Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers *

Shankar Ramaswamy and Prithviraj Banerjee

Center for Reliable and High Performance Computing
Computer Systems and Research Laboratory
1308 West Main St.
Urbana, IL 61801

E-mail : shankar,banerjee@crhc.uiuc.edu
Tel : (217)333-6564
Fax : (217)244-5685

Abstract

Appropriate data distribution has been found to be critical for obtaining good performance on Distributed Memory Multicomputers like the CM-5, Intel Paragon and IBM SP-1. It has also been found that some programs need to change their distributions during execution for better performance (redistribution). This work focuses on automatically generating efficient routines for redistribution. We present a new mathematical representation for regular distributions called PITFALLS and then discuss algorithms for redistribution based on this representation. One of the significant contributions of this work is being able to handle arbitrary source and target processor sets while performing redistribution. Another important contribution is the ability to handle an arbitrary number of dimensions for the array involved in the redistribution in a scalable manner. Our implementation of these techniques is based on an MPI-like communication library. The results presented show the low overheads for our redistribution algorithm as compared to naive runtime methods.

*This research was supported in part by the Office of Naval Research under Contract N00014-91J-1096, and in part by the National Aeronautics and Space Administration under Contract NASA NAG 1-613.

1 Introduction

1.1 Motivation for Data Redistribution

Distributed Memory Multicomputers such as the Intel Paragon, IBM SP-1 and the Connection Machine CM-5 offer significant advantages over shared memory multiprocessors in terms of cost and scalability. Unfortunately, to extract all that computational power from these machines, users have to write efficient software for them, which is an extremely laborious process. The PARADIGM compiler project at the University of Illinois is aimed at automatically generating a parallel FORTRAN program for any Distributed Memory Multicomputer given an input FORTRAN77 program. The fully implemented PARADIGM compiler will automatically:

- Determine a good data partitioning scheme for the input program [1, 2, 3, 4]
- Use the data partition determined (or user provided data distribution directives) to partition computation between the processors of the system and generate the required communication routines [5, 6, 7]
- Detect available functional and data parallelism and use this information to make program execution efficient [8, 9]
- Provide compiler and runtime support for irregular computations [10]

One of the major aspects of programming/compiling for Distributed Memory Multicomputers has been data distribution. A good distribution of data can eliminate a lot of unnecessary communication and thus provide good speedups. There have been many efforts on developing automatic data partitioning techniques [1, 3, 11, 12, 13]. In addition, user provided constructs have been proposed in some form or the other in every FORTRAN dialect for Multicomputers including FORTRAN D and HPF [14, 15]. Recently, the HPF standard has been widely adopted in industry and academia for specifying data distributions. HPF also provides directives for data redistribution dynamically during program execution. In this work, we will consider only "regular" distributions along each array dimension, i.e., one of – ALL, $BLOCK$, $CYCLIC$ or $BLOCKCYCLIC(x)$. What we mean by these terms is made clearer in Figure 1. Considering only these distributions is reasonable because a large number of scientific programs have been found to use such distributions.

In many programs, the distribution of an array needs to be changed for different phases of a program in order to achieve good performance. For instance, a 2D FFT routine comprises

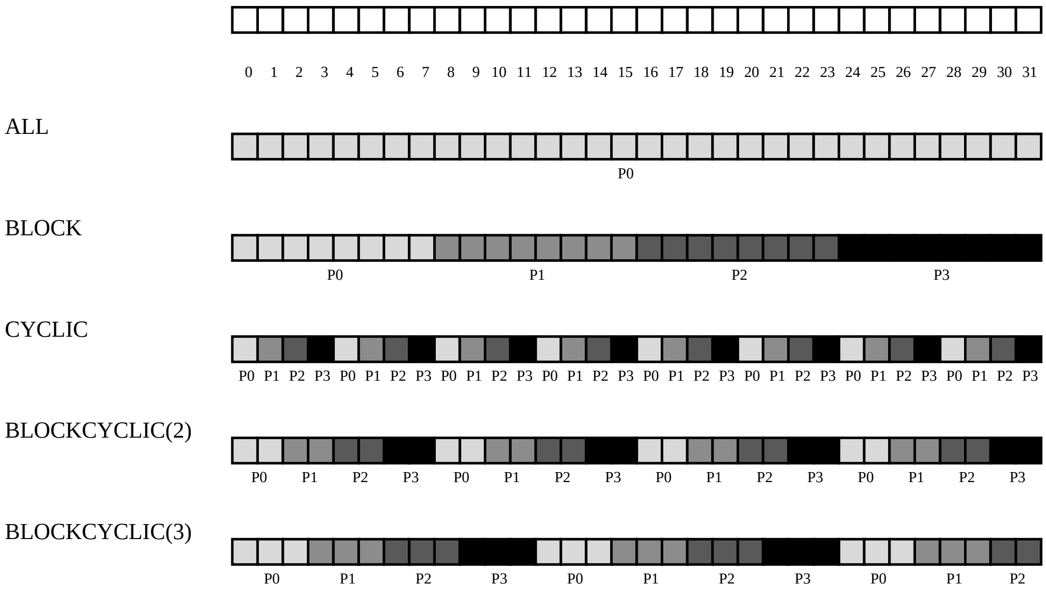

Figure 1: Examples of Regular Distributions

a sequence of two 1D FFT operations on the input matrix. First, 1D FFTs are computed along each row, this is followed by a 1D FFTs being computed along each column. The best distribution for the first phase would be $BLOCK$ or $CYCLIC$ along the row dimension. On the other hand, the second phase would be best performed using a $BLOCK$ or $CYCLIC$ distribution along the column dimension. It must be noted that these distributions would result in zero communication within each phase but require a redistribution between phases. This redistribution can be avoided by distributing the array along both dimensions, however, this will cause considerable communication within each phase. The work in [13] shows that the performance of a 2D FFT on an iPSC/860 is best when redistribution is used.

Redistribution of data is very critical for Multiple Program Multiple Data (MPMD) programming. In such programs different subsystems of a given processor system execute different parts of a program. This is in contrast to the popular Single Program Multiple Data type of programs where essentially all processors are executing the same program, but on different data sets. MPMD programs can potentially execute faster than SPMD programs by making the execution more efficient. Frequently, data dependence constraints in MPMD programs require arrays be redistributed from one subsystem to another. Figure 2 shows this clearly. Here, array A is being written into by the first set of processors and being read by the second set of processors. The PARADIGM compiler effort is among the first to consider the problem of automatic MPMD program generation. This has been one of the

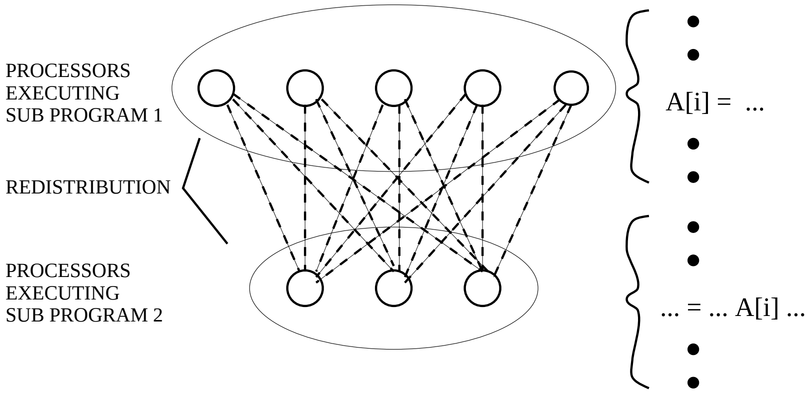

Figure 2: Need for Redistribution in MPMD Programs

primary motivations for the work presented in this paper. More details on MPMD programs can be found in [8, 9, 16, 17, 18, 19].

1.2 The Data Redistribution Problem

Having motivated the need for redistribution, we can now formally define a redistribution R to be the set of routines that – given an n-dimensional array A on a set of source processors P_s with source distribution D_s, transfer all the elements of the array to a set of target processors P_t with a target distribution D_t. In the general case, D_s and D_t can specify arbitrary data distributions along each dimension of the array. However, as mentioned before, we only handle regular distributions at this point. Therefore, a redistribution routine needs to figure out exactly what data needs to be sent(received) by each source(target) processor. It is possible to use a simple runtime resolution approach for redistribution. In this approach, each source processor computes the index of each of the elements it owns based on the source distribution; uses this index to compute the target processor for it based on the target distribution and packs it into a buffer meant for that processor; sends the contents of its buffers to the target processors. The target processors essentially do the reverse. However, as we shall see later, this approach is very costly compared to a method like ours which makes use of the distribution information in a more intelligent manner.

The important features of our redistribution method are:

- Redistribution routines are to be automatically generated at compile time. This work will be part of the PARADIGM compiler support for MPMD program generation.
- The source and target processor sets can be any arbitrary subset of the given processor

system. The motivation for this comes from MPMD programs. For such programs we must be able to handle a redistribution for example, from processors $0, 3, 4, 6$ to processors $1, 2$. We use ideas similar to those used in MPI [20] to handle arbitrary processor sets.

- Arrays being redistributed can have any of the possible regular distributions on the source and target processor sets. Note that one of the most general case of redistribution is $BLOCKCYCLIC(m)$ to $BLOCKCYCLIC(n)$, where m and n are relatively prime. We handle all the types of distributions in an uniform manner.
- Arrays being redistributed can have an arbitrary number of dimensions. The complexity of our algorithms scales linearly with an increase in the number of dimensions.
- We handle multiple arrays being redistributed at the same time using message aggregation [5, 6, 14]. This means we have just one send-recv between a pair of processors even if there is data from more than one array being communicated between them.

Our redistribution techniques rely on a mathematical representation for regular distributions which we call PITFALLS (Processor Index Tagged FAmiLy of Line Segments). Although many representations exist for regular data distributions on distributed memory machines [14, 21, 22, 6], we felt the need for a new representation for our work because none of the previous representations satisfied our requirements. A primary requirement for us was to be able to mathematically represent a regular distribution on any given subset of processors of the given system and not on the entire system as all the current representations do. In addition, we needed to be able to perform redistribution communication analysis efficiently. We therefore developed the PITFALLS representation along with a redistribution algorithm based on it. This is discussed in the next section. We have also included a brief explanation of our implementation and provided the results of a comparison of our method with a runtime resolution type of method in Section 3. Finally, we discuss the implications of our work and future extensions.

1.3 Related Work

It is possible for one to argue that redistribution can be performed using multicomputer compiler techniques such as those outlined in [5, 6, 23, 14, 21, 22]. These techniques generate the communication required for any program statement to execute correctly given the distributions of the arrays involved in the statement. One could now use a statement of the form $B = A$ where B is distributed according to the target distribution and A is distributed according to the source distribution. However, it is not clear that any of the currently implemented compilers have efficient techniques to handle all the regular distributions possible for A and B. Another major obstacle in trying to use current compiler techniques is that they cannot handle the given statement if A and B are distributed on a distinct (possibly overlapping) set of processors. On the other hand, the possibility of handling arbitrary program statements using techniques similar to the ones used in this work are being considered for the PARADIGM compiler. The reason for this is the simplicity and practicality of our techniques (the algorithms given in this paper have all been implemented).

The work by Agarwal et. al. [24] provides runtime support for redistributions. They construct schedules for redistribution at runtime and reuse schedules if a particular redistribution pattern occurs more than once. However, this work neither handles the $CYCLIC$ or $BLOCKCYCLIC$ type of distributions nor does it consider arbitrary source and target processor sets.

Recent work by Thakur et. al. [25] considers redistributions of regular arrays in detail. This work is implemented in the form of a library for a HPF compiler. The methods proposed treat possible source-target distributions in a pairwise manner. Their general approach is to use a runtime resolution approach such as the one described before; although, for specific cases of source-target data distributions, they use efficient methods. For the multi-dimensional case, [25] propose a solution which is very expensive; such redistributions are considered to be composed of a series of one-dimensional redistributions. Figure 3 illustrates the difference of their approach from ours. In this example, the basic problem is the redistribution of a two-dimensional array A from $(BLOCK, ALL)$ to $(ALL, BLOCK)$. The approach of [25] would be to carry it out as a set of two redistributions - first, from $(BLOCK, ALL)$ to (ALL, ALL) and then, from (ALL, ALL) to $(ALL, BLOCK)$. On the other hand, our approach would be to directly go from $(BLOCK, ALL)$ to $(ALL, BLOCK)$ as shown.

2 The PITFALLS Representation and Redistribution

Broadly, to perform a redistribution, for any source-target processor pair, one has to look at the set of elements owned by the source processor before redistribution (based on the source distribution) and the set of elements owned by the target processor after redistribution (based on the target distribution). The intersection of these sets is the data that needs to be

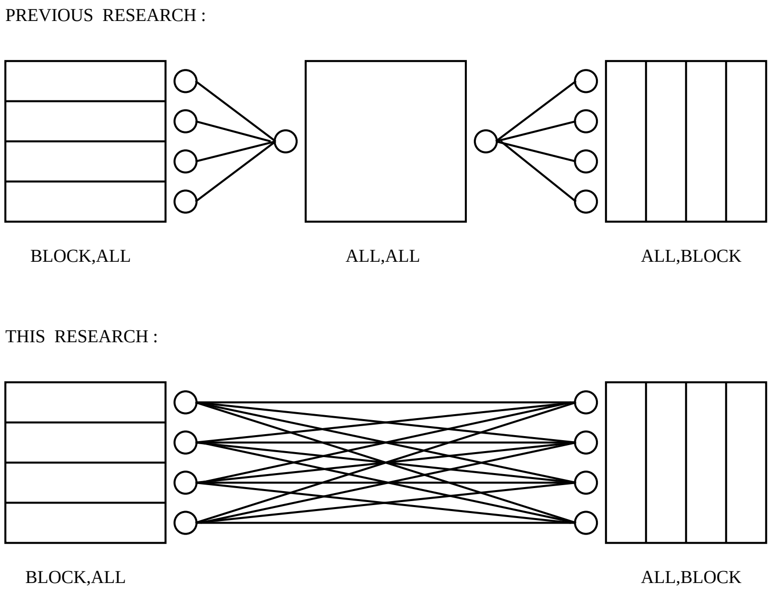

Figure 3: Need for Redistribution in MPMD Programs

transferred between the pair of source-target processors. The PITFALLS representation is particularly useful in this context as it can be used to easily determine the two things most important for redistribution – which pairs of source-target processors need to communicate and the intersection of the data sets of such a pair of source-target processors.

For simplicity, we first develop the PITFALLS representation for regular distributions of a one-dimensional array in a set by step manner. We will later extend our ideas to multiple dimensions.

2.1 Line Segments (LS)

Consider a one-dimensional array A of size n. Fundamental to the PITFALLS representation is the idea of using Line Segments (LS) to represent a contiguous block of elements. An LS L can be represented by a pair of numbers (l, r). For our representation, this LS (in the context of array A) is taken to mean the block of elements of A with indices starting at l and ending at r (numbering $r - l + 1$). We call the quantity l as the LOW of L; i.e., $l = LOW(L)$. Similarly, r is called the $HIGH$ of L ($r = HIGH(L)$). Note that a single element with index l has the LS representation (l, l).

Since our primary interest is in being able to find intersections for sets of elements, we see that the intersection of two LS's $L_1 = (l_1, r_1)$ and $L_2 = (l_2, r_2)$ (denoted by $LI_{L_1 \bigcap L_2}$) is given by:

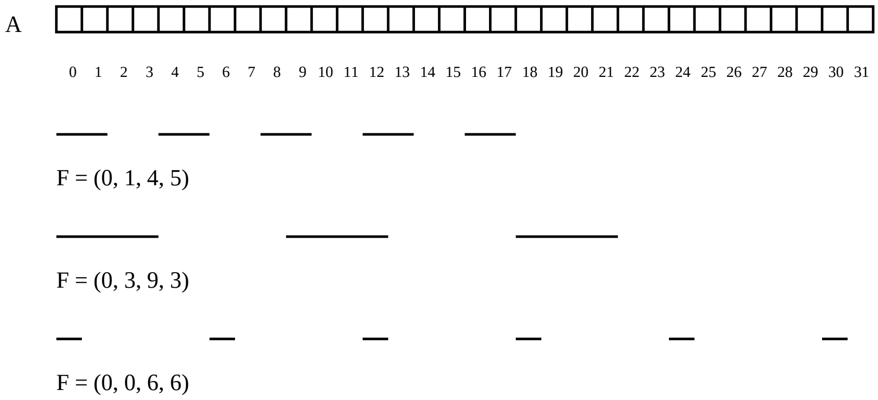

Figure 4: Examples of FALLS

$$LI_{L_1 \bigcap L_2} = \begin{cases} (\max(l_1, l_2), \min(r_1, r_2)) & if\ max(l_1, l_2) \leq min(r_1, r_2) \\ \emptyset & otherwise \end{cases}$$

2.2 FAmiLy of Line Segments (FALLS)

The LS notion can be extended to what we call a FAmiLy of Line Segments (FALLS). A FALLS F can be represented by a tuple (l, r, str, num). Intuitively, F represents a set of num equally spaced, equally sized blocks of elements; the first block starts at l and ends at r; the stride between successive l's is str. Note that these are non-overlapping blocks. The ith $(0 \leq i \leq num - 1)$ LS of F (denoted by L^i and called the ith member) is given by:

$$L^i = (l + i \times str, r + i \times str)$$

Figure 4 shows a few examples of FALLS.

Using the notion of FALLS, it is possible to represent the set of elements of A owned by a particular processor under any regular distribution. Figure 5 shows the FALLS representation for elements owned by processor 1 in a 4-processor system for various distributions of A when $n = 32$. In this example, it turns out that in every case, processor 1's elements can be represented using a single FALLS. This may not be true in the general case, where, more than one FALLS may be needed. However, it is easy to show that no more than two FALLS are needed for any regular distribution. In Figure 6, we show an example of processor 2 needing two FALLS when a array of size 32 is distributed using $BLOCKCYLIC(3)$.

Once again, in the context of redistribution, computing the intersection of two FALLS is of interest to us. Given two FALLS $F_1 = (l_1, r_1, str_1, num_1)$ and $F_2 = (l_2, r_2, str_2, num_2)$, a

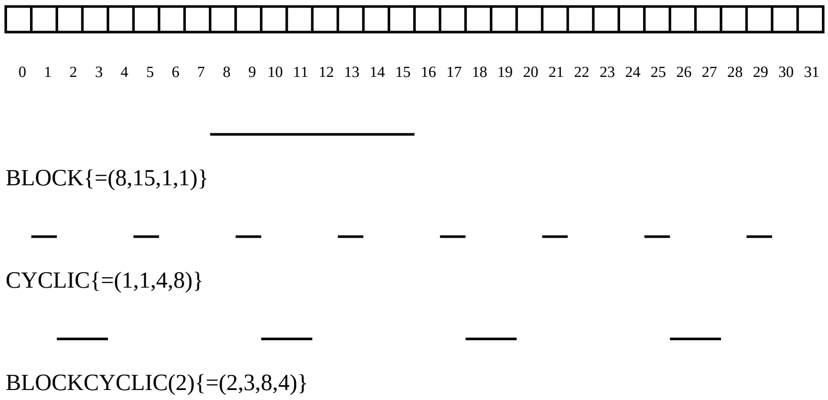

Figure 5: Examples of FALLS Describing Regular Distributions

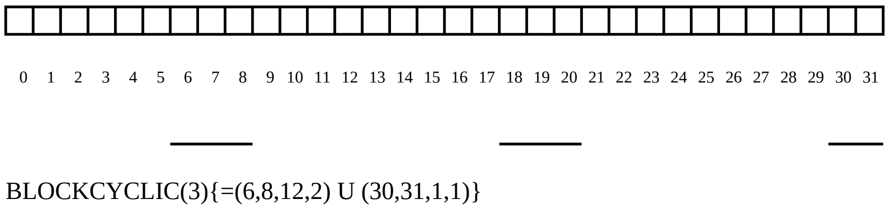

Figure 6: Examples of Distribution Requiring Multiple FALLS

simple brute force algorithm to compute the intersection ($FI_{F_1 \bigcap F_2}$) is shown in Figure 7. This brute force approach just considers every possible pair of members from the two FALLS and applies the LS intersection algorithm to them. We can see that this technique can be very inefficient by considering the example of FALLS intersection shown in Figure 8. In this example, there are just 4 non-empty intersections whereas our brute force algorithm would perform 16 iterations.

There are a couple of important observations to make in our example of Figure 8.

$$
\begin{array}{l}
FI_{F_1 \bigcap F_2} = \emptyset \\
\textbf{for } i_1 = 0, num_1 - 1 \\
\quad L_1 = (l_1 + i_1 \times str_1, r_1 + i_1 \times str_1) \\
\quad \textbf{for } i_2 = 0, num_2 - 1 \\
\quad\quad L_2 = (l_2 + i_2 \times str_2, r_2 + i_2 \times str_2) \\
\quad\quad FI_{F_1 \bigcap F_2} = FI_{F_1 \bigcap F_2} \bigcup LI_{L_1 \bigcap L_2}
\end{array}
$$

Figure 7: Brute Force FALLS Intersection Algorithm

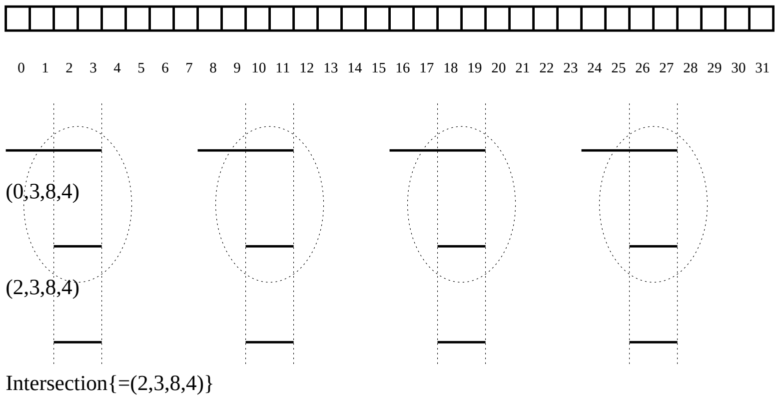

Figure 8: Example of FALLS Intersection

One of them is that the intersecting pairs of members of the two FALLS (circled in the figure) have the same relationship between them; i.e., their relative displacement is the same. This gives rise to the idea of periodicity in the relationships between members of the two FALLS. The length of the intersection period ($FP_{F_1 \bigcap F_2}$) for a given pair of FALLS $F_1 = (l_1, r_1, str_1, num_1)$ and $F_2 = (l_2, r_2, str_2, num_2)$ can be written down as:

$$FP_{F_1 \bigcap F_2} = lcm(str_1, str_2)$$

We also find it convenient to define a pair of quantities called m_1 and m_2 as follows:

$$m_1 = \frac{FP_{F_1 \bigcap F_2}}{str_1}$$
$$m_2 = \frac{FP_{F_1 \bigcap F_2}}{str_2}$$

Intuitively, these quantities represent the number of members of each FALLS occurring in a period. It can easily be verified that a pair of members from the two FALLS (i_1, i_2) will have the same relative displacement as the pair of members $(i_1 + m_1, i_2 + m_2)$. For the example of Figure 8, $FP_{F_1 \bigcap F_2} = 8$, $m_1 = 1$, and $m_2 = 1$.

Another observation to make in Figure 8 is that the intersection of the two FALLS in this case turns out to be a FALLS (as noted in the figure). These observations imply that we need only look at possible intersections between pairs of members of the two FALLS that occur within a period and extend any intersection that may result to all other periods (thus giving rise to a FALLS structure). This gives us a more efficient intersection computation algorithm shown in Figure 9. For the algorithm, (I_1, I_2) is the first pair of members of the two FALLS that intersect; all other terms have the same meaning as explained above. If we use this algorithm for the example of Figure 8, we see that $(I_1 = 0, I_2 = 0)$ is the first pair of

$$
\begin{array}{l}
FI_{F_1 \bigcap F_2} = \emptyset \\
\textbf{for } i_1 = I_1, I_1 + m_1 - 1 \\
\quad L_1 = (l_1 + i_1 \times str_1, r_1 + i_1 \times str_1) \\
\quad \textbf{for } i_2 = I_2, I_2 + m_2 - 1 \\
\quad\quad L_2 = (l_2 + i_2 \times str_2, r_2 + i_2 \times str_2) \\
\quad\quad \textbf{if } (LI_{L_1 \bigcap L_2} \neq \emptyset) \\
\quad\quad\quad l = LOW(LI_{L_1 \bigcap L_2}) \\
\quad\quad\quad r = HIGH(LI_{L_1 \bigcap L_2}) \\
\quad\quad\quad str = FP_{F_1 \bigcap F_2} \\
\quad\quad\quad num = \min(\left\lfloor \frac{num_1 - i_1 - 1}{m_1} \right\rfloor, \left\lfloor \frac{num_2 - i_2 - 1}{m_2} \right\rfloor) + 1 \\
\quad\quad\quad FI_{F_1 \bigcap F_2} = FI_{F_1 \bigcap F_2} \bigcup (l, r, str, num)
\end{array}
$$

Figure 9: FALLS Intersection Algorithm Based on Periods

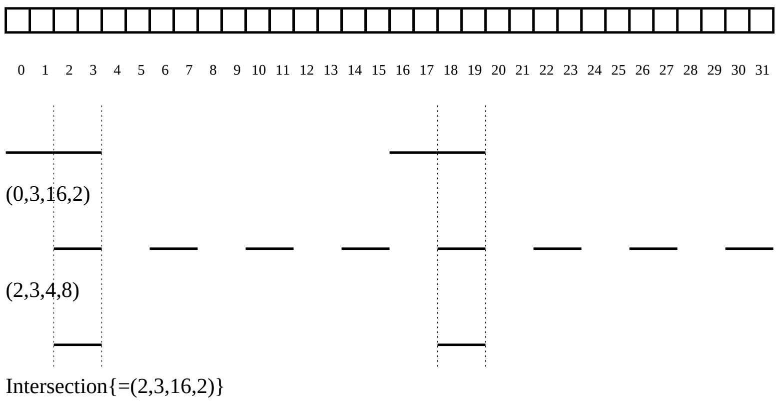

Figure 10: Another Example of FALLS Intersection

members of the two families that intersect. As seen earlier, $m_1 = m_2 = 1$ and $FP_{F_1 \bigcap F_2} = 8$. This means our algorithm will iterate just once with $L_1 = (0, 3)$ and $L_2 = (2, 3)$ which gives us a non-empty intersection $LI_{L_1 \bigcap L_2} = (2, 3)$ making $l = 2$ and $r = 3$. Using the value 8 for $FP_{F_1 \bigcap F_2}$ gives us $str = 8$. Finally, num can be calculated as 4, which gives us the intersection FALLS as $FI_{F_1 \bigcap F_2} = (2, 3, 8, 4)$.

In the example just considered, we had only one resultant FALLS. This may not be the case in general. The maximum number of FALLS that can be produced using this algorithm can be shown to be $m_1 + m_2$. However, this is not of much concern since m_1 and m_2 are very small in most situations.

Although the algorithm outlined above substantially cuts down on the number of iterations performed as compared to the brute force algorithm described first, it is still not very efficient. This can be seen by considering the example of Figure 10. For this example, $m_1 = 1$ and $m_2 = 4$, which means our algorithm iterates 4 times. However, we see that the intersection consists of just one FALLS, which means 3 of the iterations produce no FALLS and are thus wasted. This gives us the possibility of constructing another algorithm which still looks at pairs of members within a period, but, does not consider pairs that do not intersect. This pruning is done by considering the intersection of a pair of LS's $L_1 = (l_1, r_1)$ and $L_2 = (l_2, r_2)$. We see that this intersection is non-empty when $\max(l_1, l_2) \leq \min(r_1, r_2)$. This can happen under one of four conditions as listed below:

$$\begin{array}{lll} l_2 \leq l_1 & r_1 \leq r_2 & \\ l_2 \leq l_1 & l_1 \leq r_2 & r_2 \leq r_1 \\ l_1 \leq l_2 & r_2 \leq r_1 & \\ l_1 \leq l_2 & l_2 \leq r_1 & r_1 \leq r_2 \end{array}$$

Applying these conditions to determine when a pair of members (i_1, i_2) of two FALLS $F_1 = (l_1, r_1, str_1, num_1)$ and $F_2 = (l_2, r_2, str_2, num_2)$ will intersect gives us:

$$\begin{array}{lll} l_2 + i_2 \times str_2 \leq l_1 + i_1 \times str_1 & r_1 + i_1 \times str_1 \leq r_2 + i_2 \times str_2 & \\ l_2 + i_2 \times str_2 \leq l_1 + i_1 \times str_1 & l_1 + i_1 \times str_1 \leq r_2 + i_2 \times str_2 & r_2 + i_2 \times str_2 \leq r_1 \\ l_1 + i_1 \times str_1 \leq l_2 + i_2 \times str_2 & r_2 + i_2 \times str_2 \leq r_1 + i_1 \times str_1 & \\ l_1 + i_1 \times str_1 \leq l_2 + i_2 \times str_2 & l_2 + i_2 \times str_2 \leq r_1 + i_1 \times str_1 & r_1 + i_1 \times str_1 \leq r_2 + i_2 \times str_2 \end{array}$$

By performing a more detailed analysis, we can reduce these conditions to the following:

$$\begin{array}{l} i_2 \geq i_1 \times \frac{str_1}{str_2} + \frac{l_1 - r_2}{str_2} \\ i_2 \leq i_1 \times \frac{str_1}{str_2} + \frac{r_1 - l_2}{str_2} \end{array}$$

The equations above give us a method to determine which members of the two FALLS will actually intersect. It can be seen that given a member of the first FALLS we can use these conditions to determine the lower and upper bounds of members of the other FALLS (i_2s) that will intersect with the given member. We can now construct an efficient intersection algorithm as shown in Figure 11.

Note that we do not check for an empty $LI_{L_1 \bigcap L_2}$ because we are guaranteed it is non-empty by iterating over the loop bounds computed using the conditions listed above. For our example of Figure 10, we can see that our algorithm will iterate only once and produce the FALLS $FI_{F_1 \bigcap F_2} = (2, 3, 16, 2)$.

As seen before, some regular distributions may result in a processor having a set of FALLS representing its elements rather than just one FALLS. Intersection of a set of multiple FALLS

$FI_{F_1 \bigcap F_2} = \emptyset$
$I_1 = \max(0, \left\lceil \frac{l_2 - r_1}{str_1} \right\rceil)$
for $i_1 = I_1, \min(I_1 + m_1 - 1, num_1 - 1)$
$\quad L_1 = (l_1 + i_1 \times str_1, r_1 + i_1 \times str_1)$
$\quad$ **for** $i_2 = \max(0, \left\lceil \frac{i_1 \times str_1 + l_1 - r_2}{str_2} \right\rceil), \min(\left\lfloor \frac{i_1 \times str_1 + r_1 - l_2}{str_2} \right\rfloor, m_2 - 1, num_2 - 1)$
$\qquad L_2 = (l_2 + i_2 \times str_2, r_2 + i_2 \times str_2)$
$\qquad l = LOW(LI_{L_1 \bigcap L_2})$
$\qquad r = HIGH(LI_{L_1 \bigcap L_2})$
$\qquad str = FP_{F_1 \bigcap F_2}$
$\qquad num = \min(\frac{num_1 - i_1 - 1}{m_1}, \frac{num_2 - i_2 - 1}{m_2}) + 1$
$\qquad FI_{F_1 \bigcap F_2} = FI_{F_1 \bigcap F_2} \bigcup (l, r, str, num)$

Figure 11: Efficient FALLS Intersection Algorithm Based on Periods

with another set of multiple FALLS can be done by intersecting each possible pair of FALLS from the two sets (using the algorithm described above).

The conditions used for constructing the efficient FALLS intersection algorithm can also be used to construct a boolean function $B_{F_1 \bigcap F_2}$ defined as:

$$B_{F_1 \bigcap F_2} \begin{cases} TRUE & if\ FI_{F_1 \bigcap F_2} \neq \emptyset \\ FALSE & otherwise \end{cases}$$

In evaluating the function we use the parameters of the two FALLS. Intuitively, the function checks for the existence of at least one pair (i_1, i_2) satisfying the intersection conditions. Due to lack of space we are unable to present more details of this boolean function. As we will see later, it plays an important role in computing PITFALLS intersection.

2.3 Processor Index Tagged FAmiLy of Line Segments (PITFALLS)

Returning to the problem of redistribution, we can now see that a possible method could be to first construct a FALLS representation for each source processor based on the source data distribution and for each target processor based on the target data distribution. Next, we could iterate over all source-target pairs and determine the data to be sent between them using the FALLS intersection algorithm described above. However, this may not be very efficient in many cases since there may be many source-target processor pairs that do not

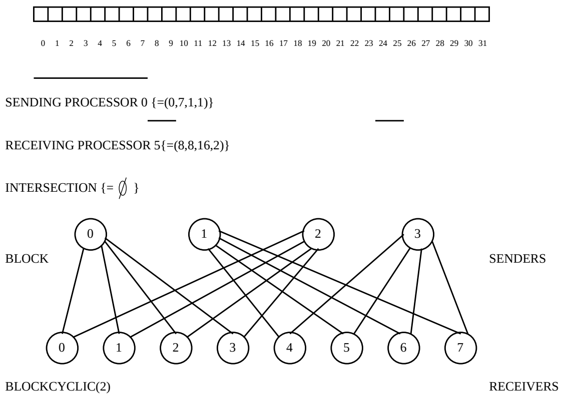

Figure 12: Example of Redistribution

communicate. Figure 12 shows a redistribution of a 32-element array from a $BLOCK$ distribution to a $BLOCKCYCLIC(2)$ distribution. Here we assume there are 4 sending processors and 8 receiving processors. If we consider the edges to represent communication, we can see that there will be no communication, for instance, between processor 0 on the sending end and processor 5 on the receiving end. In order to avoid this unnecessary iteration, we extend the FALLS representation to what is called the PITFALLS representation. A PITFALLS P is defined by a tuple $(l, r, str, num, disp, proc)$. We can see that we have two new parameters $disp$ and $proc$ as compared to the FALLS representation. Intuitively, a PITFALLS represents a set of equally spaced FALLS for a set of $proc$ processors with the spacing between the l's of successive processor FALLS being $disp$. Formally, the pth FALLS $(0 \leq p \leq proc - 1)$ of a PITFALLS $P = (l, r, str, num, disp, proc)$ (denoted by F^p and called the pth member of P) is given by:

$$F^p = (l + p \times disp, r + p \times disp, num, str)$$

The advantage of using PITFALLS is that we do not use a separate set of FALLS for each processor; instead, one set of PITFALLS is used for the entire set of processors across which an array is distributed. The PITFALLS representation is parameterized by the IDs of the processors. Thus, given an ID, we can determine the FALLS representation for the associated processor. Examples of PITFALLS for a few regular distributions of a 32 element array are shown in Figure 13. It can be shown easily that no more than three PITFALLS are needed to represent any regular distribution of an array.

As mentioned before, for redistribution, we are interested in being able to perform intersections on our representation. The advantage of the PITFALLS representation is that it not

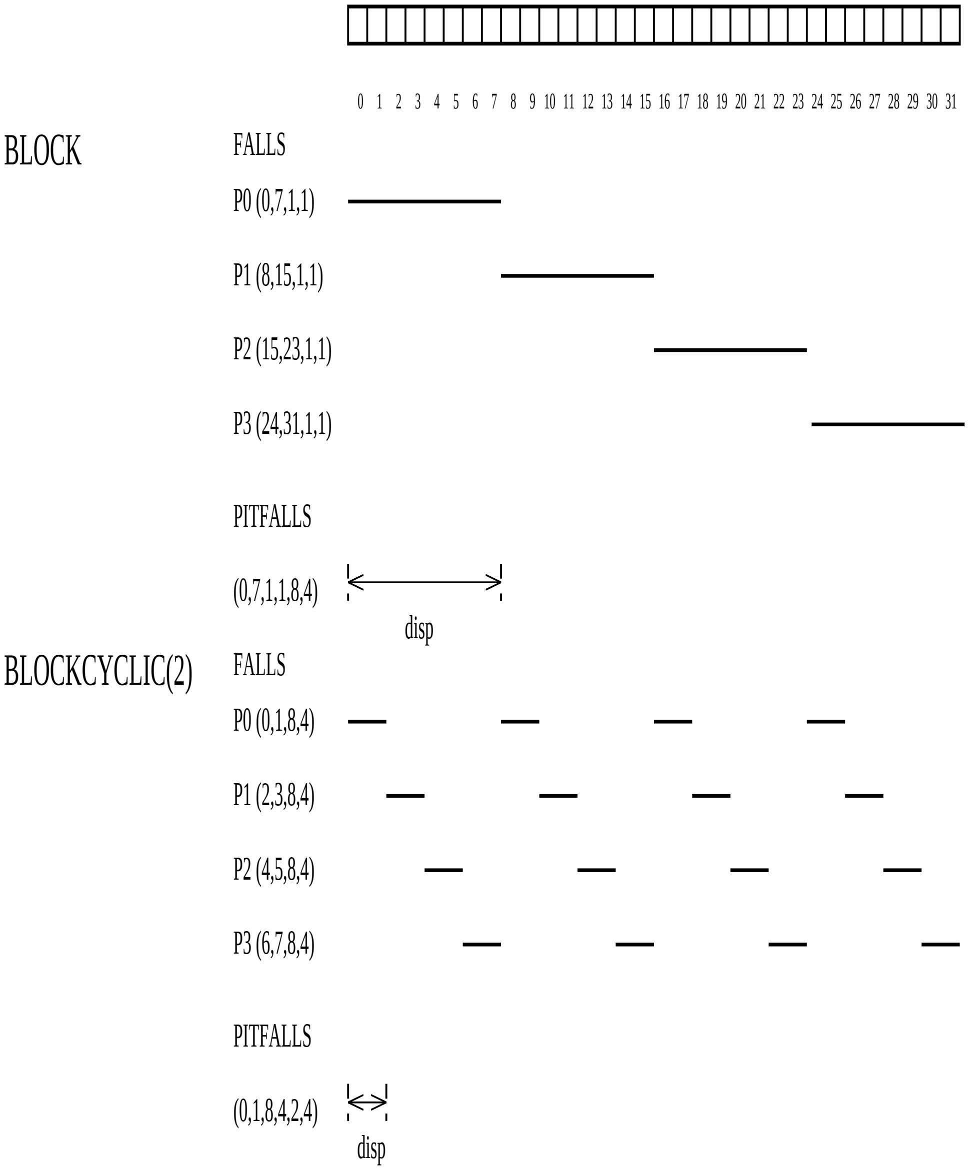

Figure 13: Examples of PITFALLS

for $p_1 = 0, proc_1 - 1$
  $F^{p_1} = (l_1 + p_1 \times disp_1, r_1 + p_1 \times disp_1, num_1, str_1)$
  for $p_2 = 0, proc_2 - 1$
    $F^{p_2} = (l_2 + p_2 \times disp_2, r_2 + p_2 \times disp_2, num_2, str_2)$
    if $B_{F^{p_1} \bigcap F^{p_2}} ==$ **true**
      compute $FI_{F^{p_1} \bigcap F^{p_2}}$

Figure 14: PITFALLS Intersection Algorithm

only helps us perform efficient intersection of data sets for a pair of processors, but also helps us determine which pair of processors will have a non-empty intersection. Consider a pair of PITFALLS $P_1 = (l_1, r_1, str_1, num_1, disp_1, proc_1)$ and $P_2 = (l_2, r_2, str_2, num_2, disp_2, proc_2)$. We can now write down the FALLS representation for a pair of members (p_1,p_2) from the two PITFALLS as:

$$F^{p_1} = (l_1 + p_1 \times disp_1, r_1 + p_1 \times disp_1, num_1, str_1)$$
$$F^{p_2} = (l_2 + p_2 \times disp_2, r_2 + p_2 \times disp_2, num_2, str_2)$$

We have previously defined a boolean function to determine whether a pair of FALLS will have a non-empty intersection. We can now use this function to determine whether the pair of FALLS (F^{p_1}, F^{p_2}) will intersect. Hence, we can decide whether the pair of processors (p_1, p_2) will need to communicate during redistribution. This is the basis for the PITFALLS intersection algorithm shown in Figure 14.

2.4 Multi-dimensional Array Redistribution

Until this point we have only considered a one-dimensional case for all our representations and algorithms. Extending these to the multi-dimensional case is trivial and can be done by simply looking at the representations for each dimension and performing intersections on them independently. An example of the two-dimensional case is provided in Figure 15. Here, we show the FALLS representation for processor 1 of a 4×2 processor grid for two given distributions. The first has the array distributed in a $(BLOCK, CYCLIC)$ manner; the second has it distributed in a $(CYCLIC, BLOCKCYCLIC(4))$ manner. We can see that the FALLS representation for each dimension is independent of the others. Our multi-dimensional FALLS intersection algorithm is shown in Figure 16.

After performing the dimension-by-dimension intersection, we can see that the set of

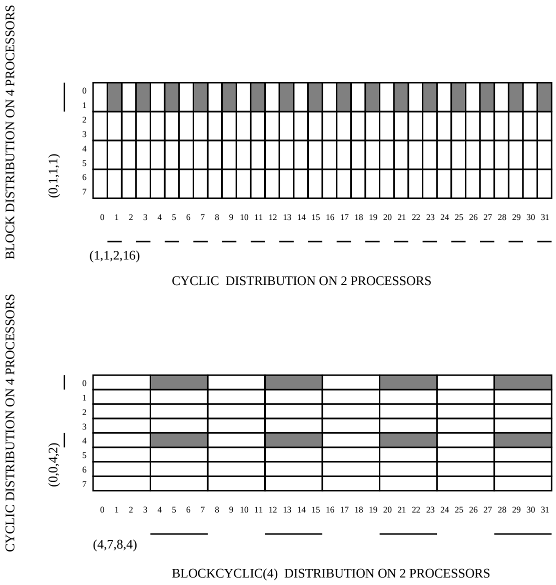

Figure 15: Examples of FALLS for Multidimensional Arrays

for $dim = 0, NumDims$

$\quad FI^{dim}_{F_1^{dim} F_2^{dim}} = \emptyset$

$\quad I_1^{dim} = \max(0, \left\lceil \frac{l_2^{dim} - r_1^{dim}}{str_1^{dim}} \right\rceil)$

$\quad$ **for** $i_1 = I_1^{dim}, \min(I_1^{dim} + m_1^{dim} - 1, num_1^{dim} - 1)$

$\quad\quad L_1 = (l_1^{dim} + i_1 \times str_1^{dim}, r_1^{dim} + i_1 \times str_1^{dim})$

$\quad\quad$ **for** $i_2 = \max(0, \left\lceil \frac{l_1^{dim} - r_2^{dim}}{str_2^{dim}} \right\rceil), \min(\left\lfloor \frac{r_1^{dim} - l_2^{dim}}{str_2^{dim}} \right\rfloor, m_2^{dim} - 1, num_2^{dim} - 1)$

$\quad\quad\quad L_2 = (l_2^{dim} + i_2 \times str_2^{dim}, r_2^{dim} + i_2 \times str_2^{dim})$

$\quad\quad\quad l = LOW(LI_{L_1 L_2})$

$\quad\quad\quad r = HIGH(LI_{L_1 L_2})$

$\quad\quad\quad str = FP^{dim}_{F_1^{dim} F_2^{dim}}$

$\quad\quad\quad num = \min(\frac{num_1^{dim} - i_1 - 1}{m_1^{dim}}, \frac{num_2^{dim} - i_2 - 1}{m_2^{dim}}) + 1$

$\quad\quad\quad FI_{F_1^{dim} F_2^{dim}} = FI_{F_1^{dim} F_2^{dim}} \bigcup (l, r, str, num)$

Figure 16: Multi-dimensional FALLS Intersection Algorithm

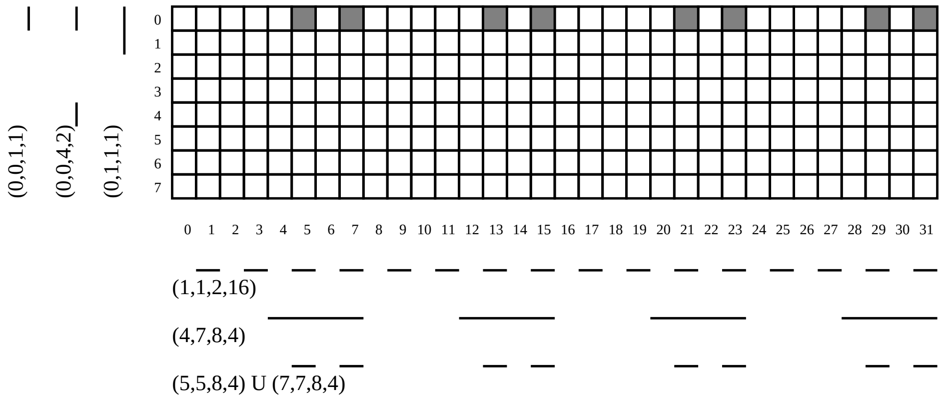

Figure 17: Example of Multidimensional FALLS Intersection

resulting FALLS represent the set of indices in each of the dimensions that need to be transferred. Thus, during the transfer we have to consider rectangular sections of elements by combining the sets of indices for all dimensions. This is made clear in the example of a two-dimensional FALLS intersection shown in Figure 17. The source and target distributions for this redistribution are assumed to be the ones shown in Figure 15. As we can see, the intersection along the rows indicates all elements of row 0 are to be sent from source processor 1 to target processor 1. The intersection along the columns indicates all elements in columns $5, 7, 13, 15, 21, 23, 29, 31$ are to be sent between these processors. Combining the two sets of indices gives us the set of elements $(0,5), (0,7), (0,13), (0,15), (0,21), (0,23), (0,29), (0,31)$. We indicate these via the shaded areas.

Our multi-dimensional algorithm scales linearly with the number of dimensions involved. This is a significant advantage over the methods of [25].

A similar approach as the one above is used for the PITFALLS intersection in multiple dimensions. We consider the PITFALLS representation for source and target processor sets in each dimension and perform the PITFALLS intersection for that dimension using our algorithm of Figure 14. Later, we combine the results of these intersections and obtain rectangular sections of data that need to be transferred.

2.5 Multi-array Redistribution

For multiple arrays being redistributed from one processor set to another processor set, we pack all the data to be transferred between a pair of processors for all the arrays into a single buffer before sending its contents. This way, we ensure that no more than one message is sent between processors even though they may communicate data for more than one array.

The advantage is that we have one long message instead of multiple short messages.

2.6 Summary

To summarize, in this section we developed the idea of a representation for regular distributions of a multi-dimensional array over a set of processors. We also provided an efficient algorithm based on this representation to compute the data transfer required for a redistribution. In the next section, we briefly discuss the implementation of our ideas in the context of the PARADIGM compiler and provide the results of a comparison of our technique with the runtime resolution method.

3 Implementation and Results

As mentioned earlier, the work proposed in this paper is aimed at supporting Multiple Program Multiple Data (MPMD) program generation in the PARADIGM compiler. Since the PARADIGM compiler is still under design and implementation, we tested our methods by automatically generating a set of functions for any given redistribution and executing these functions on the Intel PARAGON and CM-5. In order to generate these functions, we look at the source and target distributions of the array(s) being redistributed and generate the PITFALLS representation for both. Based on these representations and our PITFALLS intersection algorithm, we generate a pair of functions called GroupSend and GroupRecv to be executed by each of the source and target processors respectively. For the purposes of PITFALLS generation and intersection, we assume the processor subsystems are a contiguous block; i.e., if there are p processors in a subsystem, we number them 0 through $p-1$. We call these the virtual IDs for the processors and provide structures in the GroupSend and GroupRecv functions for any processor to determine its virtual ID based on its real ID. When an actual $SEND$ or $RECV$ is performed by any processor, it needs to remap virtual IDs to real IDs using the same structures. The structures basically specify the IDs of the processors involved in the source and target processor sets. The idea behind having such structures is very similar to the concepts of Groups, Contexts and Communicators described in the MPI standard [20]. Due to the unavailability of reliable implementations of MPI on the machines we test our methods on, we chose to use our own interface. However, we can easily modify our code to use MPI when reliable implementations become available.

We generated and timed our algorithm for a total of 27 redistributions using all possible combinations of:

D_s	D_t	Size	P_s	P_t	Naive(mS)	PITFALLS(mS)	Speedup
BC(3),B	C,BC(5)	128×128	4×4	3×5	23.91	10.98	2.18
			2×6	3×3	32.45	11.30	2.87
			3×5	4×3	26.28	10.75	2.45
		256×256	4×4	3×5	87.02	35.70	2.44
			2×6	3×3	125.72	44.64	2.81
			3×5	4×3	100.00	40.20	2.49
		512×512	4×4	3×5	344.16	130.92	2.63
			2×6	3×3	520.48	168.56	3.08
			3×5	4×3	396.89	149.38	2.66
BC(3),BC(7)	BC(5),C	128×128	5×2	4×3	36.94	16.56	2.23
			3×6	5×2	33.40	13.10	2.55
			4×5	3×3	34.27	15.86	2.16
		256×256	5×2	4×3	140.12	62.58	2.24
			3×6	5×2	125.23	44.87	2.79
			4×5	3×3	126.07	47.25	2.67
		512×512	5×2	4×3	573.66	231.96	2.47
			3×6	5×2	498.43	173.64	2.87
			4×5	3×3	501.74	174.73	2.87
B,A	A,B	128×128	8×1	1×16	29.62	9.67	3.06
			16×1	1×16	21.77	8.16	2.67
			10×1	1×18	26.51	9.08	2.92
		256×256	8×1	1×16	109.51	33.60	3.26
			16×1	1×16	75.84	22.46	3.38
			10×1	1×18	95.49	29.96	3.19
		512×512	8×1	1×16	441.90	129.96	3.40
			16×1	1×16	299.52	80.04	3.74
			10×1	1×18	372.34	115.43	3.23

Table 1: Results on the Thinking Machines CM-5

D_s	D_t	Size	P_s	P_t	Naive(mS)	PITFALLS(mS)	Speedup
BC(3),B	C,BC(5)	128×128	4×4	3×5	21.47	12.56	1.71
			2×6	3×3	26.30	10.43	2.52
			3×5	4×3	22.56	11.62	1.94
		256×256	4×4	3×5	66.42	24.84	2.67
			2×6	3×3	95.02	30.64	3.10
			3×5	4×3	75.01	27.80	2.70
		512×512	4×4	3×5	239.60	102.05	2.35
			2×6	3×3	364.69	132.93	2.74
			3×5	4×3	289.66	101.17	2.86
BC(3),BC(7)	BC(5),C	128×128	5×2	4×3	25.47	11.97	2.13
			3×6	5×2	26.23	12.90	2.03
			4×5	3×3	26.47	14.60	1.81
		256×256	5×2	4×3	91.58	33.32	2.75
			3×6	5×2	85.97	29.11	2.95
			4×5	3×3	87.23	32.64	2.67
		512×512	5×2	4×3	354.83	130.67	2.71
			3×6	5×2	317.44	109.84	2.89
			4×5	3×3	331.56	112.21	2.96
B,A	A,B	128×128	8×1	1×16	24.15	8.60	2.81
			16×1	1×16	20.42	9.40	2.17
			10×1	1×18	22.23	8.55	2.60
		256×256	8×1	1×16	88.36	21.05	4.20
			16×1	1×16	67.71	18.63	3.63
			10×1	1×18	74.38	19.28	3.86
		512×512	8×1	1×16	336.76	75.30	4.47
			16×1	1×16	232.57	60.47	3.85
			10×1	1×18	286.82	62.83	4.56

Table 2: Results on the Intel Paragon

1. 3 source-target distribution pairs:

   (a) $BC(3), B$ to $C, BC(5)$

   (b) $BC(3), BC(7)$ to $BC(5), C$

   (c) B, A to A, B

   where, $A = ALL$, $B = BLOCK$, $C = CYCLIC$ and $BC(x) = BLOCKCYCLIC(x)$.

2. 3 arrays sizes:

   (a) 128×128

   (b) 256×256

   (c) 512×512

3. 3 processor grids chosen independently for each distribution pair (shown along with the results in Tables 1 and 2).

The distributions and processor grids were chosen to show that our method can work well for all of the regular distributions on arbitrary processor sets.

To evaluate the effectiveness of our algorithm, we also implemented a runtime resolution algorithm (referred to in Tables as Naive) and timed it for the same set of redistributions (the details of such an algorithm were discussed in Section 1). The results of our study are tabulated in Tables 1 and 2. From these, we can make the following observations:

- Our algorithm performs better than the runtime resolution algorithm in all cases. In a couple of cases for the smallest array, the performance improvement is not great; this is attributed to the fact that elements being transferred between processors in these cases are very scattered and not in clustered sections making addressing them very expensive.

- The performance improvement becomes more appreciable as the array size increases. This means it is vital to use an efficient technique like ours for large array redistributions.

- It was of interest to compare the per element cost for the two methods as a function of array size for a particular redistribution. For this purpose, we selected the redistribution in which our method performs closest to the runtime resolution method for the

System	Array Size	Naive (μS)	PITFALLS (μS)
Intel Paragon	128×128	1.61	1.45
	256×256	1.33	0.76
	512×512	1.26	0.60
Thinking Machines CM-5	128×128	2.03	1.77
	256×256	1.91	1.04
	512×512	1.90	0.78

Table 3: Comparison of Per Element Costs

smallest array. The values computed are tabulated in Table 3. From this table we can see that the per element costs drop very rapidly for our method as compared to the runtime resolution method. This indicates that the overhead factor of our method is very small for large arrays.

- The improvement seems to be independent of the underlying machine. Both machines seem to show the same order of improvement.

4 Conclusions and Future Work

In this paper we have described a technique for carrying out array redistribution in an efficient manner. Our technique relies on a simple yet effective representation (PITFALLS) for regular distribution of arrays. This representation makes the communication analysis required for redistribution very simple and efficient. The results we provide show that our method is much superior to naive runtime resolution type of methods. The factor of improvement achieved is higher for large array sizes, making it critical to use an efficient technique like ours for redistribution.

We are currently exploring the possibility of using the PITFALLS representation for general communication analysis in the PARADIGM compiler. We would also like to consider redistribution of sections of an array and not the entire array. Such redistributions are needed sometimes at procedure boundaries if the procedure called modifies only a section of the input array. We are also going to undertake a more thorough analysis of the overheads of our method and look into the possibility of reducing these overheads further.

References

[1] M. Gupta, *Automatic Data Partitioning on Distributed Memory Multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[2] M. Gupta and P. Banerjee, "A Methodology for High-level Synthesis of Communication on Multicomputers," in *Proceedings of the International Conference on Supercomputing*, 1992.

[3] M. Gupta and P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," *IEEE Transactions on Parallel and Distributed Computing*, pp. 179–193, March 1992.

[4] M. Gupta and P. Banerjee, "Compile-time Estimation of Communication Costs on Multicomputers," in *International Parallel Processing Symposium*, 1992.

[5] E. Su, D. Palermo, and P. Banerjee, "Automating Parallelization of Regular Computations for Distributed Memory Machines in the PARADIGM compiler," in *The Proceedings of the International Conference on Parallel Processing*, 1993.

[6] E. Su, D. Palermo, and P. Banerjee, "Processor Tagged Descriptors: A Data Structure for Compiling for Distributed Memory Multicomputers," in *to appear in the Proceedings of the Parallel Architectures and Compiler Technology Conference*, 1994.

[7] D. Palermo, E. Su, J. Chandy, and P. Banerjee, "Communication Optimizations for Distributed Memory Multicomputers used in the PARADIGM Compiler," in *to appear in the Proceedings of the International Conference on Parallel Processing*, 1994.

[8] S. Ramaswamy and P. Banerjee, "Processor Allocation and Scheduling of Macro Dataflow Graphs on Distributed Memory Multicomputers by the PARADIGM Compiler," in *Proceedings of the International Conference on Parallel Processing*, 1993.

[9] S. Ramaswamy and P. Banerjee, "A Convex Programming Approach for Exploiting Data and Functional Parallelism," in *to appear in the Proceedings of the International Conference on Parallel Processing*, 1994.

[10] A. Lain and P. Banerjee, "Techniques to Overlap Computation and Communication in Irregular Iterative Applications," in *to appear in the Proceedings of the International Conference on Supercomputing*, 1994.

[11] J. M. Anderson and M. S. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines," in *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, 1993.

[12] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim, "An Overview of a Compiler for Scalable Parallel Machines," in *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, 1993.

[13] U. Kremer, "Automatic Data Layout for Distributed-Memory Machines," Tech. Rep. CRPC-TR93299-S, Rice University, 1993.

[14] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiling FORTRAN D for MIMD Distributed Memory Machines," *Communications of the ACM*, pp. 66–80, Aug. 1992.

[15] High Performance FORTRAN Forum, *High Performance FORTRAN Language Specification*, 1993. Version 1.

[16] I. T. Foster and K. M. Chandy, "FORTRAN M: A Language for Modular Parallel Programming," tech. rep., Argonne National Laboratory/California Institute of Technology, 1992.

[17] A. L. Cheung and A. P. Reeves, "Function-Parallel Computation in a Data Parallel Environment," in *Proceedings of the International Conference on Parallel Processing*, 1993.

[18] J. Subhlok, J. M. Stichnoth, D. R. O'Halloran, and T. Gross, "Exploiting Task and Data Parallelism on a Multicomputer," in *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[19] J. Subhlok, D. O'Halloran, T. Gross, P. A. Dinda, and J. Webb, "Communication and Memory Requirements as the Basis for Mapping Task and Data Parallel Programs," Tech. Rep. CMU-CS-94-106, Carnegie Mellon University, 1994.

[20] Message-Passing Interface Forum, *Document for a Standard Message-Passing Interface*, 1993. Version 1.0.

[21] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Evaluation of Compiler Optimizations for FORTRAN D on MIMD Distributed-Memory Machines," in *Proceedings of the International Conference on Supercomputing*, 1992.

[22] C.-W. Tseng, *An Optimizing FORTRAN D Compiler for Distributed Memory Machines.* PhD thesis, Rice University, 1993.

[23] S. P. Amarasinghe and M. S. Lam, "Communication Optimization and Code Generation for Distributed Memory Machines," in *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, 1993.

[24] G. Agarwal, A. Sussman, and J. Saltz, "Compiler and Runtime Support for Structured and Block Structured Applications," in *Proceedings of Supercomputing*, 1993.

[25] R. Thakur, A. Choudhary, and G. Fox, "Redistribution of Arrays in HPF," in *to appear in the Proceedings of the Scalable High Performance Computing Conference*, 1994.