Buffer-Safe Communication Optimization based on Data Flow Analysis and Performance Prediction

Thomas Fahringer Eduard Mehofer

Institute for Software Technology and Parallel Systems, University of Vienna
Liechtensteinstr. 22, A-1090, Vienna, Austria
{*tf,mehofer*}*@par.univie.ac.at*

Accepted for publication in IEEE Proc. of the 1997 International Conference on Parallel Architectures and Compilation Techniques, PACT 97, San Francisco, Nov. 1997.

Abstract

This paper presents a novel approach to reduce communication costs of programs for distributed memory machines. Our techniques are based on uni-directional bit-vector data flow analysis that enable vectorizing and coalescing communication, overlapping communication with computation, eliminating redundant messages and amount of data being transferred both within and across loop nests. Our data flow analysis differs from previous techniques that it does not require to explicitly model balanced communication placement and loops and does not employ interval analysis. Our techniques are based on simple yet highly effective data flow equations which are solved iteratively for arbitrary control flow graphs. Moving communication earlier to hide latency has been shown to dramatically increase communication buffer sizes and can even cause runtime errors. We use P^3T, a state-of-the-art performance estimator to create a buffer-safe program. By accurately estimating both the communication buffer sizes required and the implied communication times of every single communication of a program we can selectively choose communication that must be delayed in order to ensure a correct communication placement while maximizing communication latency hiding. Experimental results are presented to prove the efficacy of our communication optimization strategy.

1 Introduction

The overhead to access non-local data from remote processors on distributed memory architectures is commonly orders of magnitude higher than the cost of accessing local data. As a consequence, a key problem to effectively use distributed memory architectures is centered around efforts to optimize communication [4, 11, 2, 15, 10, 17, 1, 16] which includes: message vectorization, message coalescing, collective communication, communication latency hiding and pipelined communication. The effect of these optimizations is limited by the fact that most of the analysis in current parallelizing compilers is performed for a single loop nest at a time, and very few research efforts have been started to optimize communication globally across arbitrary control flow.

Most approaches [11, 18, 17, 12] for global scheduling of communication commonly rely on data flow analysis which is based on array sections and data dependence analysis.

Commonly data flow analysis is used to place SENDs as early and RECVs as late as possible in order to maximize communication latency hiding. In addition message coalescing is used to further eliminate redundant communication. Placing SENDs as early as possible increases the life-time of communication buffers that are used to unpack and reference non-local data. In [18] it has been shown that runtime or program loading errors can be caused if maximum available communication buffers of individual processors exceed.

This paper describes a novel communication optimization framework that is based on a global unidirectional bit-vector data flow analysis. Our framework places SENDs as early and RECVs as late as possible to exploit opportunities for hiding communication by overlapping communication with computation. The number of messages exchanged is reduced by hoisting communication to the outer-most possible program point. P^3T [5, 6], a state-of-the-art performance estimator, is used to predict the required communication buffer size with high accuracy. In order to ensure buffer-safe communication placement while optimizing communication we selectively delay those communication with the smallest estimated communication time (obtained by P^3T) at program points where the maximum available buffer size exceeds. Furthermore, we aggressively coalesce messages that are partially redundant which reduces both amount of data transferred and number of messages exchanged.

The important contributions of this paper are as follows:

First, our data flow analysis is based on an efficient uni-directional bit-vector algorithm. This is in contrast to communication optimization that is based on bi-directional data flow analysis [11], which either requires an algorithm with several backward and forward analysis, or must be decomposed into into a set of unidirectional problems. Bi-directional data flow analysis are more complex and commonly imply performance problems.

Second, most existing approaches require separate data flow equations to model loops and balanced (SENDs and RECVs are placed at program positions that have identical statement execution counts) communication placement as well as employ interval analysis. We use simple yet highly effective data flow equations which implicitly ensure balanced communication and are solved iteratively for arbitrary control flow graphs.

Third, hardly any communication optimization considers communication buffer constraints which is crucial to ensure that the communication placement does not exceed the maximum available communication buffer. Previous techniques [18] use very coarse grain cost functions which results in overly conservative assumptions and in turn directly impacts the resulting communication performance. In [18] a buffer-safe communication placement is presented that blocks all communication at a program node if the communication buffer exceeds at that node. Our approach selectively blocks communication with small communication time whereas communication with larger communication time is hoisted to the earliest possible program point until all buffer constraints are honored. Clearly, selective communication blocking as compared to blocking all communication in case of exceeding the communication buffer, has an increased potential for latency overlapping.

Furthermore, we use symbolic analysis to effectively examine and represent data dependences, array sections and communication patterns. For this purpose, we developed a powerful symbolic analysis package [9, 8, 7] that handles linear and non-linear array subscript expressions, complex loop bounds and deals with unknowns such as processor numbers and problem sizes. For detailed mathematical analyses of our symbolic analysis, which goes beyond the scope of this paper, the reader may refer to [9, 8, 7].

The organization of this paper is as follows. In Section 2, we describe the program model and give an overview of P^3T. Section 3 describes how to place SENDs as early and RECVs as late as possible, to coalesce messages and to ensure balanced and buffer-safe communication placement. Preliminary results that demonstrate the benefits of the proposed method are presented in Section 4. Related work will be discussed in Section 5. Section 6 will give our conclusions.

2 Preliminaries

2.1 Program model

A control flow graph (*CFG*) of a program is defined by a directed flow graph $G = (N, E, e, x)$ with a set of nodes N and a set of edges E. A node $n \in N$ represents a program instruction (statement). An edge $(m, n) \in E$ indicates transfer of control between instructions $m, n \in N$. For the ease of presentation, all of our data flow analysis is employed at the instruction level. In fact, it can be straightforward modified to work on basic blocks. Each node n has a unique *entry* and *exit* where the entry (exit) of n is immediately before (after) n. e and x are the unique *start* and *end* node of G, which are assumed not to possess any predecessors and successors, respectively. $succs(n)$ and $preds(n)$ correspond to the set of successor and predecessor nodes of n. A path in G is a sequence of nodes $[n_1, ..., n_k]$ such that $\forall 1 \leq i < k$ the following holds: $n_{i+1} \in succs(n_i)$. Every node $n \in N$ is assumed to lie on a path from e to x. An instruction may *write* or *use* data references (array or scalar variables). A program is executed by a set of processors P. The computations and data of a program are distributed to one or more processors in P according to the underlying data and work distribution strategy [3, 14, 15]. Every processor contains a private copy of all scalars. Each data element is *owned* by one or more processors. Our strategy covers both non-local uses and non-local writes. Non-local data must be fetched before it is used by a processor. If a processor writes data that is owned by another processor, then after the write operation this data must be transferred back to the owning processor. In this paper we focus on the *owner-computes-strategy* which means that the processor that owns a datum will perform the computations that make an assignment to this datum.

Let U be the set of all non-local uses and S the set of SENDs in a program.

Every non-local use implies a communication which can be realized by a specific SEND/RECV pair, by several SENDs combined with a specific RECV, or by a specific SEND combined with several RECVs. The second case can occur if the underlying data flow analysis is hoisting a SEND upwards into several different control flow branches that reach the associated non-local use as described in Section 3.1. The third case may occur if a specific SEND covers several non-local uses as a result of message coalescing (see Section 3.2). The data transferred by a communication is referred to as *communication data*.

We define a SEND s to be *associated* with a non-local use u if s originates from u such that s has been generated based on hoisting SENDs. Furthermore, we define a SEND s to *cover* a non-local use u if part or all of u is being sent by s. Every non-local use u that is associated with s is also covered by s, whereas not every non-local use u that is covered by s is also associated with s. *Sends(u)* is the set of SENDs that are associated with $u \in U$. *Uses(s)* defines the set of non-local uses that are covered by a specific SEND $s \in S$.

2.2 Performance prediction

In order to support eliminating communication buffer conflicts and enhancing communication optimization we use P^3T [5, 6], a highly accurate and efficient performance estimation tool for distributed memory parallel programs. P^3T statically estimates a set of performance parameters which includes: work distribution, number of transfers (messages exchanged), data volume transmitted, network contention, communication and computation time, and number of cache misses. In the following we briefly describe the P^3T parameters that are used to support the techniques described in this paper:

- *Amount of Data Transferred* [6] is an estimate of the number of data elements transferred by a communication statement. In order to obtain highly accurate results this parameter among others models data distribution strategies, data access patterns, control flow, and machine specific data type information.

- *Communication Time* [6] is a sensible measure combining among others amount of data transferred, number of transfers, control flow information, as well as various machine specific indices such as message startup overhead, message transfer time per byte, sizes for different data types, and even processor distances. The communication time of a specific communication statement s is estimated by the maximum communication time across all processors involved in s.

All performance parameters can be optionally estimated for a specific statement, loop, procedure and the entire program. Furthermore, the outcome of every parameter can be given for a specific processor. It is assumed that problem size and machine parameters are known at compile time which is a common assumption made for many performance estimators. Characteristic values for program unknowns (e.g. statement execution and loop iteration counts) are derived by a single profile run [5] based on the original sequential program. We have shown [5] that large portions of the profile data can be automatically adapted for many important program changes without redoing the profile run.

The original P^3T was restricted to communication based on overlap areas [3] surrounding the local portion of arrays. We have extended P^3T to cover also general buffer communication where data is received into a buffer that is allocated dynamically, and the array reference that led to communication is replaced by a reference to the buffer.

3 Communication placement framework

In this section we describe our communication optimization strategy. First, we hoist SENDs to the earliest possible program point without considering communication buffer constraints. Second, we coalesce SENDs based on the same arrays. Third, we place RECVs as late as possible. Finally, we present our approach of placing SENDs/RECVs considering buffer constraints while optimizing communication.

3.1 Earliest SEND placement without considering buffer constraints

In order to maximize latency hiding (without considering buffer constraints), SENDs must be hoisted to the earliest possible program point, while maintaining the program semantics. Starting point of this analysis is the set of non-local uses U. At the beginning every $u \in U$ implies a SEND s, which is then hoisted upward in the opposite direction of the control flow. If s is hoisted through a node n with several predecessor nodes, then s is hoisted into every predecessor node of n which results in multiple copies of s – one for each predecessor node of n. All copies of s are stored as unique SENDs in $Sends(u)$. In principle, hoisting a SEND s proceeds as long as there is no node n encountered that implies a true dependence affecting the communication data of s, or if n has at least one successor node n' such that there does not exist an $s' \in Sends(u)$ which can be hoisted to n'.

The analysis of hoisting SENDs is based on a backward directed bit-vector data flow analysis [13] which

Hoistability Analysis:

$$
\begin{aligned}
\text{N-HOIST}_n &= \text{X-HOIST}_n * \overline{\text{BLOCK}_n} + \text{USE}_n && (1)\\
\text{X-HOIST}_n &= \begin{cases} \text{false} & n = x \\ \text{UNBLOCK}_n + \prod_{m \in succs(n)} \text{N-HOIST}_m & \textit{otherwise} \end{cases} && (2)
\end{aligned}
$$

Inserting SENDs as early as possible:

$$
\begin{aligned}
\text{N-EARLIEST}_n &= \text{N-HOIST}^*_n * \begin{cases} \text{true} & n = e \\ \sum_{m \in preds(n)} \overline{\text{X-HOIST}^*_m} & \textit{otherwise} \end{cases} && (3)\\
\text{X-EARLIEST}_n &= \text{X-HOIST}^*_n * \text{BLOCK}_n && (4)
\end{aligned}
$$

Figure 1: Data flow Equations for Placing SENDs

uses the following local predicates that are defined for every CFG node $n \in N$:

> USE_n: There is a non-local use u in n that implies a SEND s.
>
> BLOCK_n: A SEND s is blocked in node n if n contains the source of a true dependence that affects the communication data of s, or n is a loop header which carries a true dependence that affects the communication data of s.
>
> UNBLOCK_n: A SEND s can be hoisted out of a loop if n corresponds to a loop header of a loop l such that l and all loops enclosed by l do not carry a true dependence that affects the communication data of s.

A node $n \in N$ may also block a SEND – by appropriately setting BLOCK_n and UNBLOCK_n – if n does not honor communication buffer constraints (see Section 3.4.2 for more details).

The data flow equations for hoisting SENDs are shown in Figure 1. They as well as the ones of Figure 5 have been partially adapted from [19] where a similar set of equations has been used to model assignment motion for sequential programs. As common for code motion systems, we also assume that *critical edges* – e.g. edges leading from a node with more than one successor to a node with more than one predecessor – are eliminated by standard techniques [19].

Equations (1) - (2) of Figure 1 present the hoistability analysis in bit-vector format, where each bit corresponds to a SEND that is associated with a non-local use u occurring in the program. Note that every SEND $s \in Sends(u)$ is mapped to the same bit. Here N-HOIST_n and X-HOIST_n intuitively mean that a SEND can be placed at the entry or the exit of an instruction n, respectively, where SEND is not blocked according to the meaning of $BLOCK_n$. X-HOIST_n is initialized to false for the end node.

Traditionally, a SEND can only be placed in a node n if the transferred data is used along all terminating paths starting at n. Therefore, hoisting SENDs outside of (zero-trip) loops is not considered semantic-preserving as new values (communication data) can be introduced on some control flow branch of the program. This strictly semantic-preserving strategy, however, would imply critical communication overheads in many cases [5, 6] as communication inside of loops is commonly more expensive (due to its increased statement execution count) than communication outside of loops. We relax this restriction by introducing a local predicate $UNBLOCK_n$ that among others allows hoisting communication outside of loops – that do not carry (or contain a loop that carries) a true dependence affecting this communication – which commonly implies a large reduction in communication overhead but at the cost of a possible over-communication (in case of zero-trip-loops).

The greatest solution of the equation system (1) - (2) – characterized by N-HOIST^*_n and X-HOIST^*_n – specify the nodes, where SENDs can be legally inserted. Equations (3) - (4) determine the earliest

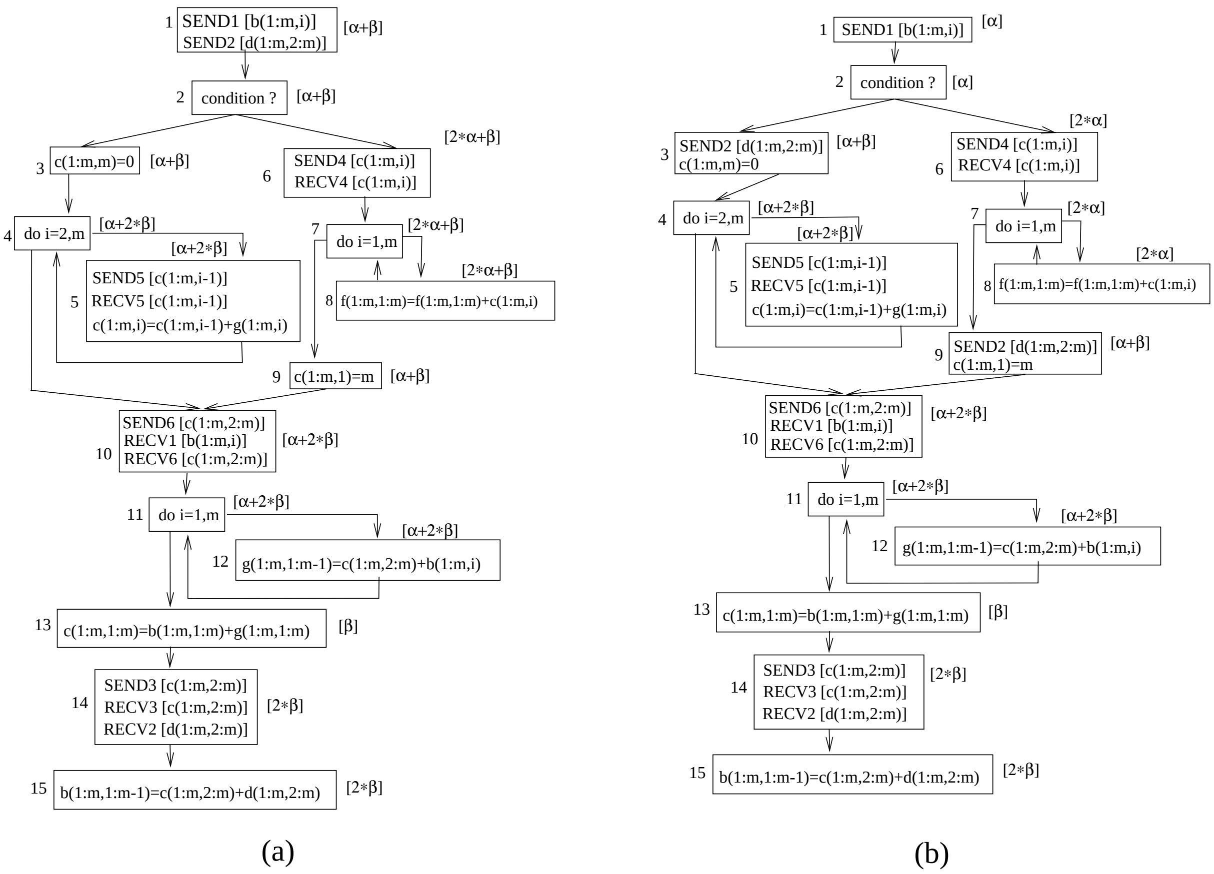

Figure 2: Running code example (SENDs/RECVs refer to non-local use that causes communication). (a) earliest SEND placement without buffer constraints, (b) earliest SEND placement with buffer constraints

possible nodes where communication can be placed if buffer constraints are not considered, by means of the functions N-EARLIEST$_n$ and X-EARLIEST$_n$.

Figures 2 (a)-(b) show our running code example where all arrays (dimension m x m) are column-wise distributed onto a set of P processors. For the sake of clarity, the guards and processor loops enclosing communication primitives are not shown in Figure 2. The SENDs of the code in Figure 2 (a) are placed as early as possible without considering buffer constraints. Note that SEND1 and SEND6 have been hoisted out of loops 11 and SEND4 out of loop 7 through appropriately initialized values for $UNBLOCK_n$. SEND5 cannot be hoisted out of loop 4 which carries a true dependence that affects *c(1:m,i-1)*, the communication data of SEND5.

Throughout the paper we will use both Fortran 77 and Fortran 90 style of describing array accesses. Furthermore, for the sake of simplicity, in all code fragments that we present, SEND and RECV statements do not actually display the array sections being communicated but instead we specify the non-local use that causes communication. For instance, *SEND1 [b(1:m,i)]* at node 1 refers to the communication necessary for accessing non-local use $b(1:m,i)$. Although the loop variable i is undefined at node 1, for the sake of demonstration, i is specified in *SEND1 [b(1:m,i)]* in order to clearly display the relation between SEND/RECVs with their corresponding non-local use. We follow these policies throughout the paper.

3.2 Message coalescing based on the same array

Once the SENDs of a program have been placed as early as possible we coalesce messages that are partially redundant and are based on the same array. Figure 3 shows our algorithm for message coalescing. *dom(s,s')* specifies that s dominates s'. If s appears

```
FOR all s ∈ S DO
    FOR all u ∈ U such that s ∉ Sends(u) DO
        FOR all s' ∈ Sends(u) DO
            IF dom(s,s') and subsume(s,s') THEN
               CommD(s'):=CommD(s')-LiveCommD(s,s')
               IF CommD(s') = φ THEN
                  S := S − s'
                  Sends(u) := Sends(u) − s'
               ENDIF
               IF LiveCommD(s,s') ≠ φ THEN
                  Uses(s) := Uses(s) ∪ Uses(s')
               ENDIF
            ENDIF
        ENDFOR
    ENDFOR
ENDFOR
FOR all u ∈ U DO
    FOR all s' ∈ Sends(u) DO
        CommD(s') := ⋃_{s' ∈ Sends(u)} CommD(s')
    ENDFOR
ENDFOR
```

Figure 3: Message coalescing based on the same arrays

on every path from the start node e to s', then s dominates s'. *subsume(s,s')* denotes that for every exchange of data d' between a specific sending processor a and receiving processor b as implied by s' there exists an exchange of data d invoked by s with the same sender a and receiver b such that d' is a subset of d. *CommD(s)* is the data communicated by a SEND s. *LiveCommD(s,s')* – by analogy with the live variable problem [13] – refers to the communication data of s' that is not written between s and s'.

The algorithm involves two phases. In phase one, every SEND s is examined whether s dominates and subsumes a SEND s' that is implied by a non-local use $u \in U$ such that $s \notin Sends(u)$. The communication data of s' that is not written between s and s' can be eliminated from s' as it is covered by s. If s' is made partially redundant by s, then the uses of s' are added to the uses of s. The communication data of s' that is written between s and s' must still be sent by s'. Note that s is not added to Sends(u).

In phase two, we determine the communication data for all SENDs. Every SEND may have been made partially redundant by some other SEND according to phase 1. In order to alleviate code generation every $s' \in Sends(u)$ is sending the union of all communication data across all $s' \in Sends(u)$. This enables to generate the same code for every $s' \in Sends(u)$ as well as the same code for every associated RECV of all SENDs in $Sends(u)$. Note that a SEND can imply several (identical) RECVs as it may cover several different non-local uses.

Figures 4 (a)-(b) show a code example (not related to the running example of Figure 2 (a)-(b)) with two 2-dimensional arrays (column-wise distributed) of dimension m x m. SEND1/RECV1 and SEND2/RECV2 are associated with the non-local uses in node 2, and SEND3/RECV3 with the use in node 7. Note that SEND4 at node 5 and 6 – as placed by the algorithm described in Section 3.1 – are associated with the same RECV4 and originate from the use in node 9. The write access to b in node 4 blocks SEND4 at node 5 which yields LiveCommD(SEND2,SEND4) = ϕ. As a consequence SEND2 does not have an impact on SEND4 in node 5. Whereas SEND4 in node 6 is made redundant by SEND2 (LiveCommD(SEND2,SEND4) = CommD(SEND4)). Note that after SEND4 at node 6 has been eliminated, the associated RECV4 is placed in node 5 in accordance with the analysis of placing RECVs (see Section 3.3).

SEND3 cannot be hoisted out of node 3 as it is only invoked in the right branch of node 3. According to the algorithm of Figure 3, SEND1 dominates and subsumes SEND3. SEND3 implies exactly the same communication pattern as SEND1, therefore, SEND3 is made redundant by SEND1 as the associated communication data of SEND3 is not written between node 1 and node 6 (therefore, LiveCommD(SEND1,SEND3) = CommD(SEND3)). Message coalescing eliminates SEND3 and also the need for inserting RECV3 (see Section 3.3).

Note that SEND1 does not subsume SEND2. SEND1 implies a message exchange of every pair of neighboring processors, whereas SEND2 requires the processor that owns the m-th column to send this column to all other processors.

Figures 4 (a) and (b) show the resulting code before and after message coalescing, respectively.

3.3 Latest RECV placement

In order to maximize latency hiding we have to place RECVs as late as possible before their non-local use's but after the associated SENDs. The analysis starts from the set of SENDs. At the beginning every SEND s implies a unique RECV r which is then delayed downward to the latest possible program point before the communication data is used.

If r is delayed below a branch node n, then r is delayed into every branch that starts at n which results in multiple copies of r.

The analysis of delaying RECVs is based on a for-

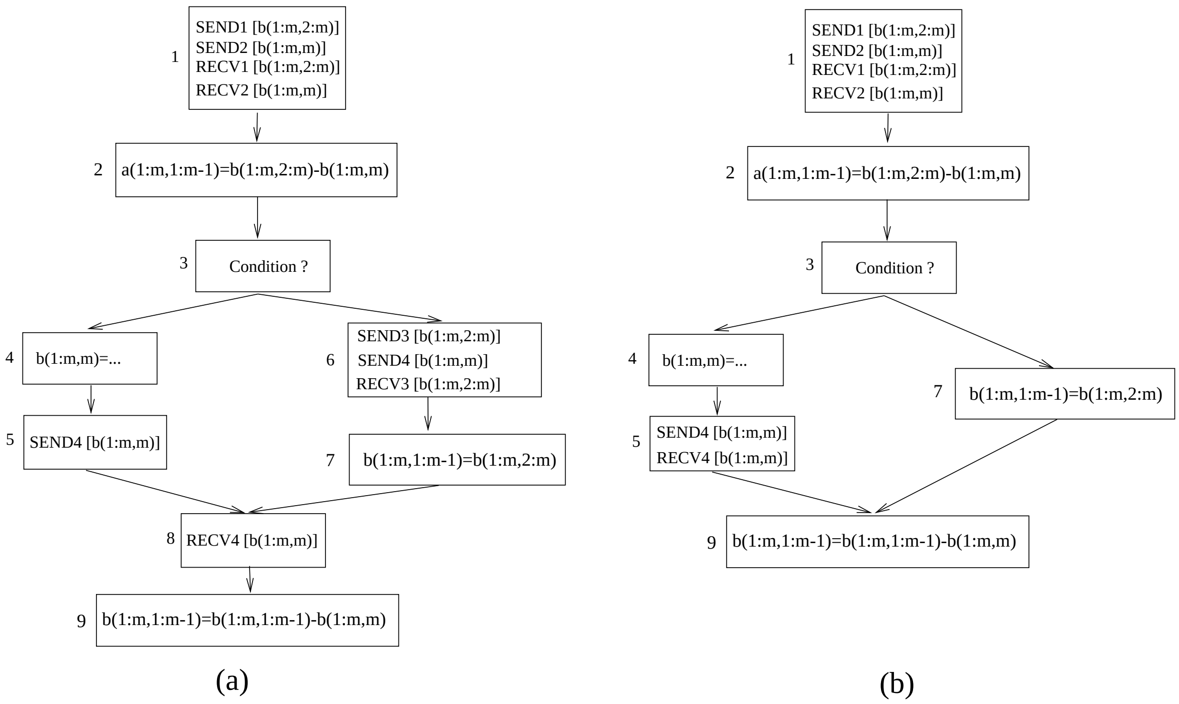

Figure 4: (a) SEND/RECV placement without message coalescing, (b) SEND/RECV placement with message coalescing

ward directed bit-vector data flow analysis which uses the following local predicates that are defined for every CFG node $n \in N$:

> USE_n: There is a non-local use u in n that uses the communication data of a RECV r.
>
> SEND-CAND_n: There is a SEND s in node n which implies a RECV r.

Equations (5) - (6) in Figure 5 present the delayability analysis. Here N-DELAY_n and X-DELAY_n intuitively mean that a RECV r associated with a SEND s can be placed at the entry or at the exit of an instruction n, respectively, where r is not blocked by a non-local use u that uses communication data of r. N-DELAY_n is initialized to false for the start node.

Equation (5) ensures that a RECV can only be inserted at the entry of n if it can be placed at the exit of all predecessor nodes, which is necessary to prevent placement of a RECV that is not reached by an associated SEND. SEND-CAND_n in Equation (6) guarantees that r cannot be placed before an associated SEND.

Equations (7)-(8) guarantee that a RECV r is inserted either at the entry or at the exit of a node n. A RECV r is placed at the entry of a node n if r can be delayed until the entry of n and there is a use of the communication data in n. r is placed at the exit of a node n if it can be placed there according to Equation (6) and there exists at least one successor node m of n such that r cannot be delayed until the entry of m. N-DELAY^*_n and X-DELAY^*_n denote the greatest solution of the equation system for delayability.

RECVs are only inserted for those non-local uses whose associated SENDs have not been eliminated by message coalescing. The communication data of a RECV is given by the union of communication data across all $s \in Sends(u)$ (see Figure 3).

Figures 2 (a)-(b) and Figure 4 (a)-(b) display several codes with RECVs appropriately inserted for their associated SENDs.

Note that our data flow equations as shown in Figures 1 and 5 implicitly guarantee balanced placement of SENDs with their associated RECVs which means that all associated SENDs and RECVs are placed in nodes with identical statement execution counts.

3.4 Earliest SEND placement considering communication buffer constraints

Commonly every communication is associated with a communication buffer that is used to unpack and reference non-local data. It has been shown [18] that the size of such buffers – depending on the application –

Delayability Analysis:

$$\text{N-DELAY}_n = \begin{cases} \text{false} & n = e \\ \prod\limits_{m \in preds(n)} \text{X-DELAY}_m & otherwise \end{cases} \quad (5)$$

$$\text{X-DELAY}_n = \text{SEND-CAND}_n + \text{N-DELAY}_n * \overline{\text{USE}_n} \quad (6)$$

Inserting RECVs as late as possible:

$$\text{N-LATEST}_n = \text{N-DELAY}^*_n * \text{USE}_n \quad (7)$$

$$\text{X-LATEST}_n = \text{X-DELAY}^*_n * \begin{cases} \text{true} & n = x \\ \sum\limits_{m \in succs(n)} \overline{\text{N-DELAY}^*_m} & otherwise \end{cases} \quad (8)$$

Figure 5: Data flow Equations for Placing RECVs

can be significantly larger than those required for local data sections. Ignoring buffer constraints when placing SENDs can lead to serious problems while loading a program onto a target architecture or even to runtime errors. Hoisting communication to the earliest possible point in a program increases the life-time of communication buffers and therefore can cause a dramatic increase of buffer requirements for non-local data. In this section we describe how our techniques for latency hiding and message coalescing are extended to be buffer-safe.

3.4.1 Add communication buffer requirements to program nodes

Our code generation policy implies the insertion of a nonblocking *receive* immediately after a *send* operation for every SEND as placed by our hoistability analysis. Moreover, every associated RECV as placed by our delayability analysis is replaced by a blocking *wait*. Therefore, a communication buffer remains *live* from a SEND s until all of its uses $u \in$ *Uses(s)*.

Let *buffer(s)* denote the buffer size required by a SEND s, and *buffer-req(n)* the sum of buffer sizes required by a node $n \in N$ according to the buffers that are live at n. *buffer(s)* is computed by P^3T's parameter for the *amount of data transferred* as described in Section 2.2. *buffer(s)* is defined as the maximum amount of data transferred across all processors involved in s.

We traverse the CFG and for every node $n \in N$ and every SEND $s \in S$ we add *buffer(s)* to *buffer-req(n)* iff s is live at n.

Continuing our example of Figure 2 (a) which displays for every node n the corresponding *buffer-req(n)* in square brackets. Note that all arrays are assumed to be of dimension m x m and are column-wise distributed onto a set of P processors. *buffer(SEND1)* = *buffer(SEND4)* = a implies a broadcast of all local segments. All other SENDS require a buffer size of β (single array column from a neighboring processor), where $\alpha = \frac{m^2*(|P|-1)}{|P|}$ and $\beta = m$. α and β are given as number of data elements of a generic data type. For instance, the buffer requirements of node 1 is therefore given by $\alpha + \beta$. α and β are respectively implied by SEND1 and SEND2.

Note that a specific use u may be associated with several SENDs. In order to prevent summing up the buffer size of the same communication multiple times we verify that *buffer-req(n)* is incremented at a maximum once for all SENDs in *Sends(u)*.

3.4.2 Delay SENDs if communication buffer exceeds

Let *node(u)* specify the associated CFG node that contains the non-local use u and *max-buffer* is the maximum size of the communication buffer size available on every single processor of a target architecture. Furthermore, *LiveSends(n)* is the set of SENDs whose associated buffers are live at node n.

Figure 6 depicts our overall algorithm for buffer-safe communication optimization. Within the REPEAT-loop we first place SENDs as early as possible without considering buffer constraints according to

```
H := S
REPEAT
  stable := true
  resolve hoistability analysis for H
  coalesce SENDs for H
  H := φ
  FOR every n ∈ N in reverse depth first order DO
      IF ∄ s such that s ∈ H and s ∈ LiveSends(n) THEN
          IF buffer requirements of all non-local uses in n exceeds max-buffer THEN
              /* Non-local uses of node n require more buffer space than available */
              EXIT
          ELSE
              WHILE buffer-req(n) > max-buffer DO
                  stable := false
                  determine s ∈ LiveSends(n) where s has smallest communication time and
                      ∄ u' ∈ Uses(s) such that node(u') = n
                  FOR all u ∈ Uses(s) DO
                      FOR all s' ∈ Sends(u) such that s' reaches n DO
                          H := H ∪ s'
                          set BLOCK_n to true for s'
                          for all loop header nodes whose loop body contains n set UNBLOCK_n to false for s'
                      ENDFOR
                  ENDFOR
                  buffer-req(n) := buffer-req(n) - buffer(s)
              ENDWHILE
          ENDIF
      ENDIF
  ENDFOR
UNTIL stable = true
resolve delayability analysis for S
```

Figure 6: Buffer-safe latency hiding and message coalescing

the hoistability analysis of Section 3.1. After coalescing SENDs according to the techniques described in Section 3.2, we traverse the entire CFG and selectively block those SENDs with smallest communication time that are live at n. The REPEAT-loop terminates as soon as all SENDs are placed without buffer conflict. Finally, the RECVs are placed as late as possible according to the algorithm of Section 3.3.

A SEND s that has been blocked due to an exceeding communication buffer, may no longer cover all of its non-local uses in $Uses(s)$ as before blocking. For this reason, if any SEND is being blocked at a node n due to a buffer conflict, the hoistability analysis and message coalescing must be redone with updated predicates (BLOCK and UNBLOCK) for every SEND s' that reaches n and that is covered by s. If s' does not reach n then s' is still covered by s even after s is being blocked at n. A SEND s reaches n if there exists a path between the node where s is placed and n.

Traversing the CFG tree in reverse depth first order favors blocking first those SENDs at a node n that are closest to any of their non-local uses. We only examine those nodes where none of the SENDs as collected in H are live. H basically contains the SENDs that have been blocked or covered by a blocked SEND (and reaching n). *buffer(s)* of all SENDs $s \in H$ may no longer be correct without redoing the entire analysis of hoisting and coalescing SENDs. Therefore, determining whether the buffer requirement of a node is below *max-buffer* can only be done for nodes n such that all SENDs that are live at n are not in H.

Before it is tried to make a program buffer-safe, we must verify whether the buffer requirements of all non-local uses of a node n can be satisfied by *max-buffer*. If not then splitting n such that the non-local uses are distributed onto several nodes could be used to overcome this case which goes beyond the scope of this paper. As long as the buffer requirements of n exceed *max-buffer* we have to delay SENDs that are live at n. For this purpose we choose – in each iteration of the WHILE-loop – the SEND with the smallest communication time whose communication data is not used in n. The latter condition is important, as we must guarantee that a SEND is initiated before its uses. Choosing a SEND for being delayed at a node n is critical in order to maintain the best choices for placing SENDS. The SEND with the smallest communication time most likely has the least impact on optimizing communication. We use P^3T to compute the communication time of SENDs as outlined in Section 2.2.

In continuation of our code example in Figure 2 (a), we assume that *max-buffer* $= 2 * \alpha$ and $\beta \leq \frac{\alpha}{2}$. As a consequence SEND2 is delayed by our algorithm at nodes 7 and 8 as SEND1 and SEND4 (broadcast communication) imply a higher communication time than SEND2 (single exchange with a neighboring processor) and *c(1:m,i)* – the associated communication data of SEND4 – is used in node 8. Afterwards all buffer constraints are honored by all nodes of the program. Figure 2 (b) shows the updated buffer requirements for all nodes. SEND2 is now placed at node 9 and 3. As SEND2 does not imply any buffer to exceed at the left branch of the branch node it can be placed at node 3. SEND2 cannot be hoisted out of node 2 as it is blocked at node 9 due to buffer constraints.

At this stage we have a buffer-safe program where SENDs are placed as early and RECVs as late as possible.

4 Experiments

The cost models as described in this paper and the underlying symbolic analysis are fully implemented. Our communication optimization strategy based on data flow analysis is currently being implemented as part of *VFCS* (Vienna Fortran Compilation System [3]), a compiler for distributed memory architectures. Nevertheless, we were able to run the following experiments to measure the potential benefits of our buffer-safe cost driven communication optimization.

We optimized the original code of Figure 2 (a) by using the current VFCS and then hand-instrumented the code to generate two communication placements C1 and C2. C1 (see Figure 2 (b)) represents a buffer-safe communication placement incorporating latency hiding and message coalescing (based on the same arrays) as described in this paper. In order to generate C2, *VFCS* automatically places communication (associated with a non-local use u) before the outermost loop which does not impose a true dependence on u, or just before the statement containing u if no such loop exists. Furthermore, C2 employs message coalescing which is applied to arrays that are placed at the same program point only. C1 focuses primarily on latency hiding as the given code has very little potential for coalescing messages based on the same arrays, whereas C2 basically employs standard message vectorization and coalescing.

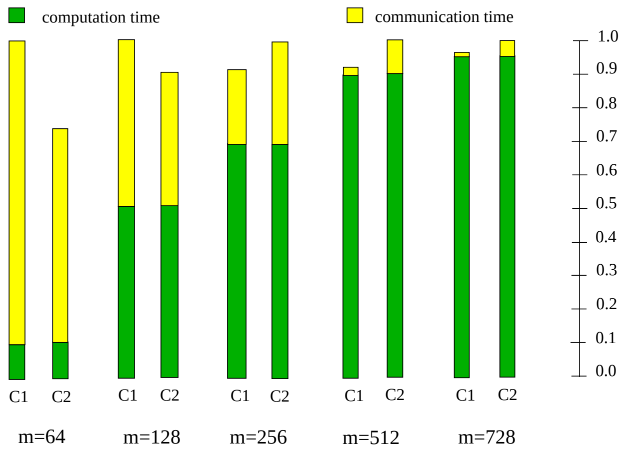

Figure 7: Normalized measured runtimes of code versions based on latency hiding and message coalescing (C1 - see Figure 2 (a)) versus standard communication optimization (C2) on a Meiko CS-2 with 16 processors

We ran our tests on 16 processors of a Meiko CS-2 distributed memory architecture for 5 different problem sizes ranging from $m = 64$ to $m = 728$ (m is the array dimension size). We report the results in Figure 7. In each bar-chart the x-axis is the problem size m. For each problem size 2 bars are plotted, one for each code version C1 and C2. The y-axis is normalized so that the code version with longest runtime has unit runtime 1.0. The dark and bright segments represent computation and communication time, respectively. The plotted communication time reflects the communication overhead that could not be overlapped with computation. The computational overhead of the code versions is of the order $\Theta(m^3)$, whereas communication overhead is much smaller. The number of transfers implied is of the order $\Theta(m)$ in the worst case and the data volume exchanged is of the order $\Theta(m^2)$. Therefore, the runtime impact of reducing communication overhead is decreasing for increasing problem sizes. For small problem sizes communication time is much higher than computation time which reflects the high message startup overhead. Applying message coalescing (reduces the number of messages) is therefore more critical than latency hiding for small problem sizes. Therefore, C2 outperforms C1 for small problem sizes. For larger problem sizes the computational overhead increases rapidly which improves the opportunities to overlap computation with communication.

Therefore, large problem sizes cause C2 to perform very poorly due to its lack of latency hiding, whereas C1 – aggressively overlaps communication with computation – becomes superior to C2.

5 Related work and discussion

Communication optimization has been extensively researched. Most existing compilers employ a combination of message vectorization and coalescing and collective communication which is based on single-loop analysis. Pipelined communication [15, 10, 17, 1] has been introduced to achieve a fine-grain communication latency hiding. Data flow analysis based communication generation has been addressed by several researchers both to eliminate redundant communication within a loop nest [1] and across loops [11, 18, 17, 12].

Von Hanxleden and Kennedy [12] optimized communication with respect to latency hiding for irregular programs. SENDs and RECVs are balanced. Buffer constraints are not considered. Arrays are treated as indivisible units which prevents exploiting compile-time knowledge about regular array subscript expressions to optimize programs.

Gupta, Schonberg, and Srinivasan [11] describe a data flow and interval analysis to place SENDs as early and RECVs as late as possible including detecting of redundant communication. SENDs and RECVs are not guaranteed to be balanced. Buffer constraints are not considered. A main advantage of their approach is that data flow analysis is performed at the granularity of array sections.

Sethi and Kennedy [18, 17] modeled pipelined communication and buffer requirements in a unified data flow and interval analysis framework. Explicit data flow equations are needed to ensure buffer-safe and balanced communication placement. Loops are explicitly modeled through interval analysis. If a program node exceeds the buffer available then all communication is blocked at that node.

Chakrabarti, Gupta and Choi [4] described a communication analysis based on static-single-assignment form, dependence analysis and available section descriptors. This approach tries to maximize reducing the number of messages exchanged by later placement of communication. They do not address the issue of buffer-safe communication placement. A variety of communication placements are generated. The final placements is determined by placing a communication in that program node where it can be coalesced with the largest number of other communication candidates. No other cost functions such as number of messages, amount of data transferred or communication time are incorporated to determine the best out of several communication placements.

6 Conclusion

We have presented a novel approach to optimize communication based on data flow analysis. Our framework places SENDs as early and RECVs as late as possible even across loop nests in order to optimize overlapping of communication with computation and vectorizing communication. P^3T, a state-of-the-art performance estimator, is crucial to support the generation of buffer-safe programs. By accurately estimating both the communication buffer sizes required and the implied communication times of every single communication of a program, we can selectively choose communication that must be delayed in order to ensure a correct communication placement while maximizing communication latency hiding. We employ a novel message coalescing algorithm which is used to aggressively detect and eliminate communication redundancy both in terms of number and volume of messages.

An important feature of our approach is that the data flow system does not require explicit modeling of balanced communication placement and loops. Instead of interval analysis we employ the generic fixed point algorithm in order to solve the data flow system. The described approach is based on uni-directional bit-vector data flow analyses that are less complex as their bi-directional counterparts.

Furthermore, we use advanced symbolic analysis techniques to effectively support analysis for linear and non-linear data dependences, array sections and communication patterns with program unknowns.

Preliminary performance results show the efficacy of our communication optimization strategy and confirms overall improvements in performance.

Most existing communication optimization approaches including the one presented in this paper are based on a fixed strategy whose quality is very sensitive to the changes of machine and problem sizes. We are currently exploring a system that creates (without actually generating code) and examines a reasonable set of promising communication placements covering several possibly conflicting communication guiding profit motives. We explore opportunities to use P^3T in the process of finding the best communication placement of all generated ones.

Future work will also involve including additional performance parameters, in particular memory and cache locality cost functions which are part of P^3T, in order to further investigate the impact of various communication optimization strategies in the overall performance of a parallel program.

References

[1] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proc. ACM SIGPLAN'93 Conf. on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[2] P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers. In *Proceedings of the First International Workshop on Parallel Processing*, pages 322–330, Bangalore, India, December 1994.

[3] S. Benkner, S. Andel, R. Blasko, P. Brezany, A. Celic, B. Chapman, M. Egg, T. Fahringer, J. Hulman, Y. Hou, E. Kelc, E. Mehofer, H. Moritsch, M. Paul, K. Sanjari, V. Sipkova, B. Velkov, B. Wender, and H.P. Zima. *Vienna Fortran Compilation System - Version 2.0 - User's Guide*, October 1995.

[4] S. Chakrabarti, M. Gupta, and J.-D. Choi. Global Communication Analysis and Optimization. In *ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, May 1996.

[5] T. Fahringer. *Automatic Performance Prediction of Parallel Programs.* Kluwer Academic Publishers, Boston, USA, ISBN 0-7923-9708-8, March 1996.

[6] T. Fahringer. Automatic Estimation of Communication Costs for Data Parallel Programs. *Journal of Parallel and Distributed Computing, Academic Press*, 39(1):46–65, Nov. 1996.

[7] T. Fahringer. Toward Symbolic Performance Prediction of Parallel Programs. In *IEEE Proc. of the 1996 International Parallel Processing Symposium*, pages 474–478, Honolulu, Hawaii, April 15 - 19, 1996.

[8] T. Fahringer. Symbolic Expression Evaluation to Support Parallelizing Compilers. In *IEEE Proc. of the 5th Euromicro Workshop on Parallel and Distributed Processing, London, UK*, pages 22–24, January 1997.

[9] T. Fahringer and B. Scholz. Symbolic Evaluation for Parallelizing Compilers. In *Proc. of the 11th ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[10] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, K. Wang, D. Shields, W.-M. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Proc. Supercomputing '95*, San Diego, CA, December 1995.

[11] M. Gupta, E. Schonberg, and H. Srinivasan. A unified framework for optimizing communication in data-parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, pages 7(7):689–704, July 1996.

[12] Hanxleden, R.v. and Kennedy, K. Give-n-Take–a balanced code placement framework. . In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, Orlando, FL, June, 20-24 1994.

[13] M. S. Hecht. *Flow Analysis of Computer Programs* . Elsevier, North-Holland, 1977.

[14] High Performance FORTRAN Language Specification. Technical Report, Version 1.0, Rice University, Houston, TX, May 1993.

[15] S. Hiranandani, K. Kennedy, and C.W. Tseng. Evaluation of Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines. In *ACM International Conference on Supercomputing 1992*, pages 1–14, Washington D.C., July 1992.

[16] J. Li and M. Chen. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, pages 2(3):361–376, July 1991.

[17] K. Kennedy and A. Sethi. A Communication Placement Framework with Unified Dependence and Data-flow Analysis . In *3rd International Conference on High Performance Computing*, Trivandrum, India, December 1996.

[18] K. Kennedy and A. Sethi. Resource-Based Communication Placement Analysis . In *Proc. of the 9th Workshop on Language and Compilers for Parallel Computing*, San Jose, CA, August 1996.

[19] J. Knoop, O. Rüthing, and B. Steffen. The Power of Assignment Motion. In *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, La Jola, CA, June 18 - 21 1995.