Efficient Compilation of Array Statements for Private Memory Multicomputers

James M. Stichnoth

February, 1993
CMU-CS-93-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

One of the core constructs of High Performance Fortran (HPF) is the array-slice assignment statement, combined with the rich choice of data distribution options available to the programmer. On a private memory multicomputer, the HPF compiler writer faces the difficult task of automatically generating the necessary communication for assignment statements involving arrays with arbitrary block-cyclic data distributions. In this paper we present a framework for representing array slices and block-cyclic distributions, and we derive efficient algorithms for sending and receiving the necessary data for array-slice assignment statements. The algorithms include a memory-efficient method of managing the layout of the distributed arrays in each processor's local memory. We also provide a means of converting the user's `TEMPLATE`, `ALIGN`, and `DISTRIBUTE` statements into a convenient *array ownership descriptor*. In addition, we present several optimizations for common distributions and easily-recognized communication patterns. The work presented makes minimal assumptions regarding the processor architecture, the communication architecture, or the underlying language being compiled.

Supported in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330. Work furnished in connection with this research is provided under prime contract MDA972–90–C–0035 issued by DARPA/CMO to Carnegie Mellon University. This material is also based upon work supported in part by a National Science Foundation Graduate Research Fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either express or implied, of DARPA, NSF, or the U.S. government.

Keywords: parallelizing compilers, High Performance Fortran, array statements

Contents

1 Introduction **1**

2 Overview of the approach **2**

3 Basic requirements **5**

4 Simplifying restrictions **6**

5 Slice intersection **6**

6 Sends **9**
6.1 Derivation . . . 9
6.2 Algorithm . . . 13
6.3 Local memory . . . 13

7 Receives **15**
7.1 Algorithm . . . 15
7.2 Local memory . . . 16

8 Computation **16**

9 Overall algorithm **17**

10 Relaxing restrictions **18**
10.1 Multiple right-hand side terms . . . 18
10.2 Multidimensional arrays . . . 20
10.3 Negative strides . . . 21
10.4 Partially replicated arrays . . . 22
10.5 Nested array references . . . 22

11 Generating array ownership descriptors **22**

12 Optimizations **27**
12.1 Reducing potential communication . . . 27
12.1.1 Communication with a small number of processors . . . 27
12.1.2 Small number of senders or receivers . . . 29
12.2 Eliminating loops . . . 30
12.3 Customizing the output . . . 30
12.4 Merging receive and computation phases . . . 31

13 Further applications of array statements **31**
13.1 Redistributing arrays . . . 31
13.2 `WHERE` statement . . . 32
13.3 Reduction operators . . . 33
13.4 Parallel independent loop iterations . . . 33

14 Conclusion **34**

A Extended algorithms for signed strides **36**
A.1 Extended LOCAL–INDEX–SEND algorithm 36
A.2 Extended LOCAL–INDEX–RECEIVE algorithm 37
A.3 Extended COMPUTE–LOOP algorithm 39

B Optimized communication for block and cyclic distributions **39**
B.1 Case A: block-cyclic, block 40
B.2 Case B: block-cyclic, cyclic 40
B.3 Case C: block, block-cyclic 41
B.4 Case D: block, block 42
B.5 Case E: block, cyclic 43
B.6 Case F: cyclic, block-cyclic 44
B.7 Case G: cyclic, block 44
B.8 Case H: cyclic, cyclic 45

C Optimized computation for block and cyclic distributions **46**
C.1 Case A: block 46
C.2 Case B: cyclic 47

D Notation **47**

1 Introduction

Private memory multiprocessors have made it possible to solve larger problems more quickly compared to single-processor computers. Problems can be solved more quickly by distributing the work among the processors. Problems with larger memory requirements can be solved due to the fact that each individual processor can typically have an amount of private memory on the order of that of a single-processor system, so the total amount of memory in the system scales up with the number of processors.

When mapping a scientific application onto such a parallel machine, a common approach is to distribute the array elements among the individual processors. When performing a piece of the computation, parallelism is obtained when related data are found on the same processor, and the absence of data dependences allows most or all of the processors to compute in parallel. Whenever nonlocal data is needed for a computation, communication is performed. In this approach, all processors follow essentially the same thread of control, so it usually suffices to write a single source program which is executed on all processors. With a careful data distribution, the computation can often be distributed evenly among the processors while maximizing the data locality.

The problem with this approach is that it is usually time-consuming and error-prone for the programmer, who must manage the memory layout and the communication within the program. The difficulty of implementing this management correctly can discourage the programmer from experimenting with different distributions of data, or from even attempting to parallelize the program in the first place.

An obvious solution to this problem is to have a compiler which automatically manages the memory layout and the communication. This approach frees the programmer from these concerns, and allows the programmer to concentrate on the more important aspects of the algorithm and to experiment with different methods of data distribution.

There have been several compiler projects with this goal. The AL compiler [17] for the Warp systolic array allows the programmer to distribute a single dimension of an array across the processors, to specify array elements that are used in the same computation, and to distribute computation across the processors. In addition, AL allows the specification of a *window relation* which allows data at block boundaries to be shared by several processors. Our framework allows window relations in the form of *left and right overlap* specified in the `ALIGN` statement (see Section 11). CM Fortran [16] allows the programmer to specify parallelism in the form of Fortran 90 [1] array statements, and it gives the programmer some freedom in choosing a data distribution. Fortran D [9] provides a rich set of data distribution methods, which are applied independently to each dimension of an array. While Fortran D uses data dependence analysis to determine which sequential code can be parallelized, Fortran 90D [6] adds the syntax of Fortran 90 array statements to help derive parallelism. Vienna Fortran [19] allows the programmer to specify the same kinds of alignments and distributions as in Fortran D, and in fact allows many more kinds, including user-specified alignment and distribution functions.

In this paper, we derive the communication and the memory management required to evaluate array statements similar to those in Fortran 90. The data distribution is given in terms similar to those in Fortran D; we assume that each dimension of each array is distributed in block-cyclic fashion. This distribution could be specified by the programmer, or the compiler could automatically derive distributions, using methods similar to those of Chatterjee et al. [5] or Wholey [18]. Parallelism is expressed in terms of array statements, as in CM Fortran and Fortran 90D. Thus our approach is similar to that taken by the High Performance Fortran Forum [8], although we do not explicitly assume Fortran as our input language. The work of automatically deriving parallelism from sequential code is beyond the scope of this paper.

Chatterjee et al. [4] present a similar framework for compiling array assignment statements, in terms of constructing a finite state machine. Chatterjee's approach accesses data in a manner that is more friendly

than our approach to a data cache, especially in the case of block-cyclic data distributions. However, our approach requires $O(P)$ less buffer space, where P is the number of processors, and our approach allows more overlapping of communication and computation. Their approach additionally allows an arbitrary integer affine alignment function, e.g. `ALIGN A`$(ai + b)$ `WITH T`(i), whereas our approach only allows a to be 0 or 1. Finally, although Chatterjee's method can handle the communication for an arbitrary array assignment statement, such as the communication that arises during a redistribution, it induces two integer divisions per element within the inner loop, although the extra cost can be made more reasonable when the divisors are known to be powers of 2.

Although other work [14, 13] addresses compile-time analysis of array statements with block and cyclic distributions, our approach and Chatterjee's approach are the only methods to date which address compile-time analysis of block-cyclic distributions.

A quite different approach for generating communication sets is taken by Saltz et al. [15]. While the methods described in this paper use compile-time analysis exclusively, their method invokes runtime analysis, in the form of the inspector/executor model, to generate the communication. Compile-time analysis is likely to produce faster code, provided that sufficient access and distribution information is known at compile time, although it does not extend well to handle the important class of unstructured and irregular data distributions.

The work presented here is meant to be sufficiently detailed to allow a compiler writer to use it as a manual for implementing a similar compiler. We have validated our work by implementing it in a prototype compiler, called Fx, for the iWarp system [2, 3]. The Fx compiler takes as input Fortran 77 statements augmented with whole-array and array-slice syntax and directives for data distribution, and produces as output Fortran 77 code augmented with communication primitives, which is passed on to the native iWarp Fortran compiler.

The paper is organized in the following manner: Section 2 gives an overview of the approach taken in the paper. Section 3 presents the permanent low-level assumptions we make, while Section 4 imposes temporary restrictions on the form of statements we compile, to simplify the analysis. These restrictions are removed in Section 10. Section 5 describes slice intersection, which is the basis of the methods presented in Sections 6 and 7, which describe how to compute the data that is sent and received, respectively, between processors. Slice intersection is also vital for Section 8, which details how a processor performs its computation after completing its communication. The communication and computation phases are put together into an overall algorithm in Section 9. Section 11 describes how processors can determine data ownership, given data distribution directives provided by the user. Section 12 presents some possible optimizations for common cases. Section 13 shows how the basic array assignment statement can be easily extended to handle other useful array constructs.

Note that Appendix D provides a partial glossary of the variable naming notation used throughout the paper.

2 Overview of the approach

In our approach, parallelism is expressed as user directives, which are given at the statement level in the form of array statements. The standard example of an array statement used in this paper is the assignment statement:

$$\mathtt{A}(f_A : l_A : s_A) = \mathcal{F}(\mathtt{B}(f_B : l_B : s_B)), \tag{1}$$

which is equivalent to the following sequential Fortran code:

```
i_B = f_B
DO i_A = f_A, l_A, s_A
   T(i_A) = B(i_B)
   i_B = i_B + s_B
END DO
DO i_A = f_A, l_A, s_A
   A(i_A) = F(T(i_A))
END DO
```

This sequential code copies the relevant section of `B` into a temporary array `T`, and then performs the assignment to `A` based on the contents of `T`. This two-step procedure is used because the semantics of the assignment statement dictate that the right-hand side terms are first read, and then the results are computed and the left-hand side terms are written. The procedure is depicted in Figure 1.

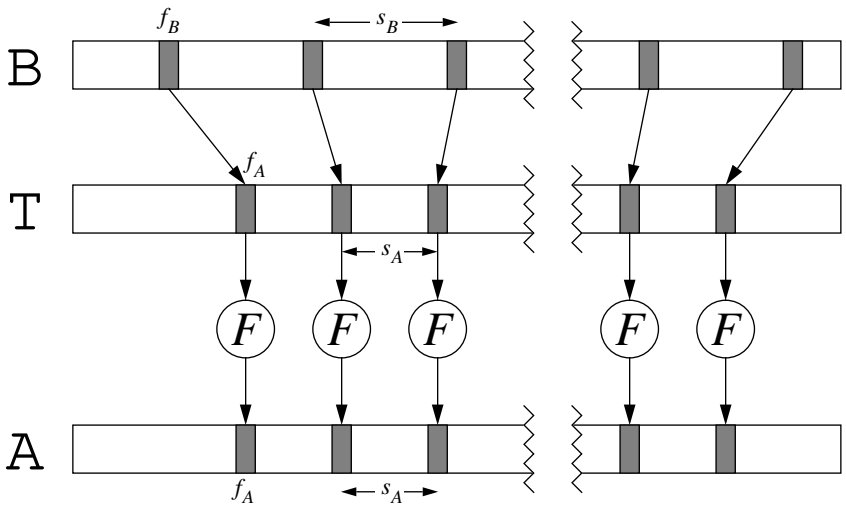

Figure 1: The two-step procedure for evaluating the basic array assignment statement.

In our approach, when we perform the translation for distributed arrays on a multiprocessor, the array `T` will be *aligned* with `A`, in the sense that corresponding elements of `A` and `T` will always be found on the same processor. This implies that the relevant section of array `B` is temporarily realigned with the relevant section of `A` to compute the result.

The array expression $\mathtt{A}(f : l : s)$ contains, as its only array index, an *array slice*. This notation is identical to the Fortran 90 *subscript triplet*. An array slice contains three colon-separated components, and represents an arithmetic sequence of integers (i.e., the difference between any two successive elements of the sequence is constant). This sequence is given by:

$$\left(f, f+s, f+2s, \ldots, f+\left\lfloor \frac{l-f}{s} \right\rfloor s\right).$$

This expression represents the sequence of integers starting at f, with stride s, and bounded above by l (bounded below by l when $s < 0$). Notice that l is not necessarily a member of the sequence, as in the slice $(1 : 10 : 2)$, which contains only odd integers.

The elements of arrays `A` and `B` are distributed across the processors in block-cyclic fashion. This method of distribution is pictured in Figure 2. The distribution could be specified by the user, or by a

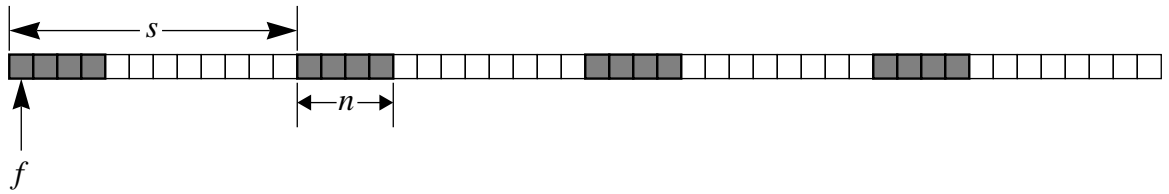

Figure 2: An example of a block-cyclic distribution. The shaded elements reside on one particular processor. The processor owns array elements corresponding to indices beginning with f, in blocks of size n. The distance between the start of one block and the start of the next is s.

separate compiler phase; in any case, we assume that the distribution has already been determined when our analysis is applied. For each processor, there is a set of indices denoting the array elements that reside on the processor. Because of the block-cyclic distribution, this set can be specified using only four parameters. We say that the processor "owns" the array elements corresponding to this index set.

We assume that scalars are owned by all processors. Therefore, every processor allocates storage for each scalar, and at any given point in the program execution, all processors have the same value for each scalar.

For the work presented in this paper, we assume that parallelism is driven by the owner-computes rule, which states that computation is performed on the processor on which the result is to be stored. Note that while the owner-computes rule is reasonably simple to implement and likely to be optimal for many statements, it is not the only model on which to base compilation algorithms. On a private memory architecture, use of the owner-computes rule implies that the compiler must generate code to perform three steps for each processor P:

Send: For every processor Q, determine which array elements are owned by P and needed for Q's computation, and send these elements from P to Q.

Receive: For every processor Q, determine which array elements are owned by Q and needed for P's computation, and receive these elements on P from Q.

Compute: For every element on the left-hand side of the equation owned by P, perform the computation and store the result locally. After having executed the previous two steps, all right-hand side data is guaranteed to be in local memory on P.

The difficult parts of these steps are:

- determining which array elements to send, and where in local memory these array elements can be found;
- determining where in local memory to store received elements;
- determining which elements to perform the computation on after receiving from all other processors.

Our approach is to devise a general method to handle all cases, and then to identify and optimize the common cases. The basis of our general method is to find intersections of slices, which can be used both in array expressions and to characterize block-cyclic ownership sets. Since a slice can be characterized by three parameters, and one additional parameter can be used to characterize an ownership set in terms of slices, the resulting intersections can also be specified using a fairly small number of parameters.

When mapping a single source program to multiple processors, a compiler can take the approach of either creating a single program to be run by all processors, or creating several different programs that are mapped onto the set of processors. In our model, parallelism is derived only at the statement (data-parallel) level, and not at the instruction-parallel level; that is, every processor is assumed to be following the same thread of control. For this reason, all processors execute similar instruction streams. Hence the analysis presented in this paper is designed to allow a compiler to produce a single program as output, which is meant to be executed by all processors, in a Single Program Multiple Data (SPMD) model.

For the remainder of the paper, we proceed using a bottom-up approach. We begin by describing how to compile the simple array assignment statement given in equation (1). This example captures the essence of the complexity of compiling array assignment statements. Then we extend the framework to handle arbitrary statements, and we describe possible optimizations. In describing the compilation of equation (1), we reduce the communication requirements to that of a single send (and the corresponding receive) between an arbitrary pair of processors. Section 6 describes how the sending processor determines which data to send, while Section 7 describes how the receiver processes and "decodes" the received data.

3 Basic requirements

There are certain basic requirements that we impose on the form of the program being compiled, and on the machine for which the program is compiled. This section describes and motivates these requirements.

We assume the existence of a general message passing system. This message passing system should be capable of supporting messages sent from one processor to itself, since the algorithms presented here make no distinction between sends to oneself and sends to a different processor. Messages received within one phase are allowed to arrive in any order; they can be processed as soon as they arrive. However, to process an incoming message correctly, the receiving processor must have a means of determining which processor sent the message. If the message passing system does not provide this functionality, the compiler must generate code to provide it (e.g., by having the sender provide its identifier along with the message).

All distributions of arrays must be block-cyclic (note that block and cyclic distributions are special cases of block-cyclic distributions). The primary reason for this choice is that the block-cyclic method provides a large number of distributions, while requiring a small, fixed number of parameters to describe. Another reason, as shown later, is that block-cyclic distributions can be described as unions of slices, as in equations (3) and (4), and slices are closed under intersection. Thus all sets over which we take intersections are described in terms of slices, as are the intersections.

For a particular array, the block size must be the same for each processor over which the array is distributed. In a heterogeneous multiprocessing environment, it might sometimes be worthwhile to assign different proportions of the array to different processors to account for differences in processing speeds. However, that kind of application is beyond the scope of this paper. In addition, if block sizes are allowed to be different, certain optimizations, such as the one presented in Section 12.1.1, cannot be performed, and certain computations must move inside an additional loop.

We use the owner-computes rule to divide the computation among the processors and to determine the communication. This assumption forms the basis of all the communication and computation analysis presented here.

Although array ownership is usually disjoint (i.e., each array element is owned by only one processor), all scalars are replicated. We wish the program execution to be deterministic and to follow the semantics of an equivalent sequential program. Thus we can think of the parallel execution as having a single thread of control. At any point in this thread of control, all processors have the same local value of a replicated variable. This restriction is necessary to preserve the sequential semantics of the program.

The final requirement involves the modulo, or "mod", operator. Throughout this paper we will use expressions in the form a mod b, where b will always be positive. However, a will sometimes be negative in this context. We use the mathematical definition of the modulo operator, in which a mod b is defined to return a value between 0 and $b - 1$, inclusive. Note that in the C programming language, the sign of $a\%b$ is implementation-dependent when either a or b is negative [11].

4 Simplifying restrictions

As our typical array statement, we have chosen the assignment statement given in equation (1). However, this statement is far from typical in most programs. For example, assignment statements could involve multiple terms on the right-hand side, arrays with more than one dimension, and scalar subscripts as well as slices.

For now, we will make a number of additional assumptions concerning the form of the array statements, all of which will be relaxed in Section 10. The assumptions are the following:

- We are considering only array assignment statements. Most other array constructs are simple extensions of the array assignment statement. The implementation of more array constructs is described in Section 13.
- Each array has only one dimension. For multidimensional array assignments, the analysis extends orthogonally for each dimension of the array, as we will show later.
- In each assignment statement, there is only one term on the right-hand side. When this restriction is relaxed, the analysis also extends orthogonally for each additional term on the right-hand side.
- For every slice, the stride component (s) is positive. One reason for this restriction involves Euclid's algorithm for computing the greatest common divisor of two integers, which requires nonnegative inputs. A more important reason is that the algorithms that will be presented assume that f is a lower bound on the sequence and l is an upper bound. This condition does not hold when the stride is negative, as in the slice $(10 : 1 : -1)$, where f is an upper bound and l is a lower bound.
- Nested array references are not allowed. An example of a nested array reference is $\mathtt{A}(\mathtt{B}(1))$ or $\mathtt{A}(\mathtt{B}(1 : n))$, where `A` and `B` are arrays. This paper will not consider ways to efficiently evaluate such nested array references, since in this case there is no longer a way, in general, to characterize the sequence of array elements using a small set of parameters.

5 Slice intersection

When deriving the algorithms below, it will be necessary to find intersections of slices. A slice parameterizes an arithmetic sequence of integers, and we need to find the intersection of two or more such slices, while preserving the ordering. As we show below, such an intersection either is empty or is another slice; thus slices are closed under intersection. Our goal is to find the *first*, *last*, and *stride* components of the intersection. Figure 3 shows two examples of slice intersections.

Our approach for finding the intersection of two slices has three conceptual steps. The first step is to extend the lower bound and the upper bound of each sequence to $-\infty$ and $+\infty$, respectively, while remembering the original bounds. The second step is to find the intersection of these infinite sequences, which, as shown below, will be either empty or another infinite sequence. The final step is to find the

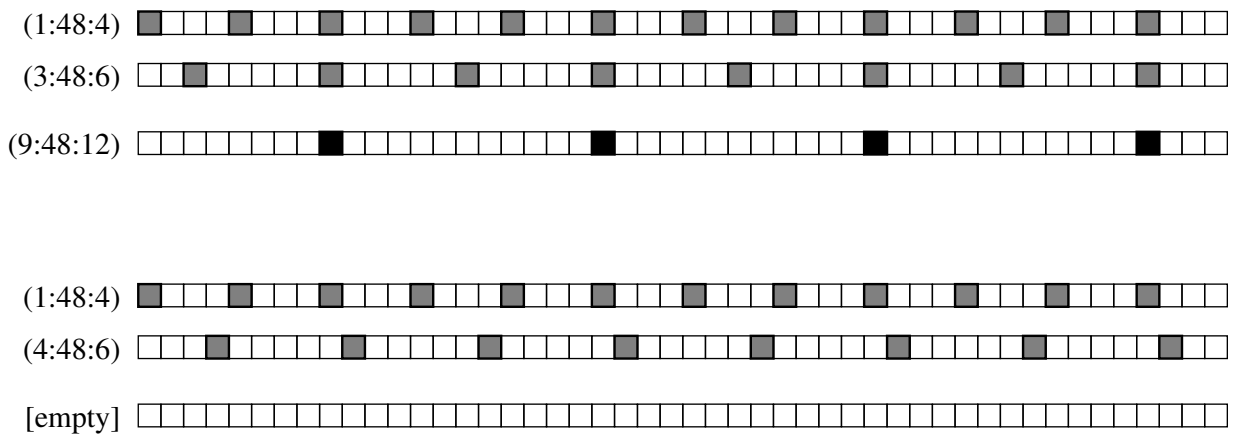

Figure 3: Two examples of slice intersection. In the first example, we see that the intersection $(1 : 48 : 4) \cap (3 : 48 : 6) = (9 : 48 : 12)$. In the second example, the intersection $(1 : 48 : 4) \cap (4 : 48 : 6)$ is empty; the slices cannot possibly share elements, since the first slice contains only odd integers, while the second slice contains only even integers.

true lower and upper bounds of the intersection by using the remembered original bounds. This approach works well for finding the intersections of three or more slices, since we can avoid the steps of converting intermediate intersections from finite sequences to infinite sequences, and vice versa. Figure 4 depicts the three steps of this approach.

Consider the slice $(f : l : s)$. Recall that f represents the first (smallest) element of the slice, l is an upper bound on elements in the set, and s is the stride. Assume that s is positive; if s is negative, then f is the largest element and l is a lower bound.

We introduce a modified representation of a slice, $(f : l : s : r)$, which represents a slice with infinite bounds, as described above, where f and l are the original remembered bounds. The s parameter serves the same purpose as before. The r parameter is a "representative" of the set, in the sense that any member of the set is equal to $r + ns$ for some integer n. For example, $(0 : 10 : 2 : 37)$ represents the same sequence as $(1 : 10 : 2)$, which is also the same as $(1 : 9 : 2)$. In this representation, the f parameter need not be a representative of the set, in the same sense that the l parameter need not be a representative of the set. The only requirement is that for any integer n, $r + ns$ is a member of the original set if and only if $f \leq r + ns \leq l$ in the new representation.

The conversion from one representation to the other is simple. The sequence given by $(f : l : s)$ is the same as that given by $(f : l : s : f)$, and the sequence given by $(f : l : s : r)$ is the same as that given by $(f + ((r - f) \bmod s) : l : s)$.

Consider the intersection of the slices $S_i = (f_i : l_i : s_i : r_i)$ and $S_j = (f_j : l_j : s_j : r_j)$. A representative of the intersection is some integer r for which there exist integers m and n such that $r = r_i + ms_i = r_j + ns_j$. This equation tells us that:

$$ms_i \equiv r_j - r_i \pmod{s_j}.$$

Let x, y, and g be integers such that $xs_i + ys_j = g = \gcd(s_i, s_j)$. These integers can be computed using Euclid's extended GCD algorithm [12, Volume 2]. A theorem of modular linear equations [7, Chapter 33] tells us that a solution exists if and only if $g|(r_j - r_i)$ [that is, g evenly divides $(r_j - r_i)$], and that a value for m is:

$$m = \frac{(r_j - r_i)x}{g}.$$

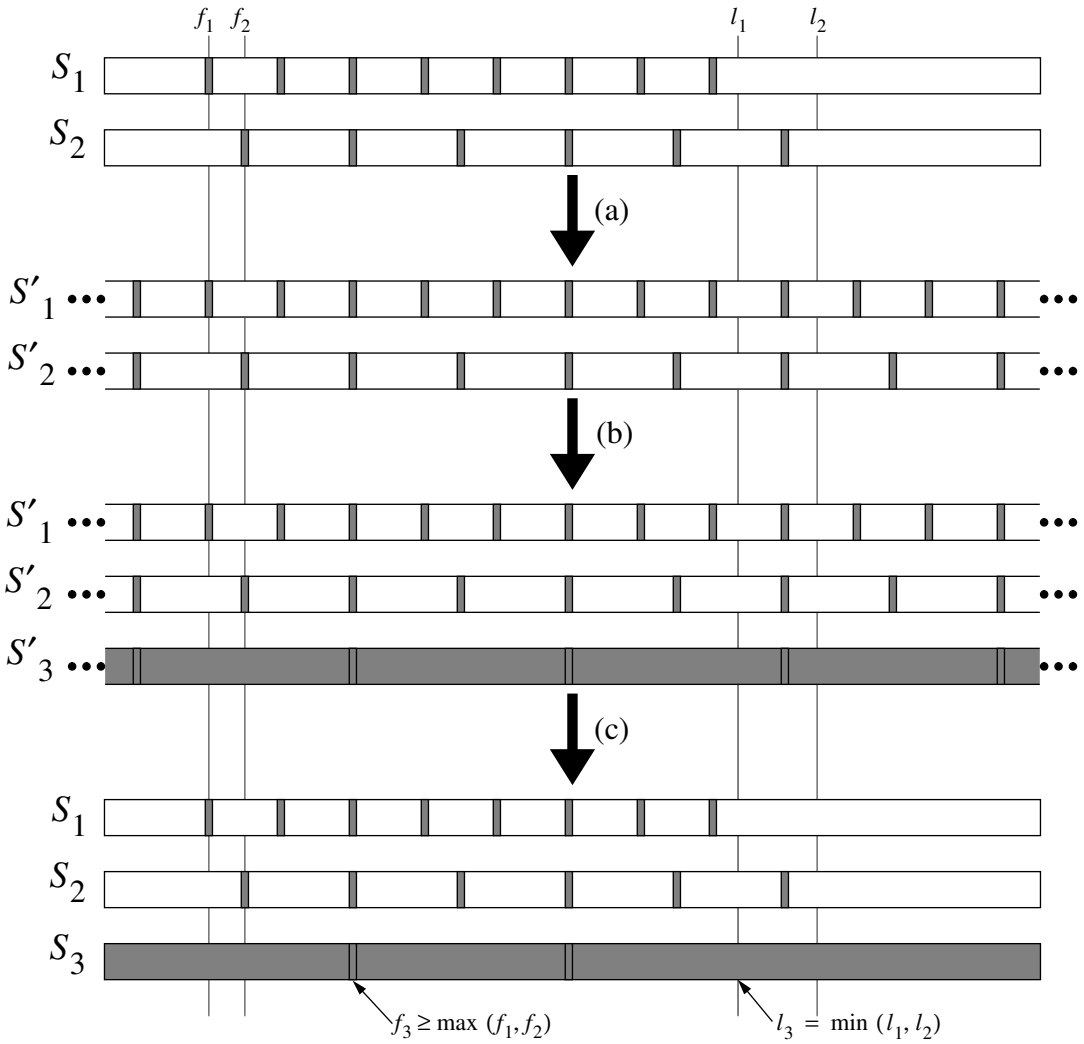

Figure 4: An illustration of the three conceptual steps involved in computing the intersection of two slices, $S_1 = (f_1 : l_1 : s_1)$ and $S_2 = (f_2 : l_2 : s_2)$. In step (a), we extend the lower and upper bounds to $-\infty$ and $+\infty$, respectively, yielding S'_1 and S'_2. In step (b), we compute $S'_3 = S'_1 \cap S'_2$. In step (c), we compute the finite bounds f_3 and l_3.

By substituting:

$$r = r_i + ms_i = r_i + \frac{(r_j - r_i)xs_i}{g}.$$

It is clear that the resulting stride of the intersection will be $\text{lcm}(s_i, s_j) = s_i s_j / g$. As our new upper and lower bounds, we must take the tightest of the two original bounds. Therefore:

$$(f_i : l_i : s_i : r_i) \cap (f_j : l_j : s_j : r_j) = (\max(f_i, f_j) : \min(l_i, l_j) : s_i s_j / g : r_i + (r_j - r_i)xs_i/g). \quad (2)$$

Note that the representative is not necessarily the same as either of the lower bounds. This observation provides the motivation for using this modified slice representation. If we are taking the intersections of three or more slices, as we will in later sections, it will be more efficient to minimize the number of conversions between representations.

6 Sends

This section describes how a processor determines what data to send to another processor. The data is described in terms of which elements of the right-hand side array are needed by the receiving processor and can be supplied by the sending processor. We also describe how the sending processor maps the global array indices to local memory.

6.1 Derivation

Suppose that we have an assignment statement in the form of equation (1). We want to determine which elements of B should be sent from processor $\mathcal{S}$ to processor $\mathcal{D}$. First, we need a means of describing the set of array elements that a particular processor owns. Four parameters suffice: f, l, s, and n. The f parameter is the lowest-numbered index of the array elements owned by the processor; the l parameter is an upper bound on the highest-numbered index of the array elements owned; the n parameter is the block size; and the s parameter is the offset between the start of one block and the start of the next. With these four parameters, a block-cyclic distribution can be described as the disjoint union of slices $\bigcup_{k=0}^{n-1}(f+k:l:s)$, as pictured in Figure 5.

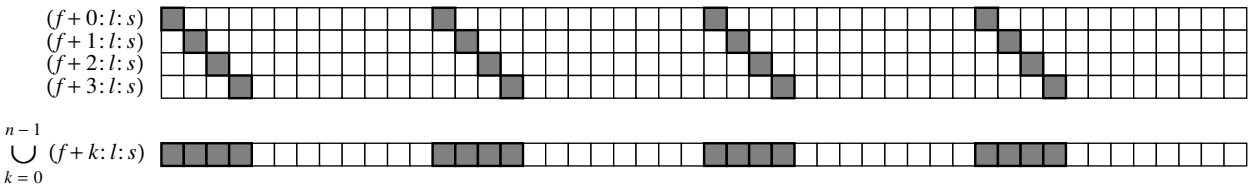

Figure 5: A block-cyclic distribution as a union of disjoint slices.

Since A and B have block-cyclic distributions, we are given integers $f_{\mathcal{D}}$, $l_{\mathcal{D}}$, $n_{\mathcal{D}}$, $s_{\mathcal{D}}$, $f_{\mathcal{S}}$, $l_{\mathcal{S}}$, $n_{\mathcal{S}}$, and $s_{\mathcal{S}}$ such that the portion of A that processor $\mathcal{D}$ owns is characterized by:

$$Own_{\mathcal{D}}(\mathtt{A}) = \bigcup_{j'=0}^{n_{\mathcal{D}}-1} (f_{\mathcal{D}} + j' : l_{\mathcal{D}} : s_{\mathcal{D}}) \tag{3}$$

and the portion of B that processor $\mathcal{S}$ owns is characterized by:

$$Own_{\mathcal{S}}(\mathtt{B}) = \bigcup_{i'=0}^{n_{\mathcal{S}}-1} (f_{\mathcal{S}} + i' : l_{\mathcal{S}} : s_{\mathcal{S}}). \tag{4}$$

Note that the names of processors $\mathcal{S}$ and $\mathcal{D}$ do not appear in the following analysis; they are implicitly encoded in the parameters $f_{\mathcal{D}}$ and $f_{\mathcal{S}}$. This encoding is described in Section 11. Also note that each ownership set is the union of disjoint sets, provided that $n_{\mathcal{D}} \leq s_{\mathcal{D}}$ and $n_{\mathcal{S}} \leq s_{\mathcal{S}}$.

The basic method for determining the communication sets is to compute unions and intersections of regular sets (i.e., slices), which can be characterized with 3 parameters. Although slices are closed under intersection, they are not closed under union; hence unions of slices, and intersections of unions of slices, require more than 3 parameters to represent, and require more complexity than simple slices to enumerate. Figure 6 gives an example of the communication that might arise between processors during the evaluation of the assignment statement $\mathtt{A}(1:n:1) = \mathtt{B}(1:n:1)$. Each array is distributed over two processors, where A has a block size of 3 and B has a block size of 5. The shaded boxes represent the elements of each array that are owned by the corresponding processor. The arrows represent the array elements that processor

s must send and that processors d_1 and d_2 must receive; hence the arrows correspond to the intersection of the block-cyclic ownership sets. Each set of arrows represents the intersection of the corresponding unions of slices; note that it is complex to represent the intersection, even though the ownership sets have simple representations as unions of slices. The complexity increases when we use non-unit stride components in the assignment statement. Note also in this example that there is little similarity between the two intersections (one intersection is represented by upward arrows, the other by downward arrows).

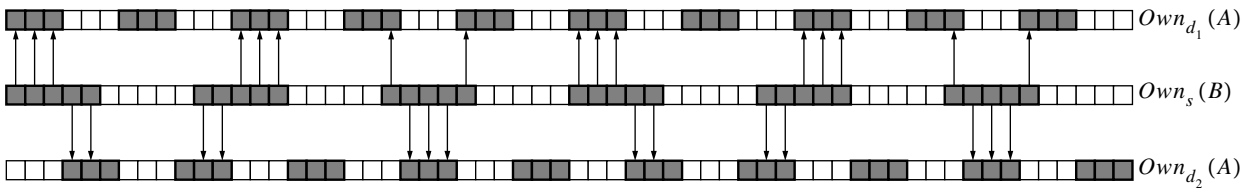

Figure 6: An example of the potential complexity of transferring data from one processor s.

The set of indices corresponding to the array elements of A owned by processor $\mathcal{D}$ that are used in the computation is given by $S_1 = Own_{\mathcal{D}}(\mathtt{A}) \cap (f_A : l_A : s_A)$. The set of indices corresponding to the array elements of B owned by processor $\mathcal{S}$ that are used in the computation is given by $S_2 = Own_{\mathcal{S}}(\mathtt{B}) \cap (f_B : l_B : s_B)$. Processor $\mathcal{S}$ sends the array element corresponding to the ith member of the sequence $(f_B : l_B : s_B)$ if and only if the ith member of the sequence $(f_B : l_B : s_B)$ is in S_2 and the ith member of the sequence $(f_A : l_A : s_A)$ is in S_1. Suppose index x is member i of the sequence $(f_B : l_B : s_B)$, where $i = 0$ represents the first member of the sequence. Then, $i = \frac{x - f_B}{s_B}$. Now member i of the sequence $(f_A : l_A : s_A)$ is $f_A + is_A$, which is equal to $\frac{x - f_B}{s_B} s_A + f_A$. Thus processor $\mathcal{S}$ sends the array element corresponding to index x if and only if $x \in S_2$ and $\frac{x - f_B}{s_B} s_A + f_A \in S_1$. Equivalently, processor $\mathcal{S}$ sends the array element corresponding to index $\frac{y - f_A}{s_A} s_B + f_B$ if and only if $y \in S_1$ and $\frac{y - f_A}{s_A} s_B + f_B \in S_2$.

The above observation motivates the definition of a Map function, which maps a member of the sequence $(f_A : l_A : s_A)$ to the corresponding member of the sequence $(f_B : l_B : s_B)$. It is defined on an index y as:

$$Map(y) = \frac{y - f_A}{s_A} \cdot s_B + f_B, \tag{5}$$

and the Map function is defined on a set or sequence in the obvious fashion. The purpose of this Map function is to associate $\mathtt{A}(f_A)$ with $\mathtt{B}(f_B) = \mathtt{B}(Map(f_A))$, to associate $\mathtt{A}(f_A + s_A)$ with $\mathtt{B}(f_B + s_B) = \mathtt{B}(Map(f_A + s_A))$, to associate $\mathtt{A}(f_A + 2s_A)$ with $\mathtt{B}(f_B + 2s_B) = \mathtt{B}(Map(f_A + 2s_A))$, etc. Then processor $\mathcal{S}$ sends the array element corresponding to index $Map(y)$ if and only if $y \in S_1$ and $Map(y) \in S_2$. Thus the set of elements of B that processor $\mathcal{S}$ sends to processor $\mathcal{D}$ corresponds to the following set of indices:

$$\Psi = S_2 \cap Map(S_1) = (Own_{\mathcal{S}}(\mathtt{B}) \cap (f_B : l_B : s_B)) \cap Map(Own_{\mathcal{D}}(\mathtt{A}) \cap (f_A : l_A : s_A)).$$

But since $Map(f_A : l_A : s_A) = (f_B : l_B : s_B)$, we can remove the $(f_B : l_B : s_B)$ term. Note that we cannot remove the $(f_A : l_A : s_A)$ term, since the Map function is only defined on elements in the set $(f_A : l_A : s_A)$. Thus the intersection to compute is the following:

$$\Psi = Own_{\mathcal{S}}(\mathtt{B}) \cap Map(Own_{\mathcal{D}}(\mathtt{A}) \cap (f_A : l_A : s_A)). \tag{6}$$

In the following derivation, we ignore the upper bounds of the slices, because in the definition of the $Own_{\mathcal{D}}(\mathtt{A})$ set in equation (3), the upper bound of each slice is independent of the value of j'. The same

holds true for $Own_{\mathcal{S}}(\mathtt{B})$ in equation (4). This independence allows us to delay the derivation of the upper bound of the intersection until the end.

Our basic algorithm for computing Ψ takes the following form:

$\Psi = \{\}$
DO $j' = 0$ UPTO $n_{\mathcal{D}} - 1$
  DO $i' = 0$ UPTO $n_{\mathcal{S}} - 1$
    $\Psi = \Psi \cup ((f_{\mathcal{S}} + i' : l_{\mathcal{S}} : s_{\mathcal{S}}) \cap Map((f_{\mathcal{D}} + j' : l_{\mathcal{D}} : s_{\mathcal{D}}) \cap (f_A : l_A : s_A)))$
  END DO
END DO

Note that some of the intersections will be empty, regardless of the upper bounds of the slices. Our algorithm takes this possibility into account, and only iterates over values of i' and j' for which the intersection is nonempty (assuming large upper bounds). Restricting the iteration set is accomplished by transforming the i' and j' indices into i and j, with index bounds chosen to avoid empty intersections.

To compute Ψ, we first compute the following intermediate set, which is a portion of equation (6):

$$Own_{\mathcal{D}}(\mathtt{A}) \cap (f_A :: s_A) = \bigcup_{j'=0}^{n_{\mathcal{D}}-1} \{(f_{\mathcal{D}} + j' :: s_{\mathcal{D}}) \cap (f_A :: s_A)\}.$$

This intermediate set is computed using the methods described in Section 5. We use Euclid's algorithm to find integers x_1, y_1, and g_1 such that $x_1 s_A + y_1 s_{\mathcal{D}} = g_1 = \gcd(s_A, s_{\mathcal{D}})$. We define a function $\mathcal{R}_I$ which extracts a set of representative from an input set, using the concept of a representative of a slice presented in Section 5. The subscript I is a list of indices. For each valid instantiation of I, there is one associated representative. For example, $\mathcal{R}_{i,j},\ 1 \le i \le 10,\ 1 \le j \le 10$, produces a set of 100 representative elements. Representatives of the intermediate set, according to equation (2), are:

$$\mathcal{R}_{j'}(Own_{\mathcal{D}}(\mathtt{A}) \cap (f_A :: s_A)) = f_A + \frac{(f_{\mathcal{D}} + j' - f_A) x_1 s_A}{g_1}$$

for the values of $0 \le j' \le n_{\mathcal{D}} - 1$ where $g_1 | (f_{\mathcal{D}} + j' - f_A)$. Where g_1 does not divide $f_{\mathcal{D}} + j' - f_A$, the corresponding slice intersection is empty and there is no representative. The resulting stride is $\frac{s_A s_{\mathcal{D}}}{g_1}$, for all j'.

We can introduce a new integer variable j where $g_1 j = f_{\mathcal{D}} + j' - f_A$, giving us $j' = g_1 j + f_A - f_{\mathcal{D}}$. Since $0 \le j' \le n_{\mathcal{D}} - 1$, we have:

$$\left\lceil \frac{f_{\mathcal{D}} - f_A}{g_1} \right\rceil \le j \le \left\lfloor \frac{f_{\mathcal{D}} - f_A + n_{\mathcal{D}} - 1}{g_1} \right\rfloor.$$

This change of variables allows us to transform our intermediate set representatives to the following:

$$\mathcal{R}_j(Own_{\mathcal{D}}(\mathtt{A}) \cap (f_A :: s_A)) = f_A + j x_1 s_A \tag{7}$$

$$\left\lceil \frac{f_{\mathcal{D}} - f_A}{g_1} \right\rceil \le j \le \left\lfloor \frac{f_{\mathcal{D}} - f_A + n_{\mathcal{D}} - 1}{g_1} \right\rfloor \tag{8}$$

Note that for some possible values of the parameters, the lower bound of j may in fact exceed the upper bound, in which case there are no legitimate values of j and hence no representatives. We apply the Map function to equation (7) to get us one step closer to Ψ, and find that a representative is:

$$\mathcal{R}_j(Map(Own_{\mathcal{D}}(\mathtt{A}) \cap (f_A :: s_A))) = f_B + jx_1 s_B \tag{9}$$
$$\left\lceil \frac{f_{\mathcal{D}} - f_A}{g_1} \right\rceil \leq j \leq \left\lfloor \frac{f_{\mathcal{D}} - f_A + n_{\mathcal{D}} - 1}{g_1} \right\rfloor$$

and the new stride is $\frac{s_B s_{\mathcal{D}}}{g_1}$, for all j.

Now we intersect equation (9) with $Own_{\mathcal{S}}(\mathtt{B})$ to yield a representative of equation (6):

$$\mathcal{R}_{j,i'}(\Psi) = \mathcal{R}_{j,i'}\left(\left(f_B + jx_1 s_B :: \frac{s_B s_{\mathcal{D}}}{g_1}\right) \cap (f_{\mathcal{S}} + i' :: s_{\mathcal{S}})\right)$$
$$\left\lceil \frac{f_{\mathcal{D}} - f_A}{g_1} \right\rceil \leq j \leq \left\lfloor \frac{f_{\mathcal{D}} - f_A + n_{\mathcal{D}} - 1}{g_1} \right\rfloor,\ 0 \leq i' \leq n_{\mathcal{S}} - 1$$

In the same manner as before, we run Euclid's algorithm on $\frac{s_B s_{\mathcal{D}}}{g_1}$ and $s_{\mathcal{S}}$ to yield integers x_2, y_2, and g_2 such that $x_2 \cdot \frac{s_B s_{\mathcal{D}}}{g_1} + y_2 s_{\mathcal{S}} = g_2 = \gcd(\frac{s_B s_{\mathcal{D}}}{g_1}, s_{\mathcal{S}})$. Since the intersection is nonempty only when $g_2 | (f_{\mathcal{S}} + i' - f_B - jx_1 s_B)$, we define as before an integer i such that $g_2 i = f_{\mathcal{S}} + i' - f_B - jx_1 s_B$, or equivalently, $i' = g_2 i - f_{\mathcal{S}} + f_B + jx_1 s_B$. Since $0 \leq i' \leq n_{\mathcal{S}} - 1$, we find that:

$$\left\lceil \frac{f_{\mathcal{S}} - f_B - jx_1 s_B}{g_2} \right\rceil \leq i \leq \left\lfloor \frac{f_{\mathcal{S}} - f_B - jx_1 s_B + n_{\mathcal{S}} - 1}{g_2} \right\rfloor.$$

Once again, note that the lower bound of i could exceed the upper bound, in which case there are no representatives for that particular value of j. Thus representatives of the final result Ψ are given by:

$$\mathcal{R}_{j,i}(\Psi) = f_B + jx_1 s_B + \frac{ix_2 s_B s_{\mathcal{D}}}{g_1} \tag{10}$$
$$\left\lceil \frac{f_{\mathcal{D}} - f_A}{g_1} \right\rceil \leq j \leq \left\lfloor \frac{f_{\mathcal{D}} - f_A + n_{\mathcal{D}} - 1}{g_1} \right\rfloor \tag{11}$$
$$\left\lceil \frac{f_{\mathcal{S}} - f_B - jx_1 s_B}{g_2} \right\rceil \leq i \leq \left\lfloor \frac{f_{\mathcal{S}} - f_B - jx_1 s_B + n_{\mathcal{S}} - 1}{g_2} \right\rfloor \tag{12}$$

The resulting stride is $\frac{s_B s_{\mathcal{D}} s_{\mathcal{S}}}{g_1 g_2}$.

Now that we have the representatives, we can find the corresponding lower bounds of the slices. Let c be the tightest lower bound, given the original lower bounds; it is defined as:

$$c = \max\{Map(f_A), Map(f_{\mathcal{D}}), f_{\mathcal{S}}\} = \max\{f_B, \left\lceil \frac{f_{\mathcal{D}} - f_A}{s_A} \right\rceil \cdot s_B + f_B, f_{\mathcal{S}}\}. \tag{13}$$

Then as shown in Section 5, the true lower bound of the slice is given as:

$$c + \left((\mathcal{R}_{j,i}(\Psi) - c) \bmod \frac{s_B s_{\mathcal{D}} s_{\mathcal{S}}}{g_1 g_2}\right) = c + \left((f_B + jx_1 s_B + \frac{ix_2 s_B s_{\mathcal{D}}}{g_1} - c) \bmod \frac{s_B s_{\mathcal{D}} s_{\mathcal{S}}}{g_1 g_2}\right). \tag{14}$$

Finally, we need to find the upper bound of the slices:

$$\min\{Map(l_A), Map(l_{\mathcal{D}}), l_{\mathcal{S}}\} = \min\left(\left\lfloor \frac{\min(l_A, l_{\mathcal{D}}) - f_A}{s_A} \right\rfloor \cdot s_B + f_B, l_{\mathcal{S}}\right). \tag{15}$$

6.2 Algorithm

Given the assignment statement from equation (1), where processor $\mathcal{S}$ owns elements of B characterized by $f_{\mathcal{S}}$, $l_{\mathcal{S}}$, $s_{\mathcal{S}}$, and $n_{\mathcal{S}}$ according to equation (4), and processor $\mathcal{D}$ owns elements of A characterized by $f_{\mathcal{D}}$, $l_{\mathcal{D}}$, $s_{\mathcal{D}}$, and $n_{\mathcal{D}}$ according to equation (3), we compute the set of elements of B that processor $\mathcal{S}$ should send to processor $\mathcal{D}$. Figure 7 gives an algorithm, GLOBAL–INDEX–SEND, to determine this set. It is derived from equations (10), (11), (12), (13), (14), and (15).

$(x_1, y_1, g_1) = \texttt{euclid}(s_A, s_{\mathcal{D}})$
$(x_2, y_2, g_2) = \texttt{euclid}(s_B s_{\mathcal{D}}/g_1, s_{\mathcal{S}})$
$\texttt{stride} = s_B s_{\mathcal{D}} s_{\mathcal{S}}/(g_1 g_2)$
$\texttt{last} = \min(f_B + s_B \lfloor(\min(l_A, l_{\mathcal{D}}) - f_A)/s_A\rfloor, l_{\mathcal{S}})$ (15)
$c = \max(f_B + s_B \lceil(\max(f_A, f_{\mathcal{D}}) - f_A)/s_A\rceil, f_{\mathcal{S}})$ (13)
DO $j = \lceil(f_{\mathcal{D}} - f_A)/g_1\rceil$ UPTO $\lfloor(f_{\mathcal{D}} - f_A + n_{\mathcal{D}} - 1)/g_1\rfloor$ (11)
  DO $i = \lceil(f_{\mathcal{S}} - f_B - jx_1 s_B)/g_2\rceil$ UPTO $\lfloor(f_{\mathcal{S}} - f_B - jx_1 s_B + n_{\mathcal{S}} - 1)/g_2\rfloor$ (12)
    $r = f_B + jx_1 s_B + ix_2 s_B s_{\mathcal{D}}/g_1$ (10)
    $\texttt{first} = c + ((r - c) \bmod \texttt{stride})$ (14)
    output-slice(first, last, stride)

Figure 7: GLOBAL–INDEX–SEND algorithm. The numbers in the right-most column refer to the equation from which the corresponding line was derived.

In practice, when computing the value of the representative r, we can sometimes get a value that is not expressible in 32 bits. This potential overflow is due to the term $ix_2 s_B s_{\mathcal{D}}/g_1$, and the fact that i and x_2 can have quite large magnitudes. In this case, we can make use of the fact that we can add or subtract multiples of the stride and still have a valid representative. For example, we can replace the term $ix_2 s_B s_{\mathcal{D}}/g_1$ with $\frac{s_B s_{\mathcal{D}}}{g_1}([x_2(i \bmod \frac{s_{\mathcal{S}}}{g_2})] \bmod \frac{s_{\mathcal{S}}}{g_2})$.

6.3 Local memory

While the analysis above determines which global indices of the array elements should be sent, we still need to map these global indices to the processor's local memory. Here we give a mapping function.

One possible mapping is simply the identity mapping, where the global indices and the local indices are the same. However, the identity mapping is quite wasteful of memory, and one goal of parallelizing applications is to allow problems with larger memory requirements to be solved.

To minimize the amount of wasted memory space, we will want to compact the array (i.e., place the blocks contiguously in memory). Hiranandani et al. [10] argue that this compaction is actually necessary, and should not be viewed merely as an optimization. Given a global index x and a block-cyclic distribution characterized as above in terms of f_i, l_i, n_i, and s_i, we find that the local index $LM_i(x)$ is defined as:

$$LM_i(x) = \overbrace{\left\lfloor \frac{x - f_i}{s_i} \right\rfloor}^{block} \cdot n_i + \overbrace{((x - f_i) \bmod s_i)}^{offset} + \texttt{K} \tag{16}$$

where K is the index of the first array element. In the C programming language, K is 0; in Fortran K is usually 1. This mapping is illustrated in Figure 8.

Note that equation (16) is only defined on elements owned by the processor, $\bigcup_{k=0}^{n_i - 1}(f_i + k : l_i : s_i)$. However, we will wish to apply the function to last (one of the parameters defined in Figure 7), which

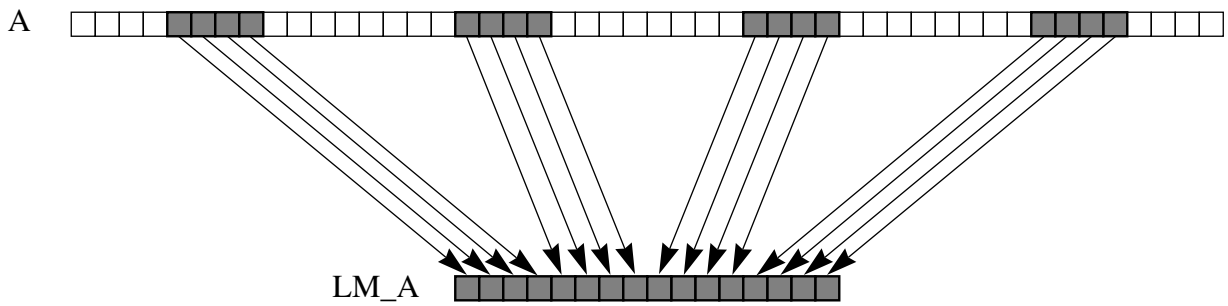

Figure 8: An example of the local memory mapping function for a particular processor. The processor owns array elements of A with global indices beginning with $f = 5$, with block size $n = 4$ and stride $s = 12$. These global indices are mapped to contiguous indices in local memory.

is not necessarily an element owned by the processor. In this case, we must simply ensure that the *offset* portion of the equation is less than n_i, by the use of the *min* function.

It is easy to see that $LM_i(x + s_i) = LM_i(x) + n_i$. This property is useful when doing the local memory mapping for a slice, where the stride component of the slice is a multiple of s_i. In this case, we need only evaluate the function for the first and the last component of the slice, and not for every element of the slice.

Another useful property of the LM_i function can be used in doing the local memory mapping for the GLOBAL–INDEX–SEND algorithm. We need to compute $LM(\texttt{first})$, which can be done more efficiently using the property that the *offset* portion of $LM(\texttt{first})$ is $(\texttt{first} - f_{\mathcal{S}}) \bmod s_{\mathcal{S}} = i'$, where $i' = g_2 i - f_{\mathcal{S}} + f_B + j x_1 s_B$.

These observations allow us to formulate an algorithm for computing the local indices of array elements to send, LOCAL–INDEX–SEND, shown in Figure 9.

$(x_1, y_1, g_1) = \texttt{euclid}(s_A, s_{\mathcal{D}})$
$(x_2, y_2, g_2) = \texttt{euclid}(s_B s_{\mathcal{D}}/g_1, s_{\mathcal{S}})$
$\texttt{stride} = s_B s_{\mathcal{D}} s_{\mathcal{S}}/(g_1 g_2)$
$\texttt{lmstride} = s_B s_{\mathcal{D}} n_{\mathcal{S}}/(g_1 g_2)$
$\texttt{last} = \min(f_B + s_B \lfloor (\min(l_A, l_{\mathcal{D}}) - f_A)/s_A \rfloor, l_{\mathcal{S}})$
$\texttt{lmlast} = \lfloor (\texttt{last} - f_{\mathcal{S}})/s_{\mathcal{S}} \rfloor n_{\mathcal{S}} + \min((\texttt{last} - f_{\mathcal{S}}) \bmod s_{\mathcal{S}}, n_{\mathcal{S}} - 1) + \texttt{K}$
$c = \max(f_B + s_B \lceil (\max(f_A, f_{\mathcal{D}}) - f_A)/s_A \rceil, f_{\mathcal{S}})$
DO $j = \lceil (f_{\mathcal{D}} - f_A)/g_1 \rceil$ UPTO $\lfloor (f_{\mathcal{D}} - f_A + n_{\mathcal{D}} - 1)/g_1 \rfloor$
  DO $i = \lceil (f_{\mathcal{S}} - f_B - j x_1 s_B)/g_2 \rceil$ UPTO $\lfloor (f_{\mathcal{S}} - f_B - j x_1 s_B + n_{\mathcal{S}} - 1)/g_2 \rfloor$
    $i' = g_2 i - f_{\mathcal{S}} + f_B + j x_1 s_B$
    $r = f_B + j x_1 s_B + i x_2 s_B s_{\mathcal{D}}/g_1$
    $\texttt{first} = c + ((r - c) \bmod \texttt{stride})$
    $\texttt{lmfirst} = (\texttt{first} - f_{\mathcal{S}} - i') n_{\mathcal{S}}/s_{\mathcal{S}} + i' + \texttt{K}$
    output-local-slice(lmfirst, lmlast, lmstride)

Figure 9: LOCAL–INDEX–SEND algorithm, derived from applying the LM function to the GLOBAL–INDEX–SEND algorithm.

7 Receives

This section describes how a processor determines what data to receive from another processor. This calculation essentially involves determining the order in which the sending processor inserted array elements into the buffer. As in Section 6, the data is described in terms of which elements of the right-hand side array are needed by the receiving processor and can be supplied by the sending processor. We also describe how the receiving processor maps the global array indices to local memory.

7.1 Algorithm

As shown in Figure 1, the first step in the evaluation of the array assignment statement is to copy $\mathtt{B}(f_B : l_B : s_B)$ into $\mathtt{T}(f_A : l_A : s_A)$. This copy step can involve interprocessor communication. We choose to copy into this particular section of T so that during the second evaluation step, in which we apply $\mathcal{F}$ to T and assign the results to A, we can use the same array index to access both A and T. (The details of this second step are described in Section 8.)

To process a received message, we must copy each element from the receive buffer into some location in T. The mapping of elements from the receive buffer into T is a complex function, and is the subject of this section. In general, due to the complexity of the formulas in the send algorithm, the receiver must re-run the send algorithm to determine which elements in the receive buffer correspond to which elements of B. Then we apply the inverse of the Map function defined in equation (5) to determine where each buffer element should be placed within T. Re-running the send algorithm and applying the inverse Map function can easily be integrated into a single operation.

To run the algorithm, we of course need to know all the original parameters. Since each processor runs the same code in the SPMD model, we already know the values of f_A, l_A, s_A, f_B, l_B, and s_B. We also know which portion of the left-hand side array we own, $Own_{\mathcal{D}}(\mathtt{A})$, characterized by the parameters $f_{\mathcal{D}}$, $l_{\mathcal{D}}$, $s_{\mathcal{D}}$, and $n_{\mathcal{D}}$. The only parameters we don't necessarily know are $f_{\mathcal{S}}$, $l_{\mathcal{S}}$, $s_{\mathcal{S}}$, and $n_{\mathcal{S}}$. However, in a block-cyclic distribution, all processors involved in the distribution will have the same values of $l_{\mathcal{S}}$, $s_{\mathcal{S}}$, and $n_{\mathcal{S}}$. Thus the only parameter that has to be passed along with the message is the sending processor's value of $f_{\mathcal{S}}$. Alternatively, if the receiving processor can determine the identity of the sending processor, the receiving processor can calculate or look up the value of $f_{\mathcal{S}}$ itself.

Our algorithm to receive a slice, then, is identical to the send algorithm, except that the inverse of the Map function is applied to the assignments to last, c, and r. In addition, the s_B factor in the stride changes to s_A. The algorithm, GLOBAL–INDEX–RECEIVE, is shown in Figure 10.

$$
\begin{array}{l}
(x_1, y_1, g_1) = \mathtt{euclid}(s_A, s_{\mathcal{D}}) \\
(x_2, y_2, g_2) = \mathtt{euclid}(s_B s_{\mathcal{D}}/g_1, s_{\mathcal{S}}) \\
\mathtt{stride} = s_A s_{\mathcal{D}} s_{\mathcal{S}}/(g_1 g_2) \\
\mathtt{last} = \min(l_A, l_{\mathcal{D}}, f_A + s_A \lfloor (l_{\mathcal{S}} - f_B)/s_B \rfloor) \\
c = \max(f_A + s_A \lceil (\max(f_B, f_{\mathcal{S}}) - f_B)/s_B \rceil, f_{\mathcal{D}}) \\
\mathtt{DO}\ j = \lceil (f_{\mathcal{D}} - f_A)/g_1 \rceil\ \ \mathtt{UPTO}\ \ \lfloor (f_{\mathcal{D}} - f_A + n_{\mathcal{D}} - 1)/g_1 \rfloor \\
\quad \mathtt{DO}\ i = \lceil (f_{\mathcal{S}} - f_B - j x_1 s_B)/g_2 \rceil\ \ \mathtt{UPTO}\ \ \lfloor (f_{\mathcal{S}} - f_B - j x_1 s_B + n_{\mathcal{S}} - 1)/g_2 \rfloor \\
\quad\quad r = f_A + j x_1 s_A + i x_2 s_A s_{\mathcal{D}}/g_1 \\
\quad\quad \mathtt{first} = c + ((r - c) \bmod \mathtt{stride}) \\
\quad\quad \mathtt{input\text{-}slice}(\mathtt{first}, \mathtt{last}, \mathtt{stride})
\end{array}
$$

Figure 10: GLOBAL–INDEX–RECEIVE algorithm.

7.2 Local memory

As in the case of sends, we must convert the global indices formed by the GLOBAL–INDEX–RECEIVE algorithm into local memory indices. The mapping function is the same as before, except that we will use parameters $f_{\mathcal{D}}$, $l_{\mathcal{D}}$, $s_{\mathcal{D}}$, and $n_{\mathcal{D}}$ rather than $f_{\mathcal{S}}$, $l_{\mathcal{S}}$, $s_{\mathcal{S}}$, and $n_{\mathcal{S}}$. This substitution is necessary because the former parameters describe the layout of the left-hand side array, with which the data is to be aligned, while the latter parameters describe the layout of the right-hand side array, which is being sent. Using the same analysis as for the LOCAL–INDEX–SEND algorithm, we derive the algorithm, LOCAL–INDEX–RECEIVE, which determines the local memory indices of the data being received, as shown in Figure 11.

$(x_1, y_1, g_1) = \texttt{euclid}(s_A, s_{\mathcal{D}})$
$(x_2, y_2, g_2) = \texttt{euclid}(s_B s_{\mathcal{D}}/g_1, s_{\mathcal{S}})$
$\texttt{stride} = s_A s_{\mathcal{D}} s_{\mathcal{S}}/(g_1 g_2)$
$\texttt{lmstride} = s_A n_{\mathcal{D}} s_{\mathcal{S}}/(g_1 g_2)$
$\texttt{last} = \min(l_A, l_{\mathcal{D}}, f_A + s_A \lfloor (l_{\mathcal{S}} - f_B)/s_B \rfloor)$
$\texttt{lmlast} = \lfloor (\texttt{last} - f_{\mathcal{D}})/s_{\mathcal{D}} \rfloor n_{\mathcal{D}} + \min((\texttt{last} - f_{\mathcal{D}}) \bmod s_{\mathcal{D}}, n_{\mathcal{D}} - 1) + \texttt{K}$
$c = \max(f_A, f_{\mathcal{D}}, f_A + s_A \lceil (f_{\mathcal{S}} - f_B)/s_B \rceil)$
DO $j = \lceil (f_{\mathcal{D}} - f_A)/g_1 \rceil$ UPTO $\lfloor (f_{\mathcal{D}} - f_A + n_{\mathcal{D}} - 1)/g_1 \rfloor$
  $t = (f_A + j x_1 s_A - f_{\mathcal{D}}) \bmod s_{\mathcal{D}}$
  DO $i = \lceil (f_{\mathcal{S}} - f_B - j x_1 s_B)/g_2 \rceil$ UPTO $\lfloor (f_{\mathcal{S}} - f_B - j x_1 s_B + n_{\mathcal{S}} - 1)/g_2 \rfloor$
    $r = f_A + j x_1 s_A + i x_2 s_A s_{\mathcal{D}}/g_1$
    $\texttt{first} = c + ((r - c) \bmod \texttt{stride})$
    $\texttt{lmfirst} = (\texttt{first} - f_{\mathcal{D}} - t) n_{\mathcal{D}}/s_{\mathcal{D}} + t + \texttt{K}$
    $\texttt{input-local-slice}(\texttt{lmfirst}, \texttt{lmlast}, \texttt{lmstride})$

Figure 11: LOCAL–INDEX–RECEIVE algorithm.

8 Computation

After the communication step completes, all data to be used for the computation will be local to each processor. In addition, the LOCAL–INDEX–RECEIVE algorithm works in such a way that the portion of the right-hand side array received will be aligned with the left-hand side array. When the received data is placed in this manner, we can use the same loop index to access both the left-hand side array and the temporarily redistributed right-hand side array.

Assume once again that we are executing the statement in equation (1). When we perform the LOCAL–INDEX–RECEIVE algorithm, we store the data in a temporary array `T` which has the same size and shape as `A`. Then we compute by looping through the arrays and assigning the results.

To loop through the arrays, we first need to determine the global indices of the array elements on which to perform the computation. This set is given by:

$$Own_{\mathcal{D}}(\texttt{A}) \cap (f_A : l_A : s_A).$$

From equations (7) and (8), we have that a representative of this set is given by $r = f_A + j x_1 s_A$ for $\lceil \frac{f_{\mathcal{D}} - f_A}{g_1} \rceil \leq j \leq \lfloor \frac{f_{\mathcal{D}} - f_A + n_{\mathcal{D}} - 1}{g_1} \rfloor$. When we set $c = \max(f_A, f_{\mathcal{D}})$ to be a lower bound, we find as before that the true lower bound is given by $c + ((r - c) \bmod s_A s_{\mathcal{D}}/g_1)$.

However, we still need to map this global index to local memory. We do this by applying the previously defined LM function from equation (16) to the upper bound and the true lower bound, and converting the global stride of $s_A s_{\mathcal{D}}/g_1$ to $s_A n_{\mathcal{D}}/g_1$.

These observations yield the algorithm COMPUTE–LOOP, shown in Figure 12.

$(x_1, y_1, g_1) = \texttt{euclid}(s_A, s_{\mathcal{D}})$
$\texttt{stride} = s_A s_{\mathcal{D}}/g_1$
$\texttt{lmstride} = s_A n_{\mathcal{D}}/g_1$
$\texttt{last} = \min(l_A, l_{\mathcal{D}})$
$\texttt{lmlast} = \lfloor(\texttt{last} - f_{\mathcal{D}})/s_{\mathcal{D}}\rfloor n_{\mathcal{D}} + \min((\texttt{last} - f_{\mathcal{D}}) \bmod s_{\mathcal{D}}, n_{\mathcal{D}} - 1) + \texttt{K}$
$c = \max(f_A, f_{\mathcal{D}})$
DO $j = \lceil(f_{\mathcal{D}} - f_A)/g_1\rceil$ UPTO $\lfloor(f_{\mathcal{D}} - f_A + n_{\mathcal{D}} - 1)/g_1\rfloor$
  $j' = g_1 j + f_A - f_{\mathcal{D}}$
  $r = f_A + j x_1 s_A$
  $\texttt{first} = c + ((r - c) \bmod \texttt{stride})$
  $\texttt{lmfirst} = (\texttt{first} - f_{\mathcal{D}} - j')n_{\mathcal{D}}/s_{\mathcal{D}} + j' + \texttt{K}$
  DO $i = \texttt{lmfirst}$ UPTO lmlast BY lmstride
    $\texttt{A}(i) = \mathcal{F}(\texttt{T}(i))$

Figure 12: COMPUTE–LOOP algorithm, which determines which elements of the array to compute once the communication has completed.

9 Overall algorithm

From the LOCAL–INDEX–SEND, LOCAL–INDEX–RECEIVE, and COMPUTE–LOOP algorithms, we now have the means to construct a general algorithm for executing a single array assignment statement. This algorithm, which executes on all processors in the SPMD style, takes the following form:

/* Send phase */
  $\mathcal{S} = \texttt{getmyid}()$
  $f_{\mathcal{S}} = \texttt{getfirst}(\texttt{B}, \mathcal{S})$
  DO $\mathcal{D} = 0$ UPTO $N - 1$
    $f_{\mathcal{D}} = \texttt{getfirst}(\texttt{A}, \mathcal{D})$
    CALL LOCAL–INDEX–SEND
  END DO
/* Receive phase */
  $\mathcal{D} = \texttt{getmyid}()$
  $f_{\mathcal{D}} = \texttt{getfirst}(\texttt{A}, \mathcal{D})$
  DO $\mathcal{S} = 0$ UPTO $N - 1$
    $f_{\mathcal{S}} = \texttt{getfirst}(\texttt{B}, \mathcal{S})$
    CALL LOCAL–INDEX–RECEIVE
  END DO
/* Computation phase */
  CALL COMPUTE–LOOP

In this pseudo-code we assume that the system contains N processors, and each processor has a unique ID, returned by the function `getmyid`, between 0 and $N - 1$, inclusive. The function `getfirst` returns

the first index of the array (the first function argument) that is owned by the processor (the second function argument), as given in equations (3) and (4). Note that this overall algorithm implies that a processor may send a message to itself and correspondingly receive from itself.

An illustration of the communication algorithm is given in Figure 13. In the first conceptual step, a sending processor computes an intersection of global array indices. In the second step, the processor maps these global indices to local indices. Next, the processor copies the required array elements into a send buffer; note the irregular order in which the elements are copied. Finally, the sending processor sends the buffer to the destination processor. The destination processor receives the data into a receive buffer, and proceeds to copy the data from the receive buffer into a temporary array corresponding to the left-hand side array. Now that the destination processor has received messages from both sending processors, it can continue with its computation.

Each of the N send phases and N receive phases are data independent and can be executed in any order if we have a buffered message passing system. However, there is a dependence between the send on one processor and the corresponding receive on the destination processor. Thus even with a buffered message passing system, we must schedule the send and receive operations to avoid deadlock. For example, scheduling deadlock could occur if the first operation on every processor is a receive, since every processor would be waiting on a receive but no processor would be sending.

Suppose the message passing system is not buffered, and the program is executing on a MIMD architecture. It is possible for a "fast" processor to complete its communication during phase n, proceed with its computation, and begin communication phase $n + 1$, while a "slow" processor is still trying to complete communication phase n. If the fast processor now sends a message to the slow processor, the slow processor has the problem of distinguishing phase n messages from phase $n + 1$ messages. This potential problem can be alleviated by inserting a barrier synchronization before each communication phase. No barrier is necessary on a SIMD architecture, since the synchronization is implicit.

10 Relaxing restrictions

In this section, we relax the simplifying restrictions imposed in Section 4, regarding the format and type of array statements allowed.

10.1 Multiple right-hand side terms

The right-hand side of the assignment statement could have multiple terms. Each term could take one of the following forms:

- An array slice, such as $\texttt{B}(1 : n)$
- A single array element, such as $\texttt{B}(i)$
- A scalar value, such as 3 or `x`

In the first case, the relevant portions of an array slice will be sent to each processor and stored in a temporary array, aligned with the left-hand side array. In the second case, the single array element will be broadcast to all processors involved in the computation, and stored in a temporary scalar variable. In the third case, the scalar value will be used in place without any communication, since its value will be consistent across all processors.

If there are several terms on the right-hand side, then we simply perform a communication step if necessary for each term. The receiving processor must have one temporary array or scalar for each term

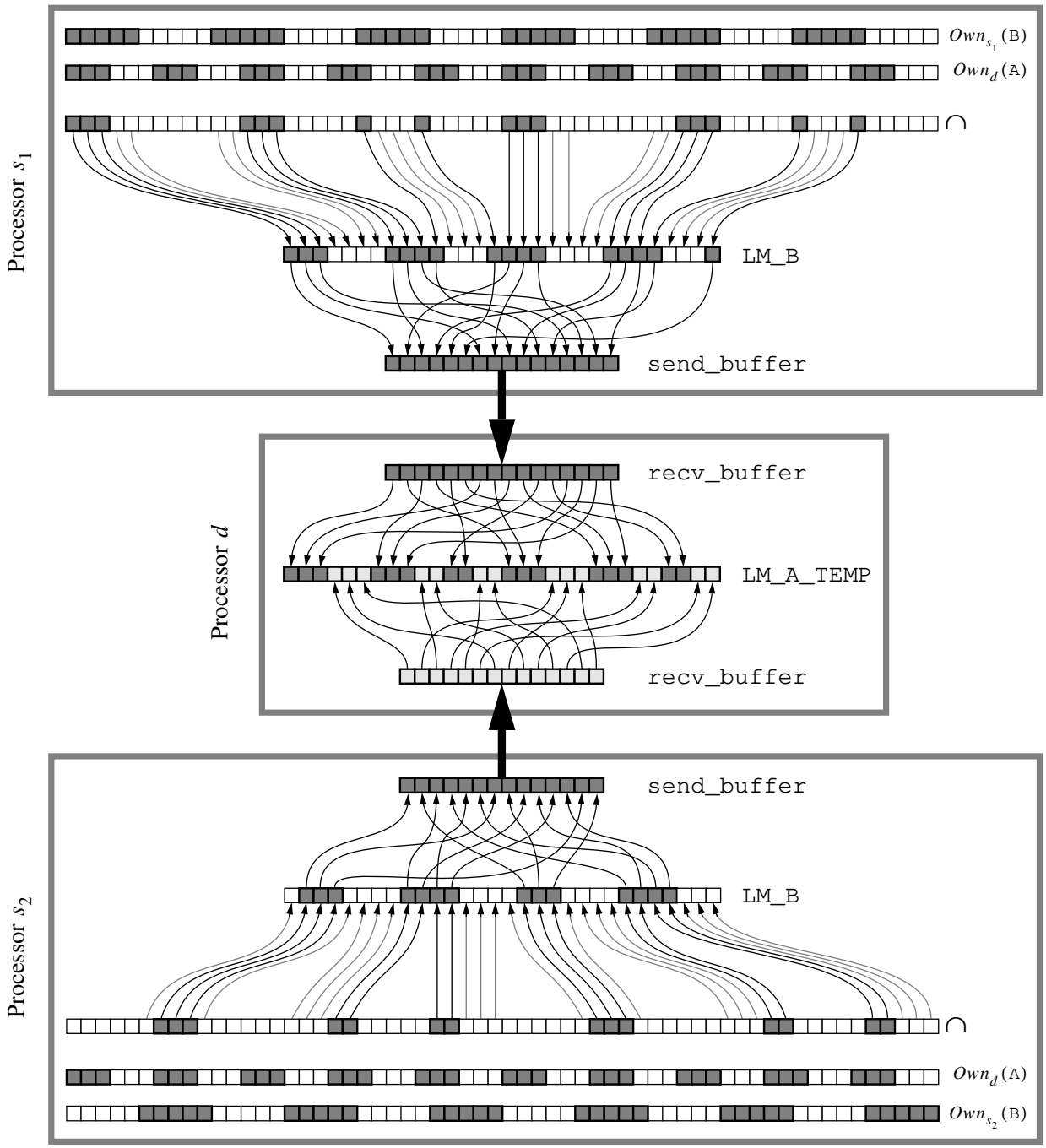

Figure 13: Communication algorithm. Processors s_1 (top) and s_2 (bottom) send to processor d (center).

on the right-hand side. The computation loop is performed as before, except that the right-hand side of the assignment statement in the inner loop has a reference to each temporary variable.

An even better method for handling multiple right-hand side terms would be to group the messages together and send only one message to each processor, since this grouping would eliminate the overhead of the additional messages. The send buffer would then contain data for the entire right-hand side, rather than for just one right-hand side term. If it was determined that there were no data dependences between several consecutive statements, a still better method would be to block all data together involving right-hand side terms for those statements.

10.2 Multidimensional arrays

For the most part, the analysis performed above extends orthogonally for multidimensional arrays. For example, if we had the statement:

$$\texttt{A}(\texttt{slice1}, \texttt{slice2}) = \mathcal{F}(\texttt{B}(\texttt{slice3}, \texttt{slice4})),$$

we would determine the indices of the array elements in the first dimension of B owned by processor $\mathcal{S}$ that needed to be sent to processor $\mathcal{D}$, and then do the same for the second dimension. The Cartesian product of these two sets would correspond to the array elements to be sent.

For example, suppose $\{i_1, \ldots, i_m\}$ is the set created by the intersections for the first dimension, and $\{j_1, \ldots, j_n\}$ is the set created by the intersections for the second dimension. Then the elements to be sent are $\texttt{B}(i_1, j_1)$, $\texttt{B}(i_1, j_2)$, ..., $\texttt{B}(i_1, j_n)$, $\texttt{B}(i_2, j_1)$, $\texttt{B}(i_2, j_2)$, ..., $\texttt{B}(i_m, j_n)$.

But problems arise when an array reference contains both scalar and slice indices, such as in the statement:

$$\texttt{A}(i, \texttt{slice1}) = \mathcal{F}(\texttt{B}(\texttt{slice2}, j)).$$

The main problem here is that the analysis presented so far assumes that all array indices are slices, not scalars. A secondary problem is that the compiler must be sure to match up slice1 with slice2, to preserve the semantics of the array assignment statement.

To compile multidimensional array statements, we must do the following. We modify the definition of the Own function such that $Own_p(\texttt{C}, k)$ represents the set of indices of array elements that processor p owns in dimension k of C. We observe that in the above statement, processor $\mathcal{S}$ performs a send only if $j \in Own_{\mathcal{S}}(\texttt{B}, 2)$. Likewise, processor $\mathcal{D}$ receives data only if $i \in Own_{\mathcal{D}}(\texttt{A}, 1)$. Therefore, each processor can participate in a send or receive only if it owns the array element corresponding to the index in each scalar dimension of the array.

After the scalar dimensions are checked, we can match up the slice dimensions one for one and run the algorithm above. As an example, consider the assignment statement:

$$\texttt{A}(i_1, \texttt{slice1}, \texttt{slice2}, j_1) = \mathcal{F}(\texttt{B}(i_2, j_2, \texttt{slice3}, \texttt{slice4})).$$

Figure 14 shows the pseudo-code that processor $\mathcal{S}$ executes to determine what portion of B to send to processor $\mathcal{D}$. Note that in the pseudo-code given, we have neglected to perform the mapping to local memory, but the true compiled code must do the mapping.

Processor $\mathcal{D}$ executes a similar loop nest to determine what array elements are sent by processor $\mathcal{S}$. In addition, processor $\mathcal{D}$ executes a similar loop for the computation phase.

```
/* Determine send for A(i1, slice1, slice2, j1) = F(B(i2, j2, slice3, slice4)) */
    If (i1 ∈ Own_D(A, 1) and j1 ∈ Own_D(A, 4)) then
        If (i2 ∈ Own_S(B, 1) and j2 ∈ Own_S(B, 2)) then
            For each t1 ∈ Own_S(B, 3) ∩ Map(Own_D(A, 2) ∩ slice1) do
                For each t2 ∈ Own_S(B, 4) ∩ Map(Own_D(A, 3) ∩ slice2) do
                    Add array element B(i2, j2, t1, t2) to the list of elements to send
```

Figure 14: Pseudo-code to determine the send for a multidimensional array assignment.

10.3 Negative strides

In an assignment statement, it is possible for either or both of the stride components of the slices to be negative. Negative strides present problems for a number of reasons.

Consider a slice $(f : l : s)$, and suppose that s is negative. While f is still a representative in the modified slice representation, it is no longer the case that f is a lower bound and l is an upper bound. Instead, the roles of f and l are reversed when s is negative. In this case, $(f : l : s)$ is equivalent to $(l : f : -s : f)$ in the modified slice representation presented in Section 5.

Another problem with negative strides lies with Euclid's GCD algorithm, which we use to intersect slices. This algorithm requires both input strides to be nonnegative.

Perhaps the most difficult problem to deal with, however, is with respect to alignment. Recall that one of the goals of the send and receive loops is for the right-hand side array to be temporarily realigned with the left-hand side array, so that the same loop index can be used to access both arrays. When the signs of the two strides differ, the right-hand side array must be somehow "reversed" during either the send or the receive phase. This reversal can be accomplished by changing the sign of the stride, and reversing the roles of the upper and lower bounds.

We address the reversal problem by reversing the order within the send algorithm when s_B is negative, and reversing the order within the receive algorithm when s_A is negative. However, there are more changes in the basic algorithm which must be made. In the send algorithm, recall from equation (13) that the lower bound c is defined as $\max\{Map(f_A), Map(f_\mathcal{D}), f_\mathcal{S}\}$. A more general form for the lower bound is:

$$\max\{\min(Map(f_A), Map(l_A)), \min(Map(f_\mathcal{D}), Map(l_\mathcal{D})), \min(f_\mathcal{S}, l_\mathcal{S})\}.$$

A more general form for the upper bound defined in equation (15) is:

$$\min\{\max(Map(f_A), Map(l_A)), \max(Map(f_\mathcal{D}), Map(l_\mathcal{D})), \max(f_\mathcal{S}, l_\mathcal{S})\}.$$

The lower bound for the receive algorithm can be defined as:

$$\max\{\min(f_A, l_A), \min(f_\mathcal{D}, l_\mathcal{D}), \min(Map^{-1}(f_\mathcal{S}), Map^{-1}(l_\mathcal{S}))\},$$

and the upper bound for the receive algorithm can be defined as:

$$\min\{\max(f_A, l_A), \max(f_\mathcal{D}, l_\mathcal{D}), \max(Map^{-1}(f_\mathcal{S}), Map^{-1}(l_\mathcal{S}))\}.$$

Note that when the resulting stride is negative, c and `last` are swapped: we use the upper bound for c and the lower bound for `last`.

In the definitions of the algorithms GLOBAL–INDEX–SEND, LOCAL–INDEX–SEND, GLOBAL–INDEX–RECEIVE, LOCAL–INDEX–RECEIVE, and COMPUTE–LOOP, we have assumed that both strides s_A and s_B are positive. For the send and receive algorithms, we must expand the algorithm to consider four cases, depending on the signs of s_A and s_B. The COMPUTE–LOOP algorithm must now have two cases, depending on the sign of s_A. These revised algorithms are given in Appendix A.

10.4 Partially replicated arrays

In a relaxation or convolution operation, each element of an array is set to be some function of itself and its neighboring elements. A natural way to parallelize this operation is have each processor compute the set of array elements corresponding to the section of the array it owns. However, there must be communication at the block boundaries, since at the block boundary some of the neighbors of the boundary element will reside on a different processor. One possibility for dealing with this communication is to temporarily redistribute the array into a form where copies of boundary elements reside on multiple processors. After the redistribution takes place, every processor can compute its results locally without any additional communication.

When redistributing the original array `A` into a temporary array `A'`, we give `A'` a distribution in which ownership is overlapped at block boundaries. The size of the overlap depends upon the distance of each element from the neighbors it needs to access. Each array element can potentially have multiple owners.

Allowing such partially replicated arrays will complicate the algorithms presented above only slightly. In the general case, no changes need be made to the algorithms given. However, in Section 10.1, we state that if the right-hand side term is a single array reference, such as $\texttt{B}(i)$, then the owner broadcasts the value. If this broadcasting is to be done in the case of partially replicated arrays, then each processor must be prepared to receive multiple broadcasts (depending on how many processors own that array element), or else it must be the case that no two owners send the array element to the same processor.

10.5 Nested array references

A nested array reference takes the form $\texttt{A}(\texttt{Q})$, where `A` is a distributed array and `Q` is an array slice or an array element. In general, we know nothing at compile time concerning the evaluation of `Q`. Therefore, we have no idea which processor or processors own $\texttt{A}(\texttt{Q})$. Thus we must broadcast `Q` to all processors, and then proceed with the owner-computes rule. This compilation method could, of course, be quite expensive, both in terms of execution time and memory usage.

We achieve efficiency in ordinary array statements by depending on the fact that the sequence of array indices can be specified as a slice expression. This assumption breaks down in the case of nested array references. An efficient solution to nested array references is beyond the scope of this paper.

11 Generating array ownership descriptors

In Section 9, the `getfirst` function was referenced. This function returns the first index of a particular array that is owned by a particular processor. (Actually, the function should also take a third argument, the array dimension of interest.) This section describes how this function can be implemented, in terms of the user-input alignment and distribution directives. In general, this function will be expensive to compute, so for each array, it may be beneficial to precompute the function value for all possible inputs (the inputs consist of $\langle$processor, dimension number$\rangle$ pairs) and store the results in a table. The size of this table will be equal to the number of processors in the system times the number of dimensions in the array.

In addition to describing the implementation of the `getfirst` function, this section also describes how to determine the block size and stride for each dimension of the array, which are the $n_{\mathcal{D}}$ and $s_{\mathcal{D}}$ parameters, respectively, used in equation (3), and the $n_{\mathcal{S}}$ and $s_{\mathcal{S}}$ parameters used in equation (4).

If the system contains a large number of processors, memory and time constraints may make it impractical or impossible to precompute all values of the `getfirst` function. In this case, the function may have to perform the computation every time it is called. However, for the remainder of this section, we will assume that a table of precomputed results is being built.

If we allow dynamic redistribution of arrays, we will need some runtime ownership descriptor for each array which is used to hold the parameters required by equations (3) and (4). In addition to the table of `getfirst` results, the runtime descriptor should also contain the block size, stride, and dimension size for each array dimension.

We assume that a subset of the Fortran D and HPF distribution syntax is being used. That is, each array is ALIGNed with a TEMPLATE, which is DISTRIBUTEd. When the distribution type is specified for a template dimension, we require that the number of processors over which to distribute the dimension is given. For example, BLOCK(n) or CYCLIC(n) specifies distributing the dimension over n processors, and BLOCK−CYCLIC(b, n) specifies distributing the dimension over n processors with a block size of b. We make the following restrictions on Fortran D's general alignment and distribution syntax:

- Arbitrary alignment functions are not allowed. Only alignment with a constant offset and a stride of 0 or 1 is allowed. For example, if A is an array and T is a template, A(i) could be aligned with T(i), T($i+1$), T($i-3$), or T(5), but not T($3i$) or T($4-i$).

  The purpose of this restriction is to force array ownership sets to take the form of equations (3) and (4). With only block-cyclic distributions allowed, the ownership of the *template* elements can always be described by these equations, but with arbitrary alignment functions, the ownership of the *array* elements is not necessary describable by these equations. With these restricted alignment functions, the ownership of the array elements can always be described by a block-cyclic distribution.

- Every dimension of a template must be distributed as BLOCK, CYCLIC, or BLOCK−CYCLIC. Note that block and cyclic are special cases of block-cyclic. Also note that non-distribution of an array or template dimension is allowed, since it is equivalent to distributing over one processor in the dimension.

In Section 10.4, we describe overlapped array ownership. Specifying such an overlap can be done with a simple extension to the ALIGN statement. Rather than using a construct such as "ALIGN A(i) WITH T(i)," we might use "ALIGN A($i-1 : i+1$) WITH T(i)," to specify that any processor that owns template element T(i) also owns array elements A($i-1 : i+1$). We will refer to the *left overlap* and *right overlap*, which in this case are -1 and 1, respectively.

The process of converting an ALIGN statement and a DISTRIBUTE statement into an ownership description is quite complex. Some of the subtleties involved are the following:

ALIGN A(i,j) WITH T(i). In this case, the second dimension of A is not aligned with any dimension of T. This alignment statement is equivalent to adding a dummy dimension to T, aligning the second dimension of A with the dummy dimension of T, and distributing the dummy dimension over one processor.

ALIGN A(i) WITH T($i,3$). Here is an example of alignment with a constant index. If the second dimension of T is distributed, there may be some processors that do not own any part of A.

ALIGN A(i,j), B(j,i) WITH T(i,j). Here we have an alignment of B involving index permutation. When computing the ownership sets of A and B, it will be important to ensure that A(i,j) and B(j,i) are owned by the same processor for all legal values of i and j.

When we convert the distribution syntax into an array ownership descriptor, the minimum requirement is the following. Suppose we have two arrays, A and B, a template T, and the statements:

```
ALIGN A(i) WITH T(i)
ALIGN B(i) WITH T(i)
```

We must ensure that A(i) and B(i) are owned by the same processor, for all legal values of i. However, suppose that instead, B is aligned with template U, which is declared and distributed identically to T. Then there is no explicit requirement that A(i) and B(i) are owned by the same processor.

Our approach, however, is to define an algorithm which will consistently produce the same ownership descriptor for a given alignment, distribution, and template definition, regardless of the name of the template. In the above example, A(i) and B(i) will be owned by the same processor regardless of whether B is aligned with T or U, provided T and U are declared and distributed identically. Our algorithm takes as input an ALIGN statement, its corresponding DISTRIBUTE statement, the TEMPLATE definition, and a processor number q. Each processor has a unique value of q between 0 and $N-1$, inclusive, where N is the number of processors in the system. The algorithm returns a vector $\vec{f}$, which contains the index corresponding to the first element owned by processor q in each corresponding dimension of the array. The algorithm can also return a "null vector", which signifies that the processor does not own any part of the array. We will assume that the compiler has already performed all necessary error checking.

The first step in the algorithm is to convert the TEMPLATE, ALIGN, and DISTRIBUTE statements into a normal form which includes no index permutation, where each dimension of the array is aligned with a dimension of the template, and where each dimension of the template is distributed. This conversion is done in the following manner (an example will follow):

1. Determine the number of processors assigned to each dimension of the template. If a dimension is specified not to be distributed (i.e., a "$*$" appears in that dimension in the DISTRIBUTE statement), change its distribution to be BLOCK-CYCLIC(n, 1), for any desired block size n. The actual block size chosen is irrelevant; the important issue is that the dimension is distributed over only one processor. Let vector $\vec{v}$ contain the quantities of processors assigned to the dimensions of the template (i.e., there are $\vec{v}_i$ processors assigned to dimension i of the template). If $\prod_j \vec{v}_j > N$, where N is the number of processors available to execute the assignment statement, then signal an error. If $q \geq \prod_j \vec{v}_j$, where q is the processor number, then return the null vector, since the array is only distributed across the first $\prod_j \vec{v}_j$ processors.

2. From $\vec{v}$ and the processor number q, compute the vector $\vec{p}$, by setting $\vec{p}_i = \lfloor q / \prod_{j<i} \vec{v}_j \rfloor \bmod \vec{v}_i$ for each value of i. For any two distinct values of q less than $\prod_j \vec{v}_j$, the corresponding $\vec{p}$ vectors will differ in at least one component.

   This $\vec{p}$ vector just assigns each processor to a unique nonnegative integer point in the k-dimensional space bounded by vector $\vec{v}$, where k is the length of vectors $\vec{p}$ and $\vec{v}$.

3. Consider all constant, "$*$", and slice indices in the template expression of the ALIGN statement. If the given processor does not own that part of the template (see below), return the null vector. Otherwise, remove these types of indices from the template expression of the ALIGN statement. Also remove the corresponding dimensions from the TEMPLATE statement, the DISTRIBUTE statement, $\vec{v}$, and $\vec{p}$.

   The processor determines whether it owns this part of the template by considering the bounds of that dimension of the template and the block size n_i. In a cyclic distribution, the block size is 1, and in a block distribution, the block size is $\lceil S_i / \vec{v}_i \rceil$, where S_i is the number of elements in that dimension, i, of the template. In a block-cyclic distribution, the block size will be given. Assume that in this

dimension, i, of the template, indices range from c_i to d_i; in Fortran, c_i is usually 1, and in C, c_i is 0. There are three possibilities to consider:

- If a constant is specified (e.g., `ALIGN A(`i`) WITH T(3)`), and $(k_i - c_i - \vec{p}_i n_i) \bmod n_i \vec{v}_i < n_i$, where k_i is the constant index ($k_i = 3$ in the example), then the processor owns the template element.
- If a "$*$" is specified (e.g., `ALIGN A(`i`) WITH T(`$*$`)`), and $q < \prod_j \vec{v}_j$, then the processor owns the section of the template.
- If a slice is specified (e.g., `ALIGN A(`i`) WITH T(1 : 10 : 2)`), and the intersection of that slice and $\bigcup_{j=0}^{n_i-1}(c_i + \vec{p}_i n_i + j : d_i : n_i \vec{v}_i)$ is nonempty, then the processor owns the template section.

4. If there is a symbolic index (such as i or j) in the array expression of the `ALIGN` statement that does not appear in the template expression, change that index to a "$*$" in the array. The goal of this step is to identify unaligned dimensions of the array. For example, `ALIGN A(`i,j`) WITH T(`i`)` would change to `ALIGN A(`$i,*$`) WITH T(`i`)`.

5. For each "$*$" in the array expression of the `ALIGN` statement, replace it with a new dummy variable. Add a corresponding 1 to the end of $\vec{v}$ and a 0 to the end of $\vec{p}$. Modify the `DISTRIBUTE` statement so that each of these dummy dimensions is distributed `BLOCK-CYCLIC(`n`, 1)` for any desired value of n. Add a dimension to the template with the same index bounds as the corresponding dimension of the array.

6. Permute the indices in the template expression of the `ALIGN` statement so that they match up with the indices in the array expression. Apply the same permutation to $\vec{v}$, $\vec{p}$, the `DISTRIBUTE` statement, and the dimension sizes of the template. For example, `ALIGN A(`i,j`) WITH T(`$j+1, i-2$`)` would change to `ALIGN A(`i,j`) WITH T(`$i-2, j+1$`)`, and the same permutation would be applied to $\vec{v}$, $\vec{p}$, the `DISTRIBUTE` statement, and the dimension sizes of the template.

Vector $\vec{v}$ now specifies the number of processors assigned to each corresponding dimension of the array, and vector $\vec{p}$ specifies the processor's order in each dimension of the array.

As an example of the steps performed so far, suppose we have the following statements, and perform the above steps:

```
ALIGN A(i, j-1 : j+2, k, l) WITH T(j, 3, k, i+2)
DISTRIBUTE T(BLOCK(2),CYCLIC(3),BLOCK(4),CYCLIC(5))
```

1. The `DISTRIBUTE` statement yields $\vec{v} = (2, 3, 4, 5)$. Note that we must have at least $2 \times 3 \times 4 \times 5 = 120$ processors executing this code section.

2. Suppose that our processor is number 119. Then $\vec{p} = (1, 2, 3, 4)$.

3. There is one constant index in the template expression of the `ALIGN` statement: the 3. Assume that this processor owns that part of the template, so we continue. Removing that dimension everywhere gives us:

```
ALIGN A(i, j-1 : j+2, k, l) WITH T(j, k, i+2)
DISTRIBUTE T(BLOCK(2),BLOCK(4),CYCLIC(5))
v = (2,4,5), p = (1,3,4)
```

4. The l appears as an index of the array expression in the ALIGN statement, but it does not appear in the template expression, so we change it to a $*$:

ALIGN A$(i, j-1:j+2, k, *)$ WITH T$(j, k, i+2)$ DISTRIBUTE T(BLOCK(2),BLOCK(4),CYCLIC(5)) $\vec{v} = (2,4,5)$, $\vec{p} = (1,3,4)$

5. There is one $*$ in the array expression of the ALIGN statement; replace it by a dummy d, resulting in:

ALIGN A$(i, j-1:j+2, k, d)$ WITH T$(j, k, i+2, d)$ DISTRIBUTE T(BLOCK(2),BLOCK(4),CYCLIC(5),BLOCK-CYCLIC$(1,1)$) $\vec{v} = (2,4,5,1)$, $\vec{p} = (1,3,4,0)$

6. Applying the permutation gives us:

ALIGN A$(i, j-1:j+2, k, d)$ WITH T$(i+2, j, k, d)$ DISTRIBUTE T(CYCLIC(5),BLOCK(2),BLOCK(4),BLOCK-CYCLIC$(1,1)$) $\vec{v} = (5,2,4,1)$, $\vec{p} = (4,1,3,0)$

At this point, we have converted the specifications to a normal form as described previously. We can now use the information given by the index bounds and the alignment offsets to generate the result vector.

Suppose that, after applying the above steps, we have the following declarations:

DIMENSION A$(a_1 : z_1, a_2 : z_2, \ldots)$ TEMPLATE T$(c_1 : d_1, c_2 : d_2, \ldots)$ ALIGN A$(i_1 + o_1^L : i_1 + o_1^R, i_2 + o_2^L : i_2 + o_2^R, \ldots)$ WITH T$(i_1 + b_1, i_2 + b_2, \ldots)$ DISTRIBUTE T$(\ldots)$

At this point, we can easily determine which elements of the *template* that processor q owns, but these template elements must be translated into elements of the *array*. In dimension j of the template, the processor owns the elements of the template corresponding to the indices:

$$\bigcup_{i=0}^{n'_j - 1} (f'_j + i : d_j : s_j).$$

The parameter n'_j is the block size in dimension j of the template, which can be found in the DISTRIBUTE statement; $f'_j = c_j + \vec{p}_j n'_j$; and $s_j = n'_j \vec{v}_j$.

When we apply the alignment function, we find that the processor owns the elements of the array corresponding to the indices:

$$\bigcup_{i=0}^{n_j - 1} (\vec{f_j} + i : d_j : s_j)$$

$$\vec{f_j} = c_j - b_j + o_j^L + \vec{p}_j n'_j, \tag{17}$$

where $n_j = n'_j + o_j^R - o_j^L$. This $\vec{f}$ vector is returned.

As mentioned previously, we need a separate function to determine the block size and stride for a dimension k of an array, given the `TEMPLATE`, `ALIGN`, and `DISTRIBUTE` statements. The first step of this algorithm is to determine which dimension of the template with which dimension k of the array is aligned by considering the `ALIGN` statement. If the array dimension is not aligned with any template dimension, then that array dimension is not distributed, and thus each processor owns every element in that dimension. In this case, we can return any values for the stride and block size, as long as they are equal.

If the array dimension is aligned with a template dimension, consider the distribution of that template dimension in the `DISTRIBUTE` statement. The `DISTRIBUTE` statement gives the number of processors v over which the dimension is distributed, and it gives some indication of the template block size. If it is a block-cyclic distribution, the template block size is given explicitly. If it is a cyclic distribution, the template block size is implicitly 1. If it is a block distribution, the template block size is $\lceil S/v \rceil$, where S is the number of elements in that dimension of the template. The true block size is the sum of the template block size and the total overlap in the corresponding array dimension. The stride is simply the product of the template block size and v. We return the stride and the block size.

12 Optimizations

The algorithms LOCAL–INDEX–SEND, LOCAL–INDEX–RECEIVE, and COMPUTE–LOOP are designed to work in the worst case, when none of the parameters is known at compile time. However, it is almost never the case that all parameters are unknown at compile time. For example, quite frequently the strides in the assignment statement will be 1; and if they are not 1, they are still likely to be compile-time constants. Knowing both strides at compile time can provide some of the best optimization potential, because Euclid's GCD algorithm can be executed at compile time rather than at runtime.

When parameters are known at compile time, potential optimizations can range from simple constant folding to tightening loop bounds or eliminating certain loops altogether.

12.1 Reducing potential communication

Recall that in the general algorithm presented in Section 9, the outer loops in the send and receive phases have each processor considering all other processors for communication. However, one of the user's goals (or the goal of a separate compiler phase) should be to carefully align and distribute the arrays to minimize communication. When the compiler detects "good" alignments and distributions at compile time, it can achieve further runtime efficiency by generating code that restricts the set of processors to consider at runtime for communication.

If the alignment and distribution are specified automatically by a separate compiler phase, that phase might already have determined what communication must occur for a given array statement. Otherwise, we can look for the common communication patterns described below.

It is important to note below that these tests must be applied separately to each dimension of each array. If, for example, we determine that communication occurs with at most two processors in each of the two array dimensions, then overall communication will be with at most four processors. We can only be certain that communication occurs with at most one processor if we can determine independently for each array dimension that communication occurs with at most one processor.

12.1.1 Communication with a small number of processors

In some cases, regardless of the upper and lower bounds of the slices, we can determine at compile time that each processor sends to at most k other processors, and correspondingly receives from at most k other

processors, where k is some small integer. This will happen when each array is distributed over the same number of processors, and the ratio of the strides in the assignment statement is equal to the ratio of the strides of the distributions. As an example of the magnitude of k, when these conditions are met, and the alignment function contains no overlap, the value of k is at most 2.

More formally, we can determine that we send to at most two processors and receive from at most two processors when:

$$s_A s_{\mathcal{S}} = s_B s_{\mathcal{D}} \qquad \text{and} \qquad v_A = v_B, \tag{18}$$

where v_A and v_B are the numbers of processors assigned to the respective array dimensions.

For this analysis, we introduce the concept of a *virtual processor*, for the following reasons. Suppose we have an infinite number of virtual processors, and block-cyclic distributions. Then we can assign each virtual processor one contiguous block of array elements. If $s_A s_{\mathcal{S}} = s_B s_{\mathcal{D}}$, then each virtual processor will send to one or two virtual processors which are a fixed (possibly negative) virtual communication distance to the right, and will correspondingly receive from a fixed virtual distance to the left. Due to the nature of the block-cyclic distribution, a virtual processor is mapped to a real processor by taking the virtual processor number modulo the number of real processors. When the virtual communication distance is fixed, it translates to a fixed real communication distance only when the corresponding array dimensions are distributed over the same number of real processors (i.e., $v_A = v_B$).

Let c_A, b_A, and o_A^L be the c, r, and o^L parameters from equation (17) corresponding to the left-hand side array of the assignment statement, and let c_B, b_B, and o_B^L be the similar parameters for the right-hand side array. We want to find $p_A - p_B$, where p_A is the virtual processor that owns $\mathtt{A}(f_A + ks_A)$ and p_B is the virtual processor that owns $\mathtt{B}(f_B + ks_B)$, for any value of k. From equation (17), we know that a particular processor p_A owns the elements corresponding to indices:

$$c_A - b_A + p_A s_{\mathcal{D}}/v_A + o_A^L : c_A - b_A + p_A s_{\mathcal{D}}/v_A + o_A^L + n_{\mathcal{D}} - 1$$

and that a particular processor p_B owns the elements corresponding to indices:

$$c_B - b_B + p_B s_{\mathcal{S}}/v_B + o_B^L : c_B - b_B + p_B s_{\mathcal{S}}/v_B + o_B^L + n_{\mathcal{S}} - 1.$$

Given a particular value of k, we would like to find bounds on p_A and p_B, where p_A is the processor that owns array element $f_A + ks_A$ and p_B is the processor that owns array element $f_B + ks_B$. We know that:

$$c_A - b_A + p_A s_{\mathcal{D}}/v_A + o_A^L \le f_A + ks_A \le c_A - b_A + p_A s_{\mathcal{D}}/v_A + o_A^L + n_{\mathcal{D}} - 1$$
$$c_B - b_B + p_B s_{\mathcal{S}}/v_B + o_B^L \le f_B + ks_B \le c_B - b_B + p_B s_{\mathcal{S}}/v_B + o_B^L + n_{\mathcal{S}} - 1.$$

These equations yield the following relations:

$$\frac{f_A + ks_A - c_A + b_A - o_A^L - n_{\mathcal{D}} + 1}{s_{\mathcal{D}}/v_A} \le p_A \le \frac{f_A + ks_A - c_A + b_A - o_A^L}{s_{\mathcal{D}}/v_A}$$
$$\frac{f_B + ks_B - c_B + b_B - o_B^L - n_{\mathcal{S}} + 1}{s_{\mathcal{S}}/v_B} \le p_B \le \frac{f_B + ks_B - c_B + b_B - o_B^L}{s_{\mathcal{S}}/v_B}$$

These equations allow us to find an upper bound on $p_A - p_B$, which is less than or equal to $\lfloor p_A - p_B \rfloor$ since p_A and p_B are integers:

$$\begin{aligned} p_A - p_B &\le \left\lfloor \frac{f_A + ks_A - c_A + b_A - o_A^L}{s_{\mathcal{D}}/v_A} - \left(\frac{f_B + ks_B - c_B + b_B - o_B^L - n_{\mathcal{S}} + 1}{s_{\mathcal{S}}/v_B} \right) \right\rfloor \\ &= \left\lfloor v_A \left(\frac{f_A - c_A + b_A + o_A^L}{s_{\mathcal{D}}} - \frac{f_B - c_B + b_B - o_B^L - n_{\mathcal{S}} + 1}{s_{\mathcal{S}}} \right) \right\rfloor \end{aligned}$$

Using similar analysis, we find that a lower bound on $p_A - p_B$ is:

$$p_A - p_B \geq \left\lceil v_A \left(\frac{f_A - c_A + b_A + o_A^L - n_{\mathcal{D}} + 1}{s_{\mathcal{D}}} - \frac{f_B - c_B + b_B - o_B^L}{s_{\mathcal{S}}} \right) \right\rceil .$$

Let D_1 be the lower bound on $p_A - p_B$, and let D_2 be the upper bound on $p_A - p_B$. Any processor p will only send to virtual processors $p + D_1$ through $p + D_2$, inclusive. Similarly, any processor q will only receive from virtual processors $q - D_2$ through $q - D_1$, inclusive. Virtual processors are transformed into real processors by taking the virtual processor modulo the number of real processors in that array dimension.

12.1.2 Small number of senders or receivers

We can determine that a small number of the processors will perform a send if one of the following conditions holds:

- $n'_{\mathcal{S}} = s_{\mathcal{S}}$. This property just means that the array dimension is distributed across only one processor. This information is not particularly useful.
- s_B is a multiple of $s_{\mathcal{S}}$. This property means that every array element accessed on the right-hand side is guaranteed to fall on one of a small set of processors (the size of the set is 1 if $o_{\mathcal{S}}^L = o_{\mathcal{S}}^R$), regardless of how many elements are accessed. Bounds on the sending virtual processor, $p_{\mathcal{S}}$, can be determined by solving the inequality:

  $$c_{\mathcal{S}} - b_{\mathcal{S}} + p_{\mathcal{S}} s_{\mathcal{S}} / v_{\mathcal{S}} + o_{\mathcal{S}}^L \leq f_B \leq c_{\mathcal{S}} - b_{\mathcal{S}} + p_{\mathcal{S}} s_{\mathcal{S}} / v_{\mathcal{S}} + o_{\mathcal{S}}^L + n_{\mathcal{S}} - 1$$

  which yields:

  $$\left\lceil \frac{b_{\mathcal{S}} - c_{\mathcal{S}} - o_{\mathcal{S}}^L - n_{\mathcal{S}} + 1 + f_B}{s_{\mathcal{S}} / v_{\mathcal{S}}} \right\rceil \leq p_{\mathcal{S}} \leq \left\lfloor \frac{b_{\mathcal{S}} - c_{\mathcal{S}} - o_{\mathcal{S}}^L + f_B}{s_{\mathcal{S}} / v_{\mathcal{S}}} \right\rfloor \tag{19}$$

- $\lfloor (f_B - c_{\mathcal{S}} + b_{\mathcal{S}}) / n_{\mathcal{S}} \rfloor = \lfloor (l_B - c_{\mathcal{S}} + b_{\mathcal{S}}) / n_{\mathcal{S}} \rfloor$ and $o_{\mathcal{S}}^L = o_{\mathcal{S}}^R$. This property means that f_B and l_B fall within the same block and are thus guaranteed to reside on the same processor, as are all elements that fall between f_B and l_B. The virtual processor $p_{\mathcal{S}}$ is given by equation (19).

Similarly, we can determine that only one of the processors will perform a receive if one of the following conditions holds:

- $n_{\mathcal{D}} = s_{\mathcal{D}}$. This property just means that the array dimension is distributed across only one processor. This information is not particularly useful.
- s_A is a multiple of $s_{\mathcal{D}}$. As above, this property means that every array element accessed on the left-hand side is guaranteed to fall on one of a small set of processors (the size of the set is 1 if $o_{\mathcal{D}}^L = o_{\mathcal{D}}^R$), regardless of how many elements are accessed. The single receiving virtual processor, $p_{\mathcal{D}}$, can be determined by solving the inequality:

  $$c_{\mathcal{D}} - b_{\mathcal{D}} + p_{\mathcal{D}} s_{\mathcal{D}} / v_{\mathcal{D}} + o_{\mathcal{D}}^L \leq f_A \leq c_{\mathcal{D}} - b_{\mathcal{D}} + p_{\mathcal{D}} s_{\mathcal{D}} / v_{\mathcal{D}} + o_{\mathcal{D}}^L + n_{\mathcal{D}} - 1$$

  which yields:

  $$\left\lceil \frac{b_{\mathcal{D}} - c_{\mathcal{D}} - o_{\mathcal{D}}^L - n_{\mathcal{D}} + 1 + f_A}{s_{\mathcal{D}} / v_{\mathcal{D}}} \right\rceil \leq p_{\mathcal{D}} \leq \left\lfloor \frac{b_{\mathcal{D}} - c_{\mathcal{D}} - o_{\mathcal{D}}^L + f_A}{s_{\mathcal{D}} / v_{\mathcal{D}}} \right\rfloor \tag{20}$$

- $\lfloor (f_A - c_{\mathcal{D}} + b_{\mathcal{D}})/n_{\mathcal{D}} \rfloor = \lfloor (l_A - c_{\mathcal{D}} + b_{\mathcal{D}})/n_{\mathcal{D}} \rfloor$ and $o^L_{\mathcal{D}} = o^R_{\mathcal{D}}$. This property means that f_A and l_A fall within the same block and are thus guaranteed to reside on the same processor, as are all elements that fall between f_A and l_A. The virtual processor $p_{\mathcal{D}}$ is given by equation (20).

The principal benefit of this single sender/receiver test is that if it applies, we can wrap the send phase or the receive phase inside a conditional which tests the processor number. In this way, the entire send phase or receive phase can be eliminated from consideration for some processors.

12.2 Eliminating loops

If the send, receive, and computation loops are implemented exactly as described in Figures 9, 11, and 12, it is possible that the costs of computing the loop bounds will dominate the execution time of the program. One obvious solution to this problem is for the compiler to perform a kind of constant folding by precomputing such parameters as strides and the results of Euclid's algorithm, at compile time. In general, it will be difficult to precompute the loop bounds, since they depend on $f_{\mathcal{D}}$ and $f_{\mathcal{S}}$, which can have different values for different processors.

At compile time, we are likely to know whether the distribution of any given array is block, cyclic, or block-cyclic. If either the left-hand side array or the right-hand side array is block or cyclic, then we need only one outer loop in the LOCAL–INDEX–SEND and LOCAL–INDEX–RECEIVE loops, rather than two. If neither array is block-cyclic, we can eliminate the i and j loops altogether. If the left-hand side array is not block-cyclic, we can eliminate the j loop in the COMPUTE–LOOP algorithm.

These outer loops are due entirely to the block-cyclic distribution. A block-cyclic set is the union of disjoint slices, and characterizing the union requires a loop. In the LOCAL–INDEX–SEND and LOCAL–INDEX–RECEIVE algorithms, we need two extra loops, since equation (6) involves two block-cyclic distributions in the intersection.

Both block and cyclic distributions can be specified as a single slice. When f is the index corresponding to the first array element that a processor owns, l is the largest array index, and the dimension is distributed across v processors, a cyclic distribution can be described as:

$$(f : l : v),$$

and a block distribution can be described as:

$$\left(f : f + \left\lceil \frac{S}{v} \right\rceil - 1 : 1\right),$$

where S is the number of elements in the template dimension that the array dimension is aligned with.

This observation gives more concise ways of defining $Own_{\mathcal{D}}(\mathtt{A})$ and $Own_{\mathcal{S}}(\mathtt{B})$ than in equations (3) and (4). With these new definitions, we can use the methods presented in Section 6 to derive new LOCAL–INDEX–SEND, LOCAL–INDEX–RECEIVE, and COMPUTE–LOOP algorithms. Notice that there are eight new sets of communication algorithms to derive, since each of the two arrays in the assignment statement can have one of three types of distributions, and that there are two new sets of computation algorithms to derive. The new LOCAL–INDEX–SEND and LOCAL–INDEX–RECEIVE algorithms are given in Appendix B, and the new COMPUTE–LOOP algorithms are given in Appendix C.

12.3 Customizing the output

If we are compiling for a MIMD architecture, further optimization can be achieved by producing customized output for each processor. If we know at compile time that only a certain subset of processors will

be executing certain operations, we normally output code that conditionally executes depending on the processor ID. But if we create customized code for each processor, the conditionals can be evaluated at compile time, not at runtime, and some loop bounds can be precomputed. This approach could further speed up execution.

Note, however, that this approach does not scale well as the number of processors in the system increases. It could take quite long just to compile if the number of processors is large. Thus we probably will not want to take this approach unless we truly want the utmost execution speed.

On the other hand, it may be the case that certain subsets of processors usually perform similar operations. In this case, we might want to partition the processors and generate one customized output file for each subset of the partition.

12.4 Merging receive and computation phases

Recall that the overall algorithm temporarily redistributes the right-hand side arrays of the assignment statement to match the distribution of the left-hand side array, and then performs the computation phase. We can improve upon this method if there is only one array slice on the right-hand side, and the memory accessed by the two array slices is disjoint (i.e., there are no data dependences). In this case, we can perform all communication of single array elements on the right-hand side (see Section 10.1), and perform the communication for the right-hand side slice last. However, rather than receiving the slice elements and storing them in a temporary array, we can receive each element, perform the computation, and directly store the result into the left-hand side array, thus allowing us to merge the receive and computation phases.

To illustrate, the sequential Fortran code given in Section 2 would conceptually change to the following after applying this optimization:

$i_B = f_B$
DO $i_A = f_A, l_A, s_A$
  A(i_A) = $\mathcal{F}$(B(i_B))
  $i_B = i_B + s_B$
END DO

13 Further applications of array statements

We have derived a general method for compiling and optimizing general array assignment statements. However, the programmer will often need to have a larger set of array operations at his disposal. In this section we describe simple extensions of array assignment statements that allow more powerful array constructs.

13.1 Redistributing arrays

From time to time, the user (or a separate compiler phase) may wish to change the distribution of an array. For example, when operating on an array, one algorithm might be more efficient when only the rows of the array are distributed, while a subsequent algorithm might demand that the array be distributed by columns.

Such a redistribution can be simulated by an array assignment statement. Suppose we wish to redistribute array A. We can create a declaration for a new array A′, with the same type and dimension bounds as A, but with the desired new alignment and distribution. The assignment statement A′ = A, when compiled in the above manner, will generate the necessary communication and copying. To use the redistributed array,

subsequent references to A should be replaced by references to A′. Provided that the memory space of A is disjoint with the memory space of A′, the optimization described in Section 12.4 always applies.

Notice that the redistribution method described here requires that distributions be statically known at compile time. If distributions are not fully known at compile, the compiler must use only the runtime array ownership descriptors to determine communication. In this case, most of the communication optimizations described above can no longer apply.

13.2 WHERE statement

The WHERE statement is a valuable construct from Fortran 90. It allows conditional data-parallel execution of array statements. The general form is (see the Fortran 90 reference [1] for a more precise definition):

```
WHERE mask-expr
        assn-stmt-1
        assn-stmt-2
ELSEWHERE
        assn-stmt-3
        assn-stmt-4
ENDWHERE
```

The sizes and shapes of the sub-arrays defined by the mask expression and the left-hand sides of the assignment statements must be the same, and the assignment statements must all be array assignment statements. We first create a boolean *mask-array* which has the same distribution as the left-hand side array. The array is initialized with the *mask-expr*, and indexed with the same array slices as the left-hand side of *assn-stmt-1*. Then *assn-stmt-1* is evaluated, subject to the true values in the *mask-array*. Next, the *mask-array* is redistributed (see above) to match *assn-stmt-2*, and the process is repeated for the rest of the assignment statements inside the WHERE section. Within the ELSEWHERE section, the same method is used, except that we use negated values from the boolean *mask-array*.

As an example, consider the following construct:

```
WHERE A(f_A : l_A : s_A) ≠ B(f_B : l_B : s_B)
        C(f_C : l_C : s_C) = D(f_D : l_D : s_D)
        E(f_E : l_E : s_E) = F(f_F : l_F : s_F)
ELSEWHERE
        G(f_G : l_G : s_G) = H(f_H : l_H : s_H)
        I(f_I : l_I : s_I) = J(f_J : l_J : s_J)
ENDWHERE
```

The transformations described above would result in the following code:

$\texttt{M1}(f_C : l_C : s_C) = (\texttt{A}(f_A : l_A : s_A) \neq \texttt{B}(f_B : l_B : s_B))$
$\texttt{C}(f_C : l_C : s_C) = \texttt{D}(f_D : l_D : s_D)$ [subject to M1]
$\texttt{M2}(f_E : l_E : s_E) = \texttt{M1}(f_C : l_C : s_C)$
$\texttt{E}(f_E : l_E : s_E) = \texttt{F}(f_F : l_F : s_F)$ [subject to M2]
$\texttt{M3}(f_G : l_G : s_G) = \texttt{M2}(f_E : l_E : s_E)$
$\texttt{G}(f_G : l_G : s_G) = \texttt{H}(f_H : l_H : s_H)$ [subject to negation of M3]
$\texttt{M4}(f_I : l_I : s_I) = \texttt{M3}(f_G : l_G : s_G)$
$\texttt{I}(f_I : l_I : s_I) = \texttt{J}(f_J : l_J : s_J)$ [subject to negation of M4]

13.3 Reduction operators

An array reduction statement takes the form:

$$\mathtt{x} = Reduce(\mathtt{A}(f_A : l_A : s_A)),$$

where $Reduce$ is a *reduction operator*. A reduction operator is an associative binary operator that reduces an array or array section to a single scalar value. Typical reduction operators are minimum value, maximum value, location of maximum value,[1] sum, and product. Note that these examples of reduction operators are also commutative. The remainder of this section will assume that the reduction operator is commutative.

The reduction can be evaluated efficiently by having each processor perform a local reduction on the specified portion of the array that it owns. This local reduction yields one value per processor, and all processors can then combine their results to yield the global reduction. This final result can be broadcast to all processors, if necessary.

To perform the local reduction, processor $\mathcal{D}$ needs to perform its local reduction on the elements of A given by indices:

$$Own_{\mathcal{D}}(\mathtt{A}) \cap (f_A : l_A : s_A).$$

The calculation of this set is identical to that in Section 8, and the COMPUTE–LOOP algorithm can be used almost unchanged to calculate the local reduction.

13.4 Parallel independent loop iterations

Our experience has shown us that the array constructs described so far are simply not powerful enough in general to produce acceptable performance for many real programs. For example, consider the two-dimensional Fast Fourier Transform (2D FFT) algorithm. This algorithm first performs a 1D FFT operation independently on each row of an $n \times n$ array, and then performs a 1D FFT operation independently on each column. The 1D FFT operation is difficult to parallelize, and array assignment statements are simply not powerful enough to effectively parallelize it.

An obvious source of parallelism for the 2D FFT algorithm is in the outer loop. Since there are no loop-carried dependences, each iteration, which consists of a 1D FFT operation on a row or a column, can proceed in parallel. Furthermore, if each processor owns only entire rows (i.e., the array is distributed only in the first dimension), no communication need occur during the row-wise 1D FFT loop. Similarly, no communication occurs within the column-wise 1D FFT loop when the array is distributed only in the second dimension. These observations motivate the definition of a basic parallel loop construct.

This parallel loop can take the following form:

```
PDO i = f,l,s
INPUT A(i,:)
OUTPUT B(:,i)
   ... statements reading A(i,:) and writing B(:,i)
END PDO
```

The INPUT section lists the distributed array sections that are read in each loop iteration, and the OUTPUT section lists the distributed array sections that are written. The PDO loop index may only appear in one dimension of these array sections. To compile this loop, we first choose an arbitrary one-dimensional

[1] The location of minimum/maximum value reduction operator actually returns a vector, rather than a single scalar value, when the reduced array expression is multidimensional.

template and distribution, and we redistribute the arrays in the input and output sections so that the dimension containing the loop index is now aligned with the new template, using the method of redistribution described above. This redistribution ensures that all data read or written during any one iteration of the loop is owned by one processor, and thus no intra-iteration communication is necessary. Each processor independently iterates through the indices of the template that it owns and that are in the slice $(f : l : s)$. This intersection can be found using equations (7) and (8). At the completion of all the loop iterations, the arrays are redistributed to their original forms.

As an optimization, the compiler can try to choose a new template and distribution that matches as many arrays in the input and output lists as possible. When the distribution and template match that of an array in the input list, it is not necessary to redistribute that array.

14 Conclusion

As High Performance Fortran (HPF) comes of age, compiler writers who wish to implement HPF for a private memory multiprocessor will first need to implement communication and memory management for distributed arrays. We have reduced this problem to finding intersections of index sets characterized by the three-parameter array slice notation, $(f : l : s)$. By treating block-cyclic distribution sets as unions of disjoint slices, we have derived an efficient means of calculating the communication between processors. This calculation includes the mapping of global indices into local memory.

In addition to calculating the communication, we have provided a means of converting `TEMPLATE`, `ALIGN`, and `DISTRIBUTE` statements into array ownership descriptors which are needed at runtime. Finally, we have derived communication optimizations for many common cases. All derivations are presented in detail to allow the compiler writer to gain a better understanding of the final results.

This work has been validated by the implementation of the Fx compiler on the iWarp system, a private memory multiprocessor, and the results, including optimizations for common cases, have been promising.

Acknowledgements

I am extremely grateful to Jonathan Shewchuk and Thomas Gross, each of whom made several passes through this document at various stages of its development. Their insightful comments greatly helped in the clarity of the presentation. I would also like to thank the members of the Fx compiler group, who helped to find bugs and inefficiencies in the concepts and implementations.

References

[1] American National Standards Institute. *Fortran 90*, May 1991. X3J3 internal document S8.118.

[2] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H. T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P. S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing '88*, pages 330–339, Orlando, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[3] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting systolic and memory communication in iWarp. In *Proc. 17th Intl. Symposium on Computer Architecture*, pages 70–81. ACM, May 1990.

[4] S. Chatterjee, J. Gilbert, F. J. E. Long, R. Schreiber, and S.-H. Teng. Generating local addresses and communication sets for data-parallel programs. In *Proc. of PPoPP*, pages 149–158, San Diego, CA, May 1993.

[5] S. Chatterjee, J. Gilbert, R. Schreiber, and S.-H. Teng. Automatic array alignment in data-parallel programs. In *Proc. 20th POPL Conf.*, pages 16–28, Charleston, SC, January 1993. ACM.

[6] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and C.-W. Tseng. Compiling Fortran 77D and 90D for MIMD distributed-memory machines. Technical Report COMP TR92-178, Dept. of Computer Science, Rice University, March 1992.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[8] High Performance Fortran Forum. High Performance Fortran language specification version 1.0, May 1993.

[9] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[10] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. Technical Report Rice COMP TR91-170, Dept. of Computer Science, Rice University, November 1991.

[11] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[12] D. E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1981.

[13] C. Koelbel. *Compiling Programs for Nonshared Memory Machines*. PhD thesis, Purdue University, August 1990.

[14] 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.

[15] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy, and K. Crowley. PARTI primitives for unstructured and block structured problems. *Computing Systems in Engineering*, 3(1–4):73–86, 1992.

[16] Thinking Machines Corporation, Cambridge, Massachusetts. *CM Fortran Programming Guide*, January 1991.

[17] P.-S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1989.

[18] S. Wholey. *Automatic Data Mapping for Distributed-Memory Parallel Computers*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991.

[19] H. Zima, P. Brezany, B. Chapman, P. Mehrota, and A. Schwald. Vienna Fortran – a language specification version 1.1. Technical Report ACPC/TR 92-4, Austrian Center for Parallel Computation, March 1992.

A Extended algorithms for signed strides

In the derivations of the GLOBAL–INDEX–SEND and LOCAL–INDEX–SEND algorithms presented in Section 6, the GLOBAL–INDEX–RECEIVE and LOCAL–INDEX–RECEIVE algorithms in Section 7, and the COMPUTE–LOOP algorithm in Section 8, we assumed that the strides s_A and s_B specified in the assignment statements were positive. In Section 10.3, we described at a high level how to modify the algorithms when one or both strides are negative. This appendix contains the algorithms which account for the signs of the strides in the assignment statement.

A.1 Extended LOCAL–INDEX–SEND algorithm

In this algorithm, note that there are four main parts. In each of the last three parts, lines that contain a difference between the corresponding line in the first part are preceded by an asterisk (*).

```
  IF s_A > 0 AND s_B > 0 THEN
     (x_1, y_1, g_1) = euclid(s_A, s_D)
     (x_2, y_2, g_2) = euclid(s_B s_D / g_1, s_S)
     stride = s_B s_D s_S / (g_1 g_2)
     lmstride = s_B s_D n_S / (g_1 g_2)
     last = min(f_B + s_B ⌊(min(l_A, l_D) − f_A)/s_A⌋, l_S)
     lmlast = ⌊(last − f_S)/s_S⌋ n_S + min((last − f_S) mod s_S, n_S − 1) + K
     c = max(f_B + s_B ⌈(max(f_A, f_D) − f_A)/s_A⌉, f_S)
     DO j = ⌈(f_D − f_A)/g_1⌉ UPTO ⌊(f_D − f_A + n_D − 1)/g_1⌋
        DO i = ⌈(f_S − f_B − j x_1 s_B)/g_2⌉ UPTO ⌊(f_S − f_B − j x_1 s_B + n_S − 1)/g_2⌋
           i' = g_2 i − f_S + f_B + j x_1 s_B
           r = f_B + j x_1 s_B + i x_2 s_B s_D / g_1
           first = c + ((r − c) mod stride)
           lmfirst = (first − f_S − i') n_S / s_S + i' + K
           output-local-slice(lmfirst, lmlast, lmstride)
* ELSE IF s_A > 0 AND s_B < 0 THEN
     (x_1, y_1, g_1) = euclid(s_A, s_D)
*    (x_2, y_2, g_2) = euclid(−s_B s_D / g_1, s_S)
     stride = s_B s_D s_S / (g_1 g_2)
     lmstride = s_B s_D n_S / (g_1 g_2)
*    last = max(f_B + s_B ⌊(min(l_A, l_D) − f_A)/s_A⌋, f_S)
     lmlast = ⌊(last − f_S)/s_S⌋ n_S + min((last − f_S) mod s_S, n_S) + K
*    c = min(f_B + s_B ⌈(max(f_A, f_D) − f_A)/s_A⌉, l_S)
     DO j = ⌈(f_D − f_A)/g_1⌉ UPTO ⌊(f_D − f_A + n_D − 1)/g_1⌋
        DO i = ⌈(f_S − f_B − j x_1 s_B)/g_2⌉ UPTO ⌊(f_S − f_B − j x_1 s_B + n_S − 1)/g_2⌋
           i' = g_2 i − f_S + f_B + j x_1 s_B
           r = f_B + j x_1 s_B + i x_2 s_B s_D / g_1
*          first = c − ((c − r) mod (−stride))
           lmfirst = (first − f_S − i') n_S / s_S + i' + K
           output-local-slice(lmfirst, lmlast, lmstride)
* ELSE IF s_A < 0 AND s_B > 0 THEN
*    (x_1, y_1, g_1) = euclid(−s_A, s_D)
     (x_2, y_2, g_2) = euclid(s_B s_D / g_1, s_S)
```

```
    stride = s_B s_D s_S/(g_1 g_2)
    lmstride = s_B s_D n_S/(g_1 g_2)
*   last = min(f_B + s_B ⌊(max(l_A, f_D) − f_A)/s_A⌋, l_S)
    lmlast = ⌊(last − f_S)/s_S⌋ n_S + min((last − f_S) mod s_S, n_S − 1) + K
*   c = max(f_B + s_B ⌈(min(f_A, l_D) − f_A)/s_A⌉, f_S)
    DO j = ⌈(f_D − f_A)/g_1⌉ UPTO ⌊(f_D − f_A + n_D − 1)/g_1⌋
       DO i = ⌈(f_S − f_B − j x_1 s_B)/g_2⌉ UPTO ⌊(f_S − f_B − j x_1 s_B + n_S − 1)/g_2⌋
          i' = g_2 i − f_S + f_B + j x_1 s_B
          r = f_B + j x_1 s_B + i x_2 s_B s_D/g_1
          first = c + ((r − c) mod stride)
          lmfirst = (first − f_S − i') n_S/s_S + i' + K
          output-local-slice(lmfirst, lmlast, lmstride)
* ELSE IF s_A < 0 AND s_B < 0 THEN
*   (x_1, y_1, g_1) = euclid(−s_A, s_D)
*   (x_2, y_2, g_2) = euclid(−s_B s_D/g_1, s_S)
    stride = s_B s_D s_S/(g_1 g_2)
    lmstride = s_B s_D n_S/(g_1 g_2)
*   last = max(f_B + s_B ⌊(max(l_A, f_D) − f_A)/s_A⌋, f_S)
    lmlast = ⌊(last − f_S)/s_S⌋ n_S + min((last − f_S) mod s_S, n_S) + K
*   c = min(f_B + s_B ⌈(min(f_A, l_D) − f_A)/s_A⌉, l_S)
    DO j = ⌈(f_D − f_A)/g_1⌉ UPTO ⌊(f_D − f_A + n_D − 1)/g_1⌋
       DO i = ⌈(f_S − f_B − j x_1 s_B)/g_2⌉ UPTO ⌊(f_S − f_B − j x_1 s_B + n_S − 1)/g_2⌋
          i' = g_2 i − f_S + f_B + j x_1 s_B
          r = f_B + j x_1 s_B + i x_2 s_B s_D/g_1
*         first = c − ((c − r) mod (−stride))
          lmfirst = (first − f_S − i') n_S/s_S + i' + K
          output-local-slice(lmfirst, lmlast, lmstride)
 ENDIF
```

A.2 Extended LOCAL–INDEX–RECEIVE algorithm

As before, there are four nearly identical parts to this algorithm, and differences are noted by asterisks.

```
 IF s_A > 0 AND s_B > 0 THEN
    (x_1, y_1, g_1) = euclid(s_A, s_D)
    (x_2, y_2, g_2) = euclid(s_B s_D/g_1, s_S)
    stride = s_A s_D s_S/(g_1 g_2)
    lmstride = s_A n_D s_S/(g_1 g_2)
    last = min(l_A, l_D, f_A + s_A ⌊(l_S − f_B)/s_B⌋)
    lmlast = ⌊(last − f_D)/s_D⌋ n_D + min((last − f_D) mod s_D, n_D − 1) + K
    c = max(f_A, f_D, f_A + s_A ⌈(f_S − f_B)/s_B⌉)
    DO j = ⌈(f_D − f_A)/g_1⌉ UPTO ⌊(f_D − f_A + n_D − 1)/g_1⌋
       t = (f_A + j x_1 s_A − f_D) mod s_D
       DO i = ⌈(f_S − f_B − j x_1 s_B)/g_2⌉ UPTO ⌊(f_S − f_B − j x_1 s_B + n_S − 1)/g_2⌋
          r = f_A + j x_1 s_A + i x_2 s_A s_D/g_1
          first = c + ((r − c) mod stride)
          lmfirst = (first − f_D − t) n_D/s_D + t + K
```

```
          input-local-slice(lmfirst, lmlast, lmstride)
* ELSE IF s_A > 0 AND s_B < 0 THEN
     (x_1, y_1, g_1) = euclid(s_A, s_D)
*    (x_2, y_2, g_2) = euclid(−s_B s_D/g_1, s_S)
     stride = s_A s_D s_S/(g_1 g_2)
     lmstride = s_A n_D s_S/(g_1 g_2)
*    last = min(l_A, l_D, f_A + s_A ⌊(f_S − f_B)/s_B⌋)
     lmlast = ⌊(last − f_D)/s_D⌋ n_D + min((last − f_D) mod s_D, n_D − 1) + K
*    c = max(f_A, f_D, f_A + s_A ⌈(l_S − f_B)/s_B⌉)
     DO j = ⌈(f_D − f_A)/g_1⌉ UPTO ⌊(f_D − f_A + n_D − 1)/g_1⌋
        t = (f_A + j x_1 s_A − f_D) mod s_D
        DO i = ⌈(f_S − f_B − j x_1 s_B)/g_2⌉ UPTO ⌊(f_S − f_B − j x_1 s_B + n_S − 1)/g_2⌋
           r = f_A + j x_1 s_A + i x_2 s_A s_D/g_1
           first = c + ((r − c) mod stride)
           lmfirst = (first − f_D − t) n_D/s_D + t + K
           input-local-slice(lmfirst, lmlast, lmstride)
* ELSE IF s_A < 0 AND s_B > 0 THEN
*    (x_1, y_1, g_1) = euclid(−s_A, s_D)
     (x_2, y_2, g_2) = euclid(s_B s_D/g_1, s_S)
     stride = s_A s_D s_S/(g_1 g_2)
     lmstride = s_A n_D s_S/(g_1 g_2)
*    last = max(l_A, f_D, f_A + s_A ⌊(l_S − f_B)/s_B⌋)
     lmlast = ⌊(last − f_D)/s_D⌋ n_D + min((last − f_D) mod s_D, n_D) + K
*    c = min(f_A, l_D, f_A + s_A ⌈(f_S − f_B)/s_B⌉)
     DO j = ⌈(f_D − f_A)/g_1⌉ UPTO ⌊(f_D − f_A + n_D − 1)/g_1⌋
        t = (f_A + j x_1 s_A − f_D) mod s_D
        DO i = ⌈(f_S − f_B − j x_1 s_B)/g_2⌉ UPTO ⌊(f_S − f_B − j x_1 s_B + n_S − 1)/g_2⌋
           r = f_A + j x_1 s_A + i x_2 s_A s_D/g_1
           first = c − ((c − r) mod (−stride))
           lmfirst = (first − f_D − t) n_D/s_D + t + K
           input-local-slice(lmfirst, lmlast, lmstride)
* ELSE IF s_A < 0 AND s_B < 0 THEN
*    (x_1, y_1, g_1) = euclid(−s_A, s_D)
*    (x_2, y_2, g_2) = euclid(−s_B s_D/g_1, s_S)
     stride = s_A s_D s_S/(g_1 g_2)
     lmstride = s_A n_D s_S/(g_1 g_2)
*    last = max(l_A, f_D, f_A + s_A ⌊(f_S − f_B)/s_B⌋)
     lmlast = ⌊(last − f_D)/s_D⌋ n_D + min((last − f_D) mod s_D, n_D) + K
*    c = min(f_A, l_D, f_A + s_A ⌈(l_S − f_B)/s_B⌉)
     DO j = ⌈(f_D − f_A)/g_1⌉ UPTO ⌊(f_D − f_A + n_D − 1)/g_1⌋
        t = (f_A + j x_1 s_A − f_D) mod s_D
        DO i = ⌈(f_S − f_B − j x_1 s_B)/g_2⌉ UPTO ⌊(f_S − f_B − j x_1 s_B + n_S − 1)/g_2⌋
           r = f_A + j x_1 s_A + i x_2 s_A s_D/g_1
*          first = c − ((c − r) mod (−stride))
           lmfirst = (first − f_D − t) n_D/s_D + t + K
           input-local-slice(lmfirst, lmlast, lmstride)
```

```
ENDIF
```

A.3 Extended COMPUTE–LOOP algorithm

In this algorithm, there are two similar cases. Differences are once again marked by an asterisk.

```
IF s_A > 0 THEN
   (x_1, y_1, g_1) = euclid(s_A, s_D)
   stride = s_A s_D / g_1
   lmstride = s_A n_D / g_1
   last = min(l_A, l_D)
   lmlast = ⌊(last − f_D)/s_D⌋ n_D + min((last − f_D) mod s_D, n_D − 1) + K
   c = max(f_A, f_D)
   DO j = ⌈(f_D − f_A)/g_1⌉ UPTO ⌊(f_D − f_A + n_D − 1)/g_1⌋
      j' = g_1 j + f_A − f_D
      r = f_A + j x_1 s_A
      first = c + ((r − c) mod stride)
      lmfirst = (first − f_D − j') n_D / s_D + j' + K
      DO i = lmfirst UPTO lmlast BY lmstride
         A(i) = F(T(i))
* ELSE IF s_A < 0 THEN
*    (x_1, y_1, g_1) = euclid(−s_A, s_D)
*    stride = −s_A s_D / g_1
*    lmstride = −s_A n_D / g_1
*    last = min(f_A, l_D)
*    lmlast = ⌊(last − f_D)/s_D⌋ n_D + min((last − f_D) mod s_D, n_D − 1) + K
*    c = max(l_A, f_D)
     DO j = ⌈(f_D − f_A)/g_1⌉ UPTO ⌊(f_D − f_A + n_D − 1)/g_1⌋
        j' = g_1 j + f_A − f_D
        r = f_A + j x_1 s_A
        first = c + ((r − c) mod stride)
        lmfirst = (first − f_D − j') n_D / s_D + j' + K
        DO i = lmfirst UPTO lmlast BY lmstride
           A(i) = F(T(i))
ENDIF
```

B Optimized communication for block and cyclic distributions

When the distributions of one or both arrays are known to be block or cyclic, rather than block-cyclic, our send and receive algorithms can be optimized. When both distributions are block-cyclic, the send and receive algorithms contain three nested loops: the j loop, the i loop, and the slice loop (see Figure 7 for an example). The j loop is due to the left-hand side array being block-cyclic, and the i loop is due to the right-hand side array being block-cyclic.

When either array is distributed block or cyclic, we no longer need its corresponding outer loop to characterize the intersections. Although some optimizations could be applied by substituting constants into the equations, we will still always have the three nested loops. We can achieve the most efficiency by rederiving the equations with the new definitions of $Own_{\mathcal{D}}(\mathtt{A})$ and $Own_{\mathcal{S}}(\mathtt{B})$.

Here we show the new LOCAL–INDEX–SEND and LOCAL–INDEX–RECEIVE algorithms.

B.1 Case A: block-cyclic, block

Array A is block-cyclic, and B is block. The ownership sets are given by:

$$Own_{\mathcal{D}}(\mathtt{A}) = \bigcup_{j'=0}^{n_{\mathcal{D}}-1} (f_{\mathcal{D}} + j' : l_{\mathcal{D}} : s_{\mathcal{D}})$$

$$Own_{\mathcal{S}}(\mathtt{B}) = (f'_{\mathcal{S}} : f_{\mathcal{S}} + n_{\mathcal{S}} - 1 : 1)$$

The revised LOCAL–INDEX–SEND algorithm is the following:

$(x_1, y_1, g_1) = \mathtt{euclid}(s_A, s_{\mathcal{D}})$
$\mathtt{stride} = s_B s_{\mathcal{D}} / g_1$
$\mathtt{lmstride} = s_B s_{\mathcal{D}} / g_1$
$\mathtt{last} = \min(f_B + s_B \lfloor (\min(l_A, l_{\mathcal{D}}) - f_A)/s_A \rfloor, f_{\mathcal{S}} + n_{\mathcal{S}} - 1)$
$\mathtt{lmlast} = \min(\mathtt{last} - f_{\mathcal{S}}, n_{\mathcal{S}} - 1) + \mathtt{K}$
$c = \max(f_B + s_B \lceil (\max(f_A, f_{\mathcal{D}}) - f_A)/s_A \rceil, f_{\mathcal{S}})$
DO $j = \lceil (f_{\mathcal{D}} - f_A)/g_1 \rceil$ UPTO $\lfloor (f_{\mathcal{D}} - f_A + n_{\mathcal{D}} - 1)/g_1 \rfloor$
  $r = f_B + j x_1 s_B$
  $\mathtt{first} = c + ((r - c) \bmod \mathtt{stride})$
  $\mathtt{lmfirst} = \mathtt{first} - f_{\mathcal{S}} + \mathtt{K}$
  output-local-slice(lmfirst, lmlast, lmstride)

The revised LOCAL–INDEX–RECEIVE algorithm is the following:

$(x_1, y_1, g_1) = \mathtt{euclid}(s_A, s_{\mathcal{D}})$
$\mathtt{stride} = s_A s_{\mathcal{D}} / g_1$
$\mathtt{lmstride} = s_A n_{\mathcal{D}} / g_1$
$\mathtt{last} = \min(l_A, l_{\mathcal{D}}, f_A + s_A \lfloor (f_{\mathcal{S}} + n_{\mathcal{S}} - 1 - f_B)/s_B \rfloor)$
$\mathtt{lmlast} = \lfloor (\mathtt{last} - f_{\mathcal{D}})/s_{\mathcal{D}} \rfloor n_{\mathcal{D}} + \min((\mathtt{last} - f_{\mathcal{D}}) \bmod s_{\mathcal{D}}, n_{\mathcal{D}} - 1) + \mathtt{K}$
$c = \max(f_A, f_{\mathcal{D}}, f_A + s_A \lceil (f_{\mathcal{S}} - f_B)/s_B \rceil)$
DO $j = \lceil (f_{\mathcal{D}} - f_A)/g_1 \rceil$ UPTO $\lfloor (f_{\mathcal{D}} - f_A + n_{\mathcal{D}} - 1)/g_1 \rfloor$
  $t = (f_A + j x_1 s_A - f_{\mathcal{D}}) \bmod s_{\mathcal{D}}$
  $r = f_A + j x_1 s_A$
  $\mathtt{first} = c + ((r - c) \bmod \mathtt{stride})$
  $\mathtt{lmfirst} = (\mathtt{first} - f_{\mathcal{D}} - t) n_{\mathcal{D}} / s_{\mathcal{D}} + t + \mathtt{K}$
  input-local-slice(lmfirst, lmlast, lmstride)

B.2 Case B: block-cyclic, cyclic

Array A is block-cyclic, and B is cyclic. The ownership sets are given by:

$$Own_{\mathcal{D}}(\mathtt{A}) = \bigcup_{j'=0}^{n_{\mathcal{D}}-1} (f_{\mathcal{D}} + j' : l_{\mathcal{D}} : s_{\mathcal{D}})$$

$$Own_{\mathcal{S}}(\mathtt{B}) = (f_{\mathcal{S}} : l_{\mathcal{S}} : s_{\mathcal{S}})$$

The revised LOCAL–INDEX–SEND algorithm is the following:

$(x_1, y_1, g_1) = \texttt{euclid}(s_A, s_\mathcal{D})$ /* x_1 must be nonnegative */
$(x_2, y_2, g_2) = \texttt{euclid}(s_B s_\mathcal{D}/g_1, s_\mathcal{S})$
$(x_3, y_3, g_3) = \texttt{euclid}(x_1 s_B, g_2)$
$\texttt{stride} = s_B s_\mathcal{D} s_\mathcal{S}/(g_1 g_2)$
$\texttt{lmstride} = s_B s_\mathcal{D}/(g_1 g_2)$
$\texttt{last} = \min(f_B + s_B \lfloor (\min(l_A, l_\mathcal{D}) - f_A)/s_A \rfloor, l_\mathcal{S})$
$\texttt{lmlast} = \lfloor (\texttt{last} - f_\mathcal{S})/s_\mathcal{S} \rfloor + \texttt{K}$
$c = \max(f_B + s_B \lceil (\max(f_A, f_\mathcal{D}) - f_A)/s_A \rceil, f_\mathcal{S})$
IF $(f_B - f_\mathcal{S}) \bmod g_3 == 0$ THEN
  $c_j = \lceil (f_\mathcal{D} - f_A)/g_1 \rceil$
  DO $j = c_j + ((f_\mathcal{S} - f_B)x_3/g_3 - c_j) \bmod g_3$ UPTO $\lfloor (f_\mathcal{D} - f_A + n_\mathcal{D} - 1)/g_1 \rfloor$
      BY g_2/g_3
    $r = f_B + j x_1 s_B + (f_\mathcal{S} - f_B - j x_1 s_B) x_2 s_B s_\mathcal{D}/(g_1 g_2)$
    $\texttt{first} = c + ((r - c) \bmod \texttt{stride})$
    $\texttt{lmfirst} = (\texttt{first} - f_\mathcal{S})/s_\mathcal{S} + \texttt{K}$
    output-local-slice(lmfirst, lmlast, lmstride)

The revised LOCAL–INDEX–RECEIVE algorithm is the following:

$(x_1, y_1, g_1) = \texttt{euclid}(s_A, s_\mathcal{D})$ /* x_1 must be nonnegative */
$(x_2, y_2, g_2) = \texttt{euclid}(s_B s_\mathcal{D}/g_1, s_\mathcal{S})$
$(x_3, y_3, g_3) = \texttt{euclid}(x_1 s_B, g_2)$
$\texttt{stride} = s_A s_\mathcal{D} s_\mathcal{S}/(g_1 g_2)$
$\texttt{lmstride} = s_A n_\mathcal{D} s_\mathcal{S}/(g_1 g_2)$
$\texttt{last} = \min(l_A, l_\mathcal{D}, f_A + s_A \lfloor (l_\mathcal{S} - f_B)/s_B \rfloor)$
$\texttt{lmlast} = \lfloor (\texttt{last} - f_\mathcal{D})/s_\mathcal{D} \rfloor n_\mathcal{D} + \min((\texttt{last} - f_\mathcal{D}) \bmod s_\mathcal{D}, n_\mathcal{D} - 1) + \texttt{K}$
$c = \max(f_A, f_\mathcal{D}, f_A + s_A \lceil (f_\mathcal{S} - f_B)/s_B \rceil)$
IF $(f_B - f_\mathcal{S}) \bmod g_3 == 0$ THEN
  $c_j = \lceil (f_\mathcal{D} - f_A)/g_1 \rceil$
  DO $j = c_j + ((f_\mathcal{S} - f_B)x_3/g_3 - c_j) \bmod g_3$ UPTO $\lfloor (f_\mathcal{D} - f_A + n_\mathcal{D} - 1)/g_1 \rfloor$
      BY g_2/g_3
    $r = f_A + j x_1 s_A + (f_\mathcal{S} - f_B - j x_1 s_B) x_2 s_A s_\mathcal{D}/(g_1 g_2)$
    $t = (f_A + j x_1 s_A - f_\mathcal{D}) \bmod s_\mathcal{D}$
    $\texttt{first} = c + ((r - c) \bmod \texttt{stride})$
    $\texttt{lmfirst} = (\texttt{first} - f_\mathcal{D} - t) n_\mathcal{D}/s_\mathcal{D} + t + \texttt{K}$
    input-local-slice(lmfirst, lmlast, lmstride)

B.3 Case C: block, block-cyclic

Array A is block, and B is block-cyclic. The ownership sets are given by:

$$Own_\mathcal{D}(\texttt{A}) = (f_\mathcal{D} : f_\mathcal{D} + n_\mathcal{D} - 1 : 1)$$

$$Own_\mathcal{S}(\texttt{B}) = \bigcup_{i'=0}^{n_\mathcal{S}-1} (f_\mathcal{S} + i' : l_\mathcal{S} : s_\mathcal{S})$$

The revised LOCAL–INDEX–SEND algorithm is the following:

$(x_2, y_2, g_2) = \texttt{euclid}(s_B, s_{\mathcal{S}})$
$\texttt{stride} = s_B s_{\mathcal{S}} / g_2$
$\texttt{lmstride} = s_B n_{\mathcal{S}} / g_2$
$\texttt{last} = \min(f_B + s_B \lfloor (\min(l_A, f_{\mathcal{D}} + n_{\mathcal{D}} - 1) - f_A)/s_A \rfloor, l_{\mathcal{S}})$
$\texttt{lmlast} = \lfloor (\texttt{last} - f_{\mathcal{S}})/s_{\mathcal{S}} \rfloor n_{\mathcal{S}} + \min((\texttt{last} - f_{\mathcal{S}}) \bmod s_{\mathcal{S}}, n_{\mathcal{S}} - 1) + \texttt{K}$
$c = \max(f_B + s_B \lceil (\max(f_A, f_{\mathcal{D}}) - f_A)/s_A \rceil, f_{\mathcal{S}})$
DO $i = \lceil (f_{\mathcal{S}} - f_B)/g_2 \rceil$ UPTO $\lfloor (f_{\mathcal{S}} - f_B + n_{\mathcal{S}} - 1)/g_2 \rfloor$
  $i' = g_2 i - f_{\mathcal{S}} + f_B$
  $r = f_B + i x_2 s_B$
  $\texttt{first} = c + ((r - c) \bmod \texttt{stride})$
  $\texttt{lmfirst} = (\texttt{first} - f_{\mathcal{S}} - i') n_{\mathcal{S}} / s_{\mathcal{S}} + i' + \texttt{K}$
  output-local-slice(lmfirst, lmlast, lmstride)

The revised LOCAL–INDEX–RECEIVE algorithm is the following:

$(x_2, y_2, g_2) = \texttt{euclid}(s_B, s_{\mathcal{S}})$
$\texttt{stride} = s_A s_{\mathcal{S}} / g_2$
$\texttt{lmstride} = s_A s_{\mathcal{S}} / g_2$
$\texttt{last} = \min(l_A, f_{\mathcal{D}} + n_{\mathcal{D}} - 1, f_A + s_A \lfloor (l_{\mathcal{S}} - f_B)/s_B \rfloor)$
$\texttt{lmlast} = \min(\texttt{last} - f_{\mathcal{D}}, n_{\mathcal{D}} - 1) + \texttt{K}$
$c = \max(f_A, f_{\mathcal{D}}, f_A + s_A \lceil (f_{\mathcal{S}} - f_B)/s_B \rceil)$
DO $i = \lceil (f_{\mathcal{S}} - f_B)/g_2 \rceil$ UPTO $\lfloor (f_{\mathcal{S}} - f_B + n_{\mathcal{S}} - 1)/g_2 \rfloor$
  $r = f_A + i x_2 s_A$
  $\texttt{first} = c + ((r - c) \bmod \texttt{stride})$
  $\texttt{lmfirst} = \texttt{first} - f_{\mathcal{D}} + \texttt{K}$
  input-local-slice(lmfirst, lmlast, lmstride)

B.4 Case D: block, block

Array A is block, and B is block. The ownership sets are given by:

$$Own_{\mathcal{D}}(\texttt{A}) = (f_{\mathcal{D}} : f_{\mathcal{D}} + n_{\mathcal{D}} - 1 : 1)$$
$$Own_{\mathcal{S}}(\texttt{B}) = (f_{\mathcal{S}} : f_{\mathcal{S}} + n_{\mathcal{S}} - 1 : 1)$$

The revised LOCAL–INDEX–SEND algorithm is the following:

$\texttt{stride} = s_B$
$\texttt{lmstride} = s_B$
$\texttt{last} = \min(f_B + s_B \lfloor (\min(l_A, f_{\mathcal{D}} + n_{\mathcal{D}} - 1) - f_A)/s_A \rfloor, f_{\mathcal{S}} + n_{\mathcal{S}} - 1)$
$\texttt{lmlast} = \min(\texttt{last} - f_{\mathcal{S}}, n_{\mathcal{S}} - 1) + \texttt{K}$
$c = \max(f_B + s_B \lceil (\max(f_A, f_{\mathcal{D}}) - f_A)/s_A \rceil, f_{\mathcal{S}})$
$r = f_B$
$\texttt{first} = c + ((r - c) \bmod \texttt{stride})$
$\texttt{lmfirst} = \texttt{first} - f_{\mathcal{S}} + \texttt{K}$
output-local-slice(lmfirst, lmlast, lmstride)

The revised LOCAL–INDEX–RECEIVE algorithm is the following:

$\texttt{stride} = s_A$
$\texttt{lmstride} = s_A$
$\texttt{last} = \min(l_A, f_\mathcal{D} + n_\mathcal{D} - 1, f_A + s_A\lfloor(f_\mathcal{S} + n_\mathcal{S} - 1 - f_B)/s_B\rfloor)$
$\texttt{lmlast} = \min(\texttt{last} - f_\mathcal{D}, n_\mathcal{D} - 1) + \texttt{K}$
$c = \max(f_A, f_\mathcal{D}, f_A + s_A\lceil(f_\mathcal{S} - f_B)/s_B\rceil)$
$r = f_A$
$\texttt{first} = c + ((r - c) \bmod \texttt{stride})$
$\texttt{lmfirst} = \texttt{first} - f_\mathcal{D} + \texttt{K}$
$\texttt{input-local-slice}(\texttt{lmfirst}, \texttt{lmlast}, \texttt{lmstride})$

B.5 Case E: block, cyclic

Array A is block, and B is cyclic. The ownership sets are given by:

$$Own_\mathcal{D}(\texttt{A}) = (f_\mathcal{D} : f_\mathcal{D} + n_\mathcal{D} - 1 : 1)$$

$$Own_\mathcal{S}(\texttt{B}) = (f_\mathcal{S} : l_\mathcal{S} : s_\mathcal{S})$$

The revised LOCAL–INDEX–SEND algorithm is the following:

$(x_2, y_2, g_2) = \texttt{euclid}(s_B, s_\mathcal{S})$
$\texttt{stride} = s_B s_\mathcal{S}/g_2$
$\texttt{lmstride} = s_B/g_2$
$\texttt{last} = \min(f_B + s_B\lfloor(\min(l_A, f_\mathcal{D} + n_\mathcal{D} - 1) - f_A)/s_A\rfloor, l_\mathcal{S})$
$\texttt{lmlast} = \lfloor(\texttt{last} - f_\mathcal{S})/s_\mathcal{S}\rfloor + \texttt{K}$
$c = \max(f_B + s_B\lceil(\max(f_A, f_\mathcal{D}) - f_A)/s_A\rceil, f_\mathcal{S})$
IF $(f_B - f_\mathcal{S}) \bmod g_2 == 0$ THEN
  $r = f_B + (f_\mathcal{S} - f_B)x_2 s_B/g_2$
  $\texttt{first} = c + ((r - c) \bmod \texttt{stride})$
  $\texttt{lmfirst} = (\texttt{first} - f_\mathcal{S})/s_\mathcal{S} + \texttt{K}$
  $\texttt{output-local-slice}(\texttt{lmfirst}, \texttt{lmlast}, \texttt{lmstride})$

The revised LOCAL–INDEX–RECEIVE algorithm is the following:

$(x_2, y_2, g_2) = \texttt{euclid}(s_B, s_\mathcal{S})$
$\texttt{stride} = s_A s_\mathcal{S}/g_2$
$\texttt{lmstride} = s_A s_\mathcal{S}/g_2$
$\texttt{last} = \min(l_A, f_\mathcal{D} + n_\mathcal{D} - 1, f_A + s_A\lfloor(l_\mathcal{S} - f_B)/s_B\rfloor)$
$\texttt{lmlast} = \min(\texttt{last} - f_\mathcal{D}, n_\mathcal{D} - 1) + \texttt{K}$
$c = \max(f_A, f_\mathcal{D}, f_A + s_A\lceil(f_\mathcal{S} - f_B)/s_B\rceil)$
IF $(f_B - f_\mathcal{S}) \bmod g_2 == 0$ THEN
  $r = f_A + (f_\mathcal{S} - f_B)x_2 s_A/g_2$
  $\texttt{first} = c + ((r - c) \bmod \texttt{stride})$
  $\texttt{lmfirst} = \texttt{first} - f_\mathcal{D} + \texttt{K}$
  $\texttt{input-local-slice}(\texttt{lmfirst}, \texttt{lmlast}, \texttt{lmstride})$

B.6 Case F: cyclic, block-cyclic

Array A is cyclic, and B is block-cyclic. The ownership sets are given by:

$$Own_{\mathcal{D}}(\mathtt{A}) = (f_{\mathcal{D}} : l_{\mathcal{D}} : s_{\mathcal{D}})$$

$$Own_{\mathcal{S}}(\mathtt{B}) = \bigcup_{i'=0}^{n_{\mathcal{S}}-1} (f_{\mathcal{S}} + i' : l_{\mathcal{S}} : s_{\mathcal{S}})$$

The revised LOCAL–INDEX–SEND algorithm is the following:

$(x_1, y_1, g_1) = \mathtt{euclid}(s_A, s_{\mathcal{D}})$
$(x_2, y_2, g_2) = \mathtt{euclid}(s_B s_{\mathcal{D}}/g_1, s_{\mathcal{S}})$
$\mathtt{stride} = s_B s_{\mathcal{D}} s_{\mathcal{S}}/(g_1 g_2)$
$\mathtt{lmstride} = s_B s_{\mathcal{D}} n_{\mathcal{S}}/(g_1 g_2)$
$\mathtt{last} = \min(f_B + s_B \lfloor(\min(l_A, l_{\mathcal{D}}) - f_A)/s_A\rfloor, l_{\mathcal{S}})$
$\mathtt{lmlast} = \lfloor(\mathtt{last} - f_{\mathcal{S}})/s_{\mathcal{S}}\rfloor n_{\mathcal{S}} + \min((\mathtt{last} - f_{\mathcal{S}}) \bmod s_{\mathcal{S}}, n_{\mathcal{S}} - 1) + \mathtt{K}$
$c = \max(f_B + s_B \lceil(\max(f_A, f_{\mathcal{D}}) - f_A)/s_A\rceil, f_{\mathcal{S}})$
IF $(f_A - f_{\mathcal{D}}) \bmod g_1 == 0$ THEN
  DO $i = \lceil(f_{\mathcal{S}} - f_B - (f_{\mathcal{D}} - f_A)x_1 s_B/g_1)/g_2\rceil$ UPTO
      $\lfloor(f_{\mathcal{S}} - f_B - (f_{\mathcal{D}} - f_A)x_1 s_B/g_1 + n_{\mathcal{S}} - 1)/g_2\rfloor$
    $i' = g_2 i - f_{\mathcal{S}} + f_B + (f_{\mathcal{D}} - f_A)x_1 s_B/g_1$
    $r = f_B + (f_{\mathcal{D}} - f_A)x_1 s_B/g_1 + i x_2 s_B s_{\mathcal{D}}/g_1$
    $\mathtt{first} = c + ((r - c) \bmod \mathtt{stride})$
    $\mathtt{lmfirst} = (\mathtt{first} - f_{\mathcal{S}} - i')n_{\mathcal{S}}/s_{\mathcal{S}} + i' + \mathtt{K}$
    output-local-slice(lmfirst, lmlast, lmstride)

The revised LOCAL–INDEX–RECEIVE algorithm is the following:

$(x_1, y_1, g_1) = \mathtt{euclid}(s_A, s_{\mathcal{D}})$
$(x_2, y_2, g_2) = \mathtt{euclid}(s_B s_{\mathcal{D}}/g_1, s_{\mathcal{S}})$
$\mathtt{stride} = s_A s_{\mathcal{D}} s_{\mathcal{S}}/(g_1 g_2)$
$\mathtt{lmstride} = s_A s_{\mathcal{S}}/(g_1 g_2)$
$\mathtt{last} = \min(l_A, l_{\mathcal{D}}, f_A + s_A \lfloor(l_{\mathcal{S}} - f_B)/s_B\rfloor)$
$\mathtt{lmlast} = \lfloor(\mathtt{last} - f_{\mathcal{D}})/s_{\mathcal{D}}\rfloor + \mathtt{K}$
$c = \max(f_A, f_{\mathcal{D}}, f_A + s_A \lceil(f_{\mathcal{S}} - f_B)/s_B\rceil)$
IF $(f_A - f_{\mathcal{D}}) \bmod g_1 == 0$ THEN
  DO $i = \lceil(f_{\mathcal{S}} - f_B - (f_{\mathcal{D}} - f_A)x_1 s_B/g_1)/g_2\rceil$ UPTO
      $\lfloor(f_{\mathcal{S}} - f_B - (f_{\mathcal{D}} - f_A)x_1 s_B/g_1 + n_{\mathcal{S}} - 1)/g_2\rfloor$
    $r = f_A + (f_{\mathcal{D}} - f_A)x_1 s_A/g_1 + i x_2 s_A s_{\mathcal{D}}/g_1$
    $\mathtt{first} = c + ((r - c) \bmod \mathtt{stride})$
    $\mathtt{lmfirst} = (\mathtt{first} - f_{\mathcal{D}})/s_{\mathcal{D}} + \mathtt{K}$
    input-local-slice(lmfirst, lmlast, lmstride)

B.7 Case G: cyclic, block

Array A is cyclic, and B is block. The ownership sets are given by:

$$Own_{\mathcal{D}}(\mathtt{A}) = (f_{\mathcal{D}} : l_{\mathcal{D}} : s_{\mathcal{D}})$$

$$Own_{\mathcal{S}}(\mathtt{B}) = (f_{\mathcal{S}} : f_{\mathcal{S}} + n_{\mathcal{S}} - 1 : 1)$$

The revised LOCAL–INDEX–SEND algorithm is the following:

$(x_1, y_1, g_1) = \mathtt{euclid}(s_A, s_{\mathcal{D}})$
$\mathtt{stride} = s_B s_{\mathcal{D}} / g_1$
$\mathtt{lmstride} = s_B s_{\mathcal{D}} / g_1$
$\mathtt{last} = \min(f_B + s_B \lfloor (\min(l_A, l_{\mathcal{D}}) - f_A)/s_A \rfloor, f_{\mathcal{S}} + n_{\mathcal{S}} - 1)$
$\mathtt{lmlast} = \min(\mathtt{last} - f_{\mathcal{S}}, n_{\mathcal{S}} - 1) + \mathtt{K}$
$c = \max(f_B + s_B \lceil (\max(f_A, f_{\mathcal{D}}) - f_A)/s_A \rceil, f_{\mathcal{S}})$
IF $(f_A - f_{\mathcal{D}}) \bmod g_1 == 0$ THEN
  $r = f_B + (f_{\mathcal{D}} - f_A) x_1 s_B / g_1$
  $\mathtt{first} = c + ((r - c) \bmod \mathtt{stride})$
  $\mathtt{lmfirst} = \mathtt{first} - f_{\mathcal{S}} + \mathtt{K}$
  output-local-slice(lmfirst, lmlast, lmstride)

The revised LOCAL–INDEX–RECEIVE algorithm is the following:

$(x_1, y_1, g_1) = \mathtt{euclid}(s_A, s_{\mathcal{D}})$
$\mathtt{stride} = s_A s_{\mathcal{D}} / g_1$
$\mathtt{lmstride} = s_A / g_1$
$\mathtt{last} = \min(l_A, l_{\mathcal{D}}, f_A + s_A \lfloor (f_{\mathcal{S}} + n_{\mathcal{S}} - 1 - f_B)/s_B \rfloor)$
$\mathtt{lmlast} = \lfloor (\mathtt{last} - f_{\mathcal{D}})/s_{\mathcal{D}} \rfloor + \mathtt{K}$
$c = \max(f_A, f_{\mathcal{D}}, f_A + s_A \lceil (f_{\mathcal{S}} - f_B)/s_B \rceil)$
IF $(f_A - f_{\mathcal{D}}) \bmod g_1 == 0$ THEN
  $r = f_A + (f_{\mathcal{D}} - f_A) x_1 s_A / g_1$
  $\mathtt{first} = c + ((r - c) \bmod \mathtt{stride})$
  $\mathtt{lmfirst} = (\mathtt{first} - f_{\mathcal{D}})/s_{\mathcal{D}} + \mathtt{K}$
  input-local-slice(lmfirst, lmlast, lmstride)

B.8 Case H: cyclic, cyclic

Array A is cyclic, and B is cyclic. The ownership sets are given by:

$$Own_{\mathcal{D}}(\mathtt{A}) = (f_{\mathcal{D}} : l_{\mathcal{D}} : s_{\mathcal{D}})$$

$$Own_{\mathcal{S}}(\mathtt{B}) = (f_{\mathcal{S}} : l_{\mathcal{S}} : s_{\mathcal{S}})$$

The revised LOCAL–INDEX–SEND algorithm is the following:

$(x_1, y_1, g_1) = \texttt{euclid}(s_A, s_{\mathcal{D}})$
$(x_2, y_2, g_2) = \texttt{euclid}(s_B s_{\mathcal{D}}/g_1, s_{\mathcal{S}})$
$\texttt{stride} = s_B s_{\mathcal{D}} s_{\mathcal{S}}/(g_1 g_2)$
$\texttt{lmstride} = s_B s_{\mathcal{D}}/(g_1 g_2)$
$\texttt{last} = \min(f_B + s_B \lfloor (\min(l_A, l_{\mathcal{D}}) - f_A)/s_A \rfloor, l_{\mathcal{S}})$
$\texttt{lmlast} = \lfloor (\texttt{last} - f_{\mathcal{S}})/s_{\mathcal{S}} \rfloor + \texttt{K}$
$c = \max(f_B + s_B \lceil (\max(f_A, f_{\mathcal{D}}) - f_A)/s_A \rceil, f_{\mathcal{S}})$
IF $(f_A - f_{\mathcal{D}}) \bmod g_1 == 0$ AND $(f_B + (f_{\mathcal{D}} - f_A)x_1 s_B/g_1 - f_{\mathcal{S}}) \bmod g_2 == 0$ THEN
  $t = (f_{\mathcal{D}} - f_A)x_1 s_B/g_1$
  $r = f_B + t + (f_{\mathcal{S}} - f_B - t)x_2 s_B s_{\mathcal{D}}/(g_1 g_2)$
  $\texttt{first} = c + ((r - c) \bmod \texttt{stride})$
  $\texttt{lmfirst} = (\texttt{first} - f_{\mathcal{S}})/s_{\mathcal{S}} + \texttt{K}$
  output-local-slice(lmfirst, lmlast, lmstride)

The revised LOCAL–INDEX–RECEIVE algorithm is the following:

$(x_1, y_1, g_1) = \texttt{euclid}(s_A, s_{\mathcal{D}})$
$(x_2, y_2, g_2) = \texttt{euclid}(s_B s_{\mathcal{D}}/g_1, s_{\mathcal{S}})$
$\texttt{stride} = s_A s_{\mathcal{D}} s_{\mathcal{S}}/(g_1 g_2)$
$\texttt{lmstride} = s_A s_{\mathcal{S}}/(g_1 g_2)$
$\texttt{last} = \min(l_A, l_{\mathcal{D}}, f_A + s_A \lfloor (l_{\mathcal{S}} - f_B)/s_B \rfloor)$
$\texttt{lmlast} = \lfloor (\texttt{last} - f_{\mathcal{D}})/s_{\mathcal{D}} \rfloor + \texttt{K}$
$c = \max(f_A, f_{\mathcal{D}}, f_A + s_A \lceil (f_{\mathcal{S}} - f_B)/s_B \rceil)$
IF $(f_A - f_{\mathcal{D}}) \bmod g_1 == 0$ AND $(f_B + (f_{\mathcal{D}} - f_A)x_1 s_B/g_1 - f_{\mathcal{S}}) \bmod g_2 == 0$ THEN
  $t = (f_{\mathcal{D}} - f_A)x_1 s_B/g_1$
  $r = f_A + (f_{\mathcal{D}} - f_A)x_1 s_A/g_1 + (f_{\mathcal{S}} - f_B - t)x_2 s_A s_{\mathcal{D}}/(g_1 g_2)$
  $\texttt{first} = c + ((r - c) \bmod \texttt{stride})$
  $\texttt{lmfirst} = (\texttt{first} - f_{\mathcal{D}})/s_{\mathcal{D}} + \texttt{K}$
  input-local-slice(lmfirst, lmlast, lmstride)

C Optimized computation for block and cyclic distributions

As in Appendix B, we can use an optimized version of the COMPUTE–LOOP algorithm when the distribution of the array on the left-hand side of the assignment statement is known to be block or cyclic. Since the distribution of only the left-hand side is relevant, we need to derive only two additional algorithms, rather than eight.

C.1 Case A: block

Array A has a block distribution. The ownership set is given by:

$$Own_{\mathcal{D}}(\texttt{A}) = (f_{\mathcal{D}} : f_{\mathcal{D}} + n_{\mathcal{D}} - 1 : 1)$$

The revised COMPUTE–LOOP algorithm is the following:

```
stride = s_A
lmstride = s_A
last = min(l_A, f_D + n_D − 1)
lmlast = last − f_D + K
c = max(f_A, f_D)
r = f_A
first = c + ((r − c) mod stride)
lmfirst = first − f_D + K
DO i = lmfirst UPTO lmlast BY lmstride
   A(i) = F(T(i))
```

C.2 Case B: cyclic

Array A has a cyclic distribution. The ownership set is given by:

$$Own_{\mathcal{D}}(\mathtt{A}) = (f_{\mathcal{D}} : l_{\mathcal{D}} : s_{\mathcal{D}})$$

The revised COMPUTE–LOOP algorithm is the following:

```
(x_1, y_1, g_1) = euclid(s_A, s_D)
IF (f_A − f_D) mod g_1 == 0 THEN
   stride = s_A s_D / g_1
   lmstride = s_A / g_1
   last = min(l_A, l_D)
   lmlast = ⌊(last − f_D)/s_D⌋ + K
   c = max(f_A, f_D)
   r = f_A + (f_D − f_A) x_1 s_A / g_1
   first = c + ((r − c) mod stride)
   lmfirst = ⌊(first − f_D)/s_D⌋ + K
   DO i = lmfirst UPTO lmlast BY lmstride
      A(i) = F(T(i))
```

D Notation

There are many variables and conventions used throughout the paper. In this appendix, we provide a description of many of these variables.

Ψ: The set of global array elements to be sent from one processor to another.

A: The array on the left-hand side of the assignment statement.

B: The array on the right-hand side of the assignment statement.

b_i: The *alignment offset* for an array dimension. The subscript refers to either a particular dimension of a template or a particular one-dimensional array.

c_i: The lower index bound of a template dimension with which a particular array dimension is aligned. The subscript refers to either a particular dimension of a template or a particular one-dimensional array.

$\mathcal{F}$: An arbitrary element-wise intrinsic function, such as sin, log, or the identity.

f: The *first* component of a slice, as in $(f : l : s)$.

f_A: The *first* component of the slice on the left-hand side of the array assignment statement, as in $\texttt{A}(f_A : l_A : s_A) = \mathcal{F}(\texttt{B}(f_B : l_B : s_B))$.

f_B: The *first* component of the slice on the right-hand side of the array assignment statement, as in $\texttt{A}(f_A : l_A : s_A) = \mathcal{F}(\texttt{B}(f_B : l_B : s_B))$.

$f_{\mathcal{D}}$: The index of the *first* element of the left-hand side array owned by receiving (destination) processor $\mathcal{D}$.

$f_{\mathcal{S}}$: The index of the *first* element of the right-hand side array owned by sending processor $\mathcal{S}$.

g: The greatest common divisor (GCD) of some pair of input integers.

K: The first index of a particular array. In the C programming language, K is always 0; in Fortran K is usually 1.

l: The *last* component of a slice, as in $(f : l : s)$.

l_A: The *last* component of the slice on the left-hand side of the array assignment statement, as in $\texttt{A}(f_A : l_A : s_A) = \mathcal{F}(\texttt{B}(f_B : l_B : s_B))$.

l_B: The *last* component of the slice on the right-hand side of the array assignment statement, as in $\texttt{A}(f_A : l_A : s_A) = \mathcal{F}(\texttt{B}(f_B : l_B : s_B))$.

$l_{\mathcal{D}}$: The index of the *last* element of the left-hand side array owned by receiving (destination) processor $\mathcal{D}$.

$l_{\mathcal{S}}$: The index of the *last* element of the right-hand side array owned by sending processor $\mathcal{S}$.

LM: A function that maps a global array index into an index of a local array on a processor. LM is defined in equation (16).

Map: A function that maps a global index of an array into an index of a per-processor local array. Map is defined in equation (5).

$n_{\mathcal{D}}$: The *block size* of the distribution of the left-hand side array.

$n_{\mathcal{S}}$: The *block size* of the distribution of the right-hand side array.

n': The block size of the template with which the array is aligned. This parameter differs from the block size, n, of the array when $o^L \neq o^R$. The identity $n' = n + o^L - o^R$ always holds.

o^L: The alignment overlap on the left. This value is 0 except in the case of partially replicated distributions (see Section 10.4).

o^R: The alignment overlap on the right. This value is 0 except in the case of partially replicated distributions (see Section 10.4).

$Own_{\mathcal{D}}(\texttt{A})$: The set of indices corresponding to the elements of left-hand side array A owned by receiving (destination) processor $\mathcal{D}$. This set can be represented using the four parameters $f_{\mathcal{D}}$, $l_{\mathcal{D}}$, $s_{\mathcal{D}}$, and $n_{\mathcal{D}}$.

$Own_{\mathcal{S}}(\texttt{B})$**:** The set of indices corresponding to the elements of right-hand side array `B` owned by sending processor $\mathcal{S}$. This set can be represented using the four parameters $f_{\mathcal{S}}$, $l_{\mathcal{S}}$, $s_{\mathcal{S}}$, and $n_{\mathcal{S}}$.

$\vec{p}$**:** A processor ID expressed as a vector, with one component for each dimension of the array, where $\vec{0} \leq \vec{p} < \vec{v}$. Two different processors will differ in at least one component of their corresponding $\vec{p}$ vectors.

$\mathcal{R}_{i,j,\ldots}$**:** A function that returns a *representative* of an input set. The subscript of $\mathcal{R}$ is an index list. For each valid instantiation of the indices in the index list, there is a different representative. Associated with $\mathcal{R}$ is a lower bound, an upper bound, and a stride, each of which is independent of the indices in the index list. Each representative, along with the lower bound, upper bound, and stride, defines a slice, and the union of the slices over the valid instantiations of the index list results in the input set.

r**:** A *representative* element of an infinite slice, as described in Section 5.

s**:** The *stride* component of a slice, as in $(f : l : s)$.

s_A**:** The *stride* component of the slice on the left-hand side of the array assignment statement, as in $\texttt{A}(f_A : l_A : s_A) = \mathcal{F}(\texttt{B}(f_B : l_B : s_B))$.

s_B**:** The *stride* component of the slice on the right-hand side of the array assignment statement, as in $\texttt{A}(f_A : l_A : s_A) = \mathcal{F}(\texttt{B}(f_B : l_B : s_B))$.

$s_{\mathcal{D}}$**:** The *stride* of the distribution of the left-hand side array. The distribution stride is defined as the difference between the starting points of two consecutive blocks owned by a processor.

$s_{\mathcal{S}}$**:** The *stride* of the distribution of the right-hand side array. The distribution stride is defined as the difference between the starting points of two consecutive blocks owned by a processor.

$\vec{v}$**:** A vector giving the number of processors assigned to each corresponding dimension of an array.

x**:** An integer that satisfies $ax + by = \gcd(a, b)$ for nonnegative integers a and b, for an integer y. Given a and b, the integers x, y, and $\gcd(a, b)$ can be found by using Euclid's extended GCD algorithm.

y**:** An integer that satisfies $ax + by = \gcd(a, b)$ for nonnegative integers a and b, for an integer x. Given a and b, the integers x, y, and $\gcd(a, b)$ can be found by using Euclid's extended GCD algorithm.