

Buffer-Safe and Cost-Driven Communication Optimization *

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

published in: Journal of Parallel and Distributed Computing, Academic Press, 57(1), pp. 33-63, April, 1999.

Abstract

This paper presents a new approach for optimizing communication of data parallel programs. Our techniques are based on unidirectional bit-vector data flow analyses that enable vectorizing, coalescing and aggregating communication, and overlapping communication with computation both within and across loop nests.

Previous techniques are based on fixed communication optimization strategies whose quality is very sensitive to changes of machine and problem sizes. Our algorithm is novel in that we carefully examine tradeoffs between enhancing communication latency hiding and reducing the number and volume of messages by systematically evaluating a reasonable set of promising communication placements for a given program covering several (possibly conflicting) communication guiding profit motives.

We use P^3T , a state-of-the-art performance estimator, to ensure communication buffer-safety and to find the best communication placement of all created ones. Employing an accurate performance estimator opens ground for more aggressive communication optimization opportunities that carefully examine performance gains and tradeoffs among applicable optimization strategies which is not achievable with any existing approach.

First results show that our method implies a significant reduction in communication costs and demonstrate the effectiveness of this analysis in improving the performance of programs.

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 [5, 17, 3, 21, 16, 23, 2, 29, 30] which includes: message vectorization (hoisting communication outside of loops), message coalescing (removing redundant communication based on the same array), communication aggregation (combine messages based on different arrays), 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 [17, 24, 23, 18] for global scheduling of communication commonly rely on data flow analysis which is used to place SENDs as early and RECVs as late as possible in order to maximize communication

* This research is partially supported by the Austrian Science Fund as part of Aurora Project "Tools" under contract SFBF1104.

latency hiding. Hardly any communication optimization considers communication buffer constraints, although it has been shown [24] that latency hiding can dramatically increase the size of communication buffers which can even result in run-time errors.

Recently it has been observed [5] that maximizing communication latency hiding can prevent valuable opportunities to reduce the number of messages and to eliminate partial redundancy. An alternative strategy has been developed in [5] which tries to schedule communication later such that as many individual communications as possible are combined at the same program point – not necessarily at the earliest possible point – in order to preserve the benefits of reducing the number and volume of messages at the cost of decreased latency hiding.

The fundamental problem that arises in most existing communication optimization approaches is that they are based on a fixed communication optimization strategy whose quality can be very sensitive to changes of machine and problem sizes. Ignoring tradeoffs between enhancing communication latency hiding and reducing the number and volume of messages may result in considerable performance slowdowns. Furthermore, many cost models, that are employed to optimize communication, lack estimation accuracy and portability to other architectures.

This paper describes a novel communication optimization framework that is based on a global unidirectional bit-vector data flow analysis and performance prediction. The approach presented differs significantly from previous work in that our communication optimization strategy is not fixed, but conflicting communication profit motives are evaluated carefully by an effective performance estimator to choose the best one to apply. We systematically create (without actually generating code) and examine a reasonable set of promising communication placements for a given program covering several (possibly conflicting) communication guiding profit motives including latency hiding and reducing the number and volume of messages. Our strategy is open for including additional communication placements of interest.

We aggressively apply communication coalescing and aggregation of messages. All communication placements are buffer-safe and balanced. In order to make a program buffer-safe, we selectively block 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.

P^3T [7, 9, 8], a state-of-the-art performance estimator for distributed memory parallel programs, is used to find the best communication placement of all created ones. Employing an accurate performance estimator opens ground for more aggressive optimization opportunities that carefully examine various performance tradeoffs which is not achievable with any previous approach.

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 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 [12, 14, 11, 10].

Preliminary performance results based on a pre-prototype implementation of our method demonstrate that the quality of a fixed communication optimization strategy may change for different problem and machine sizes. Our method shows significant reduction in communication costs and overall improvements in performance as compared to previous fixed communication strategies.

The organization of the 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. In Section 4 we illustrate how to create a variety of promising communication placements for reducing the number and volume of messages and for hiding communication latency. Furthermore, our cost functions are presented that are used to choose the best communication placement created. Preliminary results that demonstrate the benefits of the proposed method are presented in Section 5. Related work is discussed in Section 6. Section 7 gives our conclusions. Several proofs are presented in Section 8.

2 Preliminaries

2.1 Program Model

The control flow of a program is represented by a graph (CFG) $G = (N, E, e, x)$ 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 nodes $m, n \in N$. For the ease of presentation, data flow analysis is employed at instruction level. In fact, our data flow analysis can be straightforwardly modified to work on basic blocks. Nodes e and x respectively denote the unique *entry* and *exit* node of G , which are assumed not to possess any predecessors and successors. $\text{succs}(n)$ and $\text{preds}(n)$ correspond to the set of successor and predecessor nodes of n . A path in G is a sequence of nodes $[n_1, \dots, n_k]$ such that $\forall i, 1 \leq i < k: n_{i+1} \in \text{succs}(n_i)$. A path $]n_1, \dots, n_k[$ is equivalent to $]n_2, \dots, n_{k-1}[$ where $k \geq 3$. Every node $n \in N$ is assumed to lie on a path from e to x . We say a node n dominates a node n' , denoted by $\text{dom}(n, n')$, if n appears on every path from the entry node e to n' . An instruction may *write* or *use* data references (array or scalar variables).

Our communication optimization is performed as part of a parallelization strategy that is based on domain decomposition in conjunction with the Single-Program-Multiple-Data (SPMD) programming model [4, 30]. A program is executed by a set of processors P . Each data element is *owned* by one or more processors. Non-local data must be fetched before it is used by a 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. Our strategy can be easily extended to non-local writes. Non-local writes occur if a processor writes data that is owned by another processor. After the write operation this data must be transferred back to the owning processor. Note that our communication optimization strategy is not restricted to SPMD or a specific programming language. Large portions of our analysis are portable to other programming models and languages with little effort.

A SEND represents the sending component which initiates communication, whereas a RECV finalizes the communication. A SEND is implemented as a non-blocking *receive* operation immediately followed by a *send* operation. A RECV is implemented by a blocking wait. 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 as described in Section 3.1. The third case may occur if a specific SEND is associated with 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 *cover* a non-local use u if s has initially been inserted to exchange data which is needed due to u . Every s covers exactly one u which is referred to as $\text{OrigUse}(s)$. Furthermore, we define a SEND s to be *associated* with a non-local use u if part or all of u is being sent by s . Note that every non-local use u that is covered by s is also associated with s , whereas not every non-local use u that is associated with s is also covered by s . This is because a SEND s may be associated with u due to message coalescing although s is not directly implied by u . $\text{Sends}(u)$ is the set of SENDs that cover $u \in U$. $\text{Uses}(s)$ defines the set of non-local uses that are associated with a specific SEND $s \in S$.

2.2 Performance prediction

In order to support eliminating communication buffer conflicts and finding the best out of a variety of communication placements we use P^3T [7, 8, 6, 9], an accurate and effective performance estimation tool for distributed memory parallel programs. P^3T is a static performance estimator that analytically estimates the performance of data parallel programs (subset of Vienna Fortran [31], High Performance Fortran [20], Fortran90 and Fortran77) at compile-time without using simulation. P^3T has been developed to guide the selection of efficient data distribution strategies and profitable code transformations. A variety of perfor-

mance parameters each reflecting a different performance aspect are estimated: work distribution, number of transfers (messages exchanged), data volume transmitted, network contention [7], 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:

- *Work Distribution* [9] estimates how evenly the computations of a parallel program are distributed across all processors executing the program. This parameter accurately models data distribution strategies, data access patterns (array index function), and control flow information (statement execution and loop iteration counts).
- *Amount of Data Transferred* [8] is an estimate of the number of data elements transferred. In order to obtain results with good accuracy, this parameter among others models data distribution strategies, data access patterns, control flow, and machine specific data type information.
- *Communication Time* [8] is a sensitive 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 is estimated by the maximum communication time across all processors involved in the communication.
- *Computation Time* [7] is an estimate of the time it takes to execute the computations of a parallel program. In order to estimate computation times we pre-measure a large set of kernels which range from primitive operations (e.g. assignment and addition operations) to entire code patterns (e.g. matrix multiply). This parameter does not account for communication and blocking time. The kernels are measured on different architectures for varying problem sizes. The measured kernel runtimes are stored in a kernel library. In order to estimate computation times, a program is parsed to detect existing library kernels incorporating pattern matching techniques. For each kernel discovered, the pre-measured runtime is accumulated, which finally yields the overall computation time.

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 statement execution and loop iteration counts are derived by a single profile run [7] based on the original sequential program. We have shown [7] that large portions of the profile data can be automatically adapted for many important program changes without redoing the profile run.

P^3T 's performance parameters are designed as machine independent as possible. However, in order to build an accurate performance estimator we model some of the most important machine specific factors including cache line size, overall number of cache lines available, data type sizes, routing policy, startup times, distance overhead, and message transfer time per byte of the target architecture. Much of this information can be easily adapted for a variety of different architectures. Currently, P^3T predicts the performance for programs that run on the iPSC/860 hypercube, Intel Paragon, network of workstations and Meiko CS-2.

The original P^3T was restricted to communication based on overlap areas [4] 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. For detailed description of P^3T , the reader may refer to [7, 8, 6, 9].

3 Buffer-Safe Communication Latency Hiding and Message Coalescing

In this section we describe our communication optimization strategy. First, we hoist SENDs to the earliest possible program points without considering communication buffer constraints. Second, we aggressively coalesce SENDs in order to reduce the number and volume of messages. Third, we place RECVs as late as possible. Finally, we present our approach of placing SENDs/RECVs considering buffer constraints while optimizing communication.

Hoistability Analysis: ¹

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

Inserting SENDs as early as possible: ^{1,2}

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

Key:

¹ Boolean conjunctions are denoted by $*$ and Π , Boolean disjunctions by $+$ and Σ , and Boolean negation is denoted by a bar (e.g. \overline{A}).

² N-HOIST* and X-HOIST* denote the solutions of the equation systems for hoistability.

Figure 1: Data flow equations for placing SENDs

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 points, 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 upwards in the opposite direction of the control flow. If s is hoisted through a join node n (a node with two or more 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 .

The analysis of hoisting SENDs is based on a backward directed bit-vector data flow analysis [19] 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 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 .

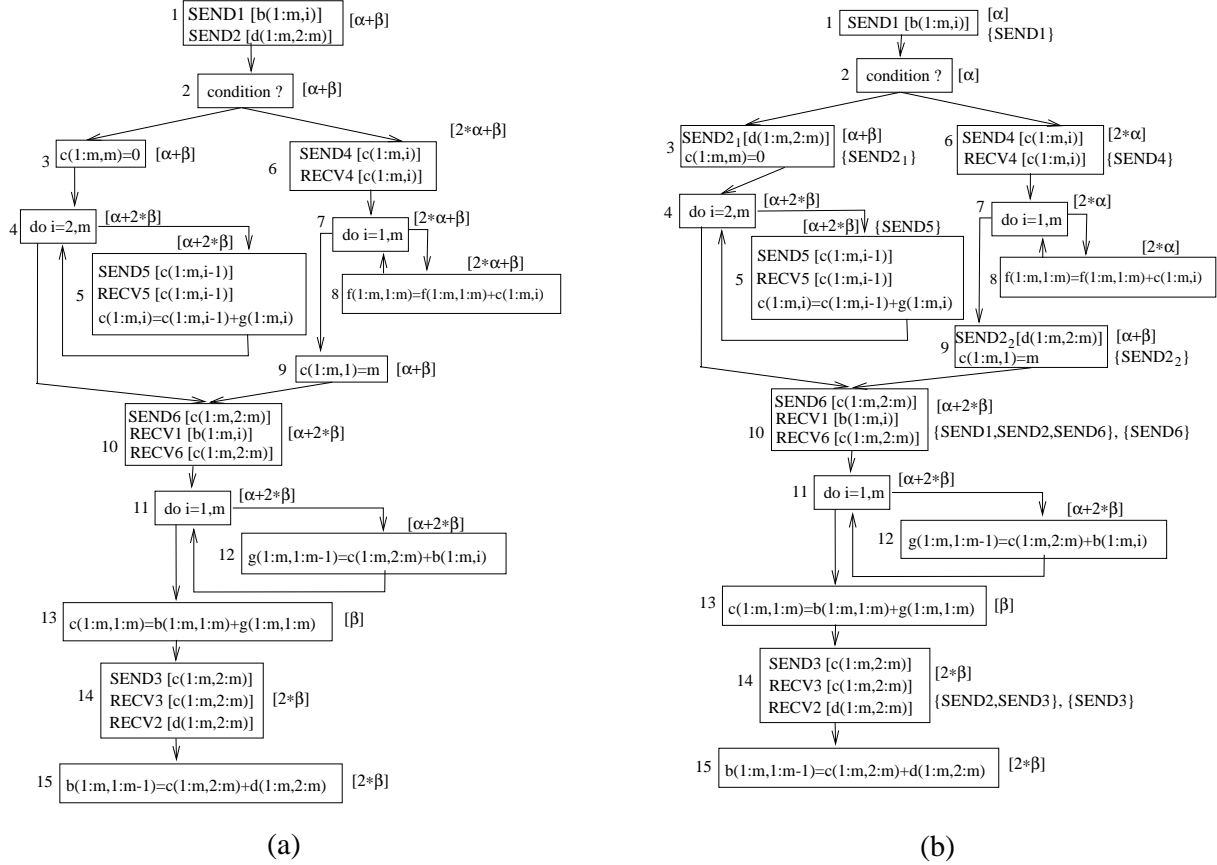


Figure 2: Running code example (SENDS/RECVs specify the non-local uses that cause communication). (a) earliest SEND placement without buffer constraints, (b) earliest SEND placement with buffer constraints

The data flow equations for hoisting SENDs are shown in Figure 1 which have been partially adapted from [26] 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 splitting edges through inserting synthetic nodes [26].

Equations (1) - (2) of Figure 1 present the hoistability analysis in bit-vector format, where each bit corresponds to a SEND that covers a non-local use u occurring in the program. Here $N\text{-HOIST}_n$ and $X\text{-HOIST}_n$ mean that a SEND can be placed at the entry or the exit of a node n , respectively. In principle, a SEND s can be hoisted to the entry of a node n denoted by $N\text{-HOIST}_n$ if s is generated in n or s can be hoisted to the exit of n and s is not blocked in n . A node n blocks s if n implies a true dependence affecting the communication data of s . A SEND s can be hoisted to the exit of a node n if s can be hoisted to the entries of all successor nodes of n . $X\text{-HOIST}_n$ is initialized to false at the exit node. Traditionally, a SEND can only be placed at a node n if the transferred data is used along all terminating paths starting at n . Therefore, hoisting SENDs out of zero-trip loops is not considered semantic-preserving since a communication which is not invoked originally is done in case of zero iterations. This strictly semantic-preserving strategy,

however, would imply critical communication overheads in many cases [7, 8] 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 the term $N\text{-HOIST}_{b(n)} * \overline{\text{BLOCK}_n}$ which allows hoisting communication outside of loops that do not carry (or contain a loop that carries) a true dependence affecting this communication. If n is a loop header, then $b(n)$ denotes the first node of the loop body of n and $N\text{-HOIST}_{b(n)}$ the corresponding predicate value, otherwise $N\text{-HOIST}_{b(n)}$ is false and $b(n)$ undefined. This policy 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 $N\text{-HOIST}_n^*$ and $X\text{-HOIST}_n^*$ – indicate nodes where SENDs can be legally inserted. $N\text{-EARLIEST}_n/X\text{-EARLIEST}_n$ (Equations (3) - (4)) denote the earliest legal positions where SENDs can be placed.

Figures 2 (a)-(b) show our running code example where all arrays (dimension $m \times m$) are column-wise distributed onto a set of P processors. 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 loop 11 and SEND4 out of loop 7 due to the term $N\text{-HOIST}_{b(n)} * \overline{\text{BLOCK}_n}$ of Eq. (2). SEND5 cannot be hoisted out of loop 4 which carries a true dependence that affects the communication data of SEND5.

Throughout the paper we will use Fortran 90 style of describing array accesses. Furthermore, for the ease of presentation, in all code fragments that we present, SEND and RECV statements are annotated with the non-local uses that cause 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 clarity, 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

Once the SENDs of a program have been placed as early as possible we coalesce messages in order to reduce the number and volume of messages that are based on the same array. This is in contrast to message aggregation which aims at reducing the number of messages sent by combining messages that are based on different arrays.

Figure 3 shows our algorithm for message coalescing. During hoistability analysis a SEND s can be hoisted through a join node which results in multiple copies of s . Every copy of s results in a unique entry in $Sends(u)$. $nodeS(s)$ and $nodeS(s')$ respectively specify the node at which s and s' are placed. $dom(nodeS(s), nodeS(s'))$ indicates that $nodeS(s)$ dominates $nodeS(s')$. $subsume(s, s')$ 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 . $LiveCommD(s, s')$ refers to the communication data of s' that is not written between s and s' .

Our algorithm for message coalescing involves two phases. In phase one, every SEND s is examined whether $nodeS(s)$ dominates $nodeS(s')$ of some SEND s' , and whether s subsumes s' such that s' covers a non-local use $u' \in U$ with $s \notin Sends(u')$. The communication data of s' that is not written between s and s' can be eliminated from s' as it is sent by s . If s' is partially redundant due to 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 $Sends(u')$, the set of SENDs that cover u' . This is because s is associated with u' but s does not cover u' according to Section 2.1.

In phase two, we determine the communication data for all SENDs of a non-local use u . Every SEND s of a specific non-local use u may have been made partially redundant by some other SEND and, therefore, different $s \in Sends(u)$ may imply different communication data according to phase 1. As a consequence, we may have to generate a specific RECV for every $s \in Sends(u)$. In order to alleviate code generation every

```

FOR every  $s \in S$ 
  FOR every  $s' \in S$  with  $s' \in Sends(u')$  and  $s \notin Sends(u')$  DO
    IF  $\text{dom}(\text{nodeS}(s), \text{nodeS}(s'))$  and  $\text{subsume}(s, s')$  THEN
       $\text{CommD}(s') := \text{CommD}(s') - \text{LiveCommD}(s, s')$ 
      IF  $\text{CommD}(s') = \phi$  THEN
         $Sends(u') := Sends(u') - s'$ 
         $S := S - s'$ 
      ENDIF
      IF  $\text{LiveCommD}(s, s') \neq \phi$  THEN
         $Uses(s) := Uses(s) \cup Uses(s')$ 
      ENDIF
    ENDIF
  ENDFOR
ENDFOR
FOR every  $u \in U$  DO
  FOR every  $s \in Sends(u)$  DO
     $\text{CommD}(s) := \bigcup_{s' \in Sends(u)} \text{CommD}(s')$ 
  ENDFOR
ENDFOR

```

Figure 3: Message coalescing

$s \in Sends(u)$ is sending the union of all communication data across all SENDs of $Sends(u)$. This enables generating the same code for every $s \in Sends(u)$.

Figures 4 (a)-(b) show a code example before and after message coalescing, respectively. All arrays (dimension $m \times m$) are column-wise distributed. SEND1/RECV1 and SEND2/RECV2 cover the non-local uses in node 2, and SEND3/RECV3 with the use in node 7. Note that SEND4₁ and SEND4₂ – respectively placed at node 5 and 6 by the algorithm described in Section 3.1 – belong to the same RECV4 and cover the use in node 9. The write access to b in node 4 blocks SEND4₁ at node 5 resulting in $\text{LiveCommD}(\text{SEND2}, \text{SEND4}_1) = \phi$. As a consequence SEND2 does not have an impact on SEND4₁. Whereas SEND4₂ is made redundant by SEND2 ($\text{LiveCommD}(\text{SEND2}, \text{SEND4}_2) = \text{CommD}(\text{SEND4}_2)$). Note that after SEND4₂ at node 6 has been eliminated, the associated RECV4 is placed in node 5 (see placement of RECVs in Section 3.3).

SEND3 cannot be hoisted to node 3 since it is only available in the right branch. According to the algorithm of Figure 3, $\text{nodeS}(\text{SEND1})$ dominates $\text{nodeS}(\text{SEND3})$ and SEND1 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 ($\text{LiveCommD}(\text{SEND1}, \text{SEND3}) = \text{CommD}(\text{SEND3})$). Message coalescing eliminates SEND3 and also the need for inserting RECV3. Note that SEND1 does not subsume SEND2. SEND1 implies a message exchange between neighboring processors, whereas SEND2 requires the processor that owns the m -th column to send this column to all other processors.

3.3 Latest RECV Placement

In order to maximize latency hiding we have to sink RECVs as far as possible. The analysis starts from the set of SENDs. Every SEND s implies a RECV r which is moved in the direction of the control flow to the latest possible program point before the communication data is used. If r is moved below a branch node n , then r is moved into every branch that starts at n which results in multiple copies of r .

The delayability analysis is based on a forward directed bit-vector data flow analysis which uses the following local predicates that are defined for every CFG node $n \in N$:

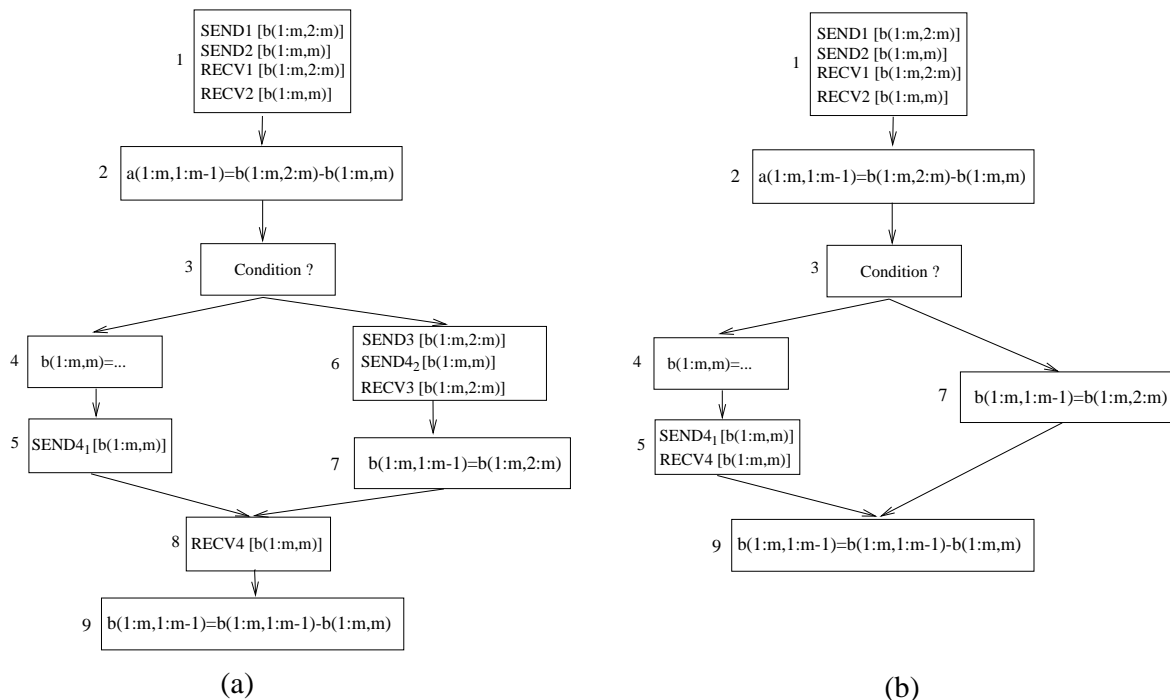


Figure 4: (a) SEND/RECV placement without message coalescing, (b) SEND/RECV placement with message coalescing

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 can be placed at the entry or at the exit of a node n , respectively. N-DELAY $_n$ is initialized to false for the entry node. Equation (5) ensures that a RECV can only be inserted at the entry of n if it is available at the exit of all predecessor nodes.

Equations (7)-(8) specify the insertion points of RECVs. A RECV r is placed at the entry of a node n if r is blocked in n . r is placed at the exit of a node n if there exists at least one successor node m of n which prevents further sinking of r . N-DELAY $_n^*$ and X-DELAY $_n^*$ denote the greatest solution of the delayability system.

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 r for a use u 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 approach implicitly guarantees balanced placement of SENDs with their associated RECVs (see proof in Section 8) which means that all associated SENDs and RECVs are placed at the same loop nesting level.

3.4 Earliest SEND Placement Considering Communication Buffer Constraints

Commonly every communication is associated with a communication buffer. It has been shown [24] that the size of such buffers – depending on the application – can be significantly larger than those required for local data sections. Hoisting communication to the earliest possible point in a program increases the life-time of

Delayability Analysis: ¹

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

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

Inserting RECVs as late as possible: ^{1,2}

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

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

Key:

¹ Boolean conjunctions are denoted by $*$ and Π , Boolean disjunctions by $+$ and Σ , and Boolean negation is denoted by a bar (e.g. \overline{A}).

² N-DELAY^* and X-DELAY^* denote the solutions of the equation systems for hoistability.

Figure 5: Data flow equations for placing RECVs

communication buffers and, therefore, can cause a dramatic increase of buffer requirements for non-local data. Increased buffer requirements can lead to serious memory problems during runtime or even while loading a program onto a target architecture [24].

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 implements a SEND as a non-blocking *receive* immediately followed by a *send* operation. Every associated RECV is replaced by a blocking *wait*. Note that commonly a RECV – implemented as a non-blocking receive followed by a blocking wait – completes only after the communication data arrived which is ensured by the blocking wait operation. Hence, the communication buffer of a SEND s remains *live* starting at s until its last use.

Let $\text{buffer}(s)$ denote the buffer size required by a SEND s , and $\text{buffer-req}(n)$ the sum of buffer sizes required by a node $n \in N$ according to the buffers that are live at n . $\text{buffer}(s)$ is computed by P^3T 's parameter for the *amount of data transferred* as described in Section 2.2. $\text{buffer}(s)$ is defined as the maximum amount of data transferred across all processors involved in s . If s and s' have been coalesced based on our algorithm of Figure 3, then s and s' use the same communication buffer and their required buffer size is set to the size of this communication buffer. Note that although message coalescing commonly reduces memory requirements, it may also happen that message coalescing increases the buffer requirements at certain program nodes and

also the life-time of some communication buffers.

We traverse the CFG and for every node $n \in N$ and for every SEND $s \in S$ we add $buffer(s)$ to $buffer-req(n)$ iff the communication buffer of s is live at n . $buffer-req(n)$ is increased only once for all SENDs that share a communication buffer due to message coalescing.

Consider our example of Figure 2 (a) where for every node n the corresponding $buffer-req(n)$ is given in square brackets. Note that all arrays are assumed to be of dimension $m \times m$ and are column-wise distributed onto a set of P processors. $buffer(SEND1) = buffer(SEND4) = \alpha$ implies a broadcast of the local segments of arrays b and c . 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.

```

solve hoistability analysis for U
coalesce SENDs for S
REPEAT
  stable := true
  H :=  $\phi$ 
  FOR every  $n \in N$  in reverse depth first order DO
    IF  $LiveSends(n) \cap H = \phi$  THEN
      IF buffer requirements of all non-local uses in n exceed max-buffer THEN
        /* Error: Non-local uses of node n require more communication buffer space than available */
        EXIT ("Insufficient Communication Buffer Size")
      ELSE
        WHILE  $buffer-req(n) > max-buffer$  DO
          stable := false
          determine s  $\in LiveSends(n)$  with smallest communication time and  $nodeU(OrigUse(s)) \neq n$ 
          H := H  $\cup$  s
           $buffer-req(n) := buffer-req(n) - buffer(s)$ 
           $LiveSends(n) = LiveSends(n) - s$ 
          FOR every  $s'$  that shares the communication buffer with  $s$  DO
             $LiveSends(n) = LiveSends(n) - s'$ 
          ENDFOR
          Set BLOCKn to true with respect to s
          FOR every loop header node  $h$  whose loop body contains  $n$  DO
            Set BLOCKh to true with respect to s
          ENDFOR
        ENDWHILE
      ENDIF
    ENDIF
  ENDFOR
  IF stable = false THEN
    solve hoistability analysis for all uses u where  $u \in Uses(s)$  and  $s \in H$ 
  ENDIF
UNTIL stable = true
solve delayability analysis for S

```

Figure 6: Buffer-safe latency hiding and message coalescing

3.4.2 Delay SENDs if communication buffer exceeds

Let $nodeU(u)$ specify the associated CFG node that contains the non-local use u and $max-buffer$ is the maximum size of the communication buffer available on every processor of the target architecture. Note that $max-buffer$ is a machine specific parameter whereas $buffer-req(n)$ is program specific. $buffer-req(n)$ is increased by the buffer sizes required for all SENDs that are live at n whereas $max-buffer$ commonly does not change for a program run on the target architecture. $LiveSends(n)$ is the set of SENDs whose associated buffers are live at node n . $OrigUse(s)$ defines the non-local use which is covered by s .

Figure 6 depicts our overall algorithm for buffer-safe communication optimization. We first place SENDs as early as possible without considering buffer constraints according to the hoistability analysis of Section 3.1. Coalescing SENDs may increase buffer requirements and the life-time of communication buffers. Therefore, message coalescing is done only once – outside of the REPEAT-loop. In every iteration of the REPEAT-loop we may selectively undo latency hiding and message coalescing by blocking SENDs in order to create a buffer-safe communication placement.

We traverse the entire CFG and examine every node n whether the buffer requirements of all non-local uses that appear in n can be satisfied by $max-buffer$. If not, then the algorithm terminates due to insufficient communication buffer size. An alternative to overcome this case (which goes beyond the scope of this paper) would be to split n such that the non-local uses are distributed onto several nodes (instructions).

If $max-buffer$ honors the buffer-requirements of all non-local uses in n , then we must verify whether the SENDs that are live at n imply a buffer conflict. As long as the buffer requirements of n exceed $max-buffer$ we have to block specific 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. Selecting a SEND for being blocked at a node n is critical in obtaining 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.

A SEND that has been blocked due to an exceeding communication buffer, may invalidate previous message coalescing and in particular the buffer requirements of the SENDs that have been involved in message coalescing. All such SENDs that are blocked due to buffer conflicts are collected in H . A blocked SEND s may no longer be associated with all its non-local uses in $Uses(s)$. Hence, hoistability analysis must be redone with updated predicate BLOCK for every non-local use $u \in Uses(s)$ with $s \in H$. Fortunately, hoistability analysis which is realized as a bit-vector problem can be done very efficiently. For reducible flow graphs the iterative approach will take on the average less than 5 iterations [27, 22].

The REPEAT-loop terminates if a node n is encountered whose non-local uses exceed the buffer capacity of n , or if all SENDs of the program are placed without buffer conflict. Finally, the RECVs are placed as late as possible according to the algorithm of Section 3.3. Note that the placement of RECVs does not influence the life-time of communication buffers. In Section 8, we prove that the algorithm displayed in Figure 6 terminates.

In continuation of our code example in Figure 2 (a), we assume that $max-buffer = 2 * \alpha$ and $\beta \leq \frac{\alpha}{2}$. Hence, SEND2 is blocked by our algorithm at node 7 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. Otherwise 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 replaced by two separate copies SEND2₁ and SEND2₂ at nodes 3 and 9, respectively. SEND2₂ is blocked at node 7 and, hence, hosting SEND2₂ stops at node 9. As a consequence, SEND2₁ cannot be hoisted beyond node 2 and is, therefore, placed at node 3 of the left branch.

At this stage we have a buffer-safe program where SENDs are placed as early and RECVs as late as possible. We refer to this program version as *ELB* (*Earliest SEND - Latest RECV - Buffer-safe*). The ELB

program version of our running example is shown in Figure 2 (b).

4 Communication Latency Hiding versus Reducing Number and Volume of Messages

Optimizing communication is faced with a critical tradeoff: On the one hand, placing SENDs as early and RECVs as late as possible tries to maximize communication latency hiding which may result in losing valuable opportunities to reduce the number and volume of messages. On the other hand, placing messages at program positions where they can be coalesced and aggregated with a maximum number of other messages may reduce the potential for communication latency hiding. Most existing compilers employ one of these two strategies but do not try to examine the performance gains and tradeoffs of both strategies in order to find the best communication placement for a given problem and machine size.

In the following we present a systematic approach that creates and examines a reasonable number of communication placements for a given program covering promising combinations of the strategies mentioned above. Moreover, an efficient cost model is employed in order to determine the best communication placement created.

4.1 Combining SENDs into Groups based on Maximum Matching SENDs

Construction of groups of SENDs is done in three phases. In the first phase, based on the ELB version we determine all possible placements of SENDs in a CFG, any one of which can be potentially chosen to exploit latency hiding and/or reducing number and volume of messages. Let $Elb(s)$ denote the set of earliest buffer-safe positions (CFG nodes) of s in ELB (earliest SEND - latest RECV - buffer-safe placement according to Section 3.4.2). For every $n \in N$ we build $NS(n)$, the set of SENDs that can be placed in n such that every SEND placement is buffer-safe. A SEND $s \in S$ is added to $NS(n)$ iff 1. $n \in Elb(s)$, or 2. every path from the entry node e to n contains a node $n' \in Elb(s)$ and n dominates all nodes at which any $u \in Uses(s)$ is located. If $NS(n) = \phi$ then there does not exist a SEND which is placed in n . Figures 7 (a) and (b) show the SEND placement of ELB of the running code example and the $NS(n)$ set for every CFG node, respectively. Note that all of the placements of s are buffer-safe, as the nodes at which s is placed are dominated by a node in $Elb(s)$ which is guaranteed to be buffer-safe.

In the second phase, we determine the maximum matches of SENDs in every $n \in N$. Let the sender-receiver relationship of a SEND s_1 be a subset of the sender-receiver relationship of a SEND s_2 iff for every message exchange of s_1 with a sending processor a and receiving processor b there exists a message exchange of s_2 with the same sender-receiver processors. Obviously, if the sender-receiver relationship of s_1 is a subset of the sender-receiver relationship of s_2 and vice versa, then s_1 and s_2 have an identical sender-receiver relationship. Based on the sender-receiver relationship we define that two SENDs s_1 and s_2 *match* with each other iff s_1 and s_2 have either an identical sender-receiver relationship or one is a subset of the other. Matching SENDs may be based on the same (message coalescing) and different arrays (message aggregation). Then, for every $n \in N$ we create a set of groups $Groups(n)$ based on $NS(n)$ such that for every $g \in Groups(n)$ the following holds: $g \subseteq NS(n)$ and all $s \in g$ match with each other. If a SEND can be put into several groups of a specific node n , then it is put into the group with the largest number of SENDs. A separate group is created for every SEND s (including SEND copies) – containing only s – that is put in every node $n \in Elb(s)$. Typically the cardinality of $NS(n)$ and $Groups(n)$ is rather small (cf. Figure 7) which makes the complexity of different grouping algorithms a secondary issue. The runtime impact of different grouping algorithms (for instance, grouping of SENDs based on the same arrays only), however, may be relevant and is left open for future investigation. Figure 7 (c) shows $Groups(n)$ in braces for the running example.

In the third phase, we traverse the CFG and delete all groups $g \in Groups(n)$ of a node $n \in N$ iff there exists a node $n' \in N$ with $n \neq n'$ and a group $g' \in Groups(n')$ such that n' dominates n and $g \subseteq g'$ (g' contains all

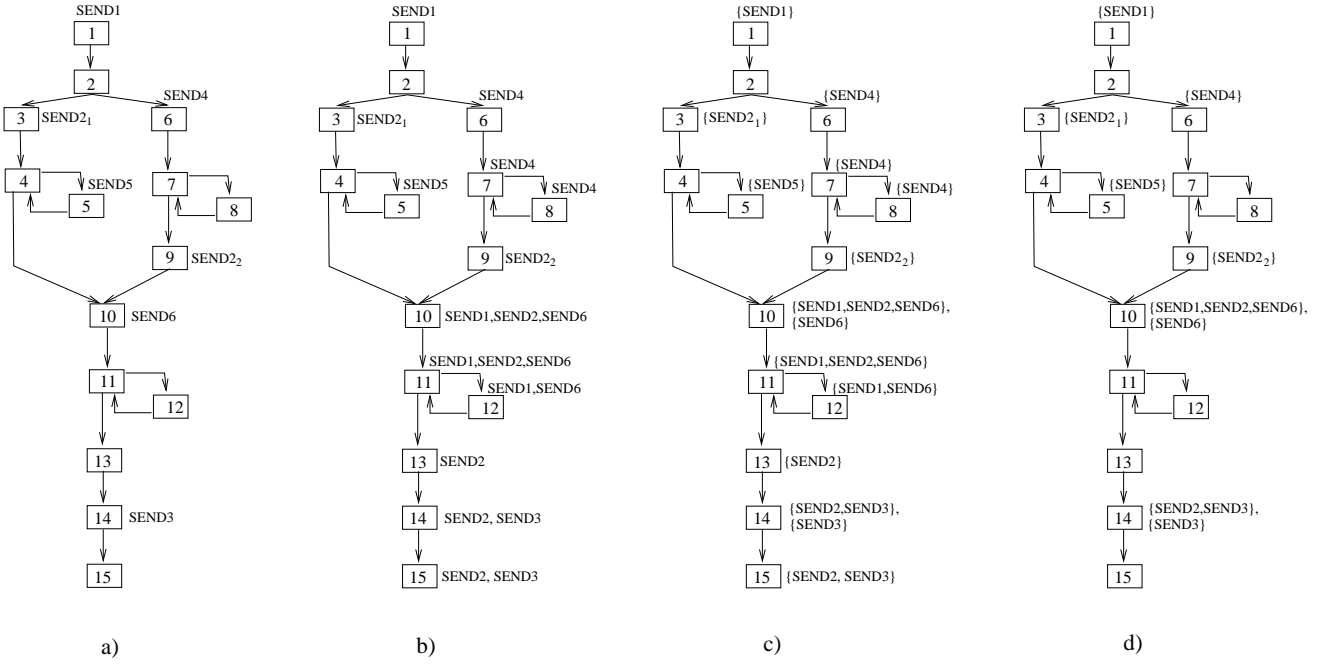


Figure 7: (a) ELB version, (b) NS(n) after phase 1, (c) Groups(n) after phase 2, (d) Groups(n) after phase 3.

SENDS that are included in g). In this way we further exploit latency hiding without changing the number and volume of messages. The resulting groups of the running example are shown in Figure 7 (d).

Note that our method is very flexible for adding additional SEND placements of special interest or change the policy for placing SENDs in groups. We commonly add the standard SEND placement where a SEND – associated with a non-local use u – is placed just before the outermost loop in which there is no true dependence on u , or just before the statement containing u if no such loop exists. Furthermore, the standard SEND placement applies message coalescing/aggregation to SENDs that are placed at the same program point.

4.2 Determine SEND Combinations and Place RECVs

In the previous Section we identified different possibilities to place SENDs. In this section we describe how to determine a variety of promising SEND placements for various communication optimizations including latency hiding and reducing the number and volume of messages.

Let $Groups_G$ be the set of all groups across all nodes of a CFG. We define a *valid SEND combination* C to be a subset of $Groups_G$ that contains every $s \in S$ with: If C includes a group with a copy of a SEND s then C must include one group for every copy of s . In all other cases s is included in exactly one group of C .

Let us illustrate the construction of valid SEND combinations with our running example. Starting point are the groups displayed in Figure 7 (d). Only SEND1 and SEND2 can be placed at different positions: SEND1 in nodes 1 and 10 and SEND2 in nodes 3/9 ($SEND2_1$ in node 3 and $SEND2_2$ in node 9), 10, and 14. SEND4, SEND5, and SEND6 have already their final positions. Hence, there exist at most 6 different SEND combinations. However, note that not all SEND combinations are valid. It is invalid to place SEND1 in node 1 and SEND2 in node 10, since the group containing SEND2 in node 10 also contains SEND1 and thus violates the definition that a SEND must be included in exactly one group. The valid SEND combinations are displayed in Figure 8: SEND1 in node 1 and SEND2 in nodes 3/9 ($C1$), SEND1 in node 1 and SEND2 in node 14 ($C2$), and both SEND1 and SEND2 in node 10 ($C3$). Note that $C4$ corresponds to the standard

SEND placement (see Section 4.1) which has been added to all other valid SEND combinations as created by our method. $C4$ focuses primarily on message aggregation but lacks communication latency hiding. The other extreme is given by $C1$ which largely ignores message aggregation but highly overlaps communication with computation. Whereas the remaining two valid SEND combinations $C2$ and $C3$ represent a compromise between the two extremes by combining latency hiding and message aggregation.

Note that a worst case scenario could result in an exponential number of valid SEND combinations for a given program, in practice, we never encountered more than $|S|^2$ (S is the set of SENDs of a program) valid SEND combinations for any program examined.

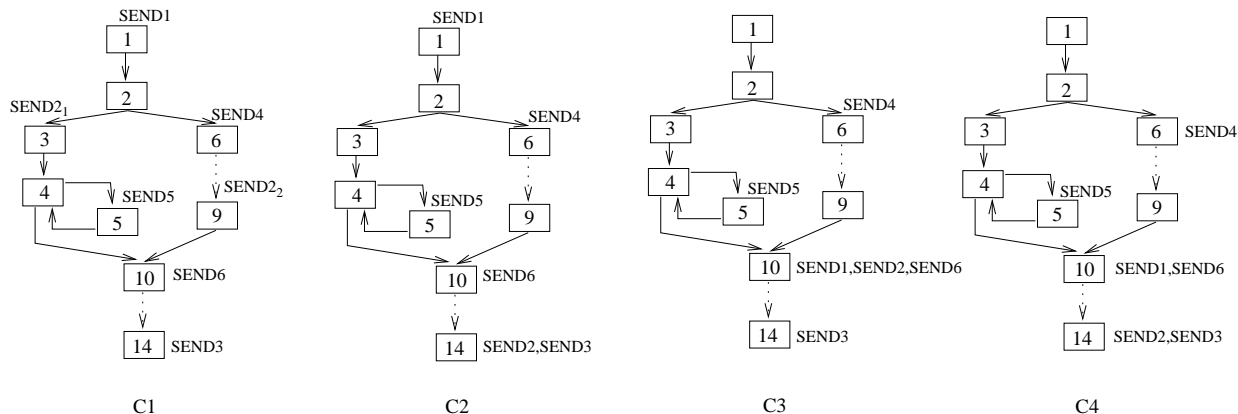


Figure 8: A variety of SEND combinations. (Irrelevant subgraphs are indicated by dotted arrows.)

Let VC specify the set of all valid SEND combinations of a CFG. In order to amortize message startup costs, we generate a single SEND (aggregate communication) for every group g as part of $C \in VC$. For every C we have to place the associated RECVs as late as possible. For this purpose we reuse our delayability analysis (see Figure 5) and apply it to groups. Let $Uses_G(g)$ specify the set of all non-local uses that are associated with the SENDs in g . If there is an aggregated SEND placed for g in n then $SEND-CAND_n$ is set to true, otherwise to false. If n contains a non-local use $u \in Uses_G(g)$, then USE_n is set to true, otherwise to false. The data flow equations – identical to those shown in Figure 5 – are now used to compute the insertion nodes of RECVs for all groups in any valid SEND combination. Aggregated SENDs and RECVs are again balanced.

4.3 Determine SEND Combination with the Best Communication Behavior

At this stage we have to determine the best $C \in VC$ of a given program. For this reason we will use an effective cost model combining both communication as well as computation times. The idea is to compute a cost function (see Figure 9) for every $C \in VC$ by determining how much communication overhead of C can be overlapped with the underlying program’s computations. A program is said to be *communication optimal* if all of its communication can be overlapped with computation. Note that there can be several communication optimal programs. The more communication that cannot be overlapped with computation, the worse the quality of a given valid SEND combination. We assume that packing and unpacking communication data is part of the communication time which reflects complex message aggregation that is specific for every different SEND combination.

Each group of a $C \in VC$ corresponds to a set of messages exchanged between processors of the target architecture and implies at a maximum one message exchange between any pair of processors, as all groups of C are implemented as aggregated communications. $CommTime(g)$, the communication time of a group g

$$CompTime_a(i) = \frac{CompTime_s(i) * freq(i)}{|P|} * (1 + WorkDist(i)) \quad (9)$$

$$Overlap(g) = \frac{1}{|Paths(g)|} * \sum_{w \in Paths(g)} (CompTime_{path}(w) - CommTime(g)) * Prob(w) \quad (10)$$

$$CommCost(C) = \sum_{g \in C} |\gamma(Overlap(g))| \quad (11)$$

$$\gamma(r) = \begin{cases} 0 & : \text{ if } r \geq 0 \\ r & : \text{ otherwise} \end{cases} \quad (12)$$

Figure 9: Computation and communication cost functions

is computed as the maximum communication time across all message exchanges implied by g . P^3T is used to compute $CommTime(g)$ (see Section 2.2).

Let $CompTime_s(i)$ specify the time required to execute a single instance of an instruction i . $CompTime_a(i)$ (see Equation (9) of Figure 9) is the time to execute all instances of i in a parallel program. $freq(i)$ is the statement execution count of i as determined by a single profile run of the original sequential program. P is the set of processors that execute the parallel program. $WorkDist(i)$ is the work distribution for i as determined by P^3T . We have presented a proof in [9] for the existence of a best and worst case work distribution ($0 \leq WorkDist(i) \leq |P| - 1$). If $WorkDist(i) = 0$ (best case work distribution) then the number of instances of i are perfectly load balanced across all processors in P . For increasing values of $WorkDist(i)$ it is assumed that the work distribution deteriorates. $CompTime_a(i) = CompTime_s(i) * freq(i)$ if $WorkDist(i) = |P| - 1$ (worst case work distribution) which means that every processor is executing all instances of i .

In order to determine the degree of overlapping communication with computation time for a group g , we must examine $Paths(g)$, the set of all possible paths (for loops only the loop header node is included) between the SEND of g and all of its associated RECVs. Note that a group g has a unique SEND but may have several associated RECVs and all SEND/RECV pairs are balanced. We use P^3T to estimate $CompTime_{path}(w)$, the computation time of a path $w \in Paths(g)$. The computation time of every single instruction on a path is computed according to Equation (9). If there are loops included in a path, then their computation time is separately computed and associated with the loop header node. Each branch must be separately considered and is weighted by its probability. The branching probability is computed based on statement execution counts. Loops that are fully included between a SEND and its associated RECV do not imply additional paths to be examined. Figure 10 shows the SEND and all its associated RECVs of a specific group $g \in Groups_G$. The edges of branches are marked with their probability of being taken during execution of the program. We must examine 3 different paths for g : $w_1 = [1, 2, 4, 5, 6]$, $w_2 = [1, 2, 4, 5, 7]$, and $w_3 = [1, 2, 4, 8]$. Note that none of the paths actually contains node 3 which is implicitly included in node 2. $Prob(w)$, the probability that a path w is actually executed, is computed by the product of all edge probabilities along this path. Therefore, $Prob(w_1) = 0.32$, $Prob(w_2) = 0.48$, and $Prob(w_3) = 0.2$.

$Overlap(g)$, the degree of overlapping computation with communication of a specific group g of C is then computed – according to Equation (10) – as the average latency hiding across all possible paths of g weighted by the paths’ probability. Equation (11) models $CommCost(C)$, the communication costs of a valid SEND combination C , which is the sum of all communication times that cannot be overlapped with computation.

P^3T selects a $C \in VC$ to be the best valid SEND combination if C has the smallest value of $CommCost(C)$ across all valid SEND combinations in VC . The complexity of computing P^3T parameters (cf. [7]) as part

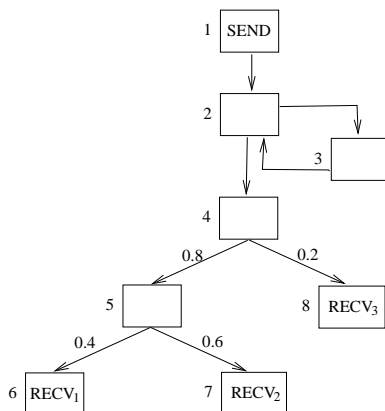


Figure 10: SEND/RECV paths of a group $g \in Groups_G$.

of the cost functions described in Figure 9 is independent of loop iteration and statement execution counts. Hence, computing the described cost functions is faster than simulating or actually compiling and executing the parallel program. Furthermore, frequently program statements imply identical P^3T performance parameter values which are stored in tables and retrieved via fast table look-ups. For instance, array assignment statements inside of loops often have the same data distributions and subscript expressions, and therefore, such statements can imply identical P^3T performance parameter values.

5 Experiments

We have implemented a pre-prototype of our communication optimization strategy as part of *VFCS* (Vienna Fortran Compilation System [4]), a compiler for distributed memory architectures. The cost models as described in this paper and the underlying symbolic analysis are fully implemented. Currently our communication optimization can handle codes with linear array subscript and loop bound expressions. Only block distributions of arrays are supported. Communication optimization cannot be done for arrays that are based on cyclic distribution. Furthermore, removing redundant communication is only done if a communication is made fully redundant by other communication. In the following we describe the results of our experiments performed to measure the potential benefits of our buffer-safe, cost driven communication optimization.

In the first experiment, we examine the performance of all four communication placements C_1, \dots, C_4 as shown in Figure 8 for various problem sizes. We have adapted *VFCS* such that it can generate code for every communication placement as created by our optimization. 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 11. In each bar-chart the x-axis specifies the problem size m . For each problem size 4 bars are plotted, one for each code version (C_1, \dots, C_4). The y-axis is normalized so that the code version with longest runtime has unit runtime 1.0. The dark and light 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 aggregation is, therefore, more critical than latency hiding for small problem sizes. C_3 and C_4 are those versions that highly focus on message aggregation. Both of them

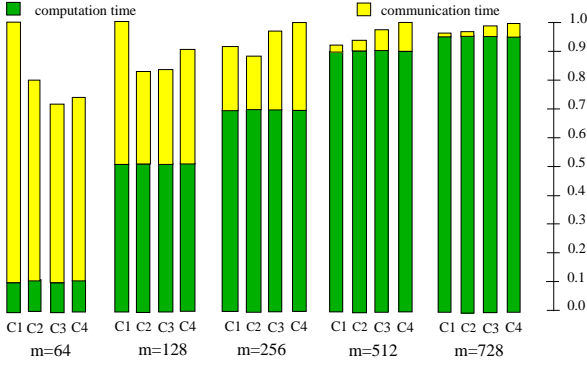


Figure 11: Normalized measured runtimes of code versions $C1, \dots, C4$ (see Figure 8 (d)) on a Meiko CS-2 with 16 processors

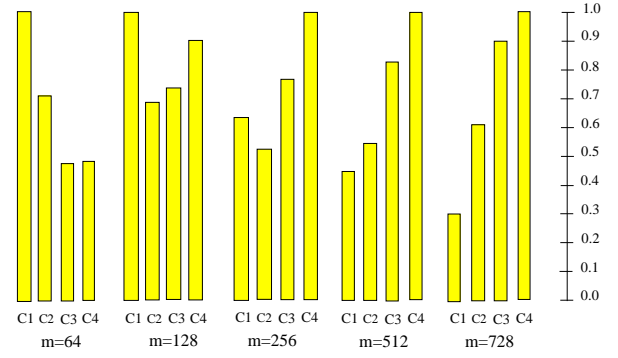


Figure 12: Normalized estimated communication cost function $CommCost$ for $C1, \dots, C4$ on a Meiko CS-2 with 16 processors

imply the same number of messages exchanged. In addition $C3$ exploits latency hiding for SEND2 which makes it slightly better than $C4$ for $m = 64$. For medium message sizes, $C2$ is superior to all other versions as it represents a good compromise of both message aggregation and latency hiding. For larger problem sizes the computational overhead increases rapidly which improves the opportunities to overlap computation with communication. Hence, large problem sizes cause $C4$ to perform very poorly due to its lack of latency hiding, whereas $C1$ – aggressively overlaps communication with computation – becomes superior to all other code versions.

Figure 12 plots our communication cost function $CommCost$ according to Equation (11) of Section 4.3 for the same set of problems sizes and code versions as shown in Figure 11. Again, the y-axis is normalized so that the code version with largest $CommCost$ figure has unit time 1.0. From this figure it can be seen that our estimates do not precisely reflect the real performance behavior. For instance, for $m = 64$ the communication time of $C3$ accounts for approximately 65 % of the communication time of $C1$ according to Figure 11, whereas in Figure 12 it is close to 50 %. The reason for this loss in estimation accuracy is due to the difficulty to accurately model computation times which impact $CommCost$. Currently our computation time cost function ($CompTime$) models only local memory hierarchy (in particular caches) and cpu-pipeline effects. It does not consider these effects globally – across different statements and loops. Nevertheless, our communication cost function delivers the same ranking of communication placements for every different problem size as shown by the measured runtimes.

The most important observation of this experiment is that for changing problem sizes different optimization strategies become the first order optimization effect. Using a fixed optimization strategy, therefore, can cause drastic performance losses for a specific problem size whereas for other problem sizes it might outperform all other optimizations. An efficient cost model is, therefore, required to determine the best communication optimization strategy for a given problem size or to find the best compromise across several optimizations. For the given experiment $C2$ seems to be the best compromise as it achieves the best average communication outcome across all problem sizes and communication placements considered. Note that $C2$ does not focus on a specific communication optimization strategy but is a combination of several strategies. $C2$ can only be obtained through systematic creation and cost evaluation of communication placements as described in this paper.

In the second experiment, we examine our communication optimization strategy for various machine sizes as applied to a Particle-in-Cell (PIC) code which determines the motion of a group of interacting particles starting with some initial configuration of positions and velocities in a specified volume of space. We imple-

mented a parallel PIC version (approximately 3500 lines of code) following a method described in [13] where both the spatial grid and the set of particles are regularly decomposed onto a set of processors. We compared 4 different PIC versions (procedures have been in-lined): C1: ELB (earliest SEND, latest RECV, and buffer-safe). C2: best code version found among those generated by our cost driven communication placement according to Section 4. C3: messages are placed at the program point where they can be aggregated with the maximum number of other messages. C4: communication is hoisted into outermost possible loop and messages are aggregated if they are placed at the same program point. The codes have been measured for 1024 particles on a network of Sun-10 and Sun-5 workstations connected via an ethernet (10 Mbits/sec bandwidth). C4 contains 17 different SENDs (excluding those required for host-node communication, and startup-synchronization). As the PIC code of our study is reasonably well structured our cost-driven communication placement strategy creates 12 (set of valid SEND combinations) different code versions including C1, C2, C3, and C4. Figure 13 shows the measurements. In each bar-chart the x-axis is the number of workstations used ranging from 2 to 16 workstations. The y-axis is normalized so that the code version with longest runtime has unit runtime 1.0.

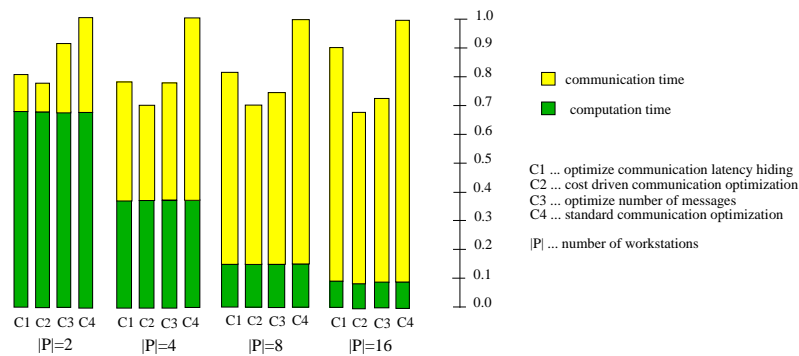


Figure 13: Normalized measured runtimes of four different PIC codes on a network of workstations

The dark and light 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 all code versions is fixed as the number of particles (1024) does not change. The communication overhead is the predominate factor of the overall runtime due to the slow network connecting the workstations and the number of messages exchanged increases linearly with the number of workstations involved. This explains also why the percentage of communication time as part of the overall runtime vastly increases for larger workstation numbers. A small number of workstations results in a significant computational overhead and fewer messages exchanged which makes latency hiding (C1) more effective than message aggregation (C3). However, message aggregation is critical for increasing number of workstations employed in order to reduce the high message startup overhead on workstation networks. C3 outperforms C1 for 8 and 16 workstations by 9 % and 19 %, respectively. C2, which is a combination of aggregating and hiding communication as found by our cost-driven method, is superior to all other code placements. C4 is inferior in terms of performance to all other versions examined as it includes only standard communication optimization.

The important observation of this experiment is as follows: C2 is not drastically faster than the second best version for every specific number of workstations evaluated. For instance, C2 respectively implies a reduction in communication by 2.5 % for $|P| = 2$ and 5.5 % for $|P| = 16$ as compared to C1 and C3. However, if compared to a fixed communication optimization policy (latency hiding or message aggregation) and depending on the workstation number evaluated, C2 is 2.5 % - 22 % and 5.5 % - 14 % faster than C1 (latency hiding)

and C3 (message aggregation), respectively. Hence, C2 is a sensitive communication placement – considering both communication latency hiding and message aggregation – that significantly outperforms every fixed communication placement examined in our study. Note also that C1 is not consistently better than C3 and vice versa for all workstation numbers evaluated. C2 has a reduced communication overhead ranging from 36 % - 70 % as compared to the standard communication placement C4.

TABLE I
Execution times (sec) of SHALLOW on a Meiko CS-2

N	P=1		P=4		P=8		P=16	
	No BCO	With BCO	No BCO	With BCO	No BCO	With BCO	No BCO	With BCO
64	0.04	0.041	0.013	0.011	0.008	0.0064	0.004	0.0027
128	0.19	0.191	0.059	0.053	0.031	0.028	0.017	0.0128
256	0.85	0.86	0.236	0.221	0.132	0.118	0.067	0.057
512	3.49	3.493	0.91	0.873	0.498	0.468	0.258	0.232
768	7.82	7.8	2.0	1.98	1.086	1.064	0.554	0.514

In the third experiment we present some performance results for SHALLOW which is a weather prediction code that uses finite-difference methods to solve a system of shallow-water equations. This code has been written by Paul Swarztrauber at the National Center for Atmospheric Research, Boulder, Colorado. In order to improve the performance of the original code we applied loop-distribution to all loops which enhances the cache performance and enables loop strip mining [4]. All arrays are distributed row-wise. After loop strip mining every processor executes only those loop iterations in which data is written that is owned by the processor. We compiled this code – excluding I/O – by using two versions of *VFCS*, one which does not apply the buffer-safe and cost-driven communication optimization, and the other one which does. Table I displays the execution times of SHALLOW for various problem (N) and machine sizes (P). Execution times are tabulated without applying buffer-safe and cost driven communication optimization (no *BCO*) and after applying this transformation (with *BCO*).

Although loop distribution improved the overall performance of the original code, it separates communication statements that have been coalesced before loop distribution. *VFCS* without *BCO* coalesces communication statements only if they are placed at the same program position. Whereas *VFCS* with *BCO* exploits the potential of redundant communication in SHALLOW by extensively coalescing and aggregating messages even across loops. Our analysis clearly has no impact on the single processor version as it does not imply any communication. The performance improvement on 16 processors varies from 32.5 % to 5 % for different problem sizes ranging from $N = 64$ to $N = 768$. The relative gain in performance is lower for larger problem sizes and for programs executed on fewer processors because of the computation time dominating the communication time. For small problem sizes we achieve a performance gain varying from 15 % to 32.5 % for different machine sizes ranging from 4 to 16 processors. Only modest performance improvement is achieved for larger problem sizes.

6 Related Work and Discussion

Communication optimization [4, 30, 5, 17, 3, 21, 16, 23, 2, 29] has been extensively researched. Most existing compilers employ a combination of message aggregation and coalescing and collective communication which is based on single-loop analysis. Pipelined communication [21, 16, 23, 2] has been introduced to achieve a fine-grain communication latency hiding. Communication placement based on data flow analysis has been addressed by several researchers both to eliminate redundant communication within a loop nest [2] and across loops [17, 24, 23, 18].

Von Hanxleden and Kennedy [18] 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 [17] describe a bidirectional data flow system together with interval analysis in order to place SENDs as early and RECVs as late as possible including detecting of redundant communication. Whereas bidirectional data flow systems are difficult to understand and to compute, our data flow system – based on unidirectional bit-vector equations – can be solved easily by interval or iterative analysis. Gupta et al. decompose the bidirectional data flow equations into two unidirectional systems in order to enable interval analysis. Communication 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 [24, 23] modeled pipelined communication and communication buffer requirements in a unified data flow and interval analysis framework. Communication optimization focuses on latency hiding which is based on the lazy code motion strategy [25]. Message coalescing is applied to communication statements that are placed at the same program point. Separate data flow equations are required to ensure buffer-safe and balanced communication placement, and loops are modeled explicitly. Our data flow framework is based on simpler hoisting and sinking analysis [26], implicitly ensures balanced communication, and is solved iteratively for arbitrary control flows. If a program node exceeds the buffer available then all communication is blocked at that node. In contrast to our approach no attempt is made to selectively block communication with lower priority (smaller communication time) until all buffer constraints are honored.

Chakrabarti, Gupta and Choi [5] 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. SENDs and RECVs are not separately considered but always placed at the same node which would make it very difficult to incorporate communication latency hiding in their approach. A variety of communication placements are generated. The final placement 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. The issue of buffer-safe communication placement is not addressed by their work.

Note that most existing approaches that are based on data flow analysis require separate data flow equations to model loops and balanced communication placement and 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.

As demonstrated by our approach, cost models are of paramount importance for communication optimizations. Besides P^3T several other performance estimators have been developed. Kremer [28] implemented an automatic data distribution tool for HPF style programs which uses performance prediction to examine different data layouts and their associated communication costs. A set of communication and computation kernel routines are pre-measured for different data layouts, processor numbers, and array sizes. P^3T uses

sophisticated control flow analysis to accurately model complex loop bounds and array reference patterns, whereas Kremer’s approach is based on simplifying assumptions (e.g. fixed loop bounds) at the cost of estimation accuracy.

Gupta and Banerjee [15] described analytical communication cost models in order to support automatic data distribution for distributed memory architectures. It is assumed that each loop index used as a subscript in an array reference varies so as to sweep over the entire range of array elements along that dimension. P^3T [8] detects the non-local array portions accessed inside of loop nests with high accuracy based on modeling loop iteration spaces in combination with array subscript expressions.

Adve et al. [1] described an integration of compilation with a performance framework to support performance analysis of data parallel programs. Static compiler analysis reports among others on data dependences, non-local accesses, type (pipelined, broadcast, shift, etc.) and volume of communication. Furthermore, it is possible to instrument the parallel code and to determine comprehensive performance data through a profile run on the target parallel machine. Due to limited static control flow modeling, the qualitative communication information reported by their compiler is much less detailed and accurate as the one provided by P^3T . Instrumenting and profiling a parallel code delivers very accurate performance results, however, at the price of actually generating and executing the code.

7 Conclusion

We have presented a novel approach to optimize communication based on data flow analysis and performance prediction. Our method differs significantly from previous work in that the communication optimization strategy is not fixed, but conflicting communication profit motives are evaluated carefully by a performance estimator to choose the best one to apply. Firstly, we create a balanced and buffer-safe program such that SENDs are placed as early and RECVs as late as possible. Buffer-safety is achieved by using P^3T , an effective cost estimator, to selectively block SENDs with low priority (smallest estimated communication time) at program nodes until all buffer constraints are honored. Secondly, a novel message coalescing algorithm is used to aggressively eliminate communication redundancy both in terms of number and volume of messages. Thirdly, based on a balanced and buffer-safe program we systematically create and examine a reasonable set of communication placements for a given program covering several (possibly conflicting) communication guiding profit motives including promising combinations of communication latency hiding and reducing the number and volume of messages. P^3T is used to determine the best choice of the created communication placements based on effective cost functions that model work distribution, computation and communication times, and degree of overlapping communication with computation. Employing an accurate performance estimator opens ground for more aggressive communication optimization opportunities that carefully examine performance gains and tradeoffs among applicable optimization strategies which is not achievable with any existing approach.

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 commonly used interval analysis we employ the generic fixed point algorithm in order to solve the data flow system. The described approach is based on unidirectional bit-vector data flow analyses that are less complex as their bidirectional counterparts.

Preliminary performance results based on a pre-prototype implementation demonstrate that our method implies significant reduction in communication costs and show the effectiveness of this analysis in improving the performance of programs.

Future work will 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. We are currently investigating symbolic cost models that are capable to model unknown machine and problem sizes. We also plan to extend our communication optimization strategy to perform interprocedural analysis.

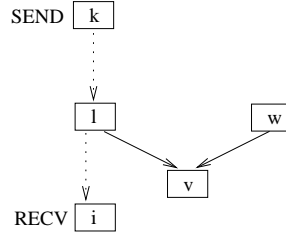


Figure 14: Fragment of a CFG used to proof balanced SEND/RECV placement

8 Appendix

Lemma 8.1

The placement of a SEND and its corresponding RECV is balanced.

Proof 8.1

Let s be a SEND which covers a non-local use u . Let i be a node satisfying either $N\text{-LATEST}_i = \text{true}$ or $X\text{-LATEST}_i = \text{true}$. Then (see Figure 14) there must exist a k with $\text{SEND-CAND}_k = \text{true}$ and a non-trivial path $]k, i[$ which is transparent with respect to u . A non-trivial path consists of at least two different nodes. Now assume that there exists a node $l \in]k, i[$ with

- $N\text{-LATEST}_l = \text{true}$: This implies that $\text{USE}_l = \text{true}$ and $X\text{-DELAY}_l = \text{false}$ which propagates through $]l, i[$ resulting in $N\text{-LATEST}_i = \text{false}$ and $X\text{-LATEST}_i = \text{false}$. This is a contradiction to the assumption of this proof.
- $X\text{-LATEST}_l = \text{true}$: This implies that there must exist a node $v \in \text{succs}(l)$ with $N\text{-DELAY}_v^* = \text{false}$ which is caused by a predecessor node w of v . However, this means that the edge between node l and v is a critical edge which has not been split. This contradicts the assumption that edge splitting is performed for the underlying CFG according to Section 3.1.

Lemma 8.2

The algorithm of Figure 6 about buffer-safe latency hiding and message coalescing terminates.

Proof 8.2

The algorithm clearly terminates if all SENDs are placed without buffer conflict, or if there exists a CFG node n whose non-local uses (included in n) imply a buffer requirement that cannot be honored by max-buffer (maximum communication buffer size of a single processor).

In all other cases a buffer conflict occurs due to SENDs that are live in a CFG node n and whose communication data is not used in n . In every iteration (except the first one) of the REPEAT-loop, a set of SENDs is moved closer to their non-local uses (by blocking the SENDs) in order to honor buffer constraints. Let u refer to the non-local use u that is covered by a SEND s . The latest possible program point to which s can be moved is given by $\text{nodeU}(\text{OrigUse}(s))$, the node where u is placed. Therefore, the algorithm terminates at the latest if all SENDs $s \in S$ are moved to $\text{nodeU}(\text{OrigUse}(s))$.

ACKNOWLEDGMENTS

The authors are grateful to Prof. Hans Zima for his support of this research and to the Vienna High Performance Compiler Research Group, in particular to Alex Pozgaj, for implementing the software infrastructure of P^3T . The authors thank the anonymous referees for their comments that greatly helped improve our presentation.

References

- [1] Vikram S. Adve, John Mellor-Crummey, Mark Anderson, Ken Kennedy, Jhy-Chun Wang, and Daniel A. Reed. Integrating Compilation and Performance Analysis for Data Parallel Programs. . In M.L. Simmons, A.H. Hayes, D.A. Reed, and Eds J. Brown, editors, *Proc. of the Workshop on Debugging and Performance Tuning for Parallel Computing Systems, IEEE Computer Society Press*, January 1996.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] T. Fahringer. Estimating and Optimizing Performance for Parallel Programs. *IEEE Computer*, 28(11):47 – 56, November 1995.
- [7] T. Fahringer. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Publishers, Boston, USA, ISBN 0-7923-9708-8, March 1996.
- [8] T. Fahringer. Compile-Time Estimation of Communication Costs for Data Parallel Programs. *Journal of Parallel and Distributed Computing, Academic Press*, 39(1):46–65, Nov. 1996.
- [9] T. Fahringer. On Estimating the Useful Work Distribution of Parallel Programs under P^3T : A Static Performance Estimator. *Concurrency, Practice and Experience (Ed. Geoffrey Fox)*, 8(4):261 – 282, May 1996.
- [10] 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. IEEE Computer Society Press.
- [11] 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. IEEE Computer Society Press, January 1997.

- [12] T. Fahringer. Efficient Symbolic Analysis for Parallelizing Compilers and Performance Estimators. *Journal of Supercomputing*, Kluwer Academic Publishers, 12(3):227–252, May 1998.
- [13] T. Fahringer, M. Haines, and P. Mehrotra. On the Utility of Threads for Data Parallel Programming. In *9th ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995. ACM Press.
- [14] T. Fahringer and B. Scholz. Symbolic Evaluation for Parallelizing Compilers. In *Proc. of the 11th ACM International Conference on Supercomputing*, pages 261–268, Vienna, Austria, July 1997. ACM Press.
- [15] M. Gupta and P. Banerjee. Compile-time estimation of communication costs on multicomputers. In *Proc. Sixth International Parallel Processing Symposium*, Beverly Hills, CA, March 1992.
- [16] 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.
- [17] 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.
- [18] Hanxleden, R.v. and Kennedy, K. Give-N-Take—a balanced code placement framework. . In *ACM SIGPLAN'94 Conference on Program Language Design and Implementation*, Orlando, FL, June, 20-24 1994.
- [19] M. S. Hecht. *Flow Analysis of Computer Programs* . Elsevier, North-Holland, 1977.
- [20] High Performance FORTRAN Language Specification. Technical Report, Version 2.0.δ, Rice University, Houston, TX, October 1996.
- [21] S. Hiranandani, K. Kennedy, and C.W. Tseng. Evaluating Compiler Optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21(1):27–45, 1994.
- [22] J.B. Kam and J.D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.
- [23] 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.
- [24] 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.
- [25] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI'92) (San Francisco, CA)*, volume 27,7 of *ACM SIGPLAN Notices*, pages 224 – 234, 1992.
- [26] 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.
- [27] D. E. Knuth. An empirical study of FORTRAN programs. *Software - Practice and Experience*, 1(2):105–133, 1971.
- [28] U. Kremer. *Automatic Data Layout for Distributed Memory Machines*. PhD thesis, Rice University, Technical Report CRPC-TR95559-S, October 1995.

- [29] 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.
- [30] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB - A Tool For Semi-Automatic MIMD/SIMD Parallelization. *Parallel Computing*, pages 1–18, 1988.
- [31] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - a language specification. Technical report, ICASE, Hampton,VA, 1992. ICASE Internal Report 21.

THOMAS FAHRINGER received a Masters degree in 1988 and a Ph.D. in 1993, all in Computer Science from the Technical University of Vienna, Austria. He was a visiting scientist at the Engineering Design Research Center at Carnegie Mellon University from 1988 - 1990. Since 1990 he is Assistant Professor of Computer Science and since 1998 Associate Professor both at the Institute for Software Technology and Parallel Systems, University of Vienna. His research focuses on software tools for parallel programs and multiprocessor architectures in particular parallelizing compilers, symbolic program and performance analysis. Readers may contact Fahringer at the Institute for Software Technology and Parallel Systems, University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria; WWW: <http://www.par.univie.ac.at> and e-mail: tf@par.univie.ac.at

EDUARD MEHOFER received his MS and PhD degrees in computer science from the Technical University of Vienna in 1989 and 1998, respectively. From 1986 to 1993 he worked at the Alcatel Research Centre. In 1993 he joined the Institute of Software Technology and Parallel Systems at the University of Vienna as Assistant Professor. His current research interests are high performance languages, parallelizing compilers, compiler optimizations for high performance computing, and scientific computing. For further information see <http://www.par.univie.ac.at>.