Expressing Complex Parallel Algorithms in DINO

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

CU-CS-430-88 March 1989

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

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

To appear in *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, Monterey, CA, Mar. 1989.

EXPRESSING COMPLEX PARALLEL ALGORITHMS IN DINO

Matthew Rosing, Robert B. Schnabel, and Robert P. Weaver
University of Colorado at Boulder

Abstract. DINO is a language, consisting of additions to C, for expressing parallel numerical programs on distributed memory multiprocessors. Its goal is to incorporate the high level features of parallel algorithms, such as the mapping of data and procedures to processes, into the language, and have low level operations such as interprocess communication and process control result implicitly. This paper describes the use of DINO to program a moderately complex, multiple-phase parallel algorithm, the parallel solution of block bordered systems of linear equations. This example illustrates the suitability of DINO for such computations but also points to some potential improvements to DINO that could be made.

1. Introduction. DINO (DIstributed Numerically Oriented language) is a language for writing numerical programs for distributed memory multiprocessors. It consists primarily of standard C augmented by several high-level parallel constructs. Its goal is to make the programming of structured, parallel numerical algorithms natural and easy, while avoiding the low -level details involved in managing processes and interprocess communication. Descriptions of DINO can be found in [5, 6]. Related work can be found in [3] [4] and [2].

We believe that DINO gives the programmer who is trying to express parallel scientific computations a relatively natural way to write such programs so that they are easily written and understood and execute reasonably efficiently. But, it is not yet clear that DINO is as useful for large, complicated problems as we have found it to be for smaller examples. For good reasons, much of the initial development work and most of the explanation of a new language is done with small, manageable examples. In this paper, we describe a considerably larger example than we have in the past, and discuss the strengths and weaknesses of DINO that this example illustrates.

Section 2 of this paper gives a brief informal description of DINO. Section 3 presents a DINO program for solving block-bordered systems of linear equations, and provides commentary on the features of DINO that this example utilizes. Section 4 gives a brief critique of DINO's effectiveness in expressing this algorithm.

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

The programmer defined structure of environments is the underlying parallel model in DINO. Typically, the structure is a single or multiple dimensional array, thus defining a virtual parallel machine with this topology. Each environment in a given structure consists of (identical) data structures and procedures, and is similar to a process. An environment may contain multiple procedures, but only one procedure in an environment may be active at a time. As the example in Section 3 will illustrate, a DINO program may contain more than one structure of environments.

The key to constructing a parallel DINO program is the mapping of data structures to the structure(s) of environments. DINO encourages the programmer to think of data structures that are operated on concurrently as single data structures that are broken up, distributed to multiple processors for computation, and then reassembled. The programmer does this by declaring a data structure as distributed data and providing its mapping to the structure of environments. Individual data elements may be mapped to only one environment, or they may be mapped to multiple environments. These mappings determine how the environments will access and share the data. DINO provides an extensive library of standard mapping functions, and the user can create additional mapping functions.

Distributed variables are the key to making interprocess communication in DINO natural and implicit. One way this occurs is by using distributed variables as parameters to composite procedures; the parameter is distributed to or collected from the appropriate environments according to its mapping function. The second way is to use a distributed variable in an expression. A local access to a distributed variable, which uses standard syntax, affects just the local copy and is the same as any standard reference to a variable. A remote access, which uses a # following the variable name, is used to generate interprocess communication. A remote assignment to a distributed variable generates a message that is sent to other environment(s) to which that variable has been distributed, while a remote read of a distributed variable will receive such a message and update the variable's value. In the default, synchronous mode, a remote read overwrites the local copy of the distributed variable with the first value that has been received since the last remote read; if no new value is present, it blocks until one is received. An asynchronous, nonblocking variant also is provided but is not discussed in this paper.

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

3. Example. The example program we describe computes the solution to a set of linear equations ($Ax = f$) where the system has a block bordered structure of the form

$$\begin{bmatrix} A_1 & & & & B_1 \\ & A_2 & & & B_2 \\ & & \ddots & & \vdots \\ & & & A_q & B_q \\ C_1 & C_2 & \cdots & C_q & P \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_q \\ x_{q+1} \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_q \\ f_{q+1} \end{bmatrix} \tag{3.1}$$

Here $A_i \in R^{n\times n}$, $B_i \in R^{n\times m}$, $C_i \in R^{m\times n}$, $P \in R^{m\times m}$, $x_i, f_i \in R^n$ $(1\le i\le q)$, $x_{q+1} f_{q+1} \in R^m$. Block bordered systems arise very commonly in parallel and sequential algorithms for solving problems in circuit design, structural analysis, and other areas (see e.g. [1]).

The solution to (3.1) is computed from the equations

$$A_i x_i + B_i x_{q+1} = f_i \quad 1\le i\le q$$

and

$$\sum_{i=1}^{q} C_i x_i + P x_{q+1} = f_{q+1}$$

Solving the first equation for x_i and substituting this into the second gives

$$(P - \sum_{i=1}^{q} C_i A_i^{-1} B_i) x_{q+1} = f_{q+1} - \sum_{i=1}^{q} C_i A_i^{-1} f_i$$

from which x_{q+1} can be derived, and put back into the first equation to find the x_i's.

The algorithm for finding the solution consists of five steps:

1) factor each A_i,
   compute $z_i = A_i^{-1} f_i$, and
   compute $W_i = A_i^{-1} B_i \quad 1 \le i \le q$
2) form $J = P - \sum_{i=1}^{q} C_i W_i$ and
   compute $b = f_{q+1} - \sum_{i=1}^{q} C_i z_i$
3) decompose J into LU form
4) solve $J x_{q+1} = b$
5) compute $x_i = -W_i x_{q+1} - z_i, \quad 1 \le i \le q$

The parallelism in the first and last step is fairly straight forward and suggests a virtual machine consisting of q processors. In the second step the formation of each $C_i W_i$ can be done in parallel and suggests the same virtual machine. The solution of the smaller system in steps 3 and 4 can either be performed on one machine or distributed across m processors, depending on its size. In our example we will distribute the solution of the smaller system, using a second virtual machine with m processors.

A DINO program for the above algorithm is given in Section 5. In the remainder of this section we discuss this program, and use it to illustrate interesting aspects of the syntax and semantics of DINO. Our discussion proceeds from the highest level of abstraction, the virtual machines, to the next level, the definition of the composite procedures and distributed data structures that are mapped onto these virtual machines, and finally to the lowest level, the statements which use these procedures and data structures.

The virtual machines for this algorithm are defined by the *environment* constructs on lines 5, 61 and 129. Line 5 defines a virtual machine, or environment structure, consisting of M processors named *solve*[0] to *solve*[$M-1$] each of which contains the data and procedures between lines 6 and 59. These environments are used to implement steps 3 and 4 of the algorithm. A second set of environments, the *node* environments, are defined in a similar fashion and are used to implement the parallel aspects of steps 1, 2, and 5 of the algorithm. Line 129 defines the *host* environment which is a single, controlling environment where execution of the program begins. On a distributed memory multiprocessor, both the *solve* environments and the *node* environments would be partitioned evenly among the processors (e.g. if $M = Q$ = the actual number of processors, then each processor would contain one copy of each environment) and the *host* environment would reside on the host processor (assuming such an arrangement, as exists on current hypercubes).

The *host* environment includes the procedure *main*, which is the first procedure to be called. The first and last statements in *main* initialize the data and print the results. They are executed (sequentially) on the host processor since that is where these procedures are defined. The next five statements in *main* are composite procedure calls (denoted by the '#' at the end of each), and implement the five parallel steps of the algorithm discussed above. For each of these calls, a copy of the composite procedure is executed in parallel on each of the environments where it is defined. For the composite procedures *dist_lu* and *dist_solve*, M copies are executed in parallel on the M *solve* environments, and for the others Q copies execute in parallel on the Q *node* environments.

As composite procedures are to be executed on remote nodes of the actual parallel machine, it is not possible to pass arrays as parameters in the normal C fashion (a pointer to the first element). Instead, DINO supports passing entire arrays between functions both as parameters and, in the case of local procedures, as results. The syntax for describing the entire length of an array along one axis is the '[]' operator. The

description of the entire array is therefore a '[]' operator for each axis. Slices and subsections are also possible and will be described later on.

The definitions of the composite procedures start on lines 6, 38, 88, 101 and 118. A composite procedure is similar to a normal C procedure. An important difference, consistent with the style of passing arrays discussed above, is that composite procedures can have value, result, and value-result parameters, whereas parameters to normal C functions are always passed by value. For example, in the procedure *form_Jb* the parameters C and fqp are passed by value, b is passed by result and P is passed by value-result. These modifications seem necessary to support parameter passing in a distributed memory environment.

In order for composite procedures to operate on different sections of data, composite procedure parameters and other data structures must be distributed over the environments on the virtual machines. This distribution is specified by the mapping function in distributed data declarations. For example, in line 89, A, the set q of diagonal blocks $A_1 \cdots A_q$, is defined as a three dimensional array which is distributed based on the programmer defined *slice* (line 63) mapping function. This mapping function partitions an array along its first axis only, thus placing the two dimensional array A_i on environment *node*[i]. The mapping is readily constructed by using the DINO-supplied *block* and *compress* mappings along the first, second and third axes respectively. Similarly, the mapping of f on line 90 places N consecutive elements of f on each environment, and the mapping function *byRow*, defined in line 1, distributes a two dimensional array row-wise over a vector of environments. A mapping function to distribute a matrix column-wise could be defined using *map byCol* = [*compress*][*block*].

The place in the program where data is declared depends on where it is to be used, and follows standard scoping conventions of block structured languages. For example the intermediate value z is declared within the *node* environment (line 66) and its scope is this structure of environments. Since it is never needed by the *host* environment it is neither stored there nor passed back and forth as a parameter. Its contents are valid for the duration of the program. An interesting case is the last part of the x vector, *xqp*. It too is an intermediate value but must be remapped between the fourth and fifth steps of the algorithm. In the fourth phase, where it is calculated, it is partitioned among the *solve* environments (renamed x). In the fifth phase, where it is required to compute each x_i, it is replicated among the *node* environments. This is indicated by the parameter definitions of x in *dist_solve* and *xqp* in *compute_xi*, and the use of *xqp* as an *out* parameter in *dist_solve* and an *in* parameter in *compute_xi*.

Finally we comment on some of the executable statements within the procedures. The composite procedures *factor_A* and *compute_xi* have no interprocess communication, are similar to normal C functions, and therefore will not be described here. In the procedure *form_Jb*, however, interprocess communications is needed for the computation of the sum of $C_i W_i$, and the right hand side of the same set of equations. The calculation of the first sum occurs in line 110. The global summation of a set of values is a typical reduction operation in which each process has a value, in this case $C_i W_i$ returned from *mat_mat_mult*, and the sum of these values is required. This is provided by the function *gsum* in line 110. This is one of several reduction functions provided in the DINO language which takes a value from each of a set of environments and applies an associative operator on them. The # indicates that a remote operation is being done. The result of the *gsum* call, an M by M array, is assigned to the temporary matrix T and the formation of J, stored in P, is than completed by subtracting T from P. The matrix P has more than one row mapped to each processor and therefore the subtraction is done for each of the rows mapped onto the current environment.

---

[1] A block mapping distributes n elements on p environments by placing n/p elements on each environment.

The summation of $C_i z_i$ in line 113 is done in a similar fashion. Note that z_i corresponds to a range of values out of the z array. In DINO the syntax for specifying a subsection is '[<i,j>]' indicating the values from i to j inclusive.

The solution for x_{q+1} is done in the composite procedures *dist_lu* and *dist_solve*. These take a matrix which is distributed by row and find the solution of the system of equations using Gaussian elimination with partial pivoting. Within *dist_lu* the pivot row is computed in line 16 using the reduction function *gmaxdex* which returns the index (second parameter) of the maximum value. This operation is only executed over values from *node*[*diag*] to the last node. This is indicated by specifying, within curly brackets, the environments which will participate.[2] The next statement swaps the rows if a pivot is required. The environment containing the pivot row first sends the row to the environment containing the diagonal row. This is done by assigning row *A*[*pivot*][] to row *A*[*diag*][]. The # indicates that a remote access is taking place and therefore a message is sent to *solve*[*diag*], the environment where *A*[*diag*][] is mapped, containing the value of *A*[*pivot*][]. This message is tagged such that a corresponding remote read of *A*[*diag*][] will receive this value and assign it to *A*[*diag*][]. The send is done in line 19 and the corresponding receive is done in line 24. The pivot environment also sends the pivot row to the rest of the environments participating in the operation as the value *prow* which is than used for the reduction step. If no pivot is required than just the pivot row is sent to the remaining participants. The next statement receives the reduction row from the environment which contains it. Note that a remote read or write to the current environment is equivalent to a local read or write.

4. Discussion. The example program illustrates that a non-trivial parallel algorithm can be rather easily implemented in DINO. The high level constructs in DINO, namely the virtual parallel machines, distributed data mappings, and composite procedures, allow the structure of the parallel program to correspond quite naturally to the underlying high level view of the parallel algorithm. For example, the ability to specify multiple virtual machines was quite helpful in making this example program easy and natural to express. These high level constructs also enable the specification of process control and communication for this program to be significantly simpler and more natural than it would be in many other currently available systems. For example, communications is implicitly generated for composite procedure calls, the remote access of distributed data, and reduction operations. These constructs also allow the compiler to ensure that the communications is correct with respect to message type and content. The result is that we find the DINO program relatively easy both to write and to understand.

This example also helps indicate several aspects of the language that could be improved upon. The first item has to do with the level of parallelism or granularity. Often, it is most natural to express a parallel algorithm at a fine grain level of parallelism. For example, *dist_lu* and *dist_solve* are written assuming there is one row per virtual processor; these procedures are considerably more cumbersome to write if each process must handle a block of rows. However, the current implementation of DINO maps each virtual process (environment) to an actual process. If there are many more rows than processors then there will be many processes on each processor, resulting in an inefficient program. The preferred solution to this problem is to have the programmer continue to write at the natural, fine grain level, and to have the compiler contract multiple environments into single processes when there are more processes than processors. We believe the structure of DINO will allow us to do this efficiently in many cases.

The other major area we would like to address is improved support for complex, multi-phase computations. The example points out several ways in the current DINO facilities for this could be improved.

[2] Any remote operation has an implicit set of environments on which an operation takes place. This set can be explicitly changed by specifying, as in this example, the participating environments. In the case of a remote read or write the implicit set of environments is the environments on which the data is mapped. In the case of a reduction operator it is the set of environments within which the operation takes place. In the case of a composite procedure call it is the set of environments in which the procedure is defined.

One is that the program requires the solution of several systems of linear equations, to find z_i, W, and xqp. In a serial program only one function would be written to handle this but in DINO we are required to write both a serial and parallel version. This problem is related to the contraction problem mentioned earlier.

A more pervasive need is to be able to nest composite procedure calls in a useful way. In the example *main* consists of several consecutive calls to composite procedures. The code would be more readable and more useful for later use if *main* consisted of a single composite procedure, say *block_border*, which could call *factor_A*, *form_Jb*, and *compute_xi*. Currently this can not be done because the fork and join aspects of a composite procedure call are tied to its being called from and returning to the host environment. A solution to this would be to allow the processes in a composite procedure to collectively come together and substitute themselves with another composite procedure. This facility would allow DINO programs to become subprograms in larger DINO programs with little or no modification.

A related but more subtle problem arises when various phases of the algorithm are most naturally expressed at different levels of parallelism. For instance, most of the example algorithm has Q fold parallelism, which is reflected by having Q *node* environments. However, for the solution of xqp, the algorithm has M fold parallelism which is reflected by using M *solve* environments. DINO currently handles this problem by moving J (stored in P) from one set of environments to another between phases. This has two disadvantages. First, it is inefficient because of the extra communication that is required and the overhead of extra environments. Second, the solution to the smaller system of equations, implemented by calling *dist_lu* and *dist_solve*, is logically a sub part of the function *form_Jb* and therefore should not be invoked within *main* but should be invoked at the end of *form_Jb*. The substitution mechanism proposed in the previous paragraph will also solve this problem as long as we allow the substitute composite procedure to use a different structure of environments.

A final problem concerning the different phases of an algorithm has to do with the remapping of data. Note that *xqp* is distributed during *dist_solve* and is replicated for *compute_xi*. In order to accomplish this remapping it must be passed up to *main* and back down to *compute_xi*. The ability to remap data without passing it back to the host would be more useful.

The problems mentioned above are either instances where the ease of expression in DINO leads to sub-optimal efficiency, or where its support for large scale program development seems sub-optimal. Although we feel that it is still easier to write an efficient distributed parallel program using DINO than using many other currently available systems, we would like to improve upon these problems. One of our goals is to determine how well a compiler might be able to convert an inefficient but easy to understand DINO program into an efficient one. A second is to explore the inclusion of a small number of additional language features in DINO.

5. Sample program

```
 1  map byRow = [block][compress];
 2  map byBlock = [block];
 3  map byElement = [block];
 4
 5  environment solve[M:id] {
 6      composite dist_lu(A)
 7          double distributed A[M][M]    map byRow;
 8          {
 9          int diag, pivot, i;
10          double pval, mult;
11          double distributed prow[M] map all;
12
```

```
13          for (diag=0; diag<M; diag++)
14              if (diag <= id){
15                  /*find pivot*/
16                  pivot = gmaxdex(dabs(A[id][diag]), id)#{ solve[<diag,M-1>]};
17                  if (pivot != diag){  /*pivot*/
18                      if (id == pivot){ /*send pivot row to everyone who needs* it/
19                          A[diag][]# = prow[]#{solve[<diag+1,M-1>]} = A[pivot][];
20                          A[pivot][] = A[pivot][]#{solve[diag]};
21                          }
22                      if (id == diag){ /*pivot*/
23                          A[pivot][]# = A[diag][];
24                          A[diag][] = A[diag][]#{solve[pivot]};
25                          }
26                      }
27                  else if (id==diag)
28                      prow[]#{solve[<diag+1,M-1>]} = A[pivot][]; /* if not pivot*/
29                  prow[] = prow[]#{solve[pivot]}; /*get pivot row*/
30                  if (diag < id){ /*eliminate*/
31                      mult = A[id][diag] /= -prow[diag]; /*compute l*/
32                      for (i=diag+1; i<M; i++)
33                          A[id][i] += mult*prow[i];
34                      }
35                  }
36          };
37
38      composite dist_solve(in lu, out x, in b)
39          double distributed lu[M][M]    map byRow;
40          double distributed x[M]        map byElement;
41          double distributed b[M]        map byElement;
42          {
43          double distributed y[M]        map byElement;
44          int i;
45
46              /*solve ly = b*/
47          y[id] = b[id];
48          for (i=0; i<id; i++)
49              y[id] -= y[i]# * lu[id][i];
50          y[id]#{node[<id+1,M>]} = y[id]/lu[id][id];
51
52              /*solve ux=y*/
53          x[id] = y[id];
54          for (i=M; i>id; i--)
55              x[id] -= x[i]# * lu[id][i];
56          x[id]#{node[<0,id-1>]} = x[id]/lu[id][id];
57          };
58
59      }
60
61  environment node[Q:id] {
62
63      map slice = [block][compress][compress];
64
65      double distributed W[Q][N][M] map slice;  /* = A^-1 B*/
66      double distributed z[Q*N]     map byBlock;
67
68      mat_vec_mult(W,v,x)[N] /*x <= Wv*/
69          double W[N][M], v[M];
70          {};
```

```
71
72      sub_vec(v,x,l) /*v <= v - x, l is length of vectors*/
73          double v[], x[];
74          {};
75
76      double mat_mat_mult(A,B)[M][M] /*returns AB*/
77          double A[M][N], B[N][M];
78          {};
79
80      seq_lu(A) /*does lu decomposition on A*/
81          double A[N][N];
82          {};
83
84      seq_solve(lu, x, b) /*solves lu x = b*/
85          double lu[N][N], x[N], b[N];
86          {};
87
88      composite factor_A(in A, in f, in B)
89          double distributed A[Q][N][N] map slice;
90          double distributed f[Q*N]     map byBlock;
91          double distributed B[Q][N][M]  map slice;
92          {
93          int i;
94
95          seq_lu(A[id]);      /*decompose each block*/
96          seq_solve(A[id], &z[id*N], &f[id*N]); /*z = A inv f*/
97          for (i=0; i<M; i++)                  /*W = A inv B*/
98              seq_solve(A[id], W[id][i], B[id][i]);
99          };
100
101     composite form_Jb(P, in C, in fqp, out b)
102         double distributed P[M][M]     map byRow;
103         double distributed C[Q][M][N]  map slice;
104         double distributed fqp[M]      map byBlock;
105         double distributed b[M]        map byBlock;
106         {
107         double tz[N], T[M][M];
108         int i;
109
110         T[][] = gsum(mat_mat_mult(C[id], W[id]))#;   /*C[id] <= C[id]*A^-1 [id]*B[id]*/
111         for (i=id*M/Q; i< id*M/Q + M/Q; i++)
112             sub_vec(P[i], T[i][], M);            /*compute J (in P)*/
113         tz[] = gsum(mat_vec_mult(C[i], z[<id*N, id*N+N-1>]))#;   /* sum z*/
114         for (i=id*M/Q; i< id*M/Q + M/Q; i++)
115             b[i] = fqp[i] - tz[i];               /*b = fqp - sum z*/
116         };
117
118     composite compute_xi(in xqp, out x)
119         double distributed xqp[M]     map all;
120         double distributed x[Q*N]     map byBlock;
121         {
122
123         mat_vec_mult(W[id], xqp, &x[id*N]);
124         neg_vec(&x[id*N]);
125         sub_vec(&x[id*N], &z[id*N], N);
126         };
127     }
128
```

```
129 environment host{
130     init_data()
131         {};
132
133     print_results()
134         {};
135
136     double A[Q][N][N], B[Q][N][M], C[Q][M][N], P[M][M],
137             fqp[M], f[Q*N], xqp[M], x[Q*N], b[M];
138     main(){
139         init_data(); /*initialize A B C P and f*/
140         factor_A( A[][][], f[], B[][][])#;
141         form_Jb( P[][], C[][][], fqp[], b[])#;
142         dist_lu( P[][])#;
143         dist_solve( P[][], xqp[], b[])#;
144         compute_xi(xqp[], x[])#;
145         print_results();
146         }
147     }
```

REFERENCES

[1] R. H. Byrd, R. B. Schnabel and X. Zhang, "Solving Nonlinear Block Bordered Circuit Equations on a Hypercube Multiprocessor", *proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, 1989.

[2] D. Gelernter, N. Carriero, S. Chandran and S. Chang, "Parallel Programming in Linda", *Proceedings of the 1985 International Conference on Parallel Processing*, 1985, 255-263.

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

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

[5] M. Rosing and R. B. Schnabel, "An Overview of Dino -- a new language for numerical computation on distributed memory multiprocessors", *proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, 1987, 312-316.

[6] M. Rosing, R. B. Schnabel and R. P. Weaver, "Dino : Summary and Examples", *proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, 1988, 472-481.