The Journal of Supercomputing, 25, 199–214, 2003

Symbolic Communication Set Generation for Irregular Parallel Applications

MINYI GUO minyi@u-aizu.ac.jp
Department of Computer Software, The University of Aizu, Aizu-Wakamatsu City, Fukushima, 965–8580, Japan

YI PAN pan@cs.gsu.edu
Department of Computer Science, Georgia State University, University Plaza, Atlanta, GA 30303, USA

ZHEN LIU liuzhen@cc.nias.ac.jp
Department of Environmental and Culture Sciences, Nagasaki Institute of Applied Science, Abe-machi, Nagasaki, 851-0193, Japan

Abstract. Communication set generation significantly influences the performance of parallel programs. However, studies seldom give attention to the problem of communication set generation for irregular applications. In this paper, we propose communication optimization techniques for the situation of irregular array references in nested loops. In our methods, the local array distribution schemes are determined so that the total number of communication messages is minimized. Then, we explain how to support communication set generation at compile-time by introducing some symbolic analysis techniques. In our symbolic analysis, symbolic solutions of a set of symbolic expression are obtained by using certain restrictions. We introduce symbolic analysis algorithms to obtain the solutions in terms of a set of equalities and inequalities. Finally, experimental results on a parallel computer CM-5 are presented to validate our approach.

Keywords: parallelizing compilers, irregular array references, communication optimization, symbolic analysis, distributed memory multicomputers compile-time optimization

1. Introduction

Parallelizing compiler is necessary to allow programs written in standard sequential languages to run efficiently on parallel machines. In order to achieve optimal performance, the compiler must be able to efficiently generate communication sets for nested loops. Parallelizing compilers that generate codes for each processor have to compute the sequence of local memory addresses accessed by each processor as well as the sequence of *sends* and *receives* for a given processor to access non-local data. The distribution of computation in most compilers follows the *owner-computes rule*. That is, a processor performs only those computations (or assignments) for which it owns the left-hand-side variable. Access to non-local right-hand-side variables is achieved by inserting *sends* and *receives*.

Communication overhead influences the performance of parallel programs significantly. According to Hockney's representation, communication overhead can be measured by a linear function of message length m, that is

$T_{comm} = T_s + m \times T_d$, where T_s is the start-up time and T_d is the per-byte transmission time. To achieve good performance, it is necessary to optimize communication by:

- exploiting local computation as much as possible,
- vectorizing and aggregating communication in order to reduce the number of communication steps, and
- reducing the message length in communication steps.

In order to compile a loop into a parallel code efficiently, one must generate the communication set for each processor at compile-time. If the loop bounds are constants and array subscripts are represented as linear (affine) functions of loop index variables, the problem is similar to compiling a typical HPF-style [12] assignment statement $A(l_1 : h_1 : s_1) = B(l_2 : h_2 : s_2)$, where $l_1, h_1, s_1 (l_2, h_2, s_2)$ are the lower bound, upper bound, and access stride of $A(B)$, respectively. There has been much research to generate the code including the communication for each processor for a given array statement with HPF-style data mappings [3, 10, 13, 21]. Moreover, the methods to determine data distribution schemes for regular loop nests are discussed by many researchers [10, 17, 18].

However, if the array subscript expressions are not of the linear form—called nonlinear, which appears in some irregular applications—the above mentioned techniques cannot be applied in this situation. Consider a loop nest with non-linear array referencing which is very similar to a code excerpt as found in the TRFD code of the Perfect benchmarks [1], shown in Figure 1. In the loop two array reference functions are $f = i_1 * (i_1 - 1)/2 + i_2$ and $g = i_2 * (i_2 - 1)/2 + i_1$, respectively. General affine communication set generation techniques cannot be applied to these kinds of irregular application because there is no affine relationship between the array global addresses of LHS and RHS. Communication generation for these kinds of loops has not received sufficient attention yet.

In this paper, we propose certain communication optimization techniques for the situation of irregular array references in nested loops. In our methods, the local array distribution schemes are determined so that the total number of communication messages is minimized. Then, we explain how to support communication set generation at compile-time by introducing symbolic analysis techniques. In our symbolic analysis, symbolic solutions of a set of symbolic expressions is obtained by

```
Example 1
 $L_1$: DO $i_1 = 1, N$
    $L_2$: DO $i_2 = 1, i_1$
          $IA = i_1 * (i_1 - 1)/2 + i_2$
          $IB = i_2 * (i_2 - 1)/2 + i_1$
    $S$:     $A(IA) = \mathfrak{F}(B(IB))$
          ENDDO
          ...
    ENDDO
```

Figure 1. Sample loop nest with irregular array references.

employing certain restrictions. Experiments demonstrating the effectiveness of our approach to parallelizing compilers are described.

The remainder of this paper is organized as follows: Section 2 illustrates the need for communication optimization of irregular array references and introduces background knowledge used in the subsequent sections. Section 3 outlines how to determine array distribution schemes for irregular loops. Section 4 describes a symbolic analysis method for communication set generation. Experimental evaluations are shown in Section 5. Section 6 introduces related work in this area. Finally, Section 7 presents the conclusions.

2. Problem description and preliminaries

Given a perfectly nested loop $\mathscr{L}$ as shown in the following.

```
$L_1$ : DO $i_1 = X_1, Y_1, Z_1$
  ......
$L_n$ : DO $i_n = X_n, Y_n, Z_n$
$S$ :    $A(f(i_1, i_2, \ldots, i_n)) =$
            $\mathscr{F}(B(g(i_1, i_2, \ldots, i_n)))$
       ENDDO
   ......
  ENDDO
```

For the sake of simplicity, we will assume that the referenced array A and B have only one dimension. The array access functions (f and g), the loop's lower and upper bounds (X_i, Y_i), and stride (Z_i) may be arbitrary symbolic expressions made up of loop-invariant variables and loop indices of enclosing loops. We will also assume that all loop strides are positive. It is not difficult to extend our method to handle imperfectly nested loops, negative strides, multidimensional arrays, and loop-variant variables. Furthermore, let the arrays A and B be distributed in a block-cyclic fashion with block sizes of β_1 and β_2, respectively, across P processors. These are also known as *cyclic*(β_1) and *cyclic*(β_2) distributions [12, 14].

We assume that the array access functions f and g are non-linear functions. Non-linear subscript functions are commonly caused by induction variable substitution, linearizing arrays, parameterizing parallel programs with a symbolic number of processors and problem sizes, and so forth.

Compilers currently parallelize irregular references using the *inspector* and *executor* approach. The inspector phase partitions loop iterations, allocates local memory for each unique nonlocal array element accessed by a loop, and builds a communication schedule to prefetch required nonlocal data. In the executor phase, the actual communication and computation are carried out. This approach was adopted by the CHAOS run-time library [20]. The inspector-executor approach incurs significant overhead caused by the inspector and by mapping nonlocal indices into local buffers. The inspector must be re-executed each time the access pattern changes. In this approach, the algorithm for communication set generation is shown in Figure 2. (The meaning of the sets *Index* and *Send* are explained below.)

In order to reduce the overhead for executing irregular loops, we need to decide the distribution schemes β_1 and β_2 so that the total number of communication steps and the amount of message lengths are minimized because the communication patterns are different when using different data distribution schemes, even for the same loop nest and the same arrays A and B. Then, we will compute the necessary communication sets for each processor. That is, we must be able to obtain an array subscript set for a processor pair (p, q), named $Send(q, p)$, which represents the elements of array B that are sent from q to p. We only generate sending communication set in this paper because receiving set generation is very similar to the algorithm for sending set generation.

In our previous work [7, 8], array elements distributed on a specific processor can be represented as a 4-tuple $\delta = (o, b, s, m)$, where o is the starting subscript of the global array elements distributed on that processor; b the length of the block; s the stride between two consecutive blocks; and m is the number of blocks distributed onto the processor. For instance, if an array with the size 180 is distributed across 4 processors with the distribution scheme $cyclic(3)$, the local data owned by processor P_0 and P_1 can be expressed as $\delta_0 = (0, 3, 12, 15), \delta_1 = (3, 3, 12, 15)$, etc. A δ

```
Input:
• the distribution scheme $cyclic(\beta_1)$ and $cyclic(\beta_2)$;
• the loop $\mathcal{L}$'s information (including the number of loop nests, the bounds and
  strides of each loop nest, and the loop body);
• the source, destination processor numbers $p, q$;
Output: the communication set $Send(q, p)$;

1. initialization: $Index(A, p) \leftarrow \{\}$; $Index(B, q) \leftarrow \{\}$; $Send(q, p) \leftarrow \{\}$;
2. compute the solutions $(i_1, i_2, \ldots, i_n)$ where $A(f(i_1, \ldots, i_n))$ are distributed onto
$p$;
   for $i_1 = X_1, Y_1, Z_1$
     ...
     for $i_n = X_n, Y_n, Z_n$
       if $p = (f(i_1, \ldots, i_n)$ div $\beta_1)$ mod $P$ then
         $Index(A, p) = Index(A, p) \cup \{(i_1, \ldots, i_n)\}$;
       endif
       if $q = (g(i_1, \ldots, i_n)$ div $\beta_2)$ mod $P$ then
         $Index(B, q) = Index(B, q) \cup \{(i_1, \ldots, i_n)\}$;
       endif
     endfor
     ...
   endfor
3. for each processor pair $(p, q)$, compute $Send(q, p)$:
   for $(i_1, \ldots, i_n) \in Index(A, p)$
     if $(i_1, \ldots, i_n) \in Index(B, q)$ then
       $Send(q, p) = Send(q, p) \cup \{g(i_1, \ldots, i_n)\}$;
     endif
   endfor
```

Figure 2. Inspector-executor method of communication set generation.

corresponds to a set of the global array subscript defined as $S_\delta = \{i|o + s * k \leq i < o + b + s * k, 0 \leq k < m\}$. Intuitively, δ can represent the set of elements of an array owned by a processor under any regular distribution. Furthermore, if $\delta(A,p)$ represents the 4-tuple of the array A distributed onto processor p, then $Index(A,p)$—a set of loop indices $(i_1, i_2, \ldots, i_n)$ where $A[f \times (i_1, \ldots, i_n)]$ is distributed on p—can be defined as

$$Index(A,p) = \{(i_1, \ldots, i_n)|f(i_1, \ldots, i_n) \in S_{\delta(A,p)}\}.$$

Thus, the communication set of processor pair (p,q) is

$$Send(q,p) = \{g(i_1, \ldots, i_n)|(i_1, \ldots, i_n) \in Index(A,p) \wedge g(i_1, \ldots, i_n) \in S_\delta(B,q)\}.$$

These formulas will be used in our symbolic communication generation algorithm.

Returning to Example 1, let $N = 15$, $P = 4$, $\beta_1 = 3$, $\beta_2 = 2$ (Figure 3(a)). According to the algorithm shown in Figure 2, we can compute all $Send(q,p), 0 \leq p, q \leq 3$. For instance, for the elements which must be sent from P_3 to P_1, it is $Send(3,1) = \{6, 7, 15, 22, 31, 38\}$. Figure 3(b) shows the loop indices i_1 and i_2 and the corresponding values of IA and IB. The items in the small boxes are those indices for which $B[IB]$ must be sent to P_1 from P_3.

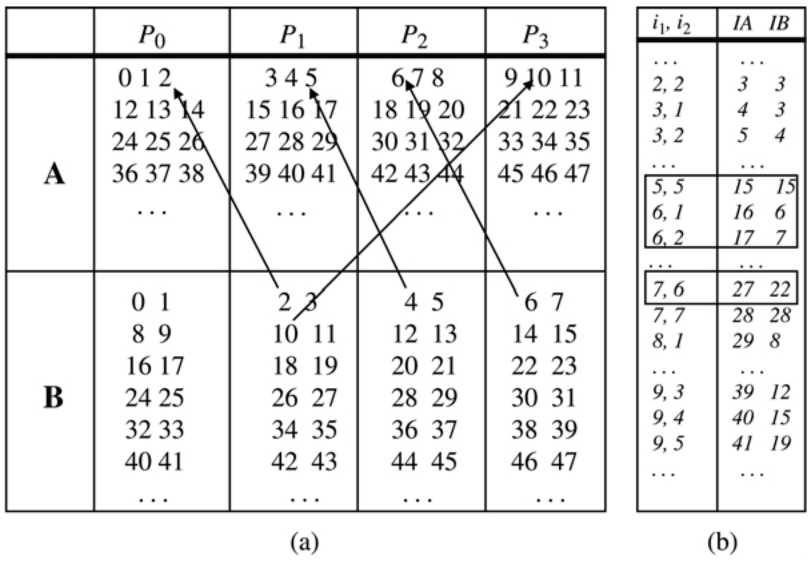
Although we can determine the sending communication set through the above algorithm, two serious problems result in the above approach not being able to be practically applied. One is that this approach can only be invoked at run-time execution when the bounds of nested loops include non-constants, and the other is that the computation complexity of the algorithm is $O(y^n)$ (assuming that the average iteration of each loop is y). The high computation overhead is intolerable for parallel program execution.

3. Selecting array distribution schemes for irregular loops

As mentioned in the introduction, an effective data distribution strategy can reduce the number of inter-processor communications and total message length. Assuming that, in an iteration $(i_1, \ldots, i_n)$, if $A[f(i_1, \ldots, i_n)]$ and $B[g(i_1, \ldots, i_n)]$ are distributed onto the same processor, communication is not needed. For regular loops, based on analysis of affine subscripts, some data distribution techniques, such as the constraint-based method [10], and linear algebraic frameworks [14, 18], are proposed to maximize local accesses and minimize remote accesses (inter-processor communication).

With respect to an irregular reference loop, if the loop is interlined between two regular loops, the arrays in the irregular loop must be distributed in the same way as the previous scheme in order to avoid redistribution overhead, because the reduced cost of communication may be larger than the redistribution cost. However, if an irregular loop is independent or in the first loop nest, we must determine the distribution schemes for arrays in the loop to optimize communications.

Given the global address of an array element, we can easily determine the processor that owns this element and the local address of the element on that



	P_0	P_1	P_2	P_3
A	0 1 2 12 13 14 24 25 26 36 37 38 ...	3 4 5 15 16 17 27 28 29 39 40 41 ...	6 7 8 18 19 20 30 31 32 42 43 44 ...	9 10 11 21 22 23 33 34 35 45 46 47 ...
B	0 1 8 9 16 17 24 25 32 33 40 41 ...	2 3 10 11 18 19 26 27 34 35 42 43 ...	4 5 12 13 20 21 28 29 36 37 44 45 ...	6 7 14 15 22 23 30 31 38 39 46 47 ...

(a)

i_1, i_2	IA IB
...	...
2, 2	3 3
3, 1	4 3
3, 2	5 4
...	...
5, 5	15 15
6, 1	16 6
6, 2	17 7
...	...
7, 6	27 22
7, 7	28 28
8, 1	29 8
...	...
9, 3	39 12
9, 4	40 15
9, 5	41 19
...	...

(b)

Figure 3. Communication set of the sample loop in Example 1. (a) Array distribution with $P = 4,\ \beta_1 = 3,\ \beta_2 = 2$. The arrows indicate that the array elements must be sent to the remote processor when the loop is executed. (b) The values of IA, IB and the corresponding loop indices i_1, i_2 for which $A[IA]$ are distributed on P_1. Furthermore, the items in the small boxes are the indices whose elements must be sent to P_1 from P_3.

processor using the expressions

$$EQ_{proc}(p, i) : p = (i \text{ div } \beta) \text{ mod } P,$$
$$EQ_{loc}(L, i) : L = \beta^*(i \text{ div } P\beta) + i \text{ mod } \beta,$$

where i is a global address, p the processor to which i belongs, P the number of processors executing the parallel program, L the local address of i on processor p, and β is the distribution scheme of the array. For the purpose of local execution of the statement S in loop $\mathscr{L}$, we must compute β_1 and β_2 for which the following formula is true:

$$\exists\ \beta_1, \beta_2.\ \max(|\{(i_1, \ldots, i_n)|(f/\beta_1) \text{ mod } P = (g/\beta_2) \text{ mod } P,$$
$$1 \le \beta_1, \beta_2 \le \frac{N_1}{P}, \frac{N_2}{P}, (\vec{X} \le \vec{I} \le \vec{Y}\}|).$$

The β_1 and β_2 that satisfy the above formula can easily be obtained. However, even at the compile-phase, the computing time may be very long. In order to efficiently decide the distribution schemes, we should pay attention to the fact that because of random accesses of array elements, the optimal distribution schemes for a small range should also be suitable for a larger range. This conjecture is verified by our experiments. For Example 1, when N is selected as 500, 1,000, or 5,000, the best calculated values of β_1 and β_2 all show 7 and 3, respectively. Thus, we need not to traverse β_1 and β_2 within the entire range ($\vec{X} \leq \vec{I} \leq \vec{Y}$), but just to calculate them in a small range. Optimal distribution schemes can be determined at compile-time.

4. Symbolic analysis methods for generating communication sets

In order to compute communication sets when array subscripts and loop bounds are symbolic and nonlinear expressions, a symbolic analysis method is proposed. A symbolic expression may consist of arbitrary arithmetic operators and operands that can be array subscripts, loop indices, loop bounds and strides, integer constants or infinity symbols $(-\infty, +\infty)$.

A *restriction* is a set of symbolic qualities and inequalities defined over loop variables and parameters (loop invariant) which are commonly derived from loop bounds, array subscript expressions, conditional statements, data declarations, and data and computation distribution schemes of a program. In this paper, the symbolic variables are designated as loop indices. The integer solution to a set of restrictions is a set of loop indices satisfying all of the constraints.

There are three sub-restrictions. They are:

- loop bounds and control-flow restriction C_1,
- array A reference and distribution restriction C_2, and
- array B reference and distribution restriction C_3.

To generate communication sets, our goal is to solve the symbolic solution vector $\vec{I}$ to satisfy restriction $C = C_1 \cap C_2 \cap C_3$.

4.1. *Simplifying symbolic expressions*

The first sub-restriction is derived from the loop bounds and control-flow (e.g., condition statements). If there are no control-flow constraints in the loop body, only loop bounds determine the restriction C_1, that is, $C_1 = \{X_j \leq i_j \leq Y_j, (i_j - X_j) \bmod Z_j = 0, 1 \leq j \leq n\}$. Before computing expressions, in order to solve the equations as easily as possible, we must simplify symbolic expressions. The simplification can be handled based on a set of rules of simplification. After simplifying this restriction, the initial lower and upper bounds, $low(i)$ and $up(i)$, for each symbolic variable i can be deduced. Also, we use $eva(i)$ to represent the evaluated symbolic value under the restrictions. Thus we can obtain $low(i) \leq eva(i) \leq up(i)$. Furthermore, we define $eva(f(i_1, \ldots, i_n)) = f(eva(i_1), \ldots, eva(i_n))$.

With respect to Example 1, for instance, $C_1 = \{1 \leq i_1 \leq N, 1 \leq i_2 \leq i_1\}$. Then $low(i_1) = 1, up(i_1) = N, low(i_2) = 1, up(i_2) = i_1$.

For the k-th block of the array A distributed onto processor p, we have

$$C_2 = \{p = (f(\vec{I}) \text{ div } \beta_1) \text{ mod } P\}, \text{or} \tag{1}$$
$$C_2 = \{LB_k \leq f(\vec{I}) \leq UB_k, 0 \leq k < m_p\}. \tag{2}$$

Also for the k-th block of the array B distributed onto processor q, we have

$$C_3 = \{q = (g(\vec{I}) \text{ div } \beta_2) \text{ mod } P\}, \text{or} \tag{3}$$
$$C_3 = \{LB'_k \leq g(\vec{I}) \leq UB'_k, 0 \leq k < m_q\}, \tag{4}$$

where $LB_k = o_p + s_p * k, UB_k = o_p + b_p + s_p * k, LB'_k = o_q + s_q * k$, and $UB'_k = o_q + b_q + s_q * k$, and $o_p, b_p, s_p, m_p(o_q, b_q, s_q, m_q)$ are starting subscript, block size, stride, and the number of blocks for processor $p(q)$ in the 4-tuple $\delta_p(\delta_q)$.

Our evaluation method cyclically replace the old, wide bounds accordingly from the restriction C_2, until the greatest lower and least upper bounds for a symbolic variable are found. The most intuitive method to replace its bounds is to physically substitute each occurrence of the variable in the given expression with the bounds of the variable, then to simplify the resulting expression until the minimum range can be obtained. The replacement process progressively applies rewrite rules at each point where the variable is replaced by its bounds. The rewrite rules are displayed in Figure 4. All these rewrite rules are in the form of a range of inequalities. If the derived new range is wider than the old one, the replacement does not occur. A similar method is applied to restriction C_3.

Let us re-examine Example 1. If we want to generate the communication set $Send(3, 1)$, the symbolic solutions of the variables i_1 and i_2 must be solved from restrictions C_2 and C_3, where $C_2 = \{3 + 12 * k \leq (i_1 * (i_1 - 1))/2 + i_2 \leq 5 + 12 * k\}$ and $C_3 = \{6 + 8 * k \leq (i_2 * (i_2 - 1))/2 + i_1 \leq 7 + 8 * k\}$. We first evaluate the solution of inequality $(i_1 * (i_1 - 1))/2 + i_2 \geq 3 + 12 * k$, using our rewrite rule system, $eva((i_1 * (i_1 - 1))/2 + i_2) \geq eva(3 + 12 * k) \Rightarrow eva((i_1 * (i_1 - 1))/2) + eva \times (i_2) \geq 3 + 12 * k$. In order to solve this inequality, we need to eliminate one of the variables i_1 and i_2 in the inequality. Notice that $1 \leq eva(i_2) \leq i_1$. If the left side of the inequality is a monotonic function of i_2, we can replace $eva(i_2)$ by its upper bound i_1. The following subsection discusses the monotonicity replacement in our symbolic system.

4.2. *Monotonicity replacement*

In some cases, using the replacement as shown above cannot determine the exact lower and upper bounds for an expression. To determine these bounds, one needs to observe whether $f(i_1, \ldots, i_n)$ is monotonic for i_k. We can define f as monotonically non-decreasing or non-increasing for $\vec{I}$ as follows.

Definition A function $f(\vec{I})$ is monotonically non-decreasing for index i_k iff $f \times (i_1, \ldots, i_k, \ldots, i_n) \leq f(i_1, \ldots, j_k, \ldots, i_n)$ wherever $X_k \leq i_k \leq j_k \leq Y_k$.

$$
\begin{array}{ll}
eva(c) = c & ;c \text{ is an integer constant} \\
eva(i) = i & ;i \text{ is a symbolic variable} \\
low(i) \le eva(i) \le up(i) & ;\text{initial inequality} \\
-up(i) \le eva(-i) \le -low(i) & \\
low(i) + low(j) \le eva(i+j) \le up(i) + up(j) & \\
low(i) - up(j) \le eva(i-j) \le up(i) - low(j) & \\
low(i) * low(j) \le eva(i * j) \le up(i) * up(j) & \\
\frac{low(i)}{up(j)} \le eva\left(\frac{i}{j}\right) \le \frac{up(i)}{low(j)} & \\
(min(low(i)^c, up(i)^c) \le eva(i^c) & \\
\le max(low(i)^c, up(i)^c) &
\end{array}
$$

Figure 4. Rewrite rules for simplifying symbolic expressions. The variables i and j are non-negative integers.

Similarly, a function $f(\vec{I})$ is monotonically non-increasing for index i_k iff $f \times (i_1, \ldots, i_k, \ldots, i_n) \ge f(i_1, \ldots, j_k, \ldots, i_n)$ wherever $X_k \le i_k \le j_k \le Y_k$.

Determining whether $f(i)$ is monotonically non-decreasing or monotonically non-increasing is not difficult. One can prove that $f(x)$ is monotonically non-decreasing for x by proving that $f(x+1) - f(x) \ge 0$.

The range determination of a loop index i_k is easier if the access functions f and g are monotonically non-decreasing (non-increasing) for i_k because we can replace $eva(i_k)$ simply with its known lower or upper bound in Formula (2). That is, if f is monotonically non-decreasing for i_k,

$$
\begin{aligned}
& LB_k \le eva(f(i_1, \ldots, i_j, \ldots, i_n)) \le UB_k \Rightarrow \\
& LB_k \le f(eva(i_1), \ldots, eva(i_j), \ldots, eva(i_n)) \\
& \qquad \le f(eva(i_1), \ldots, up(i_j), \ldots, eva(i_n) \wedge \\
& UB_k \ge eva(f(i_1, \ldots, i_j, \ldots, i_n)) \\
& \qquad \ge f(eva(i_1), \ldots, low(i_j), \ldots, eva(i_n)).
\end{aligned} \tag{5}
$$

Let us continue to consider Example 1. Because $i_1 * (i_1 - 1)/2 + i_2$ is a monotonic function of i_2 and $eva(i_2) \le up(i_2)$, then $eva((i_1 * (i_1 - 1))/2) + up(i_2) \ge eva((i_1 * (i_1 - 1))/2) + eva(i_2) = eva((i_1 * (i_1 - 1))/2 + i_2) \ge 3 + 12 * k \Rightarrow i_1 * (i_1 - 1)/2 + i_1 \ge 3 + 12 * k \Rightarrow i_1^2 + i_1 \ge 6 + 24 * k$.

Similarly, because $eva(i_2) \ge 1$, we have $i_1 * (i_1 - 1)/2 + 1 \le 5 + 12 * k \Rightarrow i_1^2 - i_1 \le 8 + 24 * k$. Thus, for $k = 0, 1, 2, \ldots$, the variable i_1 can be restricted to the smaller ranges of its values:

$$
\begin{array}{ll}
k = 0, & 2 \le i_1 \le 3 \\
k = 1, & 5 \le i_1 \le 6 \\
k = 2, & 7 \le i_1 \le 8 \\
k = 3, & 9 \le i_1 \le 9. \\
\cdots\cdots & \cdots\cdots
\end{array}
$$

A similar method can be applied to restriction C_3. Finally, a set of the loop indices (i_1, i_2) that satisfy all the restrictions is derived from this symbolic method. Then, all $B[g(i_1, i_2)]$, where i_2 and i_2 satisfy the restriction C, are the elements which must be sent from processor P_3 to processor P_1.

4.3. *Algorithm*

The algorithm of communication set generation using symbolic expression evaluation is shown in Figure 5.

Step 3 describes the core of the algorithm. Executing this step assures that only one symbolic variable occurs in the symbolic expression set. Steps 3.1 and 3.2 are used to replace symbolic variables by some symbolic constants. Step 4 obtains the greatest lower bound and least upper bound for each symbolic variable i using rewrite rules and symbolic replacement. After determining the range of all the symbolic variables $(i_1, \ldots, i_n)$, Step 5 extracts the subset of the values of $(i_1, \ldots, i_n)$ that satisfies the restriction C_2 and C_3. Notice that the order in which one replaces bounds for variables is very important. A poorly chosen replacement order may require more replacements and result in less accurate bounds [**2**].

```
Input: the restriction $C$;
Output: $Send(q, p)$;

1. initialization: $Send(q, p) \leftarrow \{\}$, $I \leftarrow \{i_1, \ldots, i_n\}$;
2. yield the initial bounds $low(i)$, $up(i)$ for each loop index $i$ (restricted by $C_1$);
for each symbolic variable $i$
3. repeat
      select an $i_k \in I$,
        if $f$ is monotonically for $i_k$ then
3.1       using Formula (5) to replace $eva(i_k)$ with its bounds $low(i_k)$, $up(i_k)$;
        else
3.2       using rewrite rules to replace $eva(i_k)$ with its bounds $low(i_k)$, $up(i_k)$;
        endif
      $I = I - \{i_k\}$;
  until only one $i \in I$;
4. simplify the inequalities, generating the new $low(i)$, $up(i)$;
end for
5. while $(i_1, \ldots, i_n) \in \{low(i_1) \ldots up(i_1)\} \times \cdots \times \{low(i_n) \ldots up(i_n)\}$
      satisfies Formula (1) do
     if $g(i_1, \ldots, i_n)$ satisfies the Formula (3) and (4) then
       $Send(q, p) = Send(q, p) \cup \{g(i_1, \ldots, i_n)\}$;
     endif
   endwhile
```

Figure 5. Symbolic communication set generation algorithm.

5. Experiments

We have evaluated our symbolic analysis algorithm on a 32-node distributed memory parallel computer CM-5, using the MPI communication library and using the `gettimeofday()` system call to measure execution times. We selected a subroutine OLDA from the code TRFD, appearing in Perfect benchmark [1]. A simplified version of this loop nest is shown in the left side of Figure 6. After using induction variable substitution to replace the induction variable *mrsij* in statement S_1, the optimized version is shown in the right side of Figure 6. There is a nonlinear array subscript for *xrsij* in S_2. To parallelize this loop nest, the communication set generation and address translation routine must be used.

The best distribution case of *cyclic*(2) for array *xrsij*, and *cyclic*(4) for *xij* is selected when $N = 16$ (with global array size 18,632) and the best distribution case of *cyclic*(3) and *cyclic*(7) is selected when $N = 20$ (with global array size 44,310). Figures 7 and 8 show the total loop execution times when $N = 16$ and $N = 20$, respectively. The terms *runtime algo* and *symbolic algo* indicate communication set generation using the runtime algorithm and symbolic algorithm, respectively. We observe that as the number of nodes increases, a proportionate improvement in execution time is not obtained because each processor has to communicate with an increasing number of nodes. Although the communication set generation only involves the computation overhead, the performance can also be improved using the symbolic analysis algorithm.

Figure 9 is the measured result for computation part of the communication generation. From this figure we can observe that because the local array size decreases as the number of nodes increases, the performance gap becomes even larger. The symbolic communication generation algorithm can improve the computation overhead by about 50% to 80% when parallel programs are executed using different numbers of processors.

```
mrsij0 = 0                                        DO mrs = 0, (N*N+N)/2-1
DO mrs = 0, (N*N+N)/2-1                              DO mi = 0, N-1
   mrsij = mrsij0                                       DO mj = 0, mi-1
   DO mi = 0, N-1                                 S1:      mrsij = (mi*mi+mi+              &
      DO mj = 0, mi-1                                           mrs*(N*N+N))/2+mj+1
S1:      mrsij = mrsij+1                 ⟹        S2:      xrsij(mrsij) = xij(mj)
S2:      xrsij(mrsij) = xij(mj)                         END DO
      END DO                                         END DO
   END DO                                         END DO
   mrsij0 = mrsij0+(N*N+N)/2
END DO
```

Figure 6. Simplified version of loop nest OLDA from TRFD.

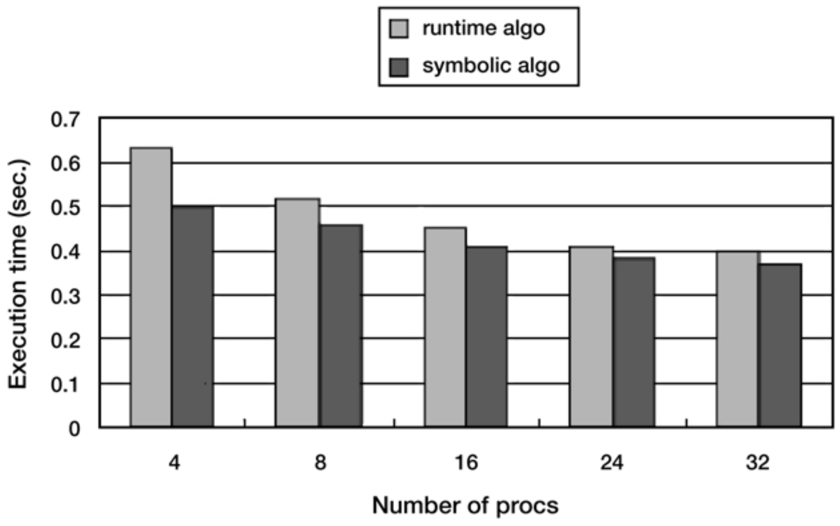


Figure 7. Results of TRFD loop nest OLDA when $N = 16$, $\beta_1 = 2, \beta_2 = 4$ on CM-5.

6. Related work

Much research has focused on the problem of communication set generation under regular array reference in parallel loop nests, or array statements such as $A(l_1 : u_1 : s_1) = B(l_2 : u_2 : s_2)$ in data-parallel languages such as HPF and Fortran D [12], with block-cyclic distribution. For instance, Gupta et al. [11] proposed closed

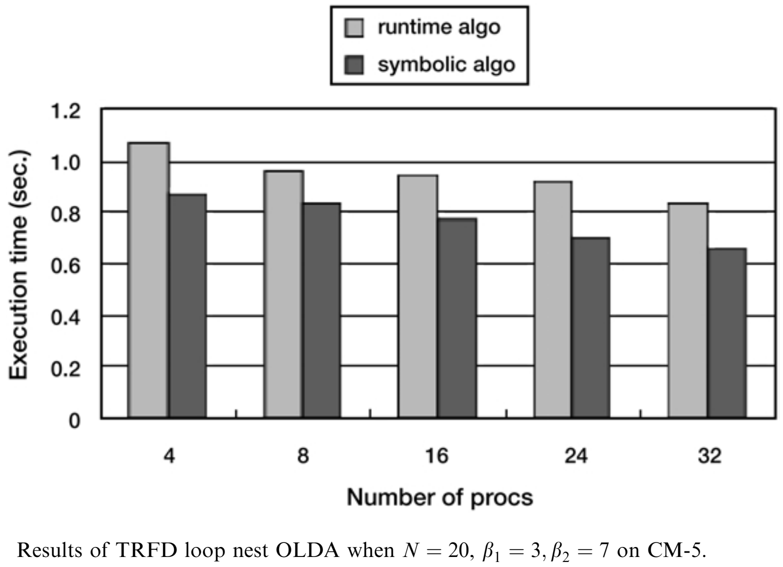


Figure 8. Results of TRFD loop nest OLDA when $N = 20$, $\beta_1 = 3, \beta_2 = 7$ on CM-5.

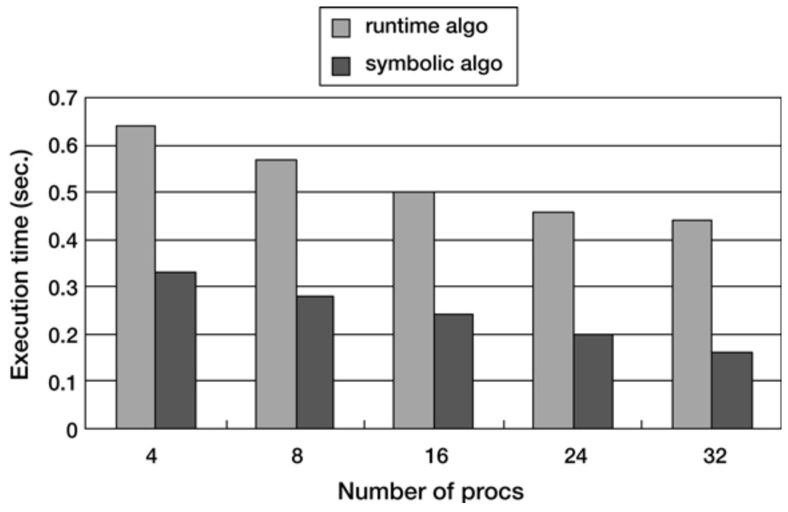


Figure 9. Computation overhead for generating communication sets of TRFD loop nest OLDA when $N = 20$, $\beta_1 = 3, \beta_2 = 7$ on CM-5.

forms for representing communication sets. These closed forms are then used with a virtual processor approach to obtain a solution for arrays with *block-cyclic* distribution [10]. Chatterjee et al. [3] enumerated the local memory access sequence based on a finite-state machine (FSM). Their run-time algorithm involves a solution of b_1 linear Diophantine equations to determine the pattern of accessed addresses, followed by the sorting of these addresses to derive the accesses in a linear order. Kennedy et al. [13] adopted an integer lattice method to generate the memory access sequence. Recently, Tseng and Gaudlot [22] derived an algebraic solution for an integer lattice that models the communication set using the Smith-Normal-Form analysis, thus generating the enumeration of the communication set. In this approach, the authors claimed that the SPMD program can be constructed without any inspector-like run-time codes. Paek et al. [19] presented a compiler framework for communication generation that has the potential to reduce the time-consuming hand-tuning that would otherwise be necessary to achieve good performance for non-cache-coherent multiprocessors.

However, little research gives attention to the problem of generating communication for irregular access in loop nests. Lain et al. [17] implemented a library, called PILAR, to exploit regularity in irregular applications. Lain et al. [17] dealt with irregular applications through detection of irregularity, feasibility of inspection, and placement of inspectors and interprocessor communication schedules. They presented a number of internal representations suited to particular access patterns and showed how various preprocessing structures such as translation tables, trace arrays, and interprocessor communication schedules can be encoded in terms of one or more of these representations. The CHAOS/PARTI library [20], and in particular, the original PARTI library, had a significant impact on the design of PILAR. Similarly, LPARX [15] is a C++ library that provides run-time support for dynamic, block-structured, irregular problems in a variety of platforms. Our recent work tried to reduce the communication cost of irregular loop partitioning. We partitioned a loop iteration to a processor for which the minimal communication cost is ensured when executing that iteration. Then, after all iterations are partitioned into various processors, we gave global vs. local data transformation rules, indirection array remapping, and communication optimization methods [9].

Many researchers have worked on deriving techniques for the automatic decision of data distribution,. Gupta and Banerjee [10] described the PARADIGM compiler that decomposes the data distribution problem into a number of sub-problems, each dealing with a different distribution parameter for all the arrays of the input program (align pass, block-cyclic pass, block-size pass, and num-proc pass). Lim and Lam [18] presented an algorithm to find the optimal affine transform that maximizes the degree of parallelism while minimizing the degree of synchronization in a program with arbitrary loop nests and affine data accesses. Garcia et al. [6] presented a framework for automatic data mapping that is based on the alignment, distribution, and redistribution problems being solved together using a single graph representation called the communication parallelism graph (CPG). This graph is the structure that holds symbolic information about the potential data movement and parallelism inherent to the whole program. The CPG is then particularized for a given problem size and target system and used to find a minimal cost path through the graph using a general purpose linear 0–1 integer programming solver.

With respect to symbolic analysis, Blume and Eigenmann [2] used a symbolic analysis technique to test data dependence for irregular access. They also developed a method called range propagation to compare symbolic expressions. Fahringer [4] and Fahringer and Scholz [5] proposed techniques for counting the number of solutions to a system of symbolic equations, simplifying systems of constraints, and determining the relationship among symbolic expressions, and the condition under which control flow reaches a program statement. They introduced the program context, a novel representation for comprehensive and compact control as well as data flow analysis information. Haghighat and Polychronopoulos [16] presented a dependence test to handle nonlinear, symbolic expressions. Their algorithm is essentially a symbolic version of Banerjee's inequalities test. Most of existing studies focused on the comparison of symbolic expressions to test irregular dependence, and there is no technique to obtain the solutions of symbolic restrictions. Our current work tries to address this issue.

7. Conclusions

Communication set generation influences the performance of parallel programs significantly. In this paper, we have proposed a symbolic analysis method to generate communication sets for irregular array references. In our approach, the local array distribution schemes are determined first so that interprocessor communication is reduced as much as possible. Then we introduced symbolic analysis techniques to generate the communication set for irregular loop execution. We proposed an algorithm for obtaining the symbolic values and deflating the value of loop indices and array subscripts.

This technique overcomes the drawback of existing libraries where significant overhead is caused by the inspector and by mapping nonlocal indices into local buffers. It completes the computation for generating communication at compile-time as much as possible. Thus, the whole performance of parallel programs including loop nests with nonlinear array references can be enhanced.

Acknowledgment

This research was supported in part by The Japanese Okawa Foundation for Information and Telecommunications under Grant Program 01-12, and by the U.S. National Science Foundation under Grant ECS-0196569.

References

1. M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orzag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: effective performance evaluation of supercomputers. *International Journal of Supercomputing Applications*, 3(3):5–40, 1989.
2. W. Blume and R. Eigenmann. Nonlinear and symbolic data dependence testing. *IEEE Transactions on Parallel and Distributed Systems*, 9(12):1180–1194, Dec. 1998.
3. 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. *Journal of Parallel and Distributed Computing*, 26:72–84, 1995.
4. T. Fahringer. Efficient symbolic analysis for parallelizing compilers and performance estimators. *The Journal of Supercomputing*, 12:227–252, 1998.
5. T. Fahringer and B. Scholz. A unified symbolic evaluation framework for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 11(11):1105–1125, 2000.
6. J. Garcia, E. Ayguade, and J. Labarta. A framework for integrating data alignment, distribution, and redistribution in distributed memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 12(4):416–431, 2001.
7. M. Guo, I. Nakata, and Y. Yamashita. Contention-free communication scheduling for array redistribution. *Parallel Computing*, 26(2000):1325–1343, 2000.
8. M. Guo, Y. Yamashita, and I. Nakata. Efficient implementation of multi-dimensional array redistribution. *IEICE Transactions on Information and Systems*, E81-D(11), Nov. 1998.
9. M. Guo, Z. Liu, C. Liu, and L. Li. Reducing communication cost for parallelizing irregular scientific codes. *The 6th International Conference on Applied Parallel Computing*, Finland, Lecture Notes in Computer Science 2367, pp. 203–216, June 2002.
10. M. Gupta, and P. Banerjee, PARADIGM: A compiler for automatic data distribution on multicomputers. *Proc. 1993 ACM International Conference on Supercomputing*, ACM, pp. 87–96, 1993.
11. S. K. S. Gupta, S. S. Kaushik, C. H. Huang, and P. Sadayappan. Compiling array expressions for efficient execution on distributed-memory machines. *Journal of Parallel and Distributed Computing*, 32:155–172, 1996.
12. C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*, The MIT Press, 1994.

13. K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient address generation for block-cyclic distributions. In *Proc. of ACM International Conf. on Supercomputing*, pp. 180–184. Barcelona, Spain, July 1995.
14. U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle, Automatic data layout for distribute-memory machines in the D programming environment. In C. W. Kessler, ed., *Automatic Parallelization-New Approaches to Code Generation, Data Distribution, and Performance Prediction*, Vieweg Advanced Studies in Computer Science, Verlag Vieweg, Wiesbaden, Germany, pp. 136–147, 1993.
15. Scott R. Kohn and Scott B. Baden. A robust parallel programming model for dynamic non-uniform scientific computations. In *Proc. SHPCC*, 1994.
16. M. Haghighat and C. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization, and scheduling of programs. In *Proc. Sixth Ann. Workshop Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.
17. A. Lain, D. R. Chakrabarti, and P. Banerjee. Compiler and run-time support for exploiting regularity within irregular applications. *IEEE Transactions on Parallel and Distributed Systems*, 11(2), February, 2000.
18. A. W. Lim, and M. S. Lam, Maximizing parallelism and minimizing synchronization with affine transforms. *Proc. 24th Annual ACM SIGPALN-SIGART Symposium on Principles of Programming Languages*, Paris, France, January 1997.
19. Y. Paek, A. Navarro, E. Zapata, J. Hoeflinger, and D. Padua. An advanced compiler framework for non-cache-coherent multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):241–259, 2002.
20. R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. In *Proc. SuperComputing'93*, pp. 361–370, November 1993.
21. A. Thirumalai, J. Ramanujam, and A. Venkatachar. Communication generation for data-parallel programs using integer lattices. In P. Sadayappan et al., ed., *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, Springer-Verlag, 1996.
22. E. H.-Y. Tseng and J.-L. Gaudlot. Communication generation for aligned and cyclic(k) distributions using integer lattice. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):136–146, 1999.