An Array Data Flow Analysis based Communication Optimizer*

Xin Yuan, Rajiv Gupta, and Rami Melhem

Dept. of Computer Science, Univ. of Pittsburgh, Pittsburgh, PA 15260

Abstract. We present an efficient array data flow analysis based global communication optimizer which manages the analysis cost by partitioning the data flow problems into subproblems and solving the subproblems *one at a time* in a demand driven manner. In comparison to traditional array data flow based techniques, our scheme greatly reduces the memory requirement and manages the analysis time more effectively. The optimizer performs message vectorization, global redundant communication elimination and global communication scheduling. Our experience with the optimizer suggests that array data flow analysis for communication optimization is efficient and effective.

1 Introduction

Communication optimization is crucial for obtaining good performance for programs compiled to execute on distributed memory systems. Traditionally, data dependence analysis has been used to perform communication optimizations within a single loop nest [2, 10, 14]. Recently, data flow analysis techniques have been developed to obtain information for global communication optimizations [4, 7, 9, 12, 13].

One approach, which will be referred to as the *array dependence approach*, refines data flow analysis for scalar with data dependence analysis [4, 12, 13]. Another approach, which we will refer to as the *array dataflow approach*, performs global array data flow analysis [7, 9]. The array dataflow approach can obtain more accurate data flow information at a higher analysis cost than the array dependence approach. The high analysis cost in the array dataflow approach results from the complexity of the data flow descriptor [7, 9] and that operations on the descriptors often result in lists of descriptors. In traditional data flow analysis, to obtain data flow information at one point, data flow solutions at all points must be computed. This requires large memory space and does not allow control over the analysis time. Thus, only a very simplified version of the array dataflow approach has been previously implemented [9] and it was uncertain whether the array dataflow approach is practical for large programs.

In this paper, we present an array data flow analysis based communication optimizer. The optimizer performs *message vectorization*, *redundant communication elimination* and *global message scheduling* [4]. To address the analysis cost

* Partially supported by NSF Presidential Young Investigator Award CCR-9157371

problem, the data flow analysis for the optimizations is partitioned into phases. Within each phase, the data flow problems are partitioned into subproblems and the subproblems are solved *one at a time* using a demand driven algorithm [16]. Each subproblem contains the data flow effect of a single communication.

Our scheme offers many advantages over traditional data flow analysis techniques for communication optimizations. First, since subproblems are solved one at a time, same memory space can be reused repeatedly. Second, the analysis time can be easily managed. Given a limited time, partitioning allows our optimizer to obtain partial data flow results and perform optimizations based upon these results. Third, our scheme improves analysis time by reducing the time for data flow set operations performed when a set changes during propagation. Traditional methods simultaneously operate on all communications in a program, thus, a new data flow set must be recomputed when any element in the set is changed. Partitioning data flow problems reduces data flow set size and thus decreases the likelihood of recomputing a data flow set. Fourth, our optimizer performs optimization and analysis in an interleaved fashion. This allows the optimizer to discard intermediate data flow solutions. Although partitioning of data flow problems requires multiple traversals of the program graph, it does not significantly affect the analysis cost since the set operations dominate the analysis time.

Our optimizer is implemented on top of the Stanford SUIF compiler [1]. Experiments were conducted to evaluate the performance of the optimizer. To our knowledge, this is the first implementation of a full scale array data flow analysis based communication optimizer. The rest of the paper is organized as follows. Section 2 describes the program representation. Section 3 introduces the *section communication descriptor* (SCD), which is used in the optimizer to represent communications, and its operations. Section 4 presents the optimizer. Section 5 reports our experience with the optimizer. Section 6 concludes the paper.

2 Program Representation

We consider structured programs that contain conditionals and nested loops, but no arbitrary goto statements. A loop is controlled by a basic induction variable and no statement in the loop contains an assignment to this variable. To simplify the presentation we assume that loops are normalized. However, our implementation also handles loops that are not normalized. Array references in nested loops are restricted to references whose subscripts are affine functions of loop induction variables, that is, references are of the form $X(f_1(\mathbf{i}), f_2(\mathbf{i}), ..., f_s(\mathbf{i}))$, where $\mathbf{i}$ is the vector representing the loop induction variables and $f_k(\mathbf{i})$, $k = 1..s$, are affine functions of $\mathbf{i}$.

The optimizer performs optimizations on each subroutine. A subroutine is represented as an *interval flow graph* $G = (N, E)$, with nodes N and edges E. Our analysis is based upon a variant of Tarjan's intervals [15]. The analysis requires that there are no *critical edges* which are edges that connect a node with multiple outgoing edges to a node with multiple incoming edges. The critical edges can be eliminated by edge splitting transformation[9]. Fig. 1 shows an example code and its corresponding interval flow graph.

```
      ALIGN (i, j) with VPROCS(i, j) :: x, y, z
      ALIGN (i, j) with VPROCS(2*j, i+1) :: w
(s1)  do i = 1, 100
(s2)    do j = 1, 100
(s3)      x(i,j)=...
(s4)    enddo
(s5)  enddo
(s6)  do i = 1, 100
(s7)    do j = 1, 100
(s8)      y(i,j)=w(i,j)
(s9)    enddo
(s10) enddo
(s11) do i = 1, 100
(s12)   do j = 1, 100
(s13)     z(i, j) = x(i+1, j)* w(i, ,j)
(s14)     z(i, j) = z(i, j)* y(i+1, ,j)
(s15)   end do
(s16)   w(i+1, 100) = ...
(s17) end do
```

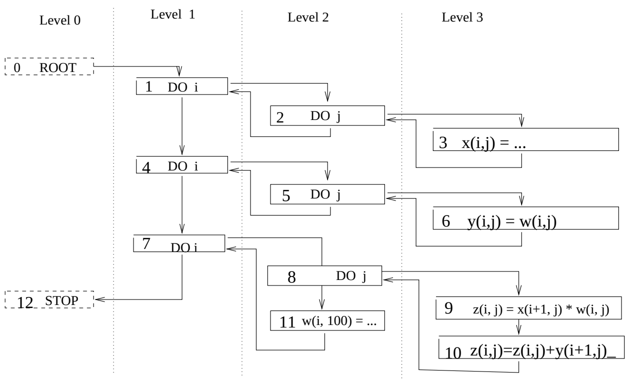

Fig. 1. An example program and its interval flow graph

3 Section Communication Descriptor (SCD)

In this section, we introduce the *Section Communication Descriptor* (SCD), which is used in the optimizer to represent communications. This descriptor augments the descriptor we designed previously [16] in two ways: (1) it has the ability of describing communications that occur specifically in certain iterations of a loop, and (2) it is more powerful in describing communication relations. The augmentation slightly increases the complexity of the descriptor and its operations and gives the optimizer more power in communication optimizations.

3.1 The SCD Descriptor

The processor space is considered as an unbounded grid of virtual processors. The abstract processor space is similar to a *template* in High Performance Fortran (HPF) [11], which is a grid over which different arrays are aligned. In the rest of the paper, when we refer to communication, we mean communication on the virtual processor grid.

A *Section Communication Descriptor* (SCD), denoted as $< N, D, M, Q >$, is composed of three parts. First, an array name, N, and a region descriptor, D, form an *array region descriptor* which describes the array elements that are involved in the communication. Second, a *communication mapping descriptor*, M, describes the source–destination relationship of the communication on the virtual processor grid. Third, a *qualifier descriptor*, Q, describes the iterations during which the communication is performed.

The *bounded regular section descriptor* (BRSD) is used as the region descriptor. The region D is a vector of subscript values. Each element in the vector is either (1) an expression of the form $\alpha * i + \beta$, where α and β are invariants and i is a loop index variable, or (2) a triple $l : u : s$, where l, u and s are invariants. The triple, $l : u : s$, defines a set of values, $\{l,\ l+s,\ l+2s,\ ...,\ u\}$, as in HPF.

The source–destination mapping M is denoted as $< src, dst, qual >$. The source, src, is a vector whose elements are of the form $\alpha * i + \beta$, where α and β are invariants and i is a loop index variable. The destination, dst, is a vector whose elements are of the form $\sum_{j=1}^{n} \alpha_j * i_j + \beta_j$, where α_j's and β_j's are invariants and i_j's are loop index variables. The *mapping qualifier* list, $qual$, is a range descriptor list. Each range descriptor is of the form $i = l : u : s$, where l, u and s are invariants and i is a loop induction variable. Notation $qual = \perp$ or $qual = nil$ denotes that no mapping qualifier is needed. In this case, the source–destination relation is determined solely from src and dst. The mapping qualifier is used to describe the broadcast effect, which may be needed during message vectorization. Notation $M = \top$ denotes all mapping relations.

The *qualifier descriptor* Q is a range descriptor of the form $i = l : u : s$. Notation $Q = \perp$ or $Q = nil$ is used to indicate that the communication is to be performed in every iteration. Q will be referred to as the *communication qualifier*. The communication qualifier, Q, is included in a SCD to describe communications that are performed during certain iterations of a loop. This is needed for partial message vectorization where communications in some iterations of a loop are vectorized and hoisted out of the loop while communications in other iterations remain inside the loop.

3.2 Operations on SCD

Operations, such as intersection, difference and union, on SCD descriptors are defined next. Since in many cases, operations do not have sufficient information to yield exact results, *subset* and *superset* versions of these operations are implemented. The optimizer uses a proper version to obtain conservative approximations. These operations are extensions of the operations on BRSD.

Subset Mapping testing. Testing whether a mapping is a subset of another mapping is one of the most commonly used operations in the optimizer. Testing that a mapping relation M_1 ($=< s_1, d_1, q_1 >$) is a subset of another mapping relation M_2 ($=< s_2, d_2, q_2 >$) is done by checking for a solution of equations $s_1 = s_2$ and $d_1 = d_2$, where variables in M_1 are treated as constants and variables in M_2 as variables, and subrange testing $q_1 \subseteq q_2$. Note that since the elements in s_1 and s_2 are of the form $\alpha * i + \beta$, the equations can generally be solved efficiently. Two mappings, M_1 and M_2 are *related* if $M_1 \subseteq M_2$ or $M_2 \subseteq M_1$. Otherwise, they are unrelated.
Subset SCD testing. Let $S_1 =< N_1, D_1, M_1, Q_1 >$, $S_2 =< N_2, D_2, M_2, Q_2 >$, $SCD_1 \subseteq SCD_2 \Longleftrightarrow N_1 = N_2 \wedge D_1 \subseteq D_2 \wedge M_1 \subseteq M_2 \wedge Q_1 \subseteq Q_2$.
Intersection Operation. The intersection of two SCDs represents the elements constituting the common part of their array sections that have the same mapping relation. The following algorithm describes the subset version of the intersection operation. Note that the operation requires the qualifier Q_1 to be equal to Q_2 to obtain a non ϕ result. This approximation will not hurt the performance significantly since most SCDs have $Q =\perp$.

$< N_1, D_1, M_1, Q_1 > \cap < N_2, D_2, M_2, Q_2 >$
$= \phi$, if $N_1 \neq N_2$ or M_1 and M_2 have no relation or $Q_1 \neq Q_2$
$= < N_1, D_1 \cap D_2, M_1, Q_1 >$, if $N_1 = N_2$ and $M_1 \subseteq M_2$ and $Q_1 = Q_2$
$= < N_1, D_1 \cap D_2, M_2, Q_1 >$, if $N_1 = N_2$ and $M_1 \supseteq M_2$ and $Q_1 = Q_2$

Difference Operation. The difference operation causes a part of the array region associated with the first operand to be invalidated at all the processors where it was available. In our analysis, the difference operation is only used to subtract elements killed (by a statement, or by a region), which means that the SCD to be subtracted always has $M = \top$ and $Q =\perp$.

$< N_1, D_1, M_1, Q_1 > - < N_2, D_2, \top, \perp>$
$= < N_1, D_1, M_1, Q_1 >$, if $N_1 \neq N_2$
$= < N_1, D_1 - D_2, M_1, Q_1 >$, if $N_1 = N_2$.

Union operation. The union of two SCDs represents the elements that can be in either part of their array section. This operation is given by:

$< N_1, D_1, M_1, Q_1 > \cup < N_2, D_2, M_2, Q_2 >$
$= < N_1, D_1 \cup D2, M1, Q_1 >$, if $N_1 = N_2$ and $M_1 = M_2$ and $Q_1 = Q_2$
$=$ list($< N_1, D_1, M_1, Q_1 >, < N_2, D_2, M_2, Q_2 >$), otherwise.

4 The Optimizer

The optimizer performs message vectorization, redundant communication elimination and communication scheduling using algorithms based upon the demand driven analysis in [16]. The optimization steps include:

1. *Initial SCD calculation.* Here the optimizer calculates the communication requirement for each statement that contains remote memory references. Communications required by each statement are called *initial SCDs* for the statement and are placed preceding the statement.
2. *Message vectorization and available communication summary calculation.* The optimizer propagates initial SCDs to the outermost loops in which they can be placed. In addition to message vectorization optimization, this step also calculates the summary of communications that are available after each loop. This information is used in the next step for redundant communication elimination.
3. *Redundant communication elimination.* The optimizer performs redundant communication elimination using demand driven version of availability communication analysis [8], which computes communications that are available before each statement. A communication in a statement is redundant if it can be subsumed by available communications at the statement. Our optimizer also eliminates partially redundant communication.
4. *Message scheduling.* The optimizer schedules messages within each interval by placing messages with same communication patterns together and combining the messages to reduce the number of messages.

4.1 Initial SCD Calculation

We assume *owner computes* rule which requires each remote item referenced on the *rhs* of an assignment statement to be sent to the processor that owns the *lhs*. Initial SCDs for each statement represent this data movement. Since the ownership of array elements determines communication patterns, we describe the ownership of array elements before presenting the initial SCD calculation.

Ownership. All arrays are aligned to a single virtual processor grid by simple affine functions. The alignments allowed are scaling, axis alignment and offset alignment. The mapping from a point $\mathbf{d}$ in data space to a corresponding point $\mathbf{v}$ on the virtual processor grid (the owner of $\mathbf{d}$) can be specified by an alignment matrix M and an alignment offset vector $\boldsymbol{\alpha}$ such that $\mathbf{v} = M\mathbf{d} + \boldsymbol{\alpha}$. Using the alignment matrix and the offset vector, the owner of a data element can be determined. Consider the array w in the example program in Fig. 1, the alignment matrix and the offset vector are given below.

$$M_w = \begin{pmatrix} 0 & 2 \\ 1 & 0 \end{pmatrix}, \ \alpha_w = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Initial SCD Calculation. Using the ownership information, the initial SCDs are calculated as follows. Let us consider each component in an initial $SCD =< N, D, M, Q >$. N is the array to be communicated. The region D contains a single index given by the array subscript expression. The qualifier $Q = \perp$ since initial communications must be performed in every iteration. Let $M =<$

$src, dst, qual >$. Since initially communication does not perform broadcast, $qual = \perp$. Hence, the calculation of *src* and *dst*, which we will discuss in the following text, is the only non-trivial computation in the calculation of initial SCDs.

Let $\mathbf{i}$ be the vector of loop induction variables. When subscript expressions are affine functions, an array reference can be expressed as $N(G\mathbf{i}+\mathbf{g})$, where N is the array name, G is a matrix and $\mathbf{g}$ is a vector. We call G the *data access matrix* and $\mathbf{g}$ the *access offset vector*. The data access matrix, G, and the access offset vector, $\mathbf{g}$, describe a mapping from a point in the iteration space to a point in the data space. Let G_l, $\mathbf{g}_l$, M_l, $\boldsymbol{\alpha}_l$ be the data access matrix, the access offset vector, the alignment matrix and the alignment vector for the *lhs* array reference, and G_r, $\mathbf{g}_r$, M_r, $\boldsymbol{\alpha}_r$ be the corresponding quantities for the *rhs* array reference. The source processor *src* and destination processor *dst* are given by:

$$src = M_r(G_r\mathbf{i} + \mathbf{g}_r) + \boldsymbol{\alpha}_r, \qquad dst = M_l(G_l\mathbf{i} + \mathbf{g}_l) + \boldsymbol{\alpha}_l$$

Consider the communication of $w(i,j)$ in statement $s13$ in Fig. 1. The optimizer can obtain from the program the data access matrices, access offset vectors, alignment matrices and alignment vectors and from them the SCD for the communication given below. As an indication of the complexity of a SCD, the structure for this communication required 524 bytes to store.

$$M_z = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \ \alpha_z = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \ M_w = \begin{pmatrix} 0 & 2 \\ 1 & 0 \end{pmatrix}, \ \alpha_w = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$G_l = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \ \mathbf{g}_l = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \ G_r = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \ \mathbf{g}_r = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$< N = w, D = (i,j), M =< (2*j, i+1), (i,j), \perp>, Q = \perp>$$

4.2 General SCD Propagation Rules

In [16], we designed a set of general rules to propagate SCDs in an interval flow graph. These rules form a demand driven version of the interval analysis in [8]. These rules propagate each communication to its earliest (latest) possible points in backward (forward) propagation, that is, the earliest (latest) points in the program that the communication can be placed(alive). Each optimization performed by the optimizer computes the required data flow information using a subset of these rules.

The propagation starts by propagating initial SCDs which carry the communication requirement information for statements. When a SCD reaches an interval boundary, the SCD is *expanded* to represent all communications inside the loop. If there are no data dependencies that prevent the communications from being propagated out of the interval, all communications represented by the expanded SCD are propagated into the outer interval. This type of propagation is called *UP* propagation. The UP propagation propagates SCDs from an inner interval to an outer interval, expanding SCDs when crossing interval boundaries. If there are data dependencies inside the interval that prevent the communications from being propagated out of the interval, part of the expanded SCD must be propagated back into the interval to determine the exact places where the communications are killed. This type of propagation is called *DOWN* propagation. The DOWN propagation propagates SCDs, which carry summary information, from outer intervals into inner intervals. When a DOWN propagation reaches the loop header of the interval in which it originates, its propagation will terminate as opposed to UP propagations where SCDs are expanded when a loop header is reached. The UP and DOWN propagations correspond to the two phases of the interval analysis in [8].

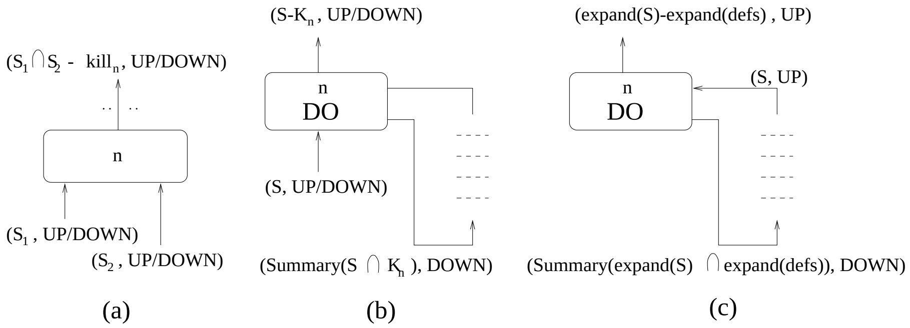

Fig. 2. General propagation rules

Fig. 2 depicts the rules for propagating SCDs backward. Rules to propagate SCDs forward are similar. In the figure, (S, $UP|DOWN$) denotes the UP or DOWN propagation of communication S. Fig. 2(a) shows the case when SCDs are propagated through a regular node. If a node has more than one successor, SCDs from all successors are intersected before subtracting the elements killed in the node. The result is then propagated to all of its predecessors. Fig. 2(b) shows the case when a SCD propagates past a loop header node in the same interval, in addition to the SCD that passes the loop header, a DOWN propagation will be generated to propagate a SCD that contains all elements that are killed into the loop body to determine the exact places where they are killed. Here, K_n is the set of array elements killed inside the loop and the $summary(S)$ function computes communications in S that are not killed at iteration i, where i is the induction variable. When an UP propagation reaches its loop header, it will be expanded into a summary of all communications inside the loop. The summary can be propagated into the outer interval if none of the elements in the summary are killed inside the loop. If part of the summary is killed inside the loop, only the part that is not killed will be propagated into the outer interval, while the part that is killed inside must be propagated back into the loop body to determine the exact places that kill the communications. Fig. 2(c) shows this case. The $expand(S)$ function in the figure computes the SCD that represents all communications (represented by S) inside the interval. When a DOWN propagation reaches its loop header, the propagation terminates.

Our optimizer partitions the communication optimization, and thus the SCD propagation process, into three phases. First, the optimizer determines the interval to place each communication by propagating the communication into its outermost interval. This is the message vectorization and available communication summary calculation step. Second, communications are propagated forward within each interval to perform availability communication analysis and redundant communication elimination. This is the redundant communication elimination step. Notice that propagating SCDs within their intervals is much more efficient than propagating SCDs through the whole program as in [8]. Third, the optimizer performs communication scheduling. In this phase, SCDs are propagated backward to find their earliest points that can be used to place the communications. In the subsequent sections we will discuss these phases in detail.

4.3 Message Vectorization and Available Communication Summary

In this phase, the optimizer calculates *backward exposed* communications, which are SCDs that can be hoisted out of a loop, and *forward exposed* communications, which are SCDs that are available after the loop. Backward exposed communications represent actual communications vectorized from inside the loop. When a SCD is vectorized, the initial SCD at the assignment statement are replaced by SCDs for backward exposed communications at loop headers. Forward exposed communications represent the communications that are performed inside a loop and are still alive after the loop. Hence they can be used to subsume communications appearing after the loop. By using data dependence information, backward and forward exposed communications are calculated by propagating SCDs from inner loop bodies to loop headers using a simplified version of the rule in Fig. 2 (c).

Algorithms for forward and backward exposed communication calculation are described in Fig 3 (a) and (b). Since only UP propagation is needed, we use $Request(S, n)$ to denote placing a propagation of S after node n. In the algorithms, S is a SCD occurring inside the interval whose header is node n and whose induction variable is i with lower bound 1 and upper bound h, $anti\_def$ is the set of definitions in the interval that have anti–dependence relation with the original array reference that causes the communication S, $flow\_def$ is the set of definitions in the interval that have flow–dependence relation with the original array reference that causes the communication S. For a SCD, S, $expand(S, i, 1 : h)$ first determines which portion of the $S =< N, D, M, Q >$ to be expanded. If D is to be expanded, that is, i occurs in D, the function will replace all single data item references $\alpha * i + \beta$ used in D by the triple $\alpha + \beta : \alpha * h + \beta : \alpha$. If D cannot be expanded, that is, after expansion D is not in the allowed form, then the communications will stay inside the loop. If $M =< src, dst, qual >$ is to be expanded, that is, i occurs in dst but not in src and D, the function will add $i = 1 : h : 1$ into the mapping qualifier list $qual$.

The algorithms essentially determine the part of communications, $Outside$, that can be hoisted out of a loop and the part, $Inside$ that cannot be hoisted out of the loop. In forward exposed communication calculation, the optimizer makes *Outside* as the forward exposed communication and ignores the *Inside* part. In backward exposed communication calculation, the optimizer makes *Outside* as backward exposed communication. In addition, the optimizer must also change the original SCD according to contents of *Inside*. In the case when the SCD can be fully vectorized, the SCD in the original statement is removed. In the case when the SCD cannot be fully vectorized, part of the communication represented by *Outside* is hoisted out of the loop, while other part represented by *Inside* stays at the original statement. Thus, the SCD in original statement must be modified by a communication qualifier to indicate that the SCD only remains in iterations that generate communications in *Inside*.

request(S, n) :
  $Outside = expand(S, i, 1:h) - \cup_{anti\_def} expand(anti\_def, i, 1:h)$
  if ($Outside \neq \phi$) then
    record $Outside$ as
      forward exposed in node n
    Let m be the header of the
      interval including node n
    $request(Outside, m)$;

(a) Forward exposed communication

request(S, n) :
  $Outside = expand(S, i, 1:h) - \cup_{flow\_def} expand(flow\_def, i, 1:h)$
  $Inside = expand(S, i, 1:h) \cap \cup_{flow\_def} expand(flow\_def, i, 1:h)$
  if ($Outside \neq \phi$) then
    convert $Inside$ in terms of S
      with qualifier, denoted as D
    if (conversion not successful) then
      stop /* fail */
    else
      change the S into D
      record $Outside$ as backward
        exposed comm. at node n.
      Let m be the header of the
        interval including node n
      $request(Outside, m)$;

(b) Backward exposed communication

Fig. 3. Algorithms for the forward and backward exposed communication

Consider communications in the loop in Fig. 4. Assume that arrays a, b and d are identically aligned to the virtual processor grid, initial SCDs, $C1$ and $C2$, are shown in Fig. 4. $C3$, $C4$ and $C5$ are the communications after backward exposed communication calculation. Calculating backward exposed communication for $C1$ results in communication $C3$ in the loop header and the removal of the communication $C1$ from its original statement. Calculating backward exposed communication for $C2$ puts $C4$ in the loop header and changes $C2$ into $C5$. Note that, there is a flow–dependence relation from b(i) to b(i-1). In calculating the backward exposed communication for SCD $C2$, $Inside = <b, (1:99:1), <(i-1, 1), (i, 1), \perp>, \perp>$. Converting $Inside$ back in terms of $C2$ results in $C5$.

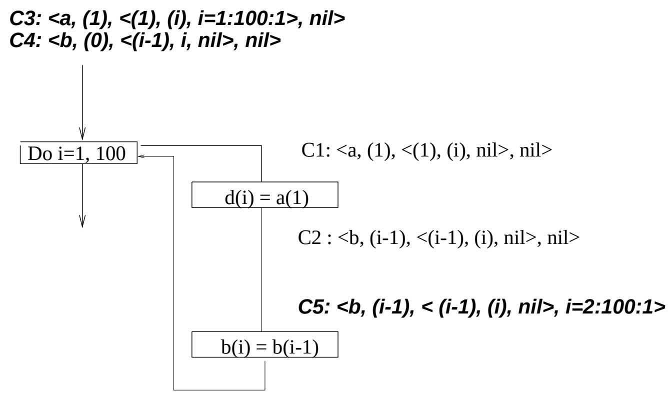

Fig. 4. Calculating backward exposed communications

4.4 Redundant Communication Elimination

This phase calculates available communications before each statement, and eliminates a communication at the statement if the communication is available. This optimization is done by propagating SCDs forward until all elements are killed. During the propagation, if another SCD that can be subsumed is encountered, that SCD is redundant and can be eliminated.

request(S_1, n, UP) $\wedge$ ...
$\wedge$request(S_k, n, UP) :
  S = $S_1 \cap ... \cap S_k$
  if (SCDs in n is a subset of S) then
    remove the SCDs
  if ($S - kill_n \neq \phi$) then
    for all $m \in succ(n)$
      request($S - kill_n, m, UP$)

(a) Actions on nodes within an interval

request(S, n, UP):
  calculate the summary of loop n
  Inside= $expand(S, i, 1 : i-1) \cap$
    $(\cup_{def} expand(def, i, 1 : i-1))$
  if (inside $\neq \phi$) then
    Let l be the first node.
    request($Inside, l, DOWN$)

(b) Actions on a loop header

Fig. 5. Actions in forward propagation

Using the interval analysis technique [8], two passes are needed to obtain the data flow solutions in an interval. Initially, UP propagations are performed. Once the UP propagations reach interval headers, summaries of the SCDs are calculated and DOWN propagations of the summaries are triggered. Note that since the data flow effect of propagating SCDs between intervals is captured in the message vectorization phase of our optimizer, both the UP and DOWN propagations are performed within an interval in this phase.

Assuming that node n has k predecessors. When propagating SCDs within an interval in forward propagation, actions in a node will be triggered only when all its predecessors place requests. The nodes calculate the SCD available by performing intersection on all SCDs that reach it, check whether communications within the node can be subsumed, and propagate the live communications forward. Fig. 5 (a) describes actions on the nodes inside the interval in an UP forward propagation. When the UP propagation reaches an interval boundary, the summary information is calculated by obtaining all the elements that are available in iteration i, and a DOWN propagation is triggered. Note that in forward propagation, communications can be safely assumed to be performed in every iteration ($Q = \perp$), since the effect of the communication must guarantee that the valid values are at the proper processors for the computation. Fig. 5 (b) shows actions at interval boundaries. The propagation of a DOWN request is similar to that of an UP request except that a DOWN propagation stops at interval boundaries.

4.5 Global Message Scheduling

After redundant communication elimination, our optimizer further reduces the number of messages using a global message scheduling algorithm proposed by Chakrabarti et al. in [4]. The idea of this optimization is to combine messages that are of the same communication pattern into a single message to reduce number of messages in a program. In order to perform message scheduling, the optimizer first determines the earliest and latest points for each communication. Placing the communication in any point between the earliest and the latest points that dominates the latest point always yields correct programs. Thus, the optimizer can schedule the placement of messages such that messages of same communication patterns are placed together and are combined to reduce the number of messages.

The latest point for a communication is the place of the SCD after redundant communication elimination. Note that after message vectorization, SCDs are placed in the outermost loops that can perform the communications. The earliest point for a SCD can be found by propagating the SCD backward. As in [4], we assume that communication for a SCD is performed at a single point. Hence, the backward propagation will stop after an assignment statement, a loop header or a branch statement where part of the SCD is killed. Since the propagation of SCDs stops at a loop header node, only the UP propagation is needed. Once the earliest and latest points for each communication are known, the greedy algorithm in [4] is used to do the communication scheduling.

5 Experimental Results

Our optimizer is implemented on top of the Stanford SUIF compiler. To evaluate its performance, we developed a communication emulation system, which takes SCDs as input, emulates the communications described by the SCDs and collects the statistics, such as total number of elements communicated and total number of messages, about the required communications. The emulation system provides an interface with C program as a library call whose arguments include all information in a SCD. Our compiler backend automatically generates the library call for each SCD remaining in the program. In this way, the communication performance of a program can be evaluated in the emulation system by running programs generated by our compiler backend.

We used six programs in our experiment. The first benchmark, L18, is the explicit hydrodynamics kernel in livermore loops (loop 18). The second benchmark, ARTDIF, is a kernel routine obtained from HYDRO2D program, which is an astrophysical program for the computation of galactical jets using hydrodynamical Navier Stokes equations. The third benchmark, TOMCATV, does the mesh generation with Thompson's solver. The fourth program, SWIM, is the SHALLOW weather prediction program. The Fifth program, MGRID, is the simple multigrid solver for computing a three dimensional potential field. This sixth program, ERHS, is part of the APPLU program, which is the solver for five coupled parabolic/elliptic partial differential equations. The programs, HYDRO2D, TOMCATV, SWIM, MGRID and APPLU, originally come from SPEC95 benchmark suite.

Table 1 shows the analysis cost of our optimizer. We ran the optimizer, which applies the algorithms on all SCDs in the programs, on SPARC 5 with 32MB memory. Row 2 and Row 3 shows the program sizes. Row 4 shows the cumulative memory requirement, which is the sum of number of SCDs passing through each node. This number is approximately equal to the memory requirement of traditional data flow analysis. The value in bracket is the maximum number of cumulative SCDs in a node, which is the extra memory needed by our optimizer. In our optimizer the size of a SCD ranges from 0.6 to about 2 kbytes. Our results show that traditional analysis method will require large amount of memory when a program is large, while our optimizer uses little extra memory. Row 5 gives the raw analysis times and row 6 shows the rate at which our optimizer operates in units of $lines/sec$. On an average our optimizer compiles 172 lines per second for the six programs. Row 9 shows the total time, which includes analysis time and the time to load and store the SUIF structure, for reference. In most cases, the analysis time is only a fraction of the load and store time.

Program	L18	ARTDIF	TOMCATV	SWIM	MGRID	ERHS
size(lines)	83	101	190	429	486	1104
# of initial SCDs	35	12	108	76	125	403
accu. memory req.	348(1)	175(1)	5078(3)	767(1)	1166(1)	6029(5)
analysis time(sec)	0.62	0.32	3.47	1.87	1.92	20.92
lines / sec	133	316	54	229	253	52
total time(sec)	2.00	1.75	6.95	6.65	12.52	35.42

Table 1. Analysis time

Table. 2 and Table. 3 show the effectiveness of our optimizer. Table. 2 shows the reduction of total number of elements to be communicated and Table. 3 shows the reduction of total number of messages. We evaluate the performace with both cyclic and block distributions on 16 PE systems. This experiment is conducted using the test input size provided by the SPEC95 benchmark for program TOMCATV, SWIM, MGRID ERHS. The outermost iteration number in MGRID is reduced to 1 (from 40). Problem sizes of 6×100 for L18 and 402×160 for ARTDIF are used. We compare the number of elements and number of messages communicated after all optimizations to those after message vectorization optimization. From Table 2, we can see that for cyclic distribution, an average of 31.5% of the total communication elements are reduced. The block distribution greatly reduces the number of elements to be communicated and affects optimization performance of the optimizer. For block distribution, the average reduction is 23.1%. From Table 3, we can see that our optimizer reduces total message number by 36.7% for cyclic distribution and by 35.1% for block distribution. These results indicate that global communication optimization results in large performance gain and our optimizer is effective in finding optimization opportunities.

Dist.	Opt.	L18	ARTDIF	TOMCATV	SWIM	MGRID	ERHS
		$\times 10^4$	$\times 10^5$	$\times 10^8$	$\times 10^7$	$\times 10^7$	$\times 10^6$
cyclic	Vector.	1.38	7.01	1.38	6.38	5.69	3.62
	Final	0.96	5.73	0.34	4.58	5.69	2.29
		69.6%	81.7%	24.6%	71.8%	100%	63.3%
		$\times 10^3$	$\times 10^4$	$\times 10^6$	$\times 10^6$	$\times 10^6$	$\times 10^6$
block	Vector.	3.26	7.17	5.74	3.38	8.49	3.11
	Final	2.57	6.97	5.12	1.08	8.49	1.65
		78.8%	97.2%	89.1%	32.0%	100%	53.1%

Table 2. Total number of elements to be communicated

Dist.	Opt.	L18	ARTDIF	TOMCATV	SWIM	MGRID	ERHS
cyclic	Vector.	368	400	68555	3892	17662	1.14×10^6
	Final	96	336	41075	1807	17662	0.72×10^6
		26.1%	84.0%	59.9%	46.4%	100%	63.1%
block	Vector.	330	185	16750	3894	14650	9.20×10^5
	Final	90	161	10915	2209	14650	4.89×10^5
		27.3%	87%	65.2%	56.7%	100%	53.2%

Table 3. Total number of messages

6 Conclusion

We presented an array data flow based communication optimizer which obtains the information by propagating the communications in the program one at a time. This approach allows control over the analysis time and reduces memory requirements. Our experiments show that the optimizer is efficient both in analysis costs and effective in its optimization results.

References

1. S. P. Amarasinghe, J. M. Anderson, M. S. Lam and C. W. Tseng, "The SUIF Compiler for Scalable Parallel Machines," *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, Feb. 1995.
2. S. P. Amarasinghe and M. S. Lam "Communication Optimization and Code Generation for Distributed Memory Machine." In *Proceedings ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, June 1993.
3. P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. "The PARADIGM Compiler for Distributed-Memory Multicomputers." in *IEEE Computer*, Vol. 28, No. 10, pages 37-47, October 1995.
4. S. Chakrabarti, M. Gupta and J. Choi "Global Communication Analysis and Optimization." In *Programming Language Design and Implementation*(PLDI), 1996, pages 68-78.
5. J.F. Collard, d. Barthou and P. Feautrier "Fuzzy Array Dataflow analysis." In *5th ACM SIGPLAN Symposium on Principle & Practice of Parallel Programming*, July 1995, Santa Barbara, CA.
6. E. Duesterwald, R. Gupta and M. L. Soffa "Demand-driven Computation of Interprocedural Data Flow" In *Symposium on Principles of Programming Languages*, Jan. 1995, San Francisco, CA.
7. C. Gong, R. Gupta and R. Melhem "Compilation Techniques for Optimizing Communication on Distributed-Memory Systems" In *International Conference on Parallel Processing*, Vol II, pages 39-46, August 1993.
8. M. Gupta and E. Schonberg "A Framework for Exploiting Data Availability to Optimize Communication." In *6th International Workshop on Languages and Compilers for Parallel Computing*, LNCS 768, pp 216-233, August 1993.
9. M. Gupta, E. Schonberg and H. Srinivasan "A Unified Framework for Optimizing Communication in Data-parallel Programs." In *IEEE Trans. on Parallel and Distributed Systems*, Vol. 7, No. 7, pages 689-704, July 1996.
10. S. Hiranandani, K. Kennedy and C. Tseng "Compiling Fortran D for MIMD Distributed–memory Machines." *Comm. of the ACM*, 35(8):66-80, August 1992.
11. High Performance Fortran Forum. "High Performance Fortran Language specification." version 1.0 Technique Report CRPC-TR92225, Rice University, 1993.
12. K. Kennedy and N. Nedeljkovic "Combining dependence and data-flow analyses to optimize communication." In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.
13. K. Kennedy and A. Sethi, "A Constraint Based Communication Placement Framework." Technique Report CRPC–TR95515-S, Center for Research on Parallel Computation, Rice University. Feb. 1995.
14. J. Li and M. Chen, "Compiling communication efficient programs for massively parallel machines." *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.
15. R.E. Tarjan "Testing flow graph reducibility." *Journal of Computer and System Sciences*, 9:355-365, 1974.
16. X. Yuan, R. Gupta and R. Melhem "Demand–driven Data Flow Analysis for Communication Optimization." *Workshop on Challenging in Compiling for Scalable Parallel Systems*, New Orleans, Louisiana, Oct. 1996.

This article was processed using the LaTeX macro package with LLNCS style