Value-Based Distributions in Fortran D: A Preliminary Report

Reinhard v. Hanxleden
Ken Kennedy
Joel Saltz

CRPC-TR93365-S
December, 1993

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

Revised February, 1994.

Value-Based Distributions in Fortran D — A Preliminary Report*

Reinhard v. Hanxleden†
CRPC
Department of Computer Science
Rice University
Houston, TX 77251

Ken Kennedy
CRPC
Department of Computer Science
Rice University
Houston, TX 77251

Joel Saltz
UMIACS
Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Compiling irregular applications written in a data-parallel, High Performance Fortran-like language presents a challenging problem of growing importance. One principal difficulty with irregular problems is the general lack of access locality when distributing data and computation naïvely, based on simple functions of array and iteration indices. To address this problem, we extend classical, index-based data distributions to *value-based distributions.* This type of distribution allows the programmer to conveniently express for example the locality characteristics of an underlying physical problem. This gives the compiler an opportunity to improve both inter- and intra-processor locality, resulting in better performance and scalability, even when this locality is not inherent in the original data structures and access patterns.

This paper reports on the experience gained from implementing value-based distributions in a Fortran 77D compiler prototype. We propose a natural extension of index-based distributions as already present in Fortran D and HPF. This paper addresses the compilation issues involved and makes a quantitative comparison of index and value-based distributions for a Molecular Dynamics application.

1 Introduction

Distributed memory parallel machines provide high performance at reasonable cost. However, the programming paradigms associated with them often differ greatly from sequential languages, and programmers often consider them to be more difficult to use than sequential languages. For example, many vendors of parallel machines offer message-passing language dialects that require the user add matching send/receive pairs for each off-processor data access.

Several research projects have aimed at providing a "machine independent parallel programming style" as a more user-friendly and investment-preserving alternative. Here the applications programmer uses a dialect of sequential Fortran and annotates it with high level distribution information. Examples include High Performance Fortran (HPF) [KLS+94], Vienna Fortran [CMZ92], and Fortran D [FHK+90], which comes in two flavors, Fortran 77D and Fortran 90D. From this annotated program, data-parallel compilers will generate codes in different native Fortran dialects for different parallel architectures. A prototype Fortran 77D compiler targeting MIMD distributed memory machines has been developed at Rice University [HKK+91, Tse93] as part of the ParaScope programming environment [CCH+88]. A Fortran 90D compiler has been written at Syracuse

*This work is supported by an IBM fellowship, and by the National Aeronautics and Space Administration/the National Science Foundation under grant #ASC-9349459.

†Corresponding author. E-mail: reinhard@rice.edu. Phone: (713) 527-8750x2740. Fax: (713) 285-5136.

University [BCF+93]. For regular problems, *i.e.*, applications with relatively simple array subscript functions and fixed communication requirements, these compilers have had considerable successes.

However, irregular problems, such as molecular dynamics or unstructured mesh codes, have turned out to be significantly harder to parallelize [SH91]. One principal problem with irregular problems is the general lack of access locality when distributing data and computation naïvely. For example, assume that the value assigned to some array element $a(i)$ depends on some other array element $b(j)$. In a regular problem, there will typically be some simple relationship between i and j, for example that they are related to each other via a linear function known at compile time. This usually implies that at least this dependence can be satisfied, with little or no communication, when distributing a and b by partitioning their index spaces in some regular fashion among processors and aligning them to each other in a certain way. We will refer to this characteristic as *index-based locality*. Exactly how arrays should be distributed and aligned in the presence of index-based locality is by no means trivial and still an active field of research [KMCKC93]. However, one can generally assume that only regular distributions (like BLOCK, CYCLIC, or BLOCK-CYCLIC) and redistributions have to be considered.

For irregular problems, this assumption cannot be made. Here subscripts i and j may each be determined by some complicated function or an array lookup, the outcome of neither one known at compile time. Furthermore, even if the compiler would know the values of the i's and j's, there would often still be no way to achieve good locality (*i.e.*, low communication requirements) via regular, index oriented distributions. For example, if i and j are different vertices in a mesh, then $a(i)$ and $b(j)$ might depend on each other if i and j are linked together by an edge. Most meshes number their vertices in a way that makes it hard to predict whether two vertices are linked together by just looking at their index. Therefore, distributing mesh points and the computation associated with them across processors by dividing their index space in some regular fashion typically results in many off-processor accesses; *i.e.*, if a processor is assigned some $a(i)$ and therefore has to know the value of $b(j)$, chances are high that $b(j)$ will be on a different processor. The potential speedup gained from distributing data and computation across processors is likely to be lost by exceedingly high communication costs when using simple distribution schemes.

To allow more general access patterns, index based mapping arrays [WSBH91] or mapping functions [CMZ92] can be used. There, however, the programmer has to provide such arrays or functions that explicitly use indices. Another alternative is to replicate the data and distribute just the computation itself and combine results at the end. This approach has the further advantage of simplicity and robustness, and for relatively small problem sizes and numbers of processors it may actually result in satisfactory performance [CM90]. However, one of the main advantages of distributed memory machines, scalability to large problem and machine sizes, has to be compromised under this approach. Therefore, after an inital parallelization based on data replication, many researchers choose to distribute the data in a way that considers the actual data dependences specific to their application [CHMS94]. These "irregular" distributions are typically harder to debug and manage than regular distributions and present an additional level of complexity beyond general message-passing style programming.

High level library routines, such as CHAOS [DMS+92, DHU+93], can assist in tasks like global-to-local name space mappings, communication schedule generation, and schedule based communication. Such libraries are essential for keeping code complexity and programming difficulties in reasonable limits. However, since each application still has its own individual communication requirements, the proper usage of such routines and preprocessing for generating their arguments is still a non-trivial task one would rather leave to the compiler.

```
do i = 1, N_atoms
    do p = 1, inb(i)
        j = partners(i, p)
        force = nbforce(x(i), x(j))
        f(i) = f(i) + force
        f(j) = f(j) - force
    enddo
enddo
```

Figure 1: Sequential version of the Non-Bonded Force (NBF) kernel. N_{atoms} is the total number of atoms, $inb(i)$ is the number of atom *partners* that are close enough to atom i to be considered for the NBF calculation. The coordinate and force arrays, x and f, are shown only one dimensional.

Fortunately, many scientific applications lacking any index-based locality that a compiler might take advantage of do offer another kind of locality, which we will refer to as *value-based locality.* This kind of locality, which will be introduced in more detail in the next section, naturally lends itself to *value-based distributions.* This paper reports on an implementation of value-based distributions within the Fortran D compiler prototype at Rice University. Here we will focus on the compiler technology specific to handling such distributions and its relationship to other aspects of the compiler. For a discussion of other, equally important aspects of compiling irregular applications, such as communication analysis and preprocessing, we refer the reader to other publications [DSvH93, Han93].

The rest of this paper is organized as follows. Section 2 introduces value-based locality and illustrates the use of value-based distributions with a kernel taken from a Molecular Dynamics code. Section 3 lists the implications of value-based distributions for the resulting message-passing node program. Section 4 describes language extensions and compiler technology for generating such a node program. Section 5 contains experimental results obtained for the Molecular Dynamics kernel introduced in Section 2 and compares the effectiveness of index-based and value-based distributions for this application. Section 6 discusses related work, and Section 7 concludes with a brief summary.

2 Value-Based Locality

In the presence of *value-based locality,* two array references $a(i)$ and $b(j)$ may not be related to each other by their indices, i and j, but instead by their values, $a(i)$ and $b(j)$, or the values of other variables, like $x(k)$ and $x(l)$, that in turn are related to $a(i)$ and $b(j)$ by their indices (for example, $k = i$ and $l = j$). Revisiting the mesh example, x might be a coordinate array storing the physical location of each mesh point (assuming 1-d for simplicity). Then the probability that vertices i and j are connected increases as $|x(i) - x(j)|$ decreases. Note that this is not a strict relationship; whether an edge actually exists or not still depends on other factors, like mesh density and topology. However, since data distribution is not a correctness but only an efficiency issue, we are usually more interested in a fast heuristic for finding a reasonably good data distribution than in a strictly optimal, but expensive solution.

Another example where data are not related by indices, but values, are molecular dynamics programs such as GROMOS [GB88], CHARMM [BBO+83], or ARGOS [SM90] that are used to simulate biomolecular systems. One important routine common to these programs is the non-bonded force (NBF) routine, which typically accounts for the bulk of the computational work (around 90%). Figure 1 shows an abstracted version of a sequential NBF calculation. An important characteristic of the NBFs is that their intensities decay very rapidly with increasing distance between the atoms involved. This locality is exploited by using a *cutoff radius*, R_{cut}, beyond which the NBF interactions are ignored. This reduces the number of atom pairs for which the NBF has to be computed and significantly reduces overall computational costs. Furthermore, we can exploit this locality when distributing data in a parallel implementation by distributing them based on the value of x, which stores the physical atom coordinates. The atom numbering itself is not related to the coordinates (instead, they are typically numbered according to some data bank standard, like the Brookhaven protein data bank, which considers amino acid types, peptide bonds, solute/solvent classifications, etc. – an ordering which is rather difficult for a compiler to take direct advantage of without user assistance). Therefore, a traditional, index based distribution would loose this locality. In fact, since each processor needs for each owned atom to know about all atoms within R_{cut} of that atom, a regular distribution typically results in each processor accessing all data, with accordingly high communication and storage requirements.

Value-based distributions allow the programmer to express value-based locality as a simple extension of the regular distribution mechanism employed in data-parallel languages. For example, let array `x` be aligned to a decomposition `decomp`. Then the directive `DISTRIBUTE decomp(VALUE(x))` can be interpreted to mean that `x` should be distributed such that the values of the elements of `x` assigned to each individual processor are from disjoint intervals. (See Section 4.1 for a more formal description of the syntax proposed for value-based distributions.) In other words, if the values of two elements of `x` are close together, they are likely to be on the same processor.

The actual use of a value-based distribution can be seen in Figure 2, which shows a Fortran D implementation of the NBF kernel outlined in Figure 1. This program is a condensed and abstracted version of the 340-line GROMOS subroutine `nonbal.f` (without the Coulombic interactions), enhanced with some data initialization. Although this program is very simplified, it still presents similar difficulties as the original code with respect to the compiler. Fortran D directives first declare a decomposition `atomD`, then align coordinates `x`, forces `f`, partner counts `inb`, and adjacency lists `partners` with `atomD`, and finally distribute `atomD`. Note that initially this is a regular, `BLOCK`-wise distribution. After reading in the initial coordinates and partner counts, `atomD` gets redistributed according to the values of coordinate array `x`. Note also that the strict owner computes rule is overridden in the NBF calculation itself via an `on_home` directive.

3 Characteristics of Programs Using Value-Based Distributions

The MIMD Fortran D compiler transforms a Fortran D program, written in a global name space and annotated with data distribution directives, into a message passing program, which is a local name space node program including communication statements. This section describes the consequences that distributing data by value has on a node program that a compiler would generate (or that a programmer directly using message passing would code by hand). We will try to illustrate, on the one hand, that handling value-based distributions is non-trivial, and, on the other hand, that it is very undesirable to burden the programmer with all the details associated with implementing value-based distributions.

3.1 Specification and State

A regular distribution can be fully specified by a simple keyword ("`BLOCK`") or a keyword enhanced with a small list of parameters ("`BLOCK_CYCLIC(blockSize)`"). There is very little state associated

```
1          PROGRAM nbf
2
3          INTEGER i, j, p, t, n$proc, atomMax, pMax, stepMax
4          PARAMETER (n$proc = 8)
5          PARAMETER (atomMax = 8000, pMax = 250, stepMax = 30)
6          INTEGER inb(atomMax), partners(atomMax, pMax)
7          REAL x(atomMax), f(atomMax), force
8
9    C     Fortran D directives
10         DECOMPOSITION atomD(atomMax)
11         ALIGN inb, x, f, partners(i,j) WITH atomD(i)
12         DISTRIBUTE atomD(BLOCK)
13
14   C     Initialize data
15         CALL read_data(x, y, z, inb, partners)
16
17   C     Redistribute atomD according to coordinate values
18         DISTRIBUTE atomD(VALUE(DIM=1,VALS=x,WEIGHT=inb))
19
20   C     Loop over timesteps
21         DO t = 1, stepMax
22
23   C        Reset forces to zero
24            DO i = 1, atomMax
25               f(i) = 0
26            ENDDO
27
28   C        Computes forces
29            EXECUTE (i) ON_HOME f(i)
30            DO i = 1, atomMax
31               DO p = 1, inb(i)
32                  j = partners(i, p)
33                  force = nbforce(x(i), x(j))
34                  f(i) = f(i) + force
35                  f(j) = f(j) - force
36               ENDDO
37            ENDDO
38
39   C        Push atoms
40            DO i = 1, atomMax
41               x(i) = x(i) + delta(f(i))
42            ENDDO
43         ENDDO
44         END
```

Figure 2: Fortran D version of Non-Bonded Force kernel (with coordinates and forces shown 1-D for simplicity). During each of the `stepMax` time steps, forces are first reset to zero, then they are computed based on the distances between atoms (similar to the code in Figure 1), and finally the forces are used for updating coordinates. Note the index-based distribution directive in line 12, followed by a value-based redistribution in line 18.

with a distribution, both at compile time and at run time. This already implies some simplicity for both the programming assignment and the resulting program. Many distribution dependent decisions, like the effect of an owner-computes rule (which links the distribution of computation to distribution of the data involved), can already be resolved before run time. The amount of additional code and variables for computing and storing the distributions and for mapping between global indices, local indices, and processor numbers is very small.

Given a value-based distribution, which for example distributes `x` according to its values, one could envision a scheme that also had very little state specifically devoted to representing the distribution. For example, one could compute the owner of some `x(i)` on the fly by sorting all elements of `x` and determining the position of `x(i)` in the sorted list. This, however, would clearly be impractical to do for each reference to an element of `x`. Instead, one should amortize the cost of determining ownership etc. by computing this information once and reusing it. (Note that this is not necessarily true for all irregular distributions. In case an application has good index-based locality, one might still want to distribute data irregularly to improve load balance, but some or all of the mapping computation can be done on the fly [Bia91, Han92, BK93, CHMS94].)

Representing a value-based distribution explicitly requires a large amount of state. A *translation table* maps global indices i_{glob} into pairs (i_{loc}, p) of local indices and processor numbers. Often the translation table itself is too large to be fully replicated and is also distributed [WSBH91]. Therefore, not only storing but also accessing the information adds complexity to the program. As mentioned in Section 2, runtime libraries such as CHAOS can take most of the complexity of this task from the programmer [DHU+93], but their use still requires explicit managing of the data structures associated with irregular distributions and communications.

3.2 Storing Value-Based Distributed Data

Again assuming that `x` is distributed according to its value, the number of elements of `x` assigned to each processor typically varies and is not known until run time. This poses particular problems when using a language that does not support dynamic memory allocation, like Fortran 77. Common strategies for circumventing this restriction are to make arrays conservatively large or declare work arrays that are shared by several variables, both of which have obvious disadvantages.

Note that the same problem occurs when regularly distributed arrays are accessed irregularly and we wish to append buffer space for off-processor data at the end of the array [Han93].

3.3 Mapping Name Spaces

Mapping between the global name space of a Fortran D program and the local name space of the node program is an important component of the parallelization process. For regular distributions, most of this task can be performed before run time, or, if run time translation is needed, the necessary code can be generated fairly easily; for example, a statement `x(i) = i`, where `i` is global, might be translated into something like `x(i) = i + my$proc*block_size`, with a local `i`.

For irregular distributions, the mapping has to be delayed until run time, and computing the mapping may be complicated.

3.4 Communication Generation

Again due to the complicated relationship between global and local name spaces, generating correct communication statements in the presence of value-based distributions can be tricky even for regular array accesses (such as `x(i) = x(i+1)`), and even more so for irregular references (like `x(a(i)) = x(b(i))`).

Resolving such references requires several mapping steps, some of which may themselves involve communication. To still generate efficient code, one should precompute and reuse as much of this information as possible. For example, the *inspector-executor* paradigm allows to message-vectorize low-locality data accesses, even in the absence of compile time knowledge. First, an "inspection phase" determines what data have to be communicated within a certain loop and generates a communication schedule. Then, an "execution phase" (repeatedly) performs the actual computation and uses the communication schedule to exchange data [MSS+88, KMV90].

3.5 A Bootstrapping Problem

One characteristic of value-based distributions is that they may pose a certain bootstrapping problem to both the user and the compiler. This is the case when an array is distributed based on its own values, which is considered perfectly legal and actually the case in the code shown in Figure 2. Here an array `x` is aligned to a decomposition `atomD`, which in turn gets distributed based on the values of `x`. So, how does one know from which processor to get the value of some `x(i)` if one doesn't the value of `x(i)` yet ? To resolve this problem, it is required that one starts out with a different, typically regular distribution, which can be used for example to initialize data as is the case in Figure 2.

4 Value-Based Distributions from the Compiler's Perspective

After outlining in the previous section the general issues associated with distributing data based on values, this section describes the implications of using value-based distributions in a data-parallel language such as Fortran D. The program shown in Figures 3 and 4, which was generated by the Fortran D compiler from the NBF kernel in Figure 2, will serve as an example.

4.1 The Input Language

Since we implemented value-based distributions as part of a Fortran D compiler prototype, the Fortran D language constructs also serve as a basis for the syntax of value-based distributions. These extensions could also be applied directly to the HPF standard [KLS+94]. We were able to limit ourselves to a simple extension of the already existing `DISTRIBUTE` directive, as was illustrated by the code in Figure 2. Here the directive `DISTRIBUTE atomD(VALUE(DIM=1,VALS=x,WEIGHT=inb))` is the only statement the programmer had to add in order to express the value-based locality of the application; the rest is done by the compiler. The value-based distribution syntax currently supported is shown in Figure 5.

4.2 Distributing the Data

Each value-based distribution directive D results in certain tasks that the compiler, in the current implementation, performs at the location of the directive itself:

1. Generate code for calculating the new distribution as follows.

   (a) Compute an array that maps local indices (based on the initial distribution) to processor numbers, *D.loc2proc*, and the resulting number of owned data, *D.cnt*, by calling a

```
      PROGRAM nbf

      INTEGER i, j, p, t, n$proc, atomMax, pMax, stepMax
      PARAMETER (n$proc=8)
      PARAMETER (atomMax=8000, pMax=250, stepMax=30)
      INTEGER inb(1000), partners(1000, pMax)
      REAL x(1000), f(1000), force

C     General Fortran D variable declarations
      COMMON /FortD/ n$p, my$p
      INTEGER n$p, my$p, numnodes, mynode

C     Irregular Fortran D variable declarations
      INTEGER atomD$loc2proc(1000)
      INTEGER x$sched, x$tab, x$offsize, j$cnt, i$, atomD$cnt, atomD$tab, atomD$sched

C     Fortran D initializations
      my$p = mynode()
      n$p = numnodes()
      IF (n$p .NE. 8) STOP

C     Initialize data
      CALL read_data(x, inb, partners)

C     Redistribute atomD according to coordinate values
      atomD$cnt = 1000
      Compute atomD$loc2proc, atomD$cnt from inb, x
      Compute atomD$tab from atomD$loc2proc, atomD$cnt
      Allocate atomD$loc2glob(atomD$cnt)
      Compute atomD$loc2glob, atomD$sched from atomD$tab, atomD$cnt
      Delete atomD$tab
      Resize inb(atomD$cnt), x(atomD$cnt), f(atomD$cnt), partners(atomD$cnt, pMax)
      Shuffle inb, x, partners according to atomD$sched

C     Counting slice for j$cnt
      j$cnt = 0
      DO i = 1, atomD$cnt
         j$cnt = j$cnt + inb(i)
      ENDDO
      Allocate j$glob(j$cnt), j$loc(j$cnt)
```

Figure 3: NBF kernel, output of Fortran D compiler (continued in Figure 4). The code is shown before converting references to dynamically allocated arrays into work array accesses. Variables containing a "$" are generated by the compiler. The CHAOS library provides run-time support.

```
C     Compute j$glob
      j$cnt = 0
      DO i = 1, atomD$cnt
         DO p = 1, inb(i)
            j$cnt = j$cnt + 1
            j = partners(i, p)
            j$glob(j$cnt) = j
         ENDDO
      ENDDO
      Compute x$tab from atomD$loc2glob, atomD$cnt
      Compute x$sched, j$loc, x$offsize from x$tab, j$glob, j$cnt, atomD$cnt
      Delete x$tab
      Resize x(atomD$cnt + x$offsize)
      Resize x(atomD$cnt + x$offsize)

C     Loop over timesteps
      DO t = 1, stepMax

C        Reset forces to zero
         DO i = 1, atomD$cnt + x$offsize
            f(i) = 0
         ENDDO

         Gather x using x$sched
C        Compute forces
         i$ = 0
         DO i = 1, atomD$cnt
            DO p = 1, inb(i)
               i$ = i$ + 1
               j = partners(i, p)
               force = nbforce(x(i), x(j$loc(i$)))
               f(i) = f(i) + force
               f(j$loc(i$)) = f(j$loc(i$)) - force
            ENDDO
         ENDDO
         Scatter_add f using x$sched

C        Push atoms
         DO i = 1, atomD$cnt
            x(i) = x(i) + delta(f(i))
         ENDDO
      ENDDO
      END
```

Figure 4: NBF kernel generated by the Fortran D compiler, continued from Figure 3.

DistDirective	**is**	`DISTRIBUTE` *decomposition-name* (`VALUE` (*DistParams*))
DistParams	**is**	*ValArrays* [, *Weight*] [,*Strategy*]
ValArrays	**is**	[`DIM =` *num-dims*,] [`VALS =`] *value-array-name* [, *value-array-name*] ...
Weight	**is**	[`WEIGHT =`] *weight-array-name*
Strategy	**is**	[`STRATEGY =`] *distribution-strategy-name*

Constraint: Number of value-array-names must equal one, or, if specified, *num-dims*.

Figure 5: Syntax of the value-based distribution directive. Square brackets indicate optional components. For example, as shown in Figure 2, the user may not select a specific strategy for distributing data. In this case the compiler chooses a default strategy, like Recursive Bisection. Furthermore, since the default number of dimensions is one and some key words are optional, the `DISTRIBUTE atomD(VALUE(DIM=1,VALS=x,WEIGHT=inb))` could also have been abbreviated as `DISTRIBUTE atomD(VALUE(x, inb))`.

partitioner (line 27 in the example). This partitioner applies the strategy specified by the programmer, or as a default by the compiler, to the values and, if specified, weights provided by the user and generates a distribution. The partitioner tries to distribute the data such that both high locality and good load balance are achieved.

(b) Based on *D.loc2proc* and *D.cnt*, compute a translation table (see Section 3.1), *D.tab* (line 28).

(c) Allocate an array that maps local to global indices, *D.loc2glob*, of size *D.cnt* (line 29).

(d) Compute *D.loc2glob* and a communication schedule *D.sched* from *D.loc2proc* and *D.cnt* by calling a remapper (line 30). This schedule is used in the communication statements that reshuffle coordinates and pair list data according to the new distribution (line 33).

2. Compute the set of arrays, ARR, that are aligned to the decomposition being distributed. Resize each $arr \in ARR$ according to *D.cnt* (line 32). As already indicated in Section 3.2, this implies a need for dynamic memory allocation capabilities. The Fortran D compiler emulates these by converting arrays that are dynamically allocated or resized to work array accesses. (However, for sake of readability, Figures 3 and 4 show the code before memory allocation). The offsets into these work arrays are generated at run time by calls to a separate memory allocation library.

   This also has to consider multidimensional arrays, such as `partners` in the example. In the prototype compiler, these arrays reference the work array section assigned to them by performing array arithmetic explicitly. For example, a reference to `partners(i, p)` becomes something like `i$wrk(partners$ind + p + (i - 1) * 200)`. Note that this does not increase the overall instruction count, it only makes explicit the index calculations that are normally hidden from the user.

3. For each $arr \in ARR$, determine whether arr is live at the location of D. If it is, then generate code for communicating the elements of arr from their old owners to their new owners (line 33). For example, we do not need to communicate `f` in the example, since forces have not been computed yet.

4.3 Satisfying Off-Processor References

Given a (regular or irregular) reference to an irregularly distributed array that might reference off-processor data, an inspector has to generate a communication schedule(see Section 3.4). In our example, the references to `x(j)` and `f(j)` from Figure 2 may require communication. Since in these references the same subscript expression references conformingly distributed arrays, one inspector (lines 35 ... 59) generating one schedule is sufficient. This schedule is used in the communication statements that gather coordinates (line 69) and add forces across processors (line 81). The inspector is relatively similar to irregular references to regularly distributed arrays, and we refer to other publications for a more detailed discussion [DSvH93, Han93]. One additional complication, however, is that a translation table has to be computed based on *D.loc2glob* (line 55). This implies the need for solving the *reaching decompositions problem* both intra- and inter-procedurally [HHKT92].

Note that the current strategy for collecting off-processor references is to record each individual subscript in a *trace array*. This trace array is created in global name space (`j$glob`, see lines 47 ... 54) and then converted into local name space (`j$loc`, line 56). Subsequent irregular subscripts (such as `x(j)`, line 33 in Figure 2) are then converted to trace references (`x(j$loc(i$))`, line 76). Since the name-space translation is already implicit in the trace array, this approach results in a relatively fast executor. However, the space requirements for storing the traces are proportional to the total number of references, instead of just the number of elements referenced. Compile time analysis can be used to shorten traces; in the example, the compiler determined that the reference pattern is the same in each time step and therefore did not include the time stepping loop in the inspector. However, the traces may still become too space consuming, in which case more space saving alternatives, such as a hash table combined with name-space translations on the fly, may be used [DSvH93].

4.4 The Actual Computation

After distributing data and prefetching off-processor references, the actual computation can be performed (lines 62 ... 87). The following are some of the issues arising here.

- When reducing loop bounds to parallelize a loop based on the owner computes rule, the number of local elements, *D.cnt* (computed in line 27), has to be retrieved. In the NBF kernel, this is the case in the `i` loops.
- Subscripts of references that might access off-processor elements have to be converted to localized trace array references (see Section 4.3). In the example, the references to `x(j)` and `f(j)` are converted to `x(j$loc(i$))` and `f(j$loc(i$))`, respectively, as shown in lines 76 and 78.
- To regenerate the global iteration index from the local name space index, the array *D.loc2glob* (from line 30) has to be consulted.
- To map global indices to processor numbers and local indices, for example when printing a certain data element, a dereferencing call using the translation table *D.tab* must be generated.

5 Experimental Results

The following questions are possible criteria for evaluating a certain language/compiler feature:

1. What additional expressiveness does the language feature provide for the user and the compiler ?

2. How well can the compiler take advantage of this additional expressiveness ? *I.e.*, how does the compiler generated code compare to hand coded parallel programs ?

While we do not attempt to fully answer these questions here, we will try to at least informally argue that for certain applications, such as the Molecular Dynamics code example presented here, value-based distributions can result in substantial performance gains.

For this kernel, the Fortran D compiler generated output is practically identical to a hand coded, on the same CHAOS run-time support relying parallel program. Therefore we are mostly interested in the performance gain obtained by value-based distributions relative to index-based distributions. Our experiments were based on the Fortran-D kernel from Figure 2 (but with 3-D instead of just 1-D coordinates and forces) with varying index- and value-based distribution directives as listed below. For each distribution strategy, the output of the Fortran-D compiler was compiled with `if77` onto an iPSC/860 with 32 nodes and 8 Megabytes of memory per node. The subsequent parallel runs used pairlist data and physical coordinates from an SOD simulation using an 8 Å cutoff radius; SOD itself is of size $53 \times 55 \times 52$ Å^3. SOD (superoxide dismutase) is a catalytic enzyme that converts the toxic free-radical, O_2^{-4}, a byproduct of aerobic respiration, to the neutral molecules O_2 and H_2O_2 [WCSM93]. Our solvated system contained 6968 atoms. We ran the simulation for 30 time steps, on 1, 2, 4, 8, 16, and 32 processors, with the following distribution strategies.

1. Index-based, block-wise: for N atoms and P processors, processor p gets assigned atoms $(p-1)N/P + 1 \ldots pN/P$ (assuming P divides N). This corresponds to using the original `BLOCK` distribution (line 12) throughout the run (*i.e.*, no redistribution in line 16).

2. Value-based, 1-dimensional bisection, no load balancing: the physical problem domain gets divided along the x-axis, assigning each processor an equal number of atoms. This corresponds to redistributing data with a "`DISTRIBUTE atomD(VALUE(DIM=1,VALS=x))`" directive (*i.e.*, no "`WEIGHT = inb`" parameter).

3. Value-based, 1-dimensional bisection, with load balancing: each atom's work load is measured by the number of interaction partners, and the partitioner divides the physical problem domain such that each processor has an even workload. The appropriate directive is already shown in Figure 2.

4. Value-based, 3-dimensional bisection, no load balancing: the physical problem domain gets recursively divided along the x, y, and z-axes, assigning each processor an equal number of atoms. The directive here is "`DISTRIBUTE atomD(VALUE(DIM=3,VALS=x, y, z)).`"

5. Value-based, 3-dimensional recursive bisection, with load balancing. This is achieved with a "`DISTRIBUTE atomD(VALUE(DIM=3,VALS=x, y, z, WEIGHT = inb))`" statement.

We were able to run all strategies on all processors sizes, except for 1 and 2 processors, where the value-based strategies where too memory intensive.

5.1 Different Distributions

Before measuring the performances of the different distribution strategies, it is instructive to visualize the resulting atom-to-processor mappings.

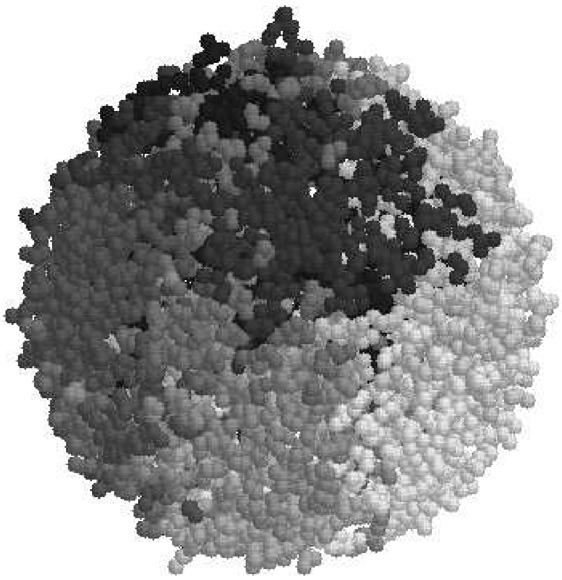

Figure 6: Index-based `BLOCK` distribution of atoms. Atoms are colored according to the processor they are mapped to, for an eight-processor configuration.

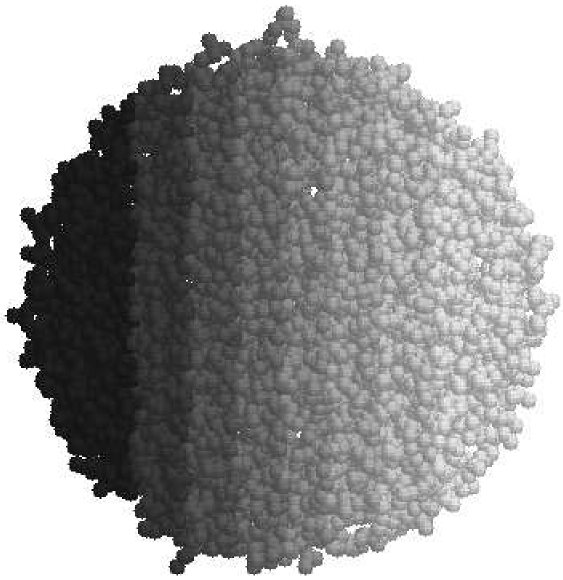

Figure 7: Value-based distribution of atoms along just one dimension.

Figures 6, 7, and 8 show the mappings resulting from strategies 1, 3, and 5, respectively. We see that the index-based strategy 1 has poor locality, each processor owns atoms scattered throughout the molecule. This is improved with strategy 3, where each processor's subdomain is a contiguous region in space. However, since these subdomains are constructed by cuts along just one dimension, they still have a relatively large surface-to-volume ratio resulting in higher communication requirements. This is improved upon in strategy 5, where cuts are made along all three dimensions and each subdomain is relatively compact.

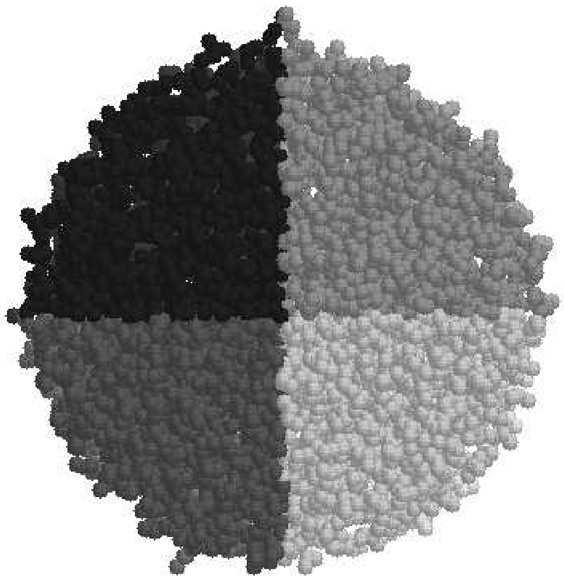

Figure 8: Value-based distribution of atoms along all three dimensions.

5.2 Locality

The number of not-owned atoms accessed by the individual processors is a measure for the inter-processor access locality of the NBF kernel. We can estimate a theoretical lower bound on this number based on the total number of atoms (6968), the space they occupy (a sphere with a little over 100 Å in diameter), and R_{cut} (8 Å), which yields that around 950 atoms will have to be buffered.

Figure 9 compares the maximum numbers of buffered atoms for different machine sizes and distribution strategies. We see that the 3-dimensional, value-based distribution performs best, with a slight additional advantage for the load balanced version, which comes very close to the predicted lower bound.

5.3 Balancedness

Figure 10 measures balancedness by comparing maximum and average workloads, expressed as floating point operation counts. The index-based distribution gets less balanced as the number of processors goes up. Value-based distributions perform consistently better, and taking the work load into account provides a further improvement. For this application, the index-based BLOCK distribution is particularly unsuited due to the diagonal nature of the adjacency matrix (based on Newtons Third Law), which shifts the work load towards atoms with smaller indices.

5.4 Individual Timings

Figure 11 shows the timings measured for the 3-dimensional, balanced recursive bisection. In the processor ranges measured, the overheads associated with preprocessing, redistributing data, and communication are about an order of magnitude lower than the cost of the actual computation. The cost of preprocessing (including the inspector which enables message vectorization) gets well amortized for the given number of time steps (ten). However, it is also apparent that the scalability of the run-time support may become critical for larger numbers of processors.

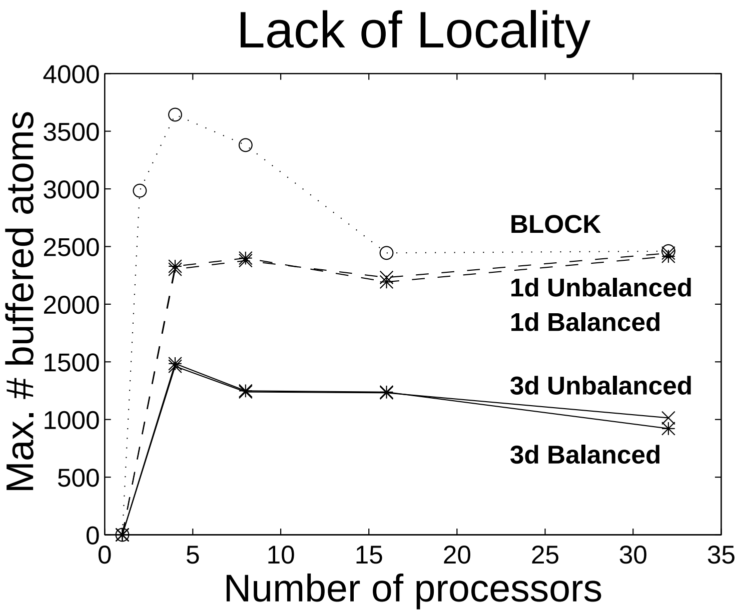

Figure 9: The number of communicated data (maximum across processors), for varying machine sizes and distribution strategies.

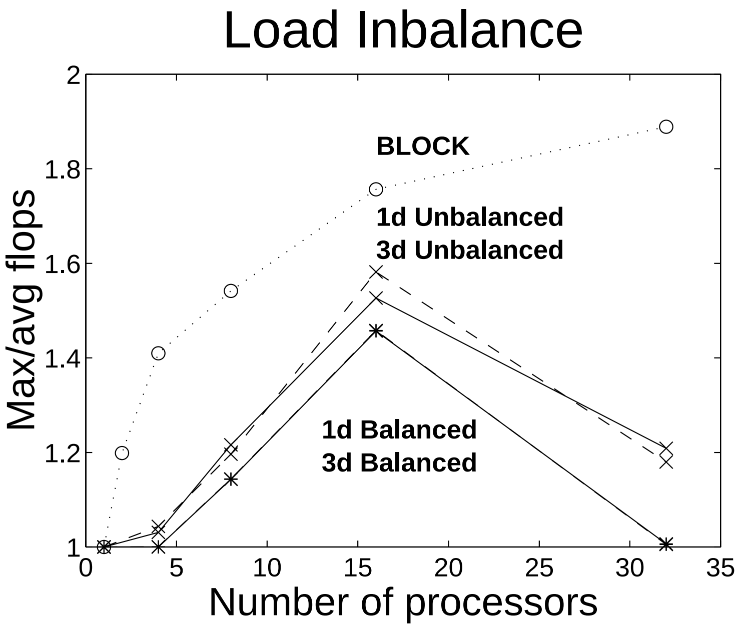

Figure 10: The fraction of maximum floating point operations (*i.e.*, workload of the slowest processor) and of the average across processors.

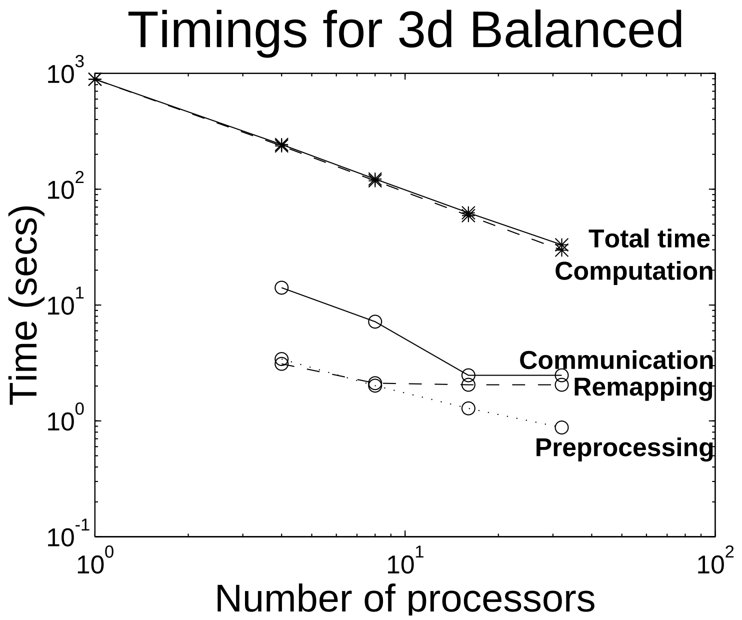

Figure 11: The timing breakdown (excluding I/O) for varying machine sizes, using a 3-dimensional, value-based balanced bisection.

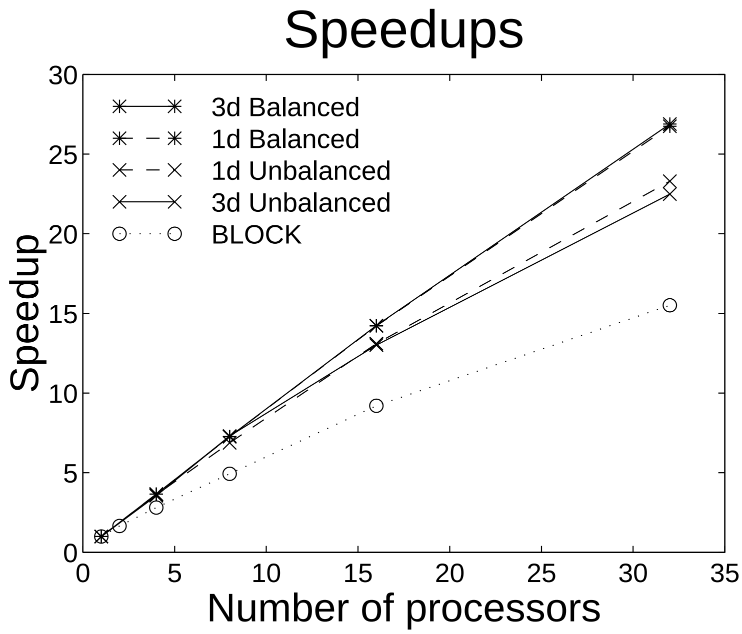

Figure 12: The speedup obtained with different processor numbers and distribution strategies.

5.5 Speedups

Figure 12 summarizes the speedups obtained for the various distribution methods. The value-based distributions outperform the index-based distribution, and explicit load balancing provides an additional advantage.

6 Related Work

The feasibility of automatically generating inspectors/executors for simple loop nests has been demonstrated by the ARF [WSBH91], KALI [KM91] and SUPERB [BGSZ92] compilers.

Slicing analysis is a technique for generating inspectors and pre-inspectors for multiple levels of indirection, which is another typical characteristic of irregular problems [DSvH93].

Run time iteration graphs can assist in improving load balance and access locality when distributing loop iterations across processors [PSC93a].

Even though the classic data-parallel paradigms as defined by Fortran D-like languages did not provide any means for directly exploiting value-based locality, application programmers have been taking advantage of this kind of locality when hand-coding in low-level parallel languages (for example with message passing). They have developed a wealth a different mapping strategies, both value-based [BB87, HS91, DHU+93], and connectivity based [Sim91], which can be use as mapping strategies for irregular distributions.

LPAR is a programming model for implementing numerical algorithms with a local structure, such as Particle-in-Cell or Multigrid, on distributed memory MIMD multiprocessors [BK93]. Given a data structure which already reflects the physical locality characteristics of the underlying problem, it distributes data and computation in a load balanced fashion.

Value-based distributions were initially proposed as an enhancement of Fortran 77D [Han92]. A variant of it, based on a *GeoCoL* (**Geo**metrical, **Co**nnectivity and/or **L**oad) data structure, has since then been implemented in a Fortran 90D prototype compiler by Ponnusamy et al. [PSC93a, PSC+93b].

The GIVE-N-TAKE framework is a generalization of partial redundancy elimination techniques that can be used for analyzing data access requirements and for efficient communication placement in the presence of arbitrary control flow and reference patterns [HK93].

Examples of SIMD specific work are the *Communication Compiler* [Dah90] for the Connection Machine and the *loop flattening* transformation to allow efficient execution of loop nests with varying inner loop bounds [HK92].

7 Summary and Conclusions

Data-parallel languages, such as HPF or Fortran D, promise a user-friendly, efficient programming environment for parallel machines. However, one of their weaknesses so far has been the lack of support for irregular problems. Here access patterns and work load distributions are not very amenable to the index and iteration space oriented data parallel paradigm.

Value-based distributions provide a specific kind of expressiveness that had been missing so far; they can give the compiler locality information that cannot be described in terms of array indices. While this is only one aspect of parallelizing irregular problems, we believe that it widens the range of important application that can be implemented efficiently in data-parallel languages. Furthermore, it appears that other problems and approaches specific to irregular problems, such as connectivity based distributions or irregular iteration space partitionings, may also profit from the techniques described in this paper.

Acknowledgements

We thank Seema Hiranandani and Chau-Wen Tseng, who developed the Fortran D compiler support for regular problems. Paul Havlak provided much of the symbolic analysis used in the Fortran D compiler.

We appreciate the help from Raja Das, Ravi Ponnusamy, and Yuan-Shin Hwang, who developed much of the compiler/run time technology involved in this project. Jim Humphries wrote the dynamic memory allocator and generated the molecule graphics.

Special thanks to Terry Clark, Ridgway Scott, and Bernie Brooks, who offered their broad expertise with molecular dynamics and pointed out many practical issues when compiling irregular real-world codes. Terry also provided the molecule test data used for the experiments in Section 5.

References

[BB87] M. J. Berger and S. Bokhari. A partitioning strategy for non-uniform problems on multiprocessors. *IEEE Transactions on Computers*, C-36(5):570–580, 1987.

[BBO+83] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A program for macromolecular energy, minimization and dynamics calculations. *Journal of Computational Chemistry*, 4(2):187–217, 1983.

[BCF+93] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. A compilation approach for Fortran 90D/HPF compilers on distributed memory MIMD computers. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[BGSZ92] P. Brezany, M. Gerndt, V. Sipkova, and H. Zima. SUPERB support for irregular scientific computations. In *Scalable High Performance Computing Conference*, pages 314–321, Williamsburg, VA, April 1992.

[Bia91] E. S. Biagioni. *Scan Directed Load Balancing*. PhD thesis, University of North Carolina at Chapel Hill, 1991.

[BK93] S. B. Baden and S. R. Kohn. Portable parallel programming of numerical problems under the LPAR system. Technical Report CS93-330, Department of Computer Science and Engineering, University of California, San Diego, 1993.

[CCH+88] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. *The International Journal of Supercomputer Applications*, 2(4):84–99, Winter 1988.

[CHMS94] T. W. Clark, R. v. Hanxleden, J. A. McCammon, and L. R. Scott. Parallelization using spatial decomposition for molecular dynamics. In *Scalable High Performance Computing Conference*, Knoxville, TN, May 1994. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93356-S`.

[CM90] T. W. Clark and J. A. McCammon. Parallelization of a molecular dynamics non-bonded force algorithm for MIMD architectures. *Computers & Chemistry*, 14(3):219–224, 1990.

[CMZ92] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[Dah90] D. Dahl. Mapping and compiled communication on the connection machine system. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[DHU+93] R. Das, Y.-S. Hwang, M. Uysal, J. Saltz, and A. Sussman. Applying the CHAOS/PARTI library to irregular problems in computational chemistry and computational aerodynamics. In *Proceedings of the Scalable Parallel Libraries Conference, Mississippi State University, Starkville, MS*, October 1993.

[DMS+92] R. Das, D. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives, AIAA-92-0562. In *Proceedings of the 30th Aerospace Sciences Meeting*. AIAA, January 1992.

[DSvH93] R. Das, J. Saltz, and R. v. Hanxleden. Slicing analysis and indirect accesses to distributed arrays. In U. Banerjee et al., editor, *Lecture Notes in Computer Science*, volume 769, pages 152–168. Springer, Berlin, August 1993. From the *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93319-S`.

[FHK+90] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990. Revised April, 1991.

[GB88] W. F. van Gunsteren and H. J. C. Berendsen. GROMOS: GROningen MOlecular Simulation software. Technical report, Laboratory of Physical Chemistry, University of Groningen, Nijenborgh, The Netherlands, 1988.

[Han92] R. v. Hanxleden. Compiler support for machine independent parallelization of irregular problems. Technical Report CRPC-TR92301-S, Center for Research on Parallel Computation, November 1992. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR92301-S`.

[Han93] R. v. Hanxleden. Handling irregular problems with Fortran D — A preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993. D Newsletter #9, available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93339-S`.

[HHKT92] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[HK92] R. v. Hanxleden and K. Kennedy. Relaxing SIMD control flow constraints using loop transformations. In *Proceedings of the ACM SIGPLAN '92 Conference on Program Language Design and Implementation*, pages 188–199, San Francisco, CA, June 1992. ACM Press. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR92207-S`.

[HK93] R. v. Hanxleden and K. Kennedy. A code placement framework and its application to communication generation. Technical Report CRPC-TR93337-S, Center for Research on Parallel Computation, October 1993. Available via anonymous ftp from `softlib.rice.edu` as `pub/CRPC-TRs/reports/CRPC-TR93337-S`.

[HKK+91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[HS91] R. v. Hanxleden and L. R. Scott. Load balancing on message passing architectures. *Journal of Parallel and Distributed Computing*, 13:312–324, 1991.

[KLS+94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[KM91] C. Koelbel and P. Mehrotra. Programming data parallel algorithms on distributed memory machines using Kali. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[KMCKC93] U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle. Automatic data layout for distributed-memory machines in the D programming environment. In Christoph W. Kessler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*, pages 136–152. Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden, Germany, 1993. Also available as technical report CRPC-TR93-298-S, Rice University.

[KMV90] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[MSS+88] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the Second International Conference on Supercomputing*, pages 140–152, St. Malo, France, July 1988.

[PSC93a] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Supercomputing '93*, pages 361–370. IEEE Computer Society Press, November 1993. Technical Report CS-TR-3055 and UMIACS-TR-93-32, University of Maryland, April '93. Available via anonymous ftp from `hyena.cs.umd.edu`.

[PSC+93b] R. Ponnusamy, J. Saltz, A. Choudhary, Y.-S. Hwang, and G. Fox. Runtime support and compilation methods for user-specified data distributions. Technical Report CS-TR-3194 and UMIACS-TR-93-135, University of Maryland, November 1993. Available via anonymous ftp from `hyena.cs.umd.edu`.

[SH91] J. P. Singh and J. L. Hennessy. An empirical investigation of the effectiveness and limitations of automatic parallelization. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, Tokyo, Japan, April 1991.

[Sim91] H. Simon. Partitioning of unstructured mesh problems for parallel processing. In *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*. Pergamon Press, 1991.

[SM90] T. P. Straatsma and J. Andrew McCammon. ARGOS, a vectorized general molecular dynamics program. *Journal of Computational Chemistry*, II(8):943–951, 1990.

[Tse93] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, January 1993.

[WCSM93] Y.-T. Wong, T. W. Clark, J. Shen, and J. A. McCammon. Molecular dynamics simulation of substrate-enzyme interactions in the active site channel of superoxide dismutase. *Journal of Molecular Simulation*, 10(2–6):277–289, 1993.

[WSBH91] J. Wu, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. ICASE Report 91-13, Institute for Computer Application in Science and Engineering, Hampton, VA, January 1991.