + ((i_ + 1) < n_%p_ ?  i_ :  n_%p_ - 1);
  M_->lover = (M_->left > l_ ?  M_->left - l_ :  0);
  M_->rover = (M_->right + r_ < n_ ?  M_->right + r_ :  n_ - 1);
}

/* This function modifies a map variable to only use elements of the array
 * which are within the given bounds.  */

void limit_map_var (l_, r_, M_)
int l_; /* Left limit */
int r_; /* Right limit */
map_var *M_; /* The map variable */

{
  if (M_->left < l_) M_->left = l_;
  if (M_->lover < l_) M_->lover = l_;
  if (M_->right > r_) M_->right = r_;
  if (M_->rover > r_) M_->rover = r_;
}
```

Example 2.5: examples/matvec/ex5.d

```
/* This program computes a matrix - vector product, dividing the matrix into
 * columns, with one column per processor.  The matrix is of size M x N. */

#include "dino.h"

#define M 16
#define N 9

environment node[N:id] {

  composite matvec (in a, in x, out y)
  double distributed a[M][N] map BlockCol;      /* Input matrix */
  double distributed x[N] map Block;            /* Input vector */
  double distributed y[M] map all;              /* Result vector */

  {
    int i;                      /* Looping variable */

    /* Each processor computes its contribution to the final result y[] */
    for (i = 0; i < M; i++)
      y[i] = a[i][id] * x[id];

    /* Now, we do a sum across all processors to compute the final result */
    y[] = gsum(y[])#;           /* ==> Reduction functions are called just like
                                       an ordinary function, with a "#" sign
                                       following it.  */

                                /* ==> Reductions of array quantities work on
                                       an element-by-element basis */

  }
}

environment host {

  double a[M][N];
  double x[N];
  double y[M];

  void main ()

  {
    int i, j;           /* Looping variables */

    /* Set up the initial data for a[][] and v[] */
```

```
    for (j = 0; j < N; j++) {
      x[j] = j;
      for (i = 0; i < M; i++)
        a[i][j] = i + j;
    }

    /* Print out the initial data */
    printf ("Initial data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%6.2f", a[i][j]);
      printf ("\n");
    }

    printf ("\nInitial data for x:\n");
    for (i = 0; i < N; i++)
      printf ("%6.2f\n", x[i]);

    /* Perform the computation */
    matvec (a[][], x[], y[])#;

    /* Printout the resulting vector y */
    printf ("\nResult data for y:\n");
    for (i = 0; i < M; i++)
      printf ("%6.2f\n", y[i]);
  }
}
```

Example 2.6: examples/matvec/ex6.d

```
/* This program computes a matrix - vector product, dividing the matrix into
 * columns, with multiple columns per processor.  The matrix is of size M x N,
 * where N must be a multiple of P. */

#include "dino.h"

#define P 3
#define M 16
#define N 9

environment node[P:id] {

  composite matvec (in a, in x, out y)
  double distributed a[M][N] map BlockCol;      /* Input matrix */
  double distributed x[N] map Block;            /* Input vector */
  double distributed y[M] map all;              /* Result vector */

  {
    int i, j;                   /* Looping variable */

    /* Each processor computes a contribution to the final result y[] */
    for (i = 0; i < M; i++) {
      y[i] = 0;
      for (j = id * N/P; j < (id + 1) * N/P; j++)
        y[i] += a[i][j] * x[j];
    }

    /* Now, we do a sum across all processors to compute the final result */
    y[] = gsum(y[])#;

  }
}

environment host {

  double a[M][N];
  double x[N];
  double y[M];

  void main ()

  {
    int i, j;           /* Looping variables */

    /* Set up the initial data for a[][] and v[] */
```

```
    for (j = 0; j < N; j++) {
      x[j] = j;
      for (i = 0; i < M; i++)
        a[i][j] = i + j;
    }

    /* Print out the initial data */
    printf ("Initial data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%6.2f", a[i][j]);
      printf ("\n");
    }

    printf ("\nInitial data for x:\n");
    for (i = 0; i < N; i++)
      printf ("%6.2f\n", x[i]);

    /* Perform the computation */
    matvec (a[][], x[], y[])#;

    /* Printout the resulting vector y */
    printf ("\nResult data for y:\n");
    for (i = 0; i < M; i++)
      printf ("%6.2f\n", y[i]);
  }
}
```

Example 2.7: examples/matvec/ex7.d

```
/* This program computes a matrix - vector product, dividing the matrix into
 * blocks, with multiple elements per processor.  The matrix is of size M x N,
 * where M must be a multiple of P1, and N must be a multiple of P2.  */

#include "dino.h"

#define P1 4
#define P2 3
#define M 16
#define N 9

environment node[P1:id1][P2:id2] {

  map BlockAll = [block][all];  /* ==> These are examples of user defined
  map AllBlock = [all][block];         mappings.    The extra dimentions had
                                       to be used because of a deficiency in
                                       DINO mappings.  */

  composite matvec (in a, in x, out y)
  double distributed a[M][N] map BlockBlock;    /* Input matrix */
  double distributed x[1][N] map AllBlock;      /* Input vector */
  double distributed y[M][1] map BlockAll;      /* Result vector */

  {
    int i, j;                   /* Looping variable */

    /* Loop through the elements of y[] mapped to this procesor */
    for (i = id1 * M/P1; i < (id1 + 1) * M/P1; i++) {

      /* Compute the contribution of this processor to the final result */
      y[i][0] = 0;
      for (j = id2 * N/P2; j < (id2 + 1) * N/P2; j++)
        y[i][0] += a[i][j] * x[0][j];
    }

    /* Now, we do a sum across all processors in my row to compute the final
     * result */
    y[<id1 * M/P1,(id1 + 1) * M/P1 - 1>][0] =
        gsum(y[<id1 * M/P1,(id1 + 1) * M/P1 - 1>][0])# {node[id1][]};
                                /* ==> This is an example of a reduction using
                                       an explicit environment set, which is
                                       placed in brackets after the "#" sign.  */

  }
}
```

```
environment host {

  double a[M][N];
  double x[1][N];
  double y[M][1];

  void main ()

  {
    int i, j;           /* Looping variables */

    /* Set up the initial data for a[][] and v[] */
    for (j = 0; j < N; j++) {
      x[0][j] = j;
      for (i = 0; i < M; i++)
        a[i][j] = i + j;
    }

    /* Print out the initial data */
    printf ("Initial data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%6.2f", a[i][j]);
      printf ("\n");
    }

    printf ("\nInitial data for x:\n");
    for (i = 0; i < N; i++)
      printf ("%6.2f\n", x[0][i]);

    /* Perform the computation */
    matvec (a[][], x[][], y[][])#;

    /* Printout the resulting vector y */
    printf ("\nResult data for y:\n");
    for (i = 0; i < M; i++)
      printf ("%6.2f\n", y[i][0]);
  }
}
```

4.4 Smoothing Algorithms

The programs in this section perform a "smoothing" algorithm. Each point is continually recomputed to be the average of some subset of the points around it. These examples demonstrate the use of overlaps in mapping functions.

Example 3.1: examples/smoothing/ex1.d

```
/* This program performs a horizontal smoothing, with one row of the matrix
 * per processor.    The matrix is of size M x N.  Each iteration of the
 * algorithm sets row i to be the average of rows i-1 and i+1, except for
 * rows on the edge of the matrix, which are left constant.  */

#include "dino.h"

#define max(x,y) (x > y ?  x :  y)
#define min(x,y) (x < y ?  x :  y)

#define M 16
#define N 11

environment node[M:id] {

  composite smooth (a, in iter)

                                /* ==> Since no "in" or "out" keyword has been
                                       used before a, it is an in/out
                                       parameter.  */

  double distributed a[M][N] map BlockRowOverlap;

  int iter;                     /* ==> Illustrates use of a non-distributed
                                       parameter to a composite procedure.
                                       Note the use of the "in" keyword in
                                       the parameter list.  */

  {
    int i, j;           /* Looping variables */

    /* Repeat the smoothing process iter times */
    for (i = 0; i < iter; i++) {

      /* Send out your data and receive it back again */

      if (i != 0) {             /* ==> We don't need to communicate on the
                                       first iteration, because the data we
```

```
                                        need has been set up by the composite
                                        procedure call.  */

        a[<id,id>][]# = a[<id,id>][];

                                    /* ==> Implicit send of the id'th row of a.
                                        Notice the use of the # sign, and how
                                        an assignment statement is used to
                                        send data.  */

                                    /* ==> Used a[<id,id>][]# instead
                                        of a[id][]# because of a bug in DINO. */

        a[<max(id-1,0),min(id+1,M-1)>][]#;

                                    /* ==> Implicit receive.    Receives rows id-1
                                        and id+1, except on the edges, where
                                        it only receives one row.  */

      }

      /* Perform the computation, but only on non-edge nodes */
      if (id != 0 && id != M - 1)
        for (j = 0; j < N; j++)
          a[id][j] = (a[id-1][j] + a[id+1][j]) / 2;

    }
  }
}

environment host {

  void main ()

  {
    double a[M][N];                     /* Input data */
    int iter;                           /* Holds the iteration count */

    int i, j;                           /* Looping variables */

    /* Set up the initial data for a[][] */
    for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
        a[i][j] = (i + 1)*(j + 1);
    for (i = 1; i < M - 1; i++)
      for (j = 0; j < N; j++)
        a[i][j] = 0;
```

```
    iter = 300;                 /* ==> We must use a variable to hold the
                                       number of iterations, because of a
                                       bug in DINO which doesn't allow
                                       passing constants to a composite
                                       procedure.  */

    /* Print out the initial data */
    printf ("Initial data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%6.2f", a[i][j]);
      printf ("\n");
    }

    /* Perform the computation */
    smooth (a[][], iter)#;

    /* Printout the results */
    printf ("Result data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%6.2f", a[i][j]);
      printf ("\n");
    }
  }
}
```

Example 3.2: examples/smoothing/ex2.d

```
/* This program performs a vertical smoothing, with the matrix partitioned
 * by columns, with multiple columns per processor.    The matrix is of size
 * M x N, where N must be a multiple of P. */

#include "dino.h"

#define max(x,y) (x > y ?  x :  y)
#define min(x,y) (x < y ?  x :  y)

#define M 16
#define N 16
#define P 4

environment node[P:id] {

  composite smooth (a, in iter)
  double distributed a[M][N] map BlockColOverlap;
  int iter;

  {
    int i, j, k;              /* Looping variables */

    /* Repeat the smoothing process iter times */
    for (i = 0; i < iter; i++) {

      /* Send out your data and receive it back again, if not the first
       * iteration */
      if (i != 0) {

        a[][<id * N/P,(id + 1) * N/P - 1>]# =
        a[][<id * N/P,(id + 1) * N/P - 1>];

                                  /* ==> This statement sends out those columns
                                         of a which are "home" on this
                                         environment.    Only those columns which
                                         have copies on another environment
                                         are actually sent.  */

        a[][<max(id * N/P - 1, 0),min((id + 1) * N/P, N - 1)>]#;

                                  /* ==> Receives the "copy" rows.    Only those
                                         columns which are copies are actually
                                         sent.    The max() and min() calls
                                         deal with the edge cases */
```

```
      }

      /* Perform the computation, but only on non-edge columns */
      for (j = 0; j < M; j++)
        for (k = max(id * N/P, 1); k <= min((id + 1) * N/P - 1, N - 2); k++)
          a[j][k] = (a[j][k-1] + a[j][k+1]) / 2;

    }
  }
}

environment host {

  void main ()

  {
    double a[M][N];                    /* Input data */
    int iter;                          /* Holds the iteration count */

    int i, j;                          /* Looping variables */

    /* Set up the initial data for a[][] */
    for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
        a[i][j] = (i + 1)*(j + 1);
    for (i = 0; i < M; i++)
      for (j = 1; j < N - 1; j++)
        a[i][j] = 0;

    /* Set up the variable which will contain the number of iterations */
    iter = 300;

    /* Print out the initial data */
    printf ("Initial data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%6.2f", a[i][j]);
      printf ("\n");
    }

    /* Perform the computation */
    smooth (a[][], iter)#;

    /* Printout the results */
    printf ("Result data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%6.2f", a[i][j]);
```

```
      printf ("\n");
    }
  }
}
```

Example 3.3: examples/smoothing/ex3.d

```
/* This program computes a five point smoothing algorithm (each point is
 * set to be the average of his neighbors to the north, south, west, and east).
 * It operates on a matrix of size M x N, blocking the matrix into small
 * chuncks on a two dimensional processor array.    M must be a multiple of P1,
 * and N a multiple of P2.  */

#include "dino.h"

#define max(x,y) (x > y ?  x :  y)
#define min(x,y) (x < y ?  x :  y)

#define M 16
#define N 16
#define P1 4
#define P2 4

environment node[P1:id1][P2:id2] {

  composite smooth (a, in iter)
  double distributed a[M][N] map FivePt;

                                /* ==> The FivePt mapping divides the matrix a
                                       into blocks of size M/P1 x N/P2.    Each
                                       environment is assigned one of these
                                       blocks, plus the one element wide strips
                                       to each of the four sides.  */

  int iter;

  {
    int i, j, k;  /* Looping variables */

    int home_n, home_s, home_w, home_e;
        /* Boundaries of the home data, not including the edges of
         * the matrix.  */

    int copy_n, copy_s, copy_w, copy_e;
        /* Boundaries of the copy data, not including the edges of
         * the matrix.  */

                                /* ==> To make the program easier to read and
                                       understand, varaibles containing
                                       the ranges of home and copy data on each
                                       processor are useful.  */
```

```
    /* Compute home_n, home_s, home_w, and home_e */
    home_n = max (M/P1 * id1, 1);
    home_s = min ((id1 + 1) * M/P1 - 1, M-2);
    home_w = max (N/P2 * id2, 1);
    home_e = min ((id2 + 1) * N/P2 - 1, N-2);

    /* Compute copy_n, copy_s, copy_w, and copy_e */
    copy_n = max (home_n - 1, 1);
    copy_s = min (home_s + 1, M-2);
    copy_w = max (home_w - 1, 1);
    copy_e = min (home_e + 1, N-2);

    /* Repeat the smoothing process iter times */
    for (i = 0; i < iter; i++) {

      /* Send out your data and receive it back again, if not the first
       * iteration */
      if (i != 0) {
        a[<home_n,home_s>][<home_w,home_e>]# =
          a[<home_n,home_s>][<home_w,home_e>];
        a[<copy_n,copy_s>][<copy_w,copy_e>]#;
      }

      /* Perform the computation, but only on non-edge elements */
      for (j = home_n; j <= home_s; j++)
        for (k = home_w; k <= home_e; k++)
          a[j][k] = (a[j][k-1] + a[j][k+1] + a[j-1][k] + a[j+1][k]) / 4;

    }
  }
}

environment host {

  void main ()

  {
    double a[M][N];                    /* Input data */
    int iter;                          /* Holds the iteration count */

    int i, j;                          /* Looping variables */

    /* Set up the initial data for a[][] */
    for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
        a[i][j] = (i + 1)*(j + 1);
    for (i = 1; i < M - 1; i++)
      for (j = 1; j < N - 1; j++)
```

```
        a[i][j] = 0;

    /* Set up the variable which will contain the number of iterations */
    iter = 500;

    /* Print out the initial data */
    printf ("Initial data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%7.2f", a[i][j]);
      printf ("\n");
    }

    /* Perform the computation */
    smooth (a[][], iter)#;

    /* Printout the results */
    printf ("Result data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%7.2f", a[i][j]);
      printf ("\n");
    }
  }
}
```

Example 3.4: examples/smoothing/ex4.d

```
/* This program computes a five point smoothing algorithm, just like ex3.d,
 * except that it deals with matrix sizes which are not a multiple of the
 * number of processors.    It uses the include file d "map.c" to do
 * this.  */

#include "dino.h"

#define max(x,y) (x > y ?  x :  y)
#define min(x,y) (x < y ?  x :  y)

#define M 11
#define N 8
#define P1 3
#define P2 3

environment node[P1:id1][P2:id2] {

#include "/anchor/student/derby/doc/examples/inc/map.c"

                                /* ==> Includes the map.c include file.    Since
                                       this file contains code, it must be
                                       included within an environment.    See
                                       the listing after example 2.4.  */

  composite smooth (a, in iter)
  double distributed a[M][N] map FivePt;
  int iter;

  {
    int i, j, k;        /* Looping variables */
    map_var a1, a2;

    /* Setup map_var's a1 and a2 for the complete data structure A */
    set_map_var (M, P1, id1, 1, 1, &a1);
    set_map_var (N, P2, id2, 1, 1, &a2);

                                /* ==> Illustrates the use of set_map_var
                                       with overlaps (see examples/matvec/ex4.d
                                       for an explanation of set_map_var).  */

    /* Limit the map_var's to not use the edges of A, which are boundary
     * conditions that don't change */

    limit_map_var (1,           /* Minimum data used */
                   M-2,         /* Maximum data used */
```

```
                    &a1);        /* Address of the map_var */

                                 /* ==> This function, defined in map.c, limits
                                        the map_var a1 to the range <1,M-2>.
                                        This means that the edge data is not
                                        send and/or received when map_var a1
                                        is used.    */

    limit_map_var (1, N-2, &a2);

    /* Repeat the smoothing process iter times */
    for (i = 0; i < iter; i++) {

      /* Send out your data and receive it back again, if not the first
       * iteration */
      if (i != 0) {
        a[<a1.left,a1.right>][<a2.left,a2.right>]# =
          a[<a1.left,a1.right>][<a2.left,a2.right>];
        a[<a1.lover,a1.rover>][<a2.lover,a2.rover>]#;
      }

      /* Perform the computation, but only on non-edge elements */
      for (j = a1.left; j <= a1.right; j++)
        for (k = a2.left; k <= a2.right; k++)
          a[j][k] = (a[j][k-1] + a[j][k+1] + a[j-1][k] + a[j+1][k]) / 4;

    }
  }
}

environment host {

  void main ()

  {
    double a[M][N];                  /* Input data */
    int iter;                        /* Holds the iteration count */

    int i, j;                        /* Looping variables */

    /* Set up the initial data for a[][] */
    for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
        a[i][j] = (i + 1)*(j + 1);
    for (i = 1; i < M - 1; i++)
      for (j = 1; j < N - 1; j++)
        a[i][j] = 0;
```

```
    /* Set up the variable which will contain the number of iterations */
    iter = 100;

    /* Print out the initial data */
    printf ("Initial data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%7.2f", a[i][j]);
      printf ("\n");
    }

    /* Perform the computation */
    smooth (a[][], iter)#;

    /* Printout the results */
    printf ("Result data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%7.2f", a[i][j]);
      printf ("\n");
    }
  }
}
```

Example 3.5: examples/smoothing/ex5.d

```
/* This program performs a nine-point smoothing, where each point is updated
 * to be the average of his eight neighbors.    The matrix is M x N, where
 * M is a multiple of P1, and N is a multiple of P2.    The use of asynchronous
 * variables is demonstrated here.  */

#include "dino.h"

#define max(x,y) (x > y ?  x :  y)
#define min(x,y) (x < y ?  x :  y)

#define M 12
#define N 8
#define P1 4
#define P2 4

environment node[P1:id1][P2:id2] {

  composite smooth (a, in iter)
  double /*asynch*/ distributed a[M][N] map NinePt;

                              /* ==> The NinePt mapping is just like FivePt,
                                     except that the four corner overlap
                                     elements are included in the overlap.  */

                              /* ==> Note the use of the "asynch" keyword,
                                     making communications using the matrix a
                                     non-blocking.    This program will run
                                     either synchronous or asynchronous.  */

  int iter;

  {
    int i, j, k;                          /* Looping variables */

    int home_n, home_s, home_w, home_e;   /* Boundaries of the home data */
    int copy_n, copy_s, copy_w, copy_e;   /* Boundaries of the copy data */

    /* Compute home_n, home_s, home_w, and home_e */
    home_n = max (M/P1 * id1, 1);
    home_s = min ((id1 + 1) * M/P1 - 1, M-2);
    home_w = max (N/P2 * id2, 1);
    home_e = min ((id2 + 1) * N/P2 - 1, N-2);

    /* Compute copy_n, copy_s, copy_w, and copy_e */
    copy_n = max (home_n - 1, 1);
```

```
    copy_s = min (home_s + 1, M-2);
    copy_w = max (home_w - 1, 1);
    copy_e = min (home_e + 1, N-2);

    /* Repeat the smoothing process iter times */
    for (i = 0; i < iter; i++) {

      /* Send out your data and receive it back again, if not the first
       * iteration */
      if (i != 0) {
        a[<home_n,home_s>][<home_w,home_e>]# =
          a[<home_n,home_s>][<home_w,home_e>];
        a[<copy_n,copy_s>][<copy_w,copy_e>]#;
      }

      /* Perform the computation, but only on non-edge elements */
      for (j = home_n; j <= home_s; j++)
        for (k = home_w; k <= home_e; k++) {
          a[j][k] = a[j-1][k-1] +
                    a[j-1][k] +
                    a[j-1][k+1] +
                    a[j][k-1];

                                /* ==> The computation has been split up into
                                       three separate pieces because some
                                       compiler's can't handle the large
                                       expression produced by DINO. */

          a[j][k] += a[j][k+1] +
                     a[j+1][k-1] +
                     a[j+1][k] +
                     a[j+1][k+1];

          a[j][k] /= 8;

        }
    }
  }
}

environment host {

  void main ()

  {
    double a[M][N];                 /* Input data */
    int iter;                       /* Holds the iteration count */
```

```
    int i, j;                              /* Looping variables */

    /* Set up the initial data for a[][] */
    for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
        a[i][j] = (i + 1)*(j + 1);
    for (i = 1; i < M - 1; i++)
      for (j = 1; j < N - 1; j++)
        a[i][j] = 0;

    /* Set up the variable which will contain the number of iterations */
    iter = 300;

    /* Print out the initial data */
    printf ("Initial data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%7.2f", a[i][j]);
      printf ("\n");
    }

    /* Perform the computation */
    smooth (a[][], iter)#;

    /* Printout the results */
    printf ("Result data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%7.2f", a[i][j]);
      printf ("\n");
    }
  }
}
```

Example 3.6: examples/smoothing/ex6.d

```
/* This program is a smoothing-type algorithm which computes Pascal's
 * triangle, to size M x N, by seting each element to the sum of the numbers
 * directly above and to the left.    It illustrates user-defined mappings.  */

#include "dino.h"

#define M 12
#define N 12
#define P1 4
#define P2 4

environment node[P1:id1][P2:id2] {

#include "/anchor/student/derby/doc/examples/inc/map.c"

                                /* ==> A listing of this file appears after
                                       after example 2.4.  */

  map LeftUpOverlap = [block overlap 1,0][block overlap 1,0];

                                /* ==> This is a user-defined mapping.    It
                                       sets up a mapping which divides a 2D
                                       array into blocks, and each environment
                                       receives a copy of the one element
                                       border strips to the west and north.  */

  composite smooth (a, in iter)

  double distributed a[M][N] map LeftUpOverlap;
  int iter;

  {
    int i, j, k;        /* Looping variables */
    map_var a1, a2;     /* Mapping variables */

    /* Setup the mapping variables */
    set_map_var (M, P1, id1, 1, 0, &a1);
    set_map_var (N, P2, id2, 1, 0, &a2);
    limit_map_var (1, M-1, &a1);
    limit_map_var (1, N-1, &a2);

    /* Repeat the smoothing process iter times */
    for (k = 0; k < iter; k++) {

      /* Send out your data and receive it back again, if not the first
```

```
       * iteration */
      if (k != 0) {

        if (id1 != P1-1 || id2 != P2-1)
          a[<a1.left,a1.right>][<a2.left,a2.right>]# =
          a[<a1.left,a1.right>][<a2.left,a2.right>];

                                  /* ==> The lower right environment has no
                                         data to send.    This if statement
                                         prevents a run-time warning about it.  */

        if (id1 != 0 || id2 != 0)
          a[<a1.lover,a1.rover>][<a2.lover,a2.rover>]#;

                                  /* ==> The upper left environment has no
                                         data to receive.    This if statement
                                         prevents a run-time warning about it.  */

      }

      /* Perform the computation, but only on non-edge nodes */
      for (i = a1.left; i <= a1.right; i++)
        for (j = a2.left; j <= a2.right; j++)
          a[i][j] = a[i-1][j] + a[i][j-1];

    }
  }
}

environment host {

  void main ()

  {
    double a[M][N];                    /* Input data */
    int iter;                          /* Holds the iteration count */

    int i, j;                          /* Looping variables */

    /* Set up the initial data for a[][] */
    for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
        a[i][j] = 0;
    for (i = 0; i < M; i++)
      a[i][0] = 1;
    for (j = 0; j < N; j++)
      a[0][j] = 1;
```

```
    /* Set up the variable which will contain the number of iterations */
    iter = 7;

    /* Print out the initial data */
    printf ("Initial data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%7.1f", a[i][j]);
      printf ("\n");
    }

    /* Perform the computation */
    smooth (a[][], iter)#;

    /* Printout the results */
    printf ("Result data for a:\n");
    for (i = 0; i < M; i++) {
      for (j = 0; j < N; j++)
        printf ("%5.1f", a[i][j]);
      printf ("\n");
    }
  }
}
```

4.5 Complex Examples

These final examples were included to show how somewhat more complex DINO programs are put together. They illustrate a number of features not found in the earlier examples.

Example 4.1: examples/complex/mm.d

```
/* This program does a matrix-matrix multiply, of two matrices of size
 * N x N, where N is a multiple of P, the number of environments.
 *
 * The basic algorithm is to block the first matrix
 * by rows and send a block to each processor, and
 * block the second matrix by columns and send a block
 * to each processor.    Each processor computes the
 * results for that sub-matrix for which it has data
 * and then the blocks of columns are all shifted to
 * the right one processor.  */

#include "dino.h"

#define N 32
#define P 16

environment node[P:id] {

  composite mult (in A, in B, out C)
  float distributed A[N][N] map BlockRow;       /* First matrix */
  float distributed B[N][N] map BlockCol;       /* Second matrix */
  float distributed C[N][N] map BlockRow;       /* Result matrix */

  {
    int i, j, k, l;             /* Looping variables */
    int my_first, my_last;      /* Columns of B which are mine */
    int left_first, left_last;  /* Columns of B which are my left neighbor's */
    int left, right;            /* Environment indices of my neighbors */

    /* Compute the environment indices of the nodes on the right and left */
    right = (id == P - 1) ?  (0) :  (id + 1);
    left = (id == 0) ?  (P - 1) :  (id - 1);

    /* Compute the starting and stopping indices of the data in my block */
    my_first = id * N/P;
    my_last = (id + 1) * N/P - 1;

    /* Compute the starting and stopping indices of the data in the block
```

```
     * of the node to the left */
    left_first = (id == 0) ?  ((P - 1) * N/P) : ((id - 1) * N/P);
    left_last = (id == 0) ?  (N - 1) :  (id * N/P - 1);

    /* Loop through the blocks of columns */
    for (i = 0; i < P; i++) {

      /* First, send out the data that the node on the right of me will
       * need for the next iteration */
      B[][<left_first, left_last>]# {node[left]} =
                    B[][<my_first, my_last>];

      /* Loop thru the block of C[][] that I currently have data for */
      for (j = my_first; j < my_last + 1; j++)
        for (k = 0; k < N/P; k++) {

          /* Compute the dot product of the appropriate data */
          C[j][k + ((id + i) * N/P) % N] = 0;
          for (l = 0; l < N; l++)
            C[j][k + ((id + i) * N/P) % N] +=
                    A[j][l] * B[l][k + my_first];
        }

      /* Finally, receive the data from the node on the right that I need
       * for the next iteration */
      B[][<my_first, my_last>] = B[][<my_first, my_last>]# {node[right]};
    }
  }
}

environment host {

  main() {

    int i, j;                              /* Looping variables */
    float a[N][N], b[N][N], c[N][N];

    /* Initialize the two multiplicand arrays */
    printf ("Initializing...\n");
    for (i = 0; i < N; i++)
       for (j = 0; j < N; j++) {
         a[i][j] = 0.1;
         b[i][j] = (i + j) / 10.0;
       }

    /* Preform the computation */
    printf ("Performing the computation...\n");
    mult(a[][], b[][], c[][])#;
```

```
    /* Print out the results -- because the data we used for a[][] and b[][]
     * generates the same results for each row, we only print out the first
     * row */

    printf("Results:\n");
    for (i = 0; i < N; i++) {
      if (i % 8 == 0) printf("\n");
      printf("   %6.2f", c[0][i]);
    }
    printf("\n");
  }
}
```

Example 4.2: examples/complex/redblack.d

```
/* This program uses a red-black algorithm to solve Poisson's equation,
 * given rectangular boundary conditions.    It uses the example package
 * of routines for dealing with uneven mappings in map.c.  */

/* Defined size of the processor grid */
#define P1 4
#define P2 4

/* Defined size of the data grid */
#define N1 8
#define N2 8

/* Define the mapping for the data */
map FivePoint = [block overlap 1,1][block overlap 1,1];

environment node[P1:id1][P2:id2] {

#include "/anchor/student/derby/doc/examples/include/map.c"

  composite iterate (a, in iter)

  double distributed a[N1][N2] map FivePoint;
  int iter;

  {
    home_var a1, a2;    /* Hold the looping information for this */
    int q;              /* Looping variable */
    int i, j;           /* Index variables */

    /* Set up the home variables for a */
    set_home_vars (N1, P1, id1, 1, 1, &a1);
    set_home_vars (N2, P2, id2, 1, 1, &a2);
    limit_home_vars (1, N1 - 2, &a1);
    limit_home_vars (1, N2 - 2, &a2);

    /* Go around iter times */
    for (q = 0; q < iter; q++) {

      /* Update the information, if not the first iteration */
      if (q > 0) {

        /* Send the home copy */
        a[<a1.l_main,a1.r_main>][<a2.l_main,a2.r_main>]# =
        a[<a1.l_main,a1.r_main>][<a2.l_main,a2.r_main>];
```

```
        /* Receive the data */
        a[<a1.l_over,a1.r_over>][<a2.l_over,a2.r_over>] =
        a[<a1.l_over,a1.r_over>][<a2.l_over,a2.r_over>]#;
      }

      /* Now, let's recompute the even entries */
      for (i = a1.l_main; i <= a1.r_main; i++)
        for (j = ((((i + a2.l_main) % 2) == 1) ?  1 :  0) + a2.l_main;
             j <= a2.r_main; j += 2)

          /* Do the recomputation */
          a[i][j] = (a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]) / 4;

      /* Redistributed the information */
      a[<a1.l_main,a1.r_main>][<a2.l_main,a2.r_main>]# =
      a[<a1.l_main,a1.r_main>][<a2.l_main,a2.r_main>];

      a[<a1.l_over,a1.r_over>][<a2.l_over,a2.r_over>] =
      a[<a1.l_over,a1.r_over>][<a2.l_over,a2.r_over>]#;

      /* Now, let's recompute the odd entries */
      for (i = a1.l_main; i <= a1.r_main; i++)
        for (j = ((((i + a2.l_main) % 2) == 0) ?  1 :  0) + a2.l_main;
             j <= a2.r_main; j += 2)

          /* Do the recomputation */
          a[i][j] = (a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]) / 4;
    }
  }
}
environment host {

  void main () {

    double m[N1][N2];           /* The array of data */
    int i, j;                   /* Index variables */
    int iter = 100;             /* Fake iteration variable */

    /* Set up the data in m */
    for (i = 0; i < N1; i++)
      for (j = 0; j < N2; j++)
        m[i][j] = (i + 1) * (j + 1);
    for (i = 1; i < N1 - 1; i++)
      for (j = 1; j < N2 - 1; j++)
        m[i][j] = 0;


    /* Do the computation */
```

```
    printf ("Running...\n");
    iterate (m[][], iter)#;

    /* Print out the results */
    for (i = 0; i < N1; i++) {
      for (j = 0; j < N2; j++)
        printf ("%6.2f ", m[i][j]);
      printf ("\n");
    }
    printf ("\nThat's it!\n");
  }
}
```

Example 4.3: examples/complex/row.d

```
/* This program solves Poisson's equation, by using a row-at-a-time update
 * algorithm.  */

#include <stdio.h>
#include <math.h>
#include "dino.h"

#define N 8
#define P 4

environment node[P:id] {

  double d[N], s[N];

  composite setup() {

    int i;

    /* Setup d and s so that they can be used by SolveRow */
    d[1] = 2;
    for (i = 2; i < N - 1; i++) {
      s[i] = -1 / d[i - 1];
      d[i] = sqrt (4 - s[i] * s[i]);
    }

  }

  void solve_row (U, i)

  double distributed U[N][N] map BlockRowOverlap;
  int i;

  {
    int j;
    double r = 1.0;
    double new[N];

    /* Perform the forward solve */
    new [1] = (U[i-1][1] + U[i+1][1] + U[i][0])/d[1];
    for (j=2; j<N-2; j++)
      new [j] = (U[i-1][j] + U[i+1][j] - s[j]*new[j-1])/d[j];
    new [N-2] = (U[i-1][N-2] +U[i+1][N-2] +U[i][N-1] -s[N-2]*new[N-3])/d[N-2];

    /* Perform the backward solve */
    new [N-2] = (new[N-2])/d[N-2];
```

```
  for (j=N-3; j > 0; j=j-1)
    new [j] = (new[j] - s[j+1]*new[j+1])/d[j];

  /* relaxation */
  for (j=1; j<N-1; j++)
   U[i][j] = r*new[j] + (1-r)*U[i][j];
}

/*  Parallel Algorithm For Row-Wise Red Black, with N/P Rows on each
 * processor */

composite solver (U)

double distributed U[N][N] map BlockRowOverlap;

{
  /* Compute the indices of the first and last rows in each block */
  int firstrow = id ?  (id * N/P) : 1;
  int lastrow = (id == (P-1)) ?  N-2 :  ((id+1) * (N/P) - 1);

  /* Compute the indices of the first even and first odd rows */
  int firsteven = firstrow%2 ?  firstrow + 1:  firstrow ;
  int firstodd = firstrow%2 ?  firstrow :  firstrow + 1;

  int i, k;           /* Looping variables */

  for (k=0; k<10; k++) {

    /* Update the odd rows in each block */
    for (i=firstodd; i <= lastrow; i=i+2) {
      if ((i==firstrow) && (i != 1) && (k != 0))
        U[i-1][]#;
      if ((i==lastrow) && (i != N-2) && (k != 0))
        U[i+1][]#;
      solve_row (U,i);
      if (((i==firstrow) && (i != 1)) || ((i==lastrow) && (i != N-2)))
        U[i][]# = U[i][];
    }

    /* Update the even rows in each block */
    for (i=firsteven; i <= lastrow; i=i+2) {
      if ((i==firstrow) && (i != 1) && (k != 0))
        U[i-1][]#;
      if ((i==lastrow) && (i != N-2) && (k != 0))
        U[i+1][]#;
      solve_row (U,i);
      if (((i==firstrow) && (i != 1)) || ((i==lastrow) && (i != N-2)))
        U[i][]# = U[i][];
```

```
      }
    }
  }
}

environment host {

  main () {

    float Uh [N][N];
    int i,j;

    /* Initialize the data */
    for (i=0; i<N; i++)
      for (j=0; j<N; j++)
        Uh[i][j] = i*j;
    for (i=1; i<N-1; i++)
      for (j=1; j<N-1; j++)
        Uh[i][j] = Uh[i][j] - .10;

    /* Initialize s[] and d[] */
    setup ()#;

    /* Compute the results */
    solver (Uh[][])#;

    /* Output the results */
    for (i=0; i<N; i++) {
      for (j=0; j<N; j++)
        printf("%6.2f ", Uh[i][j]);
      printf("\\n");
    }
  }
}
```

Example 4.4: examples/complex/lu.d

```
/* This program computes the LU decomposition of a matrix, with partial
 * pivoting.    The last row of A contains pivoting information - what
 * row was swapped with at each iteration of the algorithm.  */

#define P 6
#define N 14

#include "dino.h"

environment node[P:id] {

#include <math.h>
#include <stdio.h>

composite plu (a)

double distributed a[N+1][N] map WrapCol;

                                /* ==> Declares the variable a to be mapped
                                       by columns to the processors, but in
                                       a wrap fashion, so that each processor
                                       gets columns id, id+P, id+2P, ...  */

{
  int i, j, k;          /* Looping variables */
  double piv;           /* Value of the pivot */
  int pivrow;           /* Index of the pivot */
  double temp;          /* Temporary */

  double distributed m[N+1] map all;    /* Holds the multipliers */

                                /* ==> Illustrates the declaration of local
                                       distributed data within a composite
                                       procedure.  */

  /* Loop through the passes of the algorithm */
  for (i = 0; i < N; i++) {

    /* Check to see if we have the column for this iteration */
    if (i % P == id) {
      /* Select the pivot row */
      piv = fabs (a[i][i]);
      pivrow = i;
      for (j = i + 1; j < N; j++)
        if (fabs (a[j][i]) > piv) {
```

```
        piv = fabs (a[j][i]);
        pivrow = j;
      }

    /* Swap in the pivot column */
    temp = a[pivrow][i]; a[pivrow][i] = a[i][i]; a[i][i] = temp;

    /* Compute the multipliers */
    for (j = i + 1; j < N; j++)
      a[j][i] /= a[i][i];

    /* Put the pivot row into a, in preparation for sending */
    a[N][i] = pivrow;

    /* Send out the data to the multiplier vectors on all processors */
    m[<i+1,N>] = a[<i+1,N>][i];

                              /* ==> Ranges may be used in assignment
                                     statements to perform array
                                     assignments.  */

    m[<i+1,N>]# {node[] - node[id]} = m[<i+1,N>];

                              /* ==> Explicit sends are indicated by putting
                                     an environment set in braces after the
                                     "#" sign.  */

                              /* ==> Sends to every node but itself through
                                     the use of the "-" operator in the
                                     environment expression.  */
  }

  else

    /* Receive the multiplier data */
    m[<i+1,N>] = m[<i+1,N>]# {node[i%P]};

  /* Now, we're ready to do the elimination and pivoting at the same time */
  pivrow = m[N];
  for (j = id; j < N; j += P)
    if (j > i) {
      temp = a[i][j]; a[i][j] = a[pivrow][j]; a[pivrow][j] = temp;
      for (k = i+1; k < N; k++)
        a[k][j] -= m[k] * a[i][j];
    }
  }
}
```

```
}

environment host {

  double a[N+1][N];                    /* Holds the matrix */

  void main () {

    int i, j;                          /* Looping variables */

    /* Set up the test matrix */
    for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++)
        a[i][j] = (i + 1) * (j + 1);
      a[i][i]++;
    }

    /* Output an initial message */
    printf ("Starting LU decomposition...\n\n");

    /* Call the composite procedure */
    plu (a[][])#;

    /* Printout the results */
    printf ("Result is...\n");
    for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++)
        printf ("%6.1f", a[i][j]);
      printf ("\n");
    }

    printf ("\nWith pivots...\n");
    for (j = 0; j < N; j++)
      printf ("%.2f ", a[N][j]);
    printf ("\n");

  }
}
```

Example 4.5: examples/complex/block.d

```
/* This program is a fairly long example, designed to show that complex programs
 * can be written in DINO. */

/*****************************************************************************
 * Solves system of linear equations with a block bordered structure
 * (See Rosin, Schnabel, and Weaver's "Expressing Complex Parallel
 * Algorithms in Dino" p.  2).
 *
 * General Form:
 *
 *  A1          B       x1       f1
 *     A2       B       x2       f2
 *        ..         ..    *   ..      =   ..
 *           Aq Bq      xq       fq
 *  C  C  ..  Cq P       xqp      fqp
 *
 * Where:
 *
 *  A   is nxn
 *  B   is nxm
 *  C   is mxm
 *  P   is nx1
 *  x   is nx1
 *  xqp is mx1
 *  f   is nx1
 *  fqp is mx1
 *
 * Command Line:
 *
 *  block <filename>
 *
 * where <filename> is the name of the file describing the linear system
 * as specified in function read_data.
 *****************************************************************************/

#include <stdio.h>
#include <math.h>
#include <string.h>             /* ==> Standard include files can be used
                                       in the normal manner.  */

#include "dino.h"               /* ==> The directory containing dino.h is
                                       automatically searched.  */

map byRow     = [block][compress];
map byBlock   = [block];
```

```
map byElement = [block];
map byCol     = [compress][block];

#define N 2
#define M 3
#define Q 2

environment solve[M:id] {

    composite form_J( in sumCW, P )
    double distributed sumCW[M][M] map byRow;
    double distributed P[M][M] map byRow;
    {
        int i;
        for( i=0; i<M; i++ )
            P[id][i] = P[id][i] - sumCW[id][i];
    };

    composite form_b( b, in sumCz, in fqp )
    double distributed b[M]     map byElement;
    double distributed sumCz[M] map byElement;
    double distributed fqp[M]   map byElement;
    {
        b[id] = fqp[id] - sumCz[id];
    };

    composite dist_lu( A, RowPerm )
    double distributed A[M][M] map byCol;
    int distributed RowPerm[M] map all;
    {
        double distributed Mult[M] map all;
        int i, j, k, pivrow;
        double piv, temp;

        RowPerm[M-1] = M-1;
        for( k=0; k<M-1; k++ ) {
            if( k == id ) {
                /* select pivot and swap in pivot column */
                piv = A[k][k];
                pivrow = k;

                /* find largest row element in kth column (partial pivot) */
                for( i=k+1; i<M; i++ )
                    if( fabs(A[i][k]) > piv ) {
                        piv = A[i][k];
                        pivrow = i;
                    }
```

```
            /* swap row position in kth column */
            if( pivrow != k ) {
                temp = A[k][k];
                A[k][k] = A[pivrow][k];
                A[pivrow][k] = temp;
            }

            /* calculate l (multipliers) and row swap info.  */
            RowPerm[k] = pivrow;
            for (i=k+1; i<M; i++)
                Mult[i] = A[i][k] /= -A[k][k];

            /* broadcast Mult and RowPerm */
            Mult[<k+1, M-1>]#{ solve[] - solve[id] } =  Mult[<k+1, M-1>];
            RowPerm[k]#{ solve[] - solve[id] } = RowPerm[k];
        }

        else {
            /* receive broadcast here */
            Mult[<k+1, M-1>] = Mult[<k+1, M-1>]#{ solve[k] };
            RowPerm[k] = RowPerm[k]#{ solve[k] };
        }

        /* eliminate in your column greater than k */
        pivrow = RowPerm[k];
        if( id>k ) {
            if( pivrow != k ) {
                temp = A[k][id];
                A[k][id] = A[pivrow][id];
                A[pivrow][id] = temp;
            }
            for( i=k+1; i<M; i++ )
                A[i][id] += Mult[i]*A[k][id]; /* was k */
        }
        /* swap factors in lower triangular */
        else if( id<k )
            for( i=k; i<M; i++ ) {
                temp = A[i][id];
                A[i][id] = A[pivrow][id];
                A[pivrow][id] = temp;
            }
    }
};

composite dist_solve( in lu, out x, in b, in RowPerm )
double distributed lu[M][M]     map byRow;
double distributed x[M]         map byElement;
double distributed b[M]         map all;
```

```
    int distributed RowPerm[M]      map all;
    {
        double distributed y[M]     map byElement;
        double temp;
        int i, swap;

        /* permute b according to RowPerm */
        for( i=0; i<=M-2; i++ ) {
            swap = RowPerm[i];
            if( swap != i ) {
                temp = b[i];
                b[i] = b[swap];
                b[swap] = temp;
            }
        }

        /* solve ly = b */
        y[id] = b[id];
        for( i=0; i<=id-1; i++ )
            y[id] += lu[id][i]*y[i]#{solve[i]};
        y[id]#{solve[<id+1,M-1>]} = y[id];

        /* solve ux=y */
        x[id] = y[id];
        for( i=M-1; i>id; i-- )
            x[id] -= lu[id][i]*x[i]#{solve[i]};
        x[id] = x[id] / lu[id][id];
        x[id]#{solve[<0,id-1>]} = x[id];
    };
}

environment node[Q:id] {

    map slice = [block][compress][compress];

    double distributed W[Q][N][M] map slice;  /* = A^-1 B*/
    double distributed z[Q*N]     map byBlock;
                                /* ==> Distributed variables may be declared
                                       inside an environment, in which case
                                       they are accessable by all functions and
                                       composite procedures within that
                                       environment.  */

    /*
     * negate v of length l
     */
    neg_vec( v,l )
    double v[];
```

```
int l;
{
    int i;
    for( i=0; i<=l-1; i++ )
        v[i] = v[i] * -1.0;
};

/*
 * v will contain (v - x) where v & x are 1 x l
 */
sub_vec( v,x,l ) /* v <= v - x, l is length of vectors */
double v[];
double x[];
int l;
{
    int i;
    for( i=0; i<=l-1; i++ )
        v[i] = v[i] - x[i];
};

/*
 * v will contain (v - x) where v & x are 1 x l
 */
add_vec( v,x,l ) /* v <= v - x, l is length of vectors */
double v[];
double x[];
int l;
{
    int i;
    for( i=0; i<=l-1; i++ )
        v[i] = v[i] + x[i];
};

/*
 * returns (A*B) where A is mxn and B nxm and T is mxm
 */
mat_mat_mult(A,B,T)
double A[M][N], B[N][M], T[M][M];
{
    int row, col, i;
    for( row=0; row<=M-1; row++ )
        for( col=0; col<=M-1; col++ ) {
            T[row][col] = 0;
            /* inner product */
            for( i=0; i<=N-1; i++ )
                T[row][col] += A[row][i]*B[i][col];
        }
};
```

```
/*
 * Does lu decomposition on matrix A
 * Note:  o uses partial pivoting
 *        o assume A is nxn since seq_lu is only used on A to find z
 *          (see above mentioned paper)
 *        o after call to seq_lu A will contain lu decomposition
 */
void seq_lu( A, p )
double A[N][N];
int *p;
{
    double temp, mult;
    int i, diag, col, pivot;
    for( diag=0; diag<=N-2; diag++ ) {

        /* look for largest leading row element, i.e.  pivot */
        pivot = diag;
        for( i=diag; i<=N-1; i++ ) {
            if( A[i][diag] > A[pivot][diag] )
                pivot = i;
        }
        p[diag] = pivot;

        /* new pivot found?  */
        if( pivot != diag ) {
            /* information about row swaps stored in p such that
               p[diag] = pivot means that row pivot and diag were
               swapped at diagth iteration */
            for( i=0; i<=N-1; i++ ) { /* swap rows */
                temp = A[diag][i];
                A[diag][i] = A[pivot][i];
                A[pivot][i] = temp;
            }
        }

        /* elimate leading columns */
        for( i=diag+1; i<=N-1; i++ ) {
            /* compute l */
            mult = A[i][diag] /= -A[diag][diag];
            for( col=diag+1; col<=N-1; col++ )
                A[i][col] += mult*A[diag][col];
        }
    }
    p[N-1] = N-1; /* last row can't be swapped at last iteration */
};

/*
```

```
 * solves:  lu x = b
 * Note:  o lu is nxn (see seq_lu) and naturally x, b are 1xn
 *   o result found in x after call to seq_solve
 */
void seq_solve( lu, x, b, p ) /* solves lu x = b */
double lu[N][N];
double x[N];
double b[N];
int p[N];
{
    int i, col, diag;

    double y[N];
    double temp;

    /* solve ly = b */
    for( diag=0; diag<=N-2; diag++ ) {
        if( p[diag] != diag ) {
            temp = b[diag];
            b[diag] = b[p[diag]];
            b[p[diag]] = temp;
        }
    }

    y[0] = b[0];
    for( diag=1; diag<=N-1; diag++ ) {
        y[diag] = b[diag];
        for( col=0; col<=diag-1; col++ )
            y[diag] += lu[diag][col]*y[col];
    }

    /* solve ux=y */
    x[N-1] = y[N-1]/lu[N-1][N-1];
    for( diag=N-2; diag>=0; diag-- ) {
        x[diag] = y[diag];
        for( col=diag+1; col<=N-1; col++ )
            x[diag] -= lu[diag][col]*x[col];
        x[diag] /= lu[diag][diag];
    }

};

composite factor_A(in A, in f, in B)
double distributed A[Q][N][N] map slice;
double distributed f[Q*N]     map byBlock;
double distributed B[Q][N][M]  map slice;
{
    int i,r;
```

```
    int p[N];
    double tempW[N];
    double tempB[N];

    /* decompose each block */
    seq_lu(A[id], p);
    seq_solve(A[id], &z[id*N], &f[id*N], p);   /* z = A inv f */

    for (i=0; i<M; i++) {                      /* W = A inv B */
        tempB[] = B[id][][i];
        seq_solve(A[id], tempW, tempB, p);
        W[id][][i] = tempW[];
    }
};

composite form_sums( in C, out sumCW, out sumCz )
double distributed C[Q][M][N]     map slice;
double distributed sumCW[M][M]    map byRow;
double distributed sumCz[M]       map byElement;
{
    double T[M];
    double T2[M];
    double Temp[M][M];
    int i;
    int r,c;

    /* sum C*W  = J*/
    mat_mat_mult(C[id], W[id], Temp);

    Temp[][] = gsum( Temp[][] )#;

    for( i=id*M/Q; i<=id*M/Q + M/Q; i++) {
        sumCW[i][] = Temp[i][];
    }

    if( id==Q-1 ) {
        sumCW[M%Q+1][] = Temp[M%Q+1][];
    }

    /* sum c*z */
    for( r=0; r<=M-1; r++ ) {
        T[r] = 0;
        for( c=0; c<=N-1; c++ )
            T[r] += C[id][r][c]* z[id*N+c];
    }

    T2[] = gsum(T[])#;
```

```
        for( i=id*M/Q; i<=id*M/Q + M/Q; i++)
            sumCz[i] = T2[i];

        if( id==Q-1 )
            sumCz[M%Q+1] = T2[M%Q+1];
    };

    composite compute_xi( in xqp, out x )
    double distributed xqp[M]     map all;
    double distributed x[Q*N]     map byBlock;
    {
        int r,c;

        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 );
        add_vec( &x[id*N], &z[id*N], N );
    };
}

environment host{
    double A[Q][N][N], B[Q][N][M], C[Q][M][N], P[M][M],
           fqp[M], f[Q*N], xqp[M], x[Q*N], b[M], RowPerm[M], sumCW[M][M],
           sumCz[M];
    FILE *InFile;
    char filename[256];

    void read_dvector( InFile, v, size )
    FILE *InFile;
    double *v;
    int size;
    {
        int i;
        double temp;

        for( i=0; i<=size-1; i++ )
            fscanf( InFile, "%lf\n", &v[i] );
    };

    void read_dmatrix( InFile, m, row, col )
    FILE *InFile;
    double *m;
    int row, col;
    {
        int r, c;
```

```
    for( r=0; r<=row-1; r++ )
        read_dvector( InFile, (double *) (m+r*col), col );
};

void read_dmatrices( InFile, ms, size, row, col )
FILE *InFile;
double *ms;
int size, row, col;
{
    int i;

    for( i=0; i<=size-1; i++ )
        read_dmatrix( InFile, (double *) (ms+i*row*col), row, col );
};

/*
 * read_data reads an ascii file containing the linear system.
 * File must be in the following form (n, m, q must be set in program):
 *
 *  A1 <CR>
 *  ...
 *  Aq <CR>
 *  B1 <CR>
 *  ...
 *  Bq <CR>
 *  C1 <CR>
 *  ...
 *  Cq <CR>
 *  P <CR>
 *  f <CR>
 *  fqp <CR>
 *
 * Note:
 *
 *  Matrices must be listed in row order.
 *
 */
void read_data( )
void;
{
    InFile = fopen( filename, "r" );
    read_dmatrices( InFile, A, Q, N, N );
    read_dmatrices( InFile, B, Q, N, M );
    read_dmatrices( InFile, C, Q, M, N );
    read_dmatrix( InFile, P, M, M );
    read_dvector( InFile, f, Q*N );
    read_dvector( InFile, fqp, M );
};
```

```
    print_results()
    {
        int i, ii;

        printf( "Solution to System:  \n" );
        /* print x1 to xq */
        for( i=0; i<=(Q*N)-1; i++ )
            printf( "%lf\n", x[i] );
        /* print xqp */
        for( i=0; i<=M-1; i++ )
            printf( "%lf\n", xqp[i] );
    };

    void main( argc, argv )
    int argc;
    char *argv[];
    {
        /* parse command line */
        if( argc > 1 )
            strcpy(filename,argv[1]);
        else {
            printf( "ERROR: No file name specified.\n" );
            return;
        }
        strcpy(filename,"input.dat");

        read_data();
        factor_A( A[][][], f[], B[][][])#;
        form_sums( C[][][], sumCW[][], sumCz[])#;
        form_J( sumCW[][], P[][] )#;
        form_b( b[], sumCz[], fqp[] )#;
        dist_lu( P[][], RowPerm[])#;
        dist_solve( P[][], xqp[], b[], RowPerm[])#;
        compute_xi(xqp[], x[])#;
        print_results();
    }
}
```

Example : examples/complex/input.dat (data file for block.d)

```
1.0 3.0
2.0 9.0
8.0 6.0
1.0 7.0
1.0 8.0 9.0
5.0 4.0 3.0
1.0 1.0 2.0
2.0 1.0 3.0
9.0 4.0
8.0 5.0
3.0 7.0
6.0 3.0
4.0 2.0
5.0 1.0
2.0 6.0 1.0
1.0 5.0 2.0
7.0 6.0 4.0
5.0 4.0
2.0 3.0
5.0 8.0 7.0
(data file for block.d)
```

5 How to use DINO

5.1 Overview

The DINO compiler takes a program file written in the DINO language, and converts it into one or more C programs, one for each environment structure in the DINO program. In addition, DINO is normally set up to move these C files to the appropriate parallel machine and compile them into executables which can be run on that machine. The DINO compiler will run on a sun3 or a sun4. The following sections describe how to invoke the compiler, how to control the default options by using environment variables, and how to manually control the copying and subsequent compilation of the intermediate C files.

5.2 Setting up

In order to use DINO, you must have the correct path to the DINO compiler. DINO uses the sun convention that binaries are kept in architecture-specific directories such as: `~dino/bin/sun3` or `~dino/bin/sun4`.

For this to work correctly, you must have set the environment variable "`ARCHTYPE`" to the appropriate architecture. This can be done using the command

```
setenv ARCHTYPE `arch`
```

Once this is done, the necessary DINO directory can be added to your path with the command

```
set path=($path <path>/bin/${ARCHTYPE})
```

where <*path*> is the full path name to the master DINO directory (if you don't know where that is, see your system administrator).

These commands should be placed in your .cshrc or .login file, after the path is set up. If this is not done correctly, DINO will fail in mysterious ways.

In addition, a similar "`ARCHTYPE`" variable must be set up on the parallel machine which you're going to use. This is used so that the DINO compiler knows what libraries to link with on the target machine. It also controls options passed to the C compiler. A command such as

```
setenv ARCHTYPE <machtype>
```

should be inserted in your .cshrc file (not the .login file – because the .login file isn't looked at when rsh is used) on the target machine, where <*machtype*> is either `iPSC1`, `iPSC2`, or `i860` as appropriate.

Finally, if you want to use the facility for copying the intermediate C files to the target machine and compiling them there, you need to have appropriate permissions set up on the parallel machine to allow for this. Normally, this means having appropriate entries in your ".rhosts" file on the parallel machine. Consult your system administrator for assistance.

5.3 Invoking the Compiler

The dino compiler is invoked by using the following command:

```
dino [-<machine> ] [-w] [-c] [-n] [-e] [-l] [-o] [-p]
[-D] [-P] [-Q] [-h[elp]] [-s <suffix>] [-C <num>]
[-d <directory>] [-u <user> [-I <directory>]<file>
```

If `dino` is typed alone (or with the -D or -h options), then a list of options is printed out. Otherwise, DINO compiles the given file (which must have the `.d` extension) into one or more C files, and may optionally send these C files to the parallel machine, and compile them. Options to the compiler which are boolean (for example, -w) may be turned off by inserting a "^"; for example, the option "-^w" turns on warning messages. Compiler options may be listed an any order, and are described below:

`-<`*machine*`>` Generates code for the given parallel machine. This option sets up all of the default options for the given machine. The appropriate *<machine>* to use should be given to you by the system administrator. For example, if you want to compile a program `prog.d` for a machine named `mach`, the appropriate command would be

```
dino -mach prog.d
```

`-D` This option causes DINO to list each of the parallel machines defined, along with the options specified by them. Any of these options can be overridden by a command line option.

`-w` Suppresses the printing of warning messages from the DINO compiler.

`-c` Causes DINO to print out comments from the compiler. There are no useful comments produced by the DINO compiler at this time. This option is present for future versions of the compiler.

`-n` Causes the DINO compiler to print out notes from the compiler. There are no useful notes produced by the DINO compiler at this time. This option is present for future versions of the compiler.

`-e` Causes the DINO compiler to print out errors in the order in which the compiler detects them, rather than in lexical order. This is useful because the compiler will occasionally report a cascaded error before the actual error.

`-o` Causes the compiler to use the old suffix model, producing files of the form *<env><suffix>*`.c`, where the default *<suffix>* is `D1`, `D2`, `D8`, `S1`, `S2`, or `G`, depending on the kind of machine you're compiling for. Normally, files are named *<program>*`.`*<suffix>*`.`*<env>*`.c`, where *<program>* is the name of the DINO program being compiled (without the `.d` suffix), *<env>* is the name of the environment, and the default *<suffix>* is the name of the machine being compiled for. The *<suffix>* can be changed by using the `-s` option.

`-s <suffix>` Sets the <*suffix*> used to construct filenames for the intermediate C files. See the -o option.

`-P` This option causes DINO to stop just after generating the intermediate C files, without copying them to the parallel machine.

`-p` This option causes DINO to generate the intermediate C files, and copy them to the parallel machine, but not compile them. This option is useful when custom flags must be sent to the C compiler on the parallel machine.

`-Q` Prevents DINO from producing any output, except from error messages. Notice that this option has no effect on the C compiler invoked by DINO so if the C compiler prints out any messages (like the C compiler on the iPSC1), they will still be printed out.

`-h[elp ]` Prints out a list of available options, with brief descriptions. This action is also taken is no filename is given.

`-C <num>` Sets the dimension of the cube being compiled for. It is up to you to insure that this many nodes are actually provided when the program is run.

`-d <directory>` Tells DINO where to put your executable files. This option accepts either relative or absolute pathnames. The system administrator has defined a default "base" directory when DINO was set up. The relative path names are always with respect to this "base" directory. In a typical installation, the "base" directory would be the directory you are in if the parallel machine is a local machine, and your home directory on the parallel machine if the parallel machine is a remote machine. The "`-d`" option interprets a "`.`" which is the first character in a pathname as the current directory (this only works if the file system you're using is cross mounted). So, if you want your final files to appear in a subdirectory (of your current directory) named "test", you would use the command

```
dino -mach -d ./test row.d
```

where "mach" is a parallel machine which mounts your files. If the directory does not exist, DINO will create it automatically.

`-u <user>` Use the "`-u`" option if you want to execute commands on a remote parallel machine as someone other than the default user (usually, the default user will be the same as the user of the local machine). Note that the <*user*> must have a login on the parallel machine. Typing "`-u ~`" sets the user on the parallel machine to the the same as the current user on the local machine. If the parallel machine is the same as the local machine (for example, a simulator or a distributed sun network) then "`~`" is the only user that the "`-u`" option will accept.

`-I <directory>` This option tells DINO what directory to find include files in. More than one "-I" option may be used.

5.4 Using Environment Variables

Anything (except the name of the DINO source file) which can be set from the command line can also be specified as an environment variable. This gives the user considerable control over the default settings of the DINO compiler. In addition, many of the defaults can be set differently for different parallel machines (assuming that your DINO compiler has been configured form more than one parallel machine).

In the following descriptions, <*name*> stands form the name of a particular parallel machine. For example, if you have an iPSC2 named "mach", then D<*name*>size for mach would be "Dmachsize". Obviously, the use of <*name*> in an environment variable indicates that a separate default can be set for each parallel machine. The various environment variables are:

Dmachine The default parallel machine that will be used if no machine is specified on the command line.

Dinc List of directories which DINO will look in for include files. Corresponds to the "`-I`" command line option. Multiple directories are separated by spaces, tabs, or newlines.

D<*name*>size The default cube size for the parallel machine <*name*>. Corresponds to the "`-C`" command line option.

D<*name*>suf The default suffix for the parallel machine <*name*>. Corresponds to the "`-s`" command line option.

D<*name*>dir The default directory on the parallel machine <*name*>. Corresponds to the "`-d`" command line option.

D<*name*>usr The default user on the parallel machine <*name*>. Corresponds to the "`-u`" command line option.

D<*name*>opt A list of one or more boolean options for the parallel machine <*name*>. The "`-`" characters from the command line are omitted, but "`^`" may be used to negate an option. For example, the string "`o^nQ`" causes the compiler to use the old suffix model, not print notes, and produces quiet output.

5.5 Doing Things Manually

DINO depends on the UNIX commands "rcp" and "rsh" to copy intermediate C files to the remote parallel machine and to compile them there. If these commands don't work on your system, then you will have to do some or all of this manually. In addition, DINO does not allow you to specify options to the C compiler on the parallel machine other than the default options used by DINO. If you want more control over the C compilation, you will have to do this step manually.

You can stop DINO after the intermediate C files are created (the "-P" option), or after the intermediate C files are copied to the parallel machine (the "-p" option); see Section 5.3.

You can have DINO compiler the intermediate C files on the parallel machine with the command

`dino2` <*filenames*>

where <*filenames*> is a list of all the intermediate C files. DINO will expect those filenames to follow the default conventions for DINO suffixes. The "`dino -p`" command puts files with the appropriate names on the parallel machine. The "`dino -P`" command creates files with the correct names in the current local directory, with the exception of files for the iPSC1 and iPSC2 when the new suffix model is used. In that case, the file name in the current directory might be "row.mach.node.c", but DINO expects to see "node.c" on the parallel machine (Intel's machines do not support filenames longer than 14 characters).

Of course, you can write your own Makefile to control compilation on the parallel machine.

6 DINO Hints

6.1 Known Compiler Bugs

The following is a list of known compiler bugs. These will hopefully be fixed in a future version of DINO.

1. You cannot do a send to an explicitly specified destination where the value is a constant; the C programs generated by the DINO compiler will not compile. Example:

   ```
   int distributed A[10] map Block;

   a[5]# {node[1]} = 1;
   ```

   will not work.

2. Scalar environment declarations (other than the host) do not work now. Examples:

   ```
   environment single { . . . }
   environment one[1] { . . . }
   ```

   won't work properly.

3. Reduction functions where the reduction is done over only one environment do not work properly. For example:

   ```
   A = gsum(B)# {node[id]};
   ```

   will not work.

4. Statements that mix range assignments with multiple assignments do not work properly. The statement

```
A[<5,12>]# {node[1]} = A[<5,12>] = B[];
```

will not run correctly. If you use a range assignment, put only one assignment in a statement.

5. It turns out that there are some unforeseen limitations on the way you can specify ranges in send/receive pairs. While most pattern combinations work, you cannot specify one half of a pair as a single and the other half as a partial range. That is, the send/receive pair

```
A[b][]# = . . .
. . . = A[<c,d>][]#;
```

will not work properly. However,

```
A[][]# = . . .
. . . = A[<c,d>][]#;
```

or

```
A[b][]# = . . .
. . . = A[][]#;
```

will work.

6. Three dimensional (or higher) environment structures do not work properly.

7. If you are constructing your own mapping functions, the primitive operation "align" doesn't work properly in the case of a 1 x N or N x 1 data structure (there are some special cases where this kind of data structure is useful.) Rather than using "align", transpose the data structure.

8. DINO is quite good at only sending or receiving the proper data elements (in the completely implicit case) even if you overspecify the data to be sent or received. However, there is one obvious case that will not be handled properly. If you have a data structure which is partitioned (i.e., not distributed "all"), and you send it with "`A[][]# = . . .`" or receive it with "`. . . = A[][]`", it will not work properly.

9. We have had some trouble with complex expressions on some machines. DINO translates sends and receives into more complex expressions. Sometimes the C compilers on target machines can't cope with them. For example:

```
A[idx][idy]# = ((right?A[idx][0]:A[idx][idy-1]#) +
                (left?A[idx][N-1]:A[idx][idy+1]#) +
                (top?A[0][idy]:A[idx-1][idy]#) +
                (bottom?A[N-1][idy]:A[idx+1][idy]#) / 4);
```

translates into a C expression that won't compile on our iPSC1 and causes the C compiler on our sun3 (when we are using the iPSC1 simulator) to make an error in the expression evaluation. Beware of anything approaching this complexity.

10. Certain sends and receives do not work properly. They are those sends and receive in which

    1) you explicitly specify the source or destination environment,
    2) the data object (with any message header generated by the compiler) is greater than the DINO's message buffer size (16K on the iPSC1, 32K on the other cubes), and
    3) the compiler thinks that storage for the whole piece of data (of the correct type) doesn't exist on the node.

    So, for example, if A[100][100] is a float and B[100][100] is a double distributed which is not mapped entirely to your node, the statement

    ```
    B[][]# {node[0]} = A[][];
    ```

    will not send out the whole piece of data (it only sends the first message buffer). The place where this gets tricky is that if B[100][100] is a double distributed that is mapped to your node, then if you write

    ```
    B[][]# {node[0]};
    ```

    the compiler won't figure out that storage exists on your node and thus doesn't do the receive properly. However,

    ```
    B[][] = B[][]# {node[0]};
    ```

    will work properly.

11. If you use a range as a parameter to an ordinary C function, the DINO compiler will not catch it. Instead, it generates C code with no parameter in that position. If there is more than one parameter to that function, the code will be syntactically incorrect and the resulting C compilation will fail.

12. DINO allows you to specify a range in a larger data array as the actual parameter so long as the dimension and size(s) of the range is the same as the dimension and size(s) of the formal parameter. However, if the dimension of the larger data structure is higher than the dimension of the formal parameter, this will not work properly in most cases.

    For example, the composite procedure call `comp (A[<3,6>][2])#` will not work if the formal parameter is declared to be a one dimensional vector of length 4. If the formal parameter is two dimensional (4 x 1), it will work properly.

6.2 What the Compiler Doesn't Do (run-time checking)

In general, the DINO compiler and run-time library do very little in the way of run-time checking. The only errors which are checked for at run-time are the following:

- **Implicit send called for data that has no copies**

  This message indicates that the DINO program tried to send data which had no copies on other environments. Usually, this occurs either because the wrong mapping function was chosen, or because the wrong piece of the data structure is being sent (incorrect ranges are a common cause). The program continues running after this warning message.

- **Implicit receive called for data that is only home data**

  This message indicates that a receive has been done on data which is home data, and thus has no home to receive from. This usually occurs either because the wrong mapping function was chosen, or because the wrong piece of the data structure is being sent (incorrect ranges are a common cause). The program continues after this warning.

These are the only errors which are caught by DINO. The following errors, while not caught by the DINO run-time system, should be watched for carefully:

1. Ranges in which the left number is larger than the right. An example of this is `A[<4,3>]#`. These errors can creep in when writing programs designed to work on arbitrary sized data structures; if the size of arrays becomes smaller than the number of processors, these "backwards ranges" can occur.

2. Mismatched sends and receives cannot be caught by DINO. If your program seems to infinite loop, chances are that your sends and receives are not matching up properly.

3. DINO does no range checking. That is, references to distributed arrays that are less that zero or greater than the size of the array will not be flagged in DINO.

4. DINO does not check that a local reference to a distributed data structure is actually mapped to that particular environment.

5. DINO does not check those parts of range assignments that can only be known at run time. Thus, you can assign ranges that are not the same sizes (with, of course, unpredictable results.)

6.3 Performance Hints

The following is a list of suggestions on how to achieve the best performance from DINO. The list has been kept short, and only discusses DINO-specific issues.

1. Use of environment structures which have more environments in them than your cube has processors are implemented by running multiple processes on each node of the cube. As a result, performance is significantly impaired. If the total number of environments (in all environment structures) is less than or equal to the number of nodes, then they are distributed so that each node gets at most one environment.

2. When performing remote references, it is more efficient to do block communications than to send and receive individual values, as DINO sends one or more messages for each communications statement.

3. In addition, it is somewhat more efficient to include sends or receives to multiple environments in a single statement, where DINOallows this. This can occur in implicit communications, when a data structure has multiple overlaps, and in explicit communications.

4. Since composite procedure calls must distribute their parameters (and retrieve results), minimize the use of parameters. Data which should remain in the same environment structure over multiple composite procedure calls should be declared as a variable within that environment.

5. When referencing a distributed array, DINO allows the user to refer to the global name of an object, even when only a piece of that object is present in the environment referring to the object. As a result, DINO must translate the global indices provided by the program into local indices. This process can be somewhat time-consuming, particularly for **wrap** mappings. C compilers with sophisticated optimization routines will remove many of these computations from within loops, but we haven't seen any C compilers that do this for the supported machines.

6. When doing sends and receives, it is faster if the data to be sent (or received) from each environment resides in a contiguous piece of memory. In some cases, interchanging the axes of a data structure (in order to make the communicated pieces contiguous) can have dramatic effects on execution time.

7 Installation Notes

If your copy of DINO has not already been installed, this section of the manual should show you how to accomplish that. If you have problems, feel free to contact us at `dino@cs.colorado.edu`.

The DINO compiler is divided into two parts: A front end, which runs on a Sun workstation, and a back end, which runs on the parallel machine (this could be the same machine in the case of a simulator or sun network).

7.1 Installing the Front End

This section describes how to install the front end of DINO and how to configure it for your parallel machine(s).

7.1.1 Where to Install DINO

DINO is designed to be installed with its own user name (dino) in it's own directory. It is possible to do it in other ways, but some of the procedures in this manual will be incorrect for other kinds of installations. If you need to do a non-standard installation, contact us for more information.

Once you have the "dino" username set up, change to dino's home directory. You should either have a tape with DINO on it, or a compressed, tar'd file you obtained via FTP. If you have a tape, untar it in DINO's home directory. If you have a file, move it to DINO's home directory, uncompress it, and then untar it.

7.1.2 Reading the README file

Before continuing with the installation, be sure to read the README file in order to check for changes to this installation procedure.

7.1.3 Setting up the DINO System Files

DINO assumes that the shell is csh. If you have to use sh, some of the procedures here won't work exactly as written.

DINO uses the convention that binary files are kept in architecture specific directories, such as dino/bin/sun3. In order to make this work, DINO must know what kind of machine it's running on, both on the Sun workstation, and the parallel machine being used. On the Sun, insert the command

```
setenv ARCHTYPE `arch`
```

into your .cshrc file. On the parallel machine, a commands such as

```
setenv ARCHTYPE iPSC1
```

must be inserted in the .cshrc file. The other architectures would use "`iPSC2`" and "`i860`".

Notice that if your file system is cross mounted, then the .cshrc file must determine if it's logged onto the parallel machine or a sun workstation, and set ARCHTYPE appropriately.

Once ARCHTYPE is set up, the DINO path is added with:

```
set path=($path <DINO>/bin/${ARCHTYPE})
```

where <*DINO*> is the full path name to the master DINO directory. This must be done in your .cshrc file, because commands run via "rsh" don't look at the .login file.

Several example .cshrc files are provided in the `~dino/setup` directory - cshrc.sun, cshrc.ipsc, and cshrc.both (used for crossmounted home directories).

If your site has a Pascal compiler, make sure it is in the path. Also, the .rhosts file on the parallel machine must be set up so that DINO can run "rcp" and "rsh" on the sun to copy files to the parallel machine and run commands there.

Finally, create a .forward file in the dino home directory which contains the line "dino@cs.colorado.edu". This will allow users to send mail to "dino" on their Suns, and it will be forwarded to us.

7.1.4 How to Generate the Front End

Once you have modified your .cshrc file, log out and log in as dino. Then simply type:

```
make compiler
```

This should install the front end of the DINO compiler. If you have both Sun3 and Sun4 workstations, you will have to do this once for each kind of workstation.

7.1.5 Configuring the Front End

DINO is actually a Bourne shell script, "dino", which calls a number of executable programs, including the actual compiler. In order to allow DINO to work with a large number of parallel machines, DINO reads the configuration file "dino.init" every time it starts up. This section describes how to construct this file.

We have provided an example configuration file, "dino.init.example". Copy this file into "dino.init", and edit it. Then make the following changes.

1) Set the environment variable `Dhome` to the home directory for dino.

2) Set the environment variable `Dinc` to the appropriate list of include directories. Multiple directories should be separated by spaces, tabs, or newlines.

3) Define each parallel machine you wish to support. Notice that multiple "virtual" machines can be defined for each real machine. For each virtual machine, five words are added to the `Dmachs` variable. The exact details on how to do this are in comments in the "dino.init" file. In addition, the specialized variables `sD<`*name*`>suf` and `sD<`*name*`>opt` can be used to specify additional defaults for the virtual machine being defined. Setting `sD<`*name*`>suf` allows you to set the default suffix for the parallel machine `<`*name*`>` (see the "`-s`" command option in Section 5.3). Setting `sD<`*name*`>opt` allows you to set one or more of the boolean options for the parallel machine. The format is the same as for `D<`*name*`>opt` (see Section 5.4).

4) Define the `sDmachine` variable to be default virtual machine (which will be used if the `dino` command line doesn't specify a machine).

5) Make sure all the objects you defined are listed in the final `export` statement.

In order to make configuration easier, the DINO compiler provides an option which will print out the current configuration. In order to do this, type the command

```
dino -D
```

7.2 Installing the Back End

This section describes how to install DINO's back end onto a target parallel machine. This process must be repeated for each parallel machine you plan to use.

7.2.1 Where to Install DINO

As on the Suns, DINO is designed to be installed in its own home directory on the parallel machine. If your files are cross-mounted to the parallel machine(s) and the same directory is DINO's home directory on both the Sun and the parallel machine(s), then you are done with this part. Otherwise,

1) Create a dino account on the parallel machine.

2) Copy `~dino/<`*machine type*`>` from the Sun to `~dino/Makefile` on the parallel machine, where `<`*machine type*`>` is either `iPSC1`, `iPSC2`, or `i860`.

3) Log in as "dino" on the parallel machine, and type

```
make directories
```

4) Copy the contents of the directories

   `~dino/<`*machine type*`>/Makefile`,
   `~dino/source/inc`,
   `~dino/source/library`, and
   `~dino/source/dino2`

   to the directories of the same name on the parallel machine.

7.2.2 Setting up the DINO System Files

If you followed the instructions in Section 7.1.3, then this is already done.

7.2.3 How to Generate the Back End

Log in on the parallel machine as "dino", change to the `<`*machine type*`>` directory, and type

```
make all
```

7.2.4 Configuring the Back End

As with the front end of DINO, the back end is controlled by a Bourne shell script ("dino2"). This script uses two other shell scripts to determine your particular setup: "dino2.init" and "dino2.`<`*name*`>`", where `<`*name*`>` is the virtual machine name. Both of these files should be in dino/bin/local on the parallel machine. Example files "dino2.init.example" and "dino2.`<`*machine type*`>`.example" have been provided as a starting point.

To set up the "dino2.init" file, copy the example file to "dino2.init". Then make the following changes:

1) Set the environment variable `Dhome` to the home directory for dino.

2) Set four default parameters. These are only used when `dino2` is invoked directly from the command line. These variables are described in the example file "dino2.init.example"

The "dino2.<*name*>" file contains information on options and libraries used to compiler the intermediate C files into executables. We have provided examples for "plain vanilla" cubes under the name "dino2.<*name*>.example". Each file sets eight environment variables, which are described in the example files.

If you find that the variables provided cannot be made to work for the way you want to set things up, contact us.

7.3 Complex Sun/Parallel Machine Interactions

In some cases, a parallel machine is configured in a way which doesn't "fit" the DINO script model. For example, we've encountered an iPSC2 for which "rsh" and "rcp" commands must be done from one particular Sun. To allow for these kinds of special cases, we allow you to define a particular parallel machine as "special" (see the "dino.init" file). This will cause DINO to ignore the normal actions after the intermediate C files are generated, and instead, attempt to execute the Bourne shell code found in the "dino.special" file in `~dino/bin/local`

We've provided a skeleton file "dino.special.example", which shows how to have special code for more than one machine. If you need to use this feature, you are on your own. If you need additional information, feel free to contact us.

7.4 Updates

Each DINO installation should include a "SPECS" file in the DINO home directory on the Sun. This file describes your particular configuration, and the date of distribution. If you contact us with this information, we can supply you with an update that includes only changes since your last installation. Alternatively, you can obtain an entirely new installation, and simply reinstall it.

In order to install a new update of DINO, the same procedure is used as for the initial installation, except that you should not have to reconstruct the configuration files.

A EBNF Specification for DINO Extensions to C

This EBNF *specification uses the following notation:*

()	*precedence grouping*
[]	*optional*
*	*zero or more*
+	*one or more*
|	*alternatives*
::=	*is replaced by*
||	*one or more separated by*

Terminals are enclosed in single quotes
Non-terminals are in capitals

Program:

```
PROGRAM ::= ( ENVIRONMENT | MAPPING_FUNCTION | DATA_DEFINITION )+.
```

Environments:

```
ENVIRONMENT ::=
    'environment' IDENTIFIER DIMENSION* '{' EXTERNAL_DEFINITION+ '}'.

DIMENSION ::= '[' EXPRESSION [ ':' IDENTIFIER ] ']'.

EXTERNAL_DEFINITION ::=
    FUNCTION_DEFINITION | DATA_DEFINITION | MAPPING_FUNCTION.
```

Composite Procedure Declarations:

```
FUNCTION_DEFINITION ::=
    'composite' IDENTIFIER '(' [COMP_PARAMETER_LIST] ')'
    FUNCTION_BODY.

COMP_PARAMETER_LIST ::= ( ['in' | 'out'] IDENTIFIER ) || ','.
```

Composite Procedure Call:

```
STATEMENT ::=
    IDENTIFIER '(' [ EXPRESSION_LIST ] ')' '#' [ '{' ENV_EXP '}' ]
    [ '::' STATEMENT ].

ENV_EXP ::= EXPRESSION.
```

Distributed Data Declaration:

```
DECLARATOR ::=
    [ 'asynch' ] 'distributed' DECLARATOR
    ( '[' CONSTANT_EXPRESSION ']' )+ MAPPING.

MAPPING ::= 'map' ( 'all' | IDENTIFIER | (( IDENTIFIER IDENTIFIER )
    || 'map' )).
```

Distributed Data Use:

```
EXPRESSION ::=
    PRIMARY '#' [ '{' ENV_EXP [ 'from' EXPRESSION ] '}' ].

ENV_EXP ::= 'caller' | EXPRESSION.
```

Subarrays and Ranges:

```
PRIMARY ::= PRIMARY ( '[]' | '[<' EXPRESSION ',' EXPRESSION  >]' ).
```

Mapping Functions:

```
MAPPING_FUNCTION ::=
    'map' IDENTIFIER '=' ( '[' MAP_TYPE [ALIGN] EXPANSION* ']' )+.

MAP_TYPE ::= 'all' | 'compress' | BLOCK_MAP | WRAP_MAP.

BLOCK_MAP ::=
    'block' [ 'overlap' [ EXPRESSION ]] [ 'cross' 'axis' EXPRESSION ].

WRAP_MAP ::= 'wrap' [ EXPRESSION ].

ALIGN ::= 'align' 'axis' EXPRESSION.

EXPANSION ::= 'expand' 'axis' EXPRESSION.
```

Reduction Functions:

```
PRIMARY ::=
    REDUCTION '(' EXPRESSION [ ',' EXPRESSION ] ')' '#'
    [ '{' ENV_EXP '}' ].

REDUCTION ::=
    'gsum' | 'gprod' | 'gmin' | 'gmindex' | 'gmax' | 'gmaxdex'.
```

A Note on C Syntax:

In standard C, a PROGRAM consists of one or more EXTERNAL_DEFINITIONs, each of which can be a FUNCTION_DEFINITION or a DATA_DEFINITION. DINO complicates this somewhat by adding environments.

C DATA_DEFINITIONs are built from DECLARATION_SPECIFIERs (e.g., int, struct A int B; char C, etc.), followed by DECLARATORs, followed by INITIALIZERs. The DECLARATOR is an IDENTIFIER, optionally nested inside one or more "*", "()", or "[]" to designate respectively a pointer to, a function returning, or an array of. DINO allows a single distributed declaration in this nesting to designate a particular DATA_DEFINITION as distributed.

Ordinary C FUNCTION_DEFINITIONs are built by concatenating an optional TYPE_SPECIFIER, a FUNCTION_DECLARATOR (which includes the parameter list) and a FUNCTION_BODY (which includes the parameter declarations). DINO uses a simpler syntax for composite procedures.

Certain types of C EXPRESSIONs are called PRIMARYs to distinguish then from the larger class of all EXPRESSIONs.

Index

References

Rosing, M., Schnabel, R., and Weaver, R. The DINO parallel programming language. Department of Computer Science Technical Report CU-CS-457-90, University of Colorado at Boulder, 1990. To appear in *Journal of Parallel and Distributed Computing.*

Rosing, M., and Schnabel, R. An overview of DINO - a new language for numerical computation on distributed memory multiprocessors. In Rodrique, G. (Ed.). *Parallel Processing for Scientific Computation.* SIAM, 1989, pp. 312-316.

Rosing, M., Schnabel, R., and Weaver, R. DINO : summary and examples. In Fox, G. (Ed.). *The Third Conference on Hypercube Concurrent Computers and Applications.* ACM, 1988, pp. 472-481.

Rosing, M., Schnabel, R., and Weaver, R. Expressing complex parallel algorithms in DINO. *Proc. of Fourth Conference on Hypercubes, Concurrent Computers, and Applications,* Vol. 1 (1989), 553-560.

Rosing, M., Schnabel, R., and Weaver, R. Massive parallelism and process contraction in DINO. In Dongarra, J., Messina, P., Sorensen, D.C., and Voigt, R.G. (Eds.), *Parallel Processing for Scientific Computation,* SIAM, 1990, pp. 364-369.

Rosing, M., and Weaver, R. Mapping data to processors in distributed memory computers. *Proc. of Fifth Distributed Memory Computing Conference,* IEEE Press, 1990, pp 884-893.