Sixth Annual Workshop on Languages and Compilers for Parallel Computing, August 1993, Portland, OR

Generating Communication for Array Statements: Design, Implementation, and Evaluation

James M. Stichnoth, David O'Hallaron, and Thomas R. Gross
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Array statements as included in Fortran 90 or High Performance Fortran (HPF) are a well-accepted way to specify data parallelism in programs. When generating code for such a data parallel program for a private memory parallel system, the compiler must determine when array elements must be moved from one processor to another. This paper describes a practical method to compute the set of array elements that are to be moved; it covers all the distributions that are included in HPF: block, cyclic, and block-cyclic. This method is the foundation for an efficient protocol for modern private memory parallel systems: for each block of data to be sent, the sender processor computes the local address in the receiver's address space, and the address is then transmitted together with the data. This strategy increases the communication load but reduces the overhead on the receiving processor. We implemented this optimization in an experimental Fortran compiler, and this paper reports an empirical evaluation on a 64-node private memory iWarp system, using a number of different distributions.

1 Introduction

One approach to programming parallel systems that has received considerable attention is to let the programmer explicitly expose the data parallelism in a program and to let the compiler deal with the details of mapping the program onto the target parallel system. Array statements are an attractive linguistic construct to capture the data parallelism in a program; they succinctly describe computations for a collection of data elements. Array statements have been added to earlier versions of Fortran (e.g., CM Fortran), are part of the standard Fortran (i.e., Fortran 90), and can be found in various extensions (e.g., Fortran D, High Performance Fortran (HPF), etc.). Conceptually similar constructs are also included in other languages [1], but we concentrate in this paper on the kind of array statement as defined in Fortran 90 or HPF.

Consider an array statement such as:

$$\mathtt{A}(f_1 : l_1 : s_1) = \mathcal{F}(\mathtt{B}(f_2 : l_2 : s_2)).$$

$\mathtt{A}(f : l : s)$ defines a sequence of array elements $\mathtt{A}(f + i \times s), 0 \le i \le \lfloor (l - f)/s \rfloor$. s is called the *access stride*, and the sequence of array elements is sometimes referred to as a *slice* of the array. $\mathcal{F}$ is an element-wise function; i.e., the result of $\mathcal{F}(\mathtt{B}(\mathtt{i}))$ is stored in $\mathtt{A}(\mathtt{j})$ for corresponding values of $\mathtt{i}$ and $\mathtt{j}$. The semantics of the assignment statement require that the operands on the right-hand side are read before any of the results are written. A compiler that is to translate the above array statement for execution on a private memory parallel system must map the global name space of the data parallel program into the memory organization provided by the parallel system. That is, the compiler must address two issues:

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. Supported also in part by the Air Force Office of Scientific Research under Contract F49620–92–J–0131. This material is based upon work supported in part under a National Science Foundation Graduate Research Fellowship.

- Data location: the compiler must determine how the elements of different arrays are stored relative to each other (alignment), and how they are distributed onto the parallel processors (distribution). That is, for each array element `A(i)` of an array `A`, the compiler must define two mapping functions: an *ownership* function that produces the processor at which `A(i)` resides (the *owner*), and a *local memory* function that determines the actual address of `A(i)` in the local memory space of the owner.
- Computation location: the compiler must determine on which processor the computation of $\mathcal{F}$(`B(i)`) is to take place, for each index `i` as defined by the array statement.

The alignment and distribution information can be specified with directives by the user [10, 18] (such as with `ALIGN` and `DISTRIBUTE` directives in HPF and Fortran D), or can be derived by a compiler (e.g., [5]). A number of compilers have tackled these problems by using a simple rule to address the second issue: the processor that stores the array element on the left-hand side of an array statement (i.e., the owner) performs the computation to update this array element ("owner computes rule")[18]. Those operands on the left-hand side that are non-local must be moved to the processor that performs the update (either by calls to a communication library or by direct I/O operations).

The problem of mapping data and computation onto a parallel system is a key issue for any compiler, and a number of research projects have reported various methods (see [11, 3, 10, 19, 14, 8, 13, 7, 17]). This paper concentrates on the phase of a compiler for private memory parallel systems that deals with the generation of communication operations. Given alignment and distribution, we describe a practical machine-independent method to determine the array elements that must be copied from one processor to another. This phase is independent of how the compiler determines the alignment or distribution but has a high impact on the quality of the code produced. The method described here is the foundation for an efficient protocol for a wide range of private memory parallel systems. Section 2 states the problem; Section 3 describes our approach and presents the basic protocol for private memory parallel systems. Section 4 discusses an optimization that is attractive for parallel systems with ample communication bandwidth. This optimization is implemented in an experimental Fortran compiler for a private memory parallel system. In Section 5 we report on performance measurements obtained on a 64-node iWarp system that demonstrate the benefits of this optimization.

2 Compiling array statements

The discussion in the rest of the paper assumes a compiler uses the owner computes rule and focuses on the issues of the code generation phase.

The earlier example of an array statement included several simplifications. An array statement is any statement of the source language that allows an array `A` or an array slice such as $\mathtt{A}(f : l : s)$ as a source or destination operand. In general, arrays can have more than one dimension, and there can be multiple array operands in a single statement.

Array *assignment* statements exhibit all the interesting problems of mapping array statements onto private memory parallel systems. Any solution to this class of problems can easily be extended to cover the compilation of most other data parallel constructs, such as the `WHERE` statement, the `DOALL` statement, or building the parameter list for a subroutine call. For this reason, we concentrate on the array assignment statement in this paper. The reader is referred to [16] for a full treatment of the compilation of this statement and other classes of array statements.

Consider an array (assignment) statement such as:

$$\mathtt{A}(f_1 : l_1 : s_1, f_2 : l_2 : s_2, \ldots) = \mathtt{B}(f_a : l_a : s_a, f_b : l_b : s_b, \ldots) \oplus \mathtt{C}(f_x : l_x : s_x, f_y : l_y : s_y, \ldots)$$

where `A`, `B`, and `C` are arrays, $\oplus$ is an arbitrary element-wise binary operator, and the alignment and distribution of these arrays have already been determined.

Since the compiler uses the owner computes rule, all required elements of the arrays on the right-hand side must be present on the processor that holds the left-hand side element. If an array element is not already resident on this processor, the compiler must move (or copy) the array element. This implies that the problem of determining the communication operations for an array assignment statement can be reduced to dealing with an array assignment with a single term on the right-hand side:

$$\mathtt{A}(f_1 : l_1 : s_1, f_2 : l_2 : s_2, \ldots) = \mathtt{B}(f_a : l_a : s_a, f_b : l_b : s_b, \ldots)$$

Any solution that moves the correct elements of B can be extended to move the correct elements of C (and all other terms on the right-hand side).

Furthermore, each dimension of a multi-dimensional array can be dealt with orthogonally, provided that the data layout in one dimension is not allowed to affect that of another dimension (ALIGN and DISTRIBUTE statements from HPF satisfy this condition). That is, we only need to consider the following simple array statement:

$$\mathtt{A}(f_1 : l_1 : s_1) = \mathtt{B}(f_2 : l_2 : s_2)$$

Then the basic method to execute this statement conceptually consists of the following steps for each processor P_i [13]:

1. Send those elements of B that are owned by P_i but required by another processor P_j to this P_j, $P_j \in \{P_0, \ldots, P_{n-1}\}$.
2. Receive non-local elements of B required to update a local element of $\mathtt{A}(f_1 : l_1 : s_1)$, and store these elements locally.
3. Execute the updates of the local elements of $\mathtt{A}(f_1 : l_1 : s_1)$, using local elements $\mathtt{B}(j)$.

If $P_i = P_j$, no inter-processor communication is required. A compiler has the option to use the original elements of B in the computation in step 3, or to copy the local elements of B to the same space that holds the values received in step 2. In our experience, this latter option simplifies step 3 (and the resulting code) and is therefore implemented in our compiler. However, this decision is not central to the topic of this paper. There exist also other optimizations for evaluating the above three steps. For example, at compile time, the compiler might be able to detect the specific communication pattern, and determine exactly which data must be sent to each processor. The compiler may also be able to overlap these steps, for example by separately handling computations involving only local elements of B, as discussed above. Overlapping computations on local data could be beneficial because these computations can be started before any item is received. However this optimization is not discussed here further; instead, we concentrate on the more fundamental task of producing efficient communication code. The key issue for compiling the array statement is to determine which elements of B to send (and where to send them) and which elements to receive (and where in memory to place the received values).

3 Communication for the basic assignment statement

When executing an array statement, the communication pattern can be arbitrarily complex: anywhere from no communication at all, to a simple shift operation, to an all-to-all communication. For each processor, the compiler must determine the set of processors to which this processor must send data, and must correspondingly determine the set of processors from which to receive data. Several methods of detecting simple communication patterns are described in [16], but in some cases the compiler must be conservative and assume the possibility of communication with all other processors. Regardless of the sophistication of the compiler in detecting communication patterns, the compiler must be able to handle the general problem of determining communication between an arbitrary pair of processors.

Given the core array assignment statement discussed previously, our goal is to calculate the *exact* set of array elements that need to be sent between any two processors. An alternative approach is to calculate a superset that includes the array elements required and leave it to the receiving processor to determine which elements are actually needed. The advantages and tradeoffs of such an overestimate are not clear, so we choose to perform an exact calculation.

In the most general case, the arrays A and B of $\mathtt{A}(f_1 : l_1 : s_1) = \mathtt{B}(f_2 : l_2 : s_2)$ are distributed across the processors in block-cyclic fashion.[1] We concentrate in this paper on the general block-cyclic distributions, since a code generator phase that can handle block-cyclic distributions of arrays can also handle the simpler cases of block

[1] For the code generation phase, it is not important that alignment and distribution of an array remain unchanged during the execution of a program. Re-alignment or re-distribution can be implemented by array assignment statements, where the right-hand side operand has the old alignment or distribution, and the left-hand side operand has the new one.

and cyclic distributions. Figure 1 depicts a block-cyclic distribution; elements of the array that reside on one particular processor are shown shaded. This processor owns indices beginning with f, in blocks of size n. The distance between the start of one block and the start of the next is the *distribution stride* s, which is usually the product of the block size n and the number of processors over which the array dimension is distributed. For each processor, there is a set of indices denoting the array elements that reside on the processor. Because of the regular nature of the block-cyclic distribution, this index set can be specified using only four parameters: f, l, s, and n (l is the last element of the array). We say that the processor "owns" the array elements corresponding to this index set.

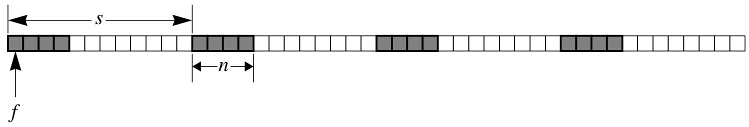

Figure 1: Block-cyclic distribution

The key parts of the compilation of the array assignment are to determine which array elements to send, and to determine where in memory to store received elements. The basis of our solution is a practical way to compute the intersections of slices. This can then be used to determine the array elements involved in an array expression as well as to characterize block-cyclic ownership sets. Since slices can be characterized by three parameters (first element, last element, access stride), and four parameters can be used to characterize ownership sets in terms of slices (as shown in Section 3.2), the resulting intersections can also be specified using a small number of parameters.

3.1 Slice intersection

A slice parameterizes an arithmetic sequence of integers, and our goal is 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 2 shows two examples of slice intersections.

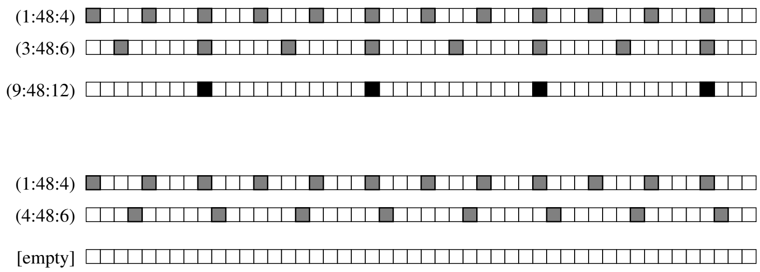

Figure 2: Two examples of slice intersection: $(1 : 48 : 4) \cap (3 : 48 : 6) = (9 : 48 : 12)$ and $(1 : 48 : 4) \cap (4 : 48 : 6) = \emptyset$.

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

Consider the slice $(f : l : s)$. Recall that f represents the first element (with the smallest index) 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 element with the largest index, and l is a lower bound.

To capture an infinite sequence that contains $(f : l : s)$ as a subsequence, we notice that there exist an infinite number of integers r such that the elements of $(f : l : s)$ can be described as $r + q \times s$ for *some* integer q. We therefore introduce a modified representation of a slice, $(f : l : s : r)$. This 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 $r + qs$ for some integer q. 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 q, $r + qs$ is a member of the original set if and only if $f \leq r + qs \leq l$ 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:

$$r = r_i + ms_i = r_j + ns_j. \tag{1}$$

From equation (1) it follows 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]. A theorem of modular linear equations ([6], Chapter 33) states 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:

$$m = \frac{(r_j - r_i)x}{g}. \tag{2}$$

Substituting (2) into (1), we obtain:

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

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, we have that:

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

Note that the representative is not necessarily the same as either of the lower bounds. This freedom provides the motivation for using this modified slice representation. If we are taking the intersections of more than two slices, as is necessary when deriving the communication sets, it is more efficient to minimize the number of conversions between representations.

3.2 Block-cyclic sets

As illustrated by Figure 1, the set of array elements stored on a processor when the array is distributed in a block-cyclic fashion can be represented using four parameters: f, l, s, and n. The f parameter is the lowest-numbered array index owned; the l parameter is an upper bound on the highest-numbered array index owned; the n parameter is the block size; and the s parameter is the distance between the start of one block and the start of the next. We call this set of indices that is induced by the block-cyclic distribution of the array a *block-cyclic set*.

We showed above that slices are closed under intersection. Unfortunately, the same does not hold for block-cyclic sets, as illustrated in Figure 3. Thus to take intersections of block-cyclic sets, we need to develop a different representation. We choose to represent a block-cyclic set as a union of disjoint slices $\bigcup_{k=0}^{n-1}(f + k : l : s)$, as pictured in Figure 4.

With the above method to compute the intersection of slices, we are now ready to introduce the algorithm to determine the array elements that must be sent and received by a given processor. Our goal is to devise a code template that can be included in the code for each processor to determine the communication sets.

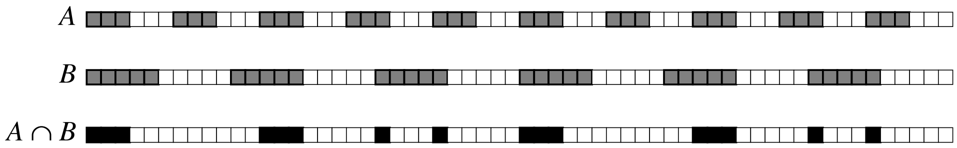

Figure 3: Block-cyclic sets are not closed under intersection.

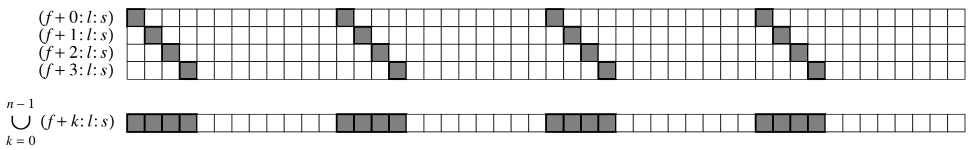

Figure 4: A block-cyclic distribution as a union of disjoint slices.

3.3 Communication: sender

The compiler must produce code for a given sending processor S and a receiving processor D such that S computes the elements of B that must be sent to D. Since A and B both have block-cyclic distributions, there exist f_3, l_3, n_3, s_3, f_4, l_4, n_4, and s_4 such that the portion of A that processor D owns is given by:

$$Own_d(\mathtt{A}) = \bigcup_{j=0}^{n_3-1} (f_3 + j : l_3 : s_3) \tag{3}$$

and the portion of B that processor S owns is given by:

$$Own_s(\mathtt{B}) = \bigcup_{i=0}^{n_4-1} (f_4 + i : l_4 : s_4). \tag{4}$$

Note that the names of processors S and D never appear in the set definitions; they are implicitly encoded in the parameters f_3 and f_4, which can be computed at runtime. Also note that each ownership set will be the union of disjoint sets, provided that $n_3 \leq s_3$ and $n_4 \leq s_4$.

The basic method of determining the communication is to take intersections of sets. All these sets can be described using either 3 or 4 parameters (3 in the case of a single slice, and 4 in the case of a block-cyclic set), although the resulting intersections are not nearly as simple to represent. Figure 5 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 indices 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; this corresponds to the intersection of the block-cyclic ownership sets. Even though the ownership sets for the sender and receiver are simple to represent in isolation, their intersection, although somewhat regular, is more complicated. Since code at runtime must step through these arrays to pick out the elements to be sent, any irregularity in the access pattern makes it more difficult for the single-node compiler to optimize these accesses (e.g., via induction variable elimination). The code for constant-stride accesses usually involves only additions, and the processor can overlap address computations with array accesses. If the access pattern is irregular, the code may include expensive operations like multiplications, divisions, or modulo operations; therefore computing the addresses takes more time than accessing the array elements, which means that address computations cannot be overlapped with accessing the array elements. The situation becomes even more complicated when we use non-unit access strides in the assignment statement. This example also illustrates that there can be little similarity between the two intersections, even though $Own_{d_1}(\mathtt{A})$ is simply the complement of $Own_{d_2}(\mathtt{A})$.

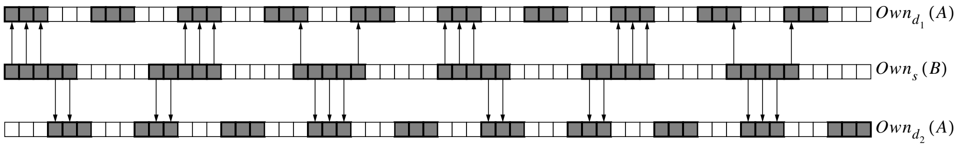

Figure 5: Communication sets that are difficult to parameterize.

The set of indices[2] of A owned by processor D that are used in the computation is given by $S_1 = Own_d(\texttt{A}) \cap (f_1 : l_1 : s_1)$. The set of indices of B owned by processor S that are used in the computation is given by $S_2 = Own_s(\texttt{B}) \cap (f_2 : l_2 : s_2)$. Processor S sends the ith member of the sequence $(f_2 : l_2 : s_2)$ if and only if the ith member of the sequence $(f_2 : l_2 : s_2)$ is in S_2, and the ith member of the sequence $(f_1 : l_1 : s_1)$ is in S_1. Suppose index x is the ith member of the sequence $(f_2 : l_2 : s_2)$, where $i = 0$ represents the first member of the sequence. This tells us that $\frac{x-f_2}{s_2} = i$. Now the ith member of the sequence $(f_1 : l_1 : s_1)$ is $f_1 + is_1$, which is equal to $\frac{x-f_2}{s_2}s_1 + f_1$. This means that processor S sends index x if and only if $x \in S_2$ and $\frac{x-f_2}{s_2}s_1 + f_1 \in S_1$. Equivalently, processor S sends index $\frac{y-f_1}{s_1}s_2 + f_2$ if and only if $y \in S_1$ and $\frac{y-f_1}{s_1}s_2 + f_2 \in S_2$.

This motivates the definition of a Map function, which maps a member of the sequence $(f_1 : l_1 : s_1)$ to the corresponding member of the sequence $(f_2 : l_2 : s_2)$. It is defined on an index y as:

$$Map(y) = \frac{y - f_1}{s_1} \cdot s_2 + f_2,$$

and the Map function is easily extended to a set or sequence in the obvious fashion. The purpose of this Map function is to associate $\texttt{A}(f_1)$ with $\texttt{B}(f_2) = \texttt{B}(Map(f_1))$, to associate $\texttt{A}(f_1 + s_1)$ with $\texttt{B}(f_2 + s_2) = \texttt{B}(Map(f_1 + s_1))$, to associate $\texttt{A}(f_1 + 2s_1)$ with $\texttt{B}(f_2 + 2s_2) = \texttt{B}(Map(f_1 + 2s_1))$, etc. Then processor S sends index $Map(y)$ if and only if $y \in S_1$ and $Map(y) \in S_2$. Thus the set of indices of B that processor S sends to processor D is the following set:

$$\Psi_S = S_2 \cap Map(S_1) = (Own_s(\texttt{B}) \cap (f_2 : l_2 : s_2)) \cap Map(Own_d(\texttt{A}) \cap (f_1 : l_1 : s_1)).$$

But since $Map(f_1 : l_1 : s_1) = (f_2 : l_2 : s_2)$, we can remove the $(f_2 : l_2 : s_2)$ term. Thus the intersection to compute is the following:

$$\Psi_S = Own_s(\texttt{B}) \cap Map(Own_d(\texttt{A}) \cap (f_1 : l_1 : s_1)). \tag{5}$$

The basic algorithm for computing the indices of the array elements to send is then the following, which can be translated directly into Fortran code:

```
DO j = 0, n3 - 1
   DO i = 0, n4 - 1
      (first, last, stride) = (f4 + i : l4 : s4) ∩ Map((f3 + j : l3 : s3) ∩ (f1 : l1 : s1))
```

This algorithm computes at most $2n_3n_4$ slice intersections. With a small change to the i and j loop bounds, fewer intersections might be required, since it is possible for i and j to loop over only those combinations for which the corresponding intersections are non-empty. In addition, since the strides and upper bounds of the slices are the same for all iterations of the loop, we can compute the resulting $last$ and $stride$ parameters outside the loop.

Local addresses

While the analysis above determines which global indices of the array must be sent, we also need to map each global index to the processor's local memory. This is the *local memory* function described previously.

One possible mapping is simply the identity mapping, where the global indices and the local indices are the same. However, this is quite wasteful of memory, especially when an array is distributed over many processors, since it

[2] When we refer to *indices* to be communicated, we actually mean the corresponding *array elements*.

necessitates that every processor have enough local memory to store all arrays in their entirety. The identity mapping would be an especially poor choice when we consider that 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 want to compact the array (i.e., put the blocks contiguous in memory). [9] argues 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:

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

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 6.

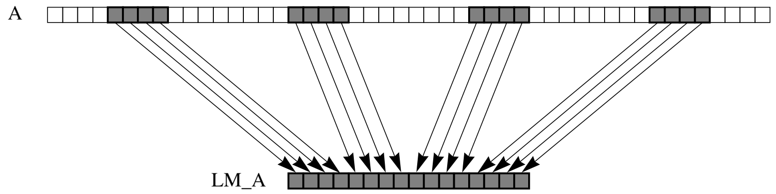

Figure 6: An example of the local memory mapping function for a particular processor. The processor owns global indices of A beginning with $f = 5$, with block size $n = 4$ and stride $s = 12$. These global indices are mapped to contiguous indices in local memory.

To calculate the local memory indices of the array elements to send, we simply apply the LM function to Ψ_S. Although the LM function seems expensive to apply to every element in Ψ_S, with a division, modulo, and multiplication operation, we can actually easily incorporate the LM function into the algorithm that computes Ψ_S, with little additional cost.

3.4 Communication: receiver

There are two problems that the compiler must solve for the receiving processor. The first problem is to determine which array elements it is supposed to receive. The second problem is to anticipate the order in which the sending processor sends the array elements, and to associate each sent element with the desired receiving index. The first problem can be addressed by using similar analysis for computing Ψ_S, which yields:

$$\Psi_R = Map^{-1}(S_2) \cap S_1 = Map^{-1}(Own_s(\mathtt{B}) \cap (f_2 : l_2 : s_2)) \cap (Own_d(\mathtt{A}) \cap (f_1 : l_1 : s_1)).$$

But since $Map^{-1}(f_2 : l_2 : s_2) = (f_1 : l_1 : s_1)$, we can remove the $(f_1 : l_1 : s_1)$ term. Thus the intersection to compute is the following:

$$\Psi_R = Map^{-1}(Own_s(\mathtt{B}) \cap (f_2 : l_2 : s_2)) \cap Own_d(\mathtt{A}).$$

To address the second problem, we simply ensure that we use the same looping structure as in the computation of Ψ_S. The corresponding code is:

```
DO j = 0, n_3 - 1
   DO i = 0, n_4 - 1
      (first, last, stride) = Map^{-1}((f_4 + i : l_4 : s_4) ∩ (f_2 : l_2 : s_2)) ∩ (f_3 + j : l_3 : s_3)
```

Once again, note that the computation of $last$ and $stride$ can be brought outside the loops. To compute the actual local memory indices, we simply apply the receiver's LM function, as in the case of the sender.

```
C Fill buffer, send to _dest_              | C Receive from processor _from_, unpack
  bufptr = 1                               |   CALL receive(buffer,from)
  DO loop0 = MAX(first_A(dest,1),1),       |   bufptr = 1
$        MIN(first_A(dest,1)+15,1024)      |   DO loop0 = 1, 16
    DO loop1 = 1, 16                       |     DO loop1 = MAX(first_B(from,1),1),
      buffer(bufptr) = B(loop1,loop0)      | $          MIN(first_B(from,1)+15,1024)
      bufptr = bufptr + 1                  |       A(loop0,loop1) = buffer(bufptr)
    ENDDO                                  |       bufptr = bufptr + 1
  ENDDO                                    |     ENDDO
  IF (bufptr.GT.1) THEN                    |   ENDDO
    CALL send(dest,buffer,(bufptr-1))      |
  ENDIF                                    |
```

Figure 7: Fortran code generated for $\mathtt{A}(1:n,1:n) = \mathtt{TRANSPOSE}(\mathtt{B}(1:n,1:n))$, compiled with 64 processors and $n = 1024$ (variable names edited for clarity).

3.5 Optimizations for special distributions

At compile time, we are likely to know in many cases whether the distribution of a 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 algorithm, rather than two. If neither array is block-cyclic, we can eliminate the i and j loops altogether.

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 algorithms, we need two outer loops, since equation (5) involves two block-cyclic distributions in the intersection.

Both block and cyclic ownership sets can be specified as a single slice. When f is the first index 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:

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

where E is the number of elements in the array dimension.

This gives more concise ways of defining $Own_d(\mathtt{A})$ and $Own_s(\mathtt{B})$ than in equations (3) and (4). In these representations, there is only a single slice, and not a union. This eliminates the need for one or both of the loops in the basic send and receive algorithms. Notice that there are nine sets of communication algorithms to derive, since each of the two arrays in the assignment statement can have one of three types of distributions.

Figure 7 shows the Fortran code generated by the compiler for a simple example, $\mathtt{A}(1:n,1:n) = \mathtt{TRANSPOSE}(\mathtt{B}(1:n,1:n))$, for $n = 1024$ and 64 processors. Both arrays have a block distribution in the first dimension, and are not distributed in the second dimension. Only the send and receive code fragments are shown; we omit the initialization and the details of the loop concerning the interleaving of the sends and receives. Timings for this code are given in Section 5.3.

4 Deposit model communication

Although the above algorithms produce efficient code to determine the set of data items to be sent or received, the fact still remains that on the order of $n_3 n_4$ multiplication, division, and modulo operations must be performed by both the sending and receiving processors. However, the sending and receiving algorithms are quite similar, and share many common subexpressions. This means that a significant fraction of the combined computation can be avoided if the sender does the computation for both itself and the receiver, avoiding the redundancy, and then passes the result on to the receiver. Although this requires more work for the sender, it usually pays off overall, since each processor tends to send to as many processors as it receives from.

One way to achieve the goal of letting the sender perform the computation for the receiver is to have the sender compute the array *indices* for the receiving processor. The indices can then be passed along with the data so that the sender can determine where to store the data. There are several disadvantages to such an implementation. The first is that the arrays may be k-dimensional, in which case an index vector of length k must be communicated along with a single data item (or data block), thus further increasing the effective communication load. The second disadvantage is that in the case of multidimensional arrays, the receiving processor must perform a set of additions and multiplications to map the index vector to a memory location. Normally, this is not a great disadvantage, since the actual destination memory address must be computed somewhere, regardless of whether the sender or the receiver computes it. However, when the sender generates the addresses in a loop, it can usually perform optimizations such as strength reduction to reduce the amount of address computation. In general, this cannot be done by a receiver that simply receives index vectors.

We can avoid these problems if the sender calculates the destination *memory addresses* for the receiver. Then the receiver can plug the data directly into memory when it arrives. We call this the *deposit model* of communication, since the sender is depositing the data directly into the final destination memory locations on the receiver. This is one example of the "door-to-door delivery" communication style [2], which moves data from the user's address space on one processor directly into the user's address space on another processor, without any intermediate copies to or from system space. Sending the addresses to the receiver minimizes the amount of work that is done by the receiver; it does not have to compute the index sets of the data items to be received. Since we are compiling to a single program which is run on all processors in SPMD fashion, the sending processor has complete knowledge of the receiver's memory space and can thus easily determine the corresponding destination memory locations. If, however, we compile under a model where the sender does not know the exact layout of the receiver's data space, the sender can simply compute relative offsets into the array rather than absolute memory locations, and the receiver can provide the base offset of the array.

Note that sending along address information for the deposit model increases the overall communication load in the system. This increase can range from minimal, when a single address is sufficient to label large blocks of data, to large, when each data item must be tagged with its destination memory address, in the form of address–data pairs. The usefulness of the deposit model depends on the ratio of communication/computation speed. For example, if the computation speed is much faster than the communication speed, the doubled cost of the communication may outweigh the time saved by the receiver not computing the destination addresses, and thus it would not be worthwhile to use the deposit model. However, the link bandwidths in parallel systems continue to increase (i.e., an iWarp node has a peak performance of 20 MFLOPS and a single iWarp link has a bandwidth of 40 MByte/sec; a Paragon XP/S node has a peak performance of 75 MFLOPS and a link bandwidth of 200 MByte/sec), so there exists a class of machines and systems that are an attractive target for this optimization[3]. We have measured the effects of the deposit model on one such system (the iWarp), and our experiments demonstrate that the deposit model pays off for a large class of communication patterns. Furthermore, it is simple for a compiler to detect those combinations of right-hand side distribution and left-hand side distribution that are not improved by the deposit model.

5 Performance results

In this section, we provide an empirical evaluation of the deposit model for compiler generated code (see [17] for more information about the compiler). The performance was measured in an implementation on a 64-node iWarp array, using a general-purpose message passing library for interprocessor communication. We base our discussion on the following examples:

[3]Link bandwidth is defined as the bandwidth over a single bus between two adjacent processors. Congestion affects communication between non-adjacent processors, an example of which is provided in the next section.

- Redistributions of one-dimensional arrays. These are measured by executing an assignment statement $\texttt{A}(1 : n) = \texttt{B}(1 : n)$, where each array has n elements. We provide data for 6 combinations of A and B having block, cyclic, and block-cyclic distributions, omitting data for the cases in which the two distributions are identical, which the compiler trivially recognizes as a simple local copy operation without any communication.
- One-dimensional shift. The assignment statement $\texttt{A}(1 : n - 1) = \texttt{B}(2 : n)$ is executed, where A and B share the same distribution. We tested block, cyclic, and block-cyclic distributions.
- Transposes of two-dimensional arrays. If we execute an assignment statement $\texttt{A}(1 : n, 1 : n) = \texttt{TRANSPOSE}(\texttt{B}(1 : n, 1 : n))$, where the first dimensions of A and B are distributed across the processors, and the other dimensions are not distributed, the program performs a transpose operation, invoking an all-to-all communication. We measured the three cases where both arrays are distributed in the same fashion (i.e., block, cyclic, or block-cyclic).

These examples are compiled in isolation, so the execution of different array statements is not overlapped at all. To see the effect of this optimization on a complete program, we measured the execution of a two-dimensional Fast Fourier Transform (2DFFT). The 2DFFT program incorporates the two-dimensional transpose operation.

5.1 One-dimensional redistribution

Figures 8 through 10 show the number of clocks for the execution of the assignment statement $\texttt{A}(1 : n) = \texttt{B}(1 : n)$, with various distributions for two single-precision arrays A and B, and different values of n. The communication statements for this array assignment are the same as the communication patterns occurring in any redistribution of any one-dimensional array. Three different distributions were chosen: block, cyclic, and block-cyclic with a block size of 2, and we measured all combinations of distributions for A and B. We do not report the cases in which A and B have the same distribution, because no communication is necessary, thus yielding identical code from the compiler for both situations. The array size varies in powers of 2 from 256 to 131072.

The results show that the deposit model always outperforms the standard model for one-dimensional redistributions, usually by a significant amount. However, if one array has a cyclic distribution and the other has a block-cyclic distribution, each processor receives messages from k other processors, where k the block size. In our case, $k = 2$, so the communication pattern is sparse, and the deposit provides little benefit over the standard method.

These experiments lead to another important conclusion: *for dense communication patterns, it is extremely important for the receiver to process incoming messages as quickly as possible.* Consider an all-to-all communication pattern. With $O(p^2)$ messages travelling through the network, at any given time many messages are likely to be blocked waiting on the completion of other messages. In addition, a receiver is able to process only a fixed number of messages at a time (one message for most systems). This means that the longer the receiver spends processing an incoming message, the longer other messages will have to wait, even messages destined for other receivers. The deposit model allows messages to be processed as quickly as possible, subject to the extra time it takes to transmit and receive address information in addition to data. A possible solution is for the receiver to buffer all messages and only process them after all messages have been received. However, this method requires $O(p)$ times more memory for buffers, and most large programs simply cannot afford to give up this amount of memory for communication.

5.2 One-dimensional shift

Figure 11 shows the time required for the execution of the assignment statement $\texttt{A}(1 : n-1) = \texttt{B}(2 : n)$, using the same distributions and problem sizes as for the one-dimensional redistribution case. We always chose the same distribution for both arrays, so that the communication follows a shift pattern. With this shift pattern, communication occurs only at block boundaries. While this minimizes the amount of communication for block distributions, it maximizes the communication for cyclic distributions, since every element is on a block boundary.

The communication time increases sharply as n crosses some threshold value. This effect is caused by an interaction between the message routing method, the hardware buffers, and the message sizes. When message sizes are sufficiently small, an entire message can fit into the hardware buffers while waiting for the receiver to become ready. This means that the sender can finish sending its message(s) and does not block on the receiver, and can thus finish any receive operations sooner. This effect propagates throughout the processors, allowing them all to complete quickly in the case of short messages. As expected, this effect shows up for smaller values of n if the deposit model is used, since both addresses and data values are transmitted.

We notice that the standard method usually outperforms the deposit model, especially in the cyclic case when the amount of data to transfer is large. In a shift operation, regardless of the distribution, the communication pattern is always sparse, thus minimizing the penalty paid for delaying the communication. Note that the results for the block distribution are nearly identical, which is not surprising given that only a single value is sent regardless of n.

5.3 Two-dimensional transpose

A statement with a more complicated communication pattern is the assignment statement $\mathtt{A}(1:n, 1:n) = \mathtt{TRANSPOSE}(\mathtt{B}(1:n, 1:n))$. Each array is of size $n \times n$ for $n \in \{128, 256, 512, 1024\}$. Both arrays are distributed in their first dimension, and the second dimensions are not distributed. For each case, the distributed dimensions have the same distribution: block, cyclic, or block-cyclic with a block size of 2. If the distributed dimensions are distributed over all the processors, this assignment statement will result in an all-to-all communication pattern.

Our measurements, depicted in Figure 12, show that the deposit model performs significantly better than the standard method for this example. Since the deposit model method processes the data more quickly as it arrives, the deposit model cuts down on the network congestion and speeds up the overall communication.

5.4 2DFFT

This two-dimensional FFT program works by performing a one-dimensional FFT (1DFFT) independently on each column of the $n \times n$ matrix, performing a transpose operation, and then performing another one-dimensional FFT independently on each column. We achieve the best parallelism by distributing the matrix by columns, and performing the 1DFFT computations on the columns with a `DOALL` loop. This means that each processor can independently perform a 1DFFT on the data it owns, without any communication except for the transpose operation. The two-dimensional transpose operation is implemented with the `TRANSPOSE` array assignment statement described above.

Figure 13 shows a scatter plot of the absolute performance for each communication method. An entry above the diagonal indicates that the deposit model performed better. We see that the deposit model provides a significant improvement over the standard communication model in all cases.

5.5 Discussion

Our tests have shown that the deposit model usually executes more quickly than the standard method, due both to the complexity in determining the array indices to send and receive, and to the increased speed with which the receiver can process incoming messages. But we also note that we can relatively easily generalize the types of array statements that perform better with the particular communication methods, simply by anticipating whether the resulting communication pattern will be sparse or dense. For example, we might instruct the compiler to use the standard communication method in the case of one-dimensional assignment statements where both arrays have the same distribution and access stride (since it invokes a sparse shift communication pattern), and to use the deposit model in the other cases. This choice will also depend somewhat on the underlying computation and communication architecture of the system. For example, as we increase the computation power of the processor relative to the communication bandwidth, eventually the deposit model will never pay off. However, most current private memory systems provide ample communication bandwidth and can easily tolerate an increase in the message size without a comparable increase in communication time. Once the startup overhead has been paid, sending extra words in a message is cheap. For such systems, the deposit model provides a low-cost optimization.

As with all measurements on a real system, there are several features of the system that influence these measurements. The message passing system used for these measurements is based on synchronous sends and receives (i.e., both processors must be explicitly calling the send and receive routines for communication to proceed). To avoid deadlock, this method requires a choice of routing and scheduling that results in extra congestion in the system. This choice of routing and scheduling influences the overall execution time, but we feel that these perturbations do not impact our assessment of the deposit model. (In fact, the longer messages of the deposit model may even be penalized more than the shorter messages of the default model.)

5.6 Related work

Compiling for private memory parallel systems is an active area of research; see [18] for a recent survey. Much work has focused on the alignment and distribution phase of the compilation system, e.g. automatic discovery of alignment information, and finding good data layouts or data distributions [11, 3, 10, 5, 14, 13, 7]. We have concentrated in this paper on the direct compilation of array statements, and our solution can be employed in the back end of other compilers. The value of sending (address, data) pairs has been discovered independently by other researchers (e.g., [4]) and is also used in other compilers (e.g., the CM Fortran compiler[15]). However, this is the first unified description and evaluation of a general algorithm for block, cyclic, and block-cyclic distributions.

6 Concluding remarks

Array statements are an attractive and powerful way for a user to specify the data parallelism in a program. When mapping an array statement onto a target private memory parallel system for a given distribution and alignment, the key task of the compiler is to generate the communication statements that provide local copies of non-local data as needed. The array assignment statement is the core statement, and solutions for this statement type can easily be extended to other array statements.

This paper introduced a novel representation of slices that does justice to the fact that real compilers do not translate array expressions in isolation but must deal with statements involving multiple operands. Similar code patterns are used to determine which elements to send or receive, but any such computation is overhead that reduces the speedup obtained from a parallel system. To keep the amount of overhead computation small and to reduce the overhead on the receiver, a compiler can exploit the similarity of determining sender and receiver information and compute this information together on the sender processor. Then the sender transmits (address, data) tagged blocks to the receiving processor. Using this method, our measurements on a 64-node iWarp system indicate savings up to 50% to 75%. Of course, the impact of this optimization depends on the specific data distribution and on the size of the data set. This optimization increases the amount of communication, but since most modern private memory parallel systems provide ample communication bandwidth, this optimization is quite worthwhile.

References

[1] G. Blelloch and J. Sipelstein. Collection-Oriented Languages. *Proc. IEEE*, 79(4):504–523, Apr 1991.

[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. Chatterjee. *Compiling Data-Parallel Programs for Efficient Execution on Shared-Memory Multiprocessors*. PhD thesis, School of Computer Science, Carnegie Mellon, Oct 1991.

[4] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating Local Addresses and Communication Sets for Data-Parallel Programs. In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 149–158, May 1993.

[5] S. Chatterjee, J. Gilbert, R. Schreiber, and S. Teng. Automatic Array Alignment in Data-Parallel Programs. In *Proc. 20th POPL Conf.*, pages 16–28. ACM, January 1993.

[6] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[7] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed Memory Compiler Methods for Irregular Problems - Data Copy Reuse and Runtime Partitioning. In *Proc. Third Workshop on Compilers for Parallel Computers*, pages 173–206, Vienna, Austria, July 1992. Austrian Center for Parallel Computation. ACPC/TR 92-8.

[8] S. K. S. Gupta, S. D. Kaushik, C. H. Huang, J.R. Johnson, R. W. Johnson, and P. Sadayappan. Towards Automatic Derivation of Data Distributions from Tensor Product Formulas. In *Proc. Third Workshop on Compilers for Parallel Computers*, pages 376–393, Vienna, Austria, July 1992. Austrian Center for Parallel Computation. ACPC/TR 92-8.

[9] S. Hiranandani, K. Kennedy, and C. W. Tseng. Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines. Technical Report COMP TR91-170, Dept. of Computer Science, Rice University, November 1991.

[10] S. Hiranandani, K. Kennedy, and C. W. Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *CACM*, 35(8):66–80, Aug 1992.

[11] K. Knobe, J. Lukas, and G. Steele Jr. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, Feb 1990.

[12] D. E. Knuth. *The Art of Computer Programming*, Volume 2. Addison-Wesley, 1981.

[13] C. Koelbel and P. Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE TPDS*, 2(4):440–451, Oct 1991.

[14] J. Ramanumjam and P. Sadayappan. Compile-Time Techniques for Data Distribution in Distributed Memory Machines. *IEEE Trans. Parallel and Distributed Systems*, 2(4):472–482, Oct 1991.

[15] G. Sabot. Internals of the CM Fortran 1.0 Compiler. Talk at Carnegie Mellon, Apr 1991.

[16] J. Stichnoth. Efficient Compilation of Array Statements for Private Memory Systems. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon, February 1993.

[17] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Programming Task and Data Parallelism on a Multicomputer. In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 13–22, May 1993.

[18] H. Zima and B. Chapman. Compiling for Distributed-Memory Systems. *Proc. IEEE*, 81(2):264–287, 1993.

[19] H. Zima, H. Bast, and M. Gerndt. SUPERB: A Tool for Semi-Automatic MIMD/SIMD Parallelization. *Parallel Computing*, 6:1–18, 1988.

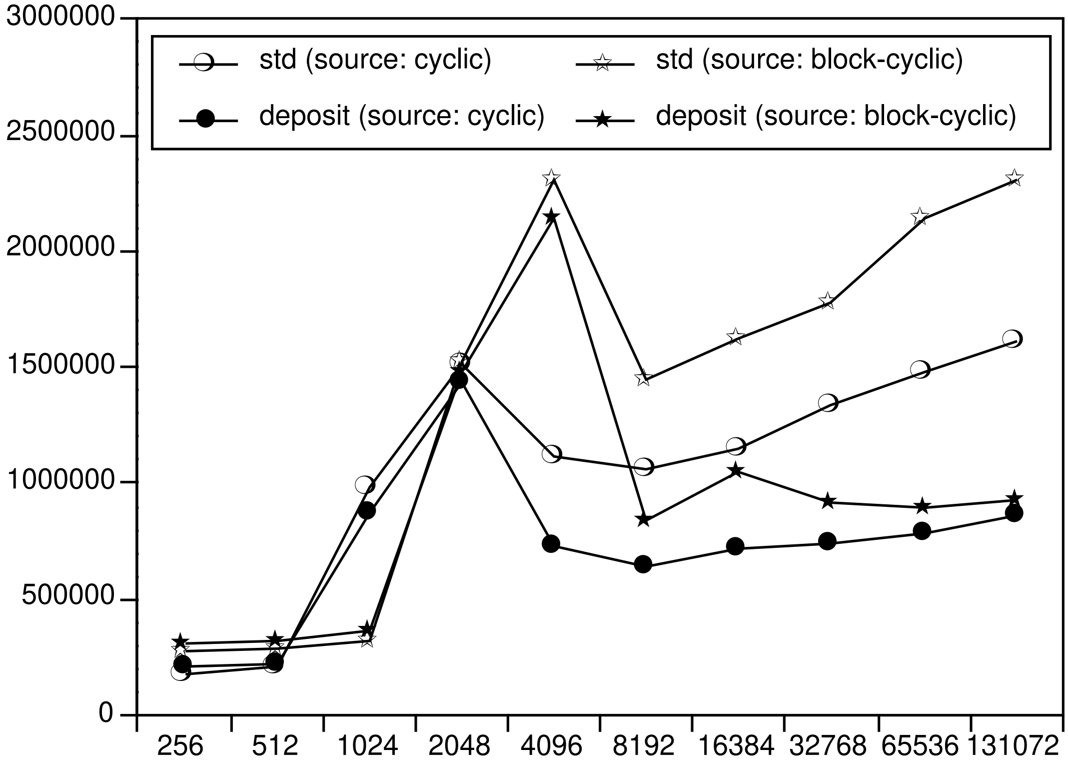

Figure 8: Execution time for $\mathtt{A}(1:n) = \mathtt{B}(1:n)$ with a *block* distribution for destination array A.

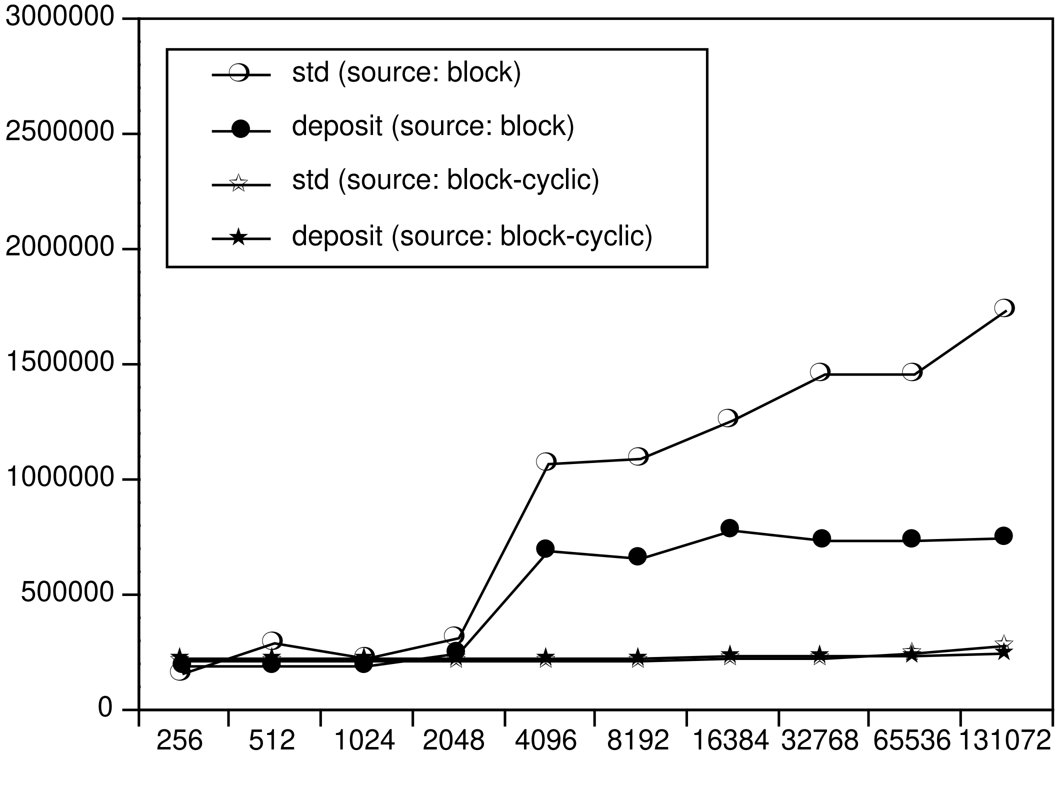

Figure 9: Execution time for $\mathtt{A}(1:n) = \mathtt{B}(1:n)$ with a *cyclic* distribution for destination array $\mathtt{A}$.

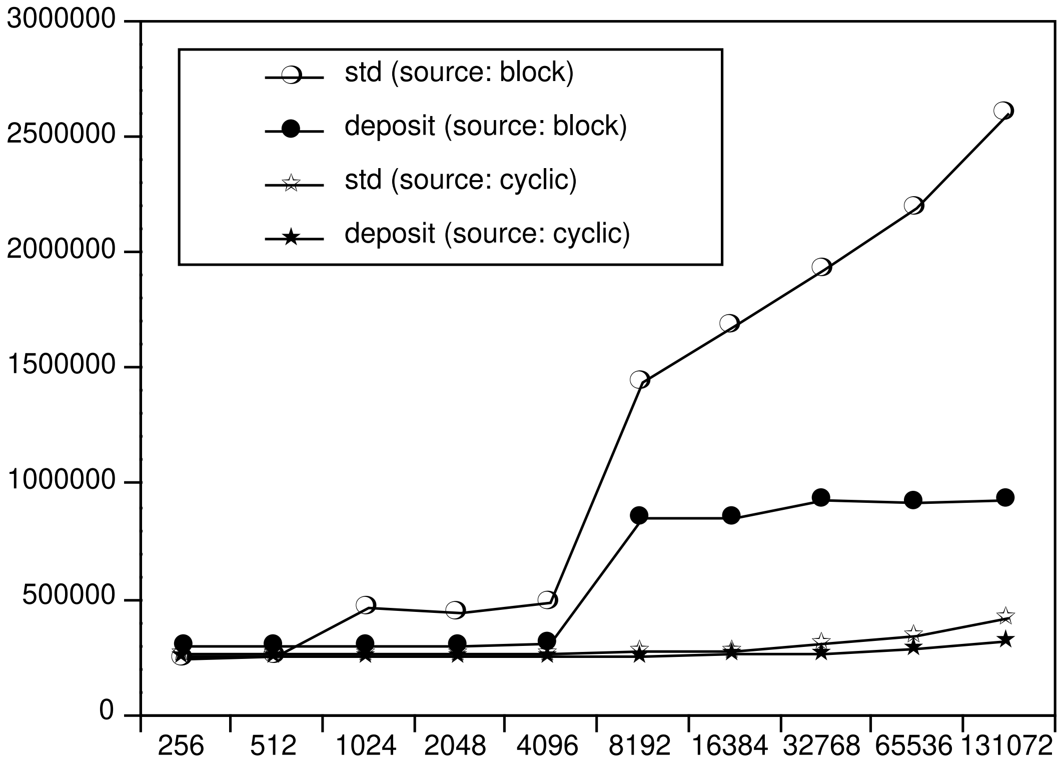

Figure 10: Execution time for $\mathtt{A}(1:n) = \mathtt{B}(1:n)$ with a *block-cyclic* distribution for destination array `A`.

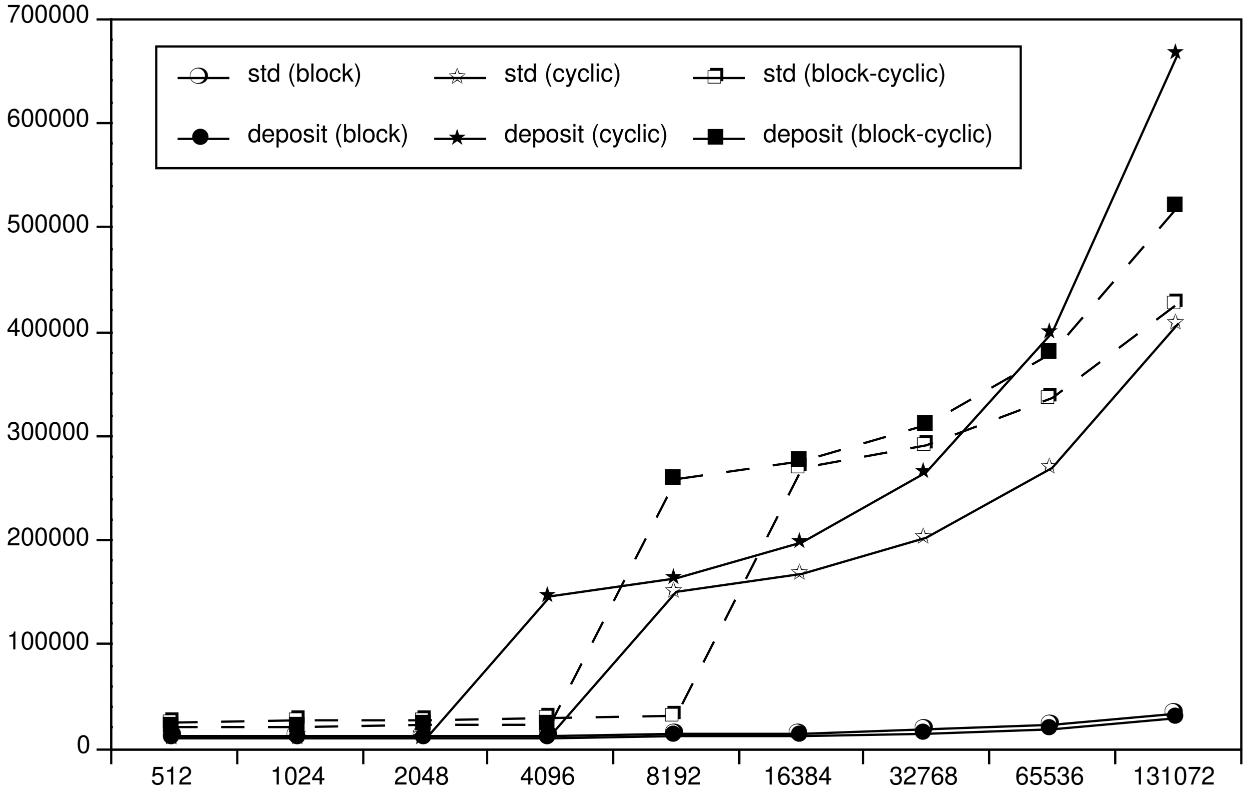

Figure 11: Execution time for $\mathtt{A}(1 : n-1) = \mathtt{B}(2 : n)$.

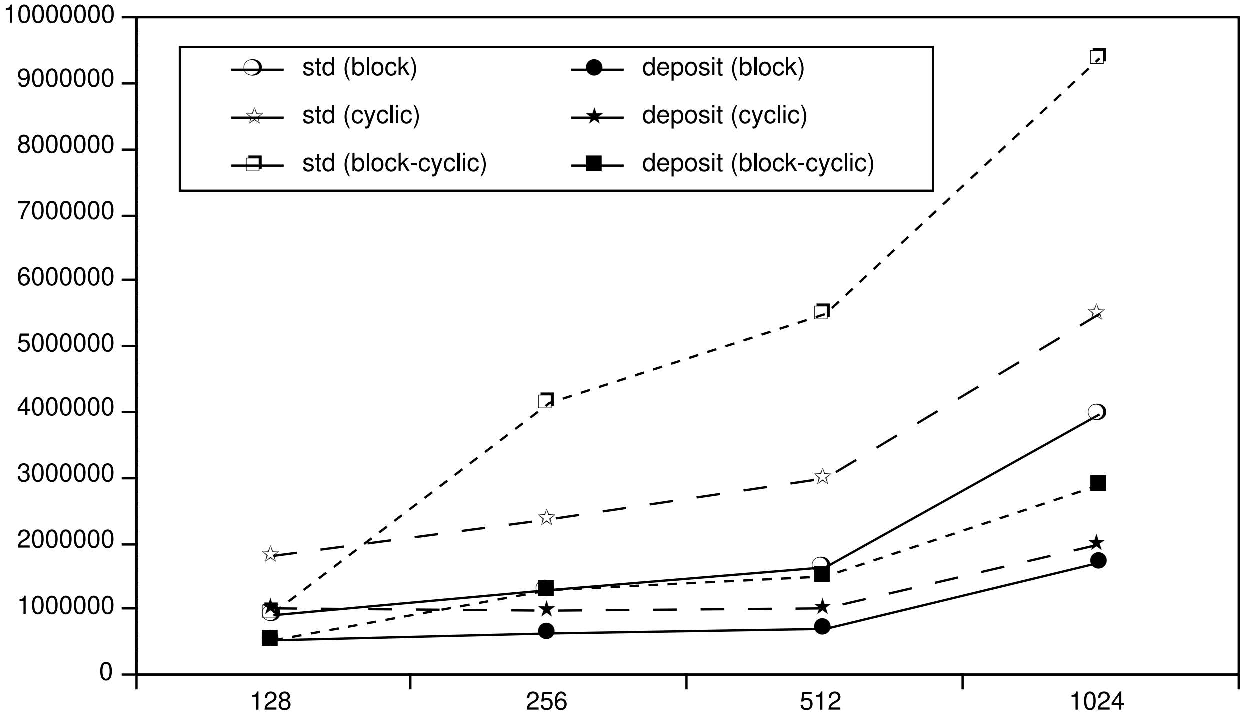

Figure 12: Execution time for $\mathtt{A}(1:n, 1:n) = \mathtt{TRANSPOSE}(\mathtt{B}(1:n, 1:n))$.

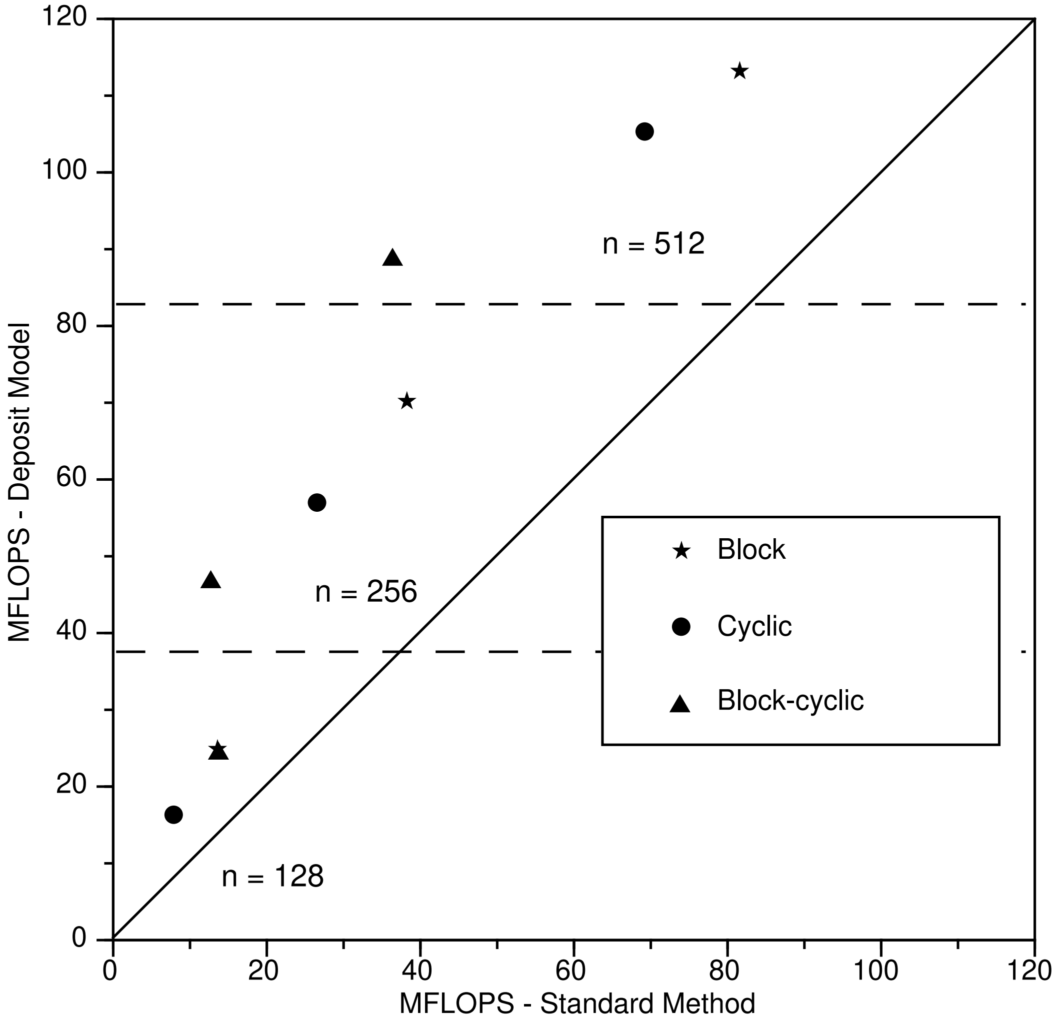

Figure 13: Execution rate comparison in MFLOPS for the 2DFFT.