Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers

Shankar Ramaswamy and Prithviraj Banerjee

Center for Reliable and High Performance Computing
Coordinated Science Laboratory
University of Illinois
1308 West Main St.
Urbana, IL 61801

shankar,banerjee@crhc.uiuc.edu

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. A significant contribution of this work is the ability to handle arbitrary source and target processor sets while performing redistribution; another is the ability to handle arbitrary dimensionality for the array being redistributed in a scalable manner. The results presented show low overheads for our redistribution algorithm as compared to naive runtime methods.

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 recent HPF effort [1] aims at easing the programming process through the use of data distribution directives. Such directives can help compilers in guiding the partitioning of computation and thus the generation of calls to appropriate communication primitives. The HPF standard also provides directives for data redistribution dynamically during program execution. This has been found to provide increased speedups for some programs like the 2D Fast Fourier Transform (FFT) and Alternating Direction Integration (ADI).

Redistribution of data is critical for programs exploiting data and functional parallelism together. In such programs, a given processor system is partitioned between a set of data parallel tasks to provide increased speedups [2, 3, 4, 5]. Frequently, data dependence constraints require arrays be redistributed from one subsystem to another. Figure 1 illustrates this clearly; array A is being written into by the data parallel task executing on the first set of processors and being read by the data parallel task on the second set of processors. Since the tasks themselves may require different distributions for A, redistribution involves not only changing the distribution but also changing the processor sets. The PARADIGM compiler effort [6] at the University of Illinois considers the problem of simultaneous exploitation of functional and data parallelism. This has been a primary motivation for the work presented in this paper.

In our work we will consider only "regular" distri-

This research was supported in part by an IBM Graduate Fellowship and in part by the National Aeronautics and Space Administration under Contract NASA NAG 1-613. Authors are also grateful to the National Center for Supercomputing Applications, the San Diego Supercomputing Center, and the Argonne National Lab for providing access to their machines.

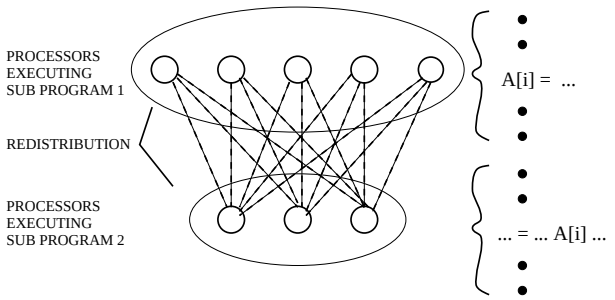

Figure 1: Need for Redistribution in MPMD Programs

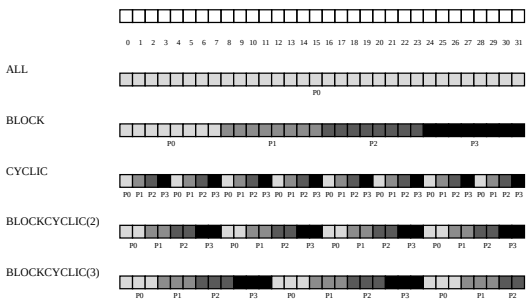

Figure 2: Examples of Regular Distributions

butions 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 2. The consideration of only such distributions is reasonable because a large number of scientific programs have been found to use them.

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 multi-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 regular data distributions along each dimension of the array. Therefore, the redistribution routines need 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; then, it 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; finally, 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 work are: (1) Redistribution routines are to be automatically generated at compile time; (2) The source and target processor sets can be any arbitrary subset of the given processor system; (3) Arrays being redistributed can have any of the possible regular distributions on the source and target processor sets. We handle all the types of distributions in an uniform manner; (4) Arrays being redistributed can have an arbitrary number of dimensions; (5) Multiple arrays can be redistributed simultaneously using message aggregation [7, 8]. 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). We felt the need for a new representation for our work because none of the current representations [8, 7] satisfied our requirements. We needed the ability to mathematically represent a regular distribution on any given subset of processors of the given system and to perform redistribution communication analysis efficiently.

The PITFALLS representation along with the redistribution algorithm based on it is discussed in the next section. Section 3 discusses our implementation and provides a feel for the efficacy of our method. Finally, in the last section, 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 compiler techniques such as those outlined in [7, 9, 8, 10]. 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, most current compiler techniques are not able to efficiently handle all possible combinations of regular distributions of A and B. In addition, they cannot handle different processor sets for A and B.

The work by Agarwal et. al. [11] 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. [12] considers routines for redistributions of regular arrays in the form of a library for a HPF compiler. The methods proposed treat possible source-target distributions in a pairwise manner. For specific cases of source-target data distributions, they use efficient methods; however, for more general cases, they use a runtime resolution approach such as the one described before. For multi-dimensional arrays, [12] proposes a solution which is very expensive for certain cases; in such cases, multi-dimensional redistributions are considered to be composed of a series of one-dimensional redistributions.

2 The PITFALLS Representation and Redistribution

Redistributing an array between a source-target pair of processor sets involves computing the elements to be sent/received between each source-target pair of processors. The PITFALLS representation is particularly useful in this context as it can be used to easily determine which pairs of source-target processors need to communicate and the intersection of the data sets of such a pair. We first develop the PITFALLS representation and redistribution algorithm for a one-dimensional array in a step by step manner. Later, we extend our ideas for multi-dimensional arrays. We do not consider the effects of alignments in our discussion here, taking them into account is easy since the PITFALLS representation is powerful enough to capture alignment information.

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 interest is in being able to find intersections for sets of elements, we note that the intersection

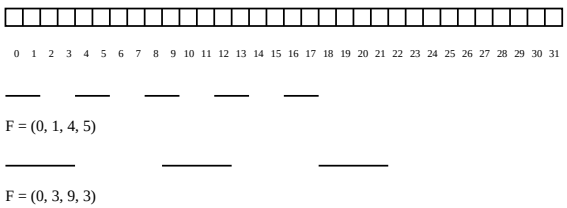

Figure 3: Examples of FALLS

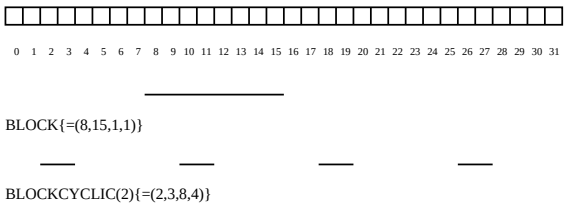

Figure 4: Examples of FALLS Describing Regular Distributions

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:

$$LI_{L_1 \bigcap L_2} = (\max(l_1, l_2), \min(r_1, r_2))$$

where $\max(l_1, l_2) \leq \min(r_1, r_2)$. If this is not true, the intersection is $\emptyset$.

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, s, n). Intuitively, F represents a set of n equally spaced, equally sized blocks of elements; the first block starts at l and ends at r; the stride between successive l's is s. Note that these are non-overlapping blocks. The ith ($0 \leq i \leq n - 1$) LS of F (denoted by L^i and called the ith member) is given by:

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

Figure 3 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 4 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 the regular distribution of any dimension of an array [13].

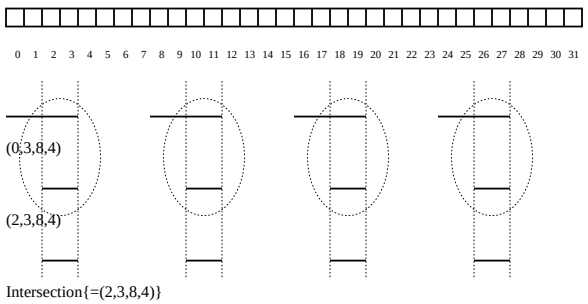

Figure 5: Example of FALLS Intersection

$FI_{F_1 \bigcap F_2} = \emptyset$
for $i_1 = I_1, I_1 + m_1 - 1$
  $L_1 = (l_1 + i_1 \times s_1, r_1 + i_1 \times s_1)$
  for $i_2 = I_2, I_2 + m_2 - 1$
    $L_2 = (l_2 + i_2 \times s_2, r_2 + i_2 \times s_2)$
    if $(LI_{L_1 \bigcap L_2} \neq \emptyset)$
      $l = LOW(LI_{L_1 \bigcap L_2})$
      $r = HIGH(LI_{L_1 \bigcap L_2})$
      $s = FP_{F_1 \bigcap F_2}$
      $n = \min(\lfloor \frac{n_1 - i_1 - 1}{m_1} \rfloor, \lfloor \frac{n_2 - i_2 - 1}{m_2} \rfloor) + 1$
      $FI_{F_1 \bigcap F_2} = FI_{F_1 \bigcap F_2} \bigcup (l, r, s, n)$

Figure 6: FALLS Intersection Algorithm Based on Periods

Given two FALLS $F_1 = (l_1, r_1, s_1, n_1)$ and $F_2 = (l_2, r_2, s_2, n_2)$, a simple brute force algorithm to compute the intersection ($FI_{F_1 \bigcap F_2}$) could simply iterate over every pair of members from the two FALLS and compute their intersection using the LS intersection algorithm. We can see that this technique can be very inefficient by considering the example of FALLS intersection shown in Figure 5. 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 5. 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 $F_1 = (l_1, r_1, s_1, n_1)$ and $F_2 = (l_2, r_2, s_2, n_2)$ can be written down as:

$$FP_{F_1 \bigcap F_2} = lcm(s_1, s_2)$$

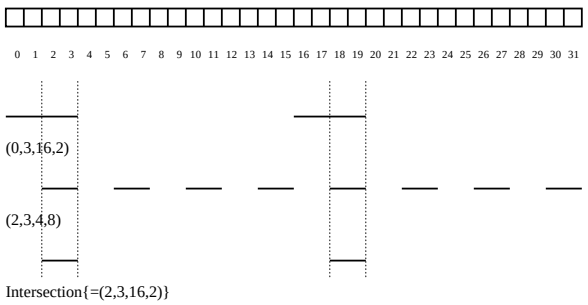

Figure 7: Another Example of FALLS Intersection

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

$$m_1 = \frac{FP_{F_1 \bigcap F_2}}{s_1}$$
$$m_2 = \frac{FP_{F_1 \bigcap F_2}}{s_2}$$

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 5, $FP_{F_1 \bigcap F_2} = 8$, $m_1 = 1$, and $m_2 = 1$.

Another observation to make in Figure 5 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 each such intersection to all other periods (thus giving rise to a FALLS structure). This gives us a more efficient intersection computation algorithm shown in Figure 6. 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.

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 7. 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

$FI_{F_1 \bigcap F_2} = \emptyset$
$I_1 = \max(0, \lceil \frac{l_2 - r_1}{s_1} \rceil)$
for $i_1 = I_1, \min(I_1 + m_1 - 1, n_1 - 1)$
  $L_1 = (l_1 + i_1 \times s_1, r_1 + i_1 \times s_1)$
  $LB_2 = \max(0, \lceil \frac{i_1 \times s_1 + l_1 - r_2}{s_2} \rceil)$
  $UB_2 = \min(\lfloor \frac{i_1 \times s_1 + r_1 - l_2}{s_2} \rfloor, m_2 - 1, n_2 - 1)$
  for $i_2 = LB_2, UB_2$
    $L_2 = (l_2 + i_2 \times s_2, r_2 + i_2 \times s_2)$
    $l = LOW(LI_{L_1 \bigcap L_2})$
    $r = HIGH(LI_{L_1 \bigcap L_2})$
    $s = FP_{F_1 \bigcap F_2}$
    $n = \min(\frac{n_1 - i_1 - 1}{m_1}, \frac{n_2 - i_2 - 1}{m_2}) + 1$
    $FI_{F_1 \bigcap F_2} = FI_{F_1 \bigcap F_2} \bigcup (l, r, s, n)$

Figure 8: Efficient FALLS Intersection Algorithm Based on Periods

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:

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

Applying these conditions to determine when a pair of members (i_1, i_2) of two FALLS $F_1 = (l_1, r_1, s_1, n_1)$ and $F_2 = (l_2, r_2, s_2, n_2)$ will intersect gives us the following conditions (both need to be satisfied):

$$\begin{array}{l} i_2 \geq i_1 \times \frac{s_1}{s_2} + \frac{l_1 - r_2}{s_2} \\ i_2 \leq i_1 \times \frac{s_1}{s_2} + \frac{r_1 - l_2}{s_2} \end{array}$$

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 8. 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 7, 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

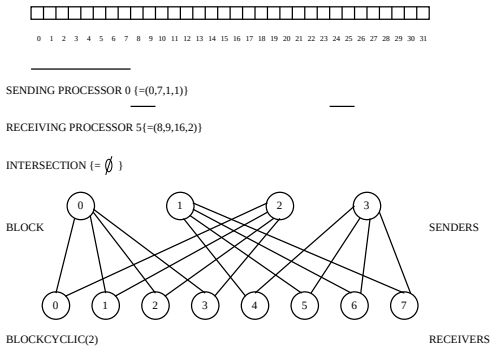

Figure 9: Example of Redistribution

of a set of multiple FALLS 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 constant time 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 communicate. Figure 9 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

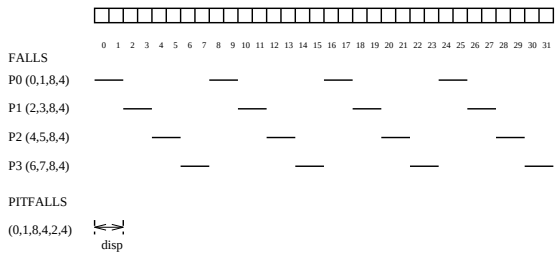

Figure 10: Example of PITFALLS

for $p_1 = 0, p_1 - 1$
  $F^{p_1} = (l_1 + p_1 \times d_1, r_1 + p_1 \times d_1, n_1, s_1)$
  for $p_2 = 0, p_2 - 1$
    $F^{p_2} = (l_2 + p_2 \times d_2, r_2 + p_2 \times d_2, n_2, s_2)$
    if $B_{F^{p_1} \bigcap F^{p_2}} ==$ **true**
      compute $FI_{F^{p_1} \bigcap F^{p_2}}$

Figure 11: PITFALLS Intersection Algorithm

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, s, n, d, p). We can see that we have two new parameters d and p as compared to the FALLS representation. Intuitively, a PITFALLS represents a set of equally spaced FALLS for a set of p processors with the spacing between the l's of successive processor FALLS being d. Formally, the pth FALLS $(0 \leq p \leq p-1)$ of a PITFALLS $P = (l, r, s, n, d, p)$ (denoted by F^p and called the pth member of P) is given by:

$$F^p = (l + p \times d, r + p \times d, n, s)$$

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. An example of PITFALLS for a $BLOCKCYCLIC(2)$ distribution of a 32 element array is shown in Figure 10. It can be shown easily that no more than three PITFALLS are needed to represent a regular distribution of any dimension of an array [13].

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 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 $P_1 = (l_1, r_1, s_1, n_1, d_1, p_1)$ and $P_2 = (l_2, r_2, s_2, n_2, d_2, p_2)$. We can now write down the FALLS representation for a pair of members (p_1, p_2) from the two PITFALLS as:

$$F^{p_1} = (l_1 + p_1 \times d_1, r_1 + p_1 \times d_1, n_1, s_1)$$
$$F^{p_2} = (l_2 + p_2 \times d_2, r_2 + p_2 \times d_2, n_2, s_2)$$

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 11.

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 simple and can be done by considering the FALLS representations for each dimension and performing intersections on them independently. The resulting FALLS in each dimension represents the set of indices that need to be transferred along each dimension. During the actual packing and transfer, we combine the indices along all dimensions. Since the multi-dimensional algorithm involves the application of the one-dimensional algorithm for each dimension independently, it scales linearly with the number of dimensions involved. This is a significant advantage over the methods of [12].

A similar approach 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 11. Later, we combine the results of these intersections and obtain rectangular sections of data that need to be transferred. More details on the multi-dimensional redistribution can be found in [13].

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 a pair of processors even if the communication involves data for more than one array. This idea is similar to the concept of message aggregation [8, 7]

3 Implementation and Results

As mentioned earlier, the work proposed in this paper is aimed at supporting exploitation of functional and data parallelism in the PARADIGM compiler. Since the PARADIGM compiler is still under design and implementation, we tested our methods by independently generating and timing redistribution routines using our algorithms for a few different redistributions on the Intel PARAGON and CM-5. For the purposes of communication analysis using PITFALLS, we assumed virtual IDs for individual processors in the source and target processor sets. We then used structures to map between virtual and real IDs during actual communication. The idea behind having such structures is very similar to the concepts of Groups, Contexts and Communicators described in the MPI standard [14]. Due to the unavailability of reliable implementations of MPI on the PARAGON and CM-5 at the time of submission, we chose to use our own interface.

We generated and timed our algorithm and the runtime resolution algorithm (described in Section 1) for a total of 18 redistributions for array sizes of 128×128 and 512×512. The results are shown in Tables 1 and 2. From these, we can see that our redistribution algorithm performs much better than the runtime resolution algorithm. The performance improvement becomes more appreciable for larger array size; making it vital to use an efficient technique such as ours for redistributions of large arrays. The improvement obtained seems to be independent of the underlying machine; the order is similar for both target machines.

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 mathematical representation (PITFALLS) for regular distribution of arrays. The 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.

Acknowledgements

The idea behind the constant time boolean test for deciding whether a pair of processors would actually communicate is due to Barbara Simons at IBM Santa Teresa Labs. We would like to thank her for permitting us to use it.

References

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

[2] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A Convex Programming Approach for Exploiting Data and Functional Parallelism," in *the Proc. of the Intl. Conf. on Parallel Processing*, (St. Charles, IL), August 1994.

[3] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A Framework for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers," tech. rep., University of Illinois, 1994.

[4] I. Foster, M. Xu, B. Avalani, and A. Choudhary, "A Compilation System That Integrates High Performance FORTRAN and FORTRAN-M," in *the Proc. of the Scalable High Performance Computing Conf.*, (Knoxville, TN), May 1994.

[5] J. Subhlok, J. M. Stichnoth, D. R. O'Halloran, and T. Gross, "Exploiting Task and Data Parallelism on a Multicomputer," in *Proc. of the Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, (San Diego, CA), May 1993.

[6] Prithviraj Banerjee et. al., "The PARADIGM Compiler for Distributed Memory Machines," in *to appear in the Proc. of the First Intl. Workshop on Parallel Processing*, (Bangalore, India), December 1994.

D_s	D_t	Size	P_s	P_t	Runtime(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
		512×512	4×4	3×5	344.16	130.92	2.63
			2×6	3×3	520.48	168.56	3.08
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
		512×512	5×2	4×3	573.66	231.96	2.47
			3×6	5×2	498.43	173.64	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
		512×512	8×1	1×16	441.90	129.96	3.40
			16×1	1×16	299.52	80.04	3.74

Table 1: Results on the Thinking Machines CM-5

D_s	D_t	Size	P_s	P_t	Runtime(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
		512×512	4×4	3×5	239.60	102.05	2.35
			2×6	3×3	364.69	132.93	2.74
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
		512×512	5×2	4×3	354.83	130.67	2.71
			3×6	5×2	317.44	109.84	2.89
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
		512×512	8×1	1×16	336.76	75.30	4.47
			16×1	1×16	232.57	60.47	3.85

Table 2: Results on the Intel Paragon

[7] E. Su, D. Palermo, and P. Banerjee, "Processor Tagged Descriptors: A Data Structure for Compiling for Distributed Memory Multicomputers," in *the Proc. of the Parallel Architectures and Compiler Technology Conf.*, (Montreal, Canada), August 1994.

[8] S. Hiranandani, K. Kennedy, and C. wen Tseng, "Compiling FORTRAN D for MIMD Distributed Memory Machines," *CACM*, August 1992.

[9] S. P. Amarasinghe and M. S. Lam, "Communication Optimization and Code Generation for Distributed Memory Machines," in *the Proc. of the SIGPLAN Conf. on Programming Language Design and Implementation*, (Albuquerque, NM), June 1993.

[10] Siddhartha Chaterjee et. al., "Generating Local Addresses and Communication Sets for Data Parallel Programs," in *the Proc. of the Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, (San Diego, CA), May 1993.

[11] G. Agarwal, A. Sussman, and J. Saltz, "Compiler and Runtime Support for Structured and Block Structured Applications," in *the Proc. of Supercomputing*, (Portland, OR), November 1993.

[12] R. Thakur, A. Choudhary, and G. Fox, "Redistribution of Arrays in HPF," in *the Proc. of the Scalable High Performance Computing Conf.*, (Knoxville, TN), May 1994.

[13] S. Ramaswamy and P. Banerjee, "Automatic Generation of Efficient Array Redistribution Routines for Distributed Memory Multicomputers," tech. rep., University of Illinois, 1994.

[14] Message-Passing Interface Forum, *Document for a Standard Message-Passing Interface*, 1993.