Compiling Array Expressions for Efficient Execution on Distributed-Memory Machines

Technical Report OSU-CISRC-4/94-19(Revised)

S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan

Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210

Abstract

Array statements are often used to express data-parallelism in scientific languages such as Fortran 90 and High Performance Fortran. In compiling array statements for a distributed-memory machine, efficient generation of communication sets and local index sets is important. We show that for arrays distributed block-cyclically on multiple processors, the local memory access sequence and communication sets can be efficiently enumerated as closed forms using regular sections. First, closed form solutions are presented for arrays that are distributed using block or cyclic distributions. These closed forms are then used with a *virtual processor approach* to give an efficient solution for arrays with block-cyclic distributions. This approach is based on viewing a block-cyclic distribution as a block (or cyclic) distribution on a set of virtual processors, which are cyclically (or block-wise) mapped to physical processors. These views are referred to as *virtual-block* or *virtual-cyclic* views depending on whether a block or cyclic distribution of the array on the virtual processors is used. The virtual processor approach permits different schemes based on the combination of the virtual processor views chosen for the different arrays involved in an array statement. These virtualization schemes have different indexing overhead. We present a strategy for identifying the virtualization scheme which will have the best performance. Performance results on a Cray T3D system are presented for hand-compiled code for array assignments. These results show that using the virtual processor approach, efficient code can be generated for execution of array statements involving block-cyclically distributed arrays.

Index Terms: Array statement, compiler optimizations, data communication, distributed-memory machines, data distribution, High Performance Fortran, Fortran 90.

1 Introduction

Programming languages, such as High Performance Fortran (HPF) [5], Fortran D [6], Vienna Fortran [3], and Distributed Fortran 90 [13] support a programming model based on a single address space and provide directives for the explicit specification of data distributions for arrays. Block, cyclic, and block-cyclic distributions are the regular data distributions provided in these languages. Array statements involving array expressions are often used to express data-parallelism in these languages. Array expressions involve array sections which consist of array elements from a lower index to an upper index at a fixed stride. In order to generate high-performance target code, compilers for distributed-memory machines should produce efficient code for array statements involving distributed arrays.

Compilation of array statements with distributed arrays for parallel execution requires partitioning of the computation among the processors. Many compilers use the owner-computes rule to partition the computation in such a manner that the processor owning the array element performs the computation which modifies it. The computation performed on a processor may involve array elements resident on other processors. In the generated code, all the non-local data needed by a processor is fetched into temporary arrays in a processor's local memory using interprocessor communication. In order to reduce the communication overhead, each processor first determines all the data it needs to send to and receive from other processors and then performs the needed communication. This reduces the communication overhead by aggregating all the data movements needed from one processor to another into a single message. After communication, each processor then performs the computation in its local address space.

The overhead to perform the data movement consists of determination of the following data index sets and processor sets for each processor p:

- *Send processor set* of processor p: set of processors to which p has to send data.
- *Send data index set* of processor p to processor q: indices of the array elements which are resident on p but are needed by q.
- *Receive processor set* of processor p: set of processors from which p has to receive data.
- *Receive data index set* of processor p from processor q: indices of the array elements which are needed by p but are resident on q.

Closed form characterization of these sets would reduce the overhead of packing data into messages on the sending processor and unpacking data at the receiving processor. If the arrays have only block or cyclic distributions, then the data index sets and the processor sets can be characterized using regular sections for closed forms [7, 8, 12]. However, for the general block-cyclic distribution, closed form characterization of these sets using simple regular sections is not possible.

This paper presents a *virtual processor approach* to efficiently enumerate the data index sets and processor sets when arrays have block-cyclic distributions. This approach is based on viewing a block-cyclic distribution as a block (or cyclic) distribution on a set of virtual processors, which are cyclically (or block-wise) mapped to physical processors. These views are referred to as *virtual-block* or *virtual-cyclic* views depending on whether a block or cyclic distribution of the array on the virtual processors is used. These virtualization views permit us to use the closed forms for block and cyclic distributions in the virtual processor domain. A processor performs the computation for the virtual processors mapped to it. A message from processor p to q consists of all the data to be sent from the virtual processors on p to the virtual processors on q.

Under the owner computes rule, a processor owning the array element in an array section on the right hand side of the array assignment statement sends data to the processor owning the corresponding element of the array section on the left hand side of the assignment. We call the array section which is assigned the value as the *target* array section and an array section of the array expression on the right hand side of the array assignment statement as a *source* array section. For an array statement, each source and target array section pair is analyzed to determine the total data to be sent and received by each processor. The indexing overhead for a source and target array section pair depends upon the virtualization views used for each source and target array dimension. There are four possible combinations of virtual views per dimension as either a virtual block or a virtual cyclic view can be used for each array axis. Although each of the possible combinations involves exactly the same communication between any pair of physical processors, they will generally have different indexing overhead. We present a selection strategy to choose the virtualization scheme with the minimum indexing overhead.

We have implemented the virtual processor approach on a Cray T3D multicomputer. The performance measurements demonstrate that the indexing overheads for the four virtualization schemes are significantly different. The selection strategy provides a good prediction of the virtualization scheme with the lowest indexing cost. The additional overhead for generating the structures facilitating the enumeration of the index sets is in the range of 5% to 50% of the execution time for the array statement.

The rest of the paper is organized as follows. In Section 2, related work is presented. Section 3 describes the block, cyclic, and block-cyclic distributions along with their indexing functions and identifies the issues involved in efficient execution of array statements involving distributed arrays. In Section 4, we present closed form solutions in terms of regular sections for processors sets and data index sets for block-wise or cyclically distributed arrays. We then use the *virtual processor approach* to extend these solutions to block-cyclic distributions in Section 5. Performance results on a Cray T3D system are presented in Section 6. Conclusions and discussions are provided in Section 7.

2 Related Work

The issue of generating code for message passing machines from a single address space program with data distribution directives was addressed by Koelbel in [12], where a closed form characterization of the data index sets was provided for computations involving identically distributed arrays with either a block or a cyclic distribution. Closed form characterizations of the processor sets were not developed and arrays with block-cyclic distributions were not considered. Compilation of array statements in Distributed Fortran 90 is described in [13], but Distributed Fortran 90 supports only the block distribution.

For an array expression of the form $B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1))$ with the arrays distributed using block-cyclic distributions, Paalvast et al. [14] present techniques to enumerate the portion of B which is modified by a processor p and the portions of array A which reside on p but are needed by the other processors. Their communication scheme is based on scanning over the active array indices of array sections to determine the elements which need to be communicated. This scheme will incur a high run-time overhead since a local-to-global and a global-to-local translation will be needed for each active element in order to determine the destination processor.

The problem of active index-set identification for array statements involving block-cyclically distributed arrays was addressed by Chatterjee et al. [4] using a finite-state machine (FSM) to traverse the local index space of each processor. If all arrays in an array statement have the same block-cyclic distribution and access stride, the order of access of the local elements with the FSM approach turns out to be the same as the access order when the virtual-block view is taken with our approach. However, if the distribution or access stride of the array section on the left-hand-side is different from that of an array section on the right-hand-side, it appears that after determination of the active local indices of the r.h.s. section using a FSM, an explicit local-to-global translation corresponding to the r.h.s. section and a global-to-local translation corresponding to the l.h.s. section will need to be performed for each active element. In [4], restricted cases involving arrays with different strides are treated, but even with these, the generation of communication sets requires explicit index translation for each active element. With the virtual processor approach explicit local-to-global and global-to-local translation is not needed, even when the array sections differ in both distribution and access stride.

Recent independent work by Stichnoth also addresses the problem of active index-set and communication-set identification for array statements involving block-cyclically distributed arrays [16, 17]. The formulation proposed has similarities to the virtualization scheme with a virtual-cyclic view at both the source and target array. However, the approach in [17] does not attempt a closed form characterization of the active processor-sets with respect to each source/destination processor. Although the proposed scheme does not require the inclusion of indexing data in the message, a protocol in which a processor sends both the data values and the indices on the receiving processor, to facilitate unpacking, is proposed and evaluated. This strategy increases the data volume transmitted but improves performance on machines with high communication bandwidth such as the iWarp system.

The Syracuse F90-D compiler's initial implementation uses compile-time characterization of communication only for block distributions [1] and relies upon run-time generation of schedules for the general block-cyclic case using the approach adopted by the PARTI system [15].

The implementation of the Fortran-D compiler at Rice University is being extended to handle arrays with block-cyclic distributions [9, 10]. Their approach for determining communication sets is based upon computing the intersection of data index sets corresponding to the left-hand side and right-hand side array references in an assignment statement. The intersection is determined using a scanning approach which efficiently exploits the repetitive pattern of the intersection of the two index sets. An approach similar to the FSM approach [4] for determining the local memory access sequence is used. Efficient techniques for the FSM table generation are presented for special cases. The approach relies upon run-time methods [15] to handle the general case.

The use of the *virtual processor* approach for addressing the problem of active index-set identification for array statements involving block-cyclic distributions and that of redistribution of arrays with block-cyclic distributions was first reported by us in [7, 8]. This paper builds on that work and extends it in several ways: explicit closed-form characterization of processor sets using regular sections, development of a cost model to determine the virtualization scheme with lowest indexing time, and more extensive experimental verification of the efficacy of the approach.

Table 1: Index mapping for regular data distributions.

	Block	Cyclic	Cyclic(b)
local to global	$i = p(\lceil N/P \rceil) + l + m$	$i = lP + p + m$	$i = (l\ div\ b)bP + bp + l\ mod\ b + m$
global to local	$l = (i - m)\ mod\ \lceil N/P \rceil$	$l = (i - m)\ div\ P$	$l = ((i - m)\ div\ Pb)b + (i - m)\ mod\ b$
global to proc	$p = (i - m)\ div\ \lceil N/P \rceil$	$p = (i - m)\ mod\ P$	$p = ((i - m)\ div\ b)\ mod\ P$

$N = (n - m + 1)$; i global index $m \le i \le n$; l local index $0 \le l < \lceil b * (\lceil N/(Pb) \rceil) $; p processor $0 \le p < P$.

3 Data Distributions and Array Statements

In this section, we describe regular data distributions and compilation of array statements involving regularly distributed arrays. Consider an array $A(m : n)$ distributed onto P processors. In a *block* distribution, contiguous blocks of size $\lceil (n - m + 1)/P \rceil$ are allocated to the processors. In a *cyclic* distribution, array elements are assigned to the processors in a cyclic fashion. In a *block-cyclic* distribution, specified as $cyclic(b)$, blocks of size b are allocated to the processors in a cyclic fashion. The block-cyclic distribution is the most general regular data distribution in languages such as HPF. An element $A(i)$ of the array A has a *global* and a *local* index. The global index i is the index in array A, as referenced in an HPF program. The portion of array A located in a processor's local memory is referred to as A_loc in the processor's node program generated by an HPF compiler. The index of element $A(i)$ in A_loc is its local index. The relationships between the global index, the local index and the processor index for regular data distributions are shown in Table 1. A processor is said to *own* the elements of the distributed array which are allocated to it.

Let arrays $A(m_1 : n_1)$ and $B(m_2 : n_2)$ be distributed on P_1 and P_2 processors, respectively. Consider a simple form of the array assignment statement:

$$B(l_2 : u_2 : s_2) \quad = \quad f(A(l_1 : u_1 : s_1)).$$

The *array section* $A(l_1 : u_1 : s_1)$ consists of elements of A with indices $\{l_1 + i * s_1 | 0 \le i \le \lfloor (u_1 - l_1)/s_1 \rfloor\}$. The set of integers $(l : u : s)$ is referred to as a *regular section*. In the array statement, $B(l_2 + is_2)$ is assigned the value $f(A(l_1 + i * s_1))$. $A(l_1 + i * s_1)$ and $B(l_2 + i * s_2)$ are referred to as corresponding elements in the array assignment statement. The assignment can be executed independently for each pair of corresponding elements. The semantics of the array assignment require that the entire section $A(l_1 : u_1 : s_1)$ is read, the result of f on each element of the section evaluated and written to the corresponding element of $B(l_2 : u_2 : s_2)$. Semantically, this corresponds to copying $A(l_1 : u_1 : s_1)$ to a temporary array section Tmp, computing the function f on the elements of Tmp and assigning the resulting values to $B(l_2 : u_2 : s_2)$. The

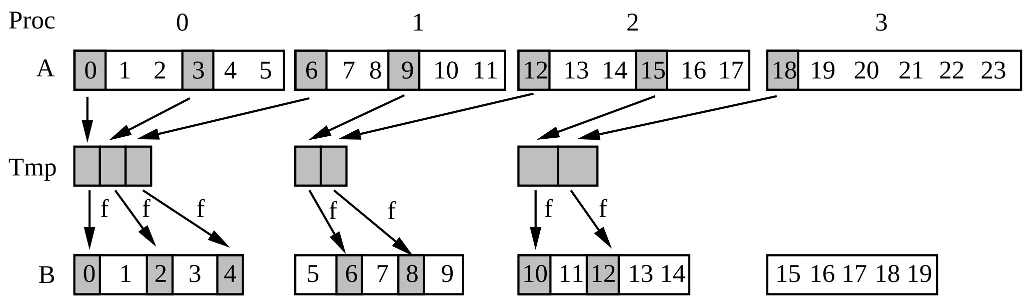

Figure 1: Illustration of execution of array statement $B(0:12:2) = f(A(0:18:3))$ for block distributed arrays $A(0:23)$ and $B(0:19)$ on four processors.

---

evaluation of the array assignment $B(0:12:2) = f(A(0:18:3))$ for block distributed arrays $A(0:23)$ and $B(0:19)$ on four processors, is illustrated in Fig. 1.

Since A and B are distributed, corresponding elements of $A(l_1 : u_1 : s_1)$ and $B(l_2 : u_2 : s_2)$ may not be located on the same processor. With the owner-computes rule the processor that owns an element of $B(l_2 : u_2 : s_2)$ is responsible for computing its value. Some elements of $A(l_1 : u_1 : s_1)$ will be communicated to the processor owning the corresponding elements of $B(l_2 : u_2 : s_2)$. Thus, for the array section $A(l_1 : u_1 : s_1)$, each processor $p,\ 0 \le p < P_1$ has to evaluate the following information for every other processor $q,\ 0 \le q < P_2$.

- $DSend(p, q)$: the set of local indices of the data elements that p owns and are required by q. Processor p has to send this set of elements to processor q.

Similarly, for the array section $B(l_2 : u_2 : s_2)$, each processor $p,\ 0 \le p < P_2$ has to evaluate the following information for every other processor $q,\ 0 \le q < P_1$.

- $DRecv(p, q)$: the set of local indices of the data elements that p needs and are located on q. Processor p will receive this set of data elements from q.
- $LIndex(p)$: the set of local indices of the elements of $B(l_2 : u_2 : s_2)$ that p owns. Processor p has to compute new values for this set of elements.

For instance, in Fig. 1, the local index set for processor 0 corresponding to the array section $B(0 : 12 : 2)$ is $LIndex(0) = (0 : 4 : 2)$, while that for processor 1 is $LIndex(1) = (1 : 4 : 2)$. Processor 0 has to receive data corresponding to the global section $A(6 : 6)$ from processor 1. Thus, $DSend(1, 0) = (0 : 0)$

/* Sending phase */
for $q \in PSend(p)$
    pack $A\_loc(DSend(p,q))$; send to q;

/* Receiving and execution phase */
for $q \in PRecv(p)$
    recv Tmp from q; $B(DRecv(p,q)) = f(Tmp(0 : |DRecv(p,q)| - 1))$

Figure 2: Node program on processor p, for $B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1))$.

---

and symmetrically $DRecv(0,1) = (4:4)$. Note that $DSend$ and $DRecv$ are expressed in terms of the local indices on the corresponding processors and in general $DSend(p,q) \neq DSend(q,p)$. Two additional sets of processor indices are defined as follows.

- $PSend(p)$: the set of processors $\{q \mid DSend(p,q) \neq \emptyset,\ 0 \le q < P_2\}$, $0 \le p < P_1$.
- $PRecv(p)$: the set of processors $\{q \mid DRecv(p,q) \neq \emptyset,\ 0 \le q < P_1\}$, $0 \le p < P_2$.

Using these sets the node program pseudo-code for the execution of the array statement is as shown in Fig. 2. In the program the receiving phase is combined with the execution phase. Rather than unpacking a message into a local temporary location and separately evaluating the function f on each element of the local section, f is applied on each element directly from the message buffer. This optimization reduces the overhead of an additional loop to traverse the local elements of the array sections and allows communication-computation overlap which is useful to hide communication latency. However, this optimization may not be applicable when the array expression has multiple array sections on the right hand side. In this case, the receiving and execution phase are performed separately. The sending and receiving phase is executed for each right hand side array section, storing the result in temporary arrays which are identically distributed and aligned as the left hand side array. The execution phase evaluates the values for the elements of the left hand side array section using the Local Index Set $LIndex()$ to access these elements, and obtains the required input elements from the appropriate temporary arrays. Since the data and processor send and receive sets and the local index sets are integral to the execution of array assignments, efficient schemes for enumerating these sets are important.

For block and cyclic distributions, the send and receive data index and processor sets can be expressed as regular sections. However, for block-cyclic distributions, these sets cannot be expressed as simple regular sections. For instance, consider the local index set of $A(0:39:2)$ when $A(0:39)$ is distributed onto

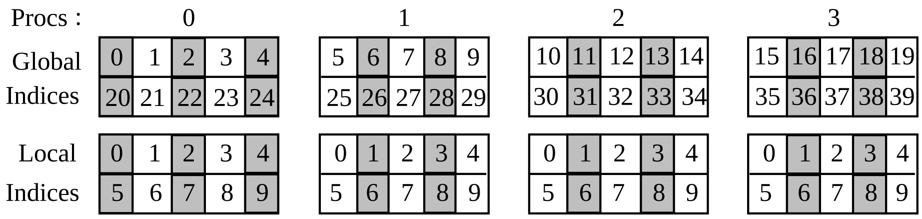

Figure 3: Variable stride in A_loc for global array section $A(0 : 40 : 2)$.

four processors using a $cyclic(5)$ distribution, as shown in Fig. 3. The array section elements are separated by a constant stride 2 in global index space. However, the elements of the array section do not have a constant stride in the local index space of a processor. For example, $LIndex(0) = (0, 2, 4, 5, 7, 9)$ and $LIndex(1) = (1, 3, 6, 8)$.

We now construct the data send and receive sets and processor send and receive sets for block and cyclically distributed arrays.

4 Communication Sets for Array Expressions

Consider two arrays $A(m_1 : n_1)$ and $B(m_2 : n_2)$ distributed using a block or cyclic data distribution and the array statement $B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1))$. We now develop closed form expressions for the send and receive processor and data index sets. In the following sections, $((l : u : s) * c - d)$ is defined as the regular section $(l * c - d : u * c - d : s * c)$. Also $((l : u : s)\ op\ c)$, where $op \in \{mod, div\}$, consists of the integers $\{(l + i * s)\ op\ c \mid 0 \leq i \leq \lfloor \frac{u-l}{s} \rfloor\}$. We initially derive the closed forms for the cases when $s_1 > 0$ and $s_2 > 0$. We then describe modifications to handle negative strides.

4.1 Block Distribution to Block Distribution

Let A and B be block distributed over P_1 and P_2 processors, respectively. The block size of array A is $b_1 = \lceil (n_1 - m_1 + 1)/P_1 \rceil$ and the block size of array B is $b_2 = \lceil (n_2 - m_2 + 1)/P_2 \rceil$.

Let processor p be the sending processor and $A(l_1 + i_1 s_1)$ and $A(l_1 + j_1 s_1)$ be the first and last array elements of $A(l_1 : u_1 : s_1)$ allocated to p. We obtain i_1 and j_1 as follows:

$$i_1 = \lceil max(m_1 + pb_1 - l_1, 0)/s_1 \rceil, \qquad j_1 = \lfloor min(m_1 + pb_1 + b_1 - 1 - l_1, u_1 - l_1)/s_1 \rfloor,$$

where $m_1 + pb_1$ and $m_1 + pb_1 + b_1 - 1$ are the global indices of the first and last elements of array A allocated

Figure 4: Determination of closed forms for $PSend(p)$ and $DSend(p,q)$ for block to block case.

---

to p. The send processor set of p includes processors that own elements in $B(l_2 + i_1 s_2 : l_2 + j_1 s_2 : s_2)$. Note that the send processor set is empty if $i_1 > j_1$.

The closed form of the send processor set must include a processor index exactly once. The first and last processors that p will send data to are $(l_2 + i_1 s_2 - m_2)\ div\ b_2$ and $(l_2 + j_1 s_2 - m_2)\ div\ b_2$, respectively. We consider two cases. If the stride s_2 is less than the block size b_2, p will send at least one element to processor q, where $(l_2 + i_1 s_2 - m_2)\ div\ b_2 \leq q \leq (l_2 + j_1 s_2 - m_2)\ div\ b_2$. For instance, Fig. 4 illustrates an example in which $s_2 < b_2$. If $s_2 \geq b_2$, processor p will send exactly one element to each receiving processor. Thus we have

$$PSend(p) \quad = \quad \begin{cases} ((l_2 + i_1 s_2 - m_2)\ div\ b_2 : (l_2 + j_1 s_2 - m_2)\ div\ b_2) & \text{if } i_1 \leq j_1 \wedge s_2 < b_2, \\ ((i_1 : j_1) * s_2 + l_2 - m_2)\ div\ b_2 & \text{if } i_1 \leq j_1 \wedge s_2 \geq b_2, \\ \emptyset & \text{if } i_1 > j_1. \end{cases}$$

Let $B(l_2 + i_2 s_2)$ and $B(l_2 + j_2 s_2)$ be the first and last elements of $B(l_2 : u_2 : s_2)$ allocated to processor q. We obtain i_2 and j_2 as follows:

$$i_2 = \lceil max(m_2 + qb_2 - l_2, 0)/s_2 \rceil, \qquad j_2 = \lfloor min(m_2 + qb_2 + b_2 - 1 - l_2, u_2 - l_2)/s_2 \rfloor.$$

The global indices of the first and last element sent from processor p to q are $l_1 + max(i_1, i_2) * s_1$ and $l_1 + min(j_1, j_2) * s_1$, respectively. Converting global indices to local indices on processor p we have:

$$DSend(p,q) \quad = \quad (l_1 + max(i_1, i_2) * s_1 - m_1 - pb_1 : l_1 + min(j_1, j_2) * s_1 - m_1 - pb_1 : s_1).$$

For the receiving phase of processor p, the receive processor and data index sets can be determined similarly. Let i_1 and j_1 be the first and last slice indices of A on processor q:

$$i_1 = \lceil max(m_1 + qb_1 - l_1, 0)/s_1 \rceil, \qquad j_1 = \lfloor min(m_1 + qb_1 + b_1 - 1 - l_1, u_1 - l_1)/s_1 \rfloor,$$

and i_2 and j_2 to be the first and last indices of B on processor p:

$$i_2 = \lceil max(m_2 + pb_2 - l_2, 0)/s_2 \rceil, \qquad j_2 = \lfloor min(m_2 + pb_2 + b_2 - 1 - l_2, u_2 - l_2)/s_2 \rfloor.$$

Then, the closed forms are as follows:

$$\begin{array}{rcl} PRecv(p) & = & \left\{ \begin{array}{ll} ((l_1 + i_2 s_1 - m_1)\ div\ b_1 : (l_1 + j_2 s_1 - m_1)\ div\ b_1) & \text{if } i_2 \leq j_2 \wedge s_1 < b_1, \\ ((i_2 : j_2) * s_1 + l_1 - m_1)\ div\ b_1 & \text{if } i_2 \leq j_2 \wedge s_1 \geq b_1, \\ \emptyset & \text{if } i_2 > j_2, \end{array} \right. \\ DRecv(p,q) & = & (l_2 + max(i_1, i_2) * s_2 - m_2 - pb_2 : l_2 + min(j_1, j_2) * s_2 - m_2 - pb_2 : s_2). \end{array}$$

Node code for processor p developed using the closed forms is shown in Fig. 5. Nonblocking sends and blocking receives are the communication primitives used. Fig. 5 shows the most general form of the code. It can be optimized by removing the unnecessary checks if some of the parameters are known at compile-time. In the sending phase, processor p uses the closed-forms for $DSend(p,q)$ to send messages to all processors $q \in PSend(p)$. In the receiving phase, processor p first determines the number of messages it will receive using $PRecv(p)$. Then, it receives the messages non-deterministically in the order they arrive. The receive command $recv(q, tmp\_buf)$ returns the sender processor's index in q and the message in tmp_buf. The recv data index set $DRecv(p,q)$ is used to unpack the received message. If $p \in PSend(p)$, then the explicit message send and receive is replaced by a local memory copy of the buffers.

4.2 Block Distribution to Cyclic Distribution

Let A be block distributed and B be cyclically distributed over P_1 and P_2 processors, respectively. The block size of array A is $b_1 = \lceil (n_1 - m_1 + 1)/P_1 \rceil$ and the number of cycles for array B is $C_2 = \lceil (n_2 - m_2 + 1)/P_2 \rceil$. Let $A(l_1 + i_1 s_1)$ and $A(l_1 + j_1 s_1)$ be the first and last array elements of $A(l_1 : u_1 : s_1)$ located on processor p. We have

$$i_1 = \lceil max(m_1 + pb_1 - l_1, 0)/s_1 \rceil, \qquad j_1 = \lfloor min(m_1 + pb_1 + b_1 - 1 - l_1, u_1 - l_1)/s_1 \rfloor.$$

If $B(l_2 + i_1 s_2)$ is located on processor q, then the next element of $B(l_2 : u_2 : s_2)$ located on q is $B(l_2 + i_1 s_2 + lcm(s_2, P_2))$. Furthermore, all the intermediate elements of $B(l_2 : u_2 : s_2)$, i.e., $B(l_2 + i_1 s_2 : l_2 + i_1 s_2 + lcm(s_2, P_2) - s_2 : s_2)$ must be located on distinct processors. Thus, the send processor set consists of processors owning elements $B(l_2 + i_1 s_2 : l_2 + i_1 s_2 + lcm(s_2, P_2) - s_2 : s_2)$, i.e.,

$$PSend(p) \quad = \quad (l_2 + (i_1 : min(j_1, i_1 + lcm(s_2, P_2)/s_2 - 1)) * s_2 - m_2)\ mod\ P_2$$

For example, Fig. 6 illustrates an example in which $P_2 = 3$ and $s_2 = 2$. It can be seen that the stride

```
/* Sending phase */
i_1 = ⌈max(m_1 + p * b_1 - l_1, 0)/s_1⌉
j_1 = ⌊min(m_1 + p * b_1 + b_1 - 1 - l_1, u_1 - l_1)/s_1⌋
if i_1 ≤ j_1 ∧ s_2 < b_2 then
    do q = (l_2 + i_1 * s_2 - m_2) div b_2, (l_2 + j_1 * s_2 - m_2) div b_2
        cnt = 0
        i_2 = ⌈max(m_2 + q * b_2 - l_2, 0)/s_2⌉
        j_2 = ⌊min(m_2 + q * b_2 + b_2 - 1 - l_2, u_2 - l_2)/s_2⌋
        do r = l_1 + max(i_1, i_2) * s_1 - m_1 - p * b_1, l_1 + min(j_1, j_2) * s_1 - m_1 - p * b_1, s_1
            tmp_buf_out(cnt) = A_loc(r)
            cnt = cnt + 1
        enddo
        send(q, tmp_buf_out, cnt)
    enddo
else if i_1 ≤ j_1 ∧ s_2 ≥ b_2 then
    do k_1 = i_1, j_1
        cnt = 0
        q = (k_1 * s_2 + l_2 - m_2) div b_2
        i_2 = ⌈max(m_2 + q * b_2 - l_2, 0)/s_2⌉
        j_2 = ⌊min(m_2 + q * b_2 + b_2 - 1 - l_2, u_2 - l_2)/s_2⌋
        r = l_1 + max(i_1, i_2) * s_1 - m_1 - p * b_1
        tmp_buf_out(cnt) = A_loc(r)
        cnt = cnt + 1
        send(q, tmp_buf_out, cnt)
    enddo
endif

/* Receiving phase /*
i_2 = ⌈max(m_2 + p * b_2 - l_2, 0)/s_2⌉
j_2 = ⌊min(m_2 + p * b_2 + b_2 - 1 - l_2, u_2 - l_2)/s_2⌋
if i_2 ≤ j_2 ∧ s_1 < b_1 then
    msg = (l_1 + j_2 * s_1 - m_1) div b_1 - (l_1 + i_2 * s_1 - m_1) div b_1
else if i_2 ≤ j_2 ∧ s_1 ≥ b_1 then
    msg = j_2 - i_2
else  msg = -1 endif
while msg ≥ 0 do
    recv(q, tmp_buf_in)
    cnt = 0
    i_1 = ⌈max(m_1 + q * b_1 - l_1, 0)/s_1⌉
    j_1 = ⌊min(m_1 + q * b_1 + b_1 - 1 - l_1, u_1 - l_1)/s_1⌋
    do r = l_2 + max(i_1, i_2) * s_2 - m_2 - p * b_2, l_2 + min(j_1, j_2) * s_2 - m_2 - p * b_2, s_2
        B_loc(r) = f(tmp_buf_in(cnt))
        cnt = cnt + 1
    enddo
    msg = msg - 1
enddo
```

Figure 5: Node code for $B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1))$. $A(m_1 : n_1)$ and $B(m_2 : n_2)$ are block distributed on P_1 and P_2 processors, respectively.

Figure 6: Determination of closed forms for $PSend(p)$ for block to cyclic case ($P_2 = 3,\ s_2 = 2$).

distance between the section elements on any processor is $lcm(s_2, P_2)$ $(= 6)$.

For each processor q in $PSend(p)$, there exists a corresponding first slice index k_1 in $(i_1 : min(j_1, i_1 + lcm(s_2, P_2))/s_2 - 1)$. The indices of all slices sent by p to q can be determined as $(k_1 : j_1 : lcm(s_2, P_2)/s_2)$. Since $DSend(p, q)$ consists of the local indices of these slices on processor p, we have

$$\begin{aligned} DSend(p,q) &= l_1 + (k_1 : j_1 : lcm(s_2, P_2)/s_2) * s_1 - m_1 - pb_1 \\ &= (l_1 + k_1 * s_1 - m_1 - p * b_1 : l_1 + j_1 * s_1 - m_1 - p * b_1 : lcm(s_2, P_2) * s_1/s_2). \end{aligned}$$

We now consider the receive processor set of p. The elements of array B located on processor p are $B(m_2 + p : n_2 : P_2)$. Processor p may contain elements of the array section $B(l_2 : u_2 : s_2)$ only if the Diophantine equation $l_2 + is_2 = m_2 + p + cP_2$ has a solution, i.e., $gcd(s_2, P_2)$ divides $(p + m_2 - l_2)$ [2, 18]. In this case, p may receive array elements of A from other processors. To determine the first array slice located on p, we solve $i_2 s_2 - c_2 P_2 = m_2 + p - l_2$ using Euclid's Extended GCD algorithm [2, 11]. Let i_2 and c_2 be the solution such that i_2 is the smallest non-negative integer for which the corresponding c_2 is also a non-negative integer. The first slice index is i_2 and the last slice index j_2 is computed as follows:

$$j_2 = i_2 + \lfloor (u_2 - l_2 - i_2 s_2)/lcm(s_2, P_2) \rfloor * lcm(s_2, P_2)/s_2.$$

The indices of slices of $B(l_2 : u_2 : s_2)$ located on processor p are $(i_2 : j_2 : lcm(s_2, P_2)/s_2)$. To determine the receive processor set, we need to compare the block size b_1 with the stride distance between two consecutive elements of $A(l_1 : u_1 : s_1)$ that are sent to processor p. The stride distance is $s_1 * lcm(s_2, P_2)/s_2$, as shown in Fig. 7. If the stride distance is less than b_1, then p will receive data elements from processors $((i_2 s_1 + l_1 - m_1)\ div\ b_1 : (j_2 s_1 + l_1 - m_1)\ div\ b_1)$. If the stride is not less than b_1, then p will receive one

Figure 7: Determination of stride distances for evaluating $PRecv(p)$ for block to cyclic case ($P_2 = 3,\ s_2 = 2$).

data element each from the processors in $((i_2 : j_2 : lcm(s_2, P_2)/s_2) * s_1 + l_1 - m_1)\ div\ b_1$. Thus, we have:

$$PRecv(p) = \begin{cases} ((l_1 + i_2 s_1 - m_1)\ div\ b_1 : (j_2 s_1 + l_1 - m_1)\ div\ b_1) & \\ & \text{if } i_2 \le j_2 \wedge s_1 * lcm(s_2, P_2)/s_2 < b_1, \\ (l_1 + (i_2 : j_2 : lcm(s_2, P_2)/s_2) * s_1 - m_1)\ div\ b_1 & \\ & \text{if } i_2 \le j_2 \wedge s_1 * lcm(s_2, P_2)/s_2 \ge b_1, \\ \emptyset & \text{if } i_2 > j_2. \end{cases}$$

To determine the receive data index set of processor p from q, we need the first and last slice indices i_r and j_r, respectively, of the data index set. These indices are computed as below:

$$\begin{aligned} i_r &= i_2 + \lceil max(m_1 + qb_1 - l_1 - i_2 s_1, 0) s_2/(s_1 * lcm(s_2, P_2)) \rceil * lcm(s_2, P_2)/s_2, \\ j_r &= i_2 + \lfloor (m_1 + qb_1 + b_1 - 1 - l_1 - i_2 s_1) s_2/(s_1 * lcm(s_2, P_2)) \rfloor * lcm(s_2, P_2)/s_2. \end{aligned}$$

Thus, the receive data index set of processor p from q is:

$$DRecv(p, q) = ((l_2 + i_r s_2 - m_2)\ div\ P_2 : (l_2 + j_r s_2 - m_2)\ div\ P_2 : lcm(s_2, P_2)/P_2).$$

Node code for processor p, can be similarly developed using these closed forms.

4.3 Cyclic Distribution to Block Distribution

The case when $A(m_1 : n_1)$ is cyclically distributed and $B(m_2 : n_2)$ is block distributed is the dual of the previous case where A was block distributed and B cyclically distributed. The send processor and send data index sets of the former become the receive processor and receive data index sets of the latter. Similarly the receive sets of the former become the send sets of the latter.

4.4 Cyclic Distribution to Cyclic Distribution

We now consider the case where arrays A and B are cyclically distributed over P_1 and P_2 processors, respectively. The section indices i_1 and j_1 of the first and last element of $A(l_1 : u_1 : s_1)$ located on processor p are determined as follows. Let i_1 and c_1 be the solution to the Diophantine equation $i_1 s_1 - c_1 P_1 = m_1 + p - l_1$ such that i_1 is the smallest non-negative integer for which the corresponding c_1 is also a non-negative integer. Let $r_1 = lcm(s_1, P_1)$. Then index j_1 is:

$$j_1 \quad = \quad i_1 + \lfloor (u_1 - l_1 - i_1 s_1)/r_1 \rfloor * r_1/s_1.$$

Let $t_2 = lcm(r_1 * s_2/s_1, P_2)$. The send processor set is defined as below:

$$PSend(p) \quad = \quad (l_2 + (i_1 : min(j_1, i_1 + t_2/s_2 - 1) : r_1/s_1) * s_2 - m_2) \ mod \ P2$$

For each processor q in $PSend(p)$, there exists a corresponding first slice index k_1 in $(i_1 : min(j_1, i_1 + t_2/s_2 - 1) : r_1/s_1)$. The indices of all slices sent to q can be determined as $(k_1 : j_1 : t_2/s_2)$. However, the data index send set of processor p to processor q is defined as the set of local indices on p of those slices of array A:

$$\begin{aligned} DSend(p,q) &= (l_1 + (k_1 : j_1 : t_2/s_2) * s_1 - m_1) \ div \ P_1 \\ &= ((l_1 + k_1 * s_1 - m_1) \ div \ P_1 : (l_1 + j_1 * s_1 - m_1) \ div \ P_1 : t_2 * s_1/(s_2 P_1)) \end{aligned}$$

Determining the communication sets for the receiving phase for cyclic distribution to cyclic distribution is a dual of the problem of the sending phase. Let i_2 and c_2 be the solution to the Diophantine equation $i_2 s_2 - c_2 P_2 = m_2 + p - l_2$ such that i_2 is the smallest non-negative integer for which the corresponding c_2 is also non-negative. Let $r_2 = lcm(s_2, P_2)$ and $t_1 = lcm(r_2 * s_1/s_2, P_1)$. Then we have:

$$\begin{aligned} j_2 &= i_2 + \lfloor (u_2 - l_2 - i_2 s_2)/r_2 \rfloor * r_2/s_2, \\ k_2 &\in (i_2 : min(j_2, i_2 + t_1/s_1 - 1) : r_2/s_2) \\ PRecv(p) &= (l_1 + (i_2 : min(j_2, i_2 + t_1/s_1 - 1) : r_2/s_2) * s_1 - m_1) \ mod \ P1 \\ DRecv(p,q) &= (l_2 + (k_2 : j_2 : t_1/s_1) * s_2 - m_2) \ div \ P_2 \\ &= ((l_2 + k_2 * s_2 - m_2) \ div \ P_2 : (l_2 + j_2 * s_2 - m_2) \ div \ P_2 : t_1 * s_2/(s_1 P_2)) \end{aligned}$$

Node code for processor p, can be similarly developed using these closed forms.

4.5 Modifications for Negative Strides

We now describe the modifications to the closed form expressions to handle negative strides. If both $s_1 < 0$ and $s_2 < 0$ then the array statement $B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1))$ is equivalent to the array statement $B(u_2' : l_2 : -s_2) = f(A(u_1' : l_1 : -s_1))$ where $u_2' = u_2 + (l_2 - u_2)\ mod\ s_2$ and $u_1' = u_1 + (l_1 - u_1)\ mod\ s_1$. Since $-s_1 > 0$ and $-s_2 > 0$, the communication sets for the array statement $B(u_2' : l_2 : -s_2) = f(A(u_1' : l_1 : -s_1))$ can be evaluated using the techniques presented in the previous sections.

The cases when s_1 and s_2 differ in sign are slightly more complicated as the direction in which the slice indices of the source and the target sections increase are different. Consider the case when $s_1 < 0$ and $s_2 > 0$. Closed form expressions for the communication sets for these cases are developed by inverting the numbering of the slice indices of section $A(l_1 : u_1 : s_1)$, noting that the slice index i_1 for the array section $A(l_1 : u_1 : s_1)$ corresponds to the slice index $k - i_1$ in the array section $B(l_2 : u_2 : s_2)$, where $k = \lfloor \frac{u_2 - l_2}{s_2} \rfloor$ is the total number of slice indices in the array sections, and then following arguments similar to those in the previous sections. For instance, consider the case when both A and B are distributed using a block distribution. Let processor p be the sending processor and $A(l_1 + i_1 s_1)$ and $A(l_1 + j_1 s_1)$ be the first and last elements of $A(l_1 : u_1 : s_1)$ allocated to p. We obtain i_1 and j_1 as follows: $i_1 = \lceil max(m_1 + pb_1 - u_1', 0)/(-s_1) \rceil$, $j_1 = \lfloor min(m_1 + pb_1 + b_1 - 1 - u_1', l_1 - u_1)/(-s_1) \rfloor$, where u_1' is as defined above. The send processor set of p includes processors that own elements in $B(l_2 + (k - j_1)s_2 : l_2 + (k - i_1)s_2 : s_2)$. Note that the slice indices in the array section $B(l_2 : u_2 : s_2)$ corresponding to i_1 and j_1 are $k - i_1$ and $k - j_1$, respectively. Thus depending on the relation between the stride s_2 and the block size b_2 we have

$$PSend(p) \quad = \quad \begin{cases} ((l_2 + (k - j_1)s_2 - m_2)\ div\ b_2 : (l_2 + (k - i_1)s_2 - m_2)\ div\ b_2) & \text{if } i_1 \le j_1 \wedge s_2 < b_2, \\ (((k - j_1) : (k - i_1)) * s_2 + l_2 - m_2)\ div\ b_2 & \text{if } i_1 \le j_1 \wedge s_2 \ge b_2, \\ \emptyset & \text{if } i_1 > j_1. \end{cases}$$

The data send set $DSend(p, q)$ for $q \in PSend(p)$ can be similarly evaluated.

5 Virtual Processor Approach for Block-Cyclic Distributions

In this section, we present a *virtual processor approach* for efficient execution of array statements involving block-cyclically distributed arrays. Let $A(m_1 : n_1)$ and $B(m_2 : n_2)$ be distributed using a $cyclic(b_1)$ and $cyclic(b_2)$ distribution on P_1 and P_2 processors, respectively. For an array statement of the form $B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1))$, the virtual processor approach involves:

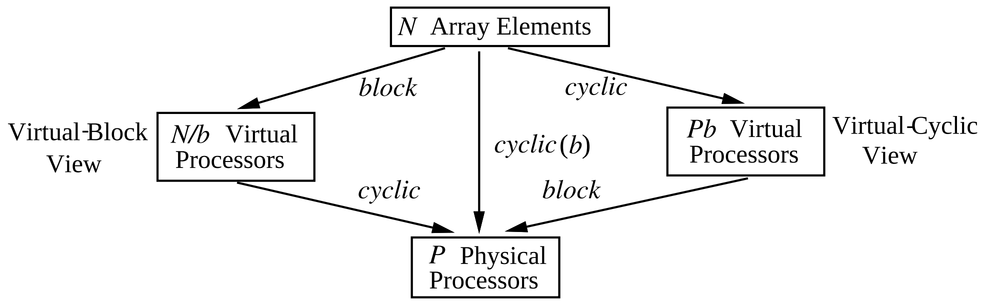

Figure 8: Virtual views of a block-cyclic distribution.

Table 2: Virtualization schemes using virtualized views for source and target distributions.

Source Distribution Virtual View	Target Distribution Virtual View	
	Virtual-Block	Virtual-Cyclic
Virtual-Block	Block to Block	Block to Cyclic
Virtual-Cyclic	Cyclic to Block	Cyclic to Cyclic

1. Viewing a *cyclic*(b_1) distribution of A as a block (or cyclic) distribution on VP_1 virtual processors which are cyclically (or block-wise) mapped to P_1 processors. These views are referred to as *virtual-block* or *virtual-cyclic* views depending on whether a block or cyclic distribution of the array on the virtual processors is used. Fig. 8 gives an schematic illustration of the two views of a block-cyclic distribution. The *cyclic*(b_2) distribution of B is similarly viewed as a block or cyclic distribution on VP_2 virtual processors.

2. The communication required to perform the array statement can be determined by using the closed forms of Section 4 in the virtual processor domain. Each physical processor bears the responsibility of of performing the computation and communication for the virtual processors mapped to it.

Thus the virtual processor approach is characterized by a two-level mapping of array elements to physical processors. The first level maps the array elements to virtual processors and the second level maps virtual processors to physical processors. The mapping at each level can be represented using simple regular sections which facilitates efficient implementation of this approach.

We now present the details of the virtualization schemes. Depending upon the virtualization views used for the source array A and the target array B, four different schemes for executing the array statement are possible. The four schemes are shown in Table 2. Each scheme is associated with a different number of source and target virtual processors, a different communication pattern in the virtual processor domain, and incurs different indexing overheads. Hence, we also present a strategy to select the communication scheme with minimum indexing overhead.

5.1 Virtualization Views

We now describe the virtual cyclic and virtual block views of a block-cyclic distribution.

5.1.1 Virtual-Block View

Let array $A(m : n)$ be distributed using a $cyclic(b)$ distribution on P processors. In the *virtual-block view*, A is assumed to be block distributed on $VP = \lceil (n - m + 1)/b \rceil$ virtual processors. These virtual processors are assigned to P processors in a cyclic fashion. The set of virtual processors on processor p is $(p : VP : P)$. Fig. 9 illustrates the virtual-block view of a $cyclic(2)$ distribution of $A(0 : 15)$ on two processors. The array has a block distribution on eight virtual processors v_0 to v_7, which are cyclically allocated to the two processors p_0 and p_1, i.e., v_0, v_2, v_4 and v_6 are mapped to p_0, while v_1, v_3, v_5 and v_7 are mapped to p_1. Processor p_0 will perform the computation and communication for virtual processors v_0, v_2, v_4 and v_6 while processor p_1 will perform the same for virtual processors v_1, v_3, v_5 and v_7.

Using closed form expressions developed in Section 4, data index sets can be evaluated in terms of the local indices of the virtual processors. However, the local index of an element on a virtual processor is not the same as its local index on the processor to which it is mapped. For instance, in Fig. 9, element $A(8)$ has a local index of 4 in A_loc on processor p_0, but a local index of 0 on virtual processor v_4. Since the physical processor p performs the computation and communication for the virtual processor v mapped to it, it is necessary to determine the translation from the virtual processor's local index space to the physical processor's index space. If the virtual processor v is mapped to processor p, then the array element with local index j on v has a local index $(v \; div \; P) * b + j$ on processor p. Under the virtual-block view, a stride of s in the local index space of a virtual processor on processor p remains unchanged in the local index space of p. Hence, an array section $(l : u : s)$ of A in the local space of a virtual processor v on processor p corresponds to array section $((v \; div \; P) * b + l : (v \; div \; P) * b + u : s)$.

Consider the array section $A(l : u : s)$. Under the virtual-block view not all virtual processors necessarily own elements of the array section. We refer to virtual processors that own array section elements as being *active*. The active virtual processors are determined as follows. If $s \leq b$, then each virtual processor $v \in ((l - m) \; div \; b : (u - m) \; div \; b)$ has at least one element of the array section. Let $v_l = (l - m) \; div \; b$ and

Figure 9: Virtual-block view of array $A(0:15)$ with $cyclic(2)$ distribution on two processors.

$v_u = (u-m)\ div\ b$. Since the set of virtual processors on processor p is $(p : VP : P)$, the active virtual processors on p for array A, denoted by $VAct\_A(p)$, are given by

$$VAct\_A(p) \quad = \quad (v_l : v_u) \cap (p : VP : P) = (max(v_l + (p - v_l)\ mod\ P, p) : min(v_u, VP) : P)$$

If $s > b$ then each active virtual processor has exactly one element of the array section. The active virtual processors are given by $(((l-m) : (u-m) : s)\ div\ b) \cap (p : VP : P)$. Each active virtual processor in $((l-m) : (u-m) : s)\ div\ b$ is scanned and a check performed to determine if it is located on processor p.

5.1.2 Virtual-Cyclic View

Under the *virtual-cyclic* view, array $A(m:n)$ distributed using a $cyclic(b)$ distribution is assumed to have a cyclic distribution on $VP = min(P * b, n - m + 1)$ virtual processors. These virtual processors are block distributed on the P processors. Thus, the set of virtual processors on a processor p are $(p * b : min(p * b + b - 1, n - m + 1))$. Fig. 10 illustrates the virtual-cyclic view of a $cyclic(2)$ distribution of $A(0:15)$ on two processors. The array has a cyclic distribution on four virtual processors. These four virtual processors are allocated to the two processors in a block fashion, i.e., v_0 and v_1 are mapped to p_0, while v_2 and v_3 are mapped to p_1.

The array element with local index j on a virtual processor v, has a local index $(v\ mod\ b) + b * j$ on the processor to which it is mapped. A stride of s in the local index space of a virtual processor corresponds to $s * b$ in the processor's local index space. Thus the array section $(l : u : s)$ in the local index space of a virtual processor v is mapped to the section $((v\ mod\ b) + b * l : (v\ mod\ b) + b * u : s * b)$ in the local index space of the processor it is mapped to.

Consider the array section $A(l : u : s)$. The active virtual processors on a processor p for the array section are determined as follows. The elements of A located on a virtual processor v have indices $(v + m : n : P * b)$. Thus v is active if the intersection $(v + m : n : P * b) \cap (l : u : s)$ is not empty. Hence, a virtual processor v is

Figure 10: Virtual-cyclic view of array $A(0:15)$ with $cyclic(2)$ distribution on two processors.

active if $gcd(s, P*b) | (v+m-l)$* and the first element of the intersection lies in the array section. The first element can be found by solving the Diophantine equation $i*s - c*P*b = v+m-l$. Let i_1 and c_1 be the solution such that i_1 is the smallest non-negative integer for which the corresponding c_1 is non-negative. If $l + i_1 * s \leq u$ then the virtual processor v is active. If the first active virtual processor on processor p is v_f, then the active virtual processors on p are

$$VAct\_A(p) \quad \subseteq \quad (v_f : min(u-m, p*b+b-1) : gcd(s, P*b))$$

Note that not every processor in $(v_f : min(u-m, p*b+b-1) : gcd(s, P*b))$ will have an element of the array section. The first virtual processor v_f is given by $v_f = ((-p*b-m+l) \ mod \ gcd(s, P*b)) + p*b$.

5.2 Virtualization Schemes

We now present virtualization schemes which use virtualized views for the source and target arrays and the communication sets developed in Section 4 for block and cyclic distributions, to determine the communication sets for $cyclic(b)$ distributions. The send and receive processor sets and data index sets for a virtual processor v_p, which is mapped to processor p are defined as follows:

$$\begin{aligned} VPSend(v_p) &= \text{Set of virtual processors } \{v\}, \text{ to which } v_p \text{ sends data.} \\ VPRecv(v_p) &= \text{Set of virtual processors } \{v\}, \text{ from which } v_p \text{ receives data.} \\ VDSend(v_p, v_q) &= \text{Local indices on processor } p \text{ of the data elements, which } v_p \text{ sends to } v_q. \\ VDRecv(v_p, v_q) &= \text{Local indices on processor } p \text{ of the data elements which } v_p \text{ receives from } v_q. \end{aligned}$$

$VPSend(v_p)$ and $VPRecv(v_p)$ are obtained from $PSend()$ and $PRecv()$ assuming a block or cyclic distribution on the virtual processors. Similarly, $VDSend(v_p, v_q)$ and $VDRecv(v_p, v_q)$ are obtained from $DSend()$

*For integers a, b, a divides b is denoted as $a|b$.

```
/* Sending phase for processor p */
for v_p ∈ VAct_A(p)
    for v_q ∈ VPSend(v_p)
        Pack A_loc(VDSend(v_p, v_q)) in tmp_out
        Send (Phy_B(v_q), tmp_out)
```

```
/* Receiving phase for processor p */
for v_p ∈ VAct_B(p)
    for v_q ∈ VPRecv(v_p)
        recv (Phy_A(v_q), tmp_in)
        B_loc(VDRecv(v_p, v_q)) = f(tmp_in)
```

Figure 11: Communication phases in the virtual processor domain.

and $DRecv()$. Note that $VDSend()$ and $VDRecv()$ are defined in terms of the local index space of the physical processor and a translation from the virtual processor local index space to physical index space will be required. Node pseudo-code for execution of the array statement $B(l_2 : u_2 : s_2) = f(A(l_1 : u_1 : s_1))$ using the processor sets and data index sets in the virtual processor domain is shown in Fig. 11. $Phy\_A(v)$ denotes the processor to which virtual processor v is mapped under the virtual view for the array $A(m_1 : n_1)$. Similarly $Phy\_B(v)$ denotes the processor to which virtual processor v is mapped under the virtual view for the array $B(m_2 : n_2)$.

The node pseudo-code in Fig. 11 could be inefficient as it may involve sending multiple messages between two processors. The additional message startup overhead can be reduced by splitting the send and receive phase into two phases. First, $PSend(p)$ and $DSend(p,q)$ are evaluated by scanning the active virtual processors and determining the processor and data send and receive index sets. These sets are defined as follows:

$$
\begin{aligned}
PSend(p) &= \bigcup_{v_p \in VAct\_A(p)} \Big(\bigcup_{v_q \in VPSend(v_p)} Phy\_B(v_q) \Big) \\
PRecv(p) &= \bigcup_{v_p \in VAct\_B(p)} \Big(\bigcup_{v_q \in VPRecv(v_p)} Phy\_A(v_q) \Big) \\
DSend(p,q) &= \bigcup_{v_p \in VAct\_A(p)} \Big(\bigcup_{v_q \in (VAct\_B(q) \cap VPSend(v_p))} VDSend(v_p, v_q) \Big) \\
DRecv(p,q) &= \bigcup_{v_p \in VAct\_B(p)} \Big(\bigcup_{v_q \in (VAct\_A(q) \cap VPRecv(v_p))} VDRecv(v_p, v_q) \Big)
\end{aligned}
$$

where $VAct\_A(p)$ denotes the set of active virtual processors on processor p corresponding to the section $A(l_1 : u_1 : s_1)$ and $VAct\_B(p)$ denotes the set of active virtual processors on processor p corresponding to the section $B(l_2 : u_2 : s_2)$. The node pseudo-code using these sets is as shown in Fig. 12. The initialization of the appropriate sets is not shown in the figure. The first phase evaluates the processor and data index send and receive sets. Note that the Phase two packs and sends the data to each processor in $PSend(p)$, receives messages from all processors $q \in PRecv(p)$ and evaluates the new values for B using $DRecv(p,q)$.

```
/* Sending Phase 1 for processor p */
for v_p ∈ VAct_A(p)
    for v_q ∈ VPSend(v_p)
      q = Phy_B(v_q)
      PSend(p) = PSend(p) ∪ {q}
      DSend(p, q) = DSend(p, q) ∪ VDSend(v_p, v_q)
    endfor
endfor

/* Sending Phase 2 for processor p */
for q ∈ PSend(p)
    pack(DSend(p, q), tmp_out)
    send(q, tmp_out)
endfor
```

```
/* Receiving phase 1 for processor p */
for v_p ∈ VAct_B(p)
    for v_q ∈ VPRecv(v_p)
      q = Phy_A(v_q)
      PRecv(p) = PRecv(p) ∪ {q}
      DRecv(p, q) = DRecv(p, q) ∪ VDRecv(v_p, v_q)
    endfor
endfor

/* Receiving Phase 2 for processor p */
for q ∈ PRecv(p)
    recv(q, tmp_in}
    B_loc(DRecv(p, q)) = f(tmp_in)
endfor
```

Figure 12: Template for node code with virtual processor approach.

The message contains all data communicated from processor p to q, i.e., data from all active virtual processors on p to their target virtual processors on q. To facilitate unpacking, the data is packed in increasing order of target virtual processor index. Data for a particular target virtual processor from all its source virtual processors on processor p is stored in the increasing order of source virtual processor index.

5.3 Strategy for Selection of Virtualization Schemes

Given block-cyclic distributions for $A(m_1 : n_1)$ and $B(m_2 : n_2)$, four choices are available for the virtualization scheme used, as shown in Table 2. The choice of the virtualization scheme depends on the additional indexing overhead per processor. Since the communication pattern between physical processors is identical for all four schemes, the scheme with lowest indexing overhead has the lowest total completion time. During the send phase, a processor p packs all the data in $DSend(p, q)$, $\forall q \in PSend(p)$. As shown in Section 5.2, $DSend(p, q)$ is a union of array sections. Associated with each array section is the time for evaluation of the loop bounds and the loop overhead. Assuming that this overhead is nearly equal for each section, the total indexing cost per processor during the sending phase, $t_s(p)$, is proportional to the total number of array sections to be communicated.

$$t_s(p) \quad = \quad t_i * \left[\sum_{v \in VAct\_A(p)} |VPSend(v)| \right] \leq t_i * VAct\_A_{max} * VPSend_{max}$$

where $VAct\_A_{max} = max_{0 \leq p < P1}(|VAct\_A(p)|)$, $VPSend_{max} = max_{0 \leq v_p < VP_1}(|VPSend(v_p)|)$, VP_1 is the total number of virtual processors corresponding to the virtual view of the source distribution, and t_i is a measure of the indexing overhead per section. Similarly for the total indexing cost per processor during the

Table 3: Send and receive processor set size estimates for various source and target distributions.

	$VPSend_{max}$	$VPRecv_{max}$
Block to Block	$\frac{b_1 * s_2}{b_2 * s_1}$ if $s_2 < b_2$ $\frac{b_1}{s_1}$ if $s_2 \geq b_2$	$\frac{b_2 * s_1}{b_1 * s_2}$ if $s_1 < b_1$ $\frac{b_2}{s_2}$ if $s_1 \geq b_1$
Block to Cyclic	$min(\frac{b_1}{s_1}, \frac{r_2}{s_2})$	$\frac{c_2 * P_2 * s_1}{b_1 * s_2}$ if $\frac{s_1 * r_2}{s_2} < b_1$ $\frac{c_2 * P_2}{r_2}$ if $\frac{s_1 * r_2}{s_2} \geq b_1$
Cyclic to Block	$\frac{c_1 * P_1 * s_2}{b_2 * s_1}$ if $\frac{s_2 * r_1}{s_1} < b_2$ $\frac{c_1 * P_1}{r_1}$ if $\frac{s_2 * r_1}{s_1} \geq b_2$	$min\left(\frac{b_2}{s_2}, \frac{r_1}{s_1}\right)$
Cyclic to Cyclic	$min\left(\frac{c_1 * P_1}{r_1}, \frac{lcm\left(r_1 * \frac{s_2}{s_1}, P_2\right) * s_1}{r_1}\right)$	$min\left(\frac{c_2 * P_2}{r_2}, \frac{lcm\left(r_2 * \frac{s_1}{s_2}, P_1\right) * s_2}{r_2}\right)$

$c_1 = N_1/(P_1 b_1),\ c_2 = N_2/(P_2 b_2),\ r_1 = lcm(s_1, P_1),\ r_2 = lcm(s_2, P_2)$

---

receiving phase, $t_r(p)$, we have

$$t_r(p) \quad = \quad t_i * \left[\sum_{v \in VAct\_B(p)} |VPRecv(v)| \right] \leq t_i * VAct\_B_{max} * VPRecv_{max}$$

where $VAct\_B_{max} = max_{0 \leq q < P2}(|VAct\_B(q)|)$, $VPRecv_{max} = max_{0 \leq v_q < VP_2} |VPRecv(v_q)|$ and VP_2 is the total number of virtual processors corresponding to the target distribution. Since t_i is a constant, $VAct\_A_{max} * VPSend_{max} + VAct\_B_{max} * VPRecv_{max}$ is used as the cost estimate of the indexing overhead.

Using the closed forms in Section 4 approximate measures for $VPSend_{max}$ and $VPRecv_{max}$ can be obtained. These measures are shown in Table 3. The source array of size N_1 is distributed on P_1 processors using a $cyclic(b_1)$ distribution and the target array of size N_2 is distributed on P_2 processors using a $cyclic(b_2)$ distribution. Approximate measures for the maximum number of virtual processors per physical processor are $VAct_{max} = \frac{N}{P * max(s,b)}$, for the virtual-block view and $VAct_{max} = \frac{b}{gcd(s, Pb)}$ for the virtual-cyclic view. Thus given a source and target block-cyclic distribution, an approximation of the maximum indexing cost can be obtained for each of the four virtualization schemes and a choice of the scheme best suited for the given distributions can be made.

6 Performance Results

In this section, we present experimental results for the virtual processor approach. The experiments were performed on a 32-node Cray T3D. The node programs used the PVM message-passing library. The time required by each processor to execute the node program for the array assignment statement was measured and the maximum time among all processors reported. Times were measured using the rtclock() wall clock timer.

The goals of the performance measurement were to determine whether the indexing costs for the four virtualization schemes were significantly different and whether the developed heuristic provided a good indication of the scheme with lowest indexing cost. Further, an estimate of the table generation costs and a comparison of the table generation costs with the execution time for the array statement was desired. Also a verification of the premise that the scheme with the lowest indexing cost also has the lowest execution time and table generation time was desired.

To reduce the number of independent parameters, we measured times for array section assignments of the form $B(0 : s_1 * M - 1 : s_1) = A(0 : s_2 * M - 1 : s_2)$. Double precision arrays A and B are distributed using $cyclic(b_1)$ and $cyclic(b_2)$ distributions, respectively. The parameter M gives the number of active elements in each array section. For our experiments we kept the number of active elements M fixed at 230400 and $P = 32$ processors were used. The experiment was performed for three data sets categorized by the relative values of s_1 and s_2:

(a) Both the source and target array sections have unit strides, i.e. $s_1 = s_2 = 1$. This case corresponds to array redistribution.

(b) Both the source and target array sections have the same non-unit stride, $s_1 = s_2 > 1$.

(c) The source and target array sections have different strides, i.e. $s_1 \neq s_2$.

Table 4 presents the indexing cost estimates (IE) for the four virtualization schemes and the corresponding indexing times (IT). The indexing time includes the time for copying the data into and out of message buffers and does not include the time spent for communication. In Table 4, vb-vb refers to the virtual block to virtual block scheme, vb-vc refers to the virtual block to virtual cyclic scheme, vc-vb refers to the virtual cyclic to virtual block scheme and vc-vc refers to the virtual cyclic to virtual cyclic scheme. Table 4(a), 4(b), and 4(c) give the indexing cost estimates and indexing times for the data sets (a), (b), and (c), respectively.

For the last three cases in each of the data sets, either the source or the target distribution is chosen to be a block distribution.

Comparing the indexing estimates and the indexing times for the scheme with lowest indexing cost indicates that the selection criteria developed in Section 5 provide a good prediction of the best indexing scheme. In all the measured cases, the "predicted best" scheme has the lowest indexing time. Furthermore, the relative order among the indexing cost estimates is the same as that between the indexing times for the four schemes. For schemes with the same predicted cost those having a virtual cyclic view for either the source or target distribution have a higher indexing time. This can be attributed to the fact that the non-unit stride between consecutive elements in the data index sets for the virtual cyclic view results in poor cache performance due to a decrease in spatial locality.

Phase two uses tables generated during phase one of the virtual processor approach as shown in Fig. 12. Table 5 gives the execution times and the table generation times for the cases shown in Table 4. The execution times reported are for phase two of the virtual processor approach and include the indexing time as well as the communication time. The best table generation and execution times are marked in Table 5. Comparing the lowest indexing times in Table 4 and the lowest execution times in Table 5 it is observed that the virtualization scheme which has the lowest indexing cost also has the lowest execution time. This follows by noting that the communication pattern among the processors for all four schemes is identical. Thus the communication time for the four schemes is nearly equal and the scheme with lowest indexing time also has the lowest execution time. Comparing the lowest execution times and lowest table generation times in Table. 5, it is further observed that the scheme with the lowest execution time also has the lowest table generation time. This can be attributed to the fact that the table generation time is also directly proportional to the total number of sections generated. A comparison of the table generation time and the execution times for the best cases shows that for most of the considered cases, the table generation time is in the range of 5% to 50% of the execution time.

7 Discussion and Conclusion

In this paper, we have presented techniques for efficient enumeration of the data index sets and processor index sets for array expression execution involving arrays distributed using the block, cyclic, and block-cyclic distributions. Closed forms expressed as regular sections for determining the send and receive data and processor sets for block and cyclic distributions were derived. A strategy based on virtual processors was used along with these closed forms to generate efficient indexing code for arrays distributed using block-cyclic distributions. A heuristic for selecting the appropriate virtual views at the source and target distribution was developed. Performance results for an implementation on the Cray T3D were presented. The performance results demonstrate that the developed heuristic provides a good indication of the virtualization scheme with the lowest execution time. It is also observed that for a majority of the cases considered the table generation overhead for the virtual processor approach is a small percentage of the execution time.

The virtual processor approach can be further optimized to handle frequently occurring cases such as data redistribution and array expressions with unit strides and/or identically distributed arrays, more efficiently. These optimizations will reduce the table generation time but are case-specific and are not presented in the paper.

The developed scheme for compilation of the array statement can be extended to cover compilation of some forms of other data parallel constructs, such as the WHERE statement in Fortran 90 and for building the parameter list when passing array sections of distributed arrays to a subroutine call. We do not address the execution of statements with vector-valued subscripts which are allowed in the HPF FORALL statement and in the Fortran 90 array and WHERE statement. In general, runtime schemes as proposed in [15] will be required to handle vector valued subscripts.

We have presented techniques for handling one-dimensional arrays. The virtual processor approach can be extended to handle multi-dimensional arrays. The communication sets are developed independently for each array dimension using the techniques presented for one-dimensional arrays. The communication sets for the array statement involving multi-dimensional arrays are obtained by taking the cross-product of the appropriate sets for each dimension. Consider arrays $A(m_{a1} : n_{a1}, m_{a2} : n_{a2})$ and $B(m_{b1} : n_{b1}, m_{b2} : n_{b2})$ distributed using block-cyclic distributions on a $P_1 \times P_2$ processor mesh. The processors in the mesh are labelled as (p_1, p_2), $0 \leq p_1 < P_1$ and $0 \leq p_2 < P_2$. Consider the array statement $B(l_{b1} : u_{b1} : s_{b1}, l_{b2} : u_{b2} : s_{b2}) = f(A(l_{a1} : u_{a1} : s_{a1}, l_{a2} : u_{a2} : s_{a2}))$. The active virtual processors on processor (p_1, p_2) are $VPAct\_A1(p_1) \otimes VPAct\_A2(p_2)$ where $VPAct\_A1(p_1)$ and $VPAct\_A2(p_2)$ are the sets of active virtual processors corresponding to the first and second dimensions of array A, respectively and $\otimes$ denotes the cross-product of two sets. The sets $VPSend()$, $VPRecv()$, $VDSend()$, and $VDRecv()$ for the multi-dimensional array are defined similarly. Thus the communication code for the multi-dimensional array statement would be as shown in Fig. 13. The send and receive sets for the first dimension have a subscript one, while those for the second dimension have a subscript two. This communication code should be expressed as two-phases

```
/* Sending phase for processor (p1, p2) */
for v_p1 ∈ VAct_A1(p1)
  for v_p2 ∈ VAct_A2(p2)
    for v_q1 ∈ VPSend1(v_p1)
      for v_q2 ∈ VPSend2(v_p2)
        Pack A_loc(VDSend1(v_p1, v_q1),
            VDSend2(v_p2, v_q2)) in tmp_out
        Send ((Phy_B1(v_q1), Phy_B2(v_q2)), tmp_out)
```

```
/* Receiving phase for processor (p1, p2) */
for v_p1 ∈ VAct_B1(p1)
  for v_p2 ∈ VAct_B2(p2)
    for v_q1 ∈ VPRecv1(v_p1)
      for v_q2 ∈ VPRecv2(v_p2)
        recv ((Phy_A1(v_q1), Phy_A2(v_q2)), tmp_in)
        B_loc(VDRecv1(v_p1, v_q1), VDRecv2(v_p2, v_q2))
         = f(tmp_in)
```

Figure 13: Communication phases for two-dimensional arrays in the virtual processor domain.

to eliminate multiple messages between two physical processors. Note that the closed form expressions developed for the communication sets in Section 4 allow for a different number of physical processors at the source and target distribution. Thus a *change of the shape* of the underlying processor grid and the multi-dimensional array can be handled by the virtual processor approach.

HPF supports a two-level mapping of data arrays to an abstract processor grid. The language introduces a Cartesian grid referred to as a *template*. Arrays are *aligned* to the template and the templates are *distributed* onto a processor array using regular data distributions. The alignment of the array elements to the template elements can be performed at a stride with an initial offset. For instance, an array $A(0 : N-1)$ is aligned with a template D such that $A(i)$ is aligned with $D(ai + b)$. The introduction of the stride a in the array-template mapping creates "holes" in the the array A_loc of each processor as some template cells do not have any array elements mapped to them. The methods presented in this paper are directly applicable to arrays which are distributed using a two-level mapping when existence of holes in the local memory arrays is reasonable. This follows by noting that the array section $A(l : u : s)$ is equivalent to the template section $D(a*l + b : a*u + b : a*s)$. By expressing all the array sections in the array statement as sections of the template D, the relevant communication sets can be constructed.

The execution of the array statement in the paper follows the owner-computes rule which is prevalent in compilers for HPF-like languages. However, the closed form expressions developed in Section 4 are not restricted to the owner computes-rule and can be either directly used or adapted to cases where the iterations are distributed using a regular distribution as in KALI [12]. For instance, consider the KALI-like code in Fig. 14(a), in which the iterations of the specified loop and the arrays A and B are distributed among the processors using a block-cyclic distribution. The execution of the loop can be performed by the equivalent sequence of array statements in Fig. 14(b). The array $T(0 : N-1)$ is distributed identically as the iterations of the loop. The communication sets for the array statements can be determined using the virtual processor

$$\begin{array}{l} \textbf{forall } i \textbf{ in } 0..N-1 \textbf{ on } (i \; div \; b) \; mod \; P \\ \quad B(l_2 + is_2) = f(A(l_1 + is_1)) \end{array}$$

(a)

$$\begin{array}{ll} T(0:n-1) = A(l_1:u_1:s_1) & \text{/* pre-communication */} \\ T(0:n-1) = f(T(0:n-1)) & \text{/* computation */} \\ B(l_2:u_2:s_2) = T(0:n-1) & \text{/* post-communication */} \end{array}$$

(b)

Figure 14: Execution of program with explicit computation distribution.

approach and the closed form expressions developed in Section. 4.

The developed code is available by anonymous ftp from `ftp.cis.ohio-state.edu:pub/hpce/compiler/Source/arraysect`.

Acknowledgments

We would like to thank J. Ramanujam for comments on an initial draft. We would also like to thank the Ohio Supercomputer Center for access to the Cray T3D.

References

[1] I. Ahmad, R. Bordawekar, Z. Bozkus, A. Choudhary, G. Fox, K. Parasuram, R. Ponnusamy, S. Ranka, and R. Thakur. Implementation and scalability of Fortan-90D intrinsic functions on distributed memory machines. Technical Report SCCS-256, Northeast Parallel Architectures Center, June 1992.

[2] D. Burton. *Elementary Number Theory.* Allyn and Bacon, Inc, Boston, revised printing edition, 1984.

[3] B. M. Chapman, P. Mehrotra, and H. P. Zima. Vienna Fortran – a Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Language, Compilers and Runtime Environments for Distributed Memory Machines*, pages 39–62. 1992.

[4] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S.-H. Teng. Generating local addresses and communication sets for data parallel programs. In *Proc. of ACM Symposium on Principles and Practices of Parallel Programming*, pages 149–158, May 1993.

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

[6] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran-D language specification. Technical Report TR-91-170, Dept. of Computer Science, Rice University, Dec. 1991.

[7] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On the generation of efficient data communication for distributed-memory machines. In *Proc. of Intl. Computing Symposium*, pages 504–513, Dec. 1992.

[8] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. In *Proc. of Intl. Conf. on Parallel Processing*, volume II, pages 301–305, Aug. 1993.

[9] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Advanced compilation techniques for Fortran D. Technical Report CRPC-TR-93-338, Center for Research on Parallel Computation, Rice University, Oct. 1993.

[10] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Compilation techniques for block-cyclic distributions. In *Proc. of Intl. Conf. on Supercomputing*, pages 392–403, July 1994.

[11] D. Knuth. *Seminumerical Algorithms*, volume II of *The Art of Computer Programming*. Addison Wesley Publishing Company, Reading, MA, second edition edition, 1981.

[12] C. Koelbel. Compile-time generation of communication for scientific programs. In *Proc. of Supercomputing '91*, pages 101–110, Nov. 1991.

[13] J. Merlin. Techniques for automatic parallelisation of 'Distributed Fortran 90'. Technical Report SNARC 92-02, Southampton Novel Architecture Research Center, June 1992.

[14] E. M. Paalvast, H. J. Sips, and A. J. van Gemund. Automatic parallel program generation and optimization from data decompositons. In *Proc. of Intl. Conf. on Parallel Processing*, volume II, pages 124–131, Aug. 1991.

[15] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, Apr. 1990.

[16] J. Stichnoth, D. O'Hallaron, and T. Gross. Generating communication for array statements: Design, implementation, and evaluation. *Journal of Parallel and Distributed Computing*, 4:150–159, 1994.

[17] J. M. Stichnoth. Efficient compilation of array statements for private memory multicomputers. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, Feb. 1993.

[18] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, Frontier Series. Addison Wesley Publishing Company, New York, New York, 1990.

Table 4: Evaluation of the strategy for the virtualization scheme selection ($P = 32$, $M = 230400$). IE: Indexing Cost Estimate, IT: Indexing Time (ms).

(a)

Parameters				vb-vb		vb-vc		vc-vb		vc-vc	
b_1	b_2	s_1	s_2	IE	IT	IE	IT	IE	IT	IE	IT
6	4	1	1	3000	17	14400	50	14400	45	24^*	$3^\dagger$
10	1	1	1	14400	42	14400	46	14400	46	20^*	$3^\dagger$
5	9	1	1	2240	14	14400	49	14400	45	90^*	$3^\dagger$
15	16	1	1	930	8	14400	49	14400	44	480^*	$4^\dagger$
32	225	1	1	449^*	$4^\dagger$	14400	48	14400	44	14400	55
5	7200	1	1	2880	10	14400	45	320^*	$3^\dagger$	14400	51
225	7200	1	1	64^*	$3^\dagger$	14400	45	14400	41	14400	49
7200	3	1	1	4800	14	192^*	$4^\dagger$	14400	44	14400	57

(b)

Parameters				vb-vb		vb-vc		vc-vb		vc-vc	
b_1	b_2	s_1	s_2	IE	IT	IE	IT	IE	IT	IE	IT
6	4	3	3	9000	38	14400	46	12600	46	24^*	$3^\dagger$
10	1	2	2	14400	63	21600	68	14400	66	30^*	$4^\dagger$
5	9	3	3	6720	34	11520	48	14400	47	90^*	$3^\dagger$
15	16	4	4	3720	26	12960	52	12960	49	432^*	$3^\dagger$
32	225	2	2	898^*	$6^\dagger$	14368	48	14368	45	14368	29
5	21600	3	3	8640	26	11520	46	320^*	$3^\dagger$	14400	52
225	36000	5	5	320^*	$5^\dagger$	14400	48	2880	12	14400	51
14400	3	2	2	9600	26	80^*	$6^\dagger$	12000	50	12000	66

(c)

Parameters				vb-vb		vb-vc		vc-vb		vc-vc	
b_1	b_2	s_1	s_2	IE	IT	IE	IT	IE	IT	IE	IT
6	4	3	2	7200	24	14400	46	14400	43	10^*	$3^\dagger$
10	1	2	3	14400	42	14400	46	14400	46	25^*	$3^\dagger$
5	9	2	3	5280	30	12960	55	12960	52	66^*	$3^\dagger$
15	16	5	3	3750	28	14400	48	13950	48	384^*	$3^\dagger$
32	225	3	5	1315^*	$9^\dagger$	13950	48	14400	45	11520	14
5	36000	2	5	5760	19	12960	50	144^*	$3^\dagger$	12960	56
225	7200	3	1	192^*	$4^\dagger$	14400	46	4800	17	14400	50
21600	3	3	2	9600	27	80^*	$6^\dagger$	12000	50	12000	66

*: predicted best., †: actual best.

Table 5: Comparison of the Execution Time and the Table Generation Time ($P = 32, M = 230400$).

(a)

Parameters				Execution Time (ms)				Table Generation Time (ms)			
b_1	b_2	s_1	s_2	vb-vb	vb-vc	vc-vb	vc-vc	vb-vb	vb-vc	vc-vb	vc-vc
6	4	1	1	27	60	55	13^*	631	5707	4436	$2^\dagger$
10	1	1	1	50	54	53	10^*	2272	3252	1886	$1^\dagger$
5	9	1	1	21	56	52	10^*	167	1910	1550	$5^\dagger$
15	16	1	1	14	55	51	10^*	30	861	739	$23^\dagger$
32	225	1	1	10^*	54	50	64	$7^\dagger$	512	556	966
5	7200	1	1	16	51	9^*	59	56	771	$3^\dagger$	677
225	7200	1	1	9^*	51	47	58	$1^\dagger$	717	859	903
7200	3	1	1	20	10^*	49	63	94	$2^\dagger$	782	821

(b)

Parameters				Execution Time (ms)				Table Generation Time (ms)			
b_1	b_2	s_1	s_2	vb-vb	vb-vc	vc-vb	vc-vc	vb-vb	vb-vc	vc-vb	vc-vc
6	4	3	3	50	57	57	14^*	4722	9123	7888	$1^\dagger$
10	1	2	2	74	79	79	13^*	6895	9443	6126	$1^\dagger$
5	9	3	3	40	55	54	10^*	1150	2839	2030	$2^\dagger$
15	16	4	4	33	58	56	10^*	253	1179	1017	$6^\dagger$
32	225	2	2	12^*	54	51	35	$14^\dagger$	538	501	429
5	21600	3	3	33	52	10^*	61	347	931	$3^\dagger$	883
225	36000	5	5	11^*	54	18	60	$4^\dagger$	742	46	706
14400	3	2	2	33	12^*	56	72	289	$1^\dagger$	1053	1189

(c)

Parameters				Execution Time (ms)				Table Generation Time (ms)			
b_1	b_2	s_1	s_2	vb-vb	vb-vc	vc-vb	vc-vc	vb-vb	vb-vc	vc-vb	vc-vc
6	4	3	2	28	50	47	6^*	3398	19438	12247	$0.30^\dagger$
10	1	2	3	50	54	54	11^*	4663	6291	3900	$1^\dagger$
5	9	2	3	38	64	62	11^*	1019	3597	2730	$2^\dagger$
15	16	5	3	34	56	54	10^*	328	1332	1013	$6^\dagger$
32	225	3	5	15^*	55	51	20	$32^\dagger$	611	509	185
5	36000	2	5	25	56	9^*	66	170	989	$2^\dagger$	902
225	7200	3	1	10^*	53	24	59	$2^\dagger$	703	101	723
21600	3	3	2	33	12^*	56	72	294	$1^\dagger$	1097	1234

*: best execution time, †: best table generation time.