Fortran Tools Newsletter #12

Value-Based Distributions and Alignments in Fortran D

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 May 1994. From the *Journal of Programming Languages*, Special Issue on Compiling and Run-Time Issues for Distributed Address Space Machines, to appear.

Value-Based Distributions and Alignments in Fortran D

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 language, like Fortran D or High Performance Fortran (HPF), presents a challenge of growing importance. A major component of data-parallel programming is the data *mapping*, *i.e.*, data distribution and alignment. Mappings may be either regular, in which case they correspond to simple functions between array indices and owning processors, or irregular. Regular mappings can already be specified fairly conveniently, with keywords such as `BLOCK` or `CYCLIC`. However, the mechanisms proposed for irregular mappings so far require the programmer to go down to a very low level, for example by specifying explicit mapping arrays or mapping functions. For irregular problems, this makes it difficult to achieve good load balance or low storage and communication requirements without sacrificing programming convenience.

Value-based mappings extend classical, index-based mappings to 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 mappings in a Fortran 77D compiler prototype. We propose a natural extension of index-based mappings 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 mappings for a Molecular Dynamics application and an Unstructured Mesh kernel.

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 typically 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 to 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 mapping 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

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

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 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 [Tse93].

However, irregular problems, such as unstructured mesh codes, molecular dynamics, galaxy simulations, or computational fluid dynamics, have turned out to be significantly harder to parallelize [SH91]. Common problems are low access locality, both within and across data structures, and poor load balance. 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, they may be 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 mapped 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 mappings (like BLOCK, CYCLIC, or BLOCK-CYCLIC) and remappings 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 mappings. 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 does not directly reflect their topology; *i.e.*, it is hard to predict whether two vertices are linked together by just looking at their indices. 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 mapping schemes.

The fact that the vertex numbering does not reflect the mesh topology is an example of poor index-locality *within* a data structure, which we consider a data *distribution* problem. However, we may also have bad locality *across* data structures. Revisiting the mesh example, we are typically operating on both vertices and edges which have a certain interrelationship (each edge has two particular vertices as end points). This relationship can usually not be determined from the node and edge numbering. However, it would generally be advantageous if the data associated with an edge would be stored on the same processor as the data associated with their end points; we consider this a data *alignment* problem.

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 Section 2, naturally lends itself to *value-based mappings*. These mappings can be used to improve locality and also to increase load

```
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. For simplicity, the coordinate and force arrays, x and f, are shown only one-dimensional.

balance. This paper reports on an implementation of value-based mappings within the Fortran D compiler prototype at Rice University. Here we will focus on the compiler technology specific to handling such mappings 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 mappings with kernels taken from a Molecular Dynamics code and an Unstructured Mesh application. Section 3 lists the implications of value-based mappings for message-passing node programs. Section 4 describes language extensions and compiler technology for generating such node programs. Section 5 contains experimental results obtained for the kernels introduced in Section 2 and compares the effectiveness of index-based and value-based mappings for these applications. Section 6 discusses related work; Section 7 concludes with a brief summary and discusses possible extensions.

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 $l = j$). In this context, "related" refers to a preference towards residing close to each other with respect to the memory hierarchy, *e.g.*, on the same processor. 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 mapping is not a correctness but only an efficiency issue, we are usually more interested in a fast heuristic for finding a reasonably good data mapping 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

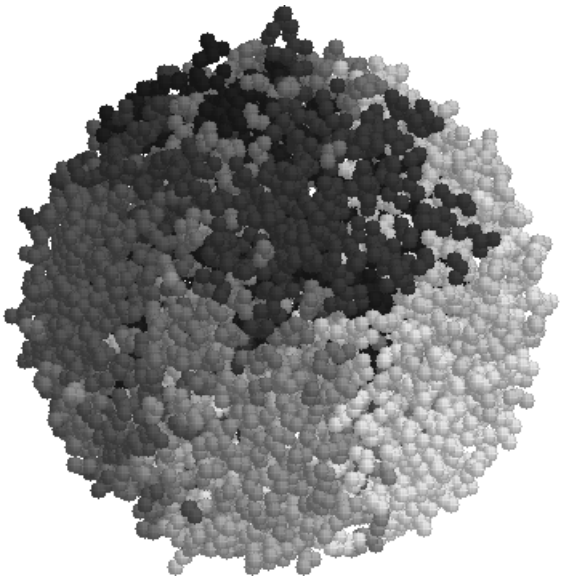

Figure 2: `BLOCK` mapping of an SOD system. Atoms are colored according to the processor they are mapped to, for an eight-processor configuration. The regular, index-based mapping results in assigning each processor very irregular subdomains, with accordingly poor locality and load balance.

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 according to the values 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 mapping would lose 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 mapping typically results in each processor accessing all data, with accordingly high communication and storage requirements. For example, Figure 2 shows the mapping resulting from distributing an SOD system ($N_{atoms} = 6968$) across eight processors[1].

Value-based mappings allow the programmer to express value-based locality as a simple extension of the regular mapping mechanism employed in data-parallel languages. We distinguish between value-based distributions and value-based alignments, corresponding to the data distribution and alignment problems identified in Section 1. For example, let array `x` be aligned to a decomposition `decomp`. Then the directive `DISTRIBUTE decomp(VALUE(x))` is a *value-based distribution* specifying 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 mappings.) In other words, if the values of two elements of

[1]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].

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 3, 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. Figures 4 and 5 show the SOD mappings resulting from one- and three-dimensional value-based mappings, respectively.

An example of a *value-based alignment* is shown in Figure 6, which shows a Fortran D version of a sweep over the edges of an unstructured mesh. There are two decompositions, `nodeD` for the node data and `edgeD` for the edge data. After reading in the data, first `nodeD` gets distributed by value according to the node coordinates, then `edgeD` gets aligned with `nodeD` according to the values of the topology arrays `ends1` and `ends2`.

3 Handling Value-Based Mappings

The MIMD Fortran D compiler transforms a Fortran D program, written in a global name space and annotated with data mapping 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).

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 with a mapping, both at compile time and at run time. This already implies some simplicity for the programming assignment and for the resulting program. Many mapping-dependent decisions, like the effect of an owner-computes rule (which links the mapping of computation to mapping of the data involved), can already be resolved before run time. The amount of additional code and variables for computing and storing the mappings and for translating 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. If an application has good index-based locality, then 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 [BK93, Han92, Bia91, CHMS94].)

```
      PROGRAM nbf

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

C     Fortran D directives
10    DECOMPOSITION atomD(atomMax)
      ALIGN inb, x, f, partners(i,j) WITH atomD(i)
      DISTRIBUTE atomD(BLOCK)

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

C     Redistribute atomD according to coordinate values
      DISTRIBUTE atomD(VALUE(DIM=1, VALS=x, WEIGHT=inb))

20 C  Loop over timesteps
      DO t = 1, stepMax

C        Reset forces to zero
         DO i = 1, atomMax
25          f(i) = 0
         ENDDO

C        Computes forces
         EXECUTE (i) ON_HOME f(i)
30       DO i = 1, atomMax
            DO p = 1, inb(i)
               j = partners(i, p)
               force = nbforce(x(i), x(j))
               f(i) = f(i) + force
35             f(j) = f(j) - force
            ENDDO
         ENDDO

C        Push atoms
40       DO i = 1, atomMax
            x(i) = x(i) + delta(f(i))
         ENDDO
      ENDDO
      END
```

Figure 3: Fortran D version of the 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.

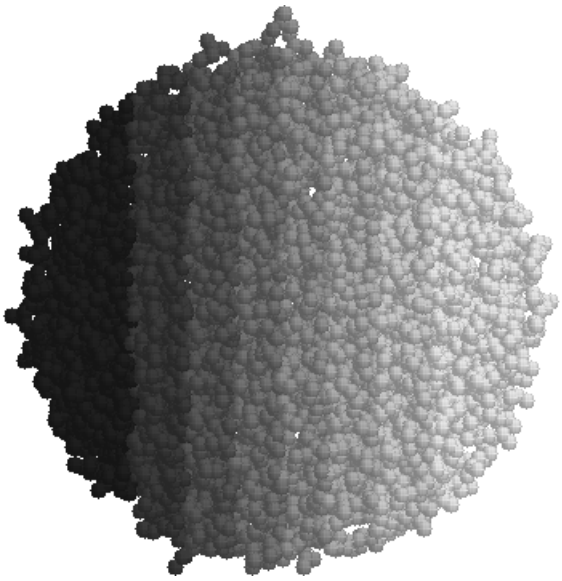

Figure 4: Value-based mapping of atoms along just one dimension. Locality is good, but communication may be expensive due to the high surface-to-volume ratio.

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 distributed instead [WSBH91]. Therefore, not only storing but also accessing the information adds complexity to the program. As mentioned in Section 2, run-time 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 Translating Name Spaces

Translating 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 mappings, 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`.

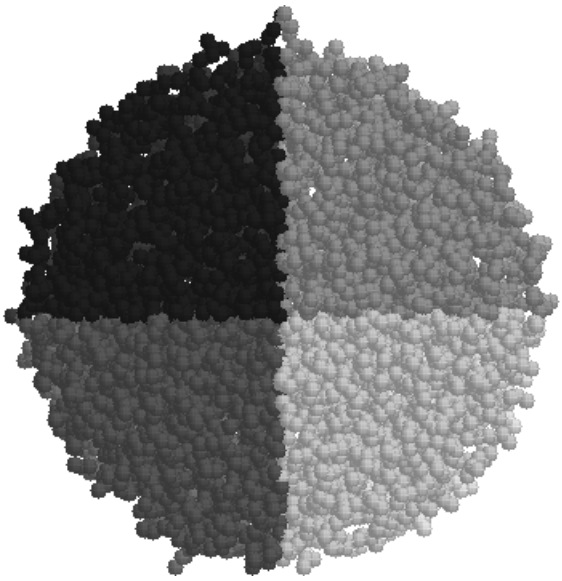

Figure 5: Value-based mapping of atoms along all three dimensions. Locality is good, and the compact subdomains minimize communication costs.

For irregular mappings, the translation has to be delayed until run time, and computing the translation 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 mappings 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 translation 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 us to message-vectorize low-locality data accesses, even in the absence of compile-time knowledge [MSS+88, KMV90]. 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.

3.5 A Bootstrapping Problem

One characteristic of value-based mappings is that they may pose a certain bootstrapping problem to both the user and the compiler, as has already been identified by Ponnusamy et al. [PSC93a]. This problem occurs 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 3. Here an array `x` is aligned to a decomposition `atomD`, which in turn gets distributed based on the values of `x`. Now, when we start initializing `x`, we need to know its mapping function to assign each processor its share of array elements. This mapping function, however, cannot be determined until we know *all* values

```
      PROGRAM mesh

      INTEGER i, n1, n2, n$proc, Nnodes, Nedges, stepMax, t
      PARAMETER (Nnodes = 10000, Nedges = 20000)
5     INTEGER ends1(Nedges), ends2(Nedges)
      REAL x(Nnodes), f(Nnodes), w(Nnodes), flux

C     Fortran D directives
      DECOMPOSITION nodeD(Nnodes), edgeD(Nedges)
10    ALIGN f, w, x, WITH nodeD
      ALIGN ends1, ends2 WITH edgeD
      DISTRIBUTE nodeD(BLOCK), edgeD(BLOCK)

C     Initialize data
15    CALL read_data(x, w, ends1, ends2)

C     Redistribute nodeD according to coordinate values
      DISTRIBUTE nodeD(VALUE(DIM=1, VALS=x))

20 C  Redistribute edgeD according to nodeD
      ALIGN egdeD WITH nodeD(VALUE(DIM=2, VALS=ends1,ends2))

      Loop over timesteps
      DO t = 1, stepMax
25
C        Reset fluxes to zero
         DO i = 1, Nnodes
            f(i) = 0
         ENDDO
30
C        Computes fluxes
         EXECUTE (i) ON_HOME ends1(i)
         DO i = 1, Nedges
            n1 = ends1(i)
35          n2 = ends2(i)
            flux = flux_func(w(n1),w(n2))
            f(n1) = f(n1) + flux
            f(n2) = f(n2) + flux
         ENDDO
40    ENDDO
      END
```

Figure 6: Fortran D kernel of a sweep over the edges of an unstructured mesh. The mesh coordinates are stored in the coordinate array `x` (again 1-D for simplicity), the topology is stored in the endpoint arrays `ends1`, `ends2`. After an initial index-based distribution of node and edge data in line 12, the node data are redistributed by value in line 17, and edge data are aligned by value with the redistributed node data in line 21.

of x. To resolve this problem, we start out with a different, typically regular mapping, which can be used for example to read in the data as is the case in Figure 3. After the data relevant for the irregular mapping are known, the decomposition is remapped based on their values.

4 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 mappings in a data-parallel language such as Fortran D. The program shown in Figures 7 and 8, which was generated by the Fortran D compiler from the NBF kernel in Figure 3, will serve as an example.

4.1 The Input Language

Since we implemented value-based mappings as part of a Fortran D compiler prototype, the Fortran D language constructs also serve as a basis for the syntax of value-based mappings. 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` and `ALIGN` directives, as was also illustrated by the code in Figure 3. Here the directive `DISTRIBUTE atomD(VALUE(DIM=1, VALS=x, WEIGHT=inb))` was the only statement the programmer had to add in order to express the value-based locality of the application; the rest was done by the compiler. The value-based mapping syntax currently supported is shown in Figure 9.

Note that the range of available mapping strategy depends more on the available run-time support than on the compiler. In fact, the strategy specified by the user might be passed through verbatim to the run-time library. However, one might still require a certain minimal set of strategies to be always available; see [PSC93a] for a comparison.

Note also that the user may not select a specific strategy for distributing data, as shown in Figure 3. 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 be written as `DISTRIBUTE atomD(VALUE(x, inb))`.

Naturally, there are several possible modifications/extensions for this syntax, which was held deliberately simple. For example, the value-based alignments could also be expressed as just another form of redistributions instead. Or, one could allow multidimensional value arrays instead of several one-dimensional arrays; for example, a program may store three-dimensional physical coordinates in one two-dimensional array, `x(3, n)`, instead of using three one-dimensional arrays, `x(n)`, `y(n)`, `z(n)`.

4.2 When to Distribute and Align

In general, mapping directives can be viewed as either static declarations (such as the HPF `DISTRIBUTE`), or as executable statements (like the HPF `REDISTRIBUTE`). There are several reasons why value-based mappings should be viewed as executable, for example because of the bootstrapping problem described in Section 3.5, or because the values relevant for the mapping might change for dynamic problems and we might want to remap periodically.

A resulting question is *when* this mapping should be performed, whether it should be done when the execution reaches the directive, or instead at some other point in the program determined by the compiler. In the latter case, one might for example envision a scheme that lets the compiler analyse

```
      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$cnt from inb, x, atomD$cnt
      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 7: NBF kernel, output of Fortran D compiler (continued in Figure 8). The code is shown before converting references to dynamically allocated arrays (which are not explicitly declared) 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$cnt
         DO p = 1, inb(i)
50          j$cnt = j$cnt + 1
            j = partners(i, p)
            j$glob(j$cnt) = j
         ENDDO
      ENDDO
55    Compute x$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 f(atomD$cnt + x$offsize)
      Resize x(atomD$cnt + x$offsize)
60
C     Loop over timesteps
      DO t = 1, stepMax

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

         Gather x using x$sched
70 C     Compute forces
         i$ = 0
         DO i = 1, atomD$cnt
            DO p = 1, inb(i)
               i$ = i$ + 1
75             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
80       ENDDO
         Scatter_add f using x$sched

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

Figure 8: NBF kernel generated by the Fortran D compiler, continued from Figure 7.

DistDirective	is	`DISTRIBUTE` *ValMapping*
AlignDirective	is	`ALIGN` *decomposition-name* `WITH` *ValMapping*
ValMapping	is	*decomposition-name* (`VALUE` (*ValArrays* [,*Weight*] [,*Strategy*]))
ValArrays	is	[`DIM =` *num-dims*,] [`VALS =`] *value-array-name* [, *value-array-name*] ...
Weight	is	[`WEIGHT =`] *weight-array-name*
Strategy	is	[`STRATEGY =`] *mapping-strategy-name*
Constraint:		Number of value-array-names must equal one, or, if specified, *num-dims*.

Figure 9: Syntax of the value-based mapping directive. Square brackets indicate optional components.

where relevant values are defined (or redefined), and where mapped data are used. However, we feel that this kind of analysis would be fairly difficult and problem dependent (for example, we might not really want to redistribute whenever one relevant value changes), and that the gain in programming convenience would be only marginal. Therefore, our current implementation performs the value-based mappings at the location corresponding to the mapping directive.

4.3 Mapping the Data

Each value-based mapping 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 partitioner (line 27 in the example). This partitioner applies the strategy specified by the programmer, or a default specified 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; in the presence of weights, the overall weight assigned to each processor should be similar. Before partitioning, `atomD$cnt` is initialized (line 26) according to the initial, regular distribution.

   (b) Based on $D.loc2proc$ and $D.cnt$, compute a translation table (see Section 3.1), $D.tab$ (line 28). $D.tab$ is actually a pointer to a data structure internal to the run-time library; within Fortran it is declared as an integer. This mechanism is also used for other internal structures.

   (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 7 and 8 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 live, 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.

4.4 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 3 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 interprocedurally [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 3) 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].

4.5 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 4.4). 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

We compiled a molecular dynamics kernel and an unstructured mesh example using various mapping strategies. In evaluating our proposed language extensions, we are mostly interested in the following questions:

1. How much does the extension improve the generated code?
2. How close is the generated code to a hand-coded implementation of the same approach?
3. How much convenience does the new extension provide over hand-coding?

To answer the first question, 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, and various performance aspects were measured as described in the following sections.

Regarding the second question, the Fortran D compiler-generated output was practically identical to a hand-coded parallel program using the same CHAOS run-time support. While this is certainly at least partly due to the simplicity and small size of the compiled kernels, it still gives reason to believe that there are no fundamental disadvantages when using the compiler to implement value-based mappings.

To answer the third question, one has to compare the complexities of original and compiled codes, with and without value-based distributions. While size alone is not a sufficient measure of complexity and programming difficulty, it still is an indication for the savings provided by the compiler; Figures 3, 7, and 8 provide a more detailed comparison. It turned out that for these kernels, the generated code (which also in size was very close to hand-coded) was about twice as large as the initial code for index-based distributions. For value-based distributions, the code grew by another 25%.

5.1 The Molecular Dynamics Kernel

Our experiments were based on the Fortran D kernel from Figure 3 (but with 3-D instead of just 1-D coordinates and forces) with varying index- and value-based distribution directives as listed below. The parallel runs used pairlist data and physical coordinates from an SOD simulation using an 8Å cutoff radius; SOD itself is of size $53 \times 55 \times 52$Å^3. We ran the simulation for 30 time steps, on 1, 2, 4, 8, 16, and 32 processors, with the following distribution strategies.

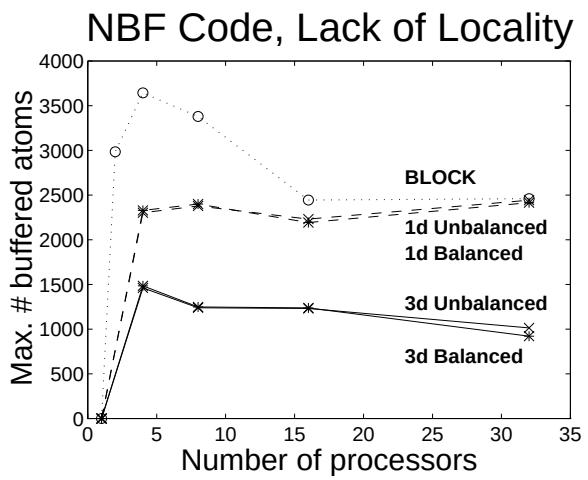

Figure 10: The number of communicated data (maximum across processors) for varying machine sizes and distribution strategies in the non-bonded force kernel.

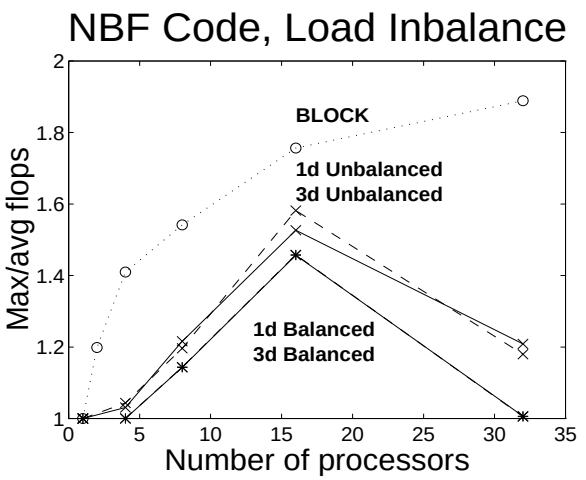

Figure 11: The fraction of maximum floating point operations (*i.e.*, workload of the slowest processor) and of the average across processors.

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), and is illustrated in Figure 2.

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 3, the resulting mapping can be seen in Figure 4.

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; see Figure 5.

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

5.1.1 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

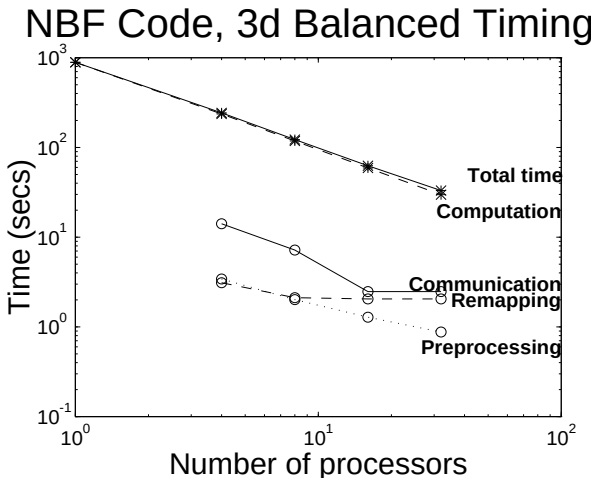

Figure 12: The timing breakdown (excluding I/O) for varying machine sizes, using a 3-dimensional, value-based balanced bisection.

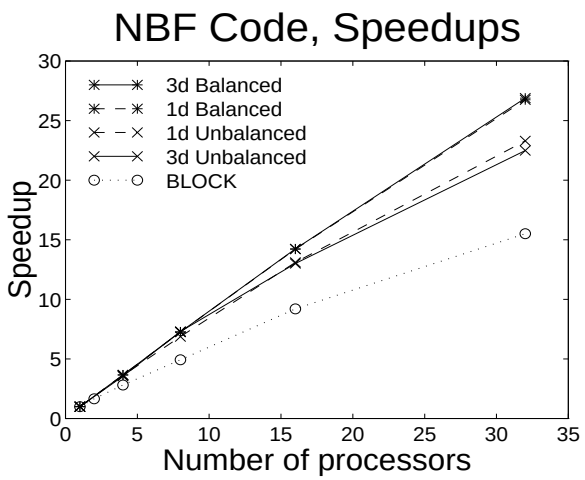

Figure 13: The speedup obtained with different processor numbers and distribution strategies.

number based on the total number of atoms (6968), the space they occupy (a sphere with a little over 50Å in diameter), and R_{cut} (8Å), which yields that around 950 atoms will have to be buffered.

Figure 10 compares the maximum number 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.

5.1.2 Balancedness

Figure 11 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 unsuitable due to the diagonal nature of the adjacency matrix (based on Newton's Third Law), which shifts the work load towards atoms with smaller indices.

5.1.3 Individual timings

Figure 12 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 (30). However, it is also apparent that the scalability of the run-time support may become critical for larger numbers of processors.

5.1.4 Speedups

Figure 13 summarizes the speedups obtained for the various distribution methods. The value-based distributions outperform the index-based distribution, and explicit load balancing provides

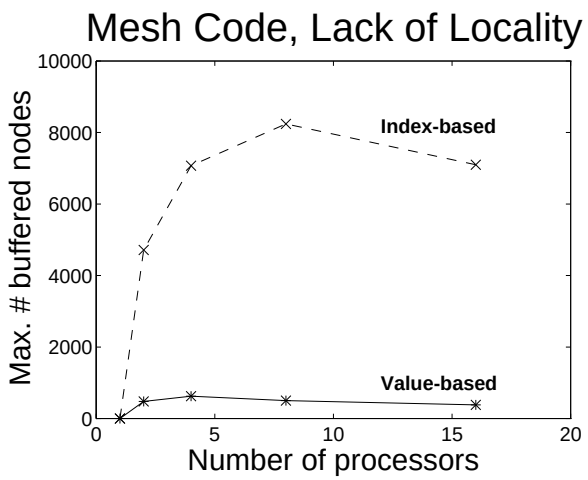

Figure 14: The number of communicated data (maximum across processors), for varying machine sizes and alignment strategies of the mesh kernel.

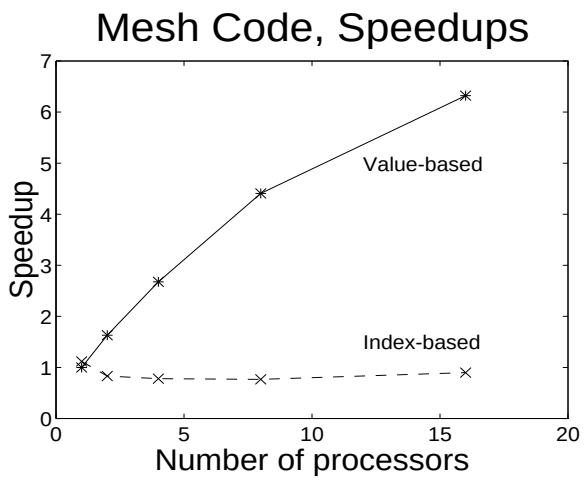

Figure 15: The speedup obtained with different processor numbers and mapping strategies.

an additional advantage.

5.2 The Unstructured Mesh Kernel

We generated code for the Fortran D kernel from Figure 6, but again with 3-D instead of just 1-D coordinates and fluxes. Here we measured only the effect of varying alignments of `edgeD`; the distribution of `nodeD` was a fixed, 3-D value-based distribution. The parallel runs were done for a mesh with 9428 nodes and 59863 edges. We ran the simulation for 30 time steps, on 1, 2, 4, 8, and 16 processors, with the following alignment strategies.

1. Index-based, block-wise: for N edges and P processors, processor p gets assigned edges $(p-1)N/P+1 \ldots pN/P$ (assuming P divides N). This corresponds to using the original `BLOCK` distribution (line 12) throughout the run (*i.e.*, no redistribution in line 21).

2. Value-based: decomposition `nodeD` and edge info `ends1`, `ends2` serve to align edge data with the node data. This corresponds to the value-based alignment directive shown in Figure 6.

5.2.1 Locality

Similar to the number of buffered atoms in the NBF kernel, here the number of not-owned nodes accessed by the individual processors is a measure for the inter-processor access locality of the mesh kernel. Figure 14 compares the maximum numbers of buffered atoms for different machine sizes and alignment strategies.

Considering that there are less than 10000 atoms total, we see that the index-based mapping requires buffering a significant fraction of the whole data domain. Together with the owned edges we are effectively replicating the edge data for $P \leq 8$. The value-based alignment has a significantly better locality, typically requiring less than 500 nodes to be buffered.

5.2.2 Speedups

Figure 15 summarizes the speedups obtained for the two mapping methods. While the index-based mapping does not result in any speedup due to the bad locality, the value-based distribution provides about 40% efficiency for 16 processors.

6 Related Work

Poor locality and load imbalances when parallelizing irregular applications are problems not limited to data-parallel implementations; application programmers have already been taking advantage of value-based locality when hand-coding in low-level parallel languages (for example with message passing). They have developed a wealth of different mapping strategies, both value-based [BB87, HS91, DHU+93], and connectivity based [Sim91], which can be used as mapping strategies for irregular distributions.

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

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, also for the compiler writer; our Fortran 77D implementation also generates code that calls the CHAOS library. 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.

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.

To specify irregular mappings in a data-parallel context, index-based mapping arrays [WSBH91] or mapping functions [CMZ92] have been proposed. For these, however, the programmer has to provide such arrays or functions that explicitly map indices to processors, even though the programmer may not be interested in what exactly these mappings look like.

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]. However, the GeoCoL structure still has to be managed explicitly by the programmer.

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 an index and iteration space oriented data-parallel paradigm.

We extended this paradigm by adding *value-based* enhancements to the already existing, so far index-based data distribution and alignment mechanisms. We illustrated their use with two typical irregular kernels and compared the performance of these kernels with index-based and value-based distributions. We also compared a Fortran D version and a message passing version, *i.e.*,

a program using run-time support explicitly as generated by the compiler (or written by hand), of the same kernel. This gave an example for the added convenience provided by the proposed language extensions.

A principal difference between index-based and value-based mapping strategies is the departure from a model that derives data-to-processor mappings from static, sequential data-to-storage information (like an array index). Instead, we consider run-time values to derive the mappings. This approach could also be used for data types other than arrays, for example for list structures in a version of data-parallel C, or for distributing spatial data structures such as oct-trees commonly used for N-body problems with very large N.

Another application of the value-based approach could be *value-based inspectors*. For example, the pair list used in the NBF kernel is typically generated with a naive $\mathcal{O}(N^2)$ algorithm, where each atom is compared against every other atom, with correspondingly high computation, communication, and storage requirements. If we know that each processor has atoms only within a relatively small subdomain, we can restrict our attention to atoms that are either within or close to this subdomain, thereby reducing the amount of non-local data that have to be buffered.

Value-based mappings provide a specific kind of expressiveness that had been missing so far; they can give the compiler information that cannot be described in terms of array indices, and they provide a convenient mechanism to specify irregular mappings. While this is only one aspect of parallelizing irregular problems, we believe that it significantly widens the range of application that can be implemented efficiently in data-parallel languages.

Acknowledgements

Seema Hiranandani and Chau-Wen Tseng, who developed the Fortran D compiler support for regular problems, have been very helpful in extending the compiler for irregular applications. Paul Havlak provided valuable symbolic analysis to the Fortran D compiler; Chuck Koelbel has been a good discussion partner for language design.

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. Ravi also provided us with the mesh kernel and test data. Jim Humphries wrote the dynamic memory allocator.

Special thanks go 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.

The molecule graphics were produced by the RasMol visualization program written by Roger Sayle.

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

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.

[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`.

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

[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, Rice University, 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*, pages 353–364, 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.

[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.

[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.