TECHNICAL PAPER
Concurrency: Practice and Experience
Submitted July 1995, revised June 1996
Accepted June 1996

# Redistribution of Block-Cyclic Data Distributions Using MPI[1]

David W. Walker
Mathematical Sciences Section
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6367
(615) 574-7401 (office)
(615) 574-0680 (fax)
walker@msr.epm.ornl.gov

Steve W. Otto
Department of Computer Sci. and Eng.
Oregon Graduate Institute
Beaverton, OR 97291-1000
otto@cse.ogi.edu

[1]This work was supported in part by ARPA under contract number DAAL03-91-C-0047 administered by ARO, and in part by DOE under contract number DE-AC05-84OR21400 and by National Science Foundation Grant number ASC-ASC-9005933.

# Redistribution of Block-Cyclic Data Distributions Using MPI[†]

D. W. Walker[‡] S. W. Otto[§]

**Abstract**

Arrays that are distributed in a block cyclic fashion are important for many applications in the computational sciences since they often lead to parallel algorithms with good load balancing properties. We consider the problem of redistributing such an array to a new block size. This operation is directly expressible in High Performance Fortran (HPF) and will arise in applications written in this language. Efficient message passing algorithms are given for the redistribution operation, expressed in the standardized message passing interface, MPI. The algorithms are analyzed and performance results from the IBM SP-1 and Intel Paragon are given and discussed. The results show that redistribution can be done in time comparable to other collective communication operations, such as broadcast and MPI_ALLTOALL.

## 1 Introduction

This paper presents different strategies for changing the distribution of an array from one block-cyclic distribution to another. Implementations using the MPI standard message passing interface are given, and performance results for different redistribution algorithms are presented for the IBM SP-1 and the Intel Paragon. These results are interpreted in terms of a simple performance model.

The block-cyclic data distribution is often used as a means of statistically load balancing inhomogeneous computations. For example, it is used in the ScaLAPACK parallel software library for load balancing dense matrix computations such as *LU* factorization (see [3] and references therein). Often the optimal block-cyclic data distribution for successive phases of an application will differ, hence redistribution is necessary to achieve the best performance.

[†]This work was supported in part by ARPA under contract number DAAL03-91-C-0047 administered by ARO, and in part by DOE under contract number DE-AC05-84OR21400 and by National Science Foundation Grant number ASC-ASC-9005933.

[‡]Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831-6367

[§]Dept. Computer Sci. and Eng., Oregon Graduate Institute, Beaverton, OR 97291-1000

High Performance Fortran (HPF) [9] contains a REDISTRIBUTE directive that can be used to change from one block-cyclic distribution to another. Our main motivation for this work has been to develop redistribution routines for use with ScaLAPACK. However, we believe the work presented here also will be of value to researchers designing HPF compilers that use MPI as a target.

MPI is a standard message passing interface for use in parallel applications and software libraries in message passing environments, particularly distributed memory concurrent computers. MPI is designed to be extensible and thread-safe, and to take advantage of features of the hardware. In this paper it is not practical to give complete descriptions of each MPI routine used in the redistribution routines. For this the reader is referred to the MPI specification [4].

The provision within HPF for data redistribution, and the importance of redistribution in the efficient use of parallel software libraries, has resulted in significant research in this area. Chatterjee et al. [1] and Gupta et al. [5] are concerned with determining the source and destination process sets and the local index mappings for distributions. Kalns and Ni [7] have presented an approach that seeks to minimize data movement by assigning processes logical process numbers. This is equivalent to renumbering the destination process numbers, and although this is permitted within the HPF specification it would not be directly compatible with the form of block-cyclic data distribution assumed in ScaLAPACK. Kalns and Ni have used this approach in their redistribution library Darel [6] which is implemented using MPI-F, the custom implementation of MPI for the IBM SP-1 and SP-2. Thakur et al. [11] have implemented an algorithm on the Intel Paragon for transforming between two block-cyclic data distributions by examining each index of the array to determine its destination process. An approach that handles the redistribution of data between block-cyclic distributions and arbitrary groups of processors has been proposed by Ramaswamy and Banerjee [10] and has been implemented on the Intel Paragon and the Thinking Machines Corporation CM-5.

The rest of this paper is arranged as follows. In Section 2 we discuss the block-cyclic data distribution and describe algorithms for increasing the block size of such distributions by an integer factor. The MPI kernels for performing these algorithms are given. In Section 3 performance results for redistributions on the IBM SP-1 and Intel Paragon are presented for the different redistribution algorithms. In Section 4 we discuss generalizations to arbitrary block sizes and multi-dimensional arrays. A summary is provided in Section 5.

## 2 Data Distribution Transformations

### 2.1 The Block Cyclic Data Distribution

The block-cyclic data distribution is widely used because it is both simple and a good method for achieving approximate static load balance in problems in which the computational load is nonuniformly spread across a domain. Given a set of $M$ items, $P$ processes, and a block size $r$, the block-cyclic data distribution first divides the items into contiguous blocks of $r$ items each (though the last block may not be full). Then the blocks are assigned to processes in round-robin fashion so that the $B$th block is assigned to process number $\text{mod}(B,P)$, where $\text{mod}(i,j)$ is the integer remainder on dividing integer $i$ by integer $j$. Thus, the block cyclic data distribution maps the global index $m$ to a process index, $p$, a block index, $b$, local to the process, and an item index, $i$, local to the block, with all indices starting at 0. The mapping $m \mapsto (p,b,i)$ may be written as

$$m \mapsto \left(\text{mod}(B,P),\ \left\lfloor \frac{B}{P} \right\rfloor,\ \text{mod}(m,r)\right), \tag{1}$$

where $B = \lfloor m/r \rfloor$ is the global block index. It should be noted that Eq. 1 reverts to an unblocked, cyclic distribution when $r = 1$, with local index $i = 0$ for all blocks. A noncyclic, block distribution is recovered when $r = \lceil M/P \rceil$, in which case there is a single block in each process with block index $b = 0$ [2].

For multi-dimensional arrays, a block-cyclic data distribution is obtained by applying the one-dimensional block-cyclic data distribution independently to the index set of each of the array dimensions, having first specified the block size and the number of processes for each dimension.

### 2.2 Redistribution Algorithms

We will now show how MPI may be used to transform a one-dimensional array from one block-cyclic data distribution to another. Initially we shall restrict our attention to the case in which the block size increases by an integer factor, $K$, from $r$ to $Kr$. The more general case of arbitrary block size changes for multi-dimensional arrays is considered in Section 4.

We shall refer to a set of $L = PK$ successive blocks as a *superblock*. Thus the blocks globally indexed by 0 to $L-1$ are the first superblock, those indexed by $L$ to $2L-1$ are the second superblock, and so on. It should be noted that the communication pattern required to redistribute each superblock is the same, a fact also remarked upon by Ramaswamy and Banerjee [10]. Figure 1 shows an example of redistribution for $P = 4$ and $K = 3$. In this case the length of a superblock is $L = 12$ blocks and, as can be seen in the figure, the communication pattern repeats after the first 12 blocks.

Each superblock is redistributed in exactly the same way, so it is sufficient to describe a redistribution algorithm for just one superblock of $L$ blocks. The

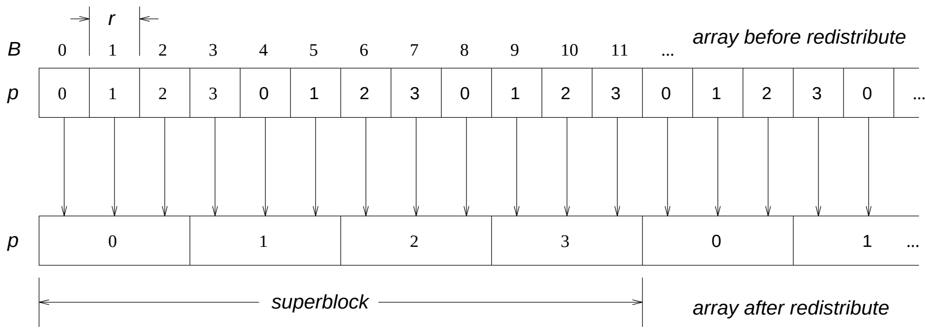
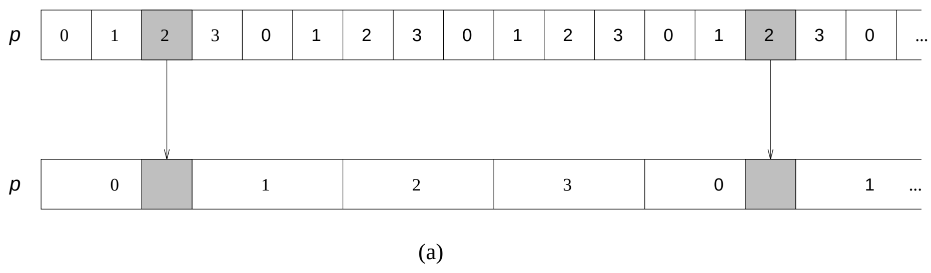


Figure 1: Block-cyclic array redistribution for the case $P = 4$ and $K = 3$. $B$ is the global block index.

last superblock may be incomplete, and this may be handled either by inserting conditional statements into the basic superblock redistribution code, or by redistributing the last superblock separately. We shall describe the algorithm for redistributing a full superblock, where each of the $P$ processes originally contains $K$ blocks of $r$ elements and after redistribution contains one block of $Kr$ elements.

Since the communication pattern between processes is the same for each superblock, blocks at the same position within a superblock are always communicated between the same pair of processes. This is shown in Fig. 2 for $P = 4$ processes, and an expansion factor of $K = 3$. The block at position 2 in the superblock (shown shaded) is sent from process 2 to process 0 in each superblock. In the algorithms presented below the redistribution takes place in $K$ communication phases. In a phase, each process sends/receives one block from/into the same position in each superblock. We also wish to do the entire communication, for all superblocks, in $K$ communication steps. To do this we can define an MPI derived datatype that picks out a block at a given position in each superblock, as shown in part (b) of Fig. 2. This datatype can then be passed to the MPI point-to-point communication routines to specify communication for all superblocks simultaneously.

In Fig. 3 we present the MPI/Fortran code for creating the derived datatype, `newtype`, which picks out one block from each superblock. `newtype` consists of a single block of data, but we have used the MPI-defined upper bound marker, MPI_UB, to set the extent of `newtype` to $K$ blocks. An alternative approach to creating a derived datatype that picks out one block from each superblock is to use the MPI vector datatype constructor, MPI_TYPE_VECTOR, with a block length of $r$ and a stride of $Kr$ elements.



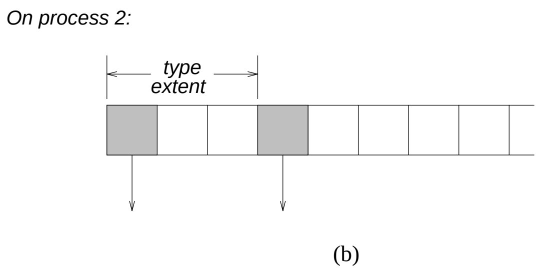


Figure 2: Using a derived datatype to transfer all superblocks together. In part (a), the shaded regions denote the data sent by process 2 during one of the $k$ steps of the redistribution algorithm. Only the first two superblocks are shown; the pattern repeats for all superblocks. In part (b), the same data movement is shown from the point of view of process 2. The process sends every third block of its local array. The type extent that produces this effect is also shown.

```
call mpi_comm_size (comm, p, ierr)
call mpi_type_extent (intype, sizeofdata, ierr)
call mpi_type_contiguous (r, intype, blocktype, ierr)
disp(1) = 0
disp(2) = sizeofdata*k*r
type(1) = blocktype
type(2) = MPI_UB
blen(1) = 1
blen(2) = 1
call mpi_type_struct (2, blen, disp, type, newtype, ierr)
call mpi_type_commit (newtype, ierr)
call mpi_type_free (blocktype, ierr)
```

Figure 3: Fortran 77 code for creating the derived datatype `newtype` for redistributing an array of type `intype`. We have assumed that all the superblocks are full, i.e., that `m` is divisible by `p*k*r`, where `p` is the number of processes in the communicator `comm`.

### 2.2.1 Redistribution Using Nonblocking Receives

The simplest approach to designing a redistribution routine is to use wildcarded, nonblocking receives. When receiving data in this way, each process needs to be able to identify the data that it receives. One way to do this is to use "self-describing" messages — when sending a block, the source process uses the routine MPI_PACK to prefix the global block index to the data sent. This global block index is extracted by the receiving process using the routine MPI_UNPACK and is used to determine where to store the blocks received.

We shall first present two versions of the redistribution routine using nonblocking receives. In version 1 only a maximum of one receive is outstanding (i.e., is posted but not yet completed) in each process. In version 2 a maximum of $K$ receives may be outstanding in each process. In the former case the redistribution routine needs to provide buffering for only one message — $M/(KP)$ data items. In the latter case buffering must be provided for $K$ messages, i.e., $M/P$ data items. Thus, the buffering required is the same as the number of data items per process.

An outline of the MPI code for versions 1 and 2 of the redistribution routine is given in Figs. 4 and 5, respectively. In version 1 each of the $K$ communication phases posts a receive, sends data, and then waits for completion of the receive. Thus, each process is synchronized with another on each pass through the loop. We shall therefore refer to this as the synchronized nonblocking receive redistribution routine. In version 2 all the $K$ receives are posted and then the corresponding $K$ sends are performed. Finally the routine waits until all the receives have completed by calling MPI_WAITANY $K$ times. Less synchronization occurs in version 2, so we shall refer to this as the unsynchronized nonblocking receive redistribution routine.

There are a couple of points to note about these two nonblocking receive redistribution algorithms. First, in the synchronized case, in phase $k$ a process receives local block $k$ of the source process. If we were to query the return status of each receive it would then be possible to compute the global block index on the receiving process, and it would not be necessary to pack the global block index into each message. In the unsynchronized case the packing/unpacking can also be avoided, but it would be necessary for each process to keep a count of how many messages it had received from each process. Then, taking advantage of the fact that MPI guarantees non-overtaking messages between pairs of processes, it is possible to find the local block index on the source process and hence to find the global block index. Finally, in the unsynchronized algorithm the data can be sent in ready mode, provided a barrier synchronization is performed after posting all the receives. This may result in improved performance on some systems.

```
create general datatype, newtype

nsuperblks = m/(p*k*r)
do istep=0,k-1
  call mpi_irecv (rbuf, rbufsize, MPI_PACKED,
        MPI_ANY_SOURCE, tag, comm, reqobj, ierr)
  b = p*istep + myrank
  soffset = istep*r
  pos = 0
  call mpi_pack (b, 1, MPI_INTEGER, sbuf, sbufsize,
        pos, comm, ierr)
  call mpi_pack (a(soffset), nsuperblks, newtype,
        sbuf, sbufsize, pos, comm, ierr)
  dest = b/k
  call mpi_send (sbuf, pos, MPI_PACKED,
        dest, tag, comm, ierr)
  call mpi_wait (reqobj, status, ierr)
  pos = 0
  call mpi_unpack (rbuf, rbufsize, pos, b, 1,
        MPI_INTEGER, comm, ierr)
  roffset = r*mod(b,k)
  call mpi_unpack (rbuf, rbufsize, pos, b(roffset),
        nsuperblks, newtype, comm, ierr)
end do
```

Figure 4: Fortran 77 code for increasing the block size of a block-cyclic data distribution by a factor `k` using nonblocking receives. `p` is the number of processes in the communicator `comm`.

```
  create general datatype, newtype

nsuperblks = m/(p*k*r)
do istep=0,k-1
  j = istep*rbufsize/k
  call mpi_irecv (rbuf(j), rbufsize, MPI_PACKED,
        MPI_ANY_SOURCE, tag, comm, reqobj(istep), ierr)
end do
do istep=0,k-1
  b = p*istep + myrank
  soffset = istep*r
  pos = 0
  call mpi_pack (b, 1, MPI_INTEGER, sbuf, sbufsize,
        pos, comm, ierr)
  call mpi_pack (a(soffset), nsuperblks, newtype,
        sbuf, sbufsize, pos, comm, ierr)
  dest = b/k
  call mpi_send (sbuf, pos, MPI_PACKED,
        dest, tag, comm, ierr)
end do
do istep=0,k-1
  call mpi_waitany (k, reqobj, indx, status, ierr)
  pos = (indx-1)*rbufsize/k
  call mpi_unpack (rbuf, rbufsize, pos, b, 1,
        MPI_INTEGER, comm, ierr)
  roffset = r*mod(b,k)
  call mpi_unpack (rbuf, rbufsize, pos, b(roffset),
        nsuperblks, newtype, comm, ierr)
end do
```

Figure 5: Fortran 77 code for increasing the block size of a block-cyclic data distribution by a factor `k` using nonblocking receives. `p` is the number of processes in the communicator `comm`.

### 2.2.2 Communication Schedules

In the synchronized algorithm, some processes have to wait for others before they can receive any data, thereby degrading communication performance. For example, in the $P = 4$, $K = 3$ case, process 3 does not receive its data until the other processes have received all their data. A corollary of this is that there are hot spots in the communication — in the first step of the algorithm the first $\max(K, P)$ processes all send data to process 0, for example. These hot spots can also degrade communication performance. The unsynchronized version avoids excessive synchronization overhead and so is faster than the synchronized version (see Section 3). However, the main drawback of the unsynchronized algorithm is its need for as much buffering as data being redistributed. We have therefore attempted to find variants of the synchronized version that are comparable in performance with the unsynchronized version.

The poorer performance that arises from excessive synchronization and communication hot spots is largely due to the simple way in which the communication in the $K$ stages of the algorithm were scheduled. We shall refer to the global index of the block sent by process $p$ in stage $k$ of the algorithm as the global block send schedule, $B(k, p)$, or the *send schedule* for short. The local index of the block sent by process $p$ in stage $k$ of the algorithm will be referred to as the *local block send schedule*, $b(k, p)$. Similarly, the rank of the process to which process $p$ sends data in stage $k$ will be referred to as the *process send schedule*, $q(k, p)$. The local block and process send schedules can be deduced from the global block send schedule, since $b(k, p) = \lfloor B(k, p)/P \rfloor$ and $q(k, p) = \mathrm{mod}(\lfloor B(k, p)/K \rfloor, P)$. In the algorithms discussed so far process $p$ sends local block $k$ at step $k$, i.e.,

$$B(k, p) = kP + p \tag{2}$$

An easy way to reduce the impact of synchronization and communication hotspots might be to send the $K$ blocks in random order. This is simple to do since the messages are self-describing. At each stage we select at random a local block index from those that have not yet been sent. Then, as before, we evaluate the corresponding global block index and the destination process, and prepend the global block index to the data sent.

Although the random send schedule improves the performance of the synchronized nonblocking receive redistribution routine (see Section 3), we might expect even better performance from a nonrandom schedule that ensures that each process receives data from exactly one process in each of the $K$ communication phases. We shall refer to such a schedule as an optimal schedule since it minimizes the effects of sychronization and communication hot spots. For an optimal schedule, if we view the send schedule $B(k, p)$ as a $K \times P$ matrix then we require the following *permutation conditions* to hold.

1. The rows of the process send schedule, $q(k, p)$, are permutations of the process ranks, i.e., permutations of the numbers $0, 1, \ldots, P - 1$.

2. The $p$th column of $q(k,p)$ is a permutation of the processes to which process $p$ must send data.

We shall now construct an optimal schedule. If $B$ is the global block index and $(p,b)$ and $(q,d)$ are the process and local block indices before and after redistribution, then

$$B = Pb + p = K(Pd + q) + t, \tag{3}$$

where $0 \leq t < K$. Here we regard each block after redistribution as having $K$ "slots" into which blocks of size $r$ are placed. One slot is occupied in each of the $K$ steps of the algorithm. Thus, $t$ in Eq. 3 is the slot index. Equation 3 means that block $b$ in process $p$ is sent to process $q$ where it is stored in slot $t$ of block $d$. Since the communication is the same for each superblock we can, without loss of generality, consider just the first superblock, i.e., $d = 0$, so Eq. 3 becomes,

$$B = Pb + p = Kq + t \tag{4}$$

where $0 \leq b, t < K$ and $0 \leq p, q < P$.

To deduce the optimal schedule we shall first factor out the greatest common factor, $g$, of $P$ and $K$ from Eq. 4. We may write $P = gP'$ and $K = gK'$, where $P'$ and $K'$ are relatively prime. Then we write

$$\begin{array}{rclcrcl} b & = & K'\beta + b', & \quad & q & = & P'\beta + q' \\ p & = & gp' + \alpha, & \quad & t & = & gt' + \alpha \end{array} \tag{5}$$

where $0 \leq \alpha, \beta < g$, $0 \leq b', t' < K'$ and $0 \leq p', q' < P'$. Substituting these expressions for $b$, $p$, $q$, and $t$ into Eq. 4 we obtain

$$B' = P'b' + p' = K'q' + t'. \tag{6}$$

Substituting for $p$ and $t$ in Eq. 4, shows that the set of processes to which a given process sends data is given by $q = \lfloor (P'b + p')/K' \rfloor$ as $b$ takes the values $0, 1, \ldots, K-1$. This means that all processes with the same value of $p'$ send data to the same set of processes. Thus, processes $gp', gp'+1, \ldots, g(p'+1)-1$ all send to the same set of processes, for $p' = 0, 1, \ldots, P'-1$.

We shall refer to Eq. 6 as the *reduced system* and to Eq. 4 as the *full system*. Our aim is to find an optimal send schedule for the reduced system, and to use this to deduce an optimal send schedule for the full system.

Equation 6 has been derived from Eq. 4 by factoring out $g$, and shows the relationship between the global block index, $B'$, and the process number and local block index before and after expanding the block size by a factor of $K'$ for a set of $P'$ processes. For this situation an optimal send solution is given by setting $t' = k'$ in Eq. 6. Thus, in step $k'$ process $p'$ sends global block $B'$ to process $q'$ where it is stored at local block index $k'$. An optimal send schedule

Global Block

| | | | |
|---|---|---|---|
| 0 | 9 | 6 | 3 |
| 4 | 1 | 10 | 7 |
| 8 | 5 | 2 | 11 |

Process

| | | | |
|---|---|---|---|
| 0 | 3 | 2 | 1 |
| 1 | 0 | 3 | 2 |
| 2 | 1 | 0 | 3 |

Local block

| | | | |
|---|---|---|---|
| 0 | 2 | 1 | 0 |
| 1 | 0 | 2 | 1 |
| 2 | 1 | 0 | 2 |

Figure 6: The optimal send schedule for $K' = 3$ and $P' = 4$.

requires us to find for each $k'$ $(0 \le k' < K')$ and $p'$ $(0 \le p' < P')$ the unique global block index $B'$ $(0 \le B' < P'K')$ satisfying

$$B' = q'K' + k' = b'P' + p'. \tag{7}$$

The solutions to this equation give an optimal schedule because they satisfy the permutation conditions stated above (see the Permutation Theorem in Appendix A). As is shown in Appendix A, for each process $p'$ and step of the algorithm $k'$, there is a unique solution $q'$ and $b'$ which gives the destination process and local block index of the data to send. It should be noted that we must work with the reduced system because the analogous equation for the full system, $B = qK + k = bP + p$, does not have unique solutions for all values of $k$ and $p$.

The optimal send schedule satisfying Eq. 7 is obtained by starting at position $(0, 0)$ in the send schedule, and moving moving along the main diagonal, wrapping around in a periodic fashion whenever we move off the edge of the matrix. As we move along the diagonal, we assign the value $B$ to the $B$th entry visited. Numerically the solution to Eq. 7 is found by first finding the integers $x$ and $y$ satisfying $g = xK + yP$ with the extended Euclid algorithm (see, for example, [8]). The solution $(q', b')$ is then given by

$$\begin{aligned} q' &= \lambda x + P'z \\ b' &= -\lambda y + K'z \end{aligned}$$

where $\lambda = p' - k'$ and $z = \lceil \lambda y / K' \rceil = \lceil -\lambda x / P' \rceil$. Figure 6 gives an optimal send schedule for $K' = 3$ and $P' = 4$, together with the corresponding process and local block send schedules.

To generate an optimal send schedule for the full system, $(K, P)$, each of the entries in the optimal send schedule of the reduced system, $(K', P')$, must be expanded into a $g \times g$ block by letting $\alpha$ and $\beta$ take the values $0, 1, \ldots, g-1$. We have,

$$\begin{aligned} B &= Pb + p = P'g(K'\beta + b') + gp' + \alpha = g(P'b' + p') + PK'\beta + \alpha \\ \Rightarrow B &= gB' + PK'\beta + \alpha. \end{aligned} \tag{8}$$

Thus, for each value of the global block index, $B'$, of the reduced system in Fig. 6 we generate a $g \times g$ block of global block indices, $B$, of the full system

$$
\begin{array}{c}
(\alpha,\beta)\\
\boxed{\begin{array}{cccc}
(0,0) & (1,1) & (2,2) & (3,3)\\
(0,3) & (1,0) & (2,1) & (3,2)\\
(0,2) & (1,3) & (2,0) & (3,1)\\
(0,1) & (1,2) & (2,3) & (3,0)
\end{array}}
\end{array}
\Rightarrow
\begin{array}{c}
B\\
\boxed{\begin{array}{cccc}
0 & 49 & 98 & 147\\
144 & 1 & 50 & 99\\
96 & 145 & 2 & 51\\
48 & 97 & 146 & 3
\end{array}}
\end{array}
$$

(a)

$$
\begin{array}{c}
(\alpha,\beta)\\
\boxed{\begin{array}{cccc}
(0,0) & (1,1) & (2,2) & (3,3)\\
(0,1) & (1,2) & (2,3) & (3,0)\\
(0,2) & (1,3) & (2,0) & (3,1)\\
(0,3) & (1,0) & (2,1) & (3,2)
\end{array}}
\end{array}
\Rightarrow
\begin{array}{c}
B\\
\boxed{\begin{array}{cccc}
0 & 49 & 98 & 147\\
48 & 97 & 146 & 3\\
96 & 145 & 2 & 51\\
144 & 1 & 50 & 99
\end{array}}
\end{array}
$$

(b)

Figure 7: Two layouts for generating permutation blocks. On the left are two possible layouts for $(\alpha, \beta)$ for $g = 4$. On the right are the corresponding send schedule permutation blocks for $P = 16$, $K = 12$, and $B' = 0$.

using Eq. 8. Doing this for each entry in the send schedule of the reduced system generates the send schedule of the full system. We will refer to a $g \times g$ block generated in this way as a *permutation block*. A number of different ways of mapping the $(\alpha, \beta)$ values to locations in the block are possible. However, two restrictions must be observed.

1. Since $p = gp' + \alpha$, the value of $\alpha$ must equal the column number of the permutation block. This ensures that the $p$th column of the send schedule refers to process $p$.

2. To ensure that each process sends data to a different destination process in each of the $K$ phases of the algorithm we require that each value of $\beta$ appear exactly once in each row of a permutation block.

Two possible layouts for the $(\alpha, \beta)$ are shown in Fig. 7, together with the corresponding permutation blocks for $P = 16$, $K = 12$, and $B' = 0$.

Once a permutation block has been generated for each entry in the send schedule of the reduced system it is a simple matter to assemble the send schedule of the full system, and to deduce the process and local block send schedules. Figure 8 shows the optimal send schedule for $P = 16$ and $K = 12$. Also shown are the corresponding process and local block send schedules. The layout shown in Fig. 7(a) was used, and will be used throughout the remainder of this paper. For this layout we have

$$\begin{aligned} \alpha &= \text{mod}(p, g) \qquad (9) \\ \beta &= \text{mod}(\alpha - \gamma + g, g) \qquad (10) \end{aligned}$$

where $\gamma = \text{mod}(k, g)$.

### 2.2.3 Redistribution using MPI_SENDRECV

If an optimal schedule is used to send the blocks in a predetermined order, then it is possible for each process to determine from which process it will receive data in each of the $K$ communication phases. In this case messages do not have to be self-describing so there is no need to pack the global block index at the start of each message. If each process knows *a priori* which process it sends to and receives from in each phase, communication can be performed with the routine MPI_SENDRECV, instead of using nonblocking receives. To use MPI_SENDRECV in this way to do the communication we must determine the receive schedules corresponding to the send schedules deduced in Section 2.2.2. The global block receive schedule, $C(k, q)$, or *receive schedule* for short, for some process, $q$, gives the global block index of the data received in each communication phase, $k$. The *process receive schedule*, $p(k, q)$ gives the process from which process $q$ must receive data in each communication phase $k$. The *local block receive schedule*, $t(k, q)$, gives the local slot index at which process $q$ must store the data received in communication phase $k$. The process and local block received schedules can be deduced from the receive schedule since $p(k, q) = \text{mod}(C(k, q), P)$ and $t(k, q) = \text{mod}(C(k, q), K)$.

The receive schedule corresponding to the optimal send schedule given by Eqs. 7, 8 and 10 is

$$C(k, q) = Kq + g\lfloor k/g \rfloor + \text{mod}(\lfloor q/P' \rfloor + \text{mod}(k, g), g) \qquad (11)$$

To show that this is the correct receive schedule for the optimal send schedule deduced in Section 2.2.2 we must prove one of the following two compatibility conditions.

$$\begin{aligned} B(k, \text{mod}(C(k, q), P)) &= C(k, q) \\ C(k, \lfloor B(k, p)/K \rfloor) &= B(k, p) \qquad (12) \end{aligned}$$

These proofs are presented in Appendix B.

Applying Eq. 11 for the case $P = 16$ and $K = 12$ we obtain the receive schedule shown in Fig. 9. An outline of the redistribution routine using MPI_SENDRECV to communicate the data is given in Fig. 10.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 49 | 98 | 147 | 36 | 85 | 134 | 183 | 24 | 73 | 122 | 171 | 12 | 61 | 110 | 159 |
| 1 | 144 | 1 | 50 | 99 | 180 | 37 | 86 | 135 | 168 | 25 | 74 | 123 | 156 | 13 | 62 | 111 |
| 2 | 96 | 145 | 2 | 51 | 132 | 181 | 38 | 87 | 120 | 169 | 26 | 75 | 108 | 157 | 14 | 63 |
| 3 | 48 | 97 | 146 | 3 | 84 | 133 | 182 | 39 | 72 | 121 | 170 | 27 | 60 | 109 | 158 | 15 |
| 4 | 16 | 65 | 114 | 163 | 4 | 53 | 102 | 151 | 40 | 89 | 138 | 187 | 28 | 77 | 126 | 175 |
| 5 | 160 | 17 | 66 | 115 | 148 | 5 | 54 | 103 | 184 | 41 | 90 | 139 | 172 | 29 | 78 | 127 |
| 6 | 112 | 161 | 18 | 67 | 100 | 149 | 6 | 55 | 136 | 185 | 42 | 91 | 124 | 173 | 30 | 79 |
| 7 | 64 | 113 | 162 | 19 | 52 | 101 | 150 | 7 | 88 | 137 | 186 | 43 | 76 | 125 | 174 | 31 |
| 8 | 32 | 81 | 130 | 179 | 20 | 69 | 118 | 167 | 8 | 57 | 106 | 155 | 44 | 93 | 142 | 191 |
| 9 | 176 | 33 | 82 | 131 | 164 | 21 | 70 | 119 | 152 | 9 | 58 | 107 | 188 | 45 | 94 | 143 |
| 10 | 128 | 177 | 34 | 83 | 116 | 165 | 22 | 71 | 104 | 153 | 10 | 59 | 140 | 189 | 46 | 95 |
| 11 | 80 | 129 | 178 | 35 | 68 | 117 | 166 | 23 | 56 | 105 | 154 | 11 | 92 | 141 | 190 | 47 |

(a) Optimal send schedule

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 4 | 8 | 12 | 3 | 7 | 11 | 15 | 2 | 6 | 10 | 14 | 1 | 5 | 9 | 13 |
| 1 | 12 | 0 | 4 | 8 | 15 | 3 | 7 | 11 | 14 | 2 | 6 | 10 | 13 | 1 | 5 | 9 |
| 2 | 8 | 12 | 0 | 4 | 11 | 15 | 3 | 7 | 10 | 14 | 2 | 6 | 9 | 13 | 1 | 5 |
| 3 | 4 | 8 | 12 | 0 | 7 | 11 | 15 | 3 | 6 | 10 | 14 | 2 | 5 | 9 | 13 | 1 |
| 4 | 1 | 5 | 9 | 13 | 0 | 4 | 8 | 12 | 3 | 7 | 11 | 15 | 2 | 6 | 10 | 14 |
| 5 | 13 | 1 | 5 | 9 | 12 | 0 | 4 | 8 | 15 | 3 | 7 | 11 | 14 | 2 | 6 | 10 |
| 6 | 9 | 13 | 1 | 5 | 8 | 12 | 0 | 4 | 11 | 15 | 3 | 7 | 10 | 14 | 2 | 6 |
| 7 | 5 | 9 | 13 | 1 | 4 | 8 | 12 | 0 | 7 | 11 | 15 | 3 | 6 | 10 | 14 | 2 |
| 8 | 2 | 6 | 10 | 14 | 1 | 5 | 9 | 13 | 0 | 4 | 8 | 12 | 3 | 7 | 11 | 15 |
| 9 | 14 | 2 | 6 | 10 | 13 | 1 | 5 | 9 | 12 | 0 | 4 | 8 | 15 | 3 | 7 | 11 |
| 10 | 10 | 14 | 2 | 6 | 9 | 13 | 1 | 5 | 8 | 12 | 0 | 4 | 11 | 15 | 3 | 7 |
| 11 | 6 | 10 | 14 | 2 | 5 | 9 | 13 | 1 | 4 | 8 | 12 | 0 | 7 | 11 | 15 | 3 |

(b) Optimal process send schedule

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 3 | 6 | 9 | 2 | 5 | 8 | 11 | 1 | 4 | 7 | 10 | 0 | 3 | 6 | 9 |
| 1 | 9 | 0 | 3 | 6 | 11 | 2 | 5 | 8 | 10 | 1 | 4 | 7 | 9 | 0 | 3 | 6 |
| 2 | 6 | 9 | 0 | 3 | 8 | 11 | 2 | 5 | 7 | 10 | 1 | 4 | 6 | 9 | 0 | 3 |
| 3 | 3 | 6 | 9 | 0 | 5 | 8 | 11 | 2 | 4 | 7 | 10 | 1 | 3 | 6 | 9 | 0 |
| 4 | 1 | 4 | 7 | 10 | 0 | 3 | 6 | 9 | 2 | 5 | 8 | 11 | 1 | 4 | 7 | 10 |
| 5 | 10 | 1 | 4 | 7 | 9 | 0 | 3 | 6 | 11 | 2 | 5 | 8 | 10 | 1 | 4 | 7 |
| 6 | 7 | 10 | 1 | 4 | 6 | 9 | 0 | 3 | 8 | 11 | 2 | 5 | 7 | 10 | 1 | 4 |
| 7 | 4 | 7 | 10 | 1 | 3 | 6 | 9 | 0 | 5 | 8 | 11 | 2 | 4 | 7 | 10 | 1 |
| 8 | 2 | 5 | 8 | 11 | 1 | 4 | 7 | 10 | 0 | 3 | 6 | 9 | 2 | 5 | 8 | 11 |
| 9 | 11 | 2 | 5 | 8 | 10 | 1 | 4 | 7 | 9 | 0 | 3 | 6 | 11 | 2 | 5 | 8 |
| 10 | 8 | 11 | 2 | 5 | 7 | 10 | 1 | 4 | 6 | 9 | 0 | 3 | 8 | 11 | 2 | 5 |
| 11 | 5 | 8 | 11 | 2 | 4 | 7 | 10 | 1 | 3 | 6 | 9 | 0 | 5 | 8 | 11 | 2 |

(c) Optimal local block send schedule

Figure 8: The optimal send schedule for $P = 16$ processes and an expansion factor of $K = 12$. Entry $(k, p)$ in a table gives (a) the global block index, (b) the destination process, (c) the local block index involved in sending data in step $k$ for process $p$. The arrays are shown divided into $4 \times 4$ permutation blocks.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 12 | 24 | 36 | 49 | 61 | 73 | 85 | 98 | 110 | 122 | 134 | 147 | 159 | 171 | 183 |
| 1 | 1 | 13 | 25 | 37 | 50 | 62 | 74 | 86 | 99 | 111 | 123 | 135 | 144 | 156 | 168 | 180 |
| 2 | 2 | 14 | 26 | 38 | 51 | 63 | 75 | 87 | 96 | 108 | 120 | 132 | 145 | 157 | 169 | 181 |
| 3 | 3 | 15 | 27 | 39 | 48 | 60 | 72 | 84 | 97 | 109 | 121 | 133 | 146 | 158 | 170 | 182 |
| 4 | 4 | 16 | 28 | 40 | 53 | 65 | 77 | 89 | 102 | 114 | 126 | 138 | 151 | 163 | 175 | 187 |
| 5 | 5 | 17 | 29 | 41 | 54 | 66 | 78 | 90 | 103 | 115 | 127 | 139 | 148 | 160 | 172 | 184 |
| 6 | 6 | 18 | 30 | 42 | 55 | 67 | 79 | 91 | 100 | 112 | 124 | 136 | 149 | 161 | 173 | 185 |
| 7 | 7 | 19 | 31 | 43 | 52 | 64 | 76 | 88 | 101 | 113 | 125 | 137 | 150 | 162 | 174 | 186 |
| 8 | 8 | 20 | 32 | 44 | 57 | 69 | 81 | 93 | 106 | 118 | 130 | 142 | 155 | 167 | 179 | 191 |
| 9 | 9 | 21 | 33 | 45 | 58 | 70 | 82 | 94 | 107 | 119 | 131 | 143 | 152 | 164 | 176 | 188 |
| 10 | 10 | 22 | 34 | 46 | 59 | 71 | 83 | 95 | 104 | 116 | 128 | 140 | 153 | 165 | 177 | 189 |
| 11 | 11 | 23 | 35 | 47 | 56 | 68 | 80 | 92 | 105 | 117 | 129 | 141 | 154 | 166 | 178 | 190 |

(a) Receive schedule

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 12 | 8 | 4 | 1 | 13 | 9 | 5 | 2 | 14 | 10 | 6 | 3 | 15 | 11 | 7 |
| 1 | 1 | 13 | 9 | 5 | 2 | 14 | 10 | 6 | 3 | 15 | 11 | 7 | 0 | 12 | 8 | 4 |
| 2 | 2 | 14 | 10 | 6 | 3 | 15 | 11 | 7 | 0 | 12 | 8 | 4 | 1 | 13 | 9 | 5 |
| 3 | 3 | 15 | 11 | 7 | 0 | 12 | 8 | 4 | 1 | 13 | 9 | 5 | 2 | 14 | 10 | 6 |
| 4 | 4 | 0 | 12 | 8 | 5 | 1 | 13 | 9 | 6 | 2 | 14 | 10 | 7 | 3 | 15 | 11 |
| 5 | 5 | 1 | 13 | 9 | 6 | 2 | 14 | 10 | 7 | 3 | 15 | 11 | 4 | 0 | 12 | 8 |
| 6 | 6 | 2 | 14 | 10 | 7 | 3 | 15 | 11 | 4 | 0 | 12 | 8 | 5 | 1 | 13 | 9 |
| 7 | 7 | 3 | 15 | 11 | 4 | 0 | 12 | 8 | 5 | 1 | 13 | 9 | 6 | 2 | 14 | 10 |
| 8 | 8 | 4 | 0 | 12 | 9 | 5 | 1 | 13 | 10 | 6 | 2 | 14 | 11 | 7 | 3 | 15 |
| 9 | 9 | 5 | 1 | 13 | 10 | 6 | 2 | 14 | 11 | 7 | 3 | 15 | 8 | 4 | 0 | 12 |
| 10 | 10 | 6 | 2 | 14 | 11 | 7 | 3 | 15 | 8 | 4 | 0 | 12 | 9 | 5 | 1 | 13 |
| 11 | 11 | 7 | 3 | 15 | 8 | 4 | 0 | 12 | 9 | 5 | 1 | 13 | 10 | 6 | 2 | 14 |

(b) Process receive schedule

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |
| 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 0 | 0 | 0 | 0 |
| 2 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 3 | 3 | 3 | 3 | 3 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 |
| 4 | 4 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 6 | 6 | 6 | 6 | 7 | 7 | 7 | 7 |
| 5 | 5 | 5 | 5 | 5 | 6 | 6 | 6 | 6 | 7 | 7 | 7 | 7 | 4 | 4 | 4 | 4 |
| 6 | 6 | 6 | 6 | 6 | 7 | 7 | 7 | 7 | 4 | 4 | 4 | 4 | 5 | 5 | 5 | 5 |
| 7 | 7 | 7 | 7 | 7 | 4 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 6 | 6 | 6 | 6 |
| 8 | 8 | 8 | 8 | 8 | 9 | 9 | 9 | 9 | 10 | 10 | 10 | 10 | 11 | 11 | 11 | 11 |
| 9 | 9 | 9 | 9 | 9 | 10 | 10 | 10 | 10 | 11 | 11 | 11 | 11 | 8 | 8 | 8 | 8 |
| 10 | 10 | 10 | 10 | 10 | 11 | 11 | 11 | 11 | 8 | 8 | 8 | 8 | 9 | 9 | 9 | 9 |
| 11 | 11 | 11 | 11 | 11 | 8 | 8 | 8 | 8 | 9 | 9 | 9 | 9 | 10 | 10 | 10 | 10 |

(c) Local block receive schedule

Figure 9: The optimal receive schedule for $P = 16$ processes and an expansion factor of $K = 12$. Entry $(k, p)$ in a table gives (a) the global block index, (b) the source process, (c) the local block index involved in receiving data in step $k$ for process $p$. The arrays are shown divided into $4 \times 4$ permutation blocks.

```
create general datatype, newtype

nsuperblks = m/(p*k*r)
do istep=0,k-1

determine process and block schedules,
source, dest, sendoffset, recvoffset

  call mpi_sendrecv (a(sendoffset), nsuperblks, newtype,
    dest, tag, b(recvoffset), nsuperblks, newtype, source,
    tag, comm, status, ierr)
end do
⋮
```

Figure 10: Outline of code for increasing the block size of a block-cyclic distribution by a factor `k` using send and receive schedules and the routine MPI_SENDRECV.

## 3 Performance Results

This section presents results for runs on the IBM SP-1 and the Intel Paragon for the redistribution algorithms described in Section 2.

### 3.1 Results on the IBM SP-1

We ran experiments on the IBM SP system located at Argonne National Laboratory in 1994 and 1995. At that time this machine had 128 Power-1 nodes each with 128 Mbytes of memory connected by both an SP-2 switch and by ethernet. The machine is officially regarded as an SP-2, although it has only Power-1 nodes. However, the implementation of MPI used in our work communicates over the ethernet and not over the fast switch. Thus, our results are more representative of an IBM SP-1 system, and we shall henceforth refer to the machine as such. Version 1.0.7 of the MPICH portable MPI library developed at Argonne National Laboratory and Mississippi State University was used.

The timing results for the IBM SP-1 are shown in Fig. 11 as a function of the expansion factor, $K$, for 3, 10, 16, 32, and 64 processors. Figure (a) shows results for the synchronized and nonsynchronized nonblocking receive version of the redistribution routine. As may be seen from this figure, the unsynchronized version of the routine is up to a factor of three times faster than the synchronized version. However, the main drawback of the unsynchronized algorithm is its need for as much buffering as data being redistributed.

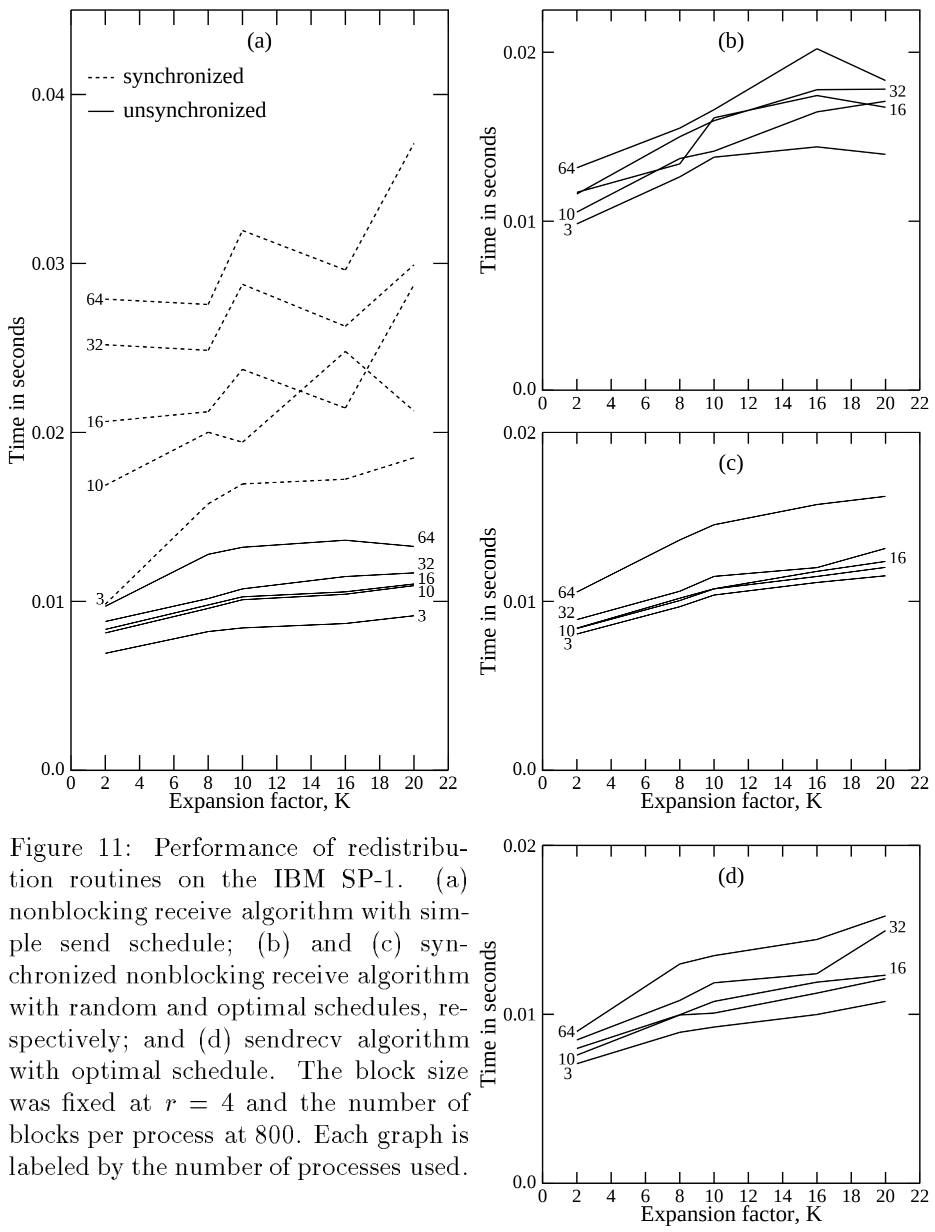


Figure 11: Performance of redistribution routines on the IBM SP-1. (a) nonblocking receive algorithm with simple send schedule; (b) and (c) synchronized nonblocking receive algorithm with random and optimal schedules, respectively; and (d) sendrecv algorithm with optimal schedule. The block size was fixed at $r = 4$ and the number of blocks per process at 800. Each graph is labeled by the number of processes used.

The reason for the poorer performance of the synchronized algorithm is that some processes have to wait for others before they can receive any data. For example, in the $P = 4$, $K = 3$ case process 3 does not receive its data until the other processes have received all their data. A related effect is that there are hot spots in the communication. The timings for the synchronized algorithm vary dramatically as $K$ is changed. We interpret this as variance in processor waiting and communication hot-spotting as $K$ is changed.

Results using a random block send schedule for the synchronized version of the redistribution routine are shown in Fig. 11(b). These results show that even though there is some additional overhead in generating the random sequence, there is a substantial performance improvement compared with results for the simple send block schedule, although the performance is still not quite as good as for the unsynchronized algorithm. The behavior as a function of $K$ is also much smoother, pointing to less processor waiting and hot-spotting due to the randomized schedule.

Another clear effect is the mild rise in redistribution time as $K$ increases. This is simply due to the number of communication phases increasing linearly with $K$. This means that the number of message startups is rising linearly with $K$, even though the total amount of data sent is independent of $K$.

Using an optimal schedule we get the performance results shown in Fig. 11(c). The optimal schedule reduces the timings so they are comparable with the unsynchronized case. This is to be expected since the optimal schedule avoids the synchronization and communication hot spots that degrade the performance for other schedules.

The performance for the version of the redistribution algorithm using the MPI_SENDRECV routine is shown in Fig. 11(d). Comparison with Fig. 11(a) shows that the version using MPI_SENDRECV is comparable in performance with the unsynchronized version using nonblocking receives.

## 3.2 Results on the Intel Paragon

The Intel Paragon used is located at Oak Ridge National Laboratory. It has 66 nodes, each with 32 Mbytes of memory, connected in a two-dimensional mesh. As in the runs on the IBM SP-1, version 1.0.7 of the MPICH portable MPI library was used. Timing results are shown in Fig. 12.

We give an interpretation similar to that for the IBM SP-1. The simple, synchronized algorithm is slow and exhibits fluctuating timings due to processor waiting and communication hot-spotting. Randomizing the send schedule smooths and improves the results, as shown in Fig. 12(b). Using the optimal send schedule (but still within the non-blocking receive algorithm) gives the good results shown in Fig. 12(c). Finally, using MPI_SENDRECV in place of separated sends and receives, with the optimal send and receive schedules, leads to the best performance results shown in Fig. 12(d).

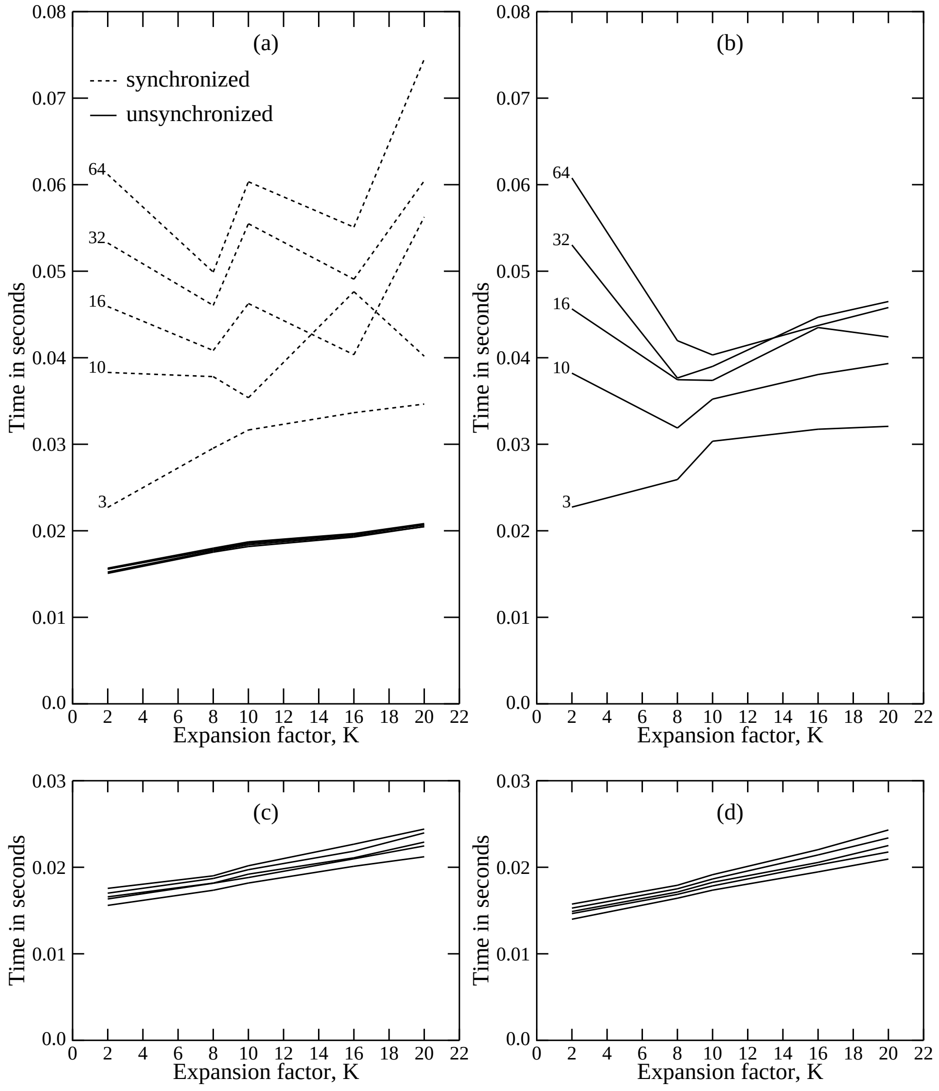


Figure 12: Same as for Fig. 11 but for the Intel Paragon.

### 3.3 Comparison with other Collective Communication Operations

In this section, we compare the performance of the redistribution routines with that of MPI collective communication routines. We wish to know how redistribution times compare to other bulk data movement operations. In one case, a single process broadcasts data to all other processes by calling the routine MPI_BCAST. In the second case each process broadcasts data to all other processes by calling the routine MPI_ALLTOALL. The all-to-all operation is similar to the communication performed in the redistribution operation. Timing results for the Intel Paragon and IBM SP-1 are presented in Table 1. For the broadcast case the number of elements broadcast by the root equals the number sent by each process in the redistributions shown in Figs. 11 and 12, i.e., 3200 integers. For the all-to-all case the amount of data communicated between each pair of processes is the same as in the redistributions.

As might be expected, broadcast is faster than the all-to-all and redistribution operations. This is because the broadcast sends fewer messages so the likelihood of network congestion is less, and communication latency is reduced. Comparing the redistribution and all-to-all operations, we find that for a small number of processors (3 and 10), redistribution is at least three times slower. For more processors, the difference becomes smaller, and redistribution becomes faster than the all-to-all case for $K = 2$ on 64 processors. This is due to the fact that not every possible pair of processors communicates in the redistribution algorithm (as long as $K < P$), while they do in all-to-all. That is, all-to-all has more messages for $P$ large.

## 4 A More General Redistribution Algorithm

In general, a block-cyclic data redistribution may involve

1. varying the block size;
2. changing the topology of the process group from one Cartesian topology to another. For example, from a $4 \times 3$ process layout to a $2 \times 6$ process layout;
3. redistributing from one process group to another, where the number of processes in the groups may differ.

In this paper we have restricted our attention to the case in which the block size is increased by a factor of $K$ from $r$ to $Kr$, and the processes involved remain fixed in number and identity. The algorithms presented in Section 2 may be readily modified to handle the case in which the block size decreases by a factor of $K$ from $Kr$ to $r$. To do this we just run the redistribution algorithm "in reverse." That is, in each of the $K$ communication phases we swap the source

| | | 3 | 10 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|
| Paragon | bcast | 1.2 | 2.7 | 3.2 | 4.3 | 5.0 |
| | alltoall | 2.9 | 5.9 | 7.9 | 12.2 | 22.9 |
| | redist (2) | 14.0 | 14.6 | 14.9 | 15.3 | 15.7 |
| | redist (20) | 20.9 | 21.8 | 22.5 | 23.4 | 24.3 |
| IBM SP-1 | bcast | 1.8 | 3.6 | 4.6 | 5.4 | 7.2 |
| | alltoall | 2.4 | 4.4 | 5.0 | 8.1 | 15.2 |
| | redist (2) | 7.1 | 7.6 | 8.0 | 8.5 | 9.0 |
| | redist (20) | 10.8 | 12.1 | 12.3 | 15.0 | 15.8 |

Table 1: Timings in milliseconds for broadcast and all-to-all operations on the Intel Paragon and IBM SP-1 using the MPI_BCAST and MPI_ALLTOALL routines, respectively. Also shown are the timings for the sendrecv version of the redistribution algorithm for the cases $K = 2$ and $K = 20$. In the broadcast case one process broadcasts 3200 integers to all processes. In the all-to-all case each process broadcasts $\lfloor 3200/N_p \rfloor$ integers to each process, where $N_p$ is the number of processes. For the redistributions each process contains 800 blocks of block size $r = 4$.

and destination processes, and the block offsets from/into which the blocks are sent/received.

The general case of transforming from block size $r_1$ to $r_2$ may then be handled in two phases. Denoting the least common multiple of $r_1$ and $r_2$ by $\ell = K_1 r_1 = K_2 r_2$, then the block size is first expanded by the factor $K_1$ and then shrunk by the factor $K_2$ (or *vice versa*).

Changing the topology of the process group, or redistributing to another process group, leads to another class of algorithms not considered in this paper.

Our redistribution algorithms can be extended to handle multi-dimensional arrays distributed across processes arranged with a virtual topology with the same number of dimensions and the same number of processors in each dimension. To do this we simply apply the one-dimensional algorithm to each direction in turn. Thus, to expand the block size of a two-dimensional array by $K_1$ in one direction and $K_2$ in the other would require $K_1 + K_2$ communication phases, and would communicate all of the data twice. Another approach would be to generalize the the one-dimensional algorithm to a genuine two-dimensional one. In this case the data would need to be moved only once, but a naive approach would require $K_1 K_2$ communication phases, possibly resulting in excessive communication startup costs. One techique for reducing this cost would be to conglomerate data being sent from one process to another into fewer messages, trading off buffer space and startup cost. These issues will be investigated in future research.

In redistributing multi-dimensional arrays using successive applications of our one-dimensional algorithm we can take advantage of MPI's ability to create and manipulate subgroups of processes since it will be necessary to form communicators whose associated groups are the rows and columns of the virtual topology. It is these communicators that are used in the one-dimensional redistributions. In the multi-dimensional case the blocks communicated will also be multi-dimensional. The datatype for the blocks can be created in a straightforward way using MPI's datatype constructor routines. As an example, consider the case of a two-dimensional array distributed over a mesh of $P \times Q$ processes. Suppose we want to change the block size from $r_1 \times c_1$ to $r_2 \times c_2$. We first apply a one-dimensional redistribution along the rows of the virtual topology which changes the block size to $r_1 \times c_2$. Then we apply a second one-dimensional redistribution along the columns of the topology which changes the block size to $r_2 \times c_2$.

## 5 Summary

We have shown how communication schedules can be constructed that permit the efficient redistribution of block-cyclic data distributuions. These schedules can be used to implement a version of the redistribution algorithm using the MPI routine MPI_SENDRECV that requires no extra buffering and so is more economical in its use of memory than the unsynchronized version using non-blocking receives. It is also comparable in performance. The ultimate objective of this research is to develop efficient redistribution algorithms for use in conjunction with the ScaLAPACK library. Thus, future research will investigate extensions of the work presented here to two-dimensional arrays. It should be noted that the MPI_ALLTOALLS routine, which has been proposed as a routine in the MPI-2 extensions, would permit the block size to be increased or decreased by an integer factor with just one MPI call. If this routine becomes part of the official MPI specification it will be interesting to compare its performance with the results obtained with the routines used in this work.

## Acknowledgments

The authors acknowledge the use of the Intel Paragon XP/S 5 computer, located in the Oak Ridge National Laboratory Center for Computational Sciences (CCS), funded by the Department of Energy's Mathematical, Information, and Computational Sciences (MICS) Division of the Office of Computational and Technology Research (OCTR). The authors also acknowledge use of the IBM SP system at Argonne National Laboratory. The authors are grateful for access to these systems.

## Appendix A

In this appendix key proofs of the properties of the optimal send schedule are given. These proofs are all concerned with the equation

$$B = qK + k = bP + p \tag{13}$$

where $0 \leq B < PK$, $0 \leq q, p < P$ and $0 \leq k, b < K$. Values of $B$, $q$, $k$, $b$, and $p$ that satisfy these constraints will be referred to as *valid* values.

We first show that if, and only if, $K$ and $P$ are relatively prime, then, given any combination of valid values of $k$ and $p$, if a valid solution $(q, b)$ exists it must be unique. Then, by showing that there is a one-to-one correspondence between valid values of $B$ and valid combinations of $k$ and $p$, we show that a valid solution exists for each valid $(k, p)$. Finally, we show that for a given value of $k$, different values of $p$ result in a solution with different values of $q$.

**Uniqueness Theorem**
Given $(k, p)$ and a solution $(q, b)$ to Eq. 13, then this solution is unique if and only if $K$ and $P$ are relatively prime.
**Proof**
Suppose the solution is not unique and that $K$ and $P$ are relatively prime, i.e., we can find $(q_1, b_1)$ and $(q_2, b_2)$ such that

$$\begin{aligned} B_1 &= q_1K + k = b_1P + p \\ B_2 &= q_2K + k = b_2P + p \end{aligned}$$

Subtracting we get $(q_2 - q_1)K = (b_2 - b_1)P$. If $K$ and $P$ are relatively prime then $(q_2 - q_1)$ must be divisible by $P$. But $0 \leq |q_2 - q_1| < P$, so we must have $q_1 = q_2$, and hence also $b_1 = b_2$ and $B_1 = B_2$ . Thus, the solution $(q, b)$ is unique.

Next we assume the solution is unique and show that $K$ and $P$ are relatively prime. Suppose that $K$ and $P$ are not relatively prime, i.e., $K = gK'$ and $P = gP'$ with $g > 1$. Then if $(q, b)$ is a solution for given $(k, p)$ then so is $(\text{mod}(q + xP', P), \text{mod}(b + xK', K))$ for $x = 0, 1, \ldots, g - 1$. This contradicts our assumption of uniqueness, so $K$ and $P$ must be relatively prime.

**Existence Theorem**
A valid solution $(q, b)$ to Eq. 13 exists for all valid $(k, p)$ if, and only if, $K$ and $P$ are relatively prime.
**Proof**
It is clear that for any valid value of $B$ we can find valid values for $q$, $k$, $b$, and $p$ satisfying Eq. 13. There are $PK$ valid values of $B$, and $PK$ corresponding

solution sets $(q, k, b, p)$. Furthermore, for each $B$ there is a unique $(k, p)$. To show this assume that the $(k, p)$ is not unique for some $B$. Then,

$$\begin{aligned} B &= q_1K + k_1 = b_1P + p_1 \\ B &= q_2K + k_2 = b_2P + p_2. \end{aligned}$$

Subtracting we get $(q_2 - q_1)K - (k_2 - k_1) = 0$. Then $(k_2 - k_1)$ must be divisible by $K$. But $0 \le |k_2 - k_1| < K$, so $(k_2 - k_1)$ can only be divisible by $K$ if $k_2 = k_1$. We also have $(b_2 - b_1)P - (p_2 - p_1) = 0$ from which it follows, by similar reasoning, that $p_2 = p_1$.

We already know from the Uniqueness Theorem that for each of the values of $(k, p)$ corresponding to a valid $B$, the value of $B$ for that $(k, p)$ is unique, if and only if $K$ and $P$ are relatively prime. It therefore follows that, if and only if $K$ and $P$ are relatively prime, there is a one-to-one corresponence between the $PK$ valid values of $B$ and the $PK$ valid combinations of $k$ and $p$. Hence, for each valid $(k, p)$ a solution to Eq. 13 exists if and only if $K$ and $P$ are relatively prime.

**Permutation Theorem**

For each valid $k$, different value of $p$ results in a solution to Eq. 13 with different values of $q$.

**Proof**

Assume the converse, i.e., that there exist $(b_1, p_1)$ and $(b_2, p_2)$ such that

$$\begin{aligned} qK + k &= b_1P + p_1 \\ qK + k &= b_2P + p_2. \end{aligned}$$

Then $(b_2 - b_1)P - (p_2 - p_1) = 0$ so $(p_2 - p_1)$ must be divisible by $P$. But $0 \le |p_2 - p_1| < P$, so $(p_2 - p_1)$ can only be divisible by $P$ if $p_1 = p_2$.

## Appendix B

In this Appendix the compatibility conditions are proven for the send and receive schedules used.

**Compatibility Theorem I**

Let

$$B(k, p) = gB'(k', p') + PK'\beta + \alpha$$

where

$$\begin{aligned} \alpha &= \mathrm{mod}(p, g), \\ \beta &= \mathrm{mod}(\alpha - \gamma + g, g), \\ \gamma &= \mathrm{mod}(k, g), \end{aligned}$$

$p' = \lfloor p/g \rfloor$, $k' = \lfloor k/g \rfloor$, and $B'(k', p')$ is the unique solution of

$$B' = q'K' + k' = b'P' + p'.$$

Furthmore, let

$$C(k, q) = Kq + g\lfloor k/g \rfloor + \mathrm{mod}(\lfloor q/P' \rfloor + \mathrm{mod}(k, g), g),$$

where $q = \beta P' + q'$, then

$$B(k, \mathrm{mod}(C(k, q), P)) = C(k, q),$$

and so $B(k, p)$ and $C(k, q)$ are compatible send and receive schedules.
**Proof**
Let

$$p = \mathrm{mod}(Kq + g\lfloor k/g \rfloor + \mathrm{mod}(\lfloor q/P' \rfloor + \mathrm{mod}(k, g), g), P)$$

First we show that

$$\alpha = \mathrm{mod}(p, g) = \mathrm{mod}(\lfloor q/P' \rfloor + \mathrm{mod}(k, g), g).$$

This follows from the fact that we can write

$$Kq + g\lfloor k/g \rfloor + \mathrm{mod}(\lfloor q/P' \rfloor + \mathrm{mod}(k, g), g) = bP + p$$

and so

$$\begin{aligned} p &= (K'q + \lfloor k/g \rfloor - P'b)g + \mathrm{mod}(\lfloor q/P' \rfloor + \mathrm{mod}(k, g), g) \\ \Rightarrow \alpha &= \mathrm{mod}(p, g) = \mathrm{mod}(\lfloor q/P' \rfloor + \mathrm{mod}(k, g), g). \end{aligned}$$

Using this value for $\alpha$ we then have

$$\begin{aligned} B(k, \mathrm{mod}(C(k, q), P)) &= gB'(k', p') + gP'K'\beta + \mathrm{mod}(\lfloor q/P' \rfloor + \mathrm{mod}(k, g), g) \\ &= g(q'K' + k') + gP'K'\beta + \mathrm{mod}(\lfloor q/P' \rfloor + \mathrm{mod}(k, g), g) \\ &= gK'(q' + P'\beta) + g\lfloor k/g \rfloor + \mathrm{mod}(\lfloor q/P' \rfloor + \mathrm{mod}(k, g), g) \end{aligned}$$

But $q = q' + P'\beta$ so

$$\begin{aligned} B(k, \mathrm{mod}(C(k, q), P)) &= Kq + g\lfloor k/g \rfloor + \mathrm{mod}(\lfloor q/P' \rfloor + \mathrm{mod}(k, g), g) \\ &= C(k, q) \end{aligned}$$

**Compatibility Theorem II**

$$C(k, \lfloor B(k, p)/K \rfloor) = B(k, p)$$

where all the definitions given in the statement of Compatibility Theorem I apply.

**Proof**
First consider

$$\begin{aligned}
&\mathrm{mod}(\lfloor\lfloor B(k,p)/K\rfloor/P'\rfloor+\mathrm{mod}(k,g),g)\\
&\qquad=\mathrm{mod}(\lfloor(gB'(k',p')+\beta P'K+\alpha)/(P'K)\rfloor+\mathrm{mod}(k,g),g)\\
&\qquad=\mathrm{mod}(\lfloor(gB'(p',k')+\alpha)/(P'K)\rfloor+\beta+\mathrm{mod}(k,g),g)\\
&\qquad=\mathrm{mod}(\beta+\mathrm{mod}(k,g),g)\\
&\qquad=\mathrm{mod}(\mathrm{mod}(\alpha-\mathrm{mod}(k,g)+g,g)+\mathrm{mod}(k,g),g)\\
&\qquad=\alpha
\end{aligned}$$

Then we have

$$\begin{aligned}
C(k,\lfloor B(k,p)/K\rfloor) &= K\lfloor B(k,p)/K\rfloor+g\lfloor k/g\rfloor+\alpha\\
&= K\lfloor B'(k',p')/K\rfloor+g\lfloor k/g\rfloor+\beta P'K+\alpha\\
&= gK'\lfloor(K'q'+k')/K'\rfloor+g\lfloor k/g\rfloor+\beta P'K+\alpha\\
&= g(K'q'+k')+\beta P'K+\alpha = B'(k',p')+\beta P'K+\alpha\\
&= B(k,p)
\end{aligned}$$

## References

[1] 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. *Journal of Parallel and Distributed Computing*, 26:72–84, 1995.

[2] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Computer Society Press, 1992.

[3] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra on high performance computers. *SIAM Review*, 37(2):151–180, June 1995.

[4] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994. Special issue on MPI.

[5] S. Gupta, S. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayyan. On the generation of efficient data communications for for distributed memory machines. In *Proceedings of the 1992 International Computer Symposium*, pages 504–513, December 1992.

[6] E. T. Kalns and L. M. Ni. Darel: A portable data redistribution library for distributed memory machines. In *Proceedings of the 1994 Scalable Parallel Libraries Conference*, October 1994.

[7] E. T. Kalns and L. M. Ni. Processor mapping techniques toward efficient data redistribution. *IEEE Trans. Parallel and Dist. Systems*, 6(12):1234–1247, December 1995.

[8] D. E. Knuth. *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*. Addison-Wesley Publishing Company, 1969.

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

[10] S. Ramaswamy and P. Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *Proceedings of Frontiers '95: the Fifth Syposium on the Frontiers of Massively Parallel Computation*, pages 342–349, February 1995.

[11] R. Thakur, A. Choudhary, and G. Fox. Runtime array redistribution in hpf programs. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pages 309–316. IEEE Computer Society Press, 1994.