to appear in the Journal of Programming Languages, 1994

Compile-Time Estimation of Communication Costs of Programs *

Manish Gupta† Prith Banerjee‡

Abstract

One of the most challenging problems in compiling for distributed memory machines is to determine how data for a program should be distributed across processors. Any compiler that makes data partitioning decisions needs a mechanism for estimating communication and computational costs of programs to compare different alternatives. This paper presents a methodology for estimating communication costs of programs written in global address space. In this approach, the compiler analyzes programs *before* generating communication, and yet takes into account important communication optimizations that will be performed. We introduce the notion of traversal properties of array references in loops, that help identify the nature and extent of data movement in terms of high-level communication primitives. This enables the compiler to obtain more precise information about the global state of communication, and in a largely machine-independent manner. The methodology described in this paper has been implemented in a compiler, PARADIGM, that automatically determines data partitioning for Fortran programs. The results obtained with PARADIGM confirm the importance of this analysis for making good data partitioning decisions. The techniques developed in this work for recognizing communication primitives that best characterize the data movement are quite general, and also form the basis of generation of communication in PTRAN-II, a prototype compiler for High Performance Fortran.

1 Introduction

Distributed-memory architectures have become extremely popular as a cost-effective method of building massively parallel computers. However, these machines are not very easy to program. The programmer has to perform low-level tasks like distributing data across processors, and managing communication among those processors. Languages like High Performance Fortran (HPF) [8] free the programmer from the burden of explicit message-passing by allowing him/her to write sequential or shared-memory parallel programs, annotated with directives specifying data decomposition. The compilers for these languages are responsible for partitioning the computation, and generating the communication necessary to fetch values of non-local data referenced by a processor. A number of such prototype compilers have been developed [13, 31, 17, 23, 16, 21, 12, 26].

The performance of these programs depends greatly on the data partitioning scheme chosen by the

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

†IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598. Email: mgupta@watson.ibm.com. This work was done while the author was at the University of Illinois.

‡Center for Reliable and High Performance Computing, University of Illinois, Urbana, IL 61801. Email: banerjee@crhc.uiuc.edu

$$
\begin{array}{ll}
 & \texttt{do } k = 1, T \\
 & \quad \ldots \\
 & \quad\quad \texttt{do } j = 1, n \\
 & \quad\quad\quad \texttt{do } i = 1, n \\
S_1: & \quad\quad\quad\quad A(i,j) = A(i,j) + s * D(i) \\
 & \quad\quad\quad\quad \ldots
\end{array}
$$

Figure 1: Effect of dependence on communication cost

programmer. In general, the best partitioning scheme depends not only on program characteristics, but also on numerous machine-specific parameters, and on the kind of optimizations performed by the compiler. While early distributed memory compilers provided no assistance to the programmer in this crucial task, a number of researchers have recently addressed the problem (or parts of the problem) of automatically determining the distribution of data [2, 4, 7, 9, 11, 14, 15, 17, 20, 22, 28].

Any strategy for automatic data partitioning needs a mechanism for cost estimation to compare different alternatives regarding data partitioning, and to evaluate the relative benefits and overheads of repartitioning data at various points. Most of the proposed approaches to automatic data distribution use very simplified metrics for communication costs. The resultant inaccuracies in the estimated cost can often prevent the compiler from reaching good data partitioning decisions, even if the underlying algorithms for the selection process are good. Based on our experience with Paradigm [11, 9], a compiler that automatically determines data distribution for Fortran programs, we have found the following features to be extremely important for any performance estimator guiding data partitioning decisions:

- *Incorporating data dependence information*: The *flow dependences* [30] to an array reference being read in a statement inside loops restrict the extent to which messages for that reference can be combined [31, 13]. For example, in Figure 1, the flow dependences involving the reference to $D(i)$ will govern the placement of its communication. If there is a dependence *carried* by the i-loop, there is a separate message needed to send the value of $D(i)$ to the processor executing statement S_1 (if it is different from the processor owning $D(i)$), during each iteration of the loops surrounding S_1. At the other extreme, if there is no flow dependence to that reference, all communication involving the array D can be done once before entering the k-loop. Thus, communication would be aggregated with respect to the two inner loops, and its redundant repetitions in the k-loop would be eliminated. Given the high cost of communication and the significant difference between startup costs and per-element costs of sending messages, it is clear that the incurred costs will differ greatly in these cases. Thus, the estimated cost should take into account dependence information, otherwise it can be considerably inaccurate.
- *Providing a high-level view of communication*: Even if the compiler generates communication using only `send` and `receive` primitives, it is useful to recognize the overall data movement taking place

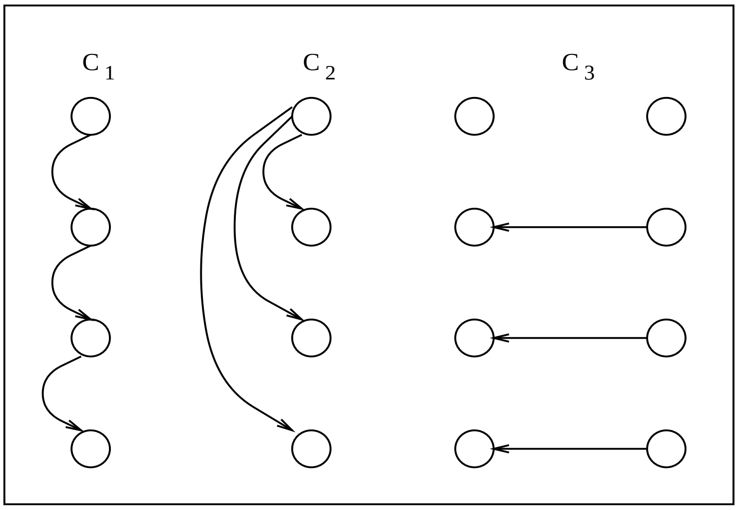

Figure 2: Need for high-level view of communication

across processors in terms of higher-level primitives. For instance, consider the different communication patterns shown in Figure 2, each consisting of three `send-receives` of identical message sizes, and with no constraints on their ordering. Even though the number of `send-receives` is exactly the same in these cases, the overall communication cost for these patterns will be different. In this example, we expect $\text{cost}(C_3) < \text{cost}(C_1) < \text{cost}(C_2)$. Characterization of data movement in terms of collective communication primitives leads to a concise and machine-independent representation that is suitable for data-parallel programs. This representation also provides richer information about the global state of communication (for instance, whether it is a one-to-many, many-to-one, or a many-to-many communication pattern), that can be useful for modeling effects like contention in the network.

In this paper, we present a methodology for estimating communication costs, that analyzes the program before communication is generated by the compiler. Our approach for estimating computational costs is described elsewhere [9]. The estimates of communication costs are expressed in terms of times to carry out certain high-level communication primitives on the target machine. Given information regarding the performance characteristics of those primitives on a multicomputer, the actual times spent on communication can then be estimated for that specific machine. This methodology has been implemented in the PARADIGM compiler [9]. The analysis we have developed introduces new techniques for determining how data movement needed in a program can be implemented efficiently using collective communication primitives, and forms the basis of generation of communication by the PTRAN II compiler [12].

Related Work Many researchers have developed tools [3, 5, 6, 27] that estimate the performance of a parallel program with explicit communication at compile time. The tool developed by Balasundaram et al. [3] employs the "training sets" approach to determine performance characteristics of collective communication routines, and then uses that information to predict performance of the parallel program. Fahringer's *Parameter based Performance Prediction Tool* [5, 6] uses profiling to obtain sequential program parameters, and then determines parallel program parameters that include measurements on load distribution, communication overhead, and data locality. The availability of detailed information about the generation of communication enhances the accuracy of cost estimation. However, a drawback of using these tools to guide data partitioning decisions is that the compiler has to generate the SPMD program with explicit communication for *each* data partitioning scheme to be evaluated. This can be prohibitively expensive.

A number of other researchers also analyze the source program without explicit message-passing to guide data partitioning decisions. Anderson and Lam [2] recognize nearest-neighbor shifts, data-reorganization communication, and pipelined communication. They use qualitative distinctions between the cost of these primitives while determining data and computation partitions. Chatterjee et al. [4] use cost measures representing the volume of communication, and ignore the effect of optimizations like aggregation of communication. In contrast to these approaches, we obtain more detailed estimates in terms of the size of data and the number of processors involved in each communication in the program. We show that this level of detail is often required for data partitioning decisions, particularly those determining the number of processors on which different array dimensions should be distributed.

Wholey [28] uses cost functions for various communication primitives on the target machine to estimate communication overheads for programs written in the ALEXI language. He deals with a simpler problem, since the ALEXI programs already have calls to high level primitives that correspond closely to collective communication operations.

Our research extends the work of Li and Chen [17], who use pattern-matching on array references to generate high-level collective communication. They deal with a simpler, single-assignment language, and their techniques are suited to a relatively restricted class of references. For instance, they do not handle references with coupled subscripts very well. We have introduced the notion of synchronous properties between array references, that enables the compiler to perform a more sophisticated analysis.

Organization of the Paper The rest of the paper is organized as follows. Section 2 presents a framework for determining some basic properties of data references, that help characterize the data movement involving multiple instances of those references. Section 3 describes how the compiler determines the placement of communication for a reference, that also governs the extent to which communication for multiple instances of that reference can be aggregated. Section 4 describes how the overall cost of communication for a reference is determined. Section 5 presents experimental results on automatic data partitioning for some programs, that show the importance of using good performance estimates to guide those decisions. Finally, Section 6 presents conclusions and ideas for future work.

2 Characterization of Data Movement

In this section, we describe properties of array references, which characterize the nature and the extent of data movement that they are involved in. We first describe how the arrays may be distributed, and how the cost of data movement is expressed. While in this paper we shall describe the estimation of communication costs only for arrays, our techniques are also applicable to communication involving scalar variables. Most compilers either replicate or privatize scalar variables. The PTRAN II compiler chooses between replication and *non-replicated alignment* of a scalar with an array reference [12], that covers privatization and mapping to a single processor as special cases. In either case, the distribution of scalars is subsumed by the possible distributions that we consider for arrays. Therefore, our techniques for characterizing communication involving arrays can be used for scalars as well.

2.1 Distribution of Arrays

In languages like HPF [8], the mapping of arrays is specified at two levels: arrays are aligned with templates or other arrays, those alignment targets are further distributed onto processors. Eventually, each array dimension is either collapsed or distributed over a processor grid dimension via (i) partitioning in a block, cyclic, or block-cyclic manner, (ii) replication, or (iii) mapping to a constant processor position [12]. We do not initially consider replication or mapping of an array dimension to a constant processor position. In Section 4.1.6, we describe how the results for these cases are obtained as straightforward extensions to the analysis shown in the rest of the paper. Given the form of alignment and distribution directives allowed by HPF, the mapping function of a partitioned array dimension, A_k (the kth dimension of array A), can be expressed as:

$$f(A_k, i) = \left\lfloor \frac{s_k * i - o_k}{b_k} \right\rfloor [\mathrm{mod} N_K]$$

This function returns the processor position to which the ith element along the array dimension is mapped. The term $\mathrm{mod} N_K$ is needed only for cyclic and block-cyclic distributions, that we shall collectively refer to as *general cyclic* distributions. The parameters s_k and o_k are referred to as the stride and offset of the distribution, while b_k is referred to as the block size.

2.2 Communication Primitives

The compiler estimates communication costs in terms of the time taken to carry out the following primitives. These primitives are similar to the collective communication primitives supported by the Message Passing Interface [19], an emerging standard for message passing on parallel machines.

- `Transfer or Send-Receive` : a single source processor sends a message to a single destination processor.

- `OneToManyMulticast` : a single source processor sends a message to all other processors in a group.
- `ManyToManyMulticast` : all processors in the group send data to all other processors in a group.
- `Scatter` : a single source processor sends *different* messages to all other processors in a group.
- `Gather` : all processors send messages to a single destination processor in a group.
- `Shift` : circular shift of data among adjacent processors in a group.
- `Reduction` : reduction of data using a simple associative and commutative operator, over all of the processors in a group.

Figure 3 illustrates the data movement associated with each routine. These primitives serve as a convenient form for representing communication costs, since they capture data movement in a fairly detailed, yet machine-independent manner. Even if these primitives (other than Send-Receive) are not supported, or are not used by the compiler in generating communication, information conveyed by their representation about the one-to-many, many-to-one, or many-to-many form of communication leads to better precision in global cost estimation.

The cost of a Transfer (Send-Receive) operation on m bytes for most distributed memory machines can be approximated as:

$$\text{Transfer}(m) = \nu + \lambda * m,$$

where ν and λ are, respectively, the start-up cost and the per-element cost of sending a message, and they may sometimes take different values depending on the message length. In this paper, we ignore the effect of distance between the processors on communication cost. This assumption is quite reasonable for most modern machines. For simplicity, we also ignore the overlap between communication and computation, and the buffer copying overhead associated with sending non-contiguous data.

The cost of collective communication operations can vary considerably in form across different implementations. Table 1 lists the cost functions used in PARADIGM corresponding to an implementation using Transfer operations, described below. The parameter p denotes the number of processors over which the primitive is carried out. Both Reduction and OneToManyMulticast take $\log_2(p)$ steps with tree-based algorithms. A Shift operation involves each processor sending data to its neighboring processor, and receiving data from another neighboring processor in the other direction, and hence is modeled as taking time equivalent to two Transfers. A ManyToManyMulticast over a group of p processors is done via $p-1$ Shifts using a ring-based algorithm. A Scatter/Gather operation is implemented as a sequence of Transfers, where the source/destination processor sends/receives different pieces of data to/from the $p-1$ other processors. These cost measures can be modified appropriately when the algorithms used are different.

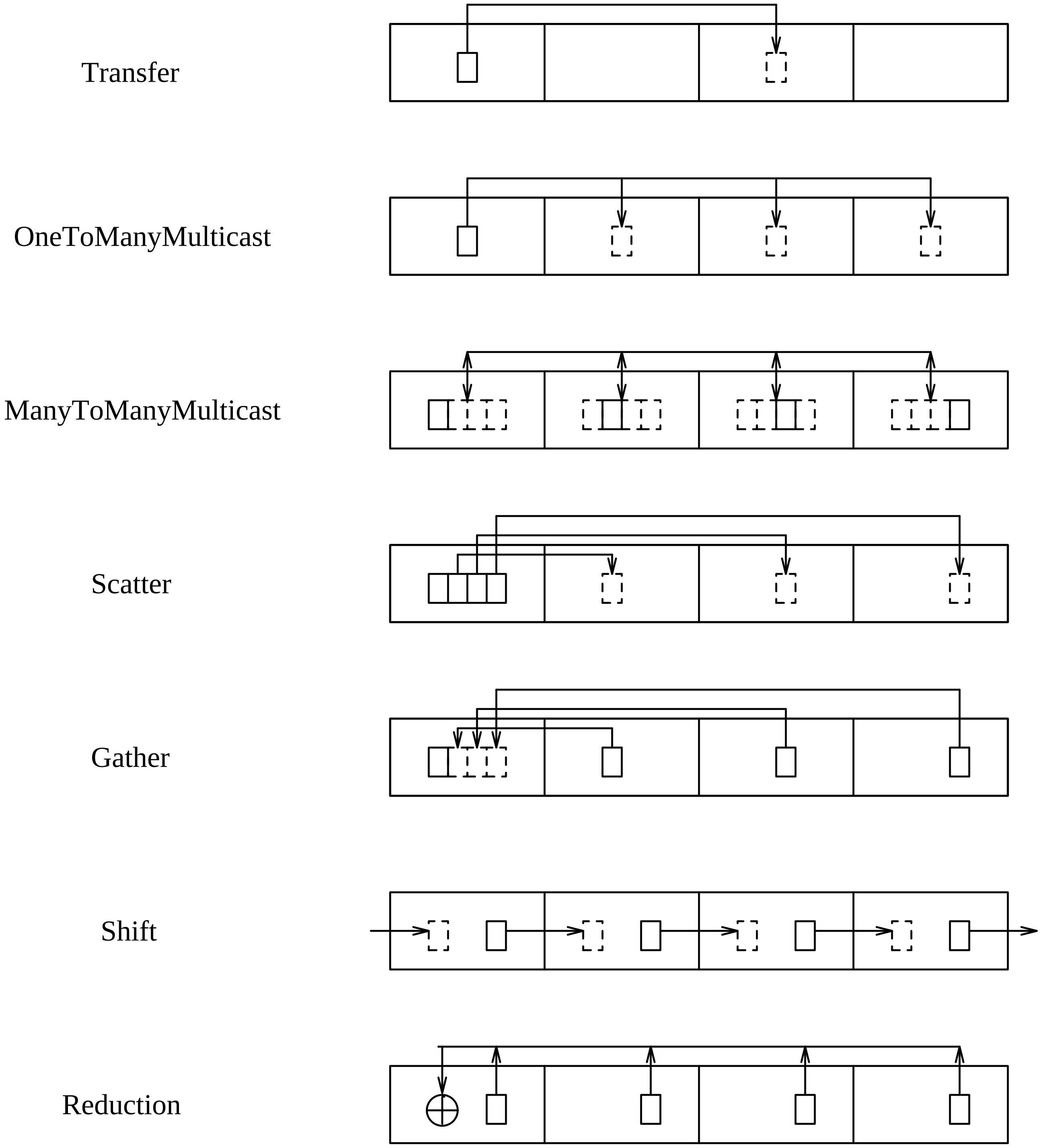

Figure 3: Communication primitives recognized by PARADIGM

Primitive	*Cost*
Reduction(m, p)	$\lceil \log_2(p) \rceil *$ Transfer(m)
Shift(m)	$2*$ Transfer(m)
OneToManyMulticast(m, p)	$\lceil \log_2(p) \rceil *$ Transfer(m)
ManyToManyMulticast(m, p)	$(p-1)*$ Shift(m)
Scatter(m, p)	$(p-1)*$ Transfer(m)
Gather(m, p)	$(p-1)*$ Transfer(m)

Table 1: Cost functions of collective communication primitives under a simple implementation

2.3 Characterization of Array References

PARADIGM analyzes the expression corresponding to each subscript, and assigns it to one of the following types: `constant`, `single-index` (affine function of a single loop induction variable), `multiple-index` (affine function of more than one loop induction variable), or `complex`. Information is also kept about its `variation-level`, i.e., the innermost loop in which the subscript changes values. In order to simplify cost estimation, all loops are normalized prior to the analysis.

We refer to a subscript associated with a distributed dimension in an array reference as a *sub-reference*. A sub-reference can be represented by a tuple $\langle r, d \rangle$, where r is an array reference, and d is the position ($d \geq 1$) of the subscript in that reference. A sub-reference varying inside a loop can be viewed as traversing a sequence of elements distributed on different processors along a grid dimension. Figure 4 shows the traversal of two sub-references, $\langle A(i, c_1), 1 \rangle$ and $\langle B(c_2, i), 2 \rangle$, by solid lines. The set of processor positions in a grid dimension traversed by a sub-reference sr in a loop L is referred to as its *processor position set* with respect to the loop, denoted as $\texttt{PPS}(sr, L)$. Similarly, the set of subscript positions along the array dimension traversed on the memory of a processor at position p is referred to as its *subscript position set*, denoted as $\texttt{SPS}(sr, L, p)$.

The number of processor positions traversed by the sub-reference, i.e., the size of its processor position set is referred to as $\texttt{NPP}(sr, L)$. The number of subscript positions traversed on a particular processor position is referred to as $\texttt{NSP}(sr, L, p)$. Another useful quantity to compute is the maximum size of any subscript position set of a sub-reference with respect to a loop, denoted as $\texttt{MNSP}(sr, L)$. It is defined as the maximum value of $\texttt{NSP}(sr, L, p)$ for all p belonging to $\texttt{PPS}(sr, L)$. For the sub-reference $sr = \langle A(i, c_1), 1 \rangle$ in Figure 4, it can be seen that $\texttt{NPP}(sr, L) = 4$, and $\texttt{MNSP}(sr, L) = \lceil n/4 \rceil$.

Each point in the loop (identified by the value of loop index) at which the sub-reference crosses a processor boundary in that grid dimension is called a *transition point* of the loop for that sub-reference. We now define some properties describing the relationships between sub-references varying inside the same loop, that help characterize the data movement for that loop. That, in turn, helps identify the communication primitive that is suitable for implementing the data movement.

```
do i = 1, n
    A(i, c1) = B(c2, i)
enddo
```

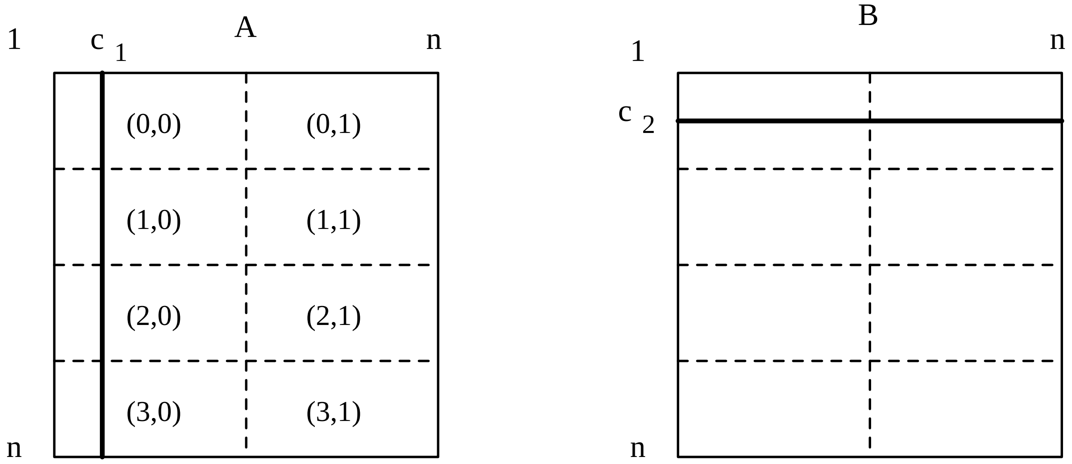

Figure 4: Traversal of sub-references in a loop

k-Synchronous Property A sub-reference sr_1 is said to be *k-synchronous* (k is a positive integer) with another sub-reference sr_2, with respect to a loop L, if (i) every transition point of L for sr_2 coincides with a transition point of L for sr_1, and (ii) between every two consecutive transition points of L for sr_2, there are exactly $k-1$ transition points of L for sr_1.

Example: In Figure 4, the sub-reference $\langle A(i, c_1), 1\rangle$ is 2-synchronous with $\langle B(c_2, i), 2\rangle$, with respect to the i-loop.

Strictly Synchronous Property A sub-references sr_1 is said to be *strictly synchronous* with another sub-reference sr_2, with respect to a loop L, if both sub-references are mapped to the same processor position during each iteration of L. We observe that this implies that (i) sr_1 is 1-synchronous with sr_2, with respect to L (i.e., every transition point of L for sr_1 is also a transition point of L for sr_2, and vice versa), and (ii) the coinciding transition points represent the cross-over points between the same processor positions in the respective grid dimensions for those sub-references.

Example: In Figure 4, if B_2 were distributed on four processors (rather than two), the sub-reference $\langle A(i, c_1), 1\rangle$ would be strictly synchronous with $\langle B(c_2, i), 2\rangle$, with respect to the i-loop.

A useful convention adopted regarding the k-synchronous property is that k can also be a reciprocal of a positive integer. In that case, a statement that sr_1 is k-synchronous with sr_2 is interpreted as conveying that sr_2 is $1/k$-synchronous with sr_1.

2.4 Static Determination of Characteristics

Given information about the distribution of arrays, we now show how the compiler determines the above characteristics of sub-references. All of the results presented in this section are for sub-references of the type `single-index`. The corresponding analysis for sub-references of the type `multiple-index` is based on these results, and for sub-references of the type `complex` cannot be done precisely at compile-time. We present results for the general case of block-cyclic distribution, of which block and cyclic distributions are special cases.

2.4.1 Traversal of Sub-references

Consider a sub-reference sr of the type `single-index`, with subscript expression $e = \alpha * j_l + \beta$, varying in a (normalized) loop L_l (loop at level l) with a range of r_l. Let α' denote $|\alpha|$, and let the corresponding array dimension A_k have a distribution function shown in Equation 1. The number of processor positions traversed by sr in L_l is:

$$\mathtt{NPP}(sr, L_l) \quad = \quad \min(N_k, \lceil (s_k \alpha' r_l)/b_k \rceil) \tag{1}$$

The second term in the above expression determines the number of data blocks traversed by the sub-reference, which also gives the number of processors traversed, in the case of block distribution. For general cyclic distributions, the above expression takes into account the wrap-around of data blocks over N_k processors.

The maximum number of subscript positions, over any processor, traversed by sr in L_l is estimated as:

$$\begin{aligned} \mathtt{MNSP}(sr, L_l) \quad = \quad & (\lfloor b_k/(s_k \alpha') \rfloor * \lfloor (s_k \alpha' r_l)/(N_k b_k) \rfloor) + \\ & \min(\lfloor b_k/(s_k \alpha') \rfloor, \lceil ((s_k \alpha' r_l) \bmod (N_k b_k))/(s_k \alpha') \rceil) \end{aligned} \tag{2}$$

The traversal of the sub-reference conceptually consists of two regions: region P, which is equally distributed among processors, and region Q, which covers only some of the processors. The term $\lfloor (s_k \alpha' r_l)/(N_k b_k) \rfloor$ denotes the number of complete blocks covered in region P on each processor. The term $\lfloor b_k/(s_k \alpha') \rfloor$ denotes the number of subscript positions visited in a single block of A_k on a processor. Thus, the product of these two terms (the first additive term in the above expression) represents the contribution corresponding to the region P. If the extent of region Q exceeds the block size of A_k, the maximum number of additional subscript positions mapped to a processor is $\lfloor b_k/(s_k \alpha') \rfloor$. Otherwise, that additional number is $\lceil ((s_k \alpha' r_l) \bmod (N_k b_k))/(s_k \alpha') \rceil$, i.e., the total number of subscript positions corresponding to the region Q.

2.4.2 Synchronous Properties

We now present tests to detect the synchronous properties for a pair of sub-references. The corresponding array dimensions must be distributed in the same manner – block or general cyclic, and in case of general cyclic distribution, on an equal number of processors. Let the subscript expressions of the two sub-references, sr_1 and sr_2, be $e_1 = \alpha_1 * i + \beta_1$, and $e_2 = \alpha_2 * i + \beta_2$. Let b_1 and b_2 be the block sizes, s_1 and s_2 be the strides, and o_1 and o_2 be the offsets of distribution of the corresponding array dimensions.

Strictly synchronous sub-references The following theorem gives sufficient conditions for the strictly synchronous property.

Theorem 1 *Let sr_1 and sr_2 be two sub-references, as described above. The sub-reference sr_1 is strictly synchronous with sr_2, with respect to loop L if either of the following sets of conditions are satisfied:*

- *(i) $s_1\alpha_1/b_1 = s_2\alpha_2/b_2$, and (ii) $(s_1\beta_1 - o_1)/b_1 = (s_2\beta_2 - o_2)/b_2$,*
  or
- *(i) $b_1 = ms_1\alpha_1$, (ii) $b_2 = ms_2\alpha_2$, and (iii) $\lfloor(s_1\beta_1 - o_1)/s_1\alpha_1\rfloor = \lfloor(s_2\beta_2 - o_2)/s_2\alpha_2\rfloor$,*
  where m is a positive integer.

Proof Given in the appendix.

k-synchronous sub-references The conditions to check whether sr_1 is k-synchronous with sr_2 are obtained in a similar manner, and are shown below:

Theorem 2 *Let sr_1 and sr_2 be two sub-references as described above. The sub-reference sr_1 is k-synchronous (k being an integer) with sr_2, with respect to loop L if either of the following sets of conditions are satisfied:*

- *(i) $s_1\alpha_1/b_1 = k * (s_2\alpha_2/b_2)$, and (ii) $(s_1\beta_1 - o_1)/b_1 = k * ((s_2\beta_2 - o_2)/b_2) + l$, where l is an integer,*
  or
- *(i) $b_1 = ms_1\alpha_1$, (ii) $b_2 = k * (ms_2\alpha_2)$, and (iii) $\lfloor(s_1\beta_1 - o_1)/s_1\alpha_1\rfloor = \lfloor(s_2\beta_2 - o_2)/s_2\alpha_2\rfloor + ml$,*
  where m and l are integers, and $m > 0$.

Proof Omitted, similar to the proof of Theorem 1.

2.4.3 Other Methods of Characterization

During the analysis of communication requirements, if the compiler is unable to establish a k-synchronous property between two sub-references of the type `single-index`, it uses one of the following two measures to characterize the relationship between the two sub-references. The first measure is the *speed-ratio*, defined as:

$$\texttt{SpRatio}(sr_1, sr_2, L) = \lceil (s_1\alpha_1 b_2)/(s_2\alpha_2 b_1) \rceil \quad (3)$$

This captures roughly, the "speed" with which the sub-reference sr_1 crosses processor boundaries in the loop L, relative to the sub-reference sr_2. This measure is a generalization of the notion of k-synchronous property, and it may be observed that a k-synchronous property between two sub-references for a given loop implies a `speed-ratio` of k.

Boundary-communication The "boundary-communication" is a specialized test performed between sub-references corresponding to aligned dimensions, and related by a `speed-ratio` of one. This test helps detect data movement taking place across boundaries of regions allocated to neighboring processors. It checks for the following conditions:

1. $s_1\alpha_1/b_1 = s_2\alpha_2/b_2$.
2. $|(s_1\beta_1 - o_1)/b_1 - (s_2\beta_2 - o_2)/b_2| \leq 1$.

If the above conditions are satisfied, the amount of data transfer across the boundary of each participating processor is estimated as:

$$\mathtt{BD}(sr_1, sr_2, L) = (\lceil|(s_1\beta_1 - o_1)/(s_1\alpha_1) - (s_2\beta_2 - o_2)/(s_2\alpha_2)|\rceil) * (\lceil(s_2\alpha_2 r_l)/(N_k b_2)\rceil) \quad (4)$$

The first term, $\lceil|(s_1\beta_1 - o_1)/(s_1\alpha_1) - (s_2\beta_2 - o_2)/(s_2\alpha_2)|\rceil$ gives the number of elements transferred across a single block of the array dimension. The second term in the product gives the number of blocks (among those traversed by the sub-reference sr_2) that are held by a single processor.

2.4.4 Handling Statically Unknown Parameters

The analysis described in this section assumed a complete knowledge of program parameters, such as loop bounds, sizes of arrays and their distribution parameters. Some of the analysis can be performed symbolically, or by reformulating the results differently. For instance, consider the test for strictly synchronous property between a pair of sub-references. Even if the block sizes are not known statically due to the number of processors being unknown, it may be known that the two block sizes are the same because of the arrays having identical shapes. The first set of conditions in Theorem 1 would simplify to $s_1\alpha_1 = s_2\alpha_2$ and $s_1\beta_1 - o_1 = s_2\beta_2 - o_2$. Similarly, the test for b_1 being an integer multiple of $s_1\alpha_1$ can be structured so as to first check for the commonly occurring case of $s_1 = \alpha_1 = 1$, in which case the test is satisfied trivially.

For program parameters that are essential for determining the traversal properties, PARADIGM queries the user for values to be used in its estimates. This method was chosen primarily for the ease of implementation. A more advanced method for determining unknown parameters such as loop bounds would be to use profiling [25, 6]. However, profiling is sometimes not feasible because of the time required to run the program sequentially for meaningful data sizes.

```
do j_1 = 1, r_1
   . . .
      do j_l = 1, r_l
         ⟨ communication for B(h_1, ..., h_q)⟩
         do j_{l+1} = 1, r_{l+1}
            . . .
               do j_m = 1, r_m
                  A(g_1, g_2, ..., g_p) = F(B(h_1, h_2, ..., h_q))
```

Figure 5: Statement requiring communication of data

3 Placement of Communication

Consider a reference appearing in a statement inside one or more loops, as shown in Figure 5. If possible, the communication for that reference should be moved outside of loops to enable *message vectorization*, which combines a sequence of messages on individual array elements into a single message [13, 31]. The compiler examines all incoming flow dependences to the reference, and the innermost loop carrying any of those dependences identifies the loop from which communication cannot legally be moved outside.

3.1 Loop Transformations for Combining Messages

Prior to determining the placement of communication while estimating costs, PARADIGM considers loop transformations that expose more opportunities for combining communication. The application of transformations is a compiler-dependent feature, and PARADIGM tries to incorporate some important optimizations that have been mentioned in the literature [31, 13, 2, 12]. When used along with a compiler that performs/does not perform certain transformations, the estimation algorithm can be modified appropriately. While estimating communication costs, PARADIGM does not consider transformations for improving data locality, such as loop fusion or loop interchange that is geared towards better cache performance. These transformations have a more significant impact on computational costs, and can usually be accomodated by first moving communication outside using the transformations we shall describe, and then transforming the inner loops for better locality [12].

It is often useful to defer the actual application of transformations for communication or even the decision to apply them, until data partitioning decisions are made, since their utility may itself depend on data partitioning. Hence, we have simply incorporated legality checks for these transformations in the algorithm for determining placement of communication in the cost estimator of PARADIGM [9]. The algorithm is described in Figure 6.

```
CommunicationLevel(R, S)
{
    lmax = 0
    for each dependence into S do
        if ((dependence-type == flow) and (dependence due to reference R))
            S' = source statement of dependence
            determine k = nesting level of dependence
            for (l = k;  l >= 1;  l--) do
                if S and S' belong to same strongly connected component in dependence graph
                at level l
                    break
                endif
            endfor
            lmax = max(l, lmax)
        endif
    endfor
    Identify the doall loops outside loop L_lmax as loops to be brought inwards.
}
```

Figure 6: Algorithm to determine communication level for reference R in statement S

```
do i = 1,n
    D(i) = D(i) + s * B(i)
    D(i) → owner(A(i,1 : n))
    do j = 1,n
        A(i,j) = A(i,j) + D(i)
    enddo
enddo
```

```
do i = 1,n
    D(i) = D(i) + s * B(i)
enddo
D(i) → owner(A(i,1 : n)), i = 1 : n
do i = 1,n
    do j = 1,n
        A(i,j) = A(i,j) + D(i)
    enddo
enddo
```

Figure 7: Loop distribution

Loop Distribution The distribution of a loop over two components enables any communication placed between those components to be aggregated with respect to that loop. For example, in the program segment shown in Figure 7, distributing the i-loop allows the communication involving D to be moved outside that loop. The algorithm shown in Figure 6 examines the legality of loop distribution by checking if the source and the target of a loop carried dependence belong to different strongly connected components in the dependence graph at that level [1].

Loop Permutations Whenever there is a parallel loop outside a loop in which communication takes place, the parallel loop should be brought inwards to ensure that communication is aggregated with respect to that loop. A loop permutation that brings a parallel loop inwards is always legal, and it cannot adversely affect the combining of communication for any other reference. In the program segment shown in Figure 8,

```
do i = 1, n
    do j = 1, n
        A(i,g(j)) → owner(A(i,j))
        A(i,j) = F[A(i,g(j))]
    enddo
enddo
```

```
do j = 1, n
    A(i,g(j)) → owner(A(i,j)), i = 1 : n
    do i = 1, n
        A(i,j) = F[A(i,g(j))]
    enddo
enddo
```

Figure 8: Loop interchange

```
do j = 1, n
    A(i,g(j)) → owner(A(i,j)), i = 1 : n
    do i = 1, n
        A(i,j) = F[A(i,g(j))]
    enddo
enddo
```

```
do j = 1, n
    do i' = 1, n, b
        A(i,g(j)) → owner(A(i,j)), i = i' : i' + b − 1
        do i = i', min(i' + b − 1, n)
            A(i,j) = F[A(i,g(j))]
        enddo
    enddo
enddo
```

Figure 9: Loop stripmining

communication for the `rhs` reference cannot be taken outside the j-loop, because of the potential flow dependence carried by that loop. Following loop interchange, which brings the outer parallel loop inwards, that communication can be vectorized.

In the above example, it is important to recognize the possibility of loop interchange while estimating communication cost when A_2 is partitioned across processors, even though loop interchange will not be needed if A_2 were collapsed in the finally chosen data partitioning scheme. Thus, it is useful to have the ability to account for the benefits of loop transformations that will be performed, without actually transforming the program.

3.2 Limits on Combining of Messages

The algorithm shown in Figure 6 determines the outermost loop level at which communication can be placed. However, the compiler may need to control the extent to which messages are combined, for instance, to limit the size of communication buffers required. This can be achieved by stripmining loops, as shown in Figure 9. On stripmining a loop with strip size b, the number of messages increases by a factor of b, but the total size of data communicated remains the same. Since stripmining is (or should be) done at large message sizes, when the start-up costs account for only a small fraction of the overall cost of sending messages, the overall effect of stripmining on communication costs can be reasonably ignored.

Another consequence of the above algorithm is that potentially, the compiler may generate more com-

```
do j = 1, n
    do i = 1, n
        if (s ≠ 0) then
            A(i, j) = D(i)/s
        end if
        ...
```

Figure 10: Extra communication in the presence of conditionals

munication than necessary. Consider the program segment shown in Figure 10. The communication for the reference to $D(i)$ would be placed outside the j-loop, and communication would be realized for all instances of that reference, even though the assignment statement may not be executed for all the loop iterations. Information about the frequency of execution of statements, possibly from profiling, can help the compiler decide between carrying out potentially extra communication and using a larger number of messages. However, we do not know of any compiler that examines this trade-off while generating communication. PARADIGM assumes that communication will be taken outside loops even in the presence of conditionals inside them.

4 Estimation of Communication Cost

This section describes how the overall cost of communication for a reference is estimated, after the compiler has determined the outermost level at which communication can be placed, as shown in Figure 5. We refer to the inner $m - l$ loops, with respect to which communication can be aggregated, as Type-1 loops, and to the outer l loops as Type-2 loops. A high-level overview of the algorithm for estimating cost for a reference, R, is given in Figure 11. The reference, L, refers to the data whose owner is assigned the computation corresponding to the statement. Usually, L is set to the left hand side (`lhs`) reference for an assignment statement, in accordance with the *owner computes* rule [13]. It is chosen differently for conditional statements, and when the compiler overrides the owner computes rule [12]. The basic idea of the algorithm in Figure 11 is to obtain a cost estimate of the sequence of communication primitives that implements the data movement required for the entire iteration space of Type-1 loops, and multiply it by the expected number of times those primitives would be executed in the program. The cost estimate of the sequence of primitives is derived by first obtaining cost terms corresponding to "one-dimensional" views of the data movement in each processor grid dimension, and then composing those terms together.

The first step of our algorithm matches pairs of sub-references corresponding to aligned array dimension. In that step, if the `lhs` and the `rhs` arrays differ in the number of dimensions, the array with fewer dimensions is augmented with dummy, "missing" dimensions, that match extra distributed dimensions of the other array. A "missing" sub-reference is regarded as one of the type `constant`.

```
CommunicationCostEstimate(R, L)
{
   Match pairs of sub-references of R and L corresponding to aligned array
   dimensions, and set variation-level of each pair to the maximum of
   variation-level values of the two sub-references.

   for (k = m;  k > l;  k-- ) do
      Obtain communication cost terms for all sub-reference pairs and for all rhs
      subscripts in collapsed dimensions, that have a variation-level of k [Section 4.1].

      Update the value of communication level, l, if indicated by the above step.
   endfor
   For each pair of sub-reference not yet considered, obtain communication cost term
   [Section 4.2].

   Compose all cost terms, and multiply them by count of repetitions [Section 4.3].
}
```

Figure 11: Algorithm to estimate cost of communicating data for reference R to owner of reference L

We now describe how communication cost terms are obtained for aggregate data movement corresponding to sub-references varying in a Type-1 loop, and for single data movement corresponding to other sub-references. We then show how these terms are composed to give an overall estimate of the communication cost.

4.1 Cost Terms for Aggregate Data Movement

The data movements required in different iterations of a Type-1 loop can legally be combined. Therefore, the compiler attempts to recognize the communication primitive that best realizes the collective movement, and obtains the term(s) representing its cost. The techniques we have developed allow this analysis to be carried out for arbitrary kinds of sub-references, and for any arbitrary number of pairs of sub-references varying in a loop. For ease of presentation, we shall describe different aspects of the analysis separately.

4.1.1 Single pair of varying sub-references

Table 2 lists the communication term(s) obtained if there is a single pair of sub-references varying in such a loop L. This table enumerates the cases corresponding to only the "basic" categories of the subscripts for the `lhs` and `rhs` sub-references. The results for the other cases (when the two subscripts of the type `single-index` have different values of `variation-level`, or when one of the subscripts is of the type `multiple-index`) are derived in terms of these results, and are presented later in this section. The column marked *conditions tested* lists the tests performed by the compiler to obtain further information about the nature of data movement. The entry *reduction op* represents the test to see if the `rhs` reference is involved

LHS (sr_1)	*RHS* (sr_2)	*Conditions Tested*	*Communication Term*
single-index	constant	default	OneToManyMulticast$(1, \mathtt{NPP}(sr_1, L))$
constant	single-index	1. reduction op 2. default	Reduction$(1, \mathtt{NPP}(sr_2, L))$ Gather$(\mathtt{MNSP}(sr_2, L), \mathtt{NPP}(sr_2, L))$
single-index	single-index	1. sr_1 strictly synch sr_2 2. sr_1 1-synch sr_2 3. boundary-commn. 4. sr_1 k-synch sr_2, $k > 1$ 5. sr_1 k-synch sr_2, $k < 1$	IDM$(\mathtt{MNSP}(sr_2, L))$ $(N_I > 1)*$ Transfer$(\mathtt{MNSP}(sr_2, L))$ (i) $(N_I > 1)*$ Shift$(\mathtt{BD}(sr_1, sr_2, L))$ (ii) IDM$(\mathtt{MNSP}(sr_2, L) - \mathtt{BD}(sr_1, sr_2, L))$ D-Scatter$(\mathtt{MNSP}(sr_2, L)/k, k)$ D-Gather$(\mathtt{MNSP}(sr_2, L), 1/k)$

Table 2: Communication terms for a pair of sub-references varying in a Type-1 loop

in a reduction operation (such as addition, max, min) inside the loop.

The case corresponding to both the sub-references being of the type `single-index` is perhaps the most commonly occurring pattern inside various loops in the scientific application programs. If the two sub-references satisfy one of the five tests shown in the table, it means that the processors can be partitioned into mutually disjoint groups over which the communication primitives indicated by the cost terms may be carried out in parallel. Figure 12 shows the different kinds of data movement corresponding to those cases. The term IDM(m) (internalized data movement of m elements) by itself does not represent any communication cost, it affects the data sizes of other communication cost terms during the composition step, as shall be discussed later. The third case corresponds to data movement that is partly intra-processor and partly inter-processor in the form of nearest-neighbor communication. Hence, it is represented by a Shift and an IDM term. In the fourth case, the symbol *D-Scatter* represents a Scatter operation over a "different" group, i.e., a group that does not include the source processor sending data values. A D-Scatter operation involves data being sent to p other processors, rather than $p - 1$ (this distinction is important for accuracy when p takes a small value). Thus, the term D-Scatter(m, p) is simply short-hand for the following two terms: (i) $(N_I > 1) *$ Transfer(m), and (ii) $(N_I > 1) *$ Scatter(m, p), where N_I is the number of processors on which the aligned array dimensions are distributed. The term D-Gather(m, p) is defined in a similar manner.

If none of the five tests indicated in the table are satisfied, the compiler computes the value of $\mathtt{SpRatio}(sr_1, sr_2, L)$ using Equation 3. Similar to the last two entries for the k-synchronous property, a D-Scatter or a D-Gather term is used depending on the whether the value of `speed-ratio` is greater than or less than one. However, in this case, the Scatter or the Gather operations corresponding to different groups do not take place entirely in parallel, since there is an overlap of processors in those groups. This overlapping between groups, and the resulting contention during the process of communication is modeled by multiplying each of those terms (represented by D-Scatter or D-Gather) by a factor of two.

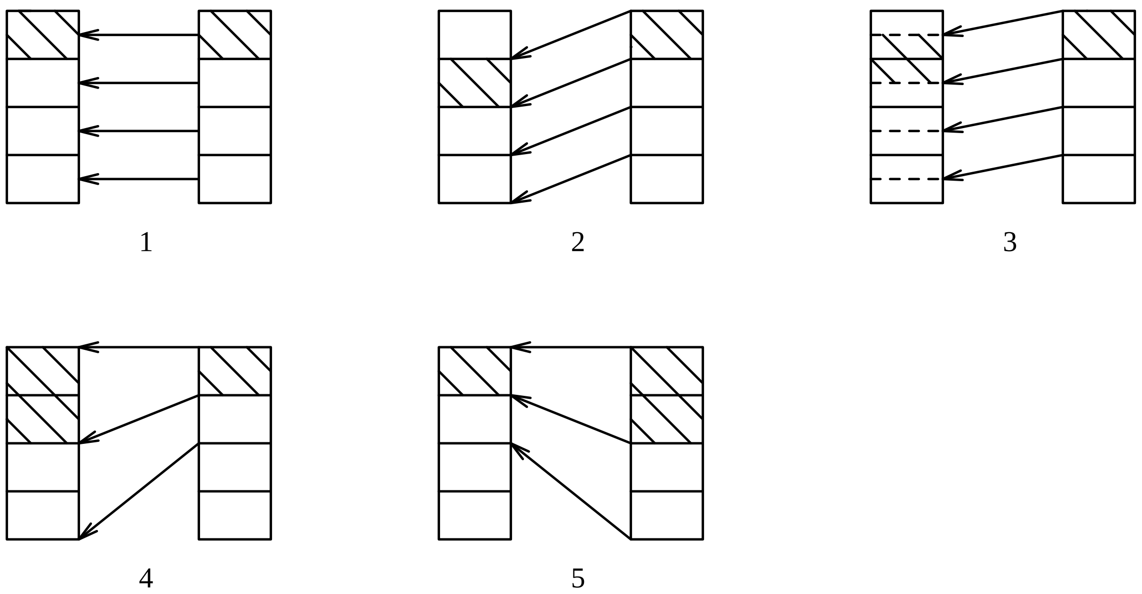

Figure 12: Data movements for sub-references of the type `single-index`

4.1.2 Multiple pairs of varying sub-references

When there is more than one pair of sub-references varying in a Type-1 loop, the analysis shown above is insufficient. For example, consider the program segment shown in Figure 13, where both the arrays A and B are distributed in a blocked manner on a grid with N_1 x N_2 processors. Depending on the relative values of N_1 and N_2, the best choice of communication primitive may be Scatter, Transfer, or Gather. It is not possible to determine the appropriate communication cost term just by analyzing the two pairs of sub-references individually.

The compiler has to analyze all of the sub-references varying in a loop together, to infer the relationship between simultaneous traversals of those sub-references in different processor grid dimensions. The choice of primitives is determined by the relationship between the fastest-varying `rhs` sub-reference and the fastest-varying `lhs` sub-reference. Table 3 presents some of the communication cost terms when there are two pairs of sub-references varying in a loop L. The unnumbered properties listed under the *conditions tested* column are those which must be satisfied, before an appropriate cost term is chosen, based on the numbered condition. The function `fv` chooses the faster-varying sub-reference between two given sub-references, i.e., if $\texttt{SpRatio}(s_1, s_2, L) = k$, $\texttt{fv}(s_1, s_2)$ is set to s_1 if $k \geq 1$, and otherwise to s_2. The symbol k_m appearing in various cost terms refers to the value of $\max(k_1, k_2)$. The symbols N_f and N_s refer to the number of processors in the two grid dimensions corresponding to the "faster-varying" and the "slower-varying" `rhs` sub-references respectively. The unnumbered conditions in the table ensure that all the processors in the two grid dimensions can be partitioned into mutually disjoint groups over which the indicated primitives can be carried out in parallel.

```
do i = 1, n
      A(i, c1) = F(B(c2, i))
enddo
```

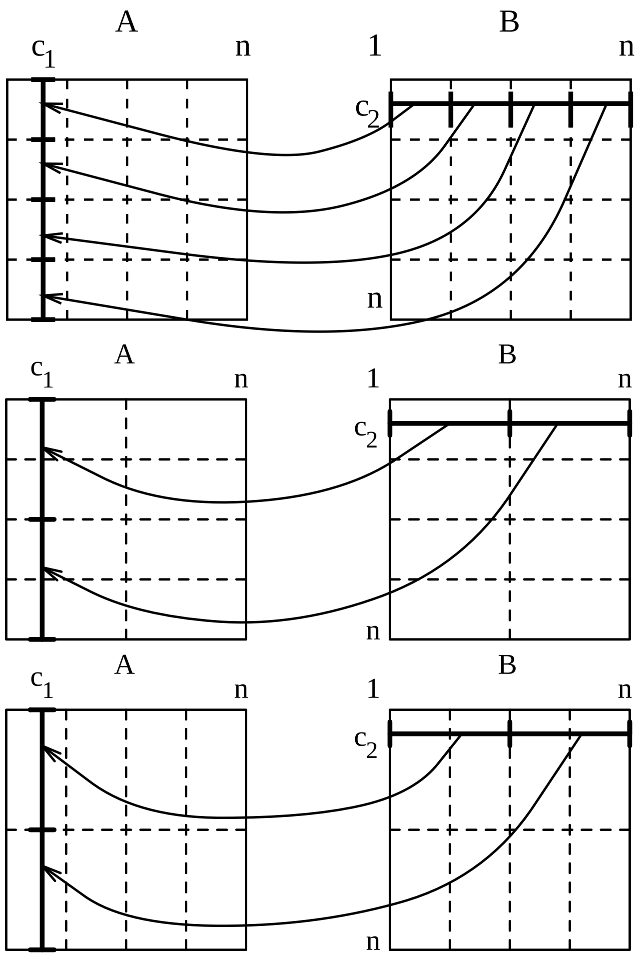

Figure 13: Variation of multiple pairs of sub-references in a loop

If the given sub-references do not satisfy the conditions shown in the table, the compiler first selects the `lhs` sub-reference with the higher `speed-ratio`, and similarly, the faster varying `rhs` sub-reference. Now, the `speed-ratio` is determined for those two sub-references, and the communication cost term obtained, as described earlier for the case of a single pair of sub-references varying in the loop.

Example For the example in Figure 13, the choice of communication cost term is governed by the test for k-synchronous property between sub-references $\langle A(i, c_1), 1\rangle$, and $\langle B(c_2, i), 2\rangle$, as shown in Table 3. For the three cases corresponding to 4 x 4, 4 x 2, and 2 x 4 processor grids, $\langle A(i, c_1), 1\rangle$ is determined to be 1-synchronous, 2-synchronous and 1/2-synchronous, respectively, with $\langle B(c_2, i), 2\rangle$, with respect to the i-loop. Hence, the compiler chooses the terms Transfer($\lceil n/4\rceil$), D-Scatter($\lceil n/4\rceil, 2$) and D-Gather($\lceil n/4\rceil, 2$)

LHS	*RHS*	*Conditions Tested*	*Communication Term*
single-index (s_1) single-index (s_3)	single-index (s_2) constant (s_4)	s_1 k_1-synch s_2, s_3 k_2-synch s_2, s_1 k_3-synch s_3. 1. $max(k_1, k_2) > 1$ 2. $max(k_1, k_2) = 1$ 3. $max(k_1, k_2) < 1$	 D-Scatter($\texttt{MNSP}(s_2, L)/k_m, k_m$) $(N_f > 1) *$ Transfer($\texttt{MNSP}(s_2, L)$) D-Gather($\texttt{MNSP}(s_2, L), 1/k_m$)
single-index (s_1) single-index (s_3)	single-index (s_2) single-index (s_4)	s_1 k_1-synch s_2, s_3 k_2-synch s_4, s_1 k_3-synch s_3, s_2 k_4-synch s_4. 1. s_1 strictly synch s_2, s_3 strictly synch s_4. 2. $max(k_1, k_2) > 1$ 3. $max(k_1, k_2) = 1$ 4. $max(k_1, k_2) < 1$	 IDM($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L)$) D-Scatter($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L)/k, k$) $(N_f > 1) *$ Transfer($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L)$) D-Gather($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L), 1/k$)
single-index (s_1) constant (s_3)	single-index (s_2) single-index (s_4)	s_1 k_1-synch s_2, s_1 k_1-synch s_4, s_2 k_3-synch s_4. 1. $max(k_1, k_2) > 1$ 2. $max(k_1, k_2) = 1$ 3. $max(k_1, k_2) < 1$	 D-Scatter($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L)/k_m, k_m$) $(N_f > 1) *$ Transfer($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L)$) D-Gather($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L), 1/k$)
single-index (s_1) single-index (s_3)	constant (s_2) constant (s_4)	s_1 k-synch s_3	(i) $(1 - 1/N_s) *$ Transfer(1) (ii) OneToManyMulticast(1, $\texttt{NPP}(\texttt{fv}(s_1, s_3), L)$)
single-index (s_1) constant (s_3)	constant (s_2) single-index (s_4)	s_1 k-synch s_4. 1. $k > 1$ 2. $k = 1$ 3. $k < 1$	 D-Scatter($\texttt{MNSP}(s_4, L)/k, k$) $(N_f > 1) *$ Transfer($\texttt{MNSP}(s_4, L)$) D-Gather($\texttt{MNSP}(s_4, L), 1/k$)
constant (s_1) constant (s_3)	single-index (s_2) single-index (s_4)	s_2 k-synch s_4 1. reduction op 2. default	(i) $(1 - 1/N_s) *$ Transfer(m) (ii) Reduction($1, \texttt{NPP}(\texttt{fv}(s_2, s_4), L)$) (ii) D-Gather($\texttt{MNSP}(\texttt{fv}(s_2, s_4), L)$, $\texttt{NPP}(\texttt{fv}(s_2, s_4), L)$)

Table 3: Communication terms for two pairs of sub-references varying in a Type-1 loop

respectively in those cases.

4.1.3 General Cyclic distributions

All of the analysis we have described above is valid for both kinds of array distributions, block and general cyclic (cyclic and block-cyclic). For general cyclic distributions, however, another condition is added to the tests for regularity of data movement. For every sub-reference with a subscript of the form $e = \alpha_1 j_l + \beta_1$, the compiler checks if b_1 is a multiple of $s_1 \alpha_1$, where s_1 is the stride and b_1 is the block size of distribution of the given dimension. The satisfaction of this condition ensures that the data elements involved in any collective communication corresponding to the given sub-reference can be accessed on the local memories of involved processors with a constant stride. Otherwise, the compiler uses cost terms corresponding to communication with a degraded performance.

4.1.4 Different/multiple loop indices

When the subscripts corresponding to a matched pair of sub-references involve different loop indices, or when one of the subscripts is of the type `multiple-index`, one simple way to analyze the data movement in terms of our earlier results would be to "freeze" (i.e., regard as Type-2) all relevant loops except for the innermost one. This would involve placing communication inside those frozen loops, and treating the corresponding loop indices as constants for the purpose of analysis. PARADIGM uses an extension of this idea, with *tiling* [29] instead of freezing of one of the outer Type-1 loops, so that communication may be combined with respect to at least the tiles of that loop.

Sub-references with different loop indices Let us first describe the analysis for a pair of sub-references of the type `single-index`, but with different values of `variation-level`. Consider sub-references sr_1 and sr_2 with subscript expressions of the form $e_1 = \alpha_1 j_q + \beta_1$, $e_2 = \alpha_2 j_p + \beta_2$, where $q > p$. If L_p is a Type-2 loop, then sr_2 is regarded as a sub-reference of the type `constant`, and the data movement is analyzed for loop L_q in a normal manner. However, if L_p is a Type-1 loop, it is assumed to be tiled suitably by the compiler. (The compiler generating communication would tile it such that the starting points of the tiles are precisely the transition points of the loop L_p for the sub-reference sr_2). In the context of this paper, we shall limit our discussion to just the part dealing with cost estimation.

The value of j_p is regarded as a constant for the purpose of analysis of communication requirements, as shown in Table 2. However, the cost term obtained is modified in the following manner to model the effect of tiling of the loop L_p. There are two cases:

1. *The sub-reference sr_2 appears on the* `rhs`: The data size of the cost term is multiplied by the tile size, given by $\texttt{MNSP}(sr_2, L_p)$, to model the effect of combining of communication with respect to the tiles of the loop L_p. Further, the term itself is multiplied by $\texttt{NPP}(sr_2, L_p)$ to account for the repetitions corresponding to the number of tiles. In fact, in this case, it is known that the cost term obtained initially is $\text{OneToManyMulticast}(1, \texttt{NPP}(sr_1, L_q))$. It is changed to $\texttt{NPP}(sr_2, L_p) * \text{OneToManyMulticast}(\texttt{MNSP}(sr_2, L_p), \texttt{NPP}(sr_1, L_q))$.

2. *The sub-reference sr_2 appears on the* `lhs`: The cost term obtained is multiplied by $\texttt{NPP}(sr_2, L_p)$ in this case too. However, the data size of the original term is left unchanged (the data corresponding to the `rhs` sub-reference is re-used in different iterations that are part of the same tile).

Some additional analysis is required when there are other pairs of sub-references that vary inside the loop L_p. Consider another pair of sub-references sr_3 and sr_4, both of the type `single-index` with a `variation-level` of m. This time, the analysis proceeds along the lines of that shown in Table 3. The compiler checks for the k-synchronous property between sr_2 and sr_3, and between sr_2 and sr_4, with respect to L_p. Tiling is used

```
do j = 1, n
     do i = 1, n
          A(i, j) = B(j, j)
     enddo
enddo
```

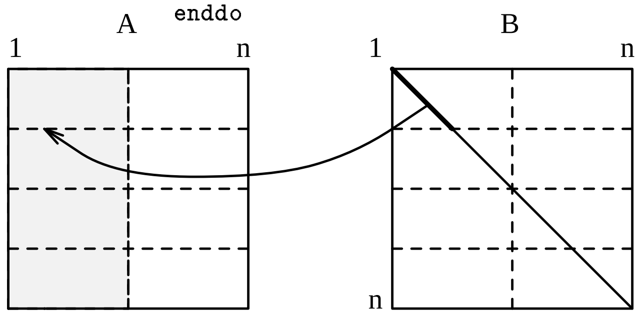

Figure 14: Example to illustrate tiling of outer loop

only if the k-synchronous property holds between each of the above pairs. The fastest-varying sub-reference among sr_2, sr_3, and sr_4 is selected to determine the tile size. The communication cost term is now modified using the method shown above. The following example illustrates this process.

Example Consider the statement and the associated data movement for a 4 x 2 processor grid shown in Figure 14. The first pair of aligned dimensions have subscripts i and j varying in different loops. The j-loop is tiled, and j is regarded as a constant for the purpose of obtaining the initial term, OneToManyMulticast(1,4). There is another pair of sub-references varying in the j-loop (L_1). The sub-reference $\langle B(j,j), 1\rangle$ is 2-synchronous with both $\langle B(j,j), 2\rangle$ and $\langle A(i,j), 2\rangle$, with respect to the j-loop. Hence, the tile size is set to $\texttt{MNSP}(\langle B(j,j), 1\rangle, L_1) = \lceil n/4 \rceil$. The number of tiles is given by $\texttt{NPP}(\langle B(j,j), 1\rangle, L_1) = 4$. Therefore, the communication term is changed to $4 * \text{OneToManyMulticast}(\lceil n/4 \rceil, 4)$.

Extension In the special case when there are two pairs of aligned sub-references, with the first pair being of the form $sr_1 = \alpha_1 j_q + \beta_1$, $sr_2 = \alpha_2 j_p + \beta_2$, and the second pair of the form $sr_3 = \alpha_3 j_p + \beta_3$, $sr_4 = \alpha_4 j_q + \beta_4$, the compiler can refine the analysis further. The data movement corresponds to a *transpose* if sr_1 is strictly synchronous with sr_4 with respect to loop L_q, and sr_2 is strictly synchronous with sr_3 with respect to loop L_p. The transpose operation is often used for changing the distribution of a two-dimensional array from row-wise distribution to column-wise distribution, or vice versa.

Sub-references with multiple loop indices For all the loops whose indices appear in a sub-reference of the type `multiple-index`, the compiler first determines those that are Type-1 loops. If there is only one such loop, the indices corresponding to the Type-2 loops are regarded as constants, and the sub-reference

```
do i = 1, n
     A(i) = ... B(D(i)) ...
enddo
```

Figure 15: Example with a sub-reference of type `complex`

LHS	*RHS*	*Communication Term*
complex	constant	OneToManyMulticast$(1, N_I)$
constant	complex	Gather$(\lceil n/N_I \rceil, N_I)$
complex	single-index multiple-index complex	ManyToManyMulticast$(\lceil n/N_I \rceil, N_I)$
single-index multiple-index	complex	ManyToManyMulticast$(\lceil n/N_I \rceil, N_I)$

Table 4: Collective communication for sub-references involving unknowns

reduces to one of the type `single-index`. If there are two or more such loops, the compiler generating communication can again use the idea discussed above of tiling the loop surrounding the innermost of those loops to reduce communication costs. PARADIGM estimates the communication cost based on the use of tiling. Further details are given in [9].

4.1.5 Sub-references of the type `complex`

If either of the pair of sub-references varying in a Type-1 loop is of the type `complex`, the compiler is unable to infer the precise pattern of data movement in that loop. The compiler can either (i) use collective communication, which would lead to over-communication, or (ii) place communication inside the loop, and use Transfer for interprocessor data movement in each iteration, or (iii) use run-time compilation techniques, such as *inspector-executor* [24, 16], to generate communication. For the purpose of estimating communication costs, PARADIGM assumes that the compiler would use collective communication. Table 4 shows the cost term obtained for the loop when at least one of the sub-references is of the type `complex`. In the entries for cost terms, n denotes the size of the corresponding array dimension, and N_I denotes the number of processors in the grid dimension. In this case, there is no special analysis needed for multiple pairs of sub-references varying in a loop. Those pairs are analyzed independently, and the resulting terms are composed in the normal manner.

Example In the program segment shown in Figure 15, if A and B are distributed on N processors, the communication cost for B is estimated as ManyToManyMulticast$(n/N, N)$. This corresponds to placing communication outside the loop without any run-time processing and using collective communication. If the compiler chooses not to move communication outside the loop, the cost would be different, as discussed in the next section. In the above example, the compiler adds another cost term for broadcasting D to all processors, if D is not already replicated on all processors.

4.1.6 Extensions: Replication and Mapping to a Constant Processor Position

We now briefly describe how our analysis extends to the case when at least one of the distributed array dimensions is not partitioned in a block or general cyclic manner. Let sr_1 and sr_2 be the `lhs` and the `rhs` sub-references respectively, with the `variation-level` corresponding to Type-1 loop L.

- When the `rhs` dimension is replicated, all data movement is internal to that dimension. The cost term contributed is IDM($\texttt{MNSP}(sr_2, L)$).
- When the `lhs` dimension is replicated, the analysis proceeds as shown above in Table 4 for the case when `lhs` sub-reference is of the type `complex`.
- When the `lhs` or the `rhs` dimension is mapped to a constant processor position, the selection of communication primitives is done exactly as for the case when the corresponding subscripts are of the type `constant`. If the `rhs` dimension is mapped to a constant processor position, the data size of the cost term is given by $\texttt{MNSP}(sr_2, L)$.

In order to simplify the description of our analysis, we will not discuss these distributions any further.

4.2 Cost Terms for Single Data Movement

A pair of sub-references of the type `constant` or varying in a Type-2 loop, represents either an internalized data movement, or communication corresponding to a single Transfer. Hence, the analysis for obtaining a communication cost term consists of determining (i) if any interprocessor communication takes place, and (ii) if so, how often the communication takes place. If the two sub-references are of the type `constant` or `single-index`, the compiler checks for the strictly-synchronous property between them. The presence of strictly synchronous property implies that all data movement is internal, and no communication cost term is needed. If any sub-reference is of the type `multiple-index` or `complex`, the compiler infers the absence of communication only if both the subscripts are identical, and both the array dimensions have identical distributions. If communication is needed, the Transfer primitive will actually be invoked only for those instances (corresponding to the iterations of Type-2 loops) when the data movement across the grid dimension is not internal to a processor. PARADIGM considers two cases:

Case 1: Boundary Communication Both sub-references, sr_1 and sr_2 are of the type single-index, and satisfy the boundary-communication test, discussed in Section 2. The number of iterations of the corresponding Type-2 loop, L, during which an interprocessor Transfer takes place is given by $\mathtt{BD}(sr_1, sr_2, L) * (\mathtt{NPP}(sr_2, L) - 1)$. The term $\mathtt{NPP}(sr_2, L) - 1$ denotes the number of processor boundaries across which communication takes place, while $\mathtt{BD}(sr_1, sr_2, L)$ denotes the number of data items that are transferred (in different iterations) across each processor boundary. The expressions for these terms have been shown earlier in Equations 1 and 4 respectively. Since communication is placed *inside* the loop L, and the obtained cost term would be multiplied by the number of iterations of that loop, say, $n_{l'}$, the normalized cost term is given by $(\mathtt{BD}(sr_1, sr_2, L) * (\mathtt{NPP}(sr_2, L) - 1) * \text{Transfer}(1))/n_{l'}$.

Case 2: General Communication This is the default case in which cost estimate is based on the following simplifying assumption. Any two arbitrary elements along the given array dimensions distributed over N_k processors are assumed to belong to the same processor with a probability of $1/N_k$. Hence, the probability that a Transfer is needed during an arbitrary loop iteration is $1 - 1/N_k$, and the cost term is obtained as $(1 - 1/N_k) * \text{Transfer}(1)$. This elegantly takes care of the important boundary case $N_k = 1$, since the communication cost term does evaluate to zero when the array dimensions are collapsed.

Example In Figure 15, if the compiler were to place communication for B inside the loop (to avoid over-communication or because of lack of space for buffer holding non-local data), the cost for a *single* loop iteration would be estimated as $(1 - 1/N) * \text{Transfer}(1)$.

4.3 Composition of Communication Terms

Once the communication terms corresponding to all the grid dimensions have been obtained, they are composed together to obtain the overall estimate of communication cost. This process consists of the following steps: (i) combining terms with the same communication primitive, (ii) determining the order in which different primitives are invoked, and obtaining the data sizes of the terms, and (iii) multiplying the terms by the expected number of repetitions of that communication during the program.

4.3.1 Communication Terms with Same Primitive

The terms involving the same primitive in different grid dimensions are combined as shown in Table 5. The combined term represents the primitive carried out over a bigger group, spanning multiple grid dimensions. As discussed earlier, a Shift term always appears in conjunction with an IDM term. Figure 16 illustrates the composition of two `shifts` in different dimensions together with the associated terms for internalized data movement. The composition leads to `shifts` in the original directions with bigger data sizes, a "diagonal"

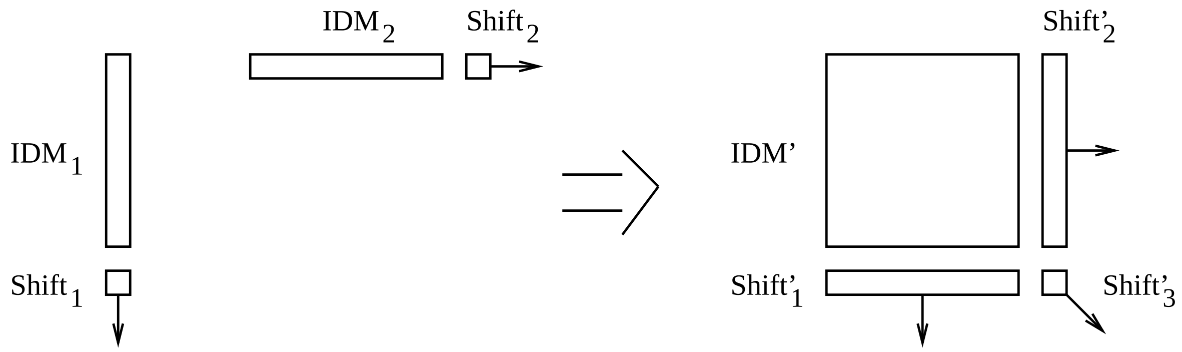

Figure 16: Composition of Shift and IDM pairs

Term1	*Term2*	*Resultant Term*
IDM(m_1)	IDM(m_2)	IDM$(m_1 * m_2)$
Gather(m_1, p_1)	Gather(m_2, p_2)	Gather$(m_1 * m_2, p_1 * p_2)$
IDM(m_1), $(N_1 > 1)$ Shift(c_1)	IDM(m_2), $(N_2 > 1)$ Shift(c_2)	IDM$(m_1 * m_2)$, $(N_1 > 1)$ Shift$(c_1 * m_2)$, $(N_2 > 1)$ Shift$(c_2 * m_1)$, $(N_1 > 1$ & $N_2 > 1)$ Shift$(c_1 * c_2)$
Reduction(m_1, p_1)	Reduction(m_2, p_2)	Reduction$(m_1 * m_2, p_1 * p_2)$
ManyToManyM'cast(m_1, p_1)	ManyToManyM'cast(m_2, p_2)	ManyToManyM'cast$(m_1 * m_2, p_1 * p_2)$
OneToManyM'cast(m_1, p_1)	OneToManyM'cast(m_2, p_2)	OneToManyM'cast$(m_1 * m_2, p_1 * p_2)$
$(1 - 1/p_1) *$ Transfer(m_1)	$(1 - 1/p_2) *$ Transfer(m_2)	$(1 - 1/(p_1 * p_2)) *$ Transfer$(m_1 * m_2)$

Table 5: Combining of terms with identical communication primitives

`shift`, and a purely internalized data movement.

4.3.2 Ordering of Primitives

Since the primitives corresponding to different terms implement the data movement in distinct grid dimensions, they can legally be composed in any order. However, the order in which they are invoked is important because the position of each primitive affects the message sizes and the number of processors involved (in parallel) in subsequent primitives. It is desirable to obtain an ordering that leads to fewer processors being involved and smaller messages handled by each processor, but sometimes, there is a trade-off between the two. For example, consider the statement shown in Figure 17, where the arrays A and B are distributed in an identical manner on a 2-D grid. The primitives required are: Gather in the first dimension, and Transfer in the second dimension. Figure 18 illustrates the two possible orderings for a 3 x 3 grid. If Gather is invoked

do $i = 1, n$
$\quad A(n, n) = \mathcal{F}(B(i, 1))$
enddo

Figure 17: Statement requiring Gather and Transfer in different dimensions

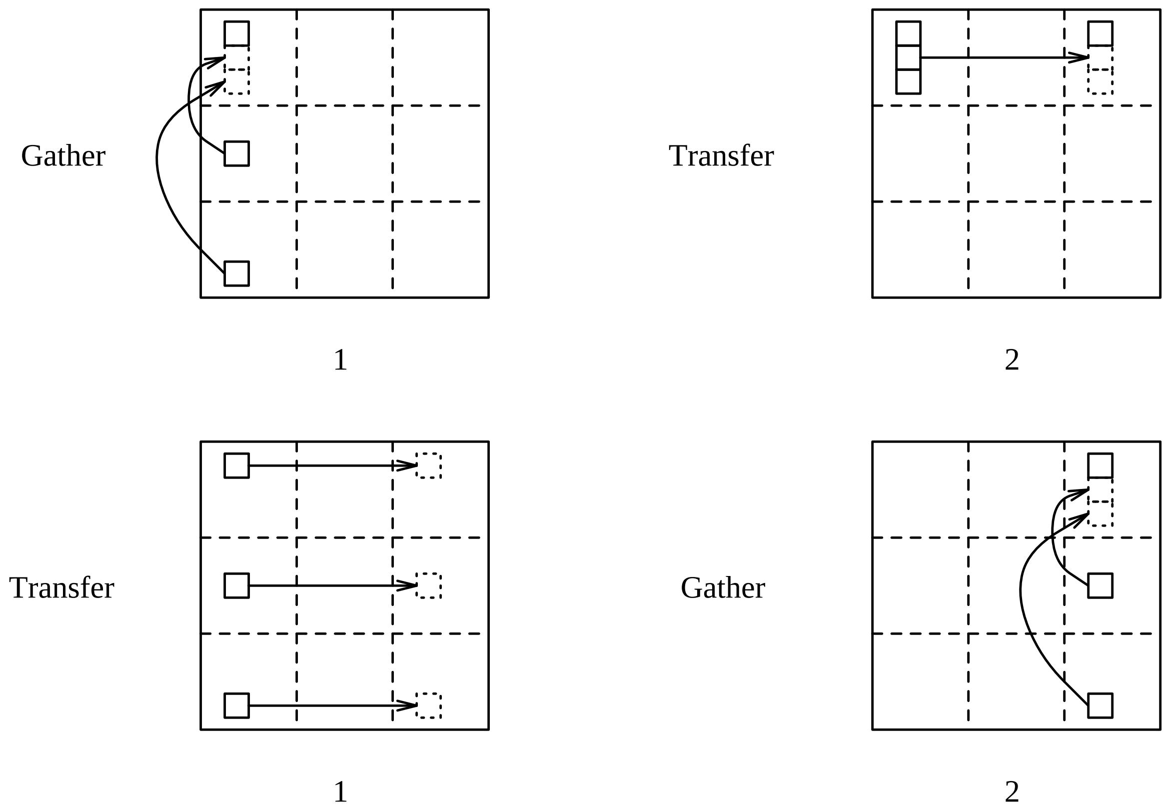

Figure 18: Possible compositions of Gather and Transfer

first, it is carried out with a data size of $n/3$ words, over 3 processors, followed by a single Transfer of n words of data. If this ordering is reversed, there are 3 parallel Transfers that take place, each involving $n/3$ words, followed by a Gather operation, also with a data size of $n/3$ words, over 3 processors. The second ordering in the above example leads to the use of parallelism in implementing communication, and would yield better performance if there were no other communications being carried out on the grid of processors. This suggests resolving the trade-off in favor of reducing the message sizes handled by processors. When there is no trade-off involved, the compiler should use an ordering that reduces the message sizes and/or the number of processors involved. These considerations suggest the ordering shown in Table 6.

Rank	*Primitive*	*Message Size*	*No. of Processors*
1	Reduction	reduced	reduced
2	Scatter	reduced	increased
3	Shift, Transfer	preserved	preserved
4	OneToManyMulticast	preserved	increased
5	Gather	increased	reduced
6	ManyToManyMulticast	increased	increased

Table 6: Ordering of communication primitives for composition

```
do j = 1, n
    do i = 1, n
        A(i, j, c1) = F(B(i, c2, c3))
    enddo
enddo
```

Figure 19: Example of composition of communication terms

Determination of message sizes

The data size associated with each communication term initially represents just one "edge" of the overall volume of data being communicated. In accordance with the ordering of primitives shown above, the compiler determines the actual message size for each primitive corresponding to a communication term. First, the compiler determines the product of individual data sizes associated with each communication term, including an IDM term. This product represents the complete volume of data to be communicated, and serves as the message size for each term corresponding to any of the first five primitives listed in Table 6. (Note that the data sizes used in terms for the Reduction and Scatter primitives already reflect the "reduction" in message size caused by those primitives, as can be seen from the entries shown in Tables 2 and 3). If there are terms present for both Gather and ManyToManyMulticast primitives, the data size for the ManyToManyMulticast is multiplied further by the number of processors shown in the Gather term, to account for the increased amount of data participating in that primitive following the Gather operation.

Example Consider the statement shown in Figure 19, where the arrays, A and B are distributed in an identical manner on a N_1 x N_2 x N_3 grid. The communication terms obtained are: IDM($\lceil n/N_1 \rceil$), OneToManyMulticast($1, N_2$), and $(1 - 1/N_3) *$ Transfer(1). These terms are composed together to give the communication cost estimate as $(1 - 1/N_3) *$ Transfer($\lceil n/N_1 \rceil$) + OneToManyMulticast($\lceil n/N_1 \rceil, N_2$).

4.4 Number of Repetitions

The sequence of communication primitives, placed outside loop L_{l+1} as shown in Figure 5, is repeated during iterations of the surrounding Type-2 loops, L_1 through L_l. Due to the presence of conditionals in the program, the flow of control may not reach that point during every iteration. The number of times the flow of control reaches the statement corresponding to loop L_{k+1} can be expressed as:

$$num(L_{k+1}) = \rho(L_{k+1}) * (\prod_{k=1}^{l} n_k),$$

where $\rho(L_{k+1})$ is the *reaching probability* [9] of that statement, and n_k is the iteration count of the loop L_k. The compiler multiplies the cost of the sequence of communication primitives obtained above by $num(L_{k+1})$, to reflect the overall cost of that communication for the entire program. The value of $num(L_{k+1})$ may not be known at compile time due to the presence of conditionals and symbolic loop bounds. A good method for obtaining estimates of the values of $num(L_{k+1})$ is to use profiling [25, 6]. As mentioned earlier, PARADIGM relies on input from the user to obtain the values of any unknown loop bounds and reaching probabilities of statements. Unless explicitly supplied a value, PARADIGM assumes a reaching probability of one for each statement.

The above analysis assumes strict sequentiality in the process of carrying out communication inside the Type-2 loops. In practice, the generation of messages may sometimes get pipelined over different iterations of the outer loops. In some special cases, the compiler can detect this pipelining and refine the cost estimate [9]. Those details are beyond the scope of this paper.

5 Results

We now present results on automatic data partitioning that show the importance of using good performance estimates to guide those decisions. These results are obtained by PARADIGM using the analysis described in this paper for estimating communication costs. While the analysis we have presented is largely machine-independent, the actual values of machine-specific parameters have been supplied for an Intel iPSC/2 hypercube with 16 processors. We will show performance results on that machine to evaluate the quality of data partitioning decisions made by the compiler.

A discussion of the techniques used by PARADIGM to avoid searching exhaustively through the space of data partitioning schemes while using these estimates is beyond the scope of this paper. Those details and more extensive results on automatic data partitioning are presented in [9, 11]. In this section, we shall discuss results from two representative programs that show a range of data partitioning decisions which require more effective techniques for estimating communication costs than those currently used by most other compilers. The first program, `Tred2` is a routine from the EISPACK library, it reduces a real symmetric matrix to a symmetric tridiagonal matrix. `Jacobi` is a simplified version of a relaxation code that performs Jacobi iterations in a loop.

5.1 Application 1: TRED2

The source code of `tred2` is listed in Figure 20, where statements are labeled by the line numbers. We have used a data size of $N = 512$ for the program supplied to the compiler.

Alignment One of the most interesting decisions for the program relates to the alignment of array dimensions, as different parts of the program give rise to conflicting requirements on alignment. For example, the references in statements 58, 71 and 77 favor alignment of D_1 with Z_1, while those in statements 18, 59 and

```
DO 5 I = 1, N
     DO 3 J = I, N
       Z(J,I) = A(J,I)
     D(I) = A(N,I)
CONTINUE
IF (N .EQ. 1) GO TO 82
DO 63 II = 2, N
     I = N + 2 - II
     L = I - 1
      H = 0.0D0
      SCALE = 0.0D0
      IF (L .LT. 2) GO TO 16
      DO 14 K = 1, L
        SCALE = SCALE + DABS(D(K))
      IF (SCALE .NE. 0.0D0) GO TO 23
      E(I) = D(L)
      DO 21 J = 1, L
            D(J) = Z(L,J)
            Z(I,J) = 0.0D0
            Z(J,I) = 0.0D0
      CONTINUE
      GO TO 62
      DO 25 K = 1, L
            D(K) = D(K) / SCALE
            H = H + D(K) * D(K)
      CONTINUE
      F = D(L)
      G = -DSIGN(DSQRT(H),F)
      E(I) = SCALE * G
      H = H - F * G
      D(L) = F - G
      DO 33 J = 1, L
        E(J) = 0.0D0
      DO 45 J = 1, L
            F = D(J)
            Z(J,I) = F
            G = E(J) + Z(J,J) * F
            JP1 = J + 1
            IF (L .LT. JP1) GO TO 43
            DO 43 K = JP1, L
                  G = G + Z(K,J) * D(K)
                  E(K) = E(K) + Z(K,J) * F
            CONTINUE
            E(J) = G
      CONTINUE
      F = 0.0D0
      DO 50 J = 1, L
            E(J) = E(J) / H
            F = F + E(J) * D(J)
      CONTINUE
      HH = F / (H + H)
      DO 53 J = 1, L
        E(J) = E(J) - HH * D(J)
      DO 61 J = 1, L
            F = D(J)
            G = E(J)
            DO 58 K = J, L
                  Z(K,J) = Z(K,J) - F * E(K) - G * D(K)
            D(J) = Z(L,J)
            Z(I,J) = 0.0D0
      CONTINUE
      D(I) = H
CONTINUE
DO 81 I = 2, N
      L = I - 1
      Z(N,L) = Z(L,L)
      Z(L,L) = 1.0D0
      H = D(I)
      IF (H .EQ. 0.0D0) GO TO 78
      DO 71 K = 1, L
        D(K) = Z(K,I) / H
      DO 78 J = 1, L
            G = 0.0D0
            DO 75 K = 1, L
              G = G + Z(K,I) * Z(K,J)
            DO 78 K = 1, L
              Z(K,J) = Z(K,J) - G * D(K)
      CONTINUE
      DO 80 K = 1, L
        Z(K,I) = 0.0D0
CONTINUE
DO 85 I = 1, N
      D(I) = Z(N,I)
      Z(N,I) = 0.0D0
CONTINUE
Z(N,N) = 1.0D0
E(1) = 0.0D0
END
```

Figure 20: Source code of `tred2` routine

83 favor alignment of D_1 with Z_2. Statement 4 favors alignment of D_1 with A_2, for which there is a further preference for alignment with Z_2. PARADIGM represents these alignment constraints using a *component affinity graph* [18], in which the nodes represent array dimensions, and weighted edges represent alignment preferences. If the edge weights were obtained simply as a count of the number of statements favoring that alignment, an optimal decision would be to mutually align A_2, Z_2, D_1 and Z_1, and to align A_1 with Z_1. However, using the techniques described in this paper, PARADIGM obtains edge weights as estimates of the extra communication cost due to mis-alignment of array dimensions. That leads to a decision to align A_1, Z_1, D_1 and E_1 together, and to align A_2 with Z_2 [9].

Block/Cyclic Distribution There are a number of statements in the program that favor cyclic distribution of all array dimensions, and a sole statement favoring block distribution. The reference on statement 16 requires nearest-neighbor communication, and hence there is a preference recorded for block distribution of D_1 and E_1, to cut down on the volume of communication [11]. However, our estimation process shows that the extra communication cost due to cyclic distribution is much smaller than the expected performance gains due to better load balance [9]. Therefore, the compiler chooses cyclic distribution for all array dimensions.

Number of Processors At a number of statements, the given program exhibits parallelism at two loop levels that traverse the rows and the columns of arrays A and Z. The program also shows communication across both the dimensions of arrays, if distributed. Hence, it is difficult to determine the relative number of processors on which to distribute the array dimensions, without using communication and computational cost estimates. Based on these estimates, PARADIGM compares different N_1 x N_2 grid configurations, varying N_1 from 1 to 16, doubling its value in each step. It selects 16 x 1 as the best grid configuration. These decisions lead to the distribution of arrays A and Z by *rows* in a *cyclic* manner, and of arrays D and E also in a cyclic manner, on all the 16 processors.

Evaluation In order to evaluate each data partitioning decision described above, we implemented different parallel versions of `tred2` on the iPSC/2. Each version was obtained by hand-simulating the compilation process under a different data partitioning scheme for that program. The execution times observed on the iPSC/2 are shown in Figure 21. The sequential time for the program is not shown because the program could not be run on a single node due to memory limitations. The data distribution scheme selected by PARADIGM, marked *row cyclic*, performs the best. A different alignment, i.e., of D_1 and E_1 with A_2 and Z_2, leads to the distribution of columns of A and Z (and arrays D and E) in a cyclic manner. This scheme, marked *column cyclic*, does not perform as well, confirming that the alignment decision made by PARADIGM, based on communication cost estimates, is a better one. The other two schemes, referred to as *row block* and *2-d cyclic* correspond to departures from the selected data partitioning regarding decisions of cyclic distribution, and of collapsing one of the dimensions. The poorer performance resulting from these distributions confirms

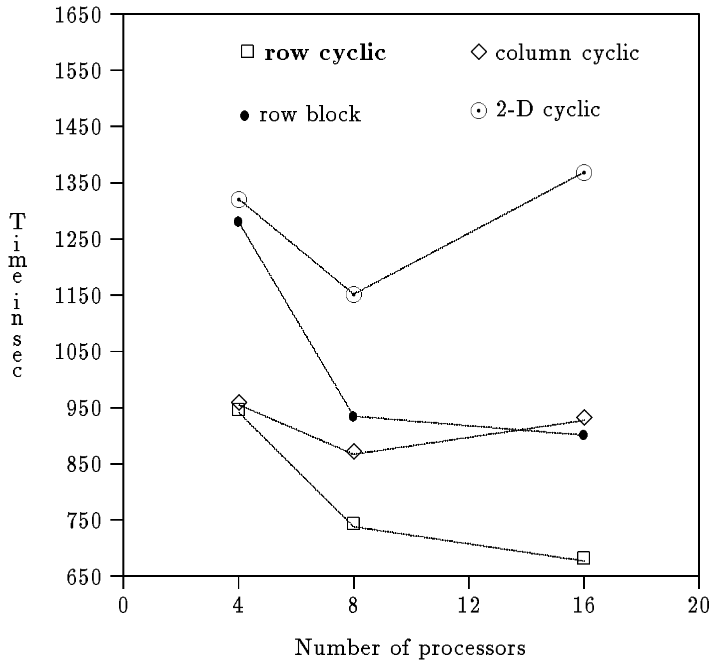

Figure 21: Performance of `tred2` on Intel iPSC/2 for data size $n = 512$

these other decisions as well.

5.2 Application 2: JACOBI

The source code of the `Jacobi` program is shown in Figure 22. This simplified version repeatedly carries out relaxation computations followed by copying of array elements to store the elements computed in the previous step.

The decisions to align A_1 with B_1, and A_2 with B_2, and to distribute each array dimension in a blocked manner are straightforward, with no conflicting requirements [9]. However, the compiler needs performance estimates to determine the relative number of processors on which to distribute the two array dimensions. A trivial solution to this problem cannot be obtained because there is parallelism to be exploited and communication in both the grid dimensions. The contribution of communication cost to the program execution time is estimated by the compiler as:

$$\text{Communication cost} \quad = \quad ncycles * (2 * (N_1 > 1) * \text{Shift}(n/N_2) + 2 * (N_2 > 1) * \text{Shift}(n/N_1))$$

Let us consider two grid configurations, 1 x 16, and 4 x 4. The first of these corresponds to a column partitioning of the arrays. It leads to two Shift operations being carried out in every cycle, with a data size of n words each. The second configurations corresponds to a "2-D" partitioning of the arrays, where both the rows and the columns are distributed on 4 processors each. That leads to four Shift operations in every cycle, each with a data size of $n/4$ words. For smaller values of n, the first scheme is better, since

```
parameter (np2 = 514, ncycles = 100)
double precision A(np2,np2), B(np2,np2)

np1 = np2 - 1
do k = 1, ncycles
    do i = 2, np1
        do j = 2, np1
            B(i,j) = 0.5 * A(i,j) + 0.125 * (A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1))
        enddo
    enddo
    do i = 2, np1
        do j = 2, np1
            A(i,j) = B(i,j)
        enddo
    enddo
enddo
```

Figure 22: Source code for `Jacobi` program

Data Size n	*Column Blocked Time (s)*	*2-D Blocked Time (s)*	*Column Cyclic Time (s)*
64	1.28	1.45	1.66
128	4.10	4.12	5.82
256	15.39	14.87	23.29
512	64.43	59.66	96.64
1024	257.84	243.24	386.94

Table 7: Performance of different versions of Jacobi on iPSC/2

the start-up costs of sending messages dominate the communication cost. As the data size is increased, the second scheme starts becoming more attractive, since the total amount of data transferred is lower under that scheme. Given different values of n that are doubled in each step, PARADIGM starts choosing the second scheme at $n = 1024$.

The execution times obtained for each of those versions running on the 16-processor iPSC/2 are shown in Table 7. These results confirm that the 2-D partitioning starts performing better than the column partitioning for larger array sizes. Similar results have been reported earlier by other researchers [3]. We have also presented results for cyclic distribution of columns to show the performance penalty if the compiler did not choose block distribution.

While these results confirm the compiler's prediction regarding the suitability of the 2-D partitioning at larger data sizes, the actual data size at which that scheme starts performing better than column partitioning is not predicted very accurately. We believe that this difference between the predicted value ($n = 1024$) and the observed value ($n = 256$) is primarily due to the cost function used for the Shift operation being inaccurate. The primary focus of our work in estimating communication costs has been to obtain the estimates in terms of cost functions of various communication primitives (which is performed satisfactorily in this case). Given more accurate performance characteristics of such primitives, obtained by approaches proposed in the literature, such as the "training set" method [3], we believe that PARADIGM could do an even better job of selecting good data partitioning schemes.

6 Conclusions

We have presented a methodology for estimating communication costs in a program, to guide data partitioning decisions. A significant feature of our approach is that the analysis is performed on the original source program in global address space. This allows various alternatives regarding data partitioning to be evaluated without actually having to generate code for each version. At the same time, the compiler accounts for important communication optimizations to maintain accuracy of the analysis.

Another contribution of this work has been to develop a systematic analysis for characterizing data movement. The notion of synchronous properties and speed-ratio between sub-references varying in a loop helps identify the nature of data movement, and hence helps select a collective communication primitive. The other traversal properties of each individual sub-reference help estimate the size of messages and the number of processors involved in communication. We have also described various issues and trade-offs in composing communication across multiple processor grid dimensions, when arrays have more than one distributed dimension. On the one hand, this analysis offers the advantage of more precise information about the global state of communication, which is useful for estimating performance, and on the other hand, it enables enhanced performance through the use of collective communication primitives [10] for a compiler generating communication [12].

The methodology presented in this paper has been implemented as part of the PARADIGM compiler. We have shown that detailed communication cost estimates, as obtained by our methodology, are essential for good data partitioning decisions in a number of cases. Our performance results so far indicate that these estimates do guide data partitioning decisions in the right direction.

There are many limitations of our present work, and a number of ways in which it can be extended. Our approach in dealing with program parameters that are unknown at compile time is simplistic, and can benefit by incorporating ideas from the work of Fahringer and Zima [5, 6]. Our approach can also be extended to consider communication costs of multiple references at the same time, and to incorporate additional compiler optimizations like coalescing messages [13], and recognizing pipeline communication, as done by PTRAN-II [12].

Acknowledgements

We are extremely grateful to the anonymous referees for their valuable comments.

References

[1] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[2] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proc. ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, June 1993.

[3] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proc. Third ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Williamsburg, VA, April 1991.

[4] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Automatic array alignment in data-parallel programs. In *Proc. Twentieth Annual ACM Symposium on Principles of Programming Languages*, Charleston, SC, January 1993.

[5] T. Fahringer. *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers.* PhD thesis, University of Vienna, September 1993.

[6] T. Fahringer and H. P. Zima. A static parameter based performance prediction tool for parallel programs. In *Proc. 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[7] P. Feautrier. Towards automatic data distribution. In *Proc. 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[8] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University, May 1993.

[9] M. Gupta. *Automatic data partitioning on distributed memory multicomputers.* PhD thesis, University of Illinois, September 1992.

[10] M. Gupta and P. Banerjee. A methodology for high-level synthesis of communication on multicomputers. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[11] M. Gupta and P. Banerjee. PARADIGM: A Compiler for Automatic Data Distribution on Multicomputers. In *Proc. 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[12] M. Gupta, S. Midkiff, E. Schonberg, P. Sweeney, K.Y. Wang, and M. Burke. PTRAN II: A compiler for High Performance Fortran. In *Proc. 4th Workshop on Compilers for Parallel Computers*, Delft, Netherlands, December 1993.

[13] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[14] K. Kennedy and U. Kremer. Automatic data alignment and distribution for loosely synchronous problems in an interactive programming environment. Technical Report TR91-155, Rice University, April 1991.

[15] K. Knobe, J. Lukas, and G. Steele Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[16] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

[17] J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Proc. Supercomputing '90*, New York, NY, November 1990.

[18] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[19] Message Passing Interface Forum. *Document for a standard message-passing interface*, November 1993. (Draft), available from University of Tennessee, Knoxville, Tennessee.

[20] M. O'Boyle and G. A. Hedayat. A transformational approach to compiling Sisal for distributed memory architectures. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[21] M.J. Quinn and P. J. Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, 7:69–76, September 1990.

[22] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–481, October 1991.

[23] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proc. SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 69–80, June 1989.

[24] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[25] V. Sarkar. Determining average program execution times and their variance. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1989.

[26] E. Su, D. J. Palermo, and P. Banerjee. Automating parallelization of regular computations for distributed memory multicomputers in the PARADIGM compiler. In *Proc. 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[27] K.-Y. Wang. *Intelligent program optimization and paralleilzation for parallel computers.* PhD thesis, Purdue University, April 1991.

[28] S. Wholey. Automatic data mapping for distributed-memory parallel computers. In *Proc. 6th ACM International Conference on Supercomputing*, Washington D.C., July 1992.

[29] M. J. Wolfe. More iteration space tiling. In *Proc. Supercomputing 89*, Reno, Nevada, November 1989.

[30] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, 1989.

[31] H. Zima, H. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

A Proof of Correctness of Theorem 1

In order to prove our results regarding the synchronous properties, we need the following two lemmas.

Lemma 1 *Given any integer x, and positive integers y and z,*

$$\lfloor x/(y*z)\rfloor = \lfloor \lfloor x/y\rfloor /z\rfloor$$

Proof

$$\begin{array}{lllll}
\text{Let} & x & = & q*y+r, & \text{where } 0\le r<y,\ q \text{ and } r \text{ are integers.}\\
\text{Let} & q & = & q'*z+r', & \text{where } 0\le r'<z,\ q', \text{ and } r' \text{ are integers.}\\
\text{We have} & \lfloor x/(y*z)\rfloor & = & \lfloor (q*y+r)/(y*z)\rfloor & \\
 & & = & \lfloor q/z+r/(y*z)\rfloor & \\
 & & = & \lfloor (q'*z+r')/z+r/(y*z)\rfloor & \\
 & & = & \lfloor q'+r'/z+r/(y*z)\rfloor & \\
 & & = & q'+\lfloor (r'*y+r)/(y*z)\rfloor & \\
\text{Now} & r'*y+r & < & (r'+1)*y & (\text{since } r<y)\\
 & r'+1 & \le & z & (\text{since } r'<z)
\end{array}$$

$$\begin{array}{llcl}
\text{Hence,} & r' * y + r & < & z * y \\
& \lfloor (r' * y + r)/(y * z) \rfloor & = & 0 \\
\text{Therefore,} & \lfloor x/(y * z) \rfloor & = & q' \\
& & = & \lfloor q/z \rfloor \\
& & = & \lfloor \lfloor x/y \rfloor /z \rfloor
\end{array}$$

□

Lemma 2 *Given any integer x, and positive integers y and z,*

$$\lfloor x_1/y_1 \rfloor = \lfloor x_2/y_2 \rfloor \quad \Rightarrow \quad \lfloor x_1/(z * y_1) \rfloor = \lfloor x_2/(z * y_2) \rfloor$$

Proof

$$\begin{array}{llcll}
& \lfloor x_1/y_1 \rfloor & = & \lfloor x_2/y_2 \rfloor & \\
\Rightarrow & \lfloor \lfloor x_1/y_1 \rfloor /z \rfloor & = & \lfloor \lfloor x_2/y_2 \rfloor /z \rfloor & \\
\Rightarrow & \lfloor x_1/(z * y_1) \rfloor & = & \lfloor x_2/(z * y_2) \rfloor & \text{(by Lemma 1)}
\end{array}$$

□

Proof of Theorem 1 During loop iteration i, the two sub-references are mapped to positions $\lfloor (s_1(\alpha_1 i + \beta_1) - o_1)/b_1 \rfloor [\bmod N_k]$ and $\lfloor (s_2(\alpha_2 i + \beta_2) - o_2)/b_2 \rfloor [\bmod N_k]$ respectively. To prove the strictly synchronous property, it is sufficient to show that

$$\lfloor (s_1\alpha_1 i + s_1\beta_1 - o_1)/b_1 \rfloor = \lfloor (s_2\alpha_2 i + s_2\beta_2 - o_2)/b_2 \rfloor \qquad \forall i \in I$$

The first set of conditions implies that

$$\begin{array}{llcll}
& (s_1\alpha_1 i)/b_1 + (s_1\beta_1 - o_1)/b_1 & = & (s_2\alpha_2 i)/b_2 + (s_2\beta_2 - o_2)/b_2, & \forall i \in I \\
\Rightarrow & \lfloor (s_1\alpha_1 i + s_1\beta_1 - o_1)/b_1 \rfloor & = & \lfloor (s_2\alpha_2 i + s_2\beta_2 - o_2)/b_2 \rfloor & \text{(Q.E.D. for case 1)}
\end{array}$$

The condition (iii) from the second set implies that

$$\begin{array}{llcll}
& \lfloor (s_1\beta_1 - o_1)/s_1\alpha_1 \rfloor + i & = & \lfloor (s_2\beta_2 - o_2)/s_2\alpha_2 \rfloor + i, & \forall i \in I \\
\Rightarrow & \lfloor (s_1\beta_1 - o_1)/s_1\alpha_1 + i \rfloor & = & \lfloor (s_2\beta_2 - o_2)/s_2\alpha_2 + i \rfloor, & \\
\Rightarrow & \lfloor (s_1\alpha_1 i + s_1\beta_1 - o_1)/s_1\alpha_1 \rfloor & = & \lfloor (s_2\alpha_2 i + s_2\beta_2 - o_2)/s_2\alpha_2 \rfloor & \\
\Rightarrow & \lfloor (s_1\alpha_1 i + s_1\beta_1 - o_1)/(m * s_1\alpha_1) \rfloor & = & \lfloor (s_2\alpha_2 i + s_2\beta_2 - o_2)/(m * s_2\alpha_2) \rfloor & \text{(by Lemma 2)} \\
\Rightarrow & \lfloor (s_1\alpha_1 i + s_1\beta_1 - o_1)/b_1 \rfloor & = & \lfloor (s_2\alpha_2 i + s_2\beta_2 - o_2)/b_2 \rfloor & \text{(from conditions (i) and (ii))}
\end{array}$$

□