Compiler Techniques for Determining Data Distribution and Generating Communication Sets on Distributed-Memory Machines

PeiZong Lee and Wen-Yao Chen
Institute of Information Science, Academia Sinica
Taipei, Taiwan, R.O.C.

Abstract

This paper is concerned with designing efficient algorithms for determining data distribution and generating communication sets on distributed memory multic和平ers. First, we propose a dynamic programming algorithm to automatically determine data distribution at compiling time. The proposed algorithm also can determine whether data redistribution is necessary between two consecutive DO-loop program fragments. Second, we propose closed forms to represent communication sets among processing elements for executing doall statements, when data arrays are distributed in a restricted block-cyclic fashion. Our methods can be included in current compilers and used when programmers fail to provide any data distribution directives. Experimental studies on a nCUBE-2 multicomputer are also presented.

1 Introduction

Arrays distribution and communication sets generation are two problems we must solve when dealing with the compilation of DO-loop program fragments for distributed memory multic和平ers. For instance, in High Performance Fortran (HPF), programmers have obligations to provide TEMPLATE, ALIGN, and DISTRIBUTE directives to specify data distribution [8]. Then, based on these directives, compilers can generate all communication instructions. In this paper, however, we try to determine data distribution automatically by compilers in contrast with previous research works, which previously only allowed programmers explicitly to specify the data distribution using language extensions. We show systematic methods for determining data distributions and for generating communication sets for each processing element (PE). Thus, the proposed algorithms can be included in compilers for automatically transforming sequential DO-loop program fragments into parallel version programs with message-passing communication primitives. For instance, our methods can be included in HPF compilers and used when programmers fail to provide any data distribution directives.

In the following, we state the problems we will address in this paper. First, given a sequence of DO-loop programs, we are interested in how to align data arrays, so that data communication incurred due to the resulting data distribution will be minimized. Conventionally, this problem can be solved by using a component alignment algorithm to determine a static data distribution scheme for the whole program [3] [12]. In contrast to giving a static solution, we will present a dynamic programming algorithm to determine whether data redistribution is necessary between two consecutive DO-loop program fragments.

Second, after determining data alignments among data arrays, we are interested in how to distribute data arrays among PEs. In order to do this, compilers must include an analytical model, which can formulate communication time and computation time. In addition, this analytical model can help to determine grain and granularity of execution space, it also can help to determine whether data arrays are distributed among PEs by a block fashion, or a cyclic fashion, or a block-cyclic fashion.

Third, after determining data distribution, we focus our attentions on generating communication sets among PEs. Previous research works have provided closed forms of generating communication sets for the special cases when an array's distribution is either in a block fashion or in a cyclic fashion [7] [9]. Recently, a lot of research works are concentrated on the more general cases when an array's distribution is in a block-cyclic fashion [2] [4] [5] [13]. However, for the general cases, none of them provided formulas which could be represented by a constant number of closed forms. We are interested in integrating previous research works and in formulating a complete set of closed forms of generating communication sets for each PE.

*This work was partially supported by the NSC under Grant NSC 85-2213-E-001-006.
2 Background

The following closed forms (regular sections) will be used in this paper.

- \([a : e_1]\) represents the set of consecutive integers from \(a\) to \(e_1\).
- \([a : e_1 : e_2]\) in behalf of the set of integers from \(a\) with a stride (period) \(s_1\) until to a maximum integer which is not greater than \(e_1\).
- \([a : e_1 : e_2 : e_3]\) is the set \([\{a : e_1\}, [a : e_1] + s_2, [a : e_1] + 2s_2, \ldots\] until not greater than \(e_2\).
- \([a : e_1 : e_2 : e_3 : e_4]\) specifies the set \([\{a : e_1 : e_2\}, [a : e_1 : e_2] + s_3, [a : e_1 : e_2] + 2s_3, \ldots\] until not greater than \(e_3\).

Suppose that array \(A([a_1 : a_2])\) is indexed from \(a_1\) to \(a_2\) and there are in total \(N\) PEs numbered from 0 to \(N - 1\). Then, if we adopt cyclic \((b)\) distribution, the set \(A([a_1 + p * b : a_1 + p * b + b - 1]) : a_2 : N * b\) is stored in PE \(p(PE_p)\). We will say that array \(A\) is distributed by a cyclic fashion if \(b = 1\); by a block-fashion if \(b = [(CQ - a1 + 1)/N]\); and by a block-cyclic fashion if \(b = 1\); by a block-cyclic fashion if \(1 < b < [(CQ - a1 + 1)/N]\).

2.1 Compiling Sequential Programs on Distributed Memory Machines

The abstract target machine we adopt is a \(q\)-D grid of \(N_1 x N_2 x \cdots x N_q\) PEs, where \(D\) stands for dimensional. A PE on the \(q\)-D grid is represented by the tuple \((p_1, p_2, \ldots, p_q)\), where \(0 < p_i < N_i - 1\) for \(1 \leq i \leq q\). The parallel program generated from a sequential program for a grid corresponds to the SPMD (Single Program Multiple Data) model, in which each PE executes the same program but operates on distinct data items [6]. In this paper, we adopt a global name space for representing data arrays among PEs. Therefore, our machine model can be regarded as a distributed shared memory model [7] [9].

When dealing with the compilation of a sequential program on a distributed memory computer, we must decide on a suitable data distribution for each data array, so that a computation load balance can be achieved; in addition, overhead due to communication can be minimized. We also must provide efficient algorithms for generating communication sets, so that performance gained due to parallel computing will not be degraded by software overhead. Previously, researchers have shown that after applying loop transformation techniques such as loop interchange; loop reversal; and loop skewing, a sequential Do-loop program fragment can be transformed into an equivalent program fragment either with doall loops in all levels, or with an outmost doserial loop in which all its inner loops are doall loops [15]. Doall loops guarantee that statements in different iterations (loop bodies) can be executed independently even in different PEs. Therefore, we can group different sets of iterations into PEs, and execute each set of iterations in different PEs independently.

Figure 1 shows two programs: (a) a sequential program for solving a linear system \(AX = B\), and (b) its corresponding doall loop program. The corresponding SPMD program in which data arrays are distributed by cyclic \((b)\) can be found in [10]. Readers can find that there is a one-to-one correspondence between statements in the original sequential program (which have been rewritten after performing loop transformations) and its corresponding doall-version program. For this reason, without any confusion, in the sequel we will frequently apply compiler techniques directly on the sequential programs. As to compile a doall loop version program to a SPMD program, it is straightforward if data distributions for all arrays (or matrices) are determined.

\[\begin{align*}
&\text{REAL A(m, m), B(m), X(m), Y(m)} \\
&\text{do } k = 0, m-1 \text{ (* A = LU. *)} \\
&\text{do } j = k+1, m-1 \text{ (* doall k = k + 1, m-1 *)} \\
&\text{A(k, j) = A(k, j)/A(k, k)} \\
&\text{do } j = k+1, m-1 \text{ (* doall j = k + 1, m-1 *)} \\
&\text{A(k, j) = A(k, j)-A(k, k)*A(k, j)} \\
&\text{endo enddo enddo} \\
&\text{do } i = 0, m-1 \text{ (* LY = B. *)} \\
&\text{Y(i) = B(i)} \\
&\text{do } j = i+1, m-1 \text{ (* doall j = i + 1, m-1 *)} \\
&\text{B(j) = B(j) - A(j, i) * Y(i)} \\
&\text{endo enddo enddo} \\
&\text{do } i = m-1, 0, -1 \text{ (* X = V. *)} \\
&\text{X(i) = X(i)/A(i, i)} \\
&\text{do } j = 0, i-1 \text{ (* doall j = 0 + 1, i-1 *)} \\
&\text{X(j) = X(j) - A(j, i) * X(i)} \\
&\text{endo enddo enddo} \\
&\text{endo enddo}
\end{align*}\]

Figure 1: Solving a linear system \(AX = B\) based on the LU decomposition: (a) the original sequential program, (b) the corresponding doall loop version.

3 Determining Data Distribution

We now show how to use a component alignment algorithm to determine data distribution. First, we analyze the relationship between left-hand-side and right-hand-side array subscript reference patterns in the
Table 1: Communication primitives used in the SPMD program when left-hand-side and right-hand-side array subscripts have some specific patterns. \(i\) and \(j\) are loop indexing variables; \(c\), \(c_1\), and \(c_2\) are constants at compile time; "unknown" means that the value is unknown at compile time; \(f_1(i)\) and \(f_2(i)\) are two affine functions of the form \(s_1 \times \sum i + c_1\) and \(s_2 \times \sum i + c_2\), respectively; \(f_3(i)\) and \(f_4(j)\) are two functions of \(i\) and \(j\), respectively. The parameter \(m\) denotes the message size in words; \(seq\) is a sequence of identifiers representing the processors in various dimensions over which the collective communication primitive is carried out. The function \(num\) applied to such a sequence simply returns the total number of processors involved.

<table>
<thead>
<tr>
<th>case</th>
<th>LHS</th>
<th>RHS</th>
<th>communication primitive</th>
<th>cost on hypercube</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(c_1)</td>
<td>(c_2)</td>
<td>Transfer((m))</td>
<td>(O(m))</td>
</tr>
<tr>
<td>2</td>
<td>(i)</td>
<td>(i + c)</td>
<td>Shift((m))</td>
<td>(O(m))</td>
</tr>
<tr>
<td>3</td>
<td>(f_1(i))</td>
<td>(f_2(i))</td>
<td>&quot;need additional analysis&quot;</td>
<td>&quot;need additional analysis&quot;</td>
</tr>
<tr>
<td>4</td>
<td>(i)</td>
<td>(c)</td>
<td>OneToManyMulticast((m, seq))</td>
<td>(O(m \times \log \text{num}(seq)))</td>
</tr>
<tr>
<td>5</td>
<td>(c)</td>
<td>(i)</td>
<td>Reduction((m, seq))</td>
<td>(O(m \times \log \text{num}(seq)))</td>
</tr>
<tr>
<td>6</td>
<td>(i)</td>
<td>unknown</td>
<td>Gather((m, seq))</td>
<td>(O(m \times \text{num}(seq)))</td>
</tr>
<tr>
<td>7</td>
<td>unknown</td>
<td>(i)</td>
<td>Scatter((m, seq))</td>
<td>(O(m \times \text{num}(seq)))</td>
</tr>
<tr>
<td>8</td>
<td>(i \times f_3(i))</td>
<td>(j \times f_4(j))</td>
<td>ManyToManyMulticast((m, seq))</td>
<td>(O(m \times \text{num}(seq)))</td>
</tr>
</tbody>
</table>

Table 2: Approximate communication costs.

<table>
<thead>
<tr>
<th>case</th>
<th>edge</th>
<th>edge</th>
<th>approximate communication cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>1-D</td>
<td>1-D</td>
<td>(N_1 \times \text{OneToManyMulticast}(\frac{m}{N_1}, {N_2 \text{ PEs}})) or (m \times \text{Transfer}(1))</td>
</tr>
<tr>
<td>C2</td>
<td>2-D</td>
<td>1-D</td>
<td>(N_1 \times \text{OneToManyMulticast}(\frac{m}{N_1}, {N \text{ PEs}}))</td>
</tr>
<tr>
<td>C3</td>
<td>1-D</td>
<td>2-D</td>
<td>(\text{OneToManyMulticast}(\frac{m}{N_1}, {N_2 \text{ PEs}})) or (N \times \text{OneToManyMulticast}(\frac{m}{N_1}, {N_2 \text{ PEs}}))</td>
</tr>
<tr>
<td>C4</td>
<td>2-D</td>
<td>2-D</td>
<td>(\text{OneToManyMulticast}(\frac{m}{N_1}, {N \text{ PEs}})) or (m^2 \times \text{Transfer}(1))</td>
</tr>
</tbody>
</table>

The component alignment problem is defined as partitioning the node set of the component affinity graph into \(q\) disjointed subsets (\(q\) is the dimension of the abstract target grid and \(q\) may be larger than the dimension of the physical target grid) so that the total weight of edges across nodes in different subsets is minimized, with the restriction that no two nodes corresponding to the same array are in the same subset. These \(q\) disjointed subsets will be used to determine data distributions for all data arrays.

We now return to our example of the linear system. Figure 2 shows the component affinity graph and the suggested component alignment of the sample program mentioned in Figure 1. Suppose that our target machine is a linear processor array with \(N\) PEs. For the purpose of parallelism, based on the suggested component alignment, matrix \(A\) will be distributed by cyclic\((b)\) along its rows; arrays \(B\), \(X\), and \(Y\) will also be distributed by cyclic\((b)\). The data distribution functions of \(A\), \(B\), \(X\), and \(Y\) are listed below.

\[
f_A(i, j) = (\lfloor \frac{i}{b} \rfloor \mod N)
\]

\[
f_B(i) = f_X(i) = f_Y(i) = (\lfloor \frac{i}{b} \rfloor \mod N)
\]

Note that, the data distribution function \(f_X(i) = p\)
means that the entry \( i \) of the one-dimensional data
array \( X \), \( X(i) \), is stored in \( PE_i \). The data distribution
function \( f_x(i, j) = p \) means that the entry \((i, j)\) of the
two-dimensional data matrix \( A \), \( A(i, j) \), is stored in
\( PE_i \). In the next subsection, we will show how to
decide the block size \( b \).

Linear systems be \( T_{TLS} \); and the total execution
time be \( T \). Then, from the corresponding SPMD program
in [10], we can formulate \( T, T_{LU} \), and \( T_{TLS} \) as follows.

\[
T = T_{LU} + T_{TLS},
\]
\[
T_{LU} = \sum_{i=1}^{m/((N*b))} \sum_{j=1}^{N} (9*t_f + \sum_{i=1}^{b} (t_f + (m - ((i - 1) * N * b + (i - 1) * b + i) + 1) * t_x) + \sum_{i=1}^{m/((N*b))} \sum_{j=1}^{N} (t_f + (m - ((i - 1) * N * b + (i - 1) * b + i) + 1) * t_x) + (log(N + 1) + b * (t_x + (m - ((i - 1) * N * b + (i - 1) * b)) * t_c)) + (log(N + 1) + b * (t_x + (m - ((i - 1) * N * b + (i - 1) * b)) * t_c)).
\]

The symbolic manipulations of the above formulas
can be solved using a computer algebra system like
"Derive" [14]. The total execution time is a function
of the problem size \( m \), the number of PEs \( N \), and the
block size \( b \). When the problem size \( m \) and the number
of PEs \( N \) are fixed, the optimal execution time can be
obtained by requiring \( \partial T / \partial b = 0 \), or by substituting all
possible \( b \) into the formula. Table 4 shows \( T_{LU}, T_{TLS}, \) and \( T \) for various block size \( b \) ranging from 1 to 64,
and for various numbers of PEs \( N \) ranging from 2 to
32, when the problem size \( m \) is 1024. We also list the
real execution time on a 32-node nCUBE-2 computer
for a comparison.

It is interesting to point out that both the optimal
execution time of the LU decomposition and the whole
program is achieved when the block size is 1; however,
the optimal execution time of two triangular linear
systems is achieved when the block size is 8 or 16.
We will discuss other details of choosing a block size
\( b \) again in Section 4.

### 3.2 Determining the Granularity of Data Distribution

There are two oracles to help decide the block size \( b \).
The load balance oracle suggests using cyclic(1) distri-
bution if the iteration space is a pyramid (such as the
iteration space of the LU decomposition), a triangle
(such as the iteration space of two triangular linear
systems), or any other non-rectangular space. The
communication oracle emphasizes not to divide the
block size too small, otherwise it will incur a high com-
munication overhead and a high indexing overhead.
These two oracles, unfortunately, are inconsistent.

We can formulate the total execution time from the
SPMD program which includes both the computation
time and the communication time. For each arithmeti-
cal operation or logical operation, we assume that the
computation time is \( t_f \); for each sassy operation which
executes a multiplication then follows an addition, and
we assume that the computation time is \( t_x \); for each
message passing operation, we assume that the
communication cost is \( t_c + k \cdot t_x \), where \( t_c \) is the start-up
time for sending a message; \( t_x \) is the communication
time of transferring a word; and \( k \) is the message size
in words. Table 3 shows the parameters \( t_f, t_x, t_c, \) and
\( t_c \) on a 32-node nCUBE-2 computer.

<table>
<thead>
<tr>
<th>parameter</th>
<th>( t_f )</th>
<th>( t_x )</th>
<th>( t_c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean (in ( \mu \text{sec} ))</td>
<td>4.56</td>
<td>7.15</td>
<td>171.34</td>
</tr>
<tr>
<td>variance (in ( \mu \text{sec} ))</td>
<td>0.18</td>
<td>0.14</td>
<td>0.94</td>
</tr>
</tbody>
</table>

Table 3: Parameters used in describing the execution
time on the nCUBE-2 computer.

We now continue our sample example of the linear
system. Let the time of executing the LU decompo-
sition be \( T_{LU} \); the time of executing two triangular

---

Based on the provided text, the following questions can be answered:

1. What are the characteristics of the data distributions for linear systems?
   - Linear systems be \( T_{TLS} \); and the total execution
time be \( T \). Then, from the corresponding SPMD program
in [10], we can formulate \( T, T_{LU} \), and \( T_{TLS} \) as follows.

2. How are the block sizes \( b \) determined for the given problem sizes \( m \) and PEs \( N \)?
   - The symbolic manipulations of the above formulas
can be solved using a computer algebra system like
"Derive" [14]. The total execution time is a function
of the problem size \( m \), the number of PEs \( N \), and the
block size \( b \). When the problem size \( m \) and the number
of PEs \( N \) are fixed, the optimal execution time can be
obtained by requiring \( \partial T / \partial b = 0 \), or by substituting all
possible \( b \) into the formula. Table 4 shows \( T_{LU}, T_{TLS}, \) and \( T \) for various block size \( b \) ranging from 1 to 64,
and for various numbers of PEs \( N \) ranging from 2 to
32, when the problem size \( m \) is 1024. We also list the
real execution time on a 32-node nCUBE-2 computer
for a comparison.

3. What is the significance of choosing a block size \( b \)?
   - It is interesting to point out that both the optimal
execution time of the LU decomposition and the whole
program is achieved when the block size is 1; however,
the optimal execution time of two triangular linear
systems is achieved when the block size is 8 or 16.
We will discuss other details of choosing a block size
\( b \) again in Section 4.

---

For more detailed analysis, refer to the original document or the referenced sections.
Table 4: The simulation time in units of seconds for solving a linear system $A_{1024 \times 1024} X_{1024} = B_{1024}$ based on the LU decomposition and two triangular linear systems. The data that are not in parentheses are obtained by running a 32-node nCUBE-2 computer; the data in parentheses are based on an analytical model.

<table>
<thead>
<tr>
<th>block size</th>
<th>#PE = 2</th>
<th>#PE = 4</th>
<th>#PE = 8</th>
<th>#PE = 16</th>
<th>#PE = 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$T_{LU}$</td>
<td>1315.1 (1286.4)</td>
<td>661.4 (646.0)</td>
<td>334.0 (327.9)</td>
<td>173.1 (169.9)</td>
</tr>
<tr>
<td>2</td>
<td>$T_{LU}$</td>
<td>1318.0 (1290.6)</td>
<td>663.5 (650.0)</td>
<td>336.8 (330.3)</td>
<td>174.0 (171.4)</td>
</tr>
<tr>
<td>3</td>
<td>$T_{LU}$</td>
<td>1316.8 (1290.1)</td>
<td>665.5 (650.7)</td>
<td>337.1 (331.6)</td>
<td>174.2 (172.8)</td>
</tr>
<tr>
<td>4</td>
<td>$T_{LU}$</td>
<td>1319.4 (1294.3)</td>
<td>665.1 (653.1)</td>
<td>338.3 (333.5)</td>
<td>175.3 (174.2)</td>
</tr>
<tr>
<td>5</td>
<td>$T_{LU}$</td>
<td>1320.5 (1297.6)</td>
<td>668.8 (658.1)</td>
<td>343.2 (336.1)</td>
<td>180.6 (180.2)</td>
</tr>
<tr>
<td>6</td>
<td>$T_{LU}$</td>
<td>1323.0 (1301.6)</td>
<td>670.3 (660.4)</td>
<td>344.2 (340.5)</td>
<td>181.4 (181.7)</td>
</tr>
<tr>
<td>7</td>
<td>$T_{LU}$</td>
<td>1325.2 (1313.6)</td>
<td>680.2 (673.1)</td>
<td>359.3 (353.9)</td>
<td>194.4 (194.8)</td>
</tr>
<tr>
<td>8</td>
<td>$T_{LU}$</td>
<td>1330.6 (1316.6)</td>
<td>681.5 (675.2)</td>
<td>357.1 (355.2)</td>
<td>195.0 (195.6)</td>
</tr>
<tr>
<td>9</td>
<td>$T_{LU}$</td>
<td>1343.7 (1342.6)</td>
<td>702.3 (702.8)</td>
<td>382.8 (383.2)</td>
<td>221.8 (223.1)</td>
</tr>
<tr>
<td>10</td>
<td>$T_{LU}$</td>
<td>1345.1 (1346.6)</td>
<td>704.6 (705.0)</td>
<td>383.6 (384.5)</td>
<td>222.4 (224.0)</td>
</tr>
<tr>
<td>11</td>
<td>$T_{LU}$</td>
<td>1375.2 (1402.2)</td>
<td>749.0 (761.6)</td>
<td>436.5 (440.1)</td>
<td>274.7 (276.3)</td>
</tr>
<tr>
<td>12</td>
<td>$T_{LU}$</td>
<td>1377.7 (1405.8)</td>
<td>751.3 (764.0)</td>
<td>436.4 (441.6)</td>
<td>278.4 (277.3)</td>
</tr>
<tr>
<td>13</td>
<td>$T_{LU}$</td>
<td>1440.2 (1452.7)</td>
<td>844.6 (876.3)</td>
<td>530.0 (547.1)</td>
<td>371.7 (386.6)</td>
</tr>
</tbody>
</table>
| 14 | $T_{LU}$ | 1442.8 (1452.6) | 846.3 (878.1) | 540.5 (548.3) | 372.8 (376.1) | }

The data distribution scheme obtained from the above algorithm is at least as good as any static data distribution scheme, whose cost is equal to $T_{i,j}$. We now briefly describe how to improve this algorithm. It is clear that $M_{i,j}(i+1) > M_{i,j} + M(i+\beta), (r-i+1)$ and $M_{i,j}(i+1, j) > M(i+\gamma) + M(i+\gamma, (i+\gamma)+1)$, for some $\beta$ where $1 < \beta < \gamma_i + 1$. For the boundary cases when $\gamma_i = s - i + 1$ or $\beta = s - i + 1$, we define dummy values $M_{i,s+i-2}; M_{s+i,1};$ and $M(\gamma+\beta), (i-1, i+\beta+2)$, so that the above assumption is satisfied. Let $\gamma$ be the maximal value among $\gamma_i$, for $1 < i < s$. For example, $\gamma = \max_i \gamma_i$. Then the above algorithm can be improved as follows.

1'. for i := 2 to s do
2'. for j := 1 to $s - i + 1$ do
3'. $T_{i,j} := \min_{1 \leq k \leq \min(i, i+1)} (T_{i-k,k} + M_{i,j} + cost(P_{i-k,k}, P_{i,j}))$;
4'. end_for
5'. end_for

The time complexity of this improved dynamic programming algorithm is $O((s^2 + \gamma) \gamma + \gamma)$, which is bounded by $O(s^2 \gamma)$. In addition, before applying this algorithm, we need to compute at most $\gamma_i + \gamma_j + \ldots + \gamma_s$ reasonable-size component alignment problems for the consecutive Do-loops $L_i, L_{i+1}, \ldots, L_{i+j-1}$, where $1 < i < s$ and $1 < j < \gamma_i + 1$. The total number of component alignment problems computed is thus no more than $s \gamma + 1$.
4 Generating Communication Sets for Doall Statements

Suppose that in a distributed-shared-memory machine, processors are numbered from 0 to \( N - 1 \). Arrays \( A(\{a_1 : a_2\}) \) and \( C(\{c_1 : c_2\}) \) are distributed in cyclic(\( b_1 \)) and cyclic(\( b_2 \)), respectively. We now show how to compute necessary communication sets in each processor due to execute the following doall statement, where \( s_1 > 0 \), \( s_2 > 0 \), and \( g \) is a function.

\[
\text{doall } i = 0, \ldots, l - 1; \quad A(i + i * s_1) = g(C(i + i * s_2)).
\]

4.1 The Structure of Generated Code

We now analyze the problem. We will say that \( f_k(i) = \lfloor i / k \rfloor \) and the inverse functions \( f_k^{-1}(l_k + i * s) = i \), for \( k = 1 \) or 2. Figure 3 shows a detailed outline of implementing a doall statement in each PE which is a generalization based on formulas in [7].

Step 1 of Figure 3 generates an iteration set which specifies iterations to be performed on \( PE_p \), and two processor sets which represent PEs that \( PE_p \) will send data to or receive data from. Step 2 calculates communication sets and sends them to other PEs. Step 3 performs computations for iterations which access only local data. Step 4 receives data message from other PEs and executes computations for iterations which access local data and some message buffers. Note that, \( exec(p) \) in substep 1.1 is only formulated for deriving other communication sets and processor sets. Since \( exec(p) = \bigcup_{q \in recvc-pe(p)} \text{iter}(p, q) \) and \( \text{iter}(p, q) = f_{l_x}^{-1}(\text{localc}(p) \cap \text{localc}(q)) \), we can combine substep 1.1 and three substeps in Step 4 into a receive-execute loop. Therefore, in practice, iteration sets \( exec(p) \) and \( \text{iter}(p, q) \) need not to be calculated. It is also interesting to point out that in order to gain efficiency by allowing overlapped execution, we have arranged communication and computation tasks interleavedly.

4.2 Determining Suitable Block Sizes

In [10], we derived communication sets and processor sets with arbitrary block sizes \( b_1 \) and \( b_2 \). These sets, however, cannot be represented by closed forms. In this section, we return to analyze the block sizes of \( b_1 \) and \( b_2 \). Our goal is to choose reasonable block sizes \( b_1 \) and \( b_2 \), so that communication sets can be represented by closed forms.

Consider the target doall statement again. We first present an ideal case. Suppose that we assign the entry \( A(j) \) to \( PE_p = (\lfloor j / s_1 \rfloor \mod N) \) and the entry \( C(j') \) to \( PE_p' = (\lfloor j' / s_2 \rfloor \mod N) \). Then, for \( i \in \{0, 1, \ldots, h-1\} \), \( A(l_1 + i * s_1) \) and \( C(l_2 + i * s_2) \) are in \( PE_0 \); for \( i \in \{h, h+1, \ldots, 2*h-1\} \), \( A(l_1 + i * s_1) \) and \( C(l_2 + i * s_2) \) are in \( PE_1 \); and so on. In addition, there is no communication overhead to perform the target doall statement. In this ideal case, we notice that \( b_1 = s_1 * h \) and \( b_2 = s_2 * h \).

We now consider the general case. Suppose that the data distribution functions for arrays \( A \) and \( C \) are \( f_A(j) = (\lfloor j / s_1 \rfloor \mod N) \) and \( f_C(j') = (\lfloor j' / s_2 \rfloor \mod N) \), respectively. We found that, even if we don’t care about the values of \( offset_1 \) and \( offset_2 \), if \( b_1 / s_1 \) is a factor of \( b_2 / s_2 \), or \( b_1 / s_1 \) is a multiple of \( b_2 / s_2 \), then the communication sets can be represented by closed forms. However, if the condition fails, it will incur communication overhead due to random access patterns whose costs are very expensive. Table 5 summarizes certain conditions where processor sets and communication sets have closed forms.

If these sets can be represented by closed forms,
then they can be implemented efficiently. Otherwise, we only can use ad hoc methods to enumerate these sets and use indirectly memory access methods to get their corresponding data. The latter case, of course, will incur certain computation overhead. Therefore, our goal is to determine suitable block sizes such that the more sets can be represented by closed forms the better. In the following, we show examples to illustrate the flavor of choosing block sizes.

**Example 1:** Suppose that the loop bodies of two consecutive doall statements are

\[ A(l + i \times s_1) = A(l + i \times s_1) + C(l + i \times s_2) \]

\[ A(l + i \times s_1) = A(l + i \times s_1) - D(l + i \times s_3) \]

In this case, we choose \( b_1 = s_1 \times h \), \( b_2 = s_2 \times h \), and \( b_3 = s_3 \times h \), where block sizes \( b_1 \), \( b_2 \), and \( b_3 \) are for arrays \( A \), \( C \), and \( D \), respectively. Then all sets: \( \text{send}_{pec}(p, q) \), \( \text{recv}_{pec}(p, q) \), and \( \text{recv}_{pec}(p) \) for the first doall statement, as well as \( \text{send}_{pd}(p, q) \), \( \text{recv}_{pd}(p, q) \), and \( \text{recv}_{pd}(p) \) for the second doall statement all have closed forms.

**Example 2:** Suppose that the loop bodies of two consecutive doall statements are

\[ A(l + i \times s_1) = A(l + i \times s_1) \times C(l + i \times s_2) \]

\[ A(l + i \times s_1) = A(l + i \times s_1) - D(l + i \times s_3) \]

In this case, we choose \( b_1 = s_1 \times h \), \( b_2 = \text{lcm}(s_2, s_3) \times h \), and \( b_3 = s_3 \times h \). Then, except \( \text{recv}_{pc}(p) \) for the first doall statement and \( \text{send}_{pc}(p) \) for the second doall statement, all other sets have closed forms.

**Example 3:** Suppose that the loop bodies of two consecutive doall statements are

\[ A(l + i \times s_1) = A(l + i \times s_1) + C(l + i \times s_2) \]

\[ A(l + i \times s_1) = A(l + i \times s_1) - D(l + i \times s_3) \]

In this case, we choose \( b_1 = \text{lcm}(s_1, s_4) \times h \), \( b_2 = s_2 \times h \), and \( b_3 = s_3 \times h \). Then, except \( \text{send}_{pc}(p) \) for the first doall statement and \( \text{send}_{pc}(p) \) for the second doall statement, all other sets have closed forms.

### 4.3 Using Closed Forms to Represent Communication Sets

In this subsection, we derive processor sets and communication sets for cases when \( b_1 = s_1 \times h \) and \( b_2 = s_2 \times h \). The derivation of closed forms of other cases can be seen in [10]. We assume that the data distribution function for array \( A \) is \( f_A(j) = \left( \frac{j - u_1}{b_1} \right) \mod N \), thus, \( \text{local}_{A}(p) = \left[ a_1 + pb_1 : a_1 + pb_1 + b_1 - 1 : a_2 : Nb_1 \right] \). The data distribution function for array \( C \) is \( f_C(j) = \left( \frac{j - u_2}{b_2} \right) \mod N \), thus, \( \text{local}_{C}(p) = \left[ c_1 + pb_2 : c_1 + pb_2 + b_2 - 1 : c_2 : Nb_2 \right] \). We also assume that \( (u_1 - l_1) \) is a multiple of \( s_1 \) and \( u_2 = \left( (u_1 - l_1) / s_1 \right) * s_2 \).

The function \( \text{nxt}(a, b, c) \) we use here, is the smallest integer greater than \( a \) and is congruent to \( b \) modulo \( c \), that is, \( \text{nxt}(a, b, c) = a + (b - a) \mod c \). We now introduce some notations which will be used later.

\[ \text{bot}(A, p, j) = a_1 + pb_1 + j \times N \]
\[ \text{top}(A, p, j) = a_1 + pb_1 + b_1 - 1 + j \times N \]
\[ \text{top}_{pd}(A, p, j) = \text{max}(\text{top}(A, p, j), l_1) \]
\[ \text{top}_{pc}(A, p, j) = \text{max}(\text{top}(A, p, j), l_1) - s_1 + 1 \]
\[ \text{bot}_{pd}(A, p, j) = \text{bot}(A, p, j) \]
\[ \text{top}_{pd}(A, p, j) = \text{top}(A, p, j) - 1 \]
\[ \text{bot}_{pd}(A, p, j) = \text{bot}(A, p, j) \]
\[ \text{top}_{pd}(A, p, j) = \text{top}(A, p, j) - 1 \]

Let \( j_{pf} \) and \( j_{pl} \) be the first \( j \) and the last \( j \) such that \( \{ \text{bot}(A, p, j) : \text{top}(A, p, j) \cap \{ l_1 : u_1 \} \neq \phi \} \), respectively; and \( k_{pf} \) and \( k_{pl} \) be the first \( k \) and the last \( k \) such that \( \{ \text{bot}(C, p, k) : \text{top}(C, p, k) \cap \{ l_2 : u_2 \} \neq \phi \} \), respectively. We now introduce a property, which will be used in \( \text{send}_{pc}(p) \) and \( \text{recv}_{pc}(p) \).

**Property 1** Suppose that array \( A \) is distributed by \( \text{cyclic}(b_1) \); \( f_A(\cdot) \) is the data distribution function of array \( A \); \( x \) and \( y \) are two indices of array \( A \), where \( x < y \). Then, we have \( f_A(\{ x : y \}) = \)

\[
\begin{cases}
0 : N - 1, & \text{if } y - x + 1 > (N - 1) \times b_1; \\
[f_A(x) : f_A(y)], & \text{if } y - x + 1 \leq (N - 1) \times b_1 \text{ and } f_A(x) \leq f_A(y); \\
[f_A(y) \cup [f_A(x) : N - 1], & \text{if } y - x + 1 \leq (N - 1) \times b_1 \text{ and } f_A(x) > f_A(y). 
\end{cases}
\]

In [10], we investigated the periodic property of communication sets. Let \( \text{period}_{ab} \) be the period of the iteration pattern in \( \text{exec}(p) \) such that \( \text{period}_{ab} + s_1 \) is a multiple of \( N \); \( \text{period}_{ab} \) be the number of blocks of...
local elements of array A whose access pattern appears periodically; period, be the period of the reference pattern of array C in sendc(p, q) whose value is a multiple of Nbz; periods be the number of blocks of local elements of array C whose reference pattern appears periodically; and period$, be the number of blocks of local elements of array A, whose reference pattern of local elements of array C (based on fi(ecec(q))) appears periodically. We have the following equations.

\[
\begin{align*}
\text{period,} &= \frac{\text{lcm}(Nbr, sr)}{sr} \\
\text{period,b} &= \frac{\text{period,} \times sl}{Nbl} \\
\text{period,} &= \text{lcm}(Nbz, \text{period,} \times sz) \\
\text{periods} &= \frac{\text{period,}}{Nba} \\
\text{period$,} &= \frac{\text{period,} \times sl}{Nb1sz}.
\end{align*}
\]

We now formulate exe(p). Since period,6 = \(\frac{\text{period,} \times sl}{Nbl}\) = \(\frac{\text{lcm}(Nbl, sl)}{Nbl}\) = 1, and Nbl/sl = Nh1, we have botl(A, p, j + 1) = botl(A, p, j) + Nh1 for j, 0 ≤ j < jph. Thus, we have exe(p) = \(f^{-1}_A(\text{local}(A) \cap [l_1 : u_1 : s_1])\) = \(f^{-1}_A(\bigcup_{j=1}^{jph} [\text{botl(A, p, j) : topl(A, p, j) : s_1}]\) = \(\bigcup_{j=1}^{jph} f^{-1}_A(\text{botl(A, p, j) : topl(A, p, j)}\).

Now, we derive processor sets and communication sets. First, we process sendc(p), which is equal to \(f^{-1}_A(\text{localc}(p) \cap [l_2 : u_2 : s_2])\). Since periodc, = \(\frac{\text{period,} \times s_1}{(Nbl)}\) = \(\frac{\text{lcm}(Nbl, periodc, \times s_2)}{(Nbl)}\) = 1, it is enough to analyze the set of PEs which use elements of array C within a block of size b2. We found that if h2 ≥ N, then every PE will use some elements of array C within a block of size b2. If h2 < N, then the left boundary element and the right boundary element of array C within a block of size b2 are referred by \(f_A(\text{botj}(C, p, kph))\) and \(f_A(\text{topj}(C, p, kph))\), respectively. Note that, if \(\text{nxt}(\text{botj}(C, p, kph), l_2, s_2) < l_2\), then \(f_A(\text{botj}(C, p, kph)) \) maybe is not equal to \(f_A(\text{botj}(C, p, kph))\). Based on Property 1, we have the following closed form.

\[
\text{sendc}(p) = \bigcup_{j=1}^{jph} \bigcup_{k=1}^{kph} f^{-1}_A(\text{botj}(C, p, j) : topj(A, p, j) : s_1)\]
\]

Second, we formulate recvc(p, q), which is equal to \(f^{-1}_A(\text{exec(p)})\). We start from exec(p) and check the elements of array C that these iterations will refer to. Recall that \(\text{exec}(p) = \bigcup_{j=0}^{jph} [\text{botj}(A, p, j) : topj(A, p, j)]\). Then, \(f_2(\text{exec}(p)) = \bigcup_{j=1}^{jph} [\text{botj}(f_1(A, p, j) : topj(A, p, j) : s_2]\), which represents the elements of array C that are referred by iterations executed in \(PE_p\); and \(f_2(\text{exec(p)})\) indicates the set of PEs that store these elements of array C. Since \(\text{periodc,} = \text{period,} \times s_1/(Nbl s_2) = h_2\), recvc(p) can be represented by a union of at most h2 + 1 closed forms.

\[
\begin{align*}
\text{recv}_c(p) &= \bigcup_{j=0}^{jph} \bigcup_{k=0}^{kph} f^{-1}_A(\text{botj}(C, p, j) : topj(A, p, j)) \bigcup_j f^{-1}_A(\text{botj}(C, p, j) : topj(A, p, j)) \bigcup_j f^{-1}_A(\text{botj}(C, p, j) : topj(A, p, j)) \\
&= \bigcup_{j=0}^{jph} \bigcup_{k=0}^{kph} f^{-1}_A(\text{botj}(C, p, j) : topj(A, p, j)) \\
&= \bigcup_{j=0}^{jph} \bigcup_{k=0}^{kph} f^{-1}_A(\text{botj}(C, p, j) : topj(A, p, j)) \\
&= \bigcup_{j=0}^{jph} \bigcup_{k=0}^{kph} f^{-1}_A(\text{botj}(C, p, j) : topj(A, p, j))
\end{align*}
\]

Note that, in the above formula, the set \(f^{-1}_A(\text{botj}(C, p, j) : topj(A, p, j))\) consists of only one or two PEs. In addition, all these PEs are distinct. However, in spite of these facts, recv(p) cannot be represented by a constant number of closed forms independent of h2.

Third, we deal with sendc(p, q), which is equal to \(\text{localc}(p) \cap \text{exec}(q)\). This set will be represented by a union of three closed forms: sheadc(p, q), sbodyc(p, q), and sbodyc(p, q). Before deriving sendc(p, q), we show an example to explain where these three closed forms come from.

Example 4: Suppose that the number of PEs is 4; the loop body of a doall statement is \(A(11 + 2i) = g(C(2 + i))\), where \(g\) is a function; and \(u_1 = 745\). Then, \(l_1 = 11; s_1 = 2; l_2 = 2; s_2 = 1;\) and \(u_2 = 369\). If we let \(h_1 = 2\) and \(h_2 = 11\), then \(b_1 = 1; s_2 = 4\) and \(b_2 = 2\).

Figure 4 shows elements of array C in PEs and the corresponding PEs which will refer to these elements. Among them, sendc(0, 1) = sheadc(0, 1) \(\cup\) sbodyc(0, 1) \(\cup\) sbodyc(0, 1), where sheadc(0, 1) = \([7 : 8 : 1] \cup [15 : 16 : 1] : 21 : 8\); sbodyc(0, 1) = \([88 : 88 : 1] : 369 : 88\]; and sbodyc(0, 1) = \([95 : 96 : 1] : 109 : 8] : 369 : 88\]. sendc(0, 2) = sheadc(0, 2) \(\cup\) sbodyc(0, 2), where sheadc(0, 2) = \([2 : 2 : 1] \cup [9 : 10 : 1] : 21 : 8\] and sbodyc(0, 2) = \([89 : 90 : 1] : 109 : 8] : 369 : 88\]. Note that, sheadc(0, 1) is on purpose written by a union of two closed forms, as we will derive a unified formula to represent sheadc(p, q). Next, sbodyc(0, 2) = \(\phi\).
3.1. The processor which will access C(95)

Figure 4: Elements of array C in PE0, where array C is distributed by cyclic(22) over four processors. In addition, sendc(0, q) = sheadc(0, q) \cup shbodyc(0, q) \cup sbodyc(2)(0, q), for 1 \leq q \leq 3.

We notice that sheadc(p, q) is not empty if nzt(botl(C, p, kpj), l2, s2) < l2; sbodyc(p, q) includes some elements if botl(C, p, k) is in between botl(A, q, j) + 1 and topj(A, q, j) for some j and k; and sbodyc(p, q) will be evaluated without any conditions. In addition, the period of fi(exe(q)) is period* s2 = Ns2h1 and period* s2 = Ns2h1. Let k'pj = kpj + 1 if nzt(botl(C, p, kpj), l2, s2) < l2, k'pj = kpj otherwise. Then, we have

\[
\text{sheadc}(p, q) = \begin{cases} 
\text{botl}(A, q, j) \cup \text{top}(A, q, j + 1) : \text{top}(A, q, j + 1) : s2 ; \\
\text{botl}(C, p, k) : Ns2h1, \\
\text{phi}, \text{ otherwise.}
\end{cases}
\]

\[
\text{shbody}(p, q) = \begin{cases} 
\text{botl}(C, p, k') : \text{Nz}(\text{botl}(C, p, k'), l2, s2) ; \\
\text{top}(A, q, j) : Ns2h1, \\
\text{phi}, \text{ otherwise.}
\end{cases}
\]

\[
\text{send}(p, q) = \text{sheadc}(p, q) \cup \text{shbody}(p, q) \cup \text{sbodyc}(p, q).
\]

4.4 Experimental Studies

In this subsection, we present experimental studies of calculating a saxpy operation on two data arrays on a nCUBE-2 computer.

Example 5: Consider the following saxpy operation:

\[
d(0) = 0, 80639 \\
A(1997 + 3 * i) = A(1997 + 3 * i) + CON * C(5 + 2 * i),
\]

where CON is a floating-point constant. In addition, array A is distributed by a cyclic(b1) distribution; array C is distributed by a cyclic(b2) distribution. Table 6 lists experimental results of implementing this saxpy operation with various block sizes b1 and b2. Experimental results can be distilled as follows.

1. The execution time of computing the cases when b1 = s1 * h and b2 = s2 * h * h' is close to that of the cases when b1 = s1 * h * h' and b2 = s2 * h.

2. When h' is less than the number of PEs, then the execution time becomes better when h' is close to 1. This is because, in these cases, each PE needs to send data messages to at most (h' + 1) PEs. Therefore, the communication time reduces when h' becomes smaller.

3. When h' is larger than or equal to the number of PEs, then the execution time improves when block sizes b1 and b2 are increasingly larger. This may illustrate that our algorithm favors the cases when block sizes are large, because in these cases, the indexing overhead for packing data messages is not significant.

4. All cases except one show scalable improvements when the number of PEs grows. One exception is when the number of PEs is 16 and b1 = 3 and b2 = 10080. This is because in the extreme block to cyclic cases or cyclic to block cases, the indexing overhead for packing data messages is significant; in addition, the communication overhead also becomes worse when the number of PEs grows because of involving certain all-to-all communications.

5. Because the iteration space is linear and each PE executes roughly the same number of iterations, there is no load unbalance problem. Therefore, according to the communication oracle, it is preferable to choose large block sizes b1 and b2.
From Table 6, we can summarize that it is preferable to choose block sizes $b_1 \geq 63$ and $b_2 \geq 42$ for this saxpy operation.

<table>
<thead>
<tr>
<th>$b_2$</th>
<th>#PE</th>
<th>3</th>
<th>9</th>
<th>63</th>
<th>315</th>
<th>945</th>
<th>3792</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>1.334</td>
<td>0.702</td>
<td>0.476</td>
<td>0.248</td>
<td>0.140</td>
<td>0.081</td>
</tr>
<tr>
<td>8</td>
<td>1.168</td>
<td>0.173</td>
<td>0.125</td>
<td>0.085</td>
<td>0.077</td>
<td>0.074</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1.238</td>
<td>0.783</td>
<td>0.564</td>
<td>0.516</td>
<td>0.508</td>
<td>0.508</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.707</td>
<td>0.302</td>
<td>0.318</td>
<td>0.270</td>
<td>0.262</td>
<td>0.269</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.356</td>
<td>0.127</td>
<td>0.186</td>
<td>0.145</td>
<td>0.137</td>
<td>0.134</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.181</td>
<td>0.099</td>
<td>0.085</td>
<td>0.085</td>
<td>0.077</td>
<td>0.074</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>0.875</td>
<td>0.582</td>
<td>0.404</td>
<td>0.380</td>
<td>0.371</td>
<td>0.369</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.472</td>
<td>0.326</td>
<td>0.203</td>
<td>0.199</td>
<td>0.195</td>
<td>0.189</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.270</td>
<td>0.186</td>
<td>0.102</td>
<td>0.105</td>
<td>0.102</td>
<td>0.099</td>
<td></td>
</tr>
<tr>
<td>210</td>
<td>0.826</td>
<td>0.554</td>
<td>0.395</td>
<td>0.394</td>
<td>0.347</td>
<td>0.349</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.426</td>
<td>0.178</td>
<td>0.186</td>
<td>0.180</td>
<td>0.179</td>
<td>0.217</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.223</td>
<td>0.149</td>
<td>0.110</td>
<td>0.094</td>
<td>0.049</td>
<td>0.034</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.124</td>
<td>0.087</td>
<td>0.057</td>
<td>0.049</td>
<td>0.049</td>
<td>0.053</td>
<td></td>
</tr>
<tr>
<td>640</td>
<td>0.819</td>
<td>0.527</td>
<td>0.387</td>
<td>0.367</td>
<td>0.331</td>
<td>0.344</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.419</td>
<td>0.271</td>
<td>0.201</td>
<td>0.187</td>
<td>0.201</td>
<td>0.177</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.216</td>
<td>0.141</td>
<td>0.107</td>
<td>0.097</td>
<td>0.101</td>
<td>0.092</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>0.116</td>
<td>0.079</td>
<td>0.050</td>
<td>0.050</td>
<td>0.050</td>
<td>0.049</td>
<td></td>
</tr>
<tr>
<td>2520</td>
<td>0.819</td>
<td>0.529</td>
<td>0.394</td>
<td>0.363</td>
<td>0.356</td>
<td>0.369</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.417</td>
<td>0.268</td>
<td>0.188</td>
<td>0.187</td>
<td>0.187</td>
<td>0.188</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.213</td>
<td>0.139</td>
<td>0.103</td>
<td>0.099</td>
<td>0.098</td>
<td>0.084</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.113</td>
<td>0.076</td>
<td>0.056</td>
<td>0.056</td>
<td>0.050</td>
<td>0.048</td>
<td></td>
</tr>
<tr>
<td>5040</td>
<td>0.818</td>
<td>0.524</td>
<td>0.383</td>
<td>0.362</td>
<td>0.358</td>
<td>0.361</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.416</td>
<td>0.268</td>
<td>0.197</td>
<td>0.187</td>
<td>0.187</td>
<td>0.188</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.215</td>
<td>0.139</td>
<td>0.103</td>
<td>0.099</td>
<td>0.098</td>
<td>0.090</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.113</td>
<td>0.076</td>
<td>0.058</td>
<td>0.056</td>
<td>0.053</td>
<td>0.049</td>
<td></td>
</tr>
<tr>
<td>10080</td>
<td>0.817</td>
<td>0.522</td>
<td>0.380</td>
<td>0.361</td>
<td>0.357</td>
<td>0.358</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.415</td>
<td>0.268</td>
<td>0.196</td>
<td>0.186</td>
<td>0.184</td>
<td>0.187</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.212</td>
<td>0.138</td>
<td>0.102</td>
<td>0.097</td>
<td>0.097</td>
<td>0.098</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0.293</td>
<td>0.135</td>
<td>0.066</td>
<td>0.057</td>
<td>0.056</td>
<td>0.050</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Execution time (second) of computing the saxpy operation in Example 5 using 2 PEs, 4 PEs, 8 PEs, and 16 PEs, respectively.

5 Conclusions

Several techniques for determining data distribution and generating communication sets on distributed memory machines have been presented in this paper. First, we proposed a cost model which emphasized that the total execution time should include both the computation time and the communication time. This cost model was then used to determine the granularity of data distribution. We also extended Li and Chen's component alignment algorithm and developed a dynamic programming algorithm for heuristically determining whether data redistribution was necessary.

Second, we derived formulas to represent communication sets of executing doall statements. We found that there existed closed forms to represent communication sets if data arrays were distributed according to certain restrictions. Experimental studies also showed that the indexing overhead of the proposed closed forms was not significant and the approach scaled well as the number of processors increased.

References


