A Data Structure for Managing Parallel Operations

Carl D. Offner
High Performance Computing Group
Digital Equipment Corporation
129 Parker Street Maynard, MA 01754

Abstract

The data distribution directives of High Performance Fortran (HPF) provide a high-level way of describing the location of regularly distributed data which is acted on in parallel. Parallel data is acted on by parallel computations. The distribution of parallel computations is not specified in HPF, but is left up to the compiler—typically, the owner-computes rule is used, not always the best choice.

We present a set of data structures which describe data layout, consistent both with HPF directives and with more general data optimization techniques; we also present an analogous set of data structures which describe the distribution of parallel computations. These two sets of data structures are related by a set of constraints which apply when data is being accessed locally. This provides a coherent way to represent parallel data and parallel computations on an equal basis and thereby enable a compiler to optimize both kinds of parallelism.

1 Data layout on a distributed-memory machine

We address the problem of generating efficient code for a data-parallel application running on a distributed-memory SIMD or MIMD machine. That is, our machine consists of a possibly large number of processors, each with its own linear memory. The data in the program is significantly parallel—for our purposes, it contains large arrays; and this data is distributed and mapped onto the separate processors in some fashion. Each processor executes the same program; the actual instructions executed on different processors may differ, however, because each processor knows its processor number, and the code is parametrized by that number. On MIMD machines, this programming model is referred to as SPMD ("single program, multiple data").

Since inter-processor communication is typically very costly on such machines, it is important to lay out the data and schedule the computations so that communication is minimized while maintaining as much parallelism as possible.

Using *data optimization* techniques ([1, 3, 4, 5]), the compiler can look at patterns of usage of arrays in the program to perform layout automatically. An alternative approach is to have the programmer insert directives specifying how arrays are laid out. This is the approach taken by High Performance Fortran (HPF) [2]. Since the HPF approach forces the programmer to go through the same analysis that the compiler would otherwise have to do, we will start by giving a brief introduction to some ideas of data optimization. The data structures that we will subsequently present apply equally, regardless of whether automatic data optimization or HPF directives are used.

2 Some details of data optimization

Data optimization analysis, as described in the papers mentioned above, is performed by constructing a *preference graph* whose nodes are expression nodes in a program and whose edges are called *preferences*. An *unhonored* preference generally implies data motion. Preferences can be honored unless there are *conflicts*.

There are several kinds of preferences; we mention only two here:

identity preferences These are preferences that reflect true dependences in the program. That is, if a data object is defined and subsequently used, the processor where it is defined should be the same processor where it is used; if not, then we will have to move the object.

conformance preferences These have been defined as preferences between corresponding elements of data objects that occur in a parallel construct.

These preferences are honored if such corresponding elements live on the same processor. We will refine this definition below in Section 4.4.

Preferences can be honored unless there are cycles in the preference graph, although a cycle does not necessarily imply an unhonored preference.

$$B(1\!:\!100,\ 1\!:\!100) = \ldots$$

$$A(1\!:\!100,\ 1\!:\!100) = B(2\!:\!101,\ 3\!:\!102)$$

$$C(1\!:\!100,\ 1\!:\!100) = A(4\!:\!103,\ 5\!:\!104) + B(5\!:\!104,\ 9\!:\!108)$$

Figure 1: Part of a preference graph with a cycle

In Figure 1, identity preferences are indicated by dashed arcs and conformance preferences by solid arcs.[1] Even though there is a cycle in the preference graph, there are no conflicts in the first dimension. If we drew the preference graph for the second dimension, however (this just amounts to moving the graph over to the right at each array reference), we would see that there is in fact a conflict. So preferences must be computed separately for each dimension.

Note that a cycle in the preference graph must contain both identity and conformance preferences.

While identity preferences directly refer to where data lives, conformance preferences actually refer to where computations happen. For instance, in the example (using the HPF **forall** construct)

$$\textbf{forall}\ (i = 1\!:\!n,\ j = 1\!:\!m)$$
$$A(i,\ j) = A(i,\ j){+}B(j,\ i,\ 3)$$

there is a conformance preference which indicates that we want $A(i,j)$ and $B(j,i,3)$ to be added together on the same processor. This is a preference which really is determined by the usage of the **forall** indexes i and j. For this reason, we will modify our notion of conformance preference below so that it refers to computations, rather than to data elements.

The HPF directives allow the programmer to describe data layout, and thereby manage identity preferences, but give no way of dealing with conformance preferences. Instead, the "owner-computes" rule has been widely promoted as a way of determining where computations take place. It is easy, however, to write examples where owner-computes is undesirable.

In contrast, the data structures presented below handle parallel data and parallel computations on an equal basis, emphasizing and explicitly supporting a point of view that was implicit in the original papers on data optimization.

Section 3 outlines the data structures which describe identity preferences. These data structures are built around a *data space*. Section 4 outlines the data structures which describe conformance preferences. These data structures are built around an *iteration space*. In Section 5 we will show how these two data structures are related. This relation occurs at data references in expression trees. Section 6 extends these data structures to handle partially replicated data and computations, Section 7 gives some simple examples of how these data structures can be used by a compiler to manage communication effectively, and Section 8 concludes with some comments on the role of the compiler.

Sections 4, 5, 6, 7, and 8, together with Figures 3 and 4 contain the original work of this paper. They give a coherent treatment of data and iteration spaces, leading to a unified view of parallel data and parallel computations. As part of this unified view, a refined definition of a conformance preferance is given, and the data structures are extended to handle partially replicated data and computations in a natural fashion.

3 Representing parallel data

In this section, we describe a family of data structures which can be used to manage the HPF directives, as well as to manage the more general preference graphs used in data optimization. The data structures are composed of certain spaces and maps between those spaces.

3.1 Data space

Data space describes data which could in principle be acted on in parallel.

With each declared object A is associated a data space. This data space is simply $\mathbf{Z}^n$, where n is the declared dimensionality of A. The elements of the data space are n-tuples which should be thought of as the subscripts of A. If A is a scalar, then of course $n = 0$ and the data space is $\mathbf{Z}^0$, a 1-point space.

Note that the declared bounds of A have no bearing on the data space used to describe A.

[1] There are also other preference edges which are not shown in the figure.

Elements of the data space will be denoted by vectors $\langle s_j \rangle$, where s_j denotes the j^{th} subscript. Each s_j is called a *data coordinate*.

3.2 The space of virtual processors

Early in the compilation process, it is convenient to pretend that we have an infinite number of processors available to us. Later in the compilation process, these virtual processors must be mapped onto physical processors, in general in a many-to-one fashion. But before this mapping takes place, we have a virtual processor space associated with every node in an expression tree. This virtual processor space will in general be many-dimensional, so it will be of the form $\mathbf{Z}^n$.

A particular virtual processor will always be denoted by a vector $\langle v_j \rangle$, where of course j runs over the range 1 to n.

In HPF, a space of virtual processors can be declared by the **template** directive. For instance,

!HPF$ **template** $t(100,\ 100)$

declares a virtual processor space of dimension 2.

3.3 Data allocation functions

Associated with each data object is both a data space and a virtual processor space.

There is a map from the data space associated with a data object to the corresponding virtual processor space which describes how that data object is laid out over the virtual processors.

Although such a map could in principle be arbitrarily complex, we restrict ourselves to maps which in each coordinate are affine in one variable. Precisely, each such map is of the form

$$f(s) = \langle f_j(s_{p(j)}) \rangle = \langle \mathit{alloc\_stride}_j * s_{p(j)} + \mathit{alloc\_offset}_j \rangle$$

where p is a 1-1 map of the subscript indexes into (and not necessarily onto) the virtual processor space indexes.

In particular, distinct data elements are mapped to distinct virtual processors. Of course, elements of different data objects can be mapped to the same virtual processor; see for instance the example below.

Each map f_j is called a *data allocation function*.

In HPF, the ALIGN directive can be used to specify the data allocation functions of a data object. For example, in the code

!HPF$ **template** $t(100,\ 100)$
!HPF$ **align** $A(i,\ j)$ **with** $t(i,\ j)$
!HPF$ **align** $B(i,\ j)$ **with** $t(j,\ i)$
!HPF$ **align** $C(i,\ j)$ **with** $t(i{+}1,\ 2{*}j{-}3)$

the data allocation functions for A, B, and C are

$$\langle i,\ j \rangle \to \langle i,\ j \rangle$$
$$\langle i,\ j \rangle \to \langle j,\ i \rangle$$
$$\langle i,\ j \rangle \to \langle i{+}1,\ 2{*}j{-}3 \rangle$$

respectively.

There are a few issues which come up in precisely specifying the data allocation maps:

- Which virtual processor coordinate is a particular data coordinate mapped to? (The function p above represents this map.)
- How can we represent the fact that the map f from data space to virtual processor space may not be onto?
- How can we represent the fact that the map f from data space to virtual processor space may actually be 1-to-many?

For instance, in this example:

real $A(100,\ 100)$
!HPF$ **template** $t(50,\ 100,\ 300,\ 25)$
!HPF$ **align** $A(i,\ j)$ **with** $t(7,\ j,\ 2{*}i{+}3,\ *)$

the map from data space to virtual processor space is not onto because no template element whose first coordinate is different from 7 is in the image of that map; and it is 1-to-many because, for instance, the data element $(1,1)$ is mapped to the virtual processor elements $(7,1,5,k)$ for all k.

As we will show below, we can manage all these considerations by representing the coordinates of the virtual processor space by data structures called *cells*.

3.4 The space of logical processors

Since there are really only a finite number of processors, the space of virtual processors will be distributed onto a space of logical processors. The space of logical processors is an n-dimensional block contained in $\mathbf{Z}^n$, of the form

$$\prod_{j=1}^{n} [0 : \mathit{strip\_length}_j - 1]$$

(This formula constitutes the definition of *strip_length*.) The elements of the space of logical processors are denoted by vectors $\langle \bar{v}_j \rangle$, where j runs over the range 1 to n.

A space of logical processors can be declared in HPF by the **processors** directive:

!HPF$ **processors** $p(32,\ 16)$

declares p to be a 32×16 array of logical processors.

3.5 The physical processors

In general, the physical processors do not constitute an n-dimensional array. But in any case, the set of physical processors corresponds in a 1-1 fashion to the space of logical processors. The precise way in which this correspondence is made is machine-dependent. (There is currently no way in HPF to specify this mapping.) For instance, the physical processors might constitute a 1-dimensional array. The space of logical processors is mapped onto this array in in column-major order.

Bear in mind that we are only talking about processor addressing here, not about the actual connectivity of the processors, or about relative costs of communication. To say that the physical processors constitute a 1-dimensional array just means that the operating system numbers them from 0 to $n-1$, not that processor 0 is closer to processor 1 than to processor 2, say. The processors might in fact be fully connected.

In other words, when we refer to physical processors, we really mean physical processors as presented by the operating system. The operating system itself may have still another view of the underlying machine.

3.6 Distribution maps

The virtual processors are *distributed* over the logical processors. HPF constrains this distribution map to be a coordinatewise map, each coordinate being mapped in one of only a few different ways:

block A block of consecutive virtual processors is mapped to the first logical processor; the next block of virtual processors is mapped to the next logical processor, and so on.

cyclic The virtual processors are dealt out to the logical processors in a cyclic manner, much as one might deal a deck of cards.

***** Serial allocation; corresponds to **block** or **cyclic** with a *strip_length* of 1.

and there are some variants of these methods as well. Thus, the map can be described in HPF by a **distribute** directive, which describes the distribution map used for each data dimension. For example:

!HPF$ **template** $t(1000, 1000, 1000)$
!HPF$ **processors** $p(32, 16)$
!HPF$ **distribute** t(**block**, **cyclic**, $*$) **onto** p

3.7 Cells

Now we show how to represent the virtual processor space in a more useful form.

The virtual processor space associated with a data space will be represented as a list of *data cells*. A cell is a data structure that holds all the information we need about one dimension of the virtual processor space. The mapping from cells to dimensions, and thence to logical processors, can be defined separately.

First consider a machine where the number of processors is a power of 2, say 2^n, and where the physical processors are arranged in a 1-dimensional array. Each processor then has a processor ID number which can be represented by an n-bit binary word. The different dimensions of the logical processor space can be made to correspond to different bit ranges in this processor ID word. For instance, on a machine with 2^{16} processors, a 3-dimensional logical processor array might be allocated as in Figure 2, where the first dimension of the logical processor array is expressed by bits 0-4 of the processor ID word, and so on.

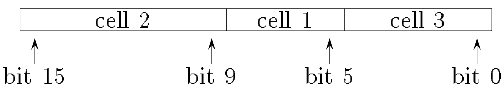


Figure 2: Processor word with three labeled cells

In this figure there are three cells, which we may number arbitrarily. If there is a 3-dimensional array A which is distributed over this logical processor space, we will number the cells so that each cell's number is the same as the dimension of A to which it corresponds.

With this cell layout, and assuming that the data allocation functions are all identity functions and that the array A is small enough so that distinct elements of the array can be assigned to different processors (so the value of each subscript is the value in the corresponding cell), the array element $A(i, j, k)$ will be found in processor $i * 2^5 + j * 2^9 + k * 2^0$.

Thus the correspondence of cells to dimensions of the virtual processor space is made by assigning to each cell a number representing the position of its lowest bit.

The mapping to logical processors, which involves folding the space of virtual processors onto the logical processor space, involves the size of the cell, i.e. the number of bits in it. 2 raised to that number of bits is what we have denoted above as the *strip_length* of the logical processor dimension. We also refer to it as the *strip_length* of the cell.

A similar technique works for a machine model with any number of processors (i.e. not just powers of 2). Each cell will have a *strip_length*; the product of these strip lengths will be the number of physical processors in the machine. Each cell will also have a *multiplier* (analogous to the factors 2^5, 2^9, and 2^0 above); these multipliers will determine how the cells fit together to form the virtual processor array dimensions. A cell is determined by its *strip_length* and its *multiplier*, which together we call the *cell place.*

In view of this representation, we usually refer to each virtual processor coordinate v_j as the *virtual contents* of a cell.

Thus, the set of data allocation functions maps data space coordinate-wise to virtual processor space, as on the right-hand side of Figure 3.

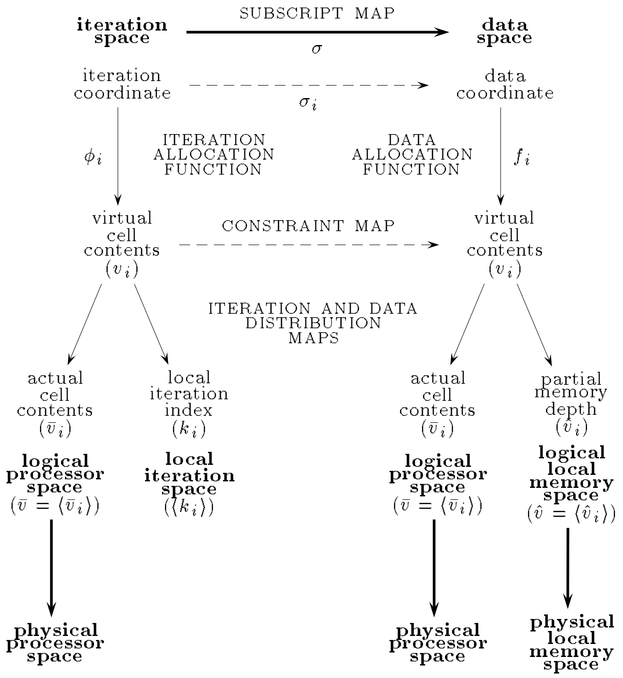
The set of distribution functions then maps the virtual processor space coordinatewise to the logical processor space. In doing so, it folds each virtual processor coordinate onto the coordinate of the logical processor space, so that for each i, more than one value of the virtual cell contents v_i may be mapped to a logical processor coordinate value $\bar{v}_i$. This is resolved by also associating with each v_i a partial memory depth (which we denote by $\hat{v}_i$). (The separate partial memory depths $\hat{v}_i$ have to be combined to form the actual memory depth in the local processor's memory.) The mapping from v_i to $\hat{v}_i$ is determined by the distribution map, just as the mapping from v_i to $\bar{v}_i$ is. These maps also appear on the right-hand side of Figure 3.

3.8 Identity preferences

Identity preferences can be expressed in terms of this data structure as follows: an identity preference between two occurrences of an array A is honored iff for each dimension of A

- the corresponding data allocation functions are equal,
- the corresponding data distribution functions are equal, and
- the corresponding data cells have the same place.

The first condition guarantees that any given element of A will be mapped to the same virtual processor at both occurrences, and the remaining conditions ensure that a virtual processor will be mapped to the same physical processor at both occurrences.



Spaces have names in boldface type. Maps between spaces are represented by thick arrows. One-dimensional subspaces (i.e. the coordinate subspaces of those spaces) are either above or below their corresponding spaces and have names in normal weight type. Maps between those one-dimensional subspaces are represented by thin arrows. Symbols in parentheses parametrize their corresponding spaces or subspaces. The maps represented by dashed arrows apply when the data access is local.

Figure 3: How data and iteration structures are related at a data access

4 Representing parallel computations

4.1 Iteration space

Iteration space describes computations which could in principle be performed in parallel.

A section of code[2] which is implicitly or explicitly parametrized by one or more variables is a candidate for being performed in parallel. The number of parameters (call it m) is said to be the dimensionality of the iteration space, and the iteration space itself is simply $\mathbf{Z}^m$. Elements of the iteration space are m-tuples

[2]The word "section" is used informally here; it is illustrated below, and it does not have the formal meaning given in some language definitions.

which should be thought of as particular choices of the m parameters.

Iteration spaces can arise from **forall** statements, array syntax, or **do** loops: the three parallel constructs

(1) **forall** $(i = 1{:}n,\ j = 1{:}m)$
$B(i,\ j) = A(i,\ j)$

(2) $B(:,\ :) = A(:,\ :)$

(3) **do** $i = 1,\ n$
do $j = 1,\ m$
$B(i,\ j) = A(i,\ j)$
enddo
enddo

are all parametrized by two variables (the second one implicitly, the first and third by i and j), so the iteration space in each case is $\mathbf{Z}^2$.

As with data space, we are taking the iteration space to be all of $\mathbf{Z}^m$, even though the parameters can only take values from a bounded set.

The bounds, however, are important; and while they may be determined in many ways, they must be the same for every expression node in the section of parallel code to which the iteration space applies. To put it another way, there is only one iteration space for each parallel section of code. In particular, there is only one iteration space for a parallel statement. So for instance, in the statement

$$A(1{:}100) = B(400{:}202{:}{-2}) + C(2{:}101)$$

the actual iteration range could be taken as $1{:}100$, or as $400{:}202{:} - 2$, or as $2{:}101$, or even as some other equivalent range. But whatever it is, it will be the same for every expression node in that statement.

Elements of the iteration space will be denoted by vectors $\langle j_i \rangle$ (each j_i denoting an iteration index). Each j_i is called an *iteration coordinate*.

It is important to keep in mind that data space and iteration space are distinct concepts. For instance, in the statement

$$C(:,\ :) = A(:,\ :) + B(:,\ 11,\ :)$$

the data space at B is $\mathbf{Z}^3$, but the iteration space for this statement is $\mathbf{Z}^2$.

4.2 Iteration allocation functions

Iteration spaces are mapped to the virtual processor space—this amounts to saying which virtual processor executes a particular instance of a parallel computation. This mapping, which is coordinatewise and affine, is composed of *iteration allocation functions*, which are entirely analogous to data allocation functions. There is no way in HPF to specify these functions; it is up to the compiler to come up with the best ones it can.

As in the case of parallel data, the virtual processor space is represented in terms of cells—we call these the *iteration cells*. We denote the individual iteration allocation functions by ϕ_i (where i denotes the i^{th} iteration coordinate; that is, the i^{th} iteration cell).

4.3 The distribution of parallel computations

After parallel computations are mapped to virtual processors, they are distributed onto the space of logical processors, just as parallel data is. The distribution maps are analogous to data distribution maps—**block**, **cyclic**, and so on. Again there is no way in HPF to specify these maps.

Here the key difference between data and computations appears: While data which is mapped to the same logical processor is located at different memory locations in that processor's memory, computations which are mapped to the same logical processor take place at different times. Thus, the iteration distribution maps define mappings both from the space of virtual processors to the space of logical processors, and from the space of virtual processors to a local iteration space, which parametrizes the computations mapped to a single logical processor. This gives us the maps shown, later on, on the left-hand side of Figure 3.

4.4 Conformance preferences

We will now recast the data optimization notion of conformance preferences as follows: two computational nodes in the compiler's internal representation of a parallel construct[3] will have a conformance preference between them which is honored iff

- the corresponding iteration allocation functions are equal,
- the corresponding iteration distribution functions are equal, and
- the corresponding iteration cells have the same place.

The first condition guarantees that on any given iteration of the parallel construct the two nodes will

[3]By a computational node we mean any node which is not a constant or a data reference; i.e. any node having an associated iteration space

be computed on the same virtual processor, and the remaining conditions ensure that a virtual processor will be mapped to the same physical processor at both nodes.

Let us look again at the example from Figure 1. Figure 4 shows the same code as it might be represented internally by the compiler (except that the arrays are now 1-dimensional; we have taken the second dimension, which is where the problem lies), and we have indicated the identity preferences as dashed arcs and the conformance preferences as solid arcs.

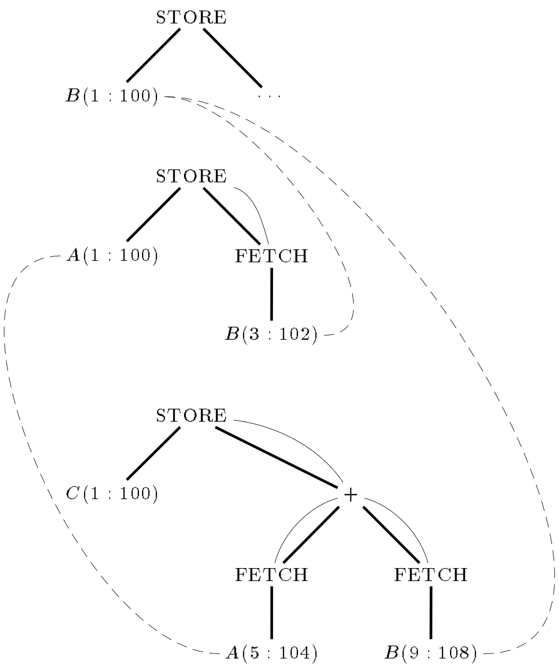


Figure 4: Identity and conformance preferences for the second dimension of figure 1

As yet, there are no cycles in this preference graph—in fact, the graph is disconnected at every data reference. We address this issue in the next section.

5 The relation between data and iteration structures

5.1 The subscript map

Every computational node in an expression tree has an iteration space associated with it. Each data reference (typically, an array reference) in an expression tree has a data space associated with it. So at each fetch or store of data there is an iteration space, and at its child (which is a data reference) there is a data space. These spaces are related by a *subscript map* which maps the iteration space into the data space. Here are some examples:

(1) $B(:,\ :) = A(:,\ 4,\ :)$

The iteration space and the data space at the store of B are both $\mathbf{Z}^2$, and the subscript map is the identity map $\mathbf{Z}^2 \to \mathbf{Z}^2$. At the fetch of A, however, the data space is $\mathbf{Z}^3$, and the subscript map is the map from $\mathbf{Z}^2 \to \mathbf{Z}^3$ given by

$$\langle i,\ j\rangle \to \langle i,\ 4,\ j\rangle$$

(2) **forall** $(i = 1{:}n,\ j = 1{:}m)$
$B(i,\ j{-}i) = A(2{*}i{+}j,\ -1,\ 4{-}3{*}j)$

Here the subscript map at the store of B is the map from $\mathbf{Z}^2 \to \mathbf{Z}^2$ given by

$$\langle i,\ j\rangle \to \langle i,\ j{-}i\rangle$$

while the subscript map at the fetch of A is the map from $\mathbf{Z}^2 \to \mathbf{Z}^3$ given by

$$\langle i,\ j\rangle \to \langle 2{*}i{+}j,\ -1,\ 4{-}3{*}j\rangle$$

(3) $A(1{:}100) = B(400{:}202{:}{-}2) + C(2{:}101)$

If the iteration range is taken as $1{:}100$, then the subscript maps at the store of A, the fetch of B, and the fetch of C are

$i \to i$
$i \to 402{-}2{*}i$
$i \to i{+}1$

respectively. On the other hand, if the iteration range is taken as $400{:}202{:}-2$, then the corresponding subscript maps would be

$i \to (402{-}i)/2$
$i \to i$
$i \to (404{-}i)/2$

and so on.

We will denote the subscript map by the letter σ. The subscript map appears at the top of Figure 3.

Figure 3 as a whole actually demonstrates the computations that occur on a processor when dealing with a reference to parallel data. After all, what does a processor know? It knows two things:

1. Who it is: its physical processor number, which it can map backwards to get its coordinates in the logical processor space.

2. What time it is: which of the local iterations is being processed. This amounts to the coordinates of a point in the local iteration space.

These two values are then pushed backwards through the iteration distribution maps, and then through the iteration allocation maps to get the point in iteration space being computed. This point is then mapped forward through the subscript map into data space and then down to arrive ultimately at a physical processor and a location in that physical processor's memory. These computations start on the lower left of Figure 3 and move up, over to the right, and then down to the lower right.[4]

Thus, starting with the two items that a physical processor knows, it can compute the location of the data it needs to access. In general, of course, this data may be on a different processor—the values in the physical processor space on the data side and on the iteration side of Figure 3 will be different. This is just an indication of the fact that in general, data accesses may be *remote*, as opposed to *local*.

5.2 Constraints

Looking at Figure 4, we see that since the identity and conformance preferences relate disjoint sets of nodes in the expression trees, the allocation functions and distribution maps can be defined in such a way that all the preferences are honored. The price we would pay for this, however, is that in general, too many data references would be remote—that is, inter-processor communication would be necessary at too many fetches and stores of data. So we have to impose some constraints between the data maps and the iteration maps. These constraints will relate the data maps at data references to the iteration maps at the parents of those references (i.e. at stores and fetches). The constraints will reflect the fact that certain data references are local.

To do this, we have to have a way of identifying data references which *a priori* should be local. One way is as follows: if the subscript map is a coordinatewise affine map—so σ can be expressed as $\langle\sigma_i\rangle$, where each σ_i is an affine function of one iteration coordinate—then we would certainly expect the data reference to be local. For example, an array reference of the form $A(i-3,\ 2*k+5,\ 7-j)$ should be local. It will be, if

1. the iteration space point $\langle i, j, k\rangle$ is mapped by the iteration allocation functions to the same virtual processor as the data space point $\langle i-3,\ 2*k+5,\ 7-j\rangle$ is mapped by the data allocation functions;

2. the corresponding distribution maps are identical;

3. the corresponding cells have the same cell place.

Items 2 and 3 are easy to check. Item 1 amounts to this: in Figure 3 the constraint map should be the identity (for all i).

The constraint for local data accesses thus consists of items 2 and 3 above together with the set of identities

$$\phi_i = f_i \circ \sigma_i$$

6 A catalog of different kinds of cells

By giving cells some additional attributes, we can extend this data structure to accomodate scalar, replicated, and irregular data.

A cell is a *data cell* or an *iteration cell* according to whether it is used on the data or iteration side of Figure 3.

The cells we have described so far are what we call *primary* cells: corresponding to each subscript of the form $a*i+b$ where i is an iteration coordinate, there is a primary iteration cell and a primary data cell. The data and iteration allocation functions for these two cells are related by the constraint

$$\phi_i = f_i \circ \sigma_i$$

Corresponding to a constant subscript in an array reference, we create a *scalar* data cell. To avoid data motion, such a scalar data cell at an array reference will be matched with a scalar iteration cell having the same cell place. It is convenient to imagine a constant iteration coordinate corresponding to the constant data coordinate, with a constraint that the iteration allocation function applied to this constant iteration coordinate value equals the data allocation function applied to the constant subscript.

Scalar cells are actually useful in more general circumstances, as when an array is aligned with a section of a template, e.g.:

!HPF$ **align** $A(i,\ j)$ **with** $t(7,\ j,\ 2*i+3)$

A reference to $A(i,j)$ will then involve a scalar data cell and a corresponding scalar iteration cell. The constant virtual cell contents will be 7.

[4] We hasten to assure the reader that in the most common cases, the compiler can generate much simpler code. But in general (for instance, when there are vector-valued subscripts), this is actually the way the computations must be generated.

Scalar cells are thus used in representing data or iterations which are not mapped completely onto the whole data or iteration space.

Replicated cells are used to represent replicated data or iterations. So for instance, given the HPF directive

!HPF$ **align** $A(i,\ j)$ **with** $t(7,\ j,\ 2{*}i{+}3,\ *)$

a reference to $A(i,j)$ will involve 1 scalar data cell, 2 primary data cells, and 1 replicated data cell. To avoid data motion, each replicated data cell will be paired with a corresponding replicated iteration cell with the same cell place. There are no subscripts or iteration coordinates corresponding to replicated cells.

A replicated cell can also be used to describe a scalar which is replicated over all the processors. A scalar which is replicated over just some of the processors would be described by two cells: a replicated cell and a scalar cell (with "complementary" or "orthogonal" cell places). In this way, a scalar data object can be regarded as a 0-dimensional array object.

Finally, a *complex* data cell is a cell corresponding to a subscript which is not constant or affine in one iteration coordinate. For example, the array reference $A(V(i),\ j{+}k)$ will have two complex data cells associated with it. A complex data cell has no associated iteration cell, because we use complex data cells as a catch-all mechanism for data that we cannot easily access in a regular fashion. The occurrence of a complex cell usually signals the need for data motion.

Another example where complex cells come up is in an array reference such as $A(i,\ i)$. Here the first i corresponds to a primary cell, but the second i cannot also be primary (logically, there can be only one iteration cell corresponding to the iteration coordinate i). Hence we make the second data cell complex. This means that in general, the second dimension of A will not be distributed over processors, but will be allocated serially "down memory". (If it is not, i.e. if the second dimension is distributed over a number of processors, then data motion will probably be necessary.)

7 Some simple examples

The reason for exposing the different maps in this data structure is that they can each serve as the locus of a compiler optimization. For an example, consider

forall $(i = 1{:}100,\ j = 1{:}100)$
  $A(i,\ j) = B(i){+}C(i)$

Let us assume first that A is distributed in both dimensions, and that B and C align with the first dimension of A:

!HPF$ **processors** $p(10,\ 10)$
!HPF$ **template** $t(100,\ 100)$
!HPF$ **distribute** $t($**block**, **block**$)$
!HPF$ **align** $A(i,\ j)$ **with** $t(i,\ j)$
!HPF$ **align** $B(i),\ C(i)$ **with** $t(i,\ 1)$

If we implement the "owner-computes" rule, then the iteration cells at each interior node will consist of two primary cells; one for i, and one for j. These cells are identical to the data cells at A. The data cells at B and C consist of one primary cell and one scalar cell. This is illustrated in Figure 5, where corresponding cells have the same cell place, and the scalar cell is denoted by s.

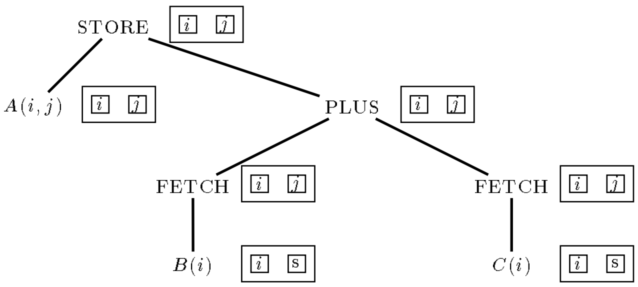


Figure 5: Example illustrating the "owner-computes" rule

This way of setting up the iteration cells forces the fetches of B and C to be remote fetches. Thus two remote operations are necessary to compute this assignment. An better alternative would be to change the iteration cells at the two FETCH nodes and the PLUS node so that in each case the j cell has *strip_length* 1, and a new scalar cell is introduced which has the same place as the former j cell, as in Figure 6.

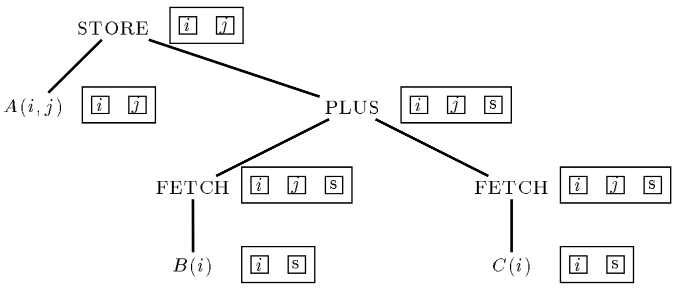


Figure 6: Alternative iteration cells.

Here constraints can be introduced so that the two fetches are local. The PLUS operation is also local, and only one data motion is needed—between the PLUS and the STORE.

Finally, suppose that B and C were actually replicated across the second dimension of A, for instance by means of the align directive

!HPF$ **align** $B(i)$, $C(i)$ **with** $t(i, *)$

Then we would have the situation in Figure 7 (the replicated cell is denoted by r).

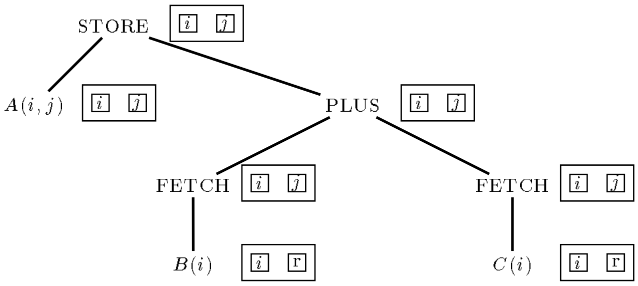


Figure 7: Example with replicated cells

In this case, no motion at all is necessary—a replicated cell in a node on the right-hand side of an assignment statement satisfies the appropriate constraint with any cell in its parent node having the same cell place.

Much more complicated cases, involving vector-valued subscripts and other complex expressions, can be dealt with similarly. In all these cases, this data structure is expressive enough to reflect the intentions of the programmer and to enable the compiler to do the analysis it needs to determine the correct placement of data motion, as well as to generate the code for that motion.

8 The role of the compiler

The fact that HPF only has directives that specify the distribution of data, but not of computations, is no accident. As these data structures make clear, it would be impractical in general to specify the distribution of computations in a program explicitly. While data distribution is defined once for each data object, and is declarative, the distribution of computations would have to be specified at each expression node in the program, and could in principle be different for each node. Such a program would not be readable or maintainable. It is more natural to let the compiler determine the distribution of computations.

9 Conclusion

Where computations are executed can be as important as where data is located. These data structures represent both phenomena in a parallel fashion, giving the compiler a unified view of both kinds of optimization.

Acknowledgements

Thanks for many helpful discussions to Kathleen Knobe, Joan Lukas, Michael Weiss, Venkat Natarajan, and Alex Nelson.

References

[1] Eugene Albert, Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Compiling Fortran 8x array features for the Connection Machine computer system. In *Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*. ACM SIGPLAN, July 1988.

[2] High Performance Fortran Forum. High performance Fortran language specification, Version 1.0. *Scientific Programming*, 2(1), May 1993. Also available as Technical Report CRPC-TR93300, Center for Research on Parallel Computation, Rice University, Houston, TX; and also via anonymous ftp from titan.cs.rice.edu in the directory public/HPFF/draft.

[3] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Massively parallel data optimization. In *Frontiers '88: The Second Symposium on the Frontiers of Massively Parallel Computation*, George Mason University, October 1988. IEEE.

[4] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[5] Kathleen Knobe and Venkataraman K. Natarajan. Data optimization: Minimizing residual interprocessor data motion on SIMD machines. In *Frontiers '90: The Third Symposium on the Frontiers of Massively Parallel Computation*, University of Maryland, October 1990. IEEE.