Compiler Support for Unstructured Scientific Computations

Charles Koelbel
Piyush Mehrorta

CRPC-TR90105
December, 1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Compiler Support for Unstructured Scientific Computations

Charles Koelbel
Center for Research in Parallel Computation
Rice University
Houston, TX 77251-1892

Piyush Mehrotra
Institute for Computer Applications in Science and Engineering
Mail Stop 132C
NASA Langley Research Center
Hampton, VA 23665

1 Introduction

Many recent advances in numerical methods have involved irregular data structures and dynamically varying computation. In many cases these new methods permit large amounts of data parallelism to be exploited, at least in principle. Actually achieving substantial parallelism on distributed-memory multiprocessors such as the Intel iPSC/860 has been difficult, however. Three major impediments to efficient implementations of unstructured scientific computations on these machines can be identified.

1. The problem of determining the distribution of program data structures among processor memories.
2. The problem of specifying such a data distribution once it is chosen.
3. The problem of specifying the communications required to implement the program.

This paper attacks the second two problems by allowing the programmer to make the specifications at a higher level than is currently possible. To understand how this is done, however, some background is needed.

The most common methodology for programming distributed-memory machines is to distribute a portion of the data to each processor. Each processor is then responsible for computing and storing its portion of the data. If a processor does not store

all of the data needed for one of its computations, then it must receive the nonlocal data using communication operations. Clearly, the distribution chosen for the data will determine the communication needed in this scheme. Since communication is expensive on these machines, this will then have a great impact on the efficiency of the program. For irregular algorithms, the pattern of the computation often depends on run-time data. This implies that the data distribution must be computed at run-time to achieve good performance; methods of doing this are the focus of active research. Solving this problem is beyond the scope of this paper. Instead, we will concentrate on implementing a distribution once it is given.

Current languages for distributed-memory machines provide very little support for the programmer to specify or implement a specific data distribution. These message-passing languages reflect the underlying architecture closely. Each processor executes a program written in terms of a private address space. These separate address spaces run directly counter to the idea of a single data structure. Instead of using a global indexing arrangement for accessing a distributed data structure, a message-passing program must perform two calculations: finding the processor storing the data and computing the index relative to that processor's local address space. Both of these calculations can be nontrivial for run-time data distributions. Forcing the programmer to perform them explicitly for every reference to a large data structure is intolerable. Such low-level details should be absorbed into the compiler wherever possible; we will show how this can be done below.

Because processors have separate address spaces, data cannot be shared directly. Instead, a processor must receive any nonlocal data it needs by receiving explicit messages. Similarly, if a processor stores data needed by others it must explicitly send messages to them. Managing this communication requires low-level, error-prone code. Additionally, the exact sequence of sends and receives is very sensitive to the distribution of data. A seemingly minor change to the data distribution can mean revising every communication statement in the program. These disadvantages are inherent in message-passing languages because of the explicit communications. Other classes of languages can avoid these problems, however. Several researchers [6, 7, 12, 19] have shown that the compiler can often generate the communication statements automatically. These are another example of a low-level detail that should be handled by the computer rather than the programmer. We will show how this can be done below.

The remainder of this paper presents a syntax for specifying data distributions and shows how the compiler can implement both the distributions and the induced communications. Section 2 presents the syntax and its semantics. The syntax presented there can represent distributions of arrays with one or more dimensions and allows distributions that depend on run-time data. Section 3 shows how the distributions can be implemented. The implementation given here uses distribution descriptors similar to the array descriptors used in implementing many programming languages. Section 4 then explains how communications induced by the distribution can be generated. This part of the implementation uses the inspector-executor

```
processors procs : array[ 0..P-1 ] with P in 1..max_procs;

var     block_size : integer;
        home : array[ 0..N-1 ] of integer dist by [block] on procs;

dist    map1 = [i] on procs[ (i/blocksize) % P ];
        map2 = [i] on procs[ home[i] ];

var     a1, b1 : array[ 0..N-1 ] of real dist by [map1];
        a2, b2 : array[ 0..N-1 ] of real dist by [map2];
        c : array[ 0..N-1, 0..M-1 ] of real dist by [map1,*];
        d : array[ 0..N-1, 0..M-1 ] of real dist by [*,map2];

distribute a1, b1, a2, b2, c;
distribute and copy a1;

forall i in 0..N-1 do
   a1[i] := a2[i];
end;
```

Figure 1: Syntax for distributions and forall statements

paradigm of [6, 4, 10]. Finally, Section 5 gives conclusions, relations to other work, and directions for future research.

2 Syntax

Figure 1 shows the syntax for data distributions used in this paper. This syntax is based on the Kali programming language [8], but can easily be adapted to any imperative language.

The **processors** declaration on the first line declares the set of processors which will execute the program. In this case, the processors are logically arranged in a one-dimensional array; a simple extension to the syntax can declare multi-dimensional arrays. We assume that any two processors can communicate via messages (although the communication times may not be the same for all processor pairs). The **with** clause selects the actual number of processors to run the program at load time. Once it is chosen, the number of processors remains constant for the duration of the program. This allows a simple parameterization of the program by number of processors. Choosing this number at load time provides flexibility by allowing the program to run on different sized machines.

The first **var** section declares two variables that are used in the distributions. The scalar *block_size* is replicated on all processors; this allows it to be used by any processor without inducing communication. The *home* array, on the other hand, is

distributed by a **block** pattern. This means that the array is divided into equal-sized sections, which are then distributed among the processors in the natural way. **Block** partitioning is a static distribution in that it does not depend on run-time data. Although the techniques of Section 3 can be used to implement such distributions, their simple nature can be exploited in their implementation. We will not consider static data distributions in detail below. Instead, we will focus on run-time distributions like those shown in the **dist** section of Figure 1.

Fundamentally, all data distributions specify a mapping from array elements to the processors that store them. Our syntax represents this by a function-like notation. The declaration of *map*1, for example, states that element i of an array is stored on processor $\lfloor \frac{i}{block\_size} \rfloor \bmod P$. Note that although this distribution depends on run-time data (*block_size*), it does not depend on distributed data. This is not the case with distribution *map*2, which assigns array element i to processor *home*[i]. The distributed array *home* allows arbitrary mappings to be specified at the price of possibly inducing some communication in evaluating the mapping. Note that both distributions are designed in terms of the global array. This is a general property of our distributions which helps preserve the appearance of a global address space. The method of declaring distributions shown here only specifies half of the mapping of the global address space onto a distributed-memory machine. It only specifies which processor will store each array element; the compiler must generate the translation of the global indices into offsets into the local section of the array. Additional syntax can be added to allow the user to specify this translation, but we will not consider that possibility here.

The next **var** section shows how distributions can be used. Arrays $a1$ and $b1$ are distributed according to the *map*1 definition. Similarly, $a2$ and $b2$ are distributed according to *map*2. The declarations of c and d show one way these distributions can be extended to multi-dimensional arrays. In those arrays, only a single dimension is divided among the processors; the other dimension is treated as a unit for purposes of distribution. Thus, the rows of c are distributed according to *map*1. This means that all elements of row 0 of c will be stored on processor 0, all elements of row *block_size* will be stored on processor 1, and so on. Similarly, the columns of array d are distributed by *map*2. If one-dimensional distributions are not sufficient, distributions with two or more dimensions can be declared in the **dist** section by using a slightly extended syntax.

Because distribution *map*1 and *map*2 depend on variables, they are not well-defined until those variables have had values assigned. In addition, it is often useful to change distributions between phases of a program by reassigning those variables. Unconstrained redistributions, however, would be very difficult to implement efficiently. For these reasons we introduce the **distribute** statement. An array with a run-time distribution can only be accessed after it has been named in a **distribute** statement. (Static distributions like **block** do not need a **distribute** statement.) All information about the distribution is fixed at that time; later changes to the variables in the distribution do not affect the distributed arrays. A later **distribute** state-

ment recomputes the distribution information, thus allowing dynamic behavior. Two forms of the **distribute** statement are available: the simple form and the copying form. The first statement in Figure 1 shows the simple form, in which the contents of the array are not preserved. This is useful for initializing distributions and changing the distribution of temporary arrays. The copying form, shown on the next line, copies the old contents of the array into their new positions. This ensures that live data are not lost during a redistribution phase.

The final three lines of Figure 1 show the **forall** loop, the basic parallel construct considered in this paper. Semantically, this is identical to a **for** loop with no inter-iteration data dependencies. Because there are no data dependencies, the loop iterations may be performed in any order. For the same reason, all data used in the loop are available when the loop begins executing. The first fact allows the loop to be executed in parallel, giving each processor the iterations that assign to its local array elements. The second fact allows all communication to be performed when the **forall** begins execution. The implementation in Section 4 will exploit both of these observations.

3 Implementation of Distributions

The fundamental idea in the implementation of data distributions is the distribution descriptor. This descriptor is an extension of the array descriptors used to implement many imperative languages. While traditional descriptors contain information related to allocating and addressing an array, the descriptor for distribution A must provide four functions:

1. Determining where element i of an array is stored. This function will be referred to as $proc_A(i)$.

2. Determining whether element i is stored locally. This function will be called $local_A(i)$.

3. If an element is local, determining that element's offset in the local section of the array. This function will be called $offset_A(i)$.

4. Iterating through the local elements of an array. This operation will be referred to as $forall_A(i)$, where i is the index variable for the iteration.

In addition, the descriptor will be able to provide the size of the local section of the array for memory allocation.

To accomplish these tasks, a descriptor will contain the following data:

1. The upper and lower bounds of the array being distributed.

2. A count of the number of arrays referencing this descriptor.

3. Copies of any undistributed variables referenced in the **dist** declaration.

4. A hash table containing offsets for each local array element.

5. A pointer to a thread through the hash table.

6. A block-distributed array of the same size as the distributed array. This array will store the values of $proc(i)$ for all values of i.

The size of the descriptor will be of the same order as the array being distributed. For this reason, we allocate one descriptor for each distinct distribution rather than one for every array. In most scientific codes, there will be many arrays sharing a single distribution; these arrays can share a single descriptor, thus amortizing the memory cost. This sharing can become complex if not all arrays are redistributed at the same time; we will refer back to this point later. Similarly, a $N \times M$ array distributed by rows uses a descriptor of size $O(N)$ rather than one of size $O(NM)$. Also, the block-distributed array can be eliminated if no distributed variables are referenced in the **dist** declaration. Figure 2 shows C representations for the descriptors for the two distributions of Figure 1.

Given the above descriptors, we can describe how each of the operations defined above is computed for single-dimensional arrays. For distributions like $map1$ that do not reference distributed variables, $proc_{map1}(i)$ can be computed almost directly from the expression in the **dist** declaration. Because of the semantics of the **distribute** statement, the values of all undistributed variables must be taken from the descriptor fields rather than from the program variables. The function $local_{map1}(i)$ is then the result of comparing $proc_{map1}(i)$ with the current processor number. Distributions like $map2$ which refer to distributed data compute $proc_{map2}(i)$ by referencing the appropriate element of the $proc$ array in their descriptors. This may require a message to be sent with the appropriate value. Fortunately, these communications can be handled efficiently by the inspector-executor strategy of Section 4. $Local_{map2}(i)$ can always be computed without communication by checking the hash table for an entry for i. Either $offset_{map1}(i)$ or $offset_{map2}(i)$ can be computed by searching the appropriate hash table. The integer offset from that table can then be added to the base address of the local array section. Similarly, the algorithms for $forall_{map1}(i)$ and $forall_{map2}(i)$ are identical: follow the $next$ pointers through the hash tables. C macros for these operations[1] are given in Figure 3. All of the macros take a distribution descriptor as a parameter, since there will often be more than one active descriptor.

The extensions of these distributions to two-dimensional arrays are straightforward. Subscripts of undistributed dimensions are ignored in computing the equivalents of $proc(i)$ and $local(i)$. Instead of using $offset(i)$ directly, its value is used in place of the appropriate subscript in the usual row-major (or column-major) subscript calculation. Finally, iterations over a distributed dimension are handled exactly as for one-dimensional arrays.

Distribution descriptors are initialized when a **distribute** statement is executed. Figure 4 shows the C code to initialize a descriptor for $map1$. First, the array bounds

[1]The code for $proc_{map2}$ is omitted because of the complexity of message-passing.

```
/* HASH_TABLE is an implementation of hash tables */
/* struct hash_entry is a single entry in that table */
struct hash_entry {
    int element;                    /* global index for this entry */
    int offset;                     /* local offset for element */
    struct hash_entry *chain;       /* chain for collisions */
    struct hash_entry *next;        /* list of local array elements */
};


struct desc_map1 {
    int low;                        /* lower bound */
    int high;                       /* upper bound */
    int ref_count;                  /* descriptor reference count */
    int block_size;                 /* copy of block_size variable */
    HASH_TABLE offset;              /* hash table of offsets */
    struct hash_entry *list;        /* thread through hash table */
};

struct desc_map2 {
    int low;                        /* lower bound */
    int high;                       /* upper bound */
    int ref_count;                  /* descriptor reference count */
    HASH_TABLE offset;              /* hash table of offsets */
    struct hash_entry *list;        /* thread through hash table */
    struct desc_block desc_proc; /* descriptor for proc array */
    int *proc;                      /* processor function tabulated */
};
```

Figure 2: Distribution descriptors for Figure 1

```
#define proc_map1(desc,i)          (((i) / (desc).block_size) % P)
#define local_map1(desc,i)         (proc_map1((desc),(i)) == MY_PROC)
#define offset_map1(desc,i)        (hash_search((desc).offset,(i)) -> offset)
#define forall_map1(desc,i)        \
        {                                                          \
           struct hash_entry *p;                                   \
           for ( p = (desc).list; p != NULL; p = p->next ) {       \
                 i = p->element;
#define end_forall_map1(desc)      \
           }                                                       \
        }

#define local_map2(desc,i)         (hash_search((desc).offset,(i)) != NULL)
#define offset_map2(desc,i)        (hash_search((desc).offset,(i)) -> offset)
#define forall_map2(desc,i)        \
        {                                                          \
           struct hash_entry *p;                                   \
           for ( p = (desc).list; p != NULL; p = p->next ) {       \
                 i = p->element;
#define end_forall_map2(desc)      \
           }                                                       \
        }
```

Figure 3: Basic distribution descriptor operations

and any undistributed variables used by the distribution are saved. Then the lists of elements on each processor are computed in two stages. Each processor takes responsibility for computing the new home addresses of one block of the array; the computations are done by brute force calculation using the expression in the **dist** declaration. This gives each processor p a bucket-sorted array containing, for every processor, a list of the elements from p's block stored there. Thus, the list of all elements stored on processor p is the concatenation of the lists for p on all processors. A global communication phase performs these concatenations and routes the lists to the correct processors. This can be done by Fox's Crystal router [3] in $O(\log P)$ stages on a hypercube; similar algorithms can be developed for other interconnection networks. Once a processor has its complete list of elements, it is a simple matter to traverse the list to build the hash table. Offsets are assigned as the elements are inserted in the table, starting with 0. The algorithm for initializing *map*2's descriptor is similar, except that the addresses of elements (calculated as $home[i]$ rather than $(i/block\_size)$ mod P) are stored in the *proc* array field as well as in the element lists. More complex distribution declarations may require nonlocal elements of distributed arrays to be fetched from other processors during the computation of the element lists. These calculations can be optimized using the inspector-executor strategy described in Section 4.

Once the new descriptors have been calculated, the **distribute** statement assigns a pointer to each array to point to its distribution descriptor. Space for the array is allocated based on the number of entries in the hash table. If the copying form of **distribute** is used, values are copied from the old array area to the new one. Note that this may involve communication, which is again handled by the inspector-executor strategy. Finally, the reference counts on the distribution descriptors are adjusted, and unreferenced descriptors can be deallocated. The time for executing the entire **distribute** statement is $O(\max_p(n_p))$ computation steps, where n_p is the number of array elements distributed to processor p. There are also $O(\log P)$ communication steps, as explained above.

4 Implementation of Communication

Section 3 showed the mechanics of defining data distributions and accessing them. In this section we show how communication statements can be generated for **forall** loops. With regular distributions and subscripts of a simple form, the compiler can completely generate the message traffic for this situation [6, 7, 12, 19]. With run-time distributions, however, the information needed to do this does not exist at compile time, even when the subscripts are simple. Instead, a run-time strategy must be used. We present such a strategy here based on the inspector-executor paradigm of [4, 10]. This strategy evaluates a **forall** loop in two phases. The inspector phase generates a description of the communication necessary for the loop. For each processor, this information consists of two lists: a receive list of array elements that must be received in messages and a send list of elements that must be sent to other processors. The

```
int i;                          /* loop index */
int low, hi;                    /* block parameters */
int count[MAX_PROCS];           /* number of elements found for each proc */
int *element[MAX_PROCS];        /* elements for each proc */
int my_count;                   /* number of elements for this proc */
int *my_subs;                   /* elements for this proc */

/* save block_size and bounds for future use */
desc_map1.block_size = block_size;
desc_map1.low = 0;
desc_map1.high = N-1;

/* compute one section of the table (defined by block dist) */
bzero( count, sizeof(int) * max_procs );
bzero( subs, sizeof(int *) * max_procs );
low = MY_PROC*(N/P);            /* bounds of local section of blocked array */
hi = low + block - 1;
for ( i = low; i <= hi; i++ ) {
   /* insert i into element list for appropriate processor */
   element_list_add( i, (i/block_size)%P, count, size, subs );
}

/* global communication to distribute element lists */
global_exchange( count, subs, &my_count, &my_subs );

/* fill hash table from received addresses */
desc_map1.offset = allocate_hash( N / P );
desc_map1.list = NULL;
for ( i = 0; i < my_count; i++ ) {
   /* add element to hash table */
   hash_add( my_subs[i], desc_map1.offset );
   /* set head of element list if necessary */
   if (desc_map1.list == NULL)
     desc_map1.list = hash_search( my_subs[i], desc_map1.offset );
}
```

Figure 4: Initializing the distribution descriptor for *map1*

executor uses this information to perform the communication and in the execution of the loop body.

Figure 5 shows the inspector generated for the **forall** loop in Figure 1. In this example, $a2[i]$ is the only reference that may refer to a nonlocal array element. Processor p therefore generates all of the references to $a2$ that it might make during the loop. Each reference is checked for locality using $local_{map2}(i)$. Note that these checks can be made without communication. Because the actual location of the referenced elements cannot be found directly, the elements are recorded on a receive list. A global operation then inverts this list so that every processor knows which other processors need addresses that it stores; the inverted list will be used as a send list. Once this send list is formed, it is used to send records of element locations to the processors requesting them, and the original receive list is used to control receipt of those messages. This is the basic paradigm of inspector and executor operations: form a receive list, inverting it into a send list, and use the two lists to control message-passing operations. Once the element location records have been received, they are used to create another receive list, which is inverted into a new send list. These lists have precisely the information needed to control sending and receiving the actual data to execute the **forall**.

In the case of complex subscripts, the above strategy may not suffice. In particular, if a subscript contains a distributed variable then the loop may require communication to form the list of references. In these cases, however, the inspector can be applied recursively starting with the innermost subscripts to produce the correct results. An important optimization of the inspector is also possible, based on the observation that communication patterns are often reused. In these cases, the cost of the inspector can be amortized by only executing it once and saving the send and receive lists for later use. This is shown in Figure 5 by the conditional surrounding the inspector.

Figure 6 shows the executor for the same **forall**. The structure here is very straightforward. First the communication is performed using the send and receive lists from the inspector, then the computation is done using both local and communicated values. The conditional is necessary in the computation phase because nonlocal values are stored in a temporary array rather than as part of the original array. Storing the values adjacent to the array could eliminate the test, but it is not clear how such a strategy could be generalized to multi-dimensional arrays. Other optimizations, such as overlapping communication and computation, are considered in [6].

5 Conclusions and Future Work

We have shown a simple syntax and implementation for run-time data distributions. The techniques shown here are handle the irregular and dynamic distributions needed for unstructured scientific computations, as well as many other applications. Implementing the distributions in the compiler simplifies the task for the programmer by removing unnecessary details from the user level. It also promotes portable programs, since the application can be reimplemented in the compiler for a new machine without

```
/* SEARCH_TABLE is some data structure supporting search operations */

int i;                                  /* loop index */
int n;                                  /* number of elements in buffer */
struct { int elem, proc; } buffer[MAX_MSGS]; /* replies from phase 1 */
SEARCH_TABLE recv_list1;                /* elements to recv from other procs */
SEARCH_TABLE recv_list2;                /* elements to recv from other procs */
SEARCH_TABLE send_list;                 /* elements to send to other procs */

if ( need_inspector() ) {

   /* phase 1: find nonlocal references and their processors */
   create_comm_list( recv_list1 );
   forall_map1( desc_map1, i )
         /* put nonlocal references to a2 in recv_list1 */
         if ( !local_map2( desc_map2, i ) ) {
            comm_list_add( i, proc_block( desc_map2.desc_proc, i ),
                           recv_list1 );
         }
   end_forall_map1( desc_map1 );
   /* global communication to identify elements to send */
   global_transpose( recv_list1, &send_list );
   /* send and receive messages with element locations */
   send_scatter_procs( send_list, desc_map2.desc_proc, desc_map2.proc );
   recv_gather( recv_list1, buffer, &n );

   /* phase 2: set up actual communication channels */
   create_comm_list( recv_list2 );
   for ( i = 0; i < n; i++ ) {
         /* take processor entries from buffer and add to recv_list2 */
         comm_list_add( buffer[i].elem, buffer[i].proc, recv_list2 );
   }
   /* global communication to generate send list */
   global_transpose( recv_list2, &send_list );

}
```

Figure 5: Inspector for **forall** loop in Figure 1

```
float tmp;                          /* temporary */
int n2;                             /* number of elements in buffer */
float buffer2[MAX_MSGS];            /* buffer for received messages */

/* phase 1: communication */
send_scatter_values( send_list, desc_map2, a2 );
recv_gather( recv_list2, buffer, &n );

/* phase 2: computation */
forall_map1( desc_map1, i ) {
   if ( local_map2(i) )
     tmp := a2[ offset_map2(i) ];
   else
     tmp := buffer2[ comm_search( recv_list2, i ) ];
   a1[ offset_map1(i) ] = a1[ offset_map1(i) ] * tmp;
}
```

Figure 6: Executor for **forall** loop in Figure 1

effort from the user. The ability to specify data distribution at a high level will also encourage research into new data distribution patterns, since it will make programs more independent of their distributions. All of this will have a very positive effect on the implementation of state-of-the-art scientific codes on scalable multiprocessing systems.

Although data partitioning is an area of active research [3, 11, 18, 20], relatively little research has been done on specifying the resulting distributions. Saltz and his coworkers [17, 16] have produced a prototype compiler which can specify irregular distributions. Williams and Glowinski's DIME programming environment [21] provides tools for partitioning of irregular meshes. Both groups use strategies similar to the inspector-executor shown here for generating communication, but the methods of specifying the distributions differ considerably from each other and from our work. A detailed comparison of our work with these two projects would be interesting. Techniques for automatically implementing regular partitions can be can be found in [1, 2, 4, 7, 13, 14, 19, 22]. Of these, only Koelbel and Mehrotra [4], Li and Chen [7], and Pingali and Rogers [13] explicitly consider run-time generation of messages. The inspector-executor strategy was introduced independently by Saltz and his coworkers [9, 15, 10] and Koelbel and Mehrotra [4, 5] in various contexts. Saltz's hashed-cache scheme [10, 16] is particularly similar to the methods presented here, but incorporates the data structures from the inspector directly into the hash table rather than separating conderns as we do.

Much work remains to be done on run-time distributions. The choice of hash function to be used was not addressed in Section 3; we are currently evaluating several possibilities. Likewise, there are several possibilities for the search structure for the

send and receive lists. The code generated for the inspector and executor is much more complex than that produced by more traditional program transformations, and it is not clear how this complexity will affect code generation. Finally, two extensions to the scope of this work stand out: distribution of non-numeric data structures such as trees, and the interaction of data distributions with more complex parallel constructs such as **doacross** loops.

Bibliography

References

[1] F. André, J.-L. Pazat, and H. Thomas. PANDORE: A system to manage data distribution. In *International Conference on Supercomputing*, pages 380–388, June 1990.

[2] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, 1988.

[3] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[4] C. Koelbel, P. Mehrotra, J. Saltz, and S. Berryman. Parallel loops on distributed machines. In *Proceedings of the 5th Distributed Memory Computing Conference*, page to appear, Charleston, SC, April 9-12 1990.

[5] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 177–186, Seattle, WA, March 14-16 1990.

[6] Charles Koelbel. *Compiling Programs for Nonshared Memory Machines.* PhD thesis, Purdue University, West Lafayette, IN, August 1990.

[7] J. Li and M. Chen. Synthesis of explicit communication from shared-memory program references. Technical Report YALEU/DCS/TR-755, Yale University, New Haven, CT, May 1990.

[8] P. Mehrotra and J. Van Rosendale. Compiling high level constructs to distributed memory architectures. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, March 1989.

[9] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 140–152, St. Malo, France, 1988.

[10] S. Mirchandaney, J. Saltz, P. Mehrotra, and H. Berryman. A scheme for supporting automatic data migration on multicomputers. ICASE Report 90-33, Institute for Computer Applications in Science and Engineering, Hampton, VA, May 1990. to appear in *Proceedings of the 5th Distributed Memory Computing Conference.*

[11] D. Reed, L. Adams, and M. Patrick. Stencils and problem partitioning: Their influence on performance of multiprocessor systems. *IEEE Transactions on Computers*, C-36(7):845–858, July 1987.

[12] A. Rogers. *Compiling for Locality of Reference.* PhD thesis, Cornell University, Ithaca, NY, August 1990.

[13] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 69–80, June 21-23 1989.

[14] M. Rosing, R. W. Schnabel, and R. P. Weaver. The DINO parallel programming language. Technical Report CU-CS-457-90, University of Colorado, Boulder, CO, April 1990.

[15] J. Saltz and M. Chen. Automated problem mapping: The crystal runtime system. In *Proceedings of the Hypercube Microprocessors Conference*, Knoxville, TN, 1986.

[16] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[17] Joel Saltz, Harry Berryman, and Janet Wu. Multiprocessors and runtime compilation. ICASE report 90-59, Institute for Computer Applications in Science and Engineering, Hampton, VA, 1990.

[18] L. Snyder and D. G. Socha. An algorithm producing balanced partitionings of data arrays. In *Proceedings of the 5th Distributed Memory Computing Conference*, page to appear, Charleston, SC, April 9-12 1990.

[19] P. S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1989.

[20] D. Vrsalovic, E. F. Gehrinnger, Z. Z. Segall, and D. P. Siewiorek. The influence of parallel decomposition strategies on the performance of multiprocessor systems. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 396–405. ACM SIGARCH, June 1985.

[21] R. D. Williams and R. Glowinski. Distributed irregular finite elements. Technical Report C3P 715, Caltech Concurrent Computation Program, Pasadena, CA, February 1989.

[22] H. Zima, H. Bast, and M. Gerndt. *Parallel Computing*, volume 6, chapter Superb: A Tool for Semi-Automatic MIMD/SIMD Parallelization, pages 1–18. North-Holland, Amsterdam, 1988.