Syracuse University
SURFACE

Northeast Parallel Architecture Center

L.C. Smith College of Engineering and Computer Science

1994

Runtime Array Redistribution in HPF Programs

Rajeev Thakur
Syracuse University, Northeast Parallel Architectures Center, thakur@npac.syr.edu

Alok Choudhary
Syracuse University, Northeast Parallel Architectures Center

Geoffrey C. Fox
Syracuse University, Northeast Parallel Architectures Center

Follow this and additional works at: http://surface.syr.edu/npac

Part of the Computer Sciences Commons

Recommended Citation
Thakur, Rajeev; Choudhary, Alok; and Fox, Geoffrey C., "Runtime Array Redistribution in HPF Programs" (1994). *Northeast Parallel Architecture Center.* Paper 12.
http://surface.syr.edu/npac/12

This Working Paper is brought to you for free and open access by the L.C. Smith College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Northeast Parallel Architecture Center by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

Runtime Array Redistribution in HPF Programs

Rajeev Thakur* **Alok Choudhary*** **Geoffrey Fox** †
Northeast Parallel Architectures Center
111 College Place, Rm 3-228
Syracuse University, Syracuse NY 13244
thakur, choudhar, gcf @npac.syr.edu

Abstract

This paper describes efficient algorithms for runtime array redistribution in HPF programs. We consider block(m) to cyclic, cyclic to block(m) and the general cyclic(x) to cyclic(y) type redistributions. We initially describe algorithms for one-dimensional arrays and then extend the methodology to multidimensional arrays. The algorithms are practical enough to be easily implemented in the runtime library of an HPF compiler and can also be directly used in application programs requiring redistribution. Performance results on the Intel Paragon are discussed.

1 Introduction

High Performance Fortran (HPF) is a language designed to support portable high performance programming on a wide variety of machines, including massively parallel SIMD and MIMD systems and vector processors [4]. It supports the data parallel programming model and uses Fortran 90 as the base language. HPF extends Fortran 90 in several areas by providing data distribution features, data parallel execution features, extended intrinsic functions and standard library, and EXTRINSIC procedures. At Syracuse University, we are developing an HPF compiler which translates HPF to Fortran 77 plus message passing node programs for distributed memory computers [1].

HPF provides directives (ALIGN and DISTRIBUTE) which specify how arrays should be partitioned among the processors of a distributed memory computer. Arrays are first aligned to a *template* or index space. The DISTRIBUTE directive specifies how the template is to be distributed among the processors. In HPF, an array (or template) can be distributed as block(m) or cyclic(m) [4]. Though the distribution of an array is specified at compile time, there are a number of reasons for which it may be necessary to redistribute an array during the execution of the program.

- HPF has directives REDISTRIBUTE and REALIGN which require arrays to be remapped.
- It is not practical to write intrinsic and runtime libraries for all possible distributions. Libraries can be written for a few common distributions and for any other distribution it is necessary to redistribute before calling the subroutine. This approach also eliminates the need for complex inter-procedural analysis. After returning from the subroutine it is necessary to redistribute back to the original distribution.
- In many applications such as 2D FFT and the ADI method for solving multidimensional PDEs, dynamic redistribution can result in significant performance improvement [2].

Array redistribution is an expensive operation as it involves data communication as well as computation of destination processors and addresses. It is necessary to do redistribution efficiently, otherwise the overhead of redistribution itself will offset the benefit of using a different distribution. In this paper, we present algorithms for redistributing arrays efficiently. We consider block(m) to cyclic, cyclic to block(m) and the general cyclic(x) to cyclic(y) type redistributions. We consider the redistribution of one-dimensional arrays in Sections 3, 4 and 5 and then extend the methodology to multidimensional arrays in Section 6. We have concentrated on making the algorithms practical enough to be easily implemented in the runtime library of an HPF compiler. They can also be directly used in application programs requiring redistribution. The algorithms make good use of cache and optimize communication. We do not make any special assumptions about the machine architecture or the message passing paradigm provided on the machine.

2 Notations and Definitions

The notations used in this paper are given in Figure 1. We assume that all arrays are indexed starting

*Also with the Dept. of Electrical and Computer Eng., Syracuse University

†Also with the Dept. of Computer and Information Science, Syracuse University

N	global array size
P	number of processors
p_i	logical processor i
p	logical number of the processor executing the program
p_s	source processor
p_d	destination processor
L	local array size
m	block size

Figure 1: Notations used in this paper

from 1, while processors are numbered starting from 0 and that arrays are identically aligned to the template. The algorithms can be easily extended for the general case. Also, the algorithms do not specify how to perform data communication because the best communication algorithms are often machine dependent. These communication algorithms are described in detail in [9, 7]. We do assume that all the data to be sent from any processor i to processor j has to be collected in a *packet* in processor i and sent in one communication operation, so as to minimize the communication startup cost. In the rest of this paper, any division involving integers should be considered as integer division unless specified otherwise.

The block(m) and cyclic(m) distributions in HPF are defined as follows. Consider an array of size N distributed over P processors. Let us define the ceiling division function $CD(j,k) = (j+k-1)/k$ and the ceiling remainder function $CR(j,k) = j - k \times CD(j,k)$. Then block($m$) distribution means that index j of the array is mapped to logical processor number $CD(j,m) - 1$. Note that for a valid block(m) distribution, $m \times P \geq N$ must be true. Block by definition means the same as block($CD(N,P)$). In a cyclic(m) distribution, index j of the global array is mapped to logical processor number $MOD(CD(j,m)-1, P)$. Cyclic by definition means the same as cyclic(1).

In other words, in a block distribution, contiguous blocks of the array are distributed among the processors. In a cyclic distribution, array elements are distributed among processors in a round-robin fashion. In a cyclic(m) distribution, blocks of size m are distributed cyclically. The cyclic(m) distribution with $1 < m < \lceil N/P \rceil$ is commonly referred to as a block-cyclic distribution with block size m [5]. Block and cyclic distributions are special cases of the general cyclic(m) distribution. A cyclic(m) distribution with $m = \lceil N/P \rceil$ is a block distribution and a cyclic(m) distribution with $m = 1$ is a cyclic distribution. The formulae for conversion between local and global indices for the different distributions are given in Table 1.

3 Block(m) to Cyclic Redistribution

We first consider the case of block(m) to cyclic redistribution.

Theorem 3.1 *Let l_{i1} and l_{i2} be the local array sizes in processor p_i corresponding to block(m) and cyclic distributions respectively. In a block(m) to cyclic redistribution, the number of processors to which p_i has to send data is*

$$\begin{array}{ll} P-1 & \text{if } l_{i1} \geq P \\ l_{i1} \text{ or } l_{i1}-1 & \text{if } l_{i1} < P \end{array}$$

The number of processors from which p_i has to receive data is

$$\begin{array}{ll} P-1 & \text{if } l_{i2} \geq P \\ l_{i2} \text{ or } l_{i2}-1 & \text{if } l_{i2} < P \end{array}$$

Proof: Note that if N is not a multiple of P, l_{i1} may not be equal to l_{i2}. If $l_{i1} < P$, each of the l_{i1} elements of p_i corresponding to a block(m) distribution will lie in a different processor when the array is distributed cyclically. At most one of the l_{i1} elements will be remapped to processor p_i itself. Therefore, p_i will have to send data to either l_{i1} or $l_{i1} - 1$ processors. If $l_{i1} \geq P$, then clearly p_i has at least one element to send to every other processor. The result for the number of processors from which p_i has to receive data can be proved similarly. □

The algorithm for block(m) to cyclic redistribution is given in Figure 2. In the send phase, each processor only needs to calculate the destination processor of the first element of the local array. The other elements have to be sent to the other processors in a round-robin fashion. Thus the array is scanned only once, which makes good use of the cache. In the receive phase, the data received from other processors has to be stored in contiguous memory locations in order of logical processor number. In every processor, the packet received from processor 0 is stored first; from processor 1 second and so on. Hence addresses do not need to be calculated. If the amount of data to be received from all processors is known, the packets can be directly received into appropriate locations in the array.

In a block(m) distribution, the last element N of the global array is mapped to processor $p_N = (N-1)/m$. If $p_N < P-1$, no elements of the array are mapped to processors $p_N+1, p_N+2, ..., P-1$. The index of the last element of the local array in processors $p < p_N$ is $last_index = m$. The index of the last element in p_N is $last_index = MOD(N-1, m)+1$. The index of the first element of $p_s \leq p_N$ that is mapped to p in a cyclic distribution is given by

$$first_index = MOD[p - MOD\{p_s\, MOD(m,P), P\} + P, P] + 1$$

If m is divisible by P, the first element of p_s that is mapped to p is $p+1$. However, if m is not divisible by

Table 1: Data Distribution and Index Conversion

Note: This assumes that arrays are indexed starting from 1 and processors are numbered starting from 0

$$CD(j,k) = (j+k-1)/k \quad \text{and} \quad CR(j,k) = j - k \times CD(j,k)$$

	BLOCK(m)	CYCLIC	CYCLIC(m)
global index (g) to processor number (p)	$p = CD(g,m) - 1$	$p = MOD(g-1,P)$	$p = MOD(CD(g,m)-1,P)$
global index (g) to local index (l)	$l = m + CR(g,m)$	$l = (g-1)/P + 1$	$l = MOD(g-1,m) + 1 + (g/(m\,P))m$
local index (l) to global index (g)	$g = l + m\,p$	$g = (l-1)P + p + 1$	$g = MOD(l-1,m) + 1 + (P((l-1)/m) + p)m$

P, a shift is introduced in this simple mapping which is taken into account by the MOD expression in the above equation. Hence, the number of elements to be sent from any processor p_s to p is

$$\begin{cases} 0, & \text{if (last_index < first_index) or } (p_s > p_N) \\ (\text{last_index} - \text{first_index})/P + 1, & \text{otherwise} \end{cases}$$

where $first_index$, $last_index$ and p_N are calculated as above.

4 Cyclic to Block(m) Redistribution

A cyclic to block(m) redistribution is essentially the reverse of a block(m) to cyclic redistribution. The algorithm for cyclic to block(m) redistribution is given in Figure 4. In a cyclic distribution, the size of the local array in processor p is

$$L = \begin{cases} \lceil N/P \rceil & \text{if } MOD(N,P) = 0, \text{ or } p < MOD(N,P) \\ \lceil N/P \rceil - 1 & \text{otherwise} \end{cases}$$

In the send phase, each processor p calculates the destination processor p_d and the destination local address l_d of the first element of its local array as $p_d = CD(p+1,m) - 1$ and $l_d = m + CR(p+1,m)$. The first $(m - l_d)/P + 1$ elements from p_s have to be sent to p_d. The next set of elements starting from $i = (m - l_d)/P + 2$ have to be sent to processor $p_{d1} = CD((i-1)P + p + 1, m) - 1$. The destination local address of element i is given by $l_{d1} = m + CR((i-1)P+p+1, m)$ and so $(m - l_{d1})/P + 1$ elements starting from i have to be sent to processor p_{d1}. This process is continued for the remaining elements of the array. Since all the elements to be sent to a particular processor are located in contiguous memory locations, there is no need to create packets.

In the receive phase, the packets received cannot be directly stored in the array as the data has to be stored with a stride equal to the number of processors. Hence the packets have to be stored in a temporary buffer in memory. This gives us two choices in implementing the receive phase :–

1. **Synchronous Method**: Each processor waits till it receives packets from all other processors, before placing any data in the local array. This increases the memory requirements of the algorithm and also increases the processor idle time. These problems worsen as the number of processors is increased, so this method is not scalable.

2. **Asynchronous Method**: The processors do not wait for data from all processors to arrive. Instead, each processor takes any packet which has arrived and places the data from that packet into appropriate locations in the local array. This method **overlaps computation and communication.** Each processor posts non-blocking receive calls and waits for any packet to arrive. As soon as a packet is received, it places the data in appropriate locations in the local array. During this time, other packets may reach the processor. When the processor has placed all the data from the earlier packet into the local array, it takes up the next packet and so on. This reduces processor idle time. Since all packets do not have to be in memory at the same time, it also reduces memory requirements. This method is scalable as neither processor idle time nor memory requirements increase as the number of processors is increased.

The array locations where incoming data has to be stored can be calculated as follows. The source processor (p_s) of the first element of the local array is given by $p_s = MOD(m\,p, P)$. The next $(P-1)$ elements will be received from the remaining processors in order of processor number. This cycle is repeated for all elements of the local array. If all packets are present in memory (Synchronous Method), the local array needs to be scanned only once to be filled. If the packets are processed one at a time (Asynchronous Method), the array elements are filled with stride P and the array has to be scanned P times. So, clearly the Synchronous Method makes better use of the cache than the Asynchronous Method. Figure 3 compares the

Send Phase

1. Create packets to send to other processors.
2. Calculate the destination processor (p_d) of the first element of the local array as $p_d = MOD(p\,m, P)$.
3. Put the first element into the packet for processor p_d.
4. For $i = 2$ to L do
5. Put element i into the packet for processor $MOD(p_d + i, P)$.
6. Exchange packets with other processors.

Receive Phase

1. Last processor with data is $p_N = (N-1)/m$
2. For $p_s = 0$ to p_N do
3. If ($p_s = p_N$) then
4. $last_index = MOD(N-1, m) + 1$
5. Else
6. $last_index = m$
7. The index of the first element of p_s mapped to p is $first_index = MOD[p - MOD\{p_s\,MOD(m,P), P\} + P, P] + 1$
8. Number of elements to be received from p_s is
   0, if ($last_index < first_index$)
   $(last_index - first_index)/P + 1$, otherwise
9. No data is to be received from processors $p_N + 1, p_N + 2, ..., P - 1$.
10. Read the incoming data directly into appropriate locations in the array.

Figure 2: Algorithm for Block(m) to Cyclic Redistribution

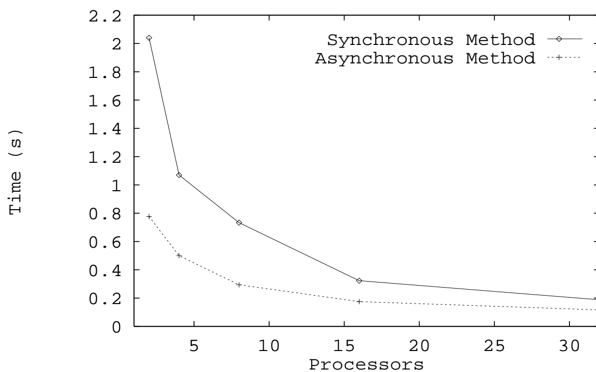

Figure 3: Cyclic to Block(m) Redistribution

performance of the Synchronous and Asynchronous Methods on the *Intel Paragon* for a global array with 1M (2^{20}) elements and the number of processors varied between 2 and 32. We observe that the Asynchronous Method performs much better than the Synchronous Method as it overlaps computation and communication and thus reduces processor idle time. This difference is larger for a small number of processors because in this case, the amount of data communicated per processor is larger.

5 Cyclic(x) to Cyclic(y) Redistribution

For a general cyclic(x) to cyclic(y) redistribution, there is no direct algebraic formula to calculate the set of elements to send to a destination processor and the local addresses of these elements at the destination [6]. Gupta et al [6] propose a virtual processor approach in which a block-cyclic distribution is considered to be either a virtual block or a virtual cyclic distribution using many virtual processors per physical processor.

There has also been some research on a slightly different problem of determining the local addresses and communication sets for array assignment statements like $A(l_1 : h_1 : s_1) = B(l_2 : h_2 : s_2)$ where A and B have different cyclic(m) distributions. Chatterjee et al [3] present an approach to calculate the sequence of local memory addresses that a given processor must access and the corresponding communication sets, using a finite state machine model which requires solving linear Diophantine equations. Stichnoth [8] defines a cyclic(m) distribution as a disjoint union of slices, where a slice is a sequence of array indices specified by a lower bound, upper bound and stride ($l : h : s$). The processor and index sets are calculated in terms of unions and intersections of slices which also involves solving linear Diophantine equations.

We propose a simpler and faster method for the cyclic(x) to cyclic(y) redistribution, which can be easily implemented and does not involve the overhead of solving linear Diophantine equations. We propose special methods for the cases where x is a multiple of y or y is a multiple of x. We use a general method when there is no particular relation between x and y.

5.1 Special case $x = k\,y$

We first consider the special case where x is a multiple of y. Let $x = k\,y$.

Theorem 5.1 *In a cyclic(x) to cyclic(y) redistribution where $x = k\,y$, if $k < P$, each processor communicates with k or $k-1$ processors. If $k \geq P$, each processor communicates with all other processors.*

Proof: Assume $k < P$. Since $x = k\,y$, each block of size x is divided into k sub-blocks of size y and distributed cyclically. Consider any processor p_i. Assume that it has to send its first sub-block of size y to processor p_j. Then the remaining $k-1$ sub-blocks of the first block are sent to the next $k-1$ processors in order. The next $k(P-1)$ sub-blocks of the global array are located in the other $P-1$ processors. This results in a total of $k\,P$ sub-blocks. Hence the $(k+1)^{th}$ sub-block of size y in p_i is also sent to p_j. Thus all

Send Phase

1. $i = 1$
2. While $(i \leq L)$ do
3. Calculate the destination processor (p_d) and destination local address (l_d) of element i of the local array as $p_d = CD((i-1)P + p + 1, m) - 1$ and $l_d = m + CR((i-1)P + p + 1, m)$
4. Elements i to $i + (m - l_d)/P$ have to be sent to processor p_d.
5. $i = i + (m - l_d)/P + 1$
6. Send packets to other processors.

Receive Phase

Synchronous Method:
1. Receive all packets.
2. Get the first element of the local array from the packet received from processor $p_s = MOD(m\,p, P)$.
3. For $i = 2$ to L do
4. Get element i of the local array from the packet received from processor $p_s = MOD(p_s + 1, P)$.

Asynchronous Method:
1. Source processor of first element of the local array is $p_s = MOD(m\,p, P)$.
2. Source processor of kth element $(2 \leq k \leq P)$ is $MOD(p_s + k - 1, P)$.
3. For $i = 0$ to $P - 1$ do
4. Receive a packet from any processor (say p_i)
5. Starting from the location calculated above, place elements from the packet into the array with stride P.

Figure 4: Algorithm for Cyclic to Block(m) Redistribution

sub-blocks from p_i are sent to k processors starting from p_j. One of these processors may be p_i itself, in which case p_i has to send data to $k-1$ processors. For the receive phase, consider the first $k\,P$ sub-blocks of size y in the global array corresponding to the first P blocks of size x. Let us number these $k\,P$ sub-blocks from 0 to $k\,P - 1$. Out of these, the sub-blocks that are mapped to processor p_i in the new distribution are numbered p_i to $P(k-1)+p_i$ with stride P. These sub-blocks come from $\frac{\{P(k-1)+p_i\}-p_i}{P} + 1 = k$ processors. One of these processors might be p_i itself, in which case p_i receives data from $k-1$ processors.

If $k \geq P$, each block of size x has to be divided into k sub-blocks and distributed cyclically, where the number of sub-blocks is greater than or equal to the number of processors. So clearly each processor has to send to and receive from all other processors (all-to-all communication). □

The algorithm for cyclic(x) to cyclic(y) redistribution, where $x = k\,y$ is given in Figure 5. In the send phase, each processor p calculates the destination processor p_d of the first element of its local array as $p_d = MOD(k\,p, P)$. The first y elements have to be sent to p_d, the next y to $MOD(p_d+1, P)$, the next to $MOD(p_d+2, P)$ and so on till the end of the first block of size x. The next k sub-blocks of size y have to be sent to the same set of k processors starting from p_d. The sequence of destination processors can be stored and need not be calculated for each block of size x. In the receive phase there are two cases depending on the value of k :-

1. $(k \leq P)$ and $(MOD(P, k) = 0)$: In this case, each processor p calculates the source processor of the first block of size y of its local array as $p_s = p/k$. The next block of size y will come from processor $MOD(p_s+P/k, P)$, the next from $MOD(p_s + 2(P/k), P)$ and so on till the first k blocks. The above sequence of processors is repeated for the remaining sets of k blocks of size x and hence can be stored and used. If the Synchronous Method is used for receiving data, the local array needs to be scanned only once and the ith block, $0 \leq i \leq \lceil L/y \rceil - 1$, of size y of the local array will be read from the packet received from processor $MOD(p_s + i(P/k), P)$. If the Asynchronous Method is used, the first block from the packet received from some processor p_i will be stored starting at the location calculated above. The remaining blocks will be stored with stride x.

2. If k does not satisfy the above condition, it is necessary to calculate the source processor of the first element $(j = i\,y + 1)$ of each block of size y, $0 \leq i \leq \lceil L/y \rceil - 1$, of the local array as $p_s = MOD[(i\,P + p)/k, P]$. The block is read from the packet received from p_s. The sequence of processors does not repeat itself and hence cannot be stored. In this case, the Synchronous Method is used.

Figure 6 compares the performance of the Synchronous and Asynchronous Methods on the *Intel Paragon* for a cyclic(4) to cyclic(2) redistribution of a global array with 1M elements. As in the case of cyclic to block(m) redistribution, we observe that the Asynchronous Method performs better than the Synchronous Method even though in this case each processor communicates with at most two other processors.

5.2 Special case $y = k\,x$

We now consider the special case where y is a multiple of x. Let $y = k\,x$. This is essentially the reverse of the case where $x = k\,y$. The algorithm for cyclic(x) to cyclic(y) redistribution, where $y = k\,x$, is given in

Send Phase

1. Create packets for communication.
2. Calculate the destination processor (p_d) of the first element of the local array as $p_d = MOD(k\,p, P)$.
3. For each block of size x in the local array do
4. For $i = 0$ to $k-1$ do
5. Put elements $(i\,y+1)$ to $(i+1)y$ of the current block of size x into the packet for processor $MOD(p_d+i, P)$.
6. Exchange packets with other processors.

Receive Phase

1. If $(k \le P)$ and $(MOD(P,k) = 0)$ then
2. Calculate the source processor (p_s) of the first element of the local array as $p_s = p/k$.
   Synchronous Method:
3. Receive packets from all processors.
4. For $j = 1$ to $\lceil L/x \rceil$ do
5. For $i = 0$ to $k-1$ do
6. Read the next block of size y from the packet received from processor $MOD(p_s + i(P/k), P)$.
   Asynchronous Method:
3. The ith block of size y, $0 \le i \le k-1$, will be received from processor $MOD(p_s + i(P/k), P)$.
4. For $i = 0$ to $k-1$ do
5. Receive a packet from any processor p_i.
6. Place the first block of size y starting from the location calculated above and the other blocks with stride x.
7. Else
8. Receive packets from all processors.
9. For $i = 0$ to $\lceil L/y \rceil - 1$ do
10. Calculate the source processor (p_s) of the first element ($j = i\,y+1$) of this block of size y as $p_s = MOD[(i\,P+p)/k, P]$
11. Read the block from the packet received from p_s.

Figure 5: Algorithm for Cyclic(x) to Cyclic(y) Redistribution, where $x = k\,y$

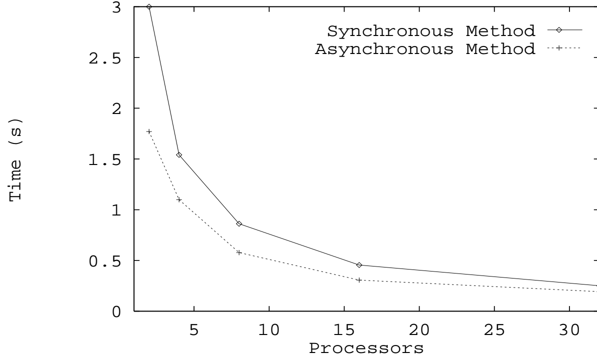

Figure 6: Cyclic($k\,y$) to Cyclic(y) Redistribution

Figure 7. In the send phase, there are two cases depending on the value of k :-

1. $(k \le P)$ and $(MOD(P,k) = 0)$: In this case, each processor p calculates the destination processor of the first block of size x of its local array as $p_d = p/k$. The next block of size x has to be sent to processor $MOD(p_d + P/k, P)$, the next to $MOD(p_d + 2(P/k), P)$ and so on till the first k blocks. The above sequence of processors is repeated for the remaining sets of k blocks of size x, and hence need not be calculated again.

2. If k does not satisfy the above condition, it is necessary to individually calculate the destination processor of each block i of size x, $0 \le i \le \lceil L/x \rceil - 1$, as $p_d = MOD[(i\,P+p)/k, P]$.

In the receive phase, each processor p calculates the source processor of the first element of its local array as $p_s = MOD[k\,p, P]$. If the Synchronous Method is used to receive data, for each block of size y of the local array, the k sub-blocks of size x are read from the packets received from the k processors starting from p_s in order of processor number. If the Asynchronous Method is used, we know that the ith block of size x of the local array, $0 \le i \le k-1$, will be received from processor $MOD(p_s+i, P)$. Thus the local index of the first block received from any source processor can be calculated. The remaining blocks have to be stored with stride y.

5.3 General Case

In a general cyclic(x) to cyclic(y) redistribution in which there is no particular relation between x and y, there is no direct way to determine which indices have to be sent to which processor and where in the local array to place incoming data. For this case, the following general method is proposed. In the send phase, each processor p scans the local array and calculates the destination processor of each element i as $p_d = MOD[\{MOD(i-1,x) + (P((i-1)/x)+p)x + y\}/y - 1, P]$. The element is put in a packet for that processor. In the receive phase, each processor p calculates the source processor p_s for each element i in the local array as $p_s = MOD[\{MOD(i-1,y) + (P((i-1)/y)+p)y+x\}/x-1, P]$. After the packets from other processors are received, the processor fetches each element from the appropriate packet in order.

Send Phase

1. Create packets for communication.
2. If ($k \leq P$) and ($MOD(P, k) = 0$) then
3. Calculate the destination processor (p_d) of the first element of the local array as $p_d = p/k$.
4. For $j = 0$ to $\lceil L/y \rceil - 1$ do
5. For $i = 0$ to $k - 1$ do
6. Place the next block of size x of the local array into the packet for processor $MOD(p_d + i(P/k), P)$.
7. Else
8. For $i = 0$ to $\lceil L/x \rceil - 1$ do
9. Calculate the destination processor (p_d) of the first element ($j = i\,x + 1$) of this block of size x as $p_d = MOD[(i\,P + p)/k, P]$.
10. Place this block into the packet for p_d.
11 Exchange packets with other processors.

Receive Phase

1. Calculate the source processor (p_s) of the first element of the local array as $p_s = MOD[k\,p, P]$.

Synchronous Method:

3. Receive packets from all processors.
4. For each block of size y in the local array do
5. For $i = 0$ to $k - 1$ do
6. Read elements ($i\,x + 1$) to ($i + 1$)x of the current block of size y from the packet received from processor $MOD(p_s + i, P)$.

Asynchronous Method:

3. The ith block of size x, $0 \leq i \leq k - 1$, will be received from processor $MOD(p_s + i, P)$.
4. For $i = 0$ to $k - 1$ do
5. Receive a packet from any processor p_i.
6. Place the first block of size x starting from the location calculated above and the other blocks with stride y.

Figure 7: Algorithm for Cyclic(x) to Cyclic(y) Redistribution, where $y = k\,x$

6 Redistribution of Multidimensional Arrays

The redistribution of two and higher dimensional arrays can be classified into two types :-

1. **Shape Retaining**: The shape of the local array remains unchanged after the redistribution, eg. (block,block) to (cyclic,cyclic).

2. **Shape Changing**: The shape of the local array changes after the redistribution, eg. (block,*) to (*,block) where '*' indicates that the corresponding dimension is collapsed. This type of redistribution is required very often for multidimensional FFT and the ADI method.

The *shape retaining* and *shape changing* redistributions are quite different from each other and require different algorithms.

6.1 Shape Retaining Redistributions

A shape retaining redistribution may involve redistribution in only one dimension {eg. (block,block) to (cyclic,block)} or more than one dimension {eg. (block,block) to (cyclic,cyclic)}. If the redistribution is only along one dimension, it is similar to the redistribution of one-dimensional arrays and the same algorithms described earlier can be used. If both dimensions have to be redistributed, it can either be done directly or indirectly as a series of one-dimensional redistributions. In the indirect method, the array is redistributed separately along each dimension. For example, if an array has to be redistributed from (block,block) to (cyclic,cyclic), it is first redistributed to (block,cyclic) and then to (cyclic,cyclic). This method has the advantage that all the optimizations developed for one-dimensional arrays in the previous sections can be easily extended to multidimensional arrays. The order in which the dimensions are redistributed does not affect the performance. This is because the order of dimensions chosen only results in a different set of data values being communicated, and does not affect the amount or type of communication. So, one could also do the above redistribution as (block,block) to (cyclic,block) to (cyclic,cyclic).

In the direct method, data is sent directly to the destination processor corresponding to the new distribution. Hence the optimized algorithms developed for the one-dimensional case cannot be used. This method requires different algorithms for different number of dimensions and different types of redistributions and these algorithms cannot be optimized much. However data needs to be communicated only once in the direct method. Figure 8 compares the performance of the direct and indirect methods for redistributing an array of size $1K \times 1K$ from (block,block) to (cyclic,cyclic) on the *Intel Paragon*. The indirect method is found to perform much better even though data is communicated twice. This is because the algorithms for one-dimensional redistribution are highly optimized and a lot of the communication during each one-dimensional redistribution actually takes place in parallel. We have observed similar results for other array sizes also. Hence the indirect method is preferred.

6.2 Shape Changing Redistributions

This type of redistribution occurs when at least one dimension of the array is collapsed before or after the redistribution. Consider the redistribution from (X,*) to (*,Y) or vice-versa, where X and Y can be either block, cyclic or cyclic(m). This is basically a collapsed to distributed type of redistribution in one of the dimensions which is done as follows. Each processor divides the local array into packets along the collapsed dimension, depending on the type of the new distri-

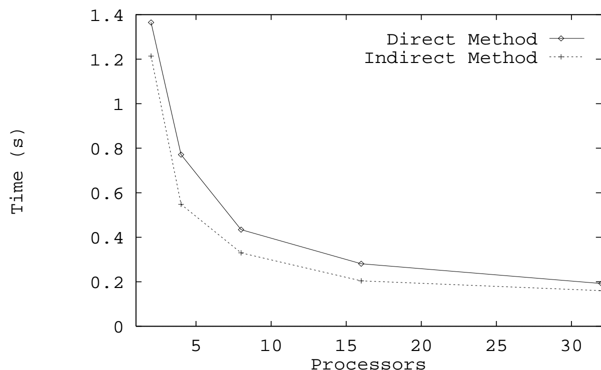

Figure 8: Redistribution of 2D arrays

bution Y. The processors then exchange packets with other processors. At the receiving end, packets are placed in the local array in order of source processor number. Data from the received packets may have to be placed in the local array either contiguously or with a stride, depending on the type of distributions X and Y.

The other type of redistribution involving change of shape of the local array is of the type (X,*) or (*,X) to (Y,Z), or vice versa. That is, in either the source or target distributions, one of the dimensions is collapsed and in the corresponding target or source distributions, both the dimensions are distributed. Each case of this type requires a different algorithm and so has to be considered separately.

7 Conclusions

Runtime array redistribution is required very often in HPF programs and it needs to be done efficiently in order to get good performance. We have described efficient algorithms for block(m) to cyclic, cyclic to block(m) and the general cyclic(x) to cyclic(y) type redistributions. The algorithms try to minimize address calculation and communication, and make good use of the cache. They are also practical enough to be easily implemented in the runtime library of an HPF compiler, or directly in application programs requiring redistribution. We have tested the performance of the algorithms on the *Intel Paragon* and observed that the Asynchronous Method performs better than the Synchronous Method as it overlaps computation and communication.

Acknowledgments

This work was sponsored in part by ARPA under contract no. DABT63-91-C-0028. The content of the information does not necessarily reflect the position or policy of the Government and no official endorsement should be inferred. Alok Choudhary's research is also supported by an NSF Young Investigator Award CCR-9357840 with a matching grant from Intel SSD.

References

[1] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance results. In *Proceedings of Supercomputing '93*, pages 351–360, November 1993.

[2] A. Carle, K. Kennedy, U. Kremer, and J. Mellor-Crummey. Automatic data layout for distributed memory machines in the D programming environment. Technical Report CRPC-TR93298, Center for Research on Parallel Computation, Rice University, February 1993.

[3] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating local addresses and communication sets for data parallel programs. In *Proceedings of Principles and Practices of Parallel Programming (PPoPP) '93*, pages 149–158, May 1993.

[4] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Version 1.0, May 1993.

[5] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. Fortran D language specifications. Technical Report COMP TR90-141, Rice University, 1990.

[6] S. Gupta, S. Kaushik, S. Mufti, S. Sharma, C. Huang, and P. Sadayappan. On the generation of efficient data communication for distributed memory machines. In *Proceedings of International Computing Symposium, Taiwan*, pages 504–513, 1992.

[7] R. Ponnusamy, R. Thakur, A. Choudhary, and G. Fox. Scheduling regular and irregular communication patterns on the CM-5. In *Proceedings of Supercomputing '92*, pages 394–402, November 1992.

[8] J. Stichnoth. Efficient compilation of array statements for private memory multicomputers. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, February 1993.

[9] R. Thakur and A. Choudhary. All-to-all communication on meshes with wormhole routing. In *Proceedings of the 8th International Parallel Processing Symposium*, April 1994.