An Implementation Framework for HPF Distributed Arrays on Message-Passing Parallel Computer Systems

Kees van Reeuwijk, Will Denissen, Henk J. Sips, and Edwin M.R.M. Paalvast

Abstract—Data parallel languages, like High Performance Fortran (HPF), support the notion of distributed arrays. However, the implementation of such distributed array structures and their access on message passing computers is not straightforward. This holds especially for distributed arrays that are aligned to each other and given a block-cyclic distribution.

In this paper, an implementation framework is presented for HPF distributed arrays on message passing computers. Methods are presented for efficient (in space and time) local index enumeration, local storage, and communication.

Techniques for local set enumeration provide the basis for constructing local iteration sets and communication sets. It is shown that both local set enumeration and local storage schemes can be derived from the same equation. Local set enumeration and local storage schemes are shown to be orthogonal, i.e., they can be freely combined. Moreover, for linear access sequences generated by our enumeration methods, the local address calculations can be moved out of the enumeration loop, yielding efficient local memory address generation.

The local set enumeration methods are implemented by using a relatively simple general transformation rule for absorbing ownership tests. This transformation rule can be repeatedly applied to absorb multiple ownership tests. Performance figures are presented for local iteration overhead, a simple communication pattern, and storage efficiency.

Index Terms—HPF, message passing, message aggregation, distributed arrays, parallel computers.

1 Introduction

This paper describes a method to implement HPF (High Performance Fortran) [1] distributed arrays on message passing parallel computer systems. The implementation framework has been used as the basis of the HPF compiler developed in the Esprit project PREPARE [2], [3].

HPF is an extension to Fortran 90 to allow the generation of efficient programs for parallel computer systems. In this paper we assume that memory in a computer system is local to a processor or a group of processors, and that *messages* must be exchanged between the processors to inform them about changes to the local memory of the other processors. Such a system is called a *message-passing parallel computer system.*

HPF programs are no different from sequential programs, but annotations advise on how data can be distributed over the processors. In practice this only makes sense for arrays, and for operations on arrays. Therefore, HPF language constructs are focused on *distributed arrays* and on array operations: Fortran 90 array intrinsics and loops that iterate over array elements.

Both Fortran 90 and HPF are large and complex languages, so to keep things tractable we will describe the implementation of a part of HPF: independent FORALL loops with linear array access functions on arbitrary aligned and distributed arrays. Extension to general HPF is sketched, and includes general FORALLs, multidimensional arrays, and affine subscript functions with multiple iteration variables. The PREPARE HPF compiler also contains optimizations for stencil-type operations (using overlap) and irregular problems [4], [5]. However, treatment of these optimizations is beyond the scope of this paper.

1.1 HPF Alignment and Distribution

Array distributions in data-parallel languages such as HPF are intended to distribute the computational load evenly over the processors. In its simplest form, an array is divided into equal parts that are distributed over the available processors (Fig. 1a). This distribution is called *block distribution* of an array. Every processor "owns" a part of a distributed array, and is responsible for storing that part of the array. It is also responsible for communicating these elements to other processors when necessary.

By convention, each processor performs the computations which modify values of the array elements it owns; this is called the *owner computes* rule. In other words, the distribution of computation is determined by the distribution of the variable at the left-hand side (abbreviation: lhs) of the computation. As a consequence, the elements at the right-hand side (abbreviation: rhs) must be sent to the processor that executes the computation. For example, in the loop

```
!HPF$ INDEPENDENT
FORALL(I = 1 : 100) A(I) = B(I)+C(I)
```

• C. van Reeuwijk and H.J. Sips are with Delft University of Technology, Advanced School of Computing and Imaging, Lorentzweg 1, 2628 CJ Delft, the Netherlands. Email: {reeuwijk, henk}@cp.tn.tudelft.nl.
• W. Denissen and E.M. Paalvast are with TNO-TPD, PO Box 155, 2600 Delft, the Netherlands. Email: den-wja@tpd.tno.nl.

Manuscript received Dec. 6, 1994.
For information on obtaining reprints of this article, please send e-mail to: transpds@computer.org, and reference IEEECS Log Number D95207.

iteration `I` is executed by the owner of `A(I)`, and for each `I`, the elements `B(I)` and `C(I)` must be sent to the owner of `A(I)`.

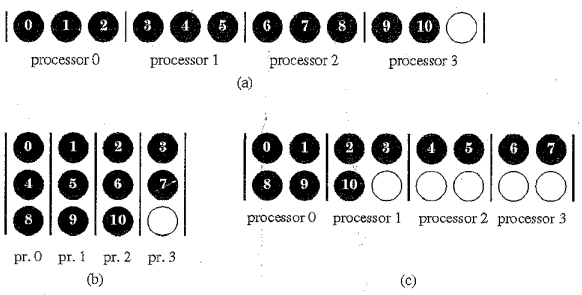

Fig. 1. Examples of HPF array distributions. In this example a one dimensional array of 11 elements, numbered 0 ... 10, is distributed over four processors. The distributions are block distribution (a), cyclic distribution (b), and cyclic(2) distribution (c).

For unconditional computations on an entire array, this gives a good load balance: Each processor owns the same number of elements, and must therefore usually perform the same amount of computation. However, array computations are not always that simple. In particular, array computations often work on only part of the array, and therefore engage only part of the processors. This causes an unbalance in the load. To alleviate this, other distribution schemes have been developed.

The most obvious is *cyclic* distribution (Fig. 1b). Every processor is given one element; after all processors have been given an element, the distribution starts again with the first processor. This is repeated until all elements have been distributed. Cyclic distribution makes load imbalance less likely, but it is likely to induce more communication. Many array computations only involve the neighbor elements of the element that is computed. If an array has cyclic distribution this implies that these neighbor elements are stored on another processor, and therefore they must be communicated to the computing processor.

As a compromise, HPF not only allows block and cyclic distribution, but also cyclic(m)[1] distribution (Fig. 1c shows cyclic(2) distribution). In this distribution, every processor is given m elements per cycle. Note that both block and cyclic distribution are special cases of cyclic(m) distribution: cyclic distribution is the same as cyclic(1) distribution, and block distribution is the same as $cyclic\left(\left\lceil n_i / n_p \right\rceil\right)$, where n_i is the size of the array, and n_p is the number of processors.

A further complication is that sometimes we want to align an array with another array. For example, to ensure local computation it might be required that every element i of array X resides on the same processor as element $2 \cdot i + 1$ of array Y. HPF allows such an alignment of arrays, provided that it is of the form $a \cdot i + b$ for arbitrary integers a and b. Since HPF allows alignment to an array that is itself aligned, alignment chains of arbitrary length can exist. However, they can always be collapsed into one ultimate alignment function. As a convenience to the user, HPF also allows alignment to a *template*: a fictitious array that only serves as a target of alignment.

Since alignment chains can be collapsed, and distributions can be copied, we assume in this paper, without loss of generality, that every array is aligned to a private template, and that that template is distributed according to one of the previously described distribution functions. For convenience of derivation, we also assume that all array dimensions start at element 0.

It is useful to visualize the effects of distribution and alignment, see Fig. 2. The dots represent template elements. Black dots are connected to array elements through an alignment function, white dots are not connected to an array element.

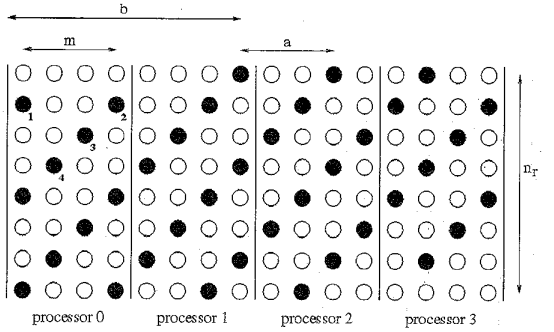

Fig. 2. An example of an array distribution (after Chatterjee et al. [6]). An array of 39 elements is aligned to a template through the function $f_{al}(i) = 3 \cdot i + 7$. The template is distributed cyclic(4) over four processors. The resulting distribution requires eight rows. In terms of the variables in this paper (see Section 2), $n_i = 39$, $n_p = 4$, $m = 4$, $a = 3$, $b = 7$, and $n_r = 8$.

1.2 Overview of the Paper

To support HPF distributed arrays, a number of separate issues must be addressed. These issues include local set enumeration and global-to-local array index conversion. Local set enumeration is needed for the generation of the local part of a loop iteration and the derivation of communication sets, global-to-local array index conversion comprises the transformation of global array references to local array references. In this paper, we will consider solutions to these two issues, and the efficiency of their implementation. Efficiency of implementation includes space and time considerations, e.g., an implementation must lead to a minimum of execution overhead, but must also remain compact to reduce complexity (no explosion of code or variables). We will show that by using a single transformation rule, which can be repeatedly applied, parallel code can be obtained in a systematic manner. We call this transformation *owner test absorption*.

In Section 3, the basic problems in deriving efficient parallel code are described. The basic implementation schemes are the subject of Section 4. In Section 5, generalizations of the results to full HPF are described, and a number of code–independent optimizations are derived

1. cyclic (m) distribution is also known as *block-cyclic* distribution.

for efficient local execution. Finally, in Section 6, the performance of our implementation is evaluated with respect to space and time overhead. To visualize the results of the translation, the translation of two small HPF programs is shown in an appendix.

2 Definition of Terms

In this paper, a number of linear functions are used to describe alignment and array access. Linear functions are of the form $f(i) = a \cdot i + b$, where a and b are integer constants. The following functions and constants are introduced:

$$f_{ix}(i) = a_{ix} \cdot i + b_{ix} \quad (1)$$

$$f_{al}(i) = a_{al} \cdot i + b_{al} \quad (2)$$

$$f_{co}(i) = a_{co} \cdot i + b_{co} \quad (3)$$

where f_{ix} is the index function, f_{al} is the alignment function, f_{co} is the composite of f_{al} and f_{ix}, with $a_{co} = a_{al} \cdot a_{ix}$, and $b_{co} = a_{al} \cdot b_{ix} + b_{al}$.

We define $g = gcd(a, m \cdot n_p)$. If we later use g_{al} it is implicitly assumed that the a and b in the expression are changed accordingly. Hence $g_{al} = gcd(a_{al}, m \cdot n_p)$. Similar assumptions are made for other expressions such as Δr. To indicate the upper bound of a variable x we use the expression x^{ub}. This means that for all x, $x < x^{ub}$.

Other symbols we use are:

- c The column number.
- g The greatest common divisor.
- i The index in the iteration space.
- m The m of the *cyclic(m)* distribution specification of HPF.
- n Required local storage for a distributed array.
- n_c The number of columns. Columns are numbered per processor.
- n_i The upper bound of the global iteration or array index i.
- n_p The number of processors in the processor array.
- n_r The number of rows.
- p The processor number.
- q The sequence number of a valid solution of a bounded Diophantine equation.
- r The row number.
- u The index in the local array, as calculated by the global-to-local function.

3 The Basic Problems

To motivate the calculations in the remainder of this paper, and to be able to discuss related work, it is useful to give an overview of the strategy we use.

3.1 Trivial Alignment

First, let us assume an array distribution with $a = 1$ and $b = 0$: the identity alignment function. There are two obvious methods to enumerate the local elements of a processor: rowwise and columnwise. This is shown in Fig. 3 and Fig. 4, respectively. Since the implementation of these alternatives is very similar, we must look at the size of the inner loop to choose between them. For rowwise enumeration the size of the inner loop is m, for columnwise enumeration the size of the inner loop is (averaged over all processors) $n_i/(m \cdot n_p)$. For each of the alternatives there are cases where the efficiency is extremely low: For rowwise enumeration this is the case when $m = 1$, and for columnwise enumeration this is the case when $n_i \leq m \cdot n_p$. Unfortunately, these cases correspond to CYCLIC(1) and BLOCK distribution in HPF, and therefore are both quite likely to occur in practice. Therefore, we conclude that both rowwise and columnwise enumeration are useful.

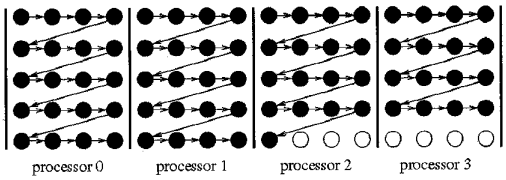

Fig. 3. Rowwise enumeration of an array: $a = 1$, $b = 0$, $m = 4$, and $n_p = 4$.

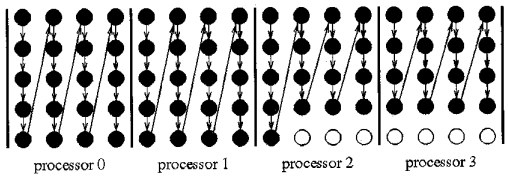

Fig. 4. Columnwise enumeration of an array: $a = 1$, $b = 0$, $m = 4$, and $n_p = 4$.

To store the elements of a distributed array, we must have a local array on each processor. For example, the local array shown in Fig. 3 and Fig. 4 would require 20 elements. These figures also indicate two possible storage schemes for local elements. The storage schemes can be interpreted as rowwise or columnwise storage of a two–dimensional array.

More generally, we can consider the distribution of a one dimensional array as a reshape into a three–dimensional array. The new array is indexed with a processor number, row number, and column number, which are derived from the original index.

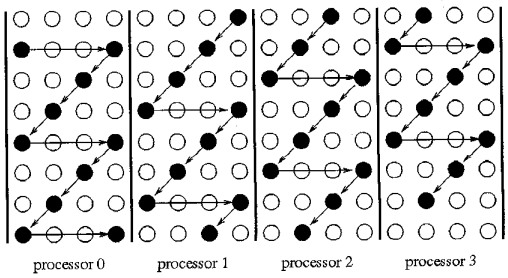

Fig. 5. Rowwise enumeration of an array: $a = 3$, $b = 7$, $m = 4$, and $n_p = 4$.

3.2 Nontrivial Alignment

For nontrivial alignment functions it is still possible to scan the array in rowwise order and columnwise order, see Fig. 5 and Fig. 6. To store the elements we can "compress" either the rows or the columns of the array, so according to Fig. 5, an 8×2 array can be used, and according to Fig. 6, a 3×4 array can be used. Both of these arrays can be stored in rowwise or columnwise order.

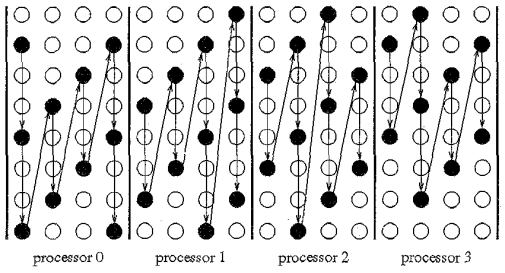

Fig. 6. Columnwise enumeration of an array: $a = 3$, $b = 7$, $m = 4$, and $n_p = 4$.

3.3 Efficiency

When evaluating the efficiency of an implementation, we should consider how *scalable* it is. Informally, this means that an implementation should be efficient for an array containing 1,000 elements that are distributed over 16 processors, but also for an array containing 1,000,000 elements that are distributed over 10,000 processors. Naive implementation schemes can be highly inefficient. Consider, for example, the following HPF program:

```
    REAL A(1000000)
!HPF$ PROCESSORS P(10000)
!HPF$ DISTRIBUTE CYCLIC(10) ONTO P :: A
    INTEGER I

!HPF$ INDEPENDENT
    FORALL(I = 1:1000000) A(I) =...
```

With *guarded execution* this would be implemented as:[2]

```
for(i = 1; i <= 1000000; i++) {
    if(owner(A[i]) == me) A[i] =...;
}
```

where **me** is a variable that holds the number of the local processor. Since **A** is distributed over 10,000 processors, on each processor only one test in every 10,000 succeeds, causing a considerable overhead. Scaleable implementation schemes must have a large fraction of guarded expressions that are executed, and preferably must eliminate the guards completely.

More subtle sources of inefficiency are **for** loops. In some situations the average size of a loop is small, possibly less than one. In that case the loop is in effect reduced to an **if** statement. In general, we want **for** loops to span ranges that are as large as possible, especially in inner loops, to reduce loop overhead.

To support HPF distributed arrays the following aspects are important:

- *Enumeration of Local Elements.* Given an independent FORALL statement, and given that the owner computes rule is used, how are the local elements of an array enumerated? Note that the FORALL of HPF has parallel assignment semantics. This means that we can execute the iterations of the FORALL loop in arbitrary order. This allows considerable freedom in the implementation.

  As we will show in this paper, guarded execution can be replaced with two different implementations where only local indices are enumerated, provided that the lhs array is accessed through an affine function. If the access is not affine, a different method may be more suitable, for example using a special communication library such as PARTI [4], [5]. For the affine case, however, we can always enumerate the local elements explicitly. Therefore we can evaluate the enumeration methods on their overhead.
- *Representation of Distributed Arrays.* How do we store the elements of a distributed array on each processor? If an array is distributed, each processor must allocate some space for it, called the *local array*. We must find a way to calculate the required size of the local array, and we must also find a function to map an index value in the global array onto an index value in the local array. This function is called the *global-to-local* function.

  For the representation of distributed arrays, both the *storage overhead* and the *execution overhead* are important. The execution overhead is caused by the fact that every reference to a global array element must be translated to a reference to a local array element. Since this translation can be costly, some implementation schemes (including ours) allow some local *holes* (unused elements) to simplify the translation. This introduces storage overhead. Hence it is important to find an implementation scheme that minimizes both the required storage and the time required for global to local index translation.

  Another problem with distributed arrays is that the results of communication must be stored. A number of special storage schemes have been proposed for this, but they usually require different expressions to access local and communicated foreign elements. Instead of these storage schemes we introduce temporary arrays wherein both local and foreign elements are copied. These temporary arrays are accessed in the usual way.
- *Communication.* How do we efficiently construct the messages that must be exchanged to implement an array assignment or must precede a computation?

  For communication, the efficiency of many systems is largely determined by the number of messages that is generated. The implementations we discuss in this paper all generate at most one message to each of the other processors for each array that is accessed, so we do not have to compare the implementations explicitly on this aspect. Instead, the implementations are compared on the processor overhead of constructing the messages.

2. Throughout this paper we use the programming language C to describe generated code or sketches thereof.

3.4 Related Work

The automatic generation of message passing programs from data distribution specifications has been explored for some time in the context of various data parallel languages [7], [8], [9], [10], [11], [12]. The recent definition of HPF [1] has added some new data alignment and data distribution features for which no efficient solutions existed. As a consequence, new results have been reported in [13], [6], [14], [15], [16], [17], [18], [19], [20], [21] and, more recently and concurrently with this paper, [22], [23], [20], [24], [25], [26].

Early optimization techniques only consider nonaligned arrays. The first optimizations were reported by Callahan and Kennedy [7] and Gerndt [8]. They considered nonaligned *block(m)* distributions with linear array access functions. Gerndt also showed how overlap can be handled. In Paalvast et al. [10] a solution for monotone array access functions and *block(m)* distributions was given. Solutions for *cyclic*(1) and linear array access functions have been independently reported by Koelbel and Mehrotra [11] and Koelbel [27] for the language Kali, and Paalvast et al. [10] for the language Booster [28], [29].

For *cyclic(m)* distributions a rowwise and columnwise solution has been given in Paalvast et al. [30], [29]. The rowwise solution allows array access functions to be monotone and the columnwise solution requires the array access function to be linear. These solutions provided a basis for the results in this paper.

More recent publications also take alignment into account. Most results assume that dimensions are independent of each other and/or have restrictions on the class of alignment functions allowed. Tseng [13] considers *block(m)* and *cyclic*(1) distributions with an alignment coefficient of one. Two related approaches have been presented by Stichnoth et al. [14], [15] and Gupta et al. [17]. Essential in these approaches is the notion of virtual processors for solving the *cyclic(m)* case. Each *block(m)* or *cyclic*(1) solution for a regular section of an array is assigned to a virtual processor, yielding a so called virtual-block or virtual-cyclic view, respectively. Stichnoth et al. [14], [15] use a virtual-cyclic view and use the intersection of array slices for communication generation. Gupta et al. [17] present closed form solutions for nonaligned arrays for both the virtual-block and the virtual-cyclic view. This method is extended to aligned arrays in Kaushik et al. [24]. To reduce calculation overhead in the virtual processor approach, Kaushik et al. [18] propose a method to reuse these calculations for classes of related access functions.

Chatterjee et al. describe a nonlinear method to enumerate local elements [6]. They observed that the sequence of positions of elements in the rows of a template shows a pattern that repeats itself after a number of rows. Let us call such a range of rows the *pattern cycle* of a distribution, and let us call this enumeration method *pattern-cyclic enumeration*. For instance, the elements {1, 2, 3, 4} in Fig. 2 form such a pattern. Chatterjee et al.'s contribution is the construction of an algorithm to find this pattern and place the elements of this pattern in an in-order sequence. From this sequence, a finite state machine (FSM) is constructed which is used to successively access each element.

In the original paper of Chatterjee et al., the construction of the FSM requires a full sorting operation. Recent papers describe more efficient methods [21], [22], [23], [26]. In [21], a linear algorithm for constructing the FSM for two special cases is given. Linear algorithms for the general case are given in [22], [23], [26]. Kennedy et al. also showed [26] that their method can be used without a table, using a demand driven evaluation scheme, at the expense of some performance.

As will be shown in this paper, rowwise and columnwise decomposition tend to be complementary: One is inefficient where the other is efficient and vice versa. The efficiency of pattern cycle enumeration is related to rowwise enumeration. In effect, pattern cycle enumeration allows one or more rows to be enumerated in the inner loop. How many rows are enumerated is strongly dependent on the alignment and index access function applied. This dependency can only be relaxed by enumerating more than one pattern cycle at once, at the expense of applying a two-level table scheme as shown in [26] or at the expense of larger tables (by unrolling the table).

A different approach to local set enumeration for distributed array access and communication is the use of linear algebra techniques. This approach is followed by Ancourt et al. [20] and Le Fur et al. [12]. Le Fur et al. give solutions for commutative loop nests on block distributed and nonaligned arrays. Ancourt et al. present a solution for the general case. Linear algebra techniques can be quite costly when not all parameter values are known at compile-time. This might be a problem in HPF programs. For example, the number of processors is often only known at run-time. To avoid this, Ancourt et al. [20] also present a symbolic solution for columnwise local index enumeration.

As already explained, the storage and access of distributed arrays is another problem that needs attention. Chatterjee et al. [6] use a local storage compression scheme in which the array elements are stored in lexicographic order without holes. Because the FSM for enumerating elements directly enumerates local elements, explicit global-to-local calculations are not necessary. In Kaushik et al. [24], similar dense storage compression schemes are employed. Each scheme is specific for an enumeration method. Changing the enumeration method for an array object leads to a reallocation of the array. Also, no global-to-local formula is given, resulting in table generation time overhead for translating the global index sets to the local index sets. Stichnoth et al. [15] use a block compression method with the cycle number as second index. Ancourt et al. [20] derive a formula for a columnwise compression method. Their global-to-local function remains rather complicated, although they do remark that in many cases more efficient solutions can be obtained.

Mahéo and Pazat [31] use another approach to simplify access to local arrays. Their method is based on page driven array management. The main advantage of their approach is that they can provide a very efficient global-to-local function. This comes at the expense of storage overhead and page management.

4 Enumeration, Storage, and Communication Schemes

For the realization of the enumeration strategies described in the previous section, we must calculate the parameters that are associated with them. There is a relation between the global index i and the processor number p, row number r, and column number c. This relation is given by (see Fig. 7):

$$a \cdot i + b = n_p \cdot m \cdot r + m \cdot p + c \tag{4}$$

We will call this equation the *position equation*. Given an i, we can derive r, c, and p as follows:

$$r = \left\lfloor \frac{a \cdot i + b}{n_p \cdot m} \right\rfloor \tag{5}$$

$$c = (a \cdot i + b) \bmod m \tag{6}$$

$$p = \left\lfloor \frac{a \cdot i + b}{m} \right\rfloor \bmod n_p \tag{7}$$

The expression for p in (7) is often called the *owner function*, and is also written in this paper as

$$owner(i) = \left\lfloor \frac{(a \cdot i + b)}{m} \right\rfloor \bmod n_p$$

We also know that

$$\begin{aligned} 0 \le i < n_i \\ 0 \le p < n_p \\ 0 \le r < n_r \\ 0 \le c < n_c \end{aligned} \tag{8}$$

Since $n_c = m$, we will use m in the remainder of the paper.

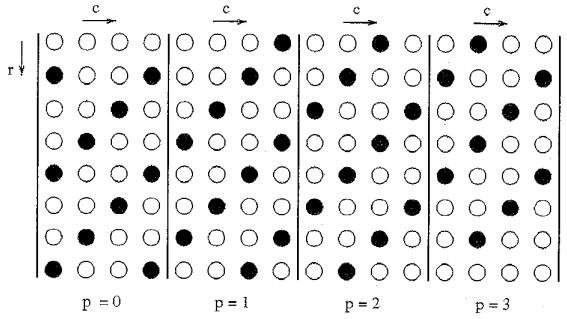

Fig. 7. The definition of the local position variables: c is the column number, p is the processor number, and r is the row number.

The variable n_r is the maximal row number; it can be derived from the distribution parameters. By substituting $i = n_i - 1$ in (5), the maximal row number can be expressed as:

$$r^{max} = \left\lfloor \frac{a \cdot (n_i - 1) + b}{n_p \cdot m} \right\rfloor$$

In a similar way the minimal row number can be calculated by substituting $i = 0$ in (5), resulting in $r^{min} = \left\lfloor b / (n_p \cdot m) \right\rfloor$. Since a multiple of $n_p \cdot m$ can be added to b without affecting the distribution of the elements over the processors (only the row numbers are changed), we assume without loss of generality that $0 \le b < n_p \cdot m$, so that r^{min} is always zero. Hence, the number of rows n_r is given by:

$$n_r = 1 + r^{max} - r^{min} = 1 + \left\lfloor \frac{a \cdot (n_i - 1) + b}{n_p \cdot m} \right\rfloor$$

For rowwise enumeration we have to find, given a processor number p and a row number r, which column numbers meet (4). For columnwise enumeration we have to find, given a processor number p and a column number c, which row numbers meet (4). In this section, it will be shown that the row numbers or column numbers that meet these conditions are easily found by solving a linear Diophantine equation. Since linear Diophantine equations are essential to our solution, we will first give a brief overview of the relevant theory and associated formulas needed in the remainder of this paper. More details can be found in books on number theory, for example [32].

4.1 Mathematical Preliminaries

Given $\alpha, \beta \in Z$, Euclid's extended algorithm calculates the *greatest common divisor* (*gcd*) of α and β. That is, it finds the maximal $g \in Z$, such that $\alpha/g \in Z$ and $\beta/g \in Z$. As a side effect, it also finds $x, y \in Z$ such that[3]

$$g = x \cdot \alpha - y \cdot \beta$$

Euclid's extended algorithm can also be used to find the solutions of a linear Diophantine equation. Given

$$\alpha, \beta, \nu \in Z$$

we want to find all $x, y \in Z$ that satisfy the equation:

$$\alpha \cdot x = \beta \cdot y + \nu \tag{9}$$

Let $g = \gcd(\alpha, \beta)$. If g does not divide ν there are no solutions. If g divides ν there are infinitely many solutions, described by

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} q + \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} j \tag{10}$$

for arbitrary $j \in Z$, where

$$q = \nu/g \tag{11}$$

$$g = x_0 \cdot \alpha - y_0 \cdot \beta \tag{12}$$

$$\Delta x = \beta/g \tag{13}$$

$$\Delta y = \alpha/g \tag{14}$$

The values of x_0, y_0, Δx, Δy, and g can be found with Euclid's extended algorithm. We write this as the following function:

$$euclid(\alpha, \beta) \rightarrow (x_0, y_0, g, \Delta x, \Delta y) \tag{15}$$

Note that $(-x_0, -y_0, g, -\Delta x, -\Delta y)$ is also a valid result; we assume a solution is chosen where $\Delta y > 0$. This simplifies

3. Actually, literature usually states that $x, y \in Z$, $g = x \cdot \alpha + y \cdot \beta$ are found. This is a trivial substitution of variables; the form we use is more convenient for the derivations in this paper.

the derivations. Now let us suppose that the variables are bounded by

$$\begin{aligned} 0 \le x < x^{ub} \\ 0 \le y < y^{ub} \end{aligned} \tag{16}$$

By using (10), we know that

$$\begin{aligned} 0 \le x_0 \cdot q + \Delta x \cdot j < x^{ub} \\ 0 \le y_0 \cdot q + \Delta y \cdot j < y^{ub} \end{aligned} \tag{17}$$

Since all other variables are known, we can use these bounds to calculate the range of j that leads to solutions within the given bounds. We have ensured that $\Delta y > 0$, so the bound on j caused by y is easily simplified to:

$$\left\lceil \frac{-y_0 \cdot q}{\Delta y} \right\rceil \le j < \left\lceil \frac{y^{ub} - y_0 \cdot q}{\Delta y} \right\rceil \tag{18}$$

If $\Delta x > 0$ the bound on j caused by x can be simplified to:

$$\left\lceil \frac{-x_0 \cdot q}{\Delta x} \right\rceil \le j < \left\lceil \frac{x^{ub} - x_0 \cdot q}{\Delta x} \right\rceil \tag{19}$$

else it becomes:

$$\left\lceil \frac{x^{ub} - x_0 \cdot q}{\Delta x} \right\rceil \le j < \left\lceil \frac{-x_0 \cdot q}{\Delta x} \right\rceil \tag{20}$$

The valid range of j is the intersection of (18) and (19) or of (18) and (20).

Now let us consider Diophantine equations with more variables. Given $\alpha, \beta_0, \cdots \beta_k, \nu \in Z$ we want to find all $x, y_0, \cdots, y_k \in Z$ that satisfy the equation:

$$\alpha \cdot x = \beta_0 \cdot y_0 + \cdots + \beta_k \cdot y_k + \nu \tag{21}$$

If we focus on the variables x and y_0, we see that there can only be a solution if the remainder of the equation is a multiple of $g_0 = \gcd(\alpha, \beta_0)$. Therefore we have:

$$\begin{aligned} \alpha \cdot x &= \beta_0 \cdot y_0 + g_0 \cdot q_0 \\ g_0 \cdot q_0 &= \beta_1 \cdot y_1 + \cdots + \beta_k \cdot y_k + \nu \end{aligned} \tag{22}$$

We have *decomposed* the original Diophantine equation into two smaller Diophantine equations. When we repeat this process, we end up with the following equations:

$$\begin{aligned} \begin{pmatrix} x \\ y_0 \end{pmatrix} &= \begin{pmatrix} x_0 \\ y_{0_0} \end{pmatrix} q_0 + \begin{pmatrix} \Delta x \\ \Delta y_0 \end{pmatrix} j_0 \\ \begin{pmatrix} q_0 \\ y_1 \end{pmatrix} &= \begin{pmatrix} q_{0_0} \\ y_{1_0} \end{pmatrix} q_1 + \begin{pmatrix} \Delta q_0 \\ \Delta y_1 \end{pmatrix} j_1 \\ &\vdots \quad\quad\quad \vdots \\ \begin{pmatrix} q_{k-1} \\ y_k \end{pmatrix} &= \begin{pmatrix} q_{k-1_0} \\ y_{k_0} \end{pmatrix} q_k + \begin{pmatrix} \Delta q_{k-1} \\ \Delta y_k \end{pmatrix} j_k \\ q_k &= v / g_k \end{aligned} \tag{23}$$

where

$$\begin{aligned} &euclid(\alpha, \beta_0) \rightarrow (x_0, y_{00}, g_0, \Delta x, \Delta y_0) \\ &euclid(g_0, \beta_1) \rightarrow (q_{00}, y_{10}, g_1, \Delta q_0, \Delta y_1) \\ &\quad\quad\quad\quad \vdots \quad\quad\quad\quad \vdots \\ &euclid(g_{k-1}, \beta_k) \rightarrow (x_{k0}, y_{k0}, g_k, \Delta x_k, \Delta y_k) \end{aligned} \tag{24}$$

Now let us assume that we have bounds on the variables x and y_i:

$$\begin{aligned} 0 \le x < x^{ub} \\ 0 \le y_i < y_i^{ub} \end{aligned} \tag{25}$$

Because there is a linear relation between j_i and y_i, all y_i can be generated by enumerating a range of j_i. The bounds on y_i give bounds for j_i (remember that $\Delta y_i > 0$):

$$\left\lceil \frac{-y_{i_0} \cdot q_i}{\Delta y_i} \right\rceil \le j < \left\lceil \frac{y_i^{ub} - y_{i_0} \cdot q_i}{\Delta y_i} \right\rceil \tag{26}$$

Moreover, the bound on x gives another bound on j_0, so that j_0 is bounded by two different bounds: one caused by y_0 and one caused by x. If $\Delta x > 0$ then:

$$\left\lceil \frac{-x_0 \cdot q_0}{\Delta x_0} \right\rceil \le j_0 < \left\lceil \frac{x_0^{ub} - x_0 \cdot q_0}{\Delta x_0} \right\rceil \tag{27}$$

else

$$\left\lceil \frac{x_0^{ub} - x_0 \cdot q_0}{\Delta x_0} \right\rceil \le j_0 < \left\lceil \frac{-x_0 \cdot q_0}{\Delta x_0} \right\rceil \tag{28}$$

If we want to generate all solutions that satisfy the bounds, we can do this by enumerating all tuples $(j_0, \cdots j_k)$ that satisfy the bounds. For j_0 the intersection of both sets of bounds must be used.

The variables j_i have a lower bound that is in general not equal to 0. If we want to have variables that have a lower bound of 0 (for example because we want to use them as array indices, see Section 4.3), we can easily introduce "shifted" variables:

$$u_i = j_i - \left\lceil \frac{-y_{i_0} \cdot q_i}{\Delta y_i} \right\rceil \tag{29}$$

In that case, the bound becomes

$$0 \le u_i < \left\lceil \frac{y_i^{ub} - y_{i_0} \cdot q_i}{\Delta y_i} \right\rceil - \left\lceil \frac{-y_{i_0} \cdot q_i}{\Delta y_i} \right\rceil \tag{30}$$

Since for arbitrary a and b it holds

$$\lceil a \rceil - 1 \le \lceil a - b \rceil - \lceil b \rceil \le \lceil a \rceil$$

this can be approximated by:

$$0 \le u_i < \left\lceil \frac{y_i^{ub}}{\Delta y_i} \right\rceil \tag{31}$$

This approximation may introduce values of u_i that are not associated with a valid value of j_i, but there are circumstances where this is acceptable; for example when the bounds are used to calculate the required local storage space of an array.

In local address calculations it is necessary to calculate u_i from y_i. From (23) it follows that

$$y_i = y_{i0} \cdot q_i + \Delta, y_i \cdot j_i,$$

or:

$$j_i = (y_i - y_{i0} \cdot q_i) \,/\, \Delta, y_i.$$

Substituting this in (29) and using $-\lceil -a / b \rceil = \lfloor a / b \rfloor$ yields:

$$u_i = \left\lfloor \frac{y_i}{\Delta y_i} \right\rfloor \tag{32}$$

Using (31) and (32), we get the very useful relation:

$$0 \le \left\lfloor \frac{y_i}{\Delta y_i} \right\rfloor < \left\lceil \frac{y_i^{ub}}{\Delta y_i} \right\rceil \tag{33}$$

4.2 The Decomposition of the Position Equation

The theory in the previous section can be used to decompose the position equation (4). In principle there are 4! = 24 possible ways to associate the variables in (4) with x and y_i of the previous section. By association we mean that the terms of (4) are rearranged to match the appropriate terms in (21). Not all possible associations are useful, however. We want the processor loop to be the outer loop, so that it can be removed for the parallel version of the code. Hence this must be the rightmost term in the position equation. Also, since we want to calculate i, this term must be put at the lhs of the equation. This leaves two possible decompositions:

- Rowwise. In this case we relate the equation

  $$a \cdot i = c + n_p \cdot m \cdot r + m \cdot p - b$$

  to (21).
- Columnwise. In this case we relate the equation

  $$a \cdot i = n_p \cdot m \cdot r + c + m \cdot p - b$$

  to (21).

4.2.1 Rowwise Decomposition

If we apply the decomposition on

$$a \cdot i = c + n_p \cdot m \cdot r + m \cdot p - b$$

we get:

$$\begin{aligned} \begin{pmatrix} i \\ c \end{pmatrix} &= \begin{pmatrix} 0 \\ -1 \end{pmatrix} n_0 + \begin{pmatrix} 1 \\ a \end{pmatrix} j_0 \\ \begin{pmatrix} n_0 \\ r \end{pmatrix} &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} n_1 + \begin{pmatrix} n_p \cdot m \\ 1 \end{pmatrix} j_1 \\ \begin{pmatrix} n_1 \\ p \end{pmatrix} &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} n_2 + \begin{pmatrix} m \\ 1 \end{pmatrix} j_2 \\ n_2 &= -b \end{aligned} \tag{34}$$

where

$$\begin{aligned} &euclid(a, 1) \rightarrow (0, -1, 1, 1, a) \\ &euclid(1, n_p \cdot m) \rightarrow (1, 0, 1, n_p \cdot m, 1) \\ &euclid(1, m) \rightarrow (1, 0, 1, m, 1) \end{aligned} \tag{35}$$

Note that the results of all of the Euclid calculations are known, so no Euclid calculations are necessary at run time or compile time. Using the bounds of (8) we get:

$$\begin{aligned} 0 \le \quad & j_2 = p < n_p \\ 0 \le \quad & j_1 = r < n_r \\ 0 \le -n_p \cdot m \cdot j_1 - m \cdot j_2 + b + a \cdot \quad & j_0 = c < m \\ 0 \le \quad & j_0 = i < n_i \end{aligned} \tag{36}$$

The bounds (36i) and (36c) together determine the valid range of j_0. For all $r < n_r - 1$ the bounds (36i) will be wider than the bound (36c), but for $r = n_r - 1$ the bound (36c) will be wider than (36*i*).

4.2.2 Columnwise Decomposition

If we apply the decomposition on

$$a \cdot i = n_p \cdot m \cdot r + c + m \cdot p - b$$

we get:

$$\begin{aligned} \begin{pmatrix} i \\ r \end{pmatrix} &= \begin{pmatrix} i_0 \\ r_0 \end{pmatrix} n_0 + \begin{pmatrix} \Delta i \\ \Delta r \end{pmatrix} j_0 \\ \begin{pmatrix} n_0 \\ c \end{pmatrix} &= \begin{pmatrix} 0 \\ -1 \end{pmatrix} n_1 + \begin{pmatrix} 1 \\ g \end{pmatrix} j_1 \\ \begin{pmatrix} n_1 \\ p \end{pmatrix} &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} n_2 + \begin{pmatrix} m \\ 1 \end{pmatrix} j_2 \\ n_2 &= -b \end{aligned} \tag{37}$$

where

$$\begin{aligned} &euclid(a, n\_p \cdot m) \rightarrow (i_0, r_0, g, \Delta i, \Delta r) \\ &euclid(g, 1) \rightarrow (0, -1, 1, 1, g) \\ &euclid(1, m) \rightarrow (1, 0, 1, m, 1) \end{aligned} \tag{38}$$

Note that two of the Euclid calculations are known, so at most a single Euclid calculation is needed at run time or compile time. Using the bounds of (8) we get:

$$\begin{aligned} 0 \le \quad & j_2 = p < n_p \\ 0 \le b - m \cdot j_2 + g \cdot j_1 = c < m \\ 0 \le \quad & i_0 \cdot j_1 + \Delta i \cdot j_0 = i < n_i \\ 0 \le \quad & r_0 \cdot j_1 + \Delta r \cdot j_0 = r < n_r \end{aligned} \tag{39}$$

The bounds (39i) and (39r) both determine the range of valid j_0. However, since n_r is derived from n_i, the upper bound of (39r) will always be wider than the upper bound of (39i).

4.3 The Global-to-Local Function

The global-to-local function must map an index in the distributed array, called the *global* index, onto an index in the local array. We have considerable freedom in defining this function; as long as every different (r, c) pair that occurs is mapped on a different local address, the function is correct (in other words, the function is bijective).

The decompositions derived in Section 4.2 can be used to implement storage schemes for the local elements of a distributed array. We will derive such a storage scheme for both rowwise storage and columnwise storage. In both schemes, we allow some *holes* to occur in the locally allocated space of the array. That is, some local memory elements may not be associated with an array element. For both derivations the percentage of holes may become very high. Fortunately, columnwise storage tends to be efficient when rowwise storage is inefficient, and vice versa, as will be shown in Section 6.

In the storage schemes below we assume that $a = a_{al}$, $b = b_{al}$, and $a > 0$. If an array has $a < 0$, we can always reverse the order in which the elements are stored by the substitutions:

$$\begin{aligned} a &\rightarrow -a' \\ b &\rightarrow b' + (n_i - 1) \cdot a' \end{aligned}$$

In Section 5.7, we will show how to eliminate global-to-local calculations from local set enumerations. To be compatible with these eliminations, it is necessary to use the same

row and column numbers in rowwise and columnwise storage. Therefore, only the bounds on r and c are used. Without this requirement, columnwise storage could use the bounds on i, resulting in some cases in (slightly) less memory occupation. Considering the advantages of global-to-local elimination, we consider the larger memory usage acceptable.

4.3.1 Rowwise Storage

Using the bounds (36r) and (36c), and using (33), we get:

$$\begin{aligned} &(r)\ \ 0 \le r < n_r \\ &(c)\ \ 0 \le \left\lfloor \frac{c}{a} \right\rfloor < \left\lceil \frac{m}{a} \right\rceil \end{aligned} \tag{40}$$

Therefore, all local elements of the distributed array can be stored in

$$n = n_r \cdot \left\lceil \frac{m}{a} \right\rceil \tag{41}$$

elements, and we can introduce the global-to-local function[4]

$$g2l'_{row}(r,c) = r \cdot \left\lceil \frac{m}{a} \right\rceil + \left\lfloor \frac{c}{a} \right\rfloor \tag{42}$$

Using (5) and (6), we get the global-to-local function:

$$g2l_{row}(i) = r \cdot \left\lfloor \frac{a \cdot i + b}{n_p \cdot m} \right\rfloor \cdot \left\lceil \frac{m}{a} \right\rceil + \left\lfloor \frac{(a \cdot i + b) \bmod m}{a} \right\rfloor \tag{43}$$

4.3.2 Columnwise Storage

Using the bounds (36r) and (36c), and using (33), we get:

$$\begin{aligned} &(r)\ \ 0 \le \left\lfloor \frac{r}{\Delta r} \right\rfloor < \left\lceil \frac{n_r}{\Delta r} \right\rceil \\ &(c)\ \ 0 \le \left\lfloor \frac{c}{g} \right\rfloor < \left\lceil \frac{m}{g} \right\rceil \end{aligned} \tag{44}$$

The bounds of (44) indicate that all local elements of the distributed array can be stored in

$$n = \left\lceil \frac{m}{g} \right\rceil \cdot \left\lceil \frac{n_r}{\Delta r} \right\rceil \tag{45}$$

elements, and we can use the global-to-local function[5]

$$g2l'_{col}(r,c) = \left\lfloor \frac{r}{\Delta r} \right\rfloor \cdot \left\lceil \frac{m}{g} \right\rceil + \left\lfloor \frac{c}{g} \right\rfloor \tag{46}$$

Using (5), (6) and (46), the global-to-local function becomes:

$$g2l_{col}(i) = \left\lfloor \frac{a \cdot i + b}{n_p \cdot m \cdot \Delta r} \right\rfloor \cdot \left\lceil \frac{m}{g} \right\rceil + \left\lfloor \frac{(a \cdot i + b) \bmod m}{g} \right\rfloor \tag{47}$$

This substitution uses the property that

$$\left\lfloor \frac{\left\lfloor \frac{a}{b} \right\rfloor}{c} \right\rfloor = \left\lfloor \frac{a}{(b \cdot c)} \right\rfloor$$

if and only if $c > 0$.

4.3.3 Storage for Block and Cyclic Distribution

Block distribution can be considered a special case of block–cyclic distribution, where we know that $r = 0$, and $n_r = 1$. Since there is only one row, rowwise storage is the most attractive. We can easily derive a global-to-local function by substituting the known constants and (6) in (42). This yields:

$$g2l_{block}(i) = \left\lfloor \frac{(a \cdot i + b) \bmod m}{a} \right\rfloor \tag{48}$$

If it is known on which processor p the element is located, this can be calculated more efficiently as:

$$g2l_{block}(i) = \left\lfloor \frac{(a \cdot i + b) - m \cdot p}{a} \right\rfloor \tag{49}$$

In a similar way, we can consider cyclic distribution a special case of block–cyclic distribution with $c = 0$, and $m = 1$. Since there is only one column, columnwise storage is the most attractive. We can easily derive a global-to-local function by substituting the known constants and (5) in (46). This yields:

$$g2l_{cyclic}(i) = \left\lfloor \frac{a \cdot i + b}{m \cdot n_p \cdot \Delta r} \right\rfloor \tag{50}$$

4.3.4 Hybrid Storage

Both rowwise storage and columnwise storage may be highly inefficient for some distributions (see Section 6). However, (43) and (47) only differ in constants. Therefore, it is possible to support both by choosing one of the two sets of constants, depending on the required size for each of the storage strategies. This choice needs only be made at the moment the array is created.

Since rowwise storage tends to be efficient (have few or no holes) where columnwise storage is inefficient and vice versa, the hybrid storage scheme will be efficient in most cases.

4.4 Local Set Enumeration

Let us now examine how to enumerate the local execution set of a FORALL. Since the owner-computes rule is used, this is equivalent with enumerating a subset of the local index set of a distributed array. For the moment we restrict ourselves to the following:

- The iteration starts at 0, and progresses with step 1.
- The FORALL contains only a single assignment statement.
- The FORALL statement is annotated as INDEPENDENT.
- The access function for the array at the lhs is affine.

Later (in Section 5) this will be generalized. For the moment we assume that the rhs array elements are available some-

4. As explained before, the distribution of a one–dimensional array can be seen as a restructuring into a three–dimensional array that is indexed by a processor, row, and column number. In this view, each processor owns a two–dimensional slice of this three–dimensional array. To map the elements of such a two–dimensional array onto storage locations, it must be "flattened" to a one–dimensional array. Here, one of two obvious ways to do this is used. If we used the other one, we would get: $g2l'_{row}(r,c) = \lfloor c/a \rfloor \cdot n_r + r$.

5. See the footnote of (42).

how. Since the owner computes rule is used, the lhs array elements are locally available. Considering these restrictions, the loop will be of the form:

```
!HPF$independent
 forall(i=0:ni-1:1)
   a(d*i+e) =...
 end forall
```

where $d = a_{ix}$ and $e = b_{ix}$ are the coefficients of the linear function f_{ix}. To use this function in the position equation, it must be composed with the alignment function of array **A**, resulting in the composite index function f_{co}. We can then substitute the composite index function in (4), resulting in:

$$a_{co} \cdot i + b_{co} = n_p \cdot m \cdot r_{co} + m \cdot p + c_{co} \tag{51}$$

Using the decompositions of the position equation shown in the previous section, the problem can be reduced to

1) enumerating all valid (j_0, j_1, j_2) tuples of (34) or (37),
2) enforcing the owner computes rule, and
3) calculating i for each tuple value.

For Step 1, the bounds of (36) suggest a possible method to enumerate the valid tuple values: enumerate all valid j_2 as specified by (36p); for each value of j_2 enumerate all valid j_1 as specified by (36r); and for each valid pair of (j_1, j_2) enumerate all valid j_0 as specified by (36c) and (36i). The bounds of (39) can be used in a similar way. We call these enumeration methods *rowwise enumeration* and *columnwise enumeration*, respectively. For Step 2, it is sufficient to let each processor handle the value of $j_2 = p$ that corresponds with its own processor number. Finally, for Step 3, we can use the relation between i and j_0 and j_1 as derived in the decompositions (34) and (37). For rowwise decomposition the relation is $i = j_0$, and for columnwise decomposition the relation is $i = i_0 \cdot j_1 + \Delta i \cdot j_0$.

Block distribution can be considered a special case of block–cyclic distribution, where we know there is only one row. Cyclic distribution can be considered as a special case of block–cyclic distribution, where we know there is only one column. For both distributions only one loop is necessary to enumerate the local elements.

4.5 Communication

To begin with, we discuss the problem of implementing

```
!HPF$  INDEPENDENT
       FORALL(I=0:N-1) A(I) = B(I)
```

for arbitrarily aligned and distributed arrays A and B. Generalization to more general cases will be discussed below. Specifically, we must find a way to send all elements of B to the processors that own the corresponding element of A. The implementation must work and be reasonably efficient for any pair of distributions. In practice this means that a processor must aggregate elements as much as possible before sending them to a destination processor.

Broadly speaking, there are two possible methods to do this. For the sake of clarity, we explicitly show the required owner tests, and do not absorb them in the iterations. Also, we only show message construction ("packing"); message unpacking first checks if any message is to be received from a processor and if so unpacks the message in a similar way it has been packed.

1) Messages can be constructed for all targets separately:

```
for(p=0; p<n_p; p++){
  sendlist[p] = <empty>;
  for(i=0; i<n_i; i++)
    if(owner(B[i]) == me &&
    owner(A[i]) == p)
      /* both conditions will be
         absorbed */
       append(sendlist[p], B[i]);
  if(size(sendlist[p])!=0)
    send(p, sendlist[p]);
}
```

   This corresponds to a post office where the heap of letters is first searched for all letters for Amsterdam, then for all letters for Rotterdam, and so on.

   It is assumed that the **send()** function does not block, so that message construction can be overlapped with message transmission.

2) We can construct messages to all targets simultaneously:

```
for(p=0; p<n_p; p++)
 sendlist[p] = <empty>;
for(i=0; i<n_i; i++)
 if(owner(B[i]) == me)
     /* condition is absorbed */
     append(sendlist[owner(A[i)],B [i);

for(p=0; p<np; p++)
 send(p, sendlist[p]);
```

   This corresponds to a post office where all letters are one by one tossed in bins labeled Amsterdam, Rotterdam, and so on. This is essentially the method as proposed in [6].

The advantages of the first method are that, although the size of each sendlist is not known, we can determine an upper bound to the sum of the sendlist sizes. Therefore, it suffices to simply allocate one send buffer of that size, and construct the individual send buffers in that buffer. In the general case duplicate elements of **B** may have to be sent, so a send buffer must be allocated that can contain the entire iteration space of the FORALL. Fortunately, we can in many cases use a significantly better estimate: if it is known that the index function of **B** is affine, and hence that every element of **B** needs to be sent only once, the send buffer size can be limited to the size of the local iteration space. Also, message construction and transmission can be overlapped: Once the first message has been constructed, it can be handled in parallel by the communications hardware.

The disadvantage of the first method is that in the general case a message is constructed for all other processors. Although in many combinations of distributions most messages will be empty, the message construction is still performed, and is likely to cause considerable overhead. This is especially significant for large processor sets. If more is known about the distribution this disadvantage can be obviated by enumerating a smaller set of processors.

The advantage of the second method is that messages are only constructed for the processors that will receive a message.

The disadvantages are that for each local index of B the

function `owner(A[i])` must be evaluated. In general this requires time consuming calculations for each element. Also, message packing and communication cannot be overlapped. Finally, in the second method we cannot easily estimate the size of the required buffers beforehand. There are a number of methods to circumvent this problem:

1) Do not estimate beforehand, but let the sendlist grow dynamically. This is simple to understand and implement, but it requires a good dynamic memory manager. This may cause problems on some systems.
2) Count the elements beforehand. This can be done using the same set enumeration method as used for send buffer packing. This implies that the iteration space must be enumerated twice.
3) Use a fixed upper bound, and send multiple messages if necessary. This an effective way to solve the problem of buffer size estimation, but message aggregation is not optimal.

Unfortunately, all of these methods to avoid pessimistic buffer size estimates have significant disadvantages.

5 Generalization and Optimization

In Section 4.4 and Section 4.5, we have shown methods to implement simple FORALL statements. We can generalize this in a number of steps with multistatement FORALLs, nonlinear access, multidimensional arrays, general FORALLs, and DO loops. The generalizations we show here merely provide an outline; describing all facets and possible optimizations would be prohibitive in the context of this paper.

5.1 FORALLs with Arbitrary Iteration Ranges

Any FORALL with an arbitrary iteration range can be transformed to a FORALL with a "normalized" iteration range. Thus,

```
!HPF$  INDEPENDENT
       FORALL(I = L : U : S) A(f(I)) =...
```

can be transformed to:

```
!HPF$  INDEPENDENT
       FORALL(I = 0:(U - L + S)/S)
         A(f(S * I + L)) =...
       END FORALL
```

5.2 Multistatement FORALLs

According to the HPF standard, multistatement FORALLs are equivalent to multiple single–statement FORALLs. Hence,

```
!HPF$  INDEPENDENT
       FORALL(I=1:N)
          A(I) = A(I)+1
          B(I) = A(I)
       END FORALL
```

is equivalent to

```
!HPF$  INDEPENDENT
       FORALL(I=1:N) A(I) = A(I)+1
!HPF$  INDEPENDENT
       FORALL(I=1:N) B(I) = A(I)
```

5.3 Nonlinear Access Functions

Nested array references on the rhs can be removed by introducing a temporary that is totally replicated (stored completely on all processors). For example, the expression `A(B(I))` can be simplified by introducing a totally replicated temporary `T`. We must first assign the contents of `B` to `T`:

```
T = B
```

(which is another communication statement). After this we can use the expression `A(T(I))`. Alternatively, one may want to use a inspector/executor run–time library such as PARTI [4], [5] for this.

5.4 Multidimensional Arrays

The techniques shown for one dimensional arrays can be applied to multidimensional arrays. This not only applies to cases where each array subscript expression is independent of the other, such as

```
!HPF$  INDEPENDENT
       FORALL (I=0:N,J=0:M) A(I,J) = 0
```

but also to more complicated index expressions. For example, statements such as

```
!HPF$  INDEPENDENT
       FORALL(I=0:N)
          FORALL(J=0:M-I) A(I,I+J) = 0
       END FORALL
```

can be handled (both iteration ranges can be reduced), because the loop over `J` is nested within the loop over `I`, and therefore `I + J` is linear in `J`. On the other hand, in the statement

```
!HPF$  INDEPENDENT
       FORALL(J=0:M)
       FORALL(I=0:N-J) A(I,I+J) = 0
       END FORALL
```

only the iteration range of `I` can be reduced, the owner test of iterator `J` remains.

Since replicated and collapsed array dimensions do not require an owner test, we do not have to optimize on these dimensions.

5.5 General FORALLs

Until now, we have assumed that the FORALLs are of the form

```
!HPF$  INDEPENDENT
       FORALL (I=0:N) A(da*i+ea)= B(db*i+eb)
```

where `A` and `B` are different arrays, or of the form

```
!HPF$  INDEPENDENT
       FORALL (I=0:N) A(d*i+e) = <expr>
```

where `<expr>` is an arbitrary expression containing only references to locally available array elements. We will now show that any FORALL can be transformed to a sequence of FORALLs that are of this restricted form.

Without the INDEPENDENT directive, all iterations of a FORALL require the value of the rhs expressions as they are at the start of the loop. Fortunately, we can always transform a FORALL to an independent FORALL by introducing temporary variables. For example, the statement

```
FORALL (I=1:N) A(I)=A(I-1)
```

is not independent, since the elements of A change during the execution of the FORALL. However, we can easily transform this to independent FORALL loops by introducing a temporary **T**:

```
!HPF$ INDEPENDENT
      FORALL (I=0:N-1) T(I)=A(I)
!HPF$ INDEPENDENT
      FORALL (I=1:N) A(I)=T(I-1)
```

Any array assignment or independent FORALL without nested array references can be rewritten to a series of communication statements and local computations. For example, the array assignment:

```
A = B + C
```

can be rewritten to

```
B_tmp = B
C_tmp = C
A = B_tmp + C_tmp
```

If we give **B_tmp** and **C_tmp** the same distribution as **A**, the first two statements are communication statements, and the last statements is a local computation.

At first sight it may seem excessive to introduce this much data copying. However, to implement a FORALL correctly it will sometimes be necessary to copy the data anyway. Nevertheless, a good compiler will try to eliminate these copy operations whenever possible with a number of optimizations such as local copy elimination, deviations from the owner computes rule, and transformations such as **A = B + C** ⇒ **A = B; A += C**. Detailed discussion of these transformations is beyond the scope of this paper, however.

5.6 DO Loops

DO loops are not easy to parallelize, because we must assume that the iterations must be executed in the given order. Potentially this requires the update of the involved variables after each iteration, making all parallelization impossible. In cases where DO loops can be executed in parallel, HPF allows an annotation to indicate this (the DO INDEPENDENT clause). In some circumstances it is also possible to detect this automatically [33], [34]. With these premises in mind, the method described in this paper can also handle independent DO loops.

5.7 Eliminating Global-to-Local Calculations

The global-to-local functions of (43) and (47) are relatively computation intensive, since in the general case they contain *mod* and *div* operations. However, for FORALL statements with linear access functions, the global-to-local calculations can be reduced to linear expressions in the inner loop. This property holds for any combination of the local execution set enumeration and local storage schemes in this paper.

To prove this, we write the global to local functions (43) and (47) as

$$g2l_{row}(i) = T_1 \cdot C_1 + T_3$$

$$g2l_{col}(i) = T_2 \cdot C_2 + T_4$$

where

$$T_1 = \left\lfloor \frac{f_{al}(i)}{m \cdot n_p} \right\rfloor$$

$$T_2 = \left\lfloor \frac{f_{al}(i)}{\Delta r_{al} \cdot m \cdot n_p} \right\rfloor$$

$$T_3 = \left\lfloor f_{al}(i) \bmod \frac{m}{a_{al}} \right\rfloor$$

$$T_4 = \left\lfloor f_{al}(i) \bmod \frac{m}{g_{al}} \right\rfloor$$

$$C_1 = \left\lceil \frac{m}{a_{al}} \right\rceil$$

and

$$C_1 = \left\lceil \frac{m}{g_{al}} \right\rceil$$

T_1 and T_2 can be interpreted as row numbers, while T_3 and T_4 can be interpreted as column numbers.

We first treat rowwise enumeration in combination with rowwise storage or columnwise storage.

5.7.1 Rowwise Enumeration and Row or Column Storage

Looking at (36c) we see that for the enumeration of j_0 holds:

$$m \cdot k \le a_{co} \cdot j_0 + b_{co} < m \cdot (k+1)$$

for $k \in \mathbb{Z}$. This implies that for $f_{co}(j_0) = a_{co} \cdot j_0 + b_{co}$, it holds that $f_{co}(j_0)\ div\ m = k$. We can rewrite the range of variable

$$j_0 \in [j_{0,min}, j_{0,max}\rangle$$

to

$$j_{0,min} + u$$

where

$$u \in [0, j_{0,max} - j_{0,min}\rangle$$

Using the property that $(x + \Delta x) \bmod m = x \bmod m + \Delta x$, if $(x + \Delta x)\ div\ m = k$ and $x\ div\ m = k$, it follows that T_3 can be written as

$$T_3 = \left\lfloor \frac{f_{co}(j_{0,min}) \bmod m + a_{co} \cdot u}{a_{al}} \right\rfloor$$

or:

$$T_3 = \left\lfloor \frac{f_{co}(j_{0,min}) \bmod m}{a_{al}} \right\rfloor + a_{ix} \cdot u$$

For columnwise storage the reasoning is completely analogous for T_4, yielding a multiplier of a_{co}/g_{al} instead of a_{ix} as in the case of rowwise storage.

From $\lfloor f_{co}(j_0)/m \rfloor = k$, it also follows that both terms T_1 and T_2 are constant for any value of j_0 within that row. Hence, we may take $j_{0,min}$ to calculate the terms. Hence, it follows for rowwise enumeration and rowwise storage that

$$g2l^{erow}_{srow}(f_{ix}(j_0)) = g2l_{row}(f_{ix}(j_{0,min})) + a_{ix} \cdot u$$

and for rowwise enumeration and columnwise storage:

$$g2l^{erow}_{scol}(f_{ix}(j_0)) = g2l_{col}(f_{ix}(j_{0,min})) + \frac{a_{co}}{g_{al}} \cdot u$$

where $g2l^{erow}_{srow}$ means rowwise enumeration combined with rowwise storage and $g2l^{erow}_{scol}$ means rowwise enumeration and columnwise storage.

Hence, only for the starting point of the inner loop of a local enumeration sequence, a more complicated global-to-local function needs to be calculated. Then the remainder of that sequence can be enumerated in a linear fashion.

5.7.2 Columnwise Enumeration and Row or Column Storage

For columnwise enumeration and columnwise storage, we rewrite the global index generation function $i_0 \cdot j_1 + \Delta i_{co} \cdot j_0$ from (39i) as

$$i_0 \cdot j_1 + \Delta i_{co} \cdot j_0 = i_0 \cdot j_1 + \Delta i_{co} \cdot (j_{0,min} + u) = \lambda + \Delta i_{co} \cdot u$$

where $\lambda = i_0 \cdot j_1 + \Delta i_{co} \cdot j_{0,min}$ denotes the starting point of the enumeration sequence. $T_2 = \lfloor f_{al}(i)/\Delta r_{al} \cdot m \cdot n_p \rfloor$ can be rewritten as $T_2 = \lfloor f_{al}(i)/\Delta r_{al} \cdot a_{al} \rfloor$. Substituting $\lambda + \Delta i_{co} \cdot u$ in T_2 yields

$$\left\lfloor \frac{f_{co}(\lambda + \Delta i_{co} \cdot u)}{a_{al} \cdot \Delta i_{al}} \right\rfloor = \left\lfloor \frac{f_{co}(\lambda)}{a_{al} \cdot \Delta i_{al}} \right\rfloor + a_{ix} \cdot \frac{\Delta i_{co}}{\Delta i_{al}} \cdot u$$

because the term $a_{ix} \cdot (\Delta i_{co}/\Delta i_{al})$ always yields an integer.

From (14) and (37) it follows that $\Delta i_{co} = m \cdot n_p/g_{co}$. Hence $f_{co}(\lambda + \Delta i_{co} \cdot u) = f_{co}(\lambda) + a_{co}\ (m \cdot n_p/g_{co}) \cdot u$. Because the term $a_{co} \cdot (m \cdot n_p/g_{co})$ is always a multiple of m, the term T_4 will always yield a constant (as will T_3 in the other case). As a result, the global-to-local function can be written as

$$g2l^{ecol}_{scol}(f_{ix}(i_0 \cdot j_1 + \Delta i_{co} \cdot j_0)) = g2l_{col}(f_{ix}(\lambda)) + a_{ix} \cdot \frac{\Delta i_{co}}{\Delta i_{al}} \cdot C_2 \cdot u$$

The derivation of columnwise enumeration with rowwise storage is analogous, yielding

$$g2l^{ecol}_{srow}(f_{ix}(i_0 \cdot j_1 + \Delta i_{co} \cdot j_0)) = g2l_{row}(f_{ix}(\lambda)) + a_{co} \cdot \frac{\Delta i_{co}}{m \cdot n_p} \cdot C_1 \cdot u$$

Hence both global-to-local functions are linear in u.

5.8 Other Optimizations

As remarked above, only the starting points of each row or column enumeration sequence require a full global-to-local function calculation. For long inner loops, this will result in little overhead, as will be shown in Section 6. For short inner loops which are to be executed more than once, it might be profitable to store the starting points in a table to reduce overhead.

6 Results

Evaluating the performance of HPF implementations in general is difficult, because very few HPF programs have yet been written. In particular, it is not known which kind of alignments and distributions are likely to occur in practice. For the moment we must therefore assume that all alignments and distributions are equally likely to occur, and we must try to make all possible cases efficient. In this paper we have therefore restricted ourselves to those cases that can be optimized without further knowledge of the specific structure of a program. In this section, we evaluate the presented method on index and communication overhead and on storage overhead.

6.1 Index Generation Overhead

We have made a comparison between our local enumeration and storage methods and the pattern cyclic method of Chatterjee et al. [6] with respect to the overhead generated to access local elements. The results that are shown here are the execution times of hand-written code that mimics the behavior of compiler-generated code. We use this code instead of real code because it makes it simpler to evaluate variants on the implementation, to simulate foreign implementation schemes, and to ensure the accuracy of the measurements. The code simulates the execution of the statement

```
A(a*I+b) = A(a*I+b) + 1
```

using a one dimensional distributed array of size 40,000. We assume that the address calculation for **A(a*I+b)** is done only once, so that in effect this statement is an *increment* operation. This statement was chosen because it makes it simple to verify that all elements of the array have been enumerated exactly once, independent of the order in which they have been enumerated.

We assume that both rowwise and columnwise enumeration show the same execution times. In theory the method we simulate, columnwise enumeration, should perform slightly worse than rowwise enumeration, because it requires an additional gcd calculation. In practice the results for both methods showed little difference. We further assume that pattern cyclic enumeration method adds an extra array dereference for the look-up table as compared to the methods described in this paper. This is more efficient than the original implementation of Chatterjee et al., and corresponds to a more recent implementation described by Kennedy et al. [23] (method (c) on page 13 in their paper). We ignore the time that is required for table construction.

For all methods three levels of optimization were assumed. In the slowest variant the global-to-local calculation of (43) or (47) is done for every element that is enumerated. In the next variant only one global-to-local calculation is done for each row, column or pattern cycle, as discussed in Section 5.7. In the last variant, all global-to-local calculations have been eliminated by using a table of precomputed values, as discussed in Section 5.8. We assume that the two optimizations can also be used in the pattern-cyclic method. Finally, the sequential version of the statement has also been included to serve as a comparison.

The experiment was done on a number of different processors: a microSPARC II and other SPARC variants, a Hewlett Packard PA-RISC 1.1, and a MIPS R4400. For the PA-RISC machine the native Hewlett Packard C compiler was used with option **-O**, in all other cases the GNU C compiler version 2.7.2 was used with option **-O2**. Essentially, all experiments yielded similar results. A typical re-

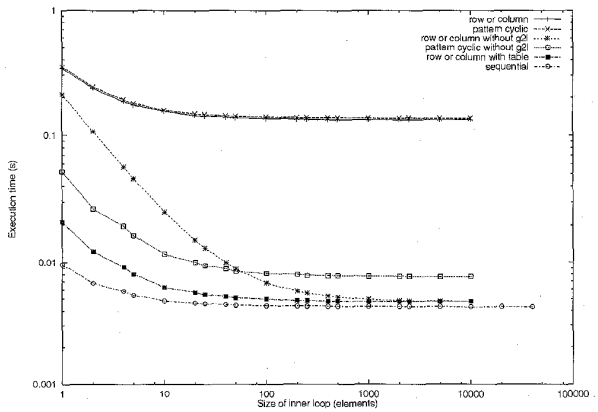

Fig. 8. Execution times for simulated versions of a number of enumeration methods. "row or column" stands for the enumeration method described in this paper. The variants "without g2l" assume that only one global-to-local calculation is necessary for each row, column, or pattern cycle, as discussed in Section 5.7. In the variant "with table" the starting points of the rows or columns are precomputed, see Section 5.8.

sult is depicted in Fig. 8 (log-log scale); this is for a micro-SPARC II on 85 MHz.

Several observations can be made from this figure:

- For linear access sequences and large inner loops, the pattern-cyclic method is about 50% slower than the method described in this paper. Measurements on other machines yielded similar results (50% to 100% slower).
- On short inner loops (<50) all methods give an increased overhead.
- For inner loops larger than 100, the loop overhead is comparable to pure sequential execution.
- Full global-to-local conversions for arbitrary array accesses are in all methods expensive (up to about 40 times sequential execution).

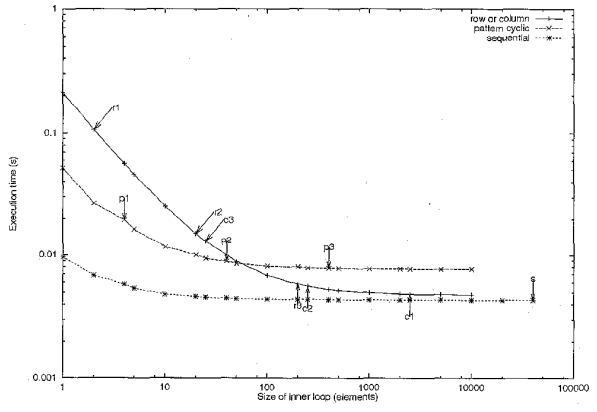

Fig. 9. The execution time of the various enumeration methods of Fig. 8 for a number of array distributions. Point s shows the execution time for sequential execution. Points $r1$, $r2$, $r3$ show execution times for rowwise enumeration; points $c1$, $c2$, $c3$ show execution times for columnwise enumeration; and points $p1$, $p2$, $p3$ show execution times for pattern cyclic enumeration. The points $r1$, $c1$, $p1$ are for an array distribution with $m = 4$, $r2$, $c2$, $p2$ are for an array distribution with $m = 40$, and $r3$, $c3$, $p3$ are for an array distribution with $m = 400$.

However, there is more to be said. The size of the inner loop cannot be freely chosen. In both the pattern cyclic and the rowwise enumeration method, the size of the inner loop is bounded by m. To illustrate this, consider the example alignment depicted in Fig. 2. We have taken three different values of m; $m = 4$, $m = 40$, and $m = 400$ and marked the outcome of each enumeration method on the curves with only global-to-local calculations in the outer loop. This is shown in Fig. 9. As is obvious from the graph, no method is best for all distributions we considered: For $m = 4$ and $m = 40$, columnwise enumeration is best, and for $m = 400$ rowwise enumeration is best. In general, for small values of m, columnwise iteration will generally produce longer vectors and hence less overhead. For larger values of m, rowwise and pattern-cyclic enumeration become more efficient. Pattern-cyclic enumeration can often span larger vectors than rowwise enumeration, but incurs more overhead.

6.2 Communication Overhead

The performance of communication is dependent on two main factors: the time to pack and unpack messages, and the overhead generated by the message passing run-time system and the underlying communications hardware. To judge our method of communication, only the first factor needs to be analyzed; the second factor is system dependent. To evaluate the overhead of message packing and unpacking, we must eliminate other factors that might influence the results. In particular, we do not want to measure the time required by the communication system. For these reasons we have chosen the statement

```
!HPF$  INDEPENDENT
       FORALL(I = 1:N) B(I) = A(N-I+1)
```

For an even number of processors this statement requires communication of all elements of the array.

The results of the measurements are shown in Fig. 10. Three distributions were chosen, block, cyclic, and cyclic(5). Sequential execution time was also measured. The measurements have been made comparable by executing the FORALL statement repeatedly, so that the same number of elements is processed in all cases. The shown execution times are the sums of the individual times per processor.

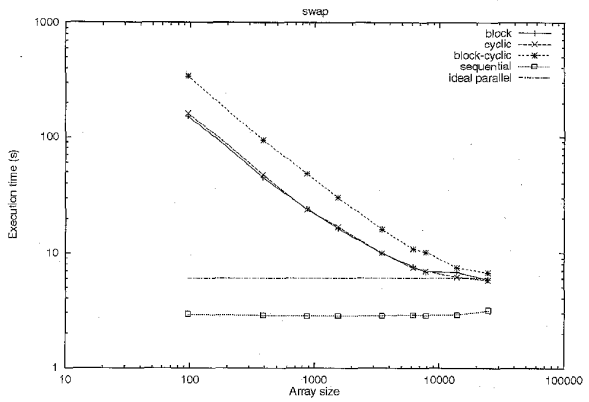

Fig. 10. The total time required for the execution of FORALL (I = 1 : N) A(I) = B(N – I + 1) on a cluster of four Sun workstations. The time spent in the communication system is excluded.

In the communication scheme we use (and in other general communication schemes), data that is communicated is copied twice: from the array to be sent into the send buffer, and from the receive buffer into the receiving array. Since in this example copying time significant, the parallel version of the code will even in the best case require twice the time of sequential execution. In Fig. 10 this is indicated by the line "ideal parallel."

The figure clearly shows that, although the message construction overhead can be large for small array sizes, it rapidly decreases for larger (and probably more realistic) array sizes. In fact, for the largest data points it approaches the 100% overhead we predicted above. Note that the sequential execution time increases slightly as the inner loops get larger. This effect is caused by system–specific effects, such as paging.

Inspection of the generated assembly code reveals that the main differences in execution times between the sequential and the parallel versions of the code are due to specific knowledge about loop variables. Further optimization of the parallel code is possible by exploiting specific knowledge of loop variables and in–lining of function calls.

However, the actual time spent in the message passing system itself will in general be larger than the packing and unpacking time. Since packing and message transmission are overlapped, it might not be useful to further optimize packing and unpacking, depending on the characteristics of the message communication system. The graphs also show that the differences between block, cyclic, or block–cyclic distributions are minor and do not result in large differences in the packing and unpacking time.

6.3 Storage Overhead

To evaluate the efficiency of rowwise and columnwise storage, we have tabulated the overhead of the rowwise and columnwise storage schemes, see Fig. 11. In principle all of the parameters a, b, m, n_p, and n_i influence the storage overhead. However, the influence of b can be neglected. For n_p we have chosen an arbitrary number, 16, but for other values the overhead plot remains essentially the same. We have chosen n_i so that for the maximal value of m that we plot we have block distribution. Thus $n_i = m \cdot n_p = 15 \cdot 16 = 240$. For larger n_i the overhead will be less. The remaining parameters, a and m, are plotted against each other.

The overhead is calculated as

$$\frac{n_p \cdot n - n_i}{n_i} \tag{52}$$

where n is the required storage size of (41) or (45).

For example, for rowwise storage the overhead of a number of (a, m) pairs is:

a	m	n_r	u_0^{ub}	n	overhead (%)
1	1	15	1	15	0
1	10	2	10	20	33
1	15	1	15	15	0
5	1	75	1	75	400
8	15	8	2	16	7
15	15	15	1	15	0

and for columnwise storage the overhead of these (a, m) pairs is:

a	m	n_r	u_1^{ub}	u_2^{ub}	n	overhead (%)
1	1	15	1	15	15	0
1	10	2	10	2	20	33
1	15	1	15	1	15	0
5	1	75	1	15	15	0
8	15	8	2	8	16	7
15	15	15	1	15	15	0

In the example of Fig. 11, for 23 (a, m) pairs rowwise storage is more efficient, for 100 pairs columnwise storage is more efficient, and for 102 pairs both are equally efficient.

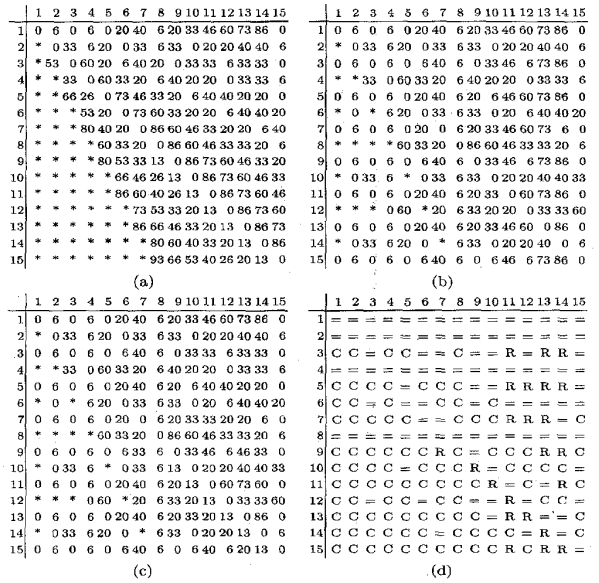

Fig.11. A table of the overhead of rowwise (a), columnwise (b), and hybrid (c) storage. In all plots m increases along the horizontal axis from 1 to 15, and a increases along the vertical axis from 1 to 15. All overheads are given as a percentage of the array size. Overheads of 100% or higher are indicated with a star. Plot (d) shows a map of the choices that are made for hybrid storage: "C" means columnwise is best, "R" means rowwise is best, and "=" means both have equal overhead.

7 Conclusions and Further Research

In this paper we have presented an implementation framework for access to and storage of distributed arrays in data parallel languages such as HPF.

Two methods for local set enumeration and local array storage have been derived. It is shown that local set enumeration and local storage methods can be freely combined. Hence the optimal storage strategy can be chosen, without having to rearrange local memory if the access method is changed during the execution of the program.

It has been shown that implementation of the methods covers the complete spectrum of HPF distribution functions, is relatively simple, and requires no tables. For linear iteration sequences access to local memory is shown to be fast and to approach sequential execution times for sufficiently large inner loops. The presented framework can be enhanced with further optimizations by exploiting specific characteristics of classes of HPF programs. Especially, local loop overhead for smaller inner loops can be reduced by making use of specific forms of alignment and array access functions.

APPENDIX

The following examples show the code generated for a few small HPF programs. The programs have been chosen to show the translation of as many different features as possible.

The code was derived from actual output of the PREPARE compiler [2], [3], but it has been edited extensively to make it more compact and readable. For the sake of compactness many declarations have been omitted. The function **sendbuf()** is assumed to return immediately after the message has been queued for transmission; the function **waitForAll()** is used to wait until all transmissions have been completed, and it is safe to free send buffers. The first program shows how local computations are handled:

```
      PROGRAM local

      INTEGER, PARAMETER :: N=10000
      INTEGER, DIMENSION(N,N) :: A, B, C
      INTEGER :: I, J

!HPF$ PROCESSORS PA(2,2)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO PA :: A, B
!HPF$ DISTRIBUTE (CYCLIC(5),CYCLIC(5))
      ONTO PA :: C

      FORALL (I = 1:N, J = 1:N)
         A(I,J) = N * I + J
      END FORALL
      FORALL (I = 1:N, J = 1:N)
         C(I,J) = N * I + J
      END FORALL
      A = B
      END PROGRAM
```

This is translated to:

```
calc_blus(Map_A, 0, ME(Map_A, 0), 1,
   10000, 1, 1, 0, &L1, &U1, &S1);
calc_blus(Map_A, 1, ME(Map_A, 1), 1,
   10000, 1, 1, 0,&L2, &U2, &S2);
lo0 = G2L(Map_A, 0, L1);
ls0 = G2L(Map_A, 0, L1+S1)-lo1;
lo1 = G2L(Map_A, 1, L2);
ls1 = G2L(Map_A, 1, L2+S2)-lo0;
J = L1;
ld0 = lo0;
for (cnt1 = (U1-L1+S1)/S1; cnt1 > 0;
   cnt1--){
   I = L2;
   ld1 = lo1;
   for (cnt2 = (U2-L2+S2)/S2; cnt2 > 0;
      cnt2--){
      *(Aloc+ld0+ld1) = 10000 * I + J;
      I += S2;
      ld1 += ls1;
   }
   J += S1;
   ld0 += ls0;
}
```

```
bclus(ES1, Map_C, 0, ME(Map_C, 0),
   1, 10000, 1, 1, 0, &L3, &U3, &S3);
T1 = L3;
for (cnt3 = (U3-L3+S3)/S3; cnt3 > 0; cnt3--){
   calc_bclus1(ES1, Map_C, 0, T1, &L4, &U4,
      &S4);
   calc_bclus(ES2, Map_C, 1, ME(Map_C, 1), 1,
      10000, 1, 1, 0, &L5, &U5, &S5);
   T2 = L5;
   for (cnt4 = (U5-L5+S5)/S5;
      cnt4 > 0; cnt4-){
      calc_bclus1(ES2, Map_C, 1,
         T2, L6, U6, S6);
      lo0 = G2L(Map_C, 0, L4);
      ls0 = G2L(Map_C, 0, L4+S4)-lo0;
      lo1 = G2L(Map_C, 1, L6);
      ls1 = G2L(Map_C, 1, L6+S6)-lo1;
      J = L4;
      ld0 = lo0;
      for (cnt5 = (U4-L4+S4)/S4; cnt5 > 0;
         cnt5-){
         I = L6;
         ld1 = lo1;
         for (cnt6 = (U6-L6+S6)/S6;
            cnt6 > 0; cnt6-){
            *(Cloc+ld0+ld1) = 10000 * I + J;
            I += S6;
            ld1 += ls1;
         }
         J += S4;
         ld0 += ls0;
      }
      T2 += S5;
   }
   T1 += S3;
}
calc_blus(Map_A, 0, ME(Map_A, 0), 0,
   9999, 1, 1, 1, L7, U7, S7);
calc_blus(Map_A, 1, ME(Map_A, 1), 0,
   9999, 1, 1, 1, L8, U8, S8);
lo0 = G2L(Map_A, 0, L7+1);
ls0 = G2L(Map_A, 0, L7+S7+1)-lo0;
lo1 = G2L(Map_A, 1, L8+1);
ls1 = G2L(Map_A, 1, L8+S8+1)-lo1;
ld0 = lo0;
for (cnt7 = (U7-L7+S7)/S7; cnt7 > 0; cnt7-){
   ld1 = lo1;
   for (cnt8 = (U8-L8+S8)/S8; cnt8 > 0;
      cnt8-){
      *(Aloc+ld0+ld1) = *(Bloc+ld0+ld1);
         ld1 += ls1;
   }
   ld0 += ls0;
}
```

The second program shows how the communication between processors is handled. For simplicity the shown code also constructs and sends messages to the local processor. The real compiler prevents this.

```
      PROGRAM comm

      INTEGER, PARAMETER :: N=10000
      INTEGER, DIMENSION(N) :: A,B
      INTEGER :: I
!HPF$ PROCESSORS PA(4)
!HPF$ DISTRIBUTE (BLOCK) ONTO PA :: A, B
      FORALL (I = 1:N)
         A(I) = B(N-I+1)
      END FORALL
      END PROGRAM
```

This is translated to:

```
sbuf = (int*) malloc(LSIZ(MapB));
sbufp = sbuf;
for (to[0] = 0; to[0] < 4; to[0]++){
   start = sbufp;
   calc_blus(MapB, 0, ME(MapB, 0), 1,
      10000, 1, -1, 10001,L1, U1, S1);
   calc_blus(Map_A, 0, to[0], L1, U1, S1,
      1, 0,&L2, &U2, &S2);
   lo = G2L(MapB, 0, 10001 - L2);
   ls = G2L(MapB, 0, 10001 - (L2+S2))-lo;
   ld = lo;
   for (cnt1 = (U2-L2+S2)/S2; cnt1 > 0;
      cnt1-){
      *(sbufp++) = *(B+ld);
      ld += ls;
   }
   if (sbufp > start)
      sendbuf(Map_A, to[0], (void*)start,
         sbufp - start);
}
rbuf = malloc(LSIZ(MapB));
for (from[0] = 0; from[0] < 4;
   from[0]++){
   empty = TRUE;
   calc_blus(MapB, 0, from[0], 1, 10000,
      1, -1, 10001,&L3, &U3, &S3);
   calc_blus(Map_A, 0, ME(Map_A, 0),
      L3, U3, S3, 1, 0, &L4, &U4, &S4);
   for (cnt1 = (U4-L4+S4)/S4; cnt1 > 0;
      cnt1-){
      empty = FALSE;
   break;
   }
   if (!empty){
      recvbuf(MapB, from[0], (void*)rbuf);
      rbufp = rbuf;
      lo = G2L(Map_A, 0, L4);
      ls = G2L(Map_A, 0, L4+S4)-lo;
      ld = lo;
      for (cnt2 = (U4-L4+S4)/S4; cnt2 > 0;
         cnt2-){
         *(A+ld) = *(rbufp++);
         ld += ls;
      }
   }
}
free(rbuf);
waitForAll();
free(sbuf);
```

ACKNOWLEDGMENTS

The authors wish to thank the referees for their valuable comments which have improved this paper substantially.

This research is sponsored by SPIN, and by Esprit through the project PREPARE.

REFERENCES

[1] "High Performance Fortran Forum," *High Performance Fortran Language Specification*, ver. 1.1, Nov. 1994.

[2] A.H. Veen and M. de Lange, "Overview of the PREPARE Project," *Proc. Fourth Int'l Workshop Compilers for Parallel Computers*, H.J. Sips, ed., pp. 365-371, Dec. 1993.

[3] F. Andre, P. Brezany, O. Chéron, W. Denissen, J.L. Pazat, and K. Sanjari, "A New Compiler Technology for Handling HPF Data Parallel Constructs," *Proc. Third Workshop on Languages, Compilers, and Run-time Systems*, B.K. Szymanski and B. Sinharoy, eds., pp. 279-282, 1995.

[4] A. Choudhary R. Ponnusamy, and J. Saltz, "Runtime-Compilation for Data Partitioning and Communication Schedule Reuse," *Proc. Supercomputing 1993*, pp. 361-370, Nov. 1993. Also available as University of Maryland Technical Report CS-TR-3055 UMIACS-TR-93-32.

[5] P. Brezany, O. Chéron, K. Sanjari, and E. van Konijnenburg, "Processing Irregular Codes Containing Arrays with Multi-Dimensional Distributions by the PREPARE HPF Compiler," *HPCN Europe '95*, pp. 526-531. Springer-Verlag, 1995.

[6] S. Chatterjee, J.R. Gilbert, F.J.E. Long, R. Schreiber, and S.-H. Teng, "Generating Local Addresses and Communication Sets for Data-Parallel Programs," *J. Parallel and Distributed Computing*, vol. 26, no. 1, pp. 72-84, April 1995. First presented at PPoPP'93.

[7] D. Callahan and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors," *J. Supercomputing*, vol. 2, no. 2, pp. 151-169, Oct. 1988.

[8] M. Gerndt, "Array Distribution in SUPERB," *Proc. Third Int'l Conference on Supercomputing*, Crete, Greece, June 1989.

[9] A. Rogers and K. Pingali, "Process Decomposition Through Locality of Reference," *Proc. ACM SIGPLAN Int'l Conf. Program Language Design and Implementation*, June 1989.

[10] E.M. Paalvast, A.J. van Gemund, and H.J. Sips, "A Method for Parallel Program Generation with an Application to the Booster Language," *Proc. 1990 Int'l Conf. Supercomputing*, pp. 457-469, June 11-15 1990.

[11] C. Koelbel and P. Mehrotra, "Compiling Global Name-Space Parallel Loops for Distributed Execution," *IEEE Trans. Parallel and Distributed Systems*, vol. 2, no. 4, pp. 440-451, Oct. 1991.

[12] M. Le Fur, J.-L Pazat, and F. Andre, "Static Domain Analysis for Compiling Commutative Loop Nests," Publication interne 757, IRISA, Campus Universitaire de Beaulieu - 35042 Rennes Cedex, France, Sept 1993. URL: ftp//irisa.irisa.fr/techreports/1993/PI-757.ps.Z.

[13] C.-W. Tseng, "An Optimizing Fortran D Compiler for MIMD Distributed Memory Machines," PhD thesis, Rice Univ., Houston, Tex., Jan. 1993.

[14] J.M. Stichnoth, "Efficient Compilation of Array Statements for Private Memory Multicomputers," Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon Univ., Feb. 1993.

[15] J.M. Stichnoth, D. O'Hallaron, and T.R. Gross, "Generating Communication for Array Statements: Design, Implementation and Evaluation," *J. Parallel and Distributed Computing*, vol. 21, pp. 150-159, 1994.

[16] S.K.S. Gupta, S.D. Kaushik, S. Mufti, S. Sharma, C.-H Huang, and P. Sadayappan, "On Compiling Array Expressions for Efficient Execution on Distributed Memory Machines," *Proc. Int'l Conf. Parallel Processing*, vol. II, pp. 301-305, Aug. 1993.

[17] S.K.S. Gupta, S.D. Kaushik, C.-H. Huang, and P. Sadayappan, "On Compiling Array Expressions for Efficient Execution on Distributed-Memory Machines," Technical Report OSU-CISRC-4/94-TR19, Ohio State Univ., Columbus, OH 43210, 1994. URL: ftp://archive.cis.ohio-state.edu/pub/tech-report/1994/TR19.ps.gz.

[18] S.D. Kaushik, C.-H. Huang, and P. Sadayappan, "Incremental Generation of Index Sets for Array Statement Execution on Distributed-Memory Machines," *Proc. Seventh Ann. Workshop Languages and Compilers for Parallel Computing*, pp. 17.1-17.18, Cornell Univ., Aug. 1994. Also published in LNCS 892, pp 251-265, 1995, Springer Verlag,.

[19] S. Benkner, P. Brezany, and H. Zima, "Processing Array Statements and Procedure Interfaces in the PREPARE High Performance Fortran Compiler," *Compiler Construction, Proc. Fifth Int'l Conf*, P.A. Fritzon, ed., vol. 786 of *LNCS*. Springer-Verlag, pp. 324-338, Apr. 1994.

[20] C. Ancourt, F. Irigoin, F. Coelho, and R. Keryell, "A Linear Algebra Framework for Static HPF Code Distribution," Technical Report A-278-CRI, Ecole des Mines, Paris, Nov. 1995. An earlier version was presented at the *Fourth Int'l Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, pp. 117-132, Dec. 1993.

[21] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi, "Compilation Techniques for Block-Cyclic Distributions," *Proc. Intl. Conf. Supercomputing*, pp. 392-403, July 1994.

[22] A. Thirumalai and J. Ramanujam, "Fast Address Sequence Generation for Data-Parallel Programs Using Integer Lattices," *Proc. Eighth Int'l Workshop Languages and Compilers for Parallel Computing*, pp. 13.1-13.19, 1995.

[23] K. Kennedy, N. Nedeljkovic', and A. Sethi, "A Linear-Time Algorithm for Computing the Memory Access Sequence in Data-Parallel Programs," *Proc. Fifth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, 1995.

[24] S.D. Kaushik, C.-H. Huang, and P. Sadayappan, "Compiling Array Statements for Efficient Execution on Distributed Memory Machines: Two-Level Mappings," *Proc. Eighth Ann. Workshop Languages and Compilers for Parallel Computing*. pp. 14.1-14.15, Ohio State Univ., Aug. 1995.

[25] K. Kennedy, N. Nedeljkovic', and A. Sethi, "A Linear Time Algorithm for Computing the Memory Access Sequence in Data-Parallel Programs," *Proc. Symp. Principles and Practice of Parallel Programming*, pp. 102-111, ACM, 1995. Proceedings also published as Sigplan Notices, vol. 30, no. 8.

[26] K. Kennedy, N. Nedeljkovic', and A. Sethi, "Efficient Address Generation for Block-Cyclic Distributions," *Proc. Intl. Conf. Supercomputing*, pp. 180-184, June 1995.

[27] C. Koelbel, "Compile-Time Generation of Regular Communication Patterns," *Proc. Supercomputing 1991*, pp. 101-110, ACM, 1991.

[28] E.M. Paalvast, H.J. Sips, and L.C. Breebaart, "Booster: A High-Level Language for Portable Parallel Algorithms," *Applied Numerical Mathematics*, vol. 8, no. 6, pp. 177-192, 1991.

[29] E.M. Paalvast, "Programming for Parallelism and Compiling for Efficiency," PhD thesis, Delft Univ. of Technology, June 1992.

[30] E.M. Paalvast, H.J. Sips, and A.J. van Gemund, "Automatic Parallel Program Generation and Optimization from Data Decompositions," *Proc. 1991 Int'l Conf. Parallel Processing*, pp. II 124-131, Aug. 1991.

[31] Y. Mahéo and J.-L. Pazat, "Distributed Array Management for HPF Compilers," Publication interne 787, IRISA, Campus Universitaire de Beaulieu-35042 Rennes Cedex, France, Dec. 1993. URL: ftp://irisa.irisa.fr/techreports/1993/PI-787.ps.Z.

[32] K.H. Rosen, *Elementary Number Theory And Its Applications*. Addison Wesley, 1984.

[33] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*. ACM Press, Frontier Series, 1991.

[34] M. Wolfe, *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.

Kees van Reeuwijk received his MSc degree in 1986 in electrical engineering and his PhD degree in 1991 from Delft University of Technology, Delft, The Netherlands. Currently he is a post doctoral researcher in the research group Computational Physics at Delft University of Technology. His research interests include compiler construction, programming language design, and silicon compilation

Will Denissen received his MS degree in electrical engineering from the Technical University of Eindhoven, The Netherlands, in 1987. He is currently employed by the Dutch Organisation for Applied Scientific Research (TNO). His research interests include parallel and distributed programming, parallel arhitectures, performance, and scalability compilation.

Henk J. Sips received his MSc degree in 1976 in electrical engineering and his PhD degree in 1984 from Delft University of Technology, Delft, The Netherlands. Currently he is an associate professor in computational physics the Delft University of Technology, and a professor in computer science at the University of Amsterdam. His research interests include computer architecture, parallel programming, parallel algorithms, and distributed systems.

Edwin M.R.M. Paalvast obtained his MS degree in computer science and mathematics at Leiden University in the area of parallel and distributed computing and operations research, respectively. Since 1987 he worked for the Dutch Organisation for Applied Scientific Research (TNO). In 1992, he finished his PhD at Delft University of Technology on the subject of compilers and languages for parallel computers. Since 1995, he has worked as a senior consultant for Bakkenist Management Consultants.