IEEE Transactions on Parallel and Distributed Systems, (7)6:587–594, June 1996.

Efficient Algorithms for Array Redistribution

*Rajeev Thakur** *Alok Choudhary*† *J. Ramanujam*‡

Abstract

Dynamic redistribution of arrays is required very often in programs on distributed memory parallel computers. This paper presents efficient algorithms for redistribution between different cyclic(k) distributions, as defined in High Performance Fortran. We first propose special optimized algorithms for a cyclic(x) to cyclic(y) redistribution when x is a multiple of y, or y is a multiple of x. We then propose two algorithms, called the GCD method and the LCM method, for the general cyclic(x) to cyclic(y) redistribution when there is no particular relation between x and y. We have implemented these algorithms on the Intel Touchstone Delta, and find that they perform well for different array sizes and number of processors.

Index Terms: array redistribution, distributed-memory computers, High Performance Fortran (HPF), data distribution, runtime support

*Rajeev Thakur is with the Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439; email: `thakur@mcs.anl.gov`.

†Alok Choudhary is with the Dept. of Electrical and Computer Engineering, Syracuse University, Syracuse, NY 13244; email: `choudhar@cat.syr.edu`.

‡J. Ramanujam is with the Dept. of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803; email: `jxr@gate.ee.lsu.edu`.

1 Introduction

In distributed-memory parallel computers, arrays have to be distributed among processors in some fashion. The distribution can either be regular i.e. block, cyclic, or block-cyclic, as in Fortran D [2] and High Performance Fortran (HPF) [4, 9], or irregular in which there is no simple arithmetic function specifying the mapping of arrays to processors. The distribution of an array does not need to remain fixed throughout the program. In fact, it is very often necessary to change the distribution of the array at run-time, which is called *array redistribution*. This requires each processor to calculate what portions of its local array to send to other processors, what portions of its local array to receive from other processors, and perform the necessary communication. It is essential to use efficient algorithms for redistribution, otherwise the performance of the program may degrade considerably.

This paper describes efficient and practical algorithms for redistributing arrays between different cyclic(k) distributions, as defined in HPF. The cyclic(k) distribution is the most general regular distribution in which blocks of size k of the array are distributed among processors in a round-robin fashion. It is also commonly known as a *block-cyclic* distribution. Redistribution from a cyclic(x) to a cyclic(y) distribution, for any general x and y, is interesting because 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.

We first propose efficient algorithms for two special cases of the cyclic(x) to a cyclic(y) redistribution—when x is a multiple of y, or y is a multiple of x. We then propose two methods called the *GCD Method* and the *LCM Method* for the general case when there is no particular relation between x and y. The GCD and LCM methods make use of the optimized algorithms developed for the above special cases. The proposed algorithms have low runtime overhead, and are simple and practical enough to be used in the runtime library of a compiler, or directly in application programs requiring redistribution.

The rest of this paper is organized as follows. The notations, assumptions and definitions used in this paper are given in Section 2. Section 3 describes the algorithm for the special case of a cyclic(x) to cyclic(y) redistribution where x is a multiple of y. Section 4 describes the algorithm for the special case where y is a multiple of x. The GCD and LCM methods for the general case are proposed in Section 5. Section 6 discusses related work in this area, followed by conclusions in Section 7.

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

2 Notations and Definitions

The notations used in this paper are given in Figure 1. We assume that all arrays are indexed starting from 1, while processors are numbered starting from 0. We also assume that the number of processors on which the array is distributed remains the same before and after the redistribution. In HPF, an array can be distributed as block(m) or cyclic(m), which 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 $mod(CD(j,m)-1,P)$[1]. Cyclic by definition means the same as cyclic(1).

In other words, in a block distribution, contiguous blocks of the array are distributed among 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. 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 in HPF are given in Table 1.

The redistribution algorithms proposed in this paper are intended to be portable. Hence, we do not specify how data communication should be performed because the best communication algorithms are often machine dependent. Redistribution requires all-to-many personalized communication in general, and in many cases it requires all-to-all personalized communication. Algorithms

[1] $mod(a,b) = a \;\; modulo \;\; b$

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)	$g = l + m\,p$	$g = (l-1)P + p + 1$	$g = mod(l-1, m) + 1 +$ $(P((l-1)/m) + p)m$

to implement these communication patterns are described in detail in [15, 10, 17, 12, 13]. The performance results presented in this paper were obtained using the communication algorithms given in [15, 10, 17]. 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. The redistribution algorithms described in this paper are for one-dimensional arrays. Multidimensional arrays can be redistributed by applying these algorithms to each dimension of the array separately. In the rest of this paper, any division involving integers should be considered as integer division unless specified otherwise.

3 Cyclic(x) to Cyclic(y) Redistribution: Special Case $x = k\,y$

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. Hence, we consider two special cases where x is a multiple of y, or y is a multiple of x. For the general case where there is no particular relation between x and y, we propose two algorithms called the GCD method and the LCM method, which make use of the optimized algorithms developed for the above two special cases.

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

Theorem 3.1 *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. As a result, all 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 may 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). $\square$

The algorithm for cyclic(x) to cyclic(y) redistribution, where $x = k\,y$ is given in Figure 2. We call this the KY_TO_Y algorithm. 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 reused. The data received from other processors cannot be directly stored in the local array as it has to be stored with a stride. As a result, the data has to be first stored in a temporary buffer in memory. This gives us two choices in implementing the receive phase:-

   - **Synchronous Method**: In this method, each processor waits till it receives data 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.

Send Phase

1. The destination processor (p_d) of the first element of the local array is $p_d = mod(k\,p, P)$.
2. For each block of size x in the local array
3.   For $i = 0$ to $k-1$
4.   The destination processor of elements $(i\,y+1)$ to $(i+1)y$ of this block of size x is $mod(p_d + i, P)$.
5. Send data to other processors.

Receive Phase

1. If $(k \leq P)$ and $(mod(P,k) = 0)$ then
2.   The source processor (p_s) of the first element of the local array is $p_s = p/k$.

    Synchronous Method:

3.   Receive data from all processors into temporary buffers.
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 data received from processor $mod(p_s + i(P/k), P)$.

    Asynchronous Method:

3.   The i^{th} block of size y, $0 \leq i \leq k-1$, is to be received from processor $mod(p_s + i(P/k), P)$.
4.   For $i = 0$ to $k-1$ do
5.   Receive data from any processor p_i into a temporary buffer.
6.   Place the first block of size y in the local array starting from the location calculated above, and the other blocks with stride x.
7. Else
8.   Receive data from all processors into temporary buffers.
9.   For $i = 0$ to $\lceil L/y \rceil - 1$ do
10.   The source processor (p_s) of the first element $(j = i\,y+1)$ of this block of size y is $p_s = mod[(i\,P + p)/k, P]$
11.   Read this block of size y from the data received from p_s.

Figure 2: KY_TO_Y algorithm for cyclic(x) to cyclic(y) redistribution, where $x = k\,y$

- **Asynchronous Method**: In this 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 into appropriate locations in the local array. This method *overlaps computation and communication.* Each processor posts non-blocking receive calls and waits for data from any processor to arrive. As soon as a packet is received, it places the data in appropriate locations in the local array. During this time, data from other processors may have arrived. When the processor has placed all 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.

If the synchronous method is used for receiving data, the local array needs to be scanned only once and the i^{th} block, $0 \le i \le \lceil L/y \rceil - 1$, of size y of the local array will be read from the data received from processor $mod(p_s + i(P/k), P)$. If the asynchronous method is used, the first block from the data 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, $0 \le i \le \lceil L/y \rceil - 1$, of the local array as $p_s = mod[(i\,P + p)/k, P]$. The block is read from the data 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.

In the synchronous method, the local array needs to be scanned only once to be filled. In the asynchronous method, array elements are filled with a certain amount of stride and the array has to be scanned P times. So, clearly the synchronous method makes better use of the cache than the asynchronous method. We have tested the performance of the KY_TO_Y algorithm using both synchronous and asynchronous methods on the Intel Touchstone Delta. Figure 3 compares the performance of the synchronous and asynchronous methods for a cyclic(4) to cyclic(2) redistribution of a global array of 1M integers for different number of processors. 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. This is because the asynchronous method overlaps computation and communication, and thus reduces processor idle time. The better cache utilization of the synchronous method does not improve its overall performance. Figure 4 shows the performance of the KY_TO_Y algorithm for a cyclic(4) to cyclic(2) redistribution on 64 processors for different array sizes. For small arrays, the difference in performance between the synchronous and asynchronous methods is small, because of the small data sets. For large arrays, the difference is significant because of the higher processor idle time in the synchronous method.

4 Cyclic(x) to Cyclic(y) Redistribution: 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 $x = k\,y$.

Theorem 4.1 *In a cyclic(x) to cyclic(y) redistribution where $y = k\,x$, if $k < P$, each processor sends data to k or $k-1$ processors and receives data from k or $k-1$ processors. If $k \ge P$, each processor has to communicate with all other processors (all-to-all communication).*

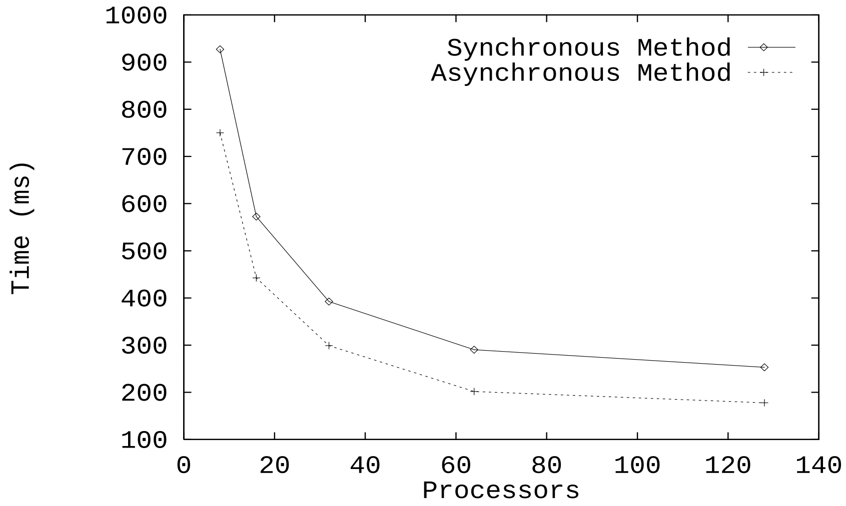

Figure 3: Performance of the KY_TO_Y algorithm for a cyclic(4) to cyclic(2) redistribution on the Intel Touchstone Delta. The array size is 1M integers, and the number of processors is varied.

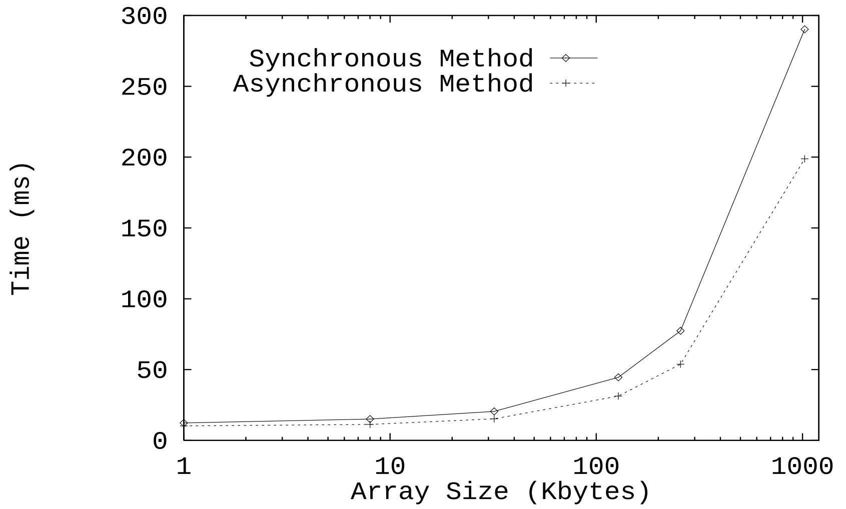

Figure 4: Performance of the KY_TO_Y algorithm for a cyclic(4) to cyclic(2) redistribution on the Intel Touchstone Delta. The number of processors is 64, and the array size is varied.

Proof: Assume $k < P$. Consider the first $k\,P$ sub-blocks of size x in the global array corresponding to the first P sub-blocks of size y. Let us number these $k\,P$ sub-blocks from 0 to $k\,P - 1$. Out of these, the sub-blocks that are located in processor p_i are numbered from p_i to $P(k-1) - 1 + p_i$ with stride P. In the new distribution, these sub-blocks will be mapped to $\frac{\{P(k-1)-1+p_i\}-p_i}{P} + 1 = k$ processors. One of these processors may be p_i itself, in which case p_i sends data to $k-1$ processors. In the receive phase, since $y = k\,x$, each block of size y in the new distribution consists of k sub-blocks of size x which will come from k processors. Consider any processor p_i. Assume that it receives its first sub-block of size x from processor p_j. Then the remaining $k-1$ sub-blocks of the first block are received from the next $k-1$ processors in order. The other $P-1$ processors receive the next $k(P-1)$ sub-blocks of the global array. This results in a total of $k\,P$ sub-blocks. Hence the next sub-block in p_i, which is the first sub-block of the next block of size y, is also received from p_j. As a result, all sub-blocks from p_i are received from k processors starting from p_j. One of these processors may be p_i itself, in which case p_i receives data from $k-1$ processors.

If $k \geq P$, each block of size y will consist of k sub-blocks of size x, 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 $y = k\,x$, is given in Figure 5. We call this the X_TO_KX algorithm. In the send 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 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, $0 \leq i \leq \lceil L/x \rceil - 1$, 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 $p_s = mod[k\,p, P]$. As in the KY_TO_Y algorithm, the receive phase can be implemented using either the synchronous method or the asynchronous method. If the synchronous method is used, 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 i^{th} block of size x of the local array, $0 \leq i \leq k-1$, will be

Send Phase	**Receive Phase**
1. If $(k \le P)$ and $(mod(P,k) = 0)$ then 2. The destination processor (p_d) of the first element of the local array is $p_d = p/k$. 3. For $j = 0$ to $\lceil L/y \rceil - 1$ 4. For $i = 0$ to $k-1$ 5. The destination processor of the next block of size x of the local array is $mod(p_d + i(P/k), P)$. 6. Else 7. For $i = 0$ to $\lceil L/x \rceil - 1$ 8. The destination processor (p_d) of the first element $(j = i\,x + 1)$ of this block of size x is $p_d = mod[(i\,P + p)/k, P]$. 9. Send data to other processors.	1. The source processor (p_s) of the first element of the local array is $p_s = mod[k\,p, P]$. *Synchronous Method*: 2. Receive data from all processors into temporary buffers. 3. For each block of size y in the local array do 4. For $i = 0$ to $k-1$ do 5. 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*: 2. The i^{th} block of size x, $0 \le i \le k-1$, is to be received from processor $mod(p_s + i, P)$. 3. For $i = 0$ to $k-1$ do 4. Receive data from any processor p_i into a temporary buffer. 5. Place the first block of size x in the local array starting from the location calculated above, and the other blocks with stride y.

Figure 5: X_TO_KX algorithm for cyclic(x) to cyclic(y) redistribution, where $y = k\,x$

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.

We have tested the performance of the X_TO_KX algorithm on the Intel Touchstone Delta for different array sizes and number of processors. Figure 6 compares the performance of the synchronous and asynchronous methods for a cyclic(2) to cyclic(4) redistribution of an array of 1M integers for different number of processors. Figure 7 compares the performance of the two methods for different array sizes on 64 processors. The results are similar to those obtained for the KY_TO_Y algorithm. The asynchronous method is found to perform better in all cases.

5 General Cyclic(x) to Cyclic(y) Redistribution:

Let us consider the general case of a cyclic(x) to cyclic(y) redistribution in which there is no particular relation between x and y. One algorithm for doing this is to explicitly calculate the destination and source processor of each element of the local array, using the formulae given in Table 1. We call this the General Method and is described below.

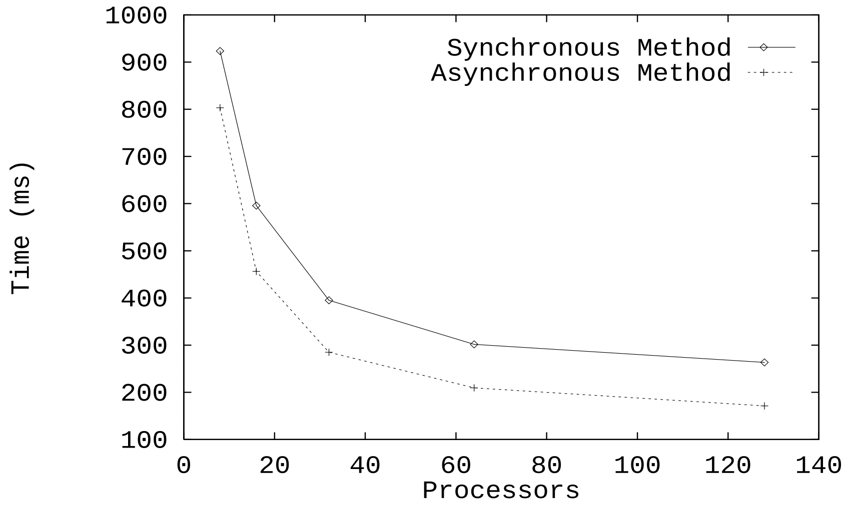

Figure 6: Performance of the X_TO_KX algorithm for a cyclic(2) to cyclic(4) redistribution on the Intel Touchstone Delta. The array size is 1M integers, and the number of processors is varied.

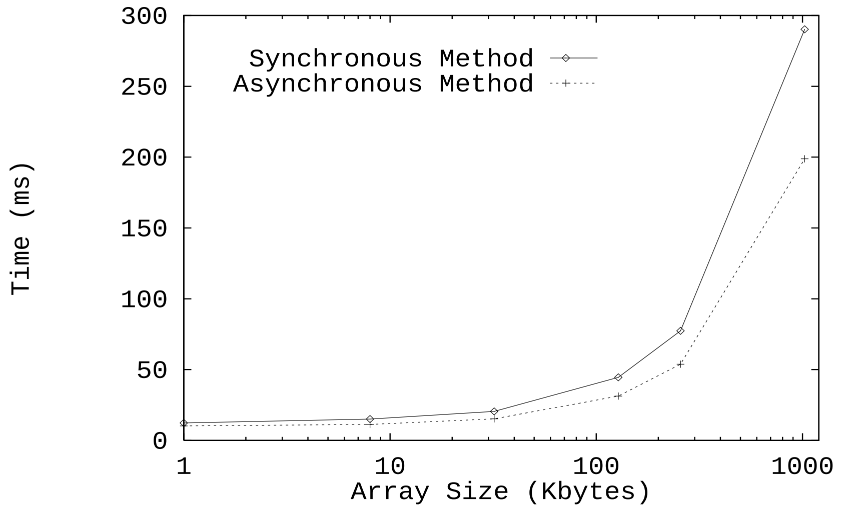

Figure 7: Performance of the X_TO_KX algorithm for a cyclic(2) to cyclic(4) redistribution on the Intel Touchstone Delta. The number of processors is 64, and the array size is varied.

5.1 General Method

In the send phase, the destination processor of each element of the local array can be determined by first calculating its global index based on the current distribution and then the destination processor based on the new distribution. These two calculations can be combined into a single expression to give the destination processor of element i of the local array as $p_d = mod[\{mod(i-1, x) + (P((i-1)/x) + p)x + y\}/y - 1, P]$. Similarly in the receive phase, the source processor of each element of the local array can be determined by first calculating its global index based on the new distribution and then the source processor based on the old distribution. These two calculations can be combined into a single expression to give the source processor of element i of the local array as $p_s = mod[\{mod(i-1, y) + (P((i-1)/y) + p)y + x\}/x - 1, P]$.

The drawback of this algorithm is that calculations are needed individually for all elements of the array. We propose two algorithms called the GCD method and the LCM method, which make use of the optimized KY_TO_Y and X_TO_KX algorithms, and hence require a lot less calculations.

5.2 GCD Method

This method takes advantage of the fact that we have developed special optimized algorithms for a cyclic(x) to cyclic(y) redistribution when $x = k\,y$ (the KY_TO_Y algorithm) and $y = k\,x$ (the X_TO_KX algorithm). In the GCD method, the redistribution is done as a sequence of two phases — cyclic(x) to cyclic(m) followed by cyclic(m) to cyclic(y), where $m = GCD(x, y)$. Since both x and y are multiples of m, the KY_TO_Y algorithm can be used for the cyclic(x) to cyclic(m) phase, and the X_TO_KX algorithm can be used for the cyclic(m) to cyclic(y) phase. This is described in Figure 8. The GCD method involves the cost of having to do two separate redistributions. For small arrays, the increased communication may outweigh the savings in computation, but for large arrays in some cases, the performance is better than that of the general method. This can be observed from Figure 9 which shows the performance of a cyclic(15) to cyclic(10) redistribution, for an array of size 1M integers on the Delta. We see that for small number of processors, the GCD method performs considerably better than the general method because of the saving in the amount of computation per processor. Since the size of the global array is kept constant, as the number of processors is increased, the size of the local array in each processor becomes smaller and each processor spends less time on address calculation. Hence the performance improvement of the GCD method over the general method is also small.

One disadvantage of the GCD method is that in the intermediate cyclic(m) distribution, the block size m is smaller than both x and y. In the KY_TO_Y and X_TO_KX algorithms, all the address and processor calculations are done for blocks of size x or y. Since m is the GCD of x and

GCD Method	LCM Method
1. Let $m = GCD(x, y)$.	1. Let $m = LCM(x, y)$.
2. Redistribute from cyclic(x) to cyclic(m) using the KY_TO_Y algorithm.	2. Redistribute from cyclic(x) to cyclic(m) using the X_TO_KX algorithm.
3. Redistribute from cyclic(m) to cyclic(y) using the X_TO_KX algorithm.	3. Redistribute from cyclic(m) to cyclic(y) using the KY_TO_Y algorithm.

Figure 8: GCD and LCM methods for the general cyclic(x) to cyclic(y) redistribution

y, m can even be equal to 1 in some cases (when x and y are relatively prime). When $m = 1$, calculations have to be done for each element, which is no better than in the general method. In this case, the general method is expected to perform better than the GCD method. Figure 10 shows the performance of cyclic(11) to cyclic(3) redistribution on the Delta for an array of size 1M integers. Since the GCD of 11 and 3 is 1, we find that the general method always performs better than the GCD method.

5.3 LCM Method

The key to getting good performance in this two-phase approach for redistribution is to have a large value for m. One way of ensuring that m is always large is by choosing m as the LCM of x and y. Since m is a multiple of both x and y, the X_TO_KX algorithm can be used for the cyclic(x) to cyclic(m) phase and the KY_TO_Y algorithm can be used for the cyclic(m) to cyclic(y) phase. This is described in Figure 8. Also, since m is larger than both x and y, all calculations are done for this larger block size. This results in fewer calculations than in the GCD and general methods. Figures 9 and 10 compare the performance of the LCM, GCD, and general methods for an array of 1M integers on different number of processors. We observe that the LCM method performs better in all cases. Figure 11 compares the performance of the LCM and general methods for a cyclic(11) to cyclic(3) redistribution keeping the number of processors fixed at 64 and varying the array size. We observe that for small arrays, the general method performs better because it has less communication, but for large arrays the LCM method performs better because the saving in computation is higher than the increase in communication.

Note that the timings for the GCD and LCM methods in Figures 9, 10, and 11 include the time for calculating the GCD and LCM. For the cyclic(15) to cyclic(10) redistribution, both the special-case redistributions within the GCD and LCM algorithms were performed using the asynchronous method, since the condition $((k \leq P)$ and $(mod(P, k) = 0))$ is satisfied in this case. For the cyclic(11) to cyclic(3) redistribution, however, the synchronous method was used in the KY_TO_Y

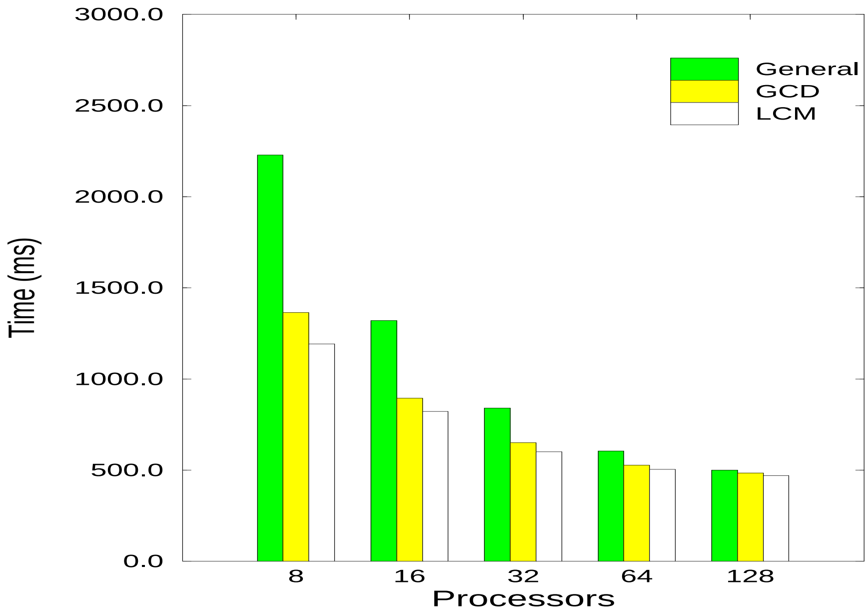

Figure 9: Comparison of the GCD, LCM, and general methods for a cyclic(15) to cyclic(10) redistribution on the Intel Touchstone Delta. The array size is 1M integers.

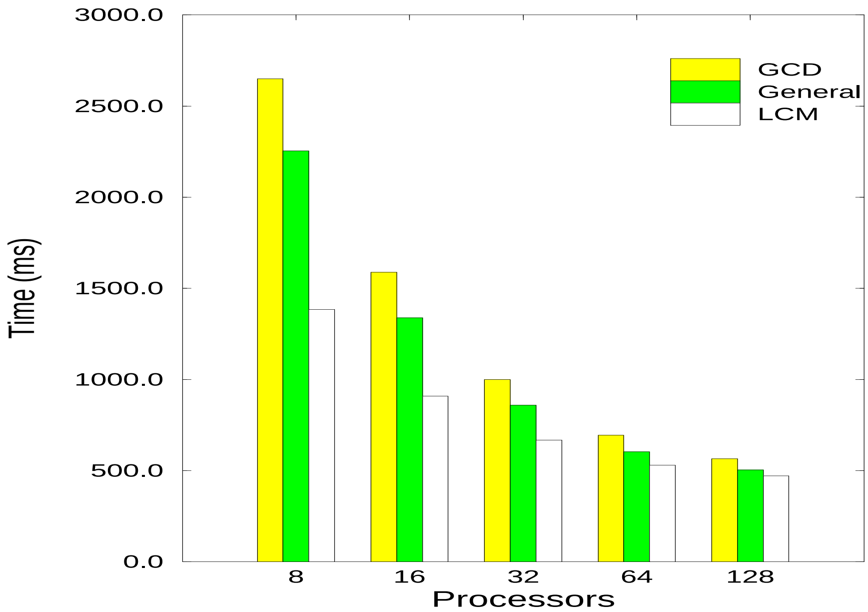

Figure 10: Comparison of the GCD, LCM, and general methods for a cyclic(11) to cyclic(3) redistribution on the Intel Touchstone Delta. The array size is 1M integers.

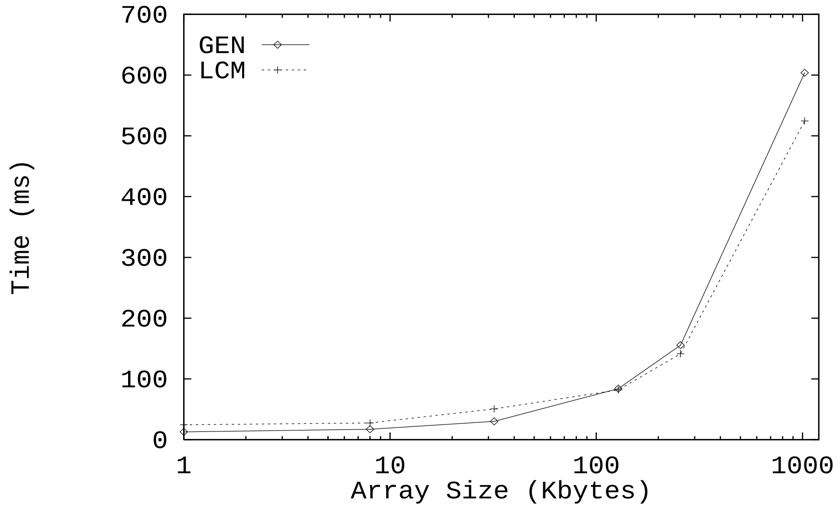

Figure 11: Comparison of the LCM and general methods for a cyclic(11) to cyclic(3) redistribution on the Intel Touchstone Delta. The number of processors is 64, and the array size is varied.

algorithm, since the condition $((k \leq P)$ and $(mod(P,k) = 0))$ is not satisfied, and the asynchronous method was used in X_TO_KX algorithm, since it does not require the above condition.

6 Related Work

Gupta et al. [3] and Koelbel [8] provide closed form expressions for determining the send and receive processor sets and data index sets for redistributing arrays between block and cyclic distributions. Efficient algorithms for block(m) to cyclic, and cyclic to block(m) redistributions are described in [16]. A model for evaluating the communication cost of data redistribution is given in [6]. A virtual processor approach for the general block-cyclic redistribution is proposed in [3]. Wakatani and Wolfe [18] describe a method of array redistribution, called strip mining redistribution, in which parts an array are redistributed in sequence, instead of redistributing the entire array at one time as a whole. The reason for doing this is to try to overlap the communication involved in redistribution with some of the computation in the program. Kalns and Ni [5] present a technique for mapping data to processors so as to minimize the total amount of data that must be communicated during redistribution. A multiphase approach to redistribution is discussed in [7]. Algorithms for redistribution, based on a mathematical representation for regular distributions called PITFALLS, are proposed in [11] .

There has also been some research on the closely related 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 [1] present an approach to calculate the sequence of local memory addresses that a given processor must access while doing a computation involving the regular array section $A(l : h : s)$, when the array A has a cyclic(k) distribution. They show that the local memory access sequence is characterized by a finite state machine of at most k states. Stichnoth [14] defines a cyclic(k) 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 for array assignment statements are calculated in terms of unions and intersections of slices.

7 Conclusions

We have presented efficient and practical algorithms for redistributing arrays between different cyclic(k) distributions, which is the most general form of redistribution. The algorithms are portable and independent of the communication mechanism used.

We find that the asynchronous method performs better than the synchronous method in all cases, because it reduces processor idle time. For the general case where there is no particular relation between x and y, the general method performs well for small arrays because it requires communication only once. However, for large arrays, the LCM method performs much better than the general method, because it requires a lot less address calculation. The GCD method also performs better than the general method for large arrays, provided the GCD of x and y is greater than 1. The LCM method always performs better than the GCD method because the LCM of x and y is always greater than their GCD.

The relative performance of the three methods may be affected by changes in the underlying architecture of the system. For example, in a system with very high communication costs, the general method may perform better since it has only one communication phase. Improved scalar compilers that optimize expensive index calculations may also improve the performance of the general method.

Acknowledgments

This work was supported in part by NSF Young Investigator Award CCR-9357840 with a matching grant from Intel SSD. J. Ramanujam is supported in part by NSF Young Investigator Award CCR-9457768, NSF grant CCR-9210422, and by the Louisiana Board of Regents through contract LEQSF (1991-94)-RD-A-09. This research was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by the Center for Research on Parallel Computation.

References

[1] 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.

[2] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. Fortran D Language Specifications. Technical Report COMP TR90-141, CRPC, Rice University, 1990.

[3] 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*, pages 504–513, 1992.

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

[5] E. Kalns and L. Ni. Processor Mapping Techniques Toward Efficient Data Redistribution. In *Proceedings of the* 8^{th} *International Parallel Processing Symposium*, pages 469–476, April 1994.

[6] S. Kaushik, C. Huang, R. Johnson, and P. Sadayappan. An Approach to Communication-Efficient Data Redistribution. In *Proceedings of the* 8^{th} *ACM International Conference on Supercomputing*, July 1994.

[7] S. Kaushik, C. Huang, J. Ramanujam, and P. Sadayappan. Multi-phase Array Redistribution: Modeling and Evaluation. In *Proceedings of the* 9^{th} *International Parallel Processing Symposium*, pages 441–445, April 1995.

[8] C. Koelbel. Compile-Time Generation of Regular Communication Patterns. In *Proceedings of Supercomputing '91*, pages 101–110, November 1991.

[9] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *High Performance Fortran Handbook.* The MIT Press, 1994.

[10] 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.

[11] S. Ramaswamy and P. Banerjee. Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers. In *Proceedings of The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 342–349, February 1995.

[12] S. Ranka, J. Wang, and M. Kumar. Irregular Personalized Communication on Distributed Memory Systems. *Journal of Parallel and Distributed Computing*, 25(1):58–71, February 15, 1995.

[13] D. Scott. Efficient All-to-All Communication Patterns in Hypercube and Mesh Topologies. In *Proceedings of the* 6^{th} *Distributed Memory Computing Conference*, pages 398–403, 1991.

[14] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating Communication for Array Statements: Design, Implementation, and Evaluation. *Journal of Parallel and Distributed Computing*, pages 150–159, April 1994.

[15] R. Thakur and A. Choudhary. All-to-All Communication on Meshes with Wormhole Routing. In *Proceedings of the 8^{th} International Parallel Processing Symposium*, pages 561–565, April 1994.

[16] R. Thakur, A. Choudhary, and G. Fox. Runtime Array Redistribution in HPF Programs. In *Proceedings of the Scalable High Performance Computing Conference*, pages 309–316, May 1994.

[17] R. Thakur, R. Ponnusamy, A. Choudhary, and G. Fox. Complete Exchange on the CM-5 and Touchstone Delta. *The Journal of Supercomputing*, 8(4):305–328, 1995.

[18] A. Wakatani and M. Wolfe. A New Approach to Array Redistribution: Strip Mining Redistribution. In *Proceedings of Parallel Architectures and Languages Europe (PARLE 94)*, pages 323–335, July 1994.