A Constraint Based Communication Placement Framework

Ken Kennedy
Ajay Sethi

CRPC-TR95515-S
February 1995

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Revised: May, 1995.

A Constraint-based Communication Placement Framework*

Ken Kennedy
ken@rice.edu

Ajay Sethi
sethi@rice.edu

Center for Research on Parallel Computation
Department of Computer Science, Rice University
6100 S.Main MS 41, Houston, TX 77005

Abstract

Communication placement and optimization is an important step in the compilation of data-parallel languages. We present a framework that maximizes latency hiding by determining, in two separate phases, the earliest safe placement for sends and the latest balanced placement for the corresponding receives. The compositional structure of our technique allows machine-dependent resource constraints, which can effect the correctness of the placement and restrict the achievable communication and computation overlap, to influence the communication placement. We use constrained buffer size to illustrate constraint analysis and constraint-based communication placement. Finally, we indicate how communication optimizations, like message vectorization, partially redundant communication elimination, message coalescing, and vector message pipelining, can be incorporated into the framework.

1 Introduction

High Performance Fortran [12], along with its predecessor data-parallel languages like Fortran D [3] and Vienna Fortran [1], allow programmers to write sequential or shared-memory parallel programs annotated with data-decomposition directives for distributed-memory machines. Compilers for these languages are responsible for partitioning the computation and generating the communication necessary to fetch values of non-local data referenced by a processor. Since interprocessor communication is typically orders of magnitude more expensive than accessing local data, it is extremely important to optimize communication. Moreover, in order to hide the communication latency of non-local accesses, the compiler must overlap communication and computation.

Latency hiding can be achieved by initiating the fetch of a required non-local data (via the send primitive) as early as possible and delaying the wait for the communicated data (the receive primitive) in order to expose the opportunities for overlapping communication with some intervening computation. There have been several efforts [5, 4, 6, 7] to use data-flow analysis to determine communication placement for maximizing latency hiding. In addition, these data-flow frameworks reduce the number of messages by combining

*This work was supported in part by ARPA contract DABT63-92-C-0038 and NSF Cooperative Agreement Number CCR-9120008. The content of this paper does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

messages between the same pair of processors and reduce the total amount of data communicated by eliminating redundant communication. However, to reduce the complexity of the problem being solved, these frameworks ignore various machine-dependent constraints.

Communication placement that doesn't take machine-dependent constraints into account might prove too eager in moving communication earlier in the program. Placing communication at the first safe place can increase contention, total size of send and receive message buffers, and other demands on the machine resources. Moreover, the underlying hardware can have some other resource constraints: a fixed maximum number of outstanding prefetches on distributed shared-memory machines being an example of such a constraint. The constraints can also affect the correctness of the placement: for example, on distributed-memory machines, if a system buffer is not available to receive messages on processors, the system can deadlock. Thus, the machine characteristics must be taken into account to ensure correctness as well as to extract maximum benefits from the communication placement.

In this paper we present a communication placement framework that can take resource constraints into account. Section 2 briefly describes the related work, and Section 3 presents the goals of our communication placement technique and the graph structure used by our framework. In Section 4 and 5 we present the equations for decomposing the communication placement problem in the absence of constraints into the safe earliest send placement and the latest balanced receive placement. Section 6 presents the data-flow equations and analyses to take constraints into account. In Section 7 we describe how communication optimizations can be incorporated into the framework and conclude in Section 8 by summarizing our contributions.

2 Related Work

Partial Redundancy Elimination (PRE), a bi-directional algorithm for the suppression of partial redundancies, was proposed by Morel and Renvoise to improve the efficiency of a program by avoiding unnecessary recomputations of values at run time [13]. Another algorithm for eliminating partial redundancies, the Lazy Code Motion technique (LCM), decomposes the bi-directional structure of PRE into a sequence of uni-directional analyses to avoid unnecessary code motion. LCM places the computation of subexpressions "as early as necessary but as late as possible" [11], and this laziness allows it to minimize register pressure. PRE and LCM avoid unnecessary recomputations by code motion and redundant expression elimination. Since determining the availability of non-local data due to communication is similar to the classical available expression analysis [6], techniques for moving and placing code can be adapted to determine communication placement.

Several researchers have utilized this correspondence to address the problem of communication placement and optimization. Granston and Veidenbaum combine PRE and dependence analysis to eliminate redundant monolithic global-memory accesses across loop nests in the presence of conditionals [5]. They eliminate reads of not-owned variables, in parallelized and vectorized codes, if these variables have already been read or

written locally.

Gong, Gupta, and Melhem present a data-flow framework to separate sends and receives by placing sends at "the earliest point at which the communication can be performed" [4]. Moreover, they show how to incorporate various communication optimizations into the data-flow framework. However, their technique does not eliminate partially redundant communication and handles only singly-nested loops and one-dimensional arrays. While the sends are placed at the earliest possible location, the receive instructions are "inserted at the points where the data are actually needed". However, in order to ensure correctness, hoisting a send out of a loop requires a corresponding movement of the matching (blocking) receive (that is, the send and receive primitives need to be balanced).

Gupta, Schonberg, and Srinivasan use PRE and available section descriptors to develop a framework for optimizing communication [6]. Their framework obtains the earliest placement of sends and performs communication vectorization, moves communication earlier to hide latency, and eliminates redundant communication. Non-blocking receives are initiated immediately after the sends and a wait primitive is used before each reference to block the statement from executing till the data is received. Although the framework ensures balanced placement of sends and receives, the waits are not balanced, and that can introduce unnecessary overhead along some program execution paths.

The Give-N-Take code placement framework, developed by von Hanxleden and Kennedy, can be used to generate balanced sends and receives using a producer-consumer concept [7]. The Give-N-Take framework provides a *production region*, instead of a single location, for communication placement, which can be used for general latency hiding. Recently, Kennedy and Nedeljković have proposed some techniques to incorporate dependence information into the framework in order to improve the precision of the framework for regular computations and to perform more extensive communication optimizations [10]. However, the complexity of the Give-N-Take framework makes the incorporation of constraints difficult.

None of the above-mentioned frameworks take constraints into account. The framework presented in this paper incorporates constraints by allowing them to influence the placement of sends (and, therefore, the placement of receives). Our framework is based on the LCM computation placement technique, since determining send placement resembles the earliest safe computation placement, while the receive placement has some similarities with the latest computation placement. However, before describing our approach, we first present some definitions in the next section.

3 Preliminaries

In this section we describe the objectives of our framework, the graph structure, and the predicates used by the framework.

3.1 Correctness Criteria

Our communication placement framework imposes the same correctness requirements as the Give-N-Take framework [7]:

1. *Balance:* A SEND corresponds to exactly one RECV[1].
2. *Safety:* Everything communicated is used; that is, there is no unnecessary communication.
3. *Sufficiency:* Each non-local reference is preceded by an appropriate communication.

The correctness criteria, together with the optimization criteria imposed by the Give-N-Take framework [7], require the framework to perform message vectorization, communication volume reduction, partially redundant communication elimination, and the earliest placement of SENDs for hiding latency. (Gupta, Schonberg, and Srinivasan [6] also perform these optimizations.) Besides these, our framework requires the latest placement of RECVs and the balanced placement of SENDs and RECVs.

3.2 Interval Flow Graph

Let $G = (N, E)$ be the interval flow graph of a structured program, with nodes N and edges E. Let s and e be the unique start and end nodes of G. Each edge in E can be classified as an entry, back or forward edge. The interval flow graph, used both for the placement of communication primitives and propagation of constraints, is similar to the graph structure used by the Give-N-Take framework. For the sake of completeness, we summarize the important properties of G:

- G is based on Tarjan intervals [15], where a Tarjan interval $T(h)$ is a set of control-flow nodes that correspond to a loop in the program text, entered through a unique header node h, where $h \notin T(h)$. Note that a node nested in multiple loops is a member of the Tarjan interval of the header of each enclosing loop.
- G is reducible; that is, each loop has a unique header node. The classical node splitting transformation [2] can be used to obtain a reducible graph.
- There are no *critical edges*, which connect a node with multiple successors to nodes with multiple predecessors. Critical edges can be eliminated by splitting edges as follows [11]: every edge leading to a node with more than one predecessor is split by inserting a *synthetic node*.
- For every non-empty interval $T(h)$, there exists an unique $g \in T(h)$ such that $(g, h) \in E$; that is, there is only one back edge out of $T(h)$. This can be achieved by adding a *post body node* to $T(h)$ [7].

[1] SEND and RECV are used to represent the communication primitives.

3.3 Definitions

Let COMM be the set of data communicated among the processors. For every node n and every data set $d \in$ COMM, we define local predicates: $Used(n,d) =_{df} \top$ if a subset of d is referenced at n and $Transp(n,d) =_{df} \top$ if no portion of d is modified at n. For the header node h, $Used(h,d) = \top$ if a subset of d is referenced in the interval $T(h)$ and $Transp(h,d) = \top$ if no node in $T(h)$ defines a part of d. (Note that the LCM approach uses similar variables, but their initialization and usage differ in the two frameworks.) We use the semilattice $L = \{\top, \text{TRUE}, \bot\}$ with the operators $\cup$ and $\cap$, and the difference operator defined in the standard way. Let the negation operator be: $\neg\top = \bot$, $\neg\text{TRUE} = \bot$, and $\neg\bot = \top$. We assume that the communication set COMM is determined in the initialization phase using dependence analysis and array kill information [10]; however, it can also be computed during the propagation phase with the help of an appropriate array section representation [8, 6].

SUCCS(n) and PREDS(n) correspond to the set of successor and predecessor nodes of n. SUCCS$^F(n)$ and SUCCS$^E(n)$ then correspond to the forward and entry edge successors of n. Additionally, the edges induce the following traversal orders over G [7]: given a forward edge (m,n), a FORWARD order visits m before n, and a BACKWARD order visits m after n. Similarly, for an entry edge (m,n) or a back edge (m',n'), an UPWARD order visits m after n and m' before n', whereas a DOWNWARD order visits m before n and m' after n'. Let *last node* be the unique last node of a loop nest (the node that is the source of the back edge). Also,

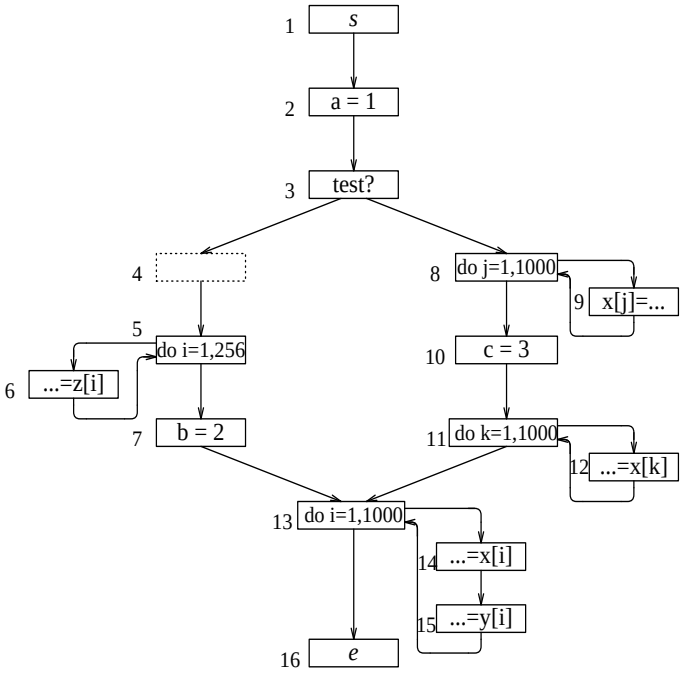

Figure 1 An example interval flow graph.

for node n, HEADER(n) gives the header node h of the enclosing interval $T(h)$. We assume the presence of a dummy header node *Root* for all nodes that occur at the outer-most level.

In order to illustrate the communication placement in presence of constraints, we use the interval flow graph shown in Figure 1. The graph shows three assignments to local data (nodes 2, 7 and 10), the references that need communication, and the definition that restricts the placement of SEND. The figure does not include the *Root* node and the synthetic nodes that do not play any role in the placement. (Actually, synthetic nodes are inserted between the nodes 3 and 5 (that is, node 4), 3 and 8, 7 and 13, 10 and 11, 11 and 13 and along every back edge.)

4 Constraint-independent Send Placement

Provided balance is ensured, the safety criteria discussed in Section 3.1 states that the SEND (and, therefore, the RECV) of a data set should be placed at node n only if there exists an use of the communicated data on all terminating paths starting at n. The node n, in this case, is called *safe* with respect to the communicated data. Sufficiency can be achieved by communicating the updated non-local data required by a processor.

We now present the equations that provide the safe and sufficient placement for SEND(d), for $d \in$ COMM. The solution of the following equation, which requires a BACKWARD and UPWARD traversal of G, gives the set of safe (and sufficient) nodes for SEND placement [11]:

$$\text{SAFE}(n,d) = \text{SAFE}(n,d) \cap [\mathit{Used}(n,d) \cup \mathit{Transp}(n,d) \cap \bigcap_{s \in \text{SUCCS}^F(n)} \text{SAFE}(s,d)]$$

where *Used*, *Transp*, and SUCCS^F are as defined in Section 3.3. SAFE(e,d) is initialized to $\perp$ for all data sets $d \in$ COMM, and if $\text{SUCCS}^F(n) = \emptyset$, $\cap_{s \in \text{SUCCS}^F(n)}\text{SAFE}(s,d) = \perp$, since no data is used at the *last node* (recall that each interval has an unique post body node, *last node*). By not considering the outstanding SENDs at SUCCS^F(HEADER(*last node*)), the equation ensures that vectorized SENDs are not hoisted back into a loop. The equation denotes that a node that has not been marked as *unsafe* is *safe* if it either (a) contains an use of the communicated data, or (b) does not modify the data being sent and all its successors are *safe* (that is, the communicated data is used along all the paths starting at node n).

Communication arising due to a true dependence carried by the loop corresponding to a header node h cannot be vectorized out of the loop. For each such communication set d, SAFE(h,d) is initialized to $\perp$; this prevents hoisting SEND(d) to a node before the loop header and, therefore, inhibits further vectorization. In the absence of constraints, for all other messages d and nodes n, SAFE(n,d) is initialized to $\top$.

Traditionally, hoisting computation out of a loop with unknown bounds is considered *unsafe* because if the loop body is not executed, it introduces a new value on an execution path of the program. On the other hand, hoisting communication out of a loop can only cause over-communication. Therefore, the importance of message vectorization in distributed-memory compilation [9] favors an optimistic handling of loops [7]. Under the relaxed safety constraint, we want to hoist communication out of a loop if it can be hoisted across

the first child of the loop header node. Thus, for a header node h, safety is determined as follows:

$$\text{SAFE}(h,d) = \text{SAFE}(h,d) \cap [\mathit{Used}(h,d) \cup \mathit{Transp}(h,d) \cap \bigcap_{s \in \text{SUCCS}^F(h)} \text{SAFE}(s,d) \cup (\mathit{Transp}(\text{SUCCS}^E(h),d) \cap \text{SAFE}(\text{SUCCS}^E(h),d))]$$

It must be noted that if a node in $T(h)$ modifies a section of d, $\mathit{Transp}(h,d) = \perp$, and it blocks movement of SEND(d) across the node h. On the other hand, if d is not modified in $T(h)$ then the corresponding SEND can be hoisted across the node h. Let `Safe`(n,d) be the solution of the equations for SAFE. Figure 2(a) shows the value of the `Safe` predicate for the data set x[1:1000].

Maximizing latency hiding requires the SENDS to be placed as early as possible. A node n is defined to be *earliest* if and only if for every SEND placed at the node n, n is the earliest possible location for the send; that is, on every path from the start node s to n, no node prior to n satisfies the safety criteria and communicates the same value as that sent at n. The following equations give the set of nodes that satisfy the earliest property [11]:

$$\text{EARLIEST}(n,d) = \begin{cases} \top & \text{if } n = s \\ \bigcup_{p \in \text{PREDS}^F(n)} [\neg \mathit{Transp}(p,d) \cup \neg \texttt{Safe}(p,d)] & \text{otherwise} \end{cases}$$

For loop header node h, communication set d, and $n = \text{SUCCS}^E(h)$, n is *earliest* if SEND(d) is not hoisted

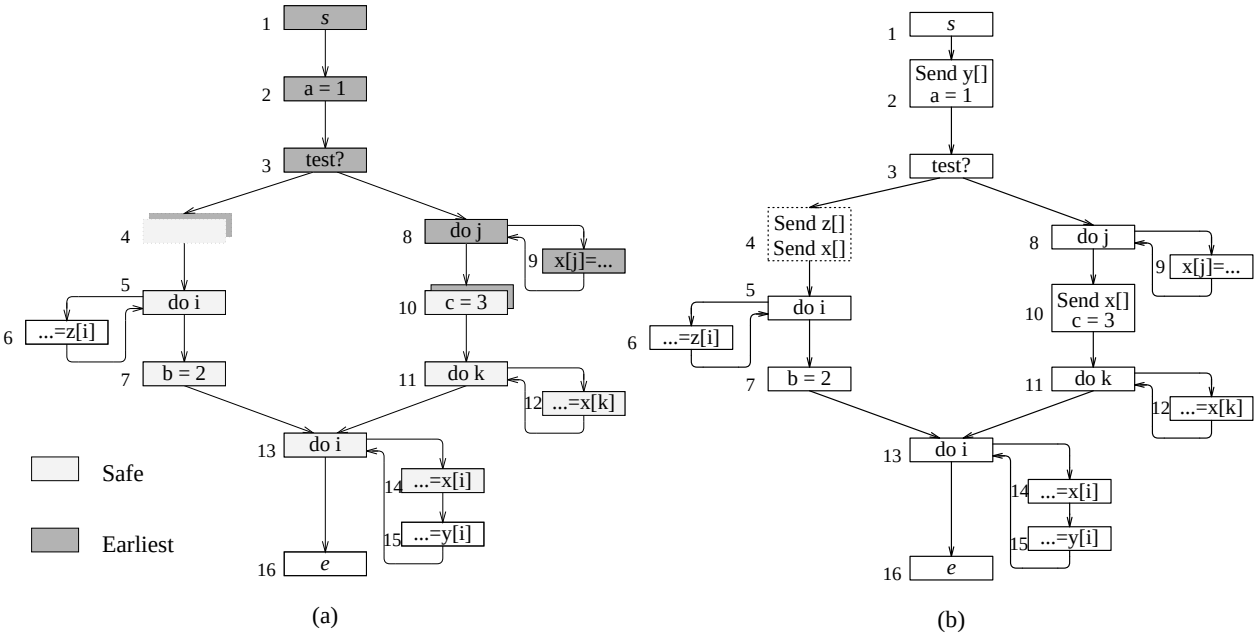

Figure 2 (a) The `Safe` and `Earliest` predicate values for the communication of x[1:1000]. (b) Placement of SENDS without taking constraints into account.

out of the loop:

$$\text{EARLIEST}(n, d) = \neg \mathit{Transp}(h, d) \cup \neg \texttt{Safe}(h, d)$$

Let **Earliest** be the solution of the equation system obtained by the FORWARD and DOWNWARD traversal of the graph. The predicate values of **Earliest** for the data set x[1:1000] are shown in Figure 2(a). **Earliest** is initialized to TRUE for the start node s. Further, nodes 2 and 3 (and 8) are not safe while nodes 8 and 9 are not transparent to the data set x[1:1000]. Therefore, **Earliest** holds for the set of nodes {1,2,3,4,8,9,10}.

It was shown by Knoop, Rüthing, and Steffen for the LCM technique that a node n is *safe* (*earliest*) if and only if **Safe**(n, d) (**Earliest**(n, d)) holds. Therefore, the solution to the two equations can be used to determine the earliest safe placement of SENDs: every node n that satisfies both the **Safe** and **Earliest** predicates defines a location for the SEND placement, since whenever **Safe**(n, d) holds there exists a node m, on every path from s to n, satisfying **Safe**(m, d) and **Earliest**(m, d) such that the data set d is not modified between m and n [11].

Figure 2(b) shows the placement of SENDs for all the three data sets. Note that the SEND x[1:1000] corresponding to the non-local reference at node 14 has been hoisted to the left branch of the **if** (node 3) to eliminate partial redundancy along the right branch. SEND y[1:1000] can be initiated at node 2 itself because it is not modified in the program. However, since z[1:256] is not used along the path {1,2,3,8,10,11,13,16}, its SEND is blocked at the node 4.

5 Balanced Receive Placement

In order to maximize latency hiding, RECVs must be placed at the latest possible node before the data is used. The balance criteria described in Section 3.1 states that each SEND should have exactly one matching RECV. Since every program execution path must have balanced SENDs and RECVs, both the communication primitives need to occur at the same loop nesting level. Figure 3 shows two examples of unbalanced placement.

From the above it can be observed that the placement must satisfy balance for each nesting level. For example, if SEND y[i] were nested in the **do** loop with header node 13, the matching RECV would also be

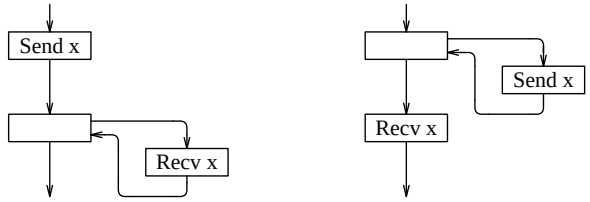

Figure 3 Unbalanced SEND/RECV placement.

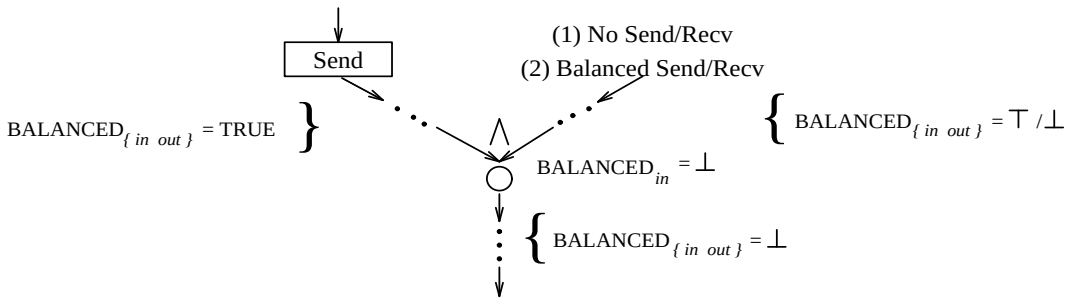

Figure 4 Evaluation of $\text{BALANCED}_{\{in\ out\}}$ using the $\wedge$ operator.

placed inside the loop. This placement can be determined independent of the placement of other data sets at the outer level and, thus, in the following equations the balance is provided by balancing all execution paths at the same level.

$\text{BALANCED}_{in}(n, d)$ is initialized to TRUE for each node that contains a $\text{SEND}(d)$ and to $\top$ otherwise. (Note that this implies that the SENDs are placed at the beginning of a node.) The nodes that would result in a balanced placement if a RECV were inserted at the node are given by the following equations:

$$\text{BALANCED}_{in}(n, d) = \text{BALANCED}_{in}(n, d) \cap \bigwedge_{p \in \text{PREDS}^F(n)} \text{BALANCED}_{out}(p, d)$$

$$\text{BALANCED}_{out}(n, d) = \text{BALANCED}_{in}(n, d) - \mathit{Used}(n, d)$$

where the operator $\wedge$ ensures that a node n is a candidate for the placement of RECV if it does not cause unbalanced placement, and $x - \top = \bot$ and $x - \bot = x$, $\forall x \in L$. While the sufficiency criteria requires that $\text{SEND}(d)$ precede the use of d, the second equation ensures that the $\text{RECV}(d)$ is also placed before the use. If RECVs are placed as determined by this equation, balance is ensured by the first equation: if one of the reaching paths at n contains a balanced SEND/RECV pair, including no SEND/RECV, then all the paths must be balanced before they reach n. As shown in Figure 4(b), $\wedge$ is defined such that $x \wedge y = \bot$, $x \neq y$ and $x, y \in L$, and $x \wedge x = x$, $x \in L$.

Let $\texttt{Balanced}_{in}(n, d)$ and $\texttt{Balanced}_{out}(n, d)$ be the predicates obtained by solving the equations. All the nodes that satisfy either of the two predicates correspond to a possible RECV placement location. The latest node for RECV placement can be determined by solving the following equations in the FORWARD order:

$$\text{LATEST}_{out}(n, d) = \text{BALANCED}_{out}(n, d) \cap \bigcup_{s \in \text{SUCCS}^F(n)} \neg\text{BALANCED}_{in}(s, d)$$

$$\text{LATEST}_{in}(n, d) = \text{LATEST}_{out}(n, d) \cup \mathit{Used}(n)$$

The equations for LATEST_{in} and LATEST_{out} recognize that if all the successors of a node n are also possible locations for the RECV placement (that is, $\text{BALANCED}_{in}(s, d) = \text{TRUE}$, $\forall s \in \text{SUCCS}^F(n)$), then n

is not the latest location and, therefore, $\text{LATEST}_{out}(n, d)$ gets set to $\perp$. $\texttt{Latest}_{in}(n, d)$ and $\texttt{Latest}_{out}(n, d)$, the solutions of the equation system, give the RECV(d) placement as follows: every node that satisfies $\texttt{Latest}_{in}(n, d)$ or $\texttt{Latest}_{out}(n, d)$ is a location for the RECV(d) placement (if both $\texttt{Latest}_{in}(n, d)$ and $\texttt{Latest}_{out}(n, d)$ are set to TRUE then RECV(d) is placed at the end of the node).

The solution to the equations gives the following placement: RECV z[1:256] is placed at the end of node 4 while RECV y[1:1000] is placed at synthetic nodes before node 13 (not shown in Figure 2; they occur between nodes 7 and 13, and nodes 11 and 13 along the two branches). Finally, RECV x[1:1000] is inserted at the end of nodes 7 and 10. Note that RECV x[1:1000] is placed at node 7 because the right branch contains a balanced placement of SEND and RECV of x[1:1000].

Besides allowing placement for different loop nests to be handled independently, our RECV placement phase can determine the placement even when some other mechanism, like dependence information based approach, is used for SEND placement. Also, we can determine the balanced receive placement even if a SEND is placed at a node such that the data is not required along every terminating path. These requirements differentiate the equations presented here from that for the latest computation placement of the LCM technique [11].

6 Taking Constraints into Account

Let *max_buffer* (1024 bytes) be the maximum size of the machine-dependent buffer that can be used by each processor for storing non-local data. We illustrate our approach by imposing the buffer size constraint on the communication placement. A message is assumed to require a buffer immediately after data is sent, and the buffer is relinquished for reuse at the last use of the received data. Since buffer is acquired by the SEND primitive, an earlier placement of SEND increases the range over which the data, and the buffer reserved for the message, needs to be live. The increased buffer requirements can violate the constraint; therefore, constraints influence the placement of SENDs.

In the following, we assume that all processors have identical communication requirements, that is, exactly same amount of non-local data is required by each processor. Also, we assume that every iteration of a loop containing a non-local reference accesses off-processor elements and that the analysis is being performed for four processors (therefore, the total buffer available among the four processors, *total_buffer*, is 4*1024 = 4096). Finally, we assume that an array element requires four bytes.

6.1 Local Constraint Analysis

In the local phase, buffer requirements for individual loop nests are computed by ignoring the buffer requirements of the statements outside the loop nest. Since the loops can be nested inside other loops, the analysis is performed by following the BACKWARD and UPWARD traversal order. The local constraint analysis requires following variables:

- $Node(n)$: Size of the buffer required for non-local accesses made at node n.
- $Loop(n)$: Buffer size for the messages sent and received in the loop with header node n.
- $Path_{min}(n)/Path_{max}(n)$: Minimum/maximum buffer required to execute a path from n to *last node* (as defined in Section 3.3).

6.1.1 Equations

$Node(n)$ is initialized to the buffer size required by the node n. For example, in Figure 2(b), $Node(14) = Node(15) = 4$, corresponding to the non-local accesses x[i] and y[i], respectively. $Loop(n)$ is initialized to zero for all nodes n. Also, $Path_{max}(e) = Path_{min}(e) = 0$. For a non-header node n:

$$Path_{max}(n) = \oplus_{s \in \text{SUCCS}^F(n)}(Node(n), Path_{min}(s))$$

$$buffer(n) = \max(buffer(n), Path_{max}(n))$$

$$Path_{min}(n) = Path_{max}(n) - \sum_{d \in \text{SEND\_SET}(n)} msg\_size(d)$$

where $\oplus$ is an addition operator, $buffer(n)$ keeps track of maximum buffer required to execute a path from a descendant[2] of n to the *last node* and is used to compute $Loop(\text{HEADER}(n))$, SEND_SET(n) is the set of messages sent at node n, and $msg\_size(d)$ is the size of the message d; it needs to be updated as messages are coalesced and vectorized.

The $Path_{min}(n)$ variable corresponds to the size of the outstanding messages at the entry of node n, that is, the size of the messages hoisted across node n. Maximum buffer required by a path from n to the *last node*, $Path_{max}(n)$, is the sum of the buffer requirement of node n and the size of the outstanding messages at the exit of node n, given by $Path_{min}(s)$ for a successor s of n. For example, in Figure 2(b), since $Path_{min}(15) = Path_{max}(15) = Node(15) = 4$, $\oplus$ is used to compute $Path_{max}(14)$ as follows: $Path_{max}(14) = \oplus(Node(14), Path_{min}(15)) = \oplus(4, 4) = 8$. The $\oplus$ operator can be made to take redundant and coalesced messages into account during buffer size computation.

Besides the communication placed in a loop due to the *safety* criteria, the messages corresponding to true dependence carried by the loop cannot be vectorized beyond the header node and require communication placement at appropriate nodes in the loop. The buffer size required for the messages placed in the loop nest is stored in the *Loop* variable for the loop header node n:

$$Loop(n) = buffer(\text{SUCCS}^E(n)) - Path_{min}(\text{SUCCS}^E(n))$$

$$Path_{max}(n) = \oplus_{s \in \text{SUCCS}^F(n)}((\text{number of iterations}) \otimes Path_{min}(\text{SUCCS}^E(n)), Path_{min}(s))$$

$$buffer(n) = \max(buffer(n), Path_{max}(n))$$

[2] A set of descendants of node n refers to the transitive closure of $\text{SUCCS}^F(n)$.

$$Path_{min}(n) = Path_{max}(n) - \sum_{d \in \text{SEND\_SET}(n)} msg\_size(d)$$

where $Path_{min}(\text{SUCCS}^E(n))$ gives the size of the messages that can be hoisted across the first node in the loop body (that is, can be vectorized); therefore, the size of the messages nested in the loop is given by $buffer(\text{SUCCS}^E(n)) - Path_{min}(\text{SUCCS}^E(n))$. The multiplication operator, $\otimes$, determines the number of non-local accesses made in the loop by taking the overlapping messages corresponding to different references into account. As an example, node 13 in Figure 2(b) corresponds to a simple application of the $\otimes$ operator: since every iteration of the **do** loop corresponding to node 13 accesses a non-local element and there are no overlapping messages, $Path_{max}(13) = \oplus(\otimes(1000, Path_{min}(14)), Path_{min}(16)) = \oplus(\otimes(1000, 8), 0) = 8000$.

Precise computation of *Loop* and, therefore, $Path_{max}$ and $Path_{min}$ variables require the compile-time knowledge of the loop bounds. In case the loop bounds are unknown, the loop bounds can be assumed to be either really small (and all the messages fits in the buffer) or really large (such that no message fits in the buffer) [14].

6.1.2 Send Placement

Each loop header node n with $Loop(n) > total\_buffer$ corresponds to a loop nest whose buffer requirements exceed the maximum allowed buffer and, thus, inhibits hoisting messages across it. In other words, node n forces the outstanding sends to be placed after the node: for each data set d with an outstanding send at node n, SAFE(n, d) is initialized to $\perp$.

Furthermore, $Loop(n) > total\_buffer$ indicates that not all messages, corresponding to the non-local references in the loop, can be vectorized without violating the constraint. In order to satisfy the constraint, there are two possibilities:

1. Select some messages heuristically and do not vectorize them any further, or
2. Stripmine the loop such that for each iteration of the inner loop the buffer requirement is less than the upper bound.

Although a compiler can use both the above-mentioned options, we demonstrate the application of the first one only. We use the following simple heuristics: select the messages in descending message size till the buffer requirement becomes less than the maximum buffer size. For each data set d selected by the heuristics, SAFE(n, d) is initialized to $\perp$; this prevents further vectorization. (If the compiler decides to stripmine the loop, SAFE(n, d) is initialized to $\perp$ for all the data sets d used in the loop and the loop header node is annotated to indicate that stripmining needs to be performed.) The constraint variables need to be updated to account for blocked messages at a node. For example, if a loop is stripmined, $Path_{min}(n)$ needs to be initialized to zero since no SENDS or RECVS are hoisted out of a stripmined loop.

In Figure 2(b), though messages corresponding to both x[i] and y[i] (at nodes 14 and 15, respectively) can be vectorized, vectorizing them requires $Path_{max}(13) - Path_{min}(16) = 8000$ bytes, more than the 4096

allowed. Using our heuristics, since both the messages are of the same size, we assume that message corresponding to y[i] is not vectorized and SAFE(13,y[i]) is initialized to $\perp$. Moreover, $Path_{max}(13)$ is set to 4000.

6.2 Global Constraint Analysis

Section 6.1 ensures that no single loop nest or message exceeds the specified upper bound. However, it does not guarantee that the sum of message sizes is also less than the maximum buffer size. We now describe how this constraint further restricts the placement of SENDs. Let $Avail(n)$ be the size of the buffer available at node n, and $Total(n)$ be the buffer required to execute a path from n to e.

6.2.1 Equations

$Avail(n)$ and $Total(n)$ are computed as follows:

$$Total(n) = Path_{min}(n) + Total(\text{SUCCS}^F(\text{HEADER}(n)))$$

$$Avail(n) = total\_buffer - Path_{max}(n) - Loop(n) - Total(\text{SUCCS}^F(\text{HEADER}(n)))$$

The size of the buffer required on a path from n to e has two components: size of the messages for a path from n to *last node*, $Path_{min}(n)$, and the buffer required for a path from *last node* to e, $Total(\text{SUCCS}^F(\text{HEADER}(n)))$. $Avail$ at a particular node depends on the amount of buffer reserved for messages that have a later use, which equals the sum of buffer required to execute a path from n to e (including the buffer required at node n itself), $Path_{max}(n) + Total(\text{SUCCS}^F(\text{HEADER}(n)))$, and the buffer required for the communication primitives placed in the interval $T(n)$, $Loop(n)$. Clearly, $Avail(e) = total\_buffer$. These equations also require a BACKWARD and UPWARD traversal of the flow graph. In fact, local and global analysis can be performed simultaneously and were described separately only to simplify presentation.

6.2.2 Send Placement

The value of $Avail(n)$, as computed above, can be less than 0. Therefore, the following represents the actual assignment:

$$Avail(n) \;=\; \max(0, Avail(n)).$$

Each node n with $Avail(n)$ that would have less than zero without the above-mentioned correction corresponds to a node that does not have sufficient buffer available to handle all the outstanding messages. In other words, node n needs to block the movement of some of the SENDs. We use the heuristics mentioned before to select the messages that are blocked by the node, and, as before, for each message d selected by the heuristics, SAFE(n, d) is initialized to $\perp$.

Figure 5 shows the propagated values of $Avail$. For example, $Avail(11) = 96$ is computed as follows: $Path_{max}(11) = \oplus(1000 \otimes Path_{min}(12), Path_{min}(13))$. Since node 12 contains x[k], a non-local access,

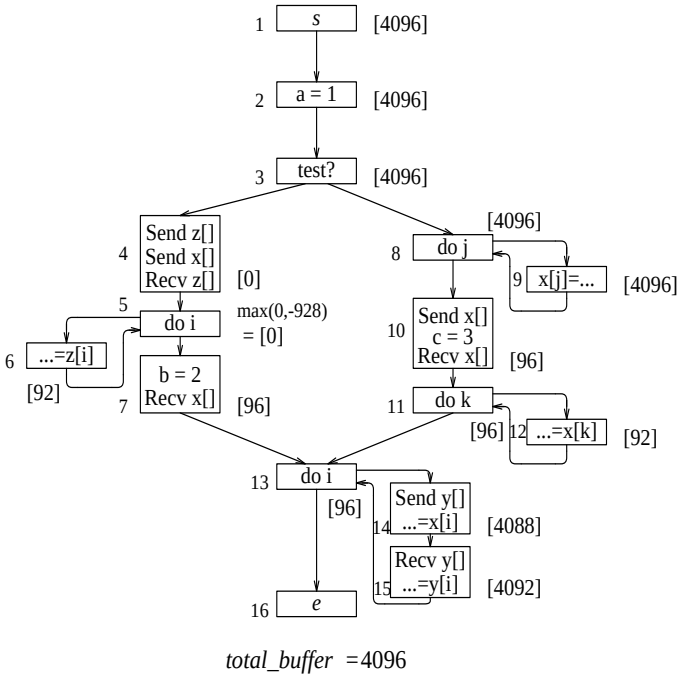

Figure 5 Propagated value of $Avail$ is shown within the brackets adjacent to the nodes.

$Path_{min}(12) = 4$, and $Path_{max}(11) = \oplus(1000 \otimes 4, 4000) = 4000$ because the message corresponding to node 12 is identical to the outstanding message due to node 14. Therefore, $Avail(11) = 4096 - Path_{max}(11) - Loop(11) - Total(\text{Succs}^F(Root)) = 4096 - 4000 = 96$, since $Loop(11)$ and $Total(\text{Succs}^F(Root) = \emptyset)$ are both zero. Node 5 shows how messages are blocked: since node 6 is the *last node*, $Path_{max}(6) = Node(6) = 4$, corresponding to the non-local reference z[i], and $Path_{max}(5) = \oplus(256 \otimes 4, 4000) = 5024$. $Avail(5) = \max(0, 4096 - 5024) = \max(0, -928) = 0$; therefore, x[1:1000] is blocked by the node 5: SAFE(5,x[1:1000]) is set to $\perp$ and the constraint variables are updated accordingly.

The earliest placement of SENDs with SAFE initialized as described in this section gives the constraint-based placement of send primitives. Figure 6 shows the send placement after the application of both local and global constraints. Note that SEND y[i] is not vectorized (node 14) and SEND x[1:1000] is placed after node 5. As described in Section 5, the matching RECV placement can be obtained that satisfies the *balancedness* property. Figure 6 also gives the placement of RECVs as specified by the $\texttt{Latest}_{in}(n, d)$ and $\texttt{Latest}_{out}(n, d)$ predicates. RECV x[1:1000] is placed at node 7 because the right branch contains a balanced placement of SEND and RECV of x[1:1000]. Since SEND y[i] occurs at loop nesting level 1, the corresponding placement of RECV y[i] is determined separately. Figure 6 shows the balanced communication placement that takes constraints into account.

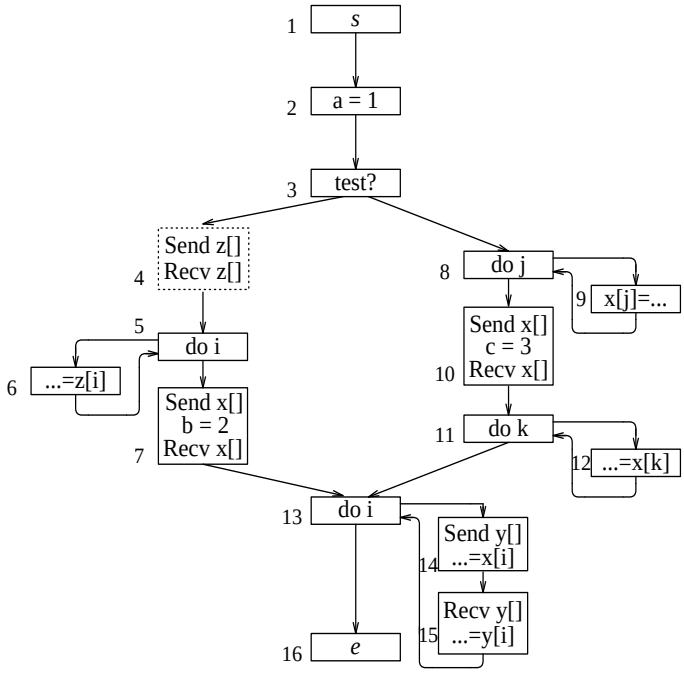

Figure 6 Communication placement after taking constraints into account.

7 Communication Optimizations

Due to space restrictions, we limit ourselves to outlining the support for different communication optimizations. The framework supports global elimination of partially redundant communication and performs message vectorization by hoisting SENDs out of loop nests whenever possible. Other communication optimizations can be incorporated by manipulating SENDs placed at the same node: communication sets can be unioned, modified or eliminated to achieve message coalescing, partly or completely redundant communication elimination, respectively. Array portions can be compared at the initialization phase to avoid array section analysis during the propagation phase [10]; that is, it is possible to implement communication placement in the absence of constraints using bit vectors, and this was the approach adopted in this paper. On the other hand, the analysis can be performed by computing and propagating array sections to obtain a more precise communication placement.

The constraint analysis phase, however, requires manipulation of communication sets. In order to compute the size of communication sets precisely, operators to determine union (for example, for message coalescing), intersection (for example, to detect redundant communication) and difference (for example, to eliminate a part of the redundant communication) are required. These operators, as well as the $\oplus$ and $\otimes$ operators described in Section 6, can be based on an appropriate array section representation [8, 4, 6].

Vector message pipelining optimization moves the SEND of a communication set towards the definition of the data. For true dependences carried by the loop corresponding to the header node h, instead of placing

all the corresponding SENDs at $\text{SUCCS}^E(h)$, vector message pipelining moves them towards the source of the dependence across the back edge and, therefore, across iterations to achieve better overlap [9]. Previous communication placement frameworks [7, 6] cannot capture this optimization since it requires the movement of sends along the back edge. However, by using the equations for send placement (Section 4) to hoist SENDs corresponding to loop-carried true dependences across the back edge to the *last node*, and its predecessors, our framework can support this optimization.

8 Summary

The communication placement framework presented in the paper has three separate phases:

1. Determine SEND placement in the absence of constraints,
2. Given a SEND placement, obtain the balanced RECV placement, and
3. Use data-flow analysis to propagate constraints and use them to determine constraint-conscious communication placement.

In the absence of constraints, steps 1 and 2 can be executed to obtain the earliest placement of SENDs and the latest placement of RECVs and, as described in Section 7, a post pass can be used to incorporate communication optimizations into the framework. We imposed the buffer size constraint, under certain simplifying assumptions, to illustrate the effect of constraints on communication placement.

Though we presented the framework in the context of communication placement, it can be adapted for other memory hierarchy related code placement problems that involve resource constraints. A separate phase for constraint analysis facilitates taking different constraints into account by initializing the SAFE variables appropriately. Furthermore, the equations for safety, earliest, latest and other placement criteria can be modified depending on the communication primitives and machine characteristics. As an example, the safety criteria can be relaxed for prefetch placement (to allow hoisting communication out of the frequently executed conditional paths, for example), or it can be restricted to the one used traditionally.

Besides placement, the framework can also be used to select appropriate primitive (prefetch or block-transfer primitive) for data movement on distributed shared-memory systems. The effect of the cache size constraint and the limited number of outstanding prefetches need to be investigated. Another potential application of the framework includes placement of I/O primitives for programs involving out-of-core I/O. For the cases where our framework does not completely hide the latency of data movement, other techniques and transformations need to be developed to maximize computation and communication overlap. Moreover, the effect of unknown (at compile time) loop bounds on the communication placement needs to be further investigated. Our future work will involve extending the framework to handle jumps out of loops and design of more precise data-flow analyses and heuristics to handle the buffer constraint.

Acknowledgments

We would like to thank the members of the Fortran D "Communication Analysis and Optimizations" subgroup, particularly Nenad Nedeljković, for many helpful discussions. We also thank Debbie Campbell and Nat McIntosh for their useful comments on an earlier draft of this paper.

References

[1] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran — A Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[2] J. Cocke and R. Miller. Some analysis techniques for optimizing computer programs. In *Proceedings of the 2nd Annual Hawaii International Conference on System Sciences*, pages 143–146, 1969.

[3] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[4] C. Gong, R. Gupta, and R. Melhem. Compilation techniques for optimizing communication on distributed-memory systems. In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[5] E. Granston and A. Veidenbaum. Detecting redundant accesses to array data. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[6] M. Gupta, E. Schonberg, and H. Srinivasan. A unified data-flow framework for optimizing communication. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[7] R. v. Hanxleden and K. Kennedy. Give-N-Take — A balanced code placement framework. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[8] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[9] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[10] K. Kennedy and N. Nedeljković. Combining dependence and data-flow analyses to optimize communication. In *Proceedings of the 9th International Parallel Processing Symposium*, Santa Barbara, CA, April 1995.

[11] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[12] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[13] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[14] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62–73, Boston, MA, October 1992.

[15] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.