

Problem and Machine Sensitive 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

Accepted for publication in Proc. of the 12th ACM ICS'98, Melbourne, Australia, July, 1998

Abstract

Reducing communication costs can significantly improve the execution time of a parallel program. This paper presents a new approach for communication optimization in data parallel programs that is based on global data flow analysis and performance prediction. Our techniques are based on simple yet highly effective data flow equations which are solved iteratively for arbitrary control flow graphs. Previous techniques are based on fixed communication optimization strategies whose quality can be very sensible to changes of problem and machine sizes. Our algorithm is novel in that we carefully examine tradeoffs between communication latency hiding and reducing the number and volume of messages (e.g. message coalescing and aggregating) by systematically evaluating a reasonable set of promising communication placements for a given program covering several (possibly conflicting) communication guiding profit motives. P^3T , a state-of-the-art performance estimator that carefully models problem and machine characteristics, is used 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 losses among applicable optimization strategies which is not achievable with most existing approaches. Our techniques are based on data flow analyses that enable vectorizing, coalescing and aggregating communication, and overlapping communication with computation both within and across loop nests all in a unified framework. We present results for the Meiko CS-2 and a network of Sparc workstations based on a preliminary implementation which show that our method implies a significant reduction in communication costs and demonstrate the effectiveness of this analysis in improving the overall performance of data parallel programs.

* Research supported by the Austrian Science Fund as part of Aurora Project "Tools" under contract SFBF1104

1 Introduction

The overhead to access non-local data from remote processors on distributed memory architectures is commonly orders of magnitude higher than the costs of accessing local data. As a consequence, a key problem to effectively use distributed memory architectures is centered around efforts to optimize communication [3, 7, 9, 8, 10, 1] which includes: message vectorization (hoisting communication outside of loops), message coalescing (removing redundant communication based on the same array), communication aggregation (combining 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. Note that reducing the number of messages can also be a valuable optimization for message passing programs that are executed on shared memory architectures, since fewer messages commonly translate into less synchronization points [11].

Most approaches [8, 10, 9] for global scheduling of communication commonly place SENDs as early and RECVs as late as possible in order to maximize communication latency hiding. Hardly any communication optimization considers communication buffer constraints, although it has been shown [10] that latency hiding can dramatically increase the size of communication buffers which can even result in run-time errors.

Recently it has been observed [3] that maximizing communication latency hiding can prevent valuable opportunities to reduce the number of messages and to avoid sending partially redundant data. An alternative strategy has been developed in [3] which tries to schedule communication later such that as many individual communications as possible are combined at the same (not necessarily at the earliest possible) program 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 fixed communication optimization strategies whose quality can be very sensible to changes of machine and problem sizes. Ignoring tradeoffs between communication latency hiding and reducing the number and volume of messages when changing machine and problem sizes may result in considerable performance losses. Furthermore, many cost models – that are employed to guide communication optimization –

lack estimation accuracy and portability to other architectures.

This paper describes a novel communication optimization framework that is based on global 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 a performance estimator – considering also machine and problem characteristics – in order to choose the best strategy. We start with an initial buffer-safe communication placement that positions SENDs to the earliest possible buffer-safe program point, aggressively coalesces SENDs, and sinks RECVs as far as possible, all of which is based on advanced global data flow analyses as described in [7]. Based on this initial communication placement we systematically create and examine a reasonable set of promising communication placements (without actually generating code) 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. Then we use P^3T [4], a state-of-the-art performance estimator for distributed memory parallel programs, 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 most existing approaches.

Furthermore, we employ advanced techniques to coalesce and aggregate messages. All communication placements are buffer-safe and balanced (SENDs and RECVs are placed at the same loop nesting level). We also 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 analysis of our symbolic analysis, which goes beyond the scope of this paper, the reader may refer to [5].

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 briefly describes our initial 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. After the discussion of related work in Section 6 we will give our conclusions in Section 7.

2 Preliminaries

2.1 Program Model

The control flow graph (CFG) of a program is represented by $G = (N, E, e, x)$ 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 $m, n \in N$. Nodes e and x denote the unique *entry* and *exit* node of G , which are assumed not to possess any predecessors and successors, respectively. A path in G is a sequence of nodes $[n_1, \dots, n_k]$ such that $\forall i, 1 \leq i < k: n_{i+1}$ is a successor node of n_i . 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' 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 [2].

A SEND represents the sending component which initiates communication, whereas a RECV finalizes the communication. We implement a SEND 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 that can be realized by a single SEND/RECV pair, by several SENDs combined with a single RECV, or by a single SEND combined with several RECVs. The second case can occur if the underlying data flow analysis is hoisting a SEND upwards through a join node (more than one predecessor). The third case may occur if a RECV is moved downwards through a branch node (more than one successor). 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 part or all of u is being sent by s . $Uses(s)$ defines the set of non-local uses that are associated with a specific SEND $s \in S$.

2.2 Performance Prediction

For solving communication buffer conflicts and finding the best communication placement out of a variety of possible placements we use P^3T [4], an accurate and effective performance estimation tool for distributed memory parallel programs. Among others, P^3T estimates at compile-time the following performance parameters:

- Work distribution [4] estimates how evenly the computations of a parallel program are distributed across all processors executing the program.
- Amount of data transferred [4] is an estimate of the number of data elements transferred.
- Communication time [4] of a specific communication statement is estimated by the maximum communication time across all processors involved in the communication.
- Computation time [4] is an estimate of the time it takes to execute the computations of a parallel program.

3 Initial Communication Placement

In this section we briefly describe our initial communication placement strategy by creating a program called ELB (Earliest SEND - Latest RECV - Buffer-safe) which serves as

a basis for our problem and machine sensitive communication optimization as described in Section 4. The ELB program is obtained by hoisting SENDs to the earliest possible program points considering also 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. For a detailed description of our initial communication placement, the reader may refer to [7].

Figure 1 shows our running code example which is used throughout the paper to demonstrate our communication optimization techniques. All arrays (dimension $m \times m$) in the example code are column-wise distributed onto a set of P processors. We will use Fortran 90 style of describing array accesses. Furthermore, for the ease of presentation, 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 SENDs with their corresponding non-local use. We will follow these policies throughout the paper.

Earliest Buffer-Safe SEND Placement

Starting point of this analysis is the set of non-local uses. Every non-local use implies a SEND. Intuitively, SENDs must be hoisted to the earliest possible program points – while maintaining the program semantics – in order to maximize latency hiding. The analysis of hoisting SENDs is based on a global data flow analysis [7] which operates on a CFG. A SEND s is blocked in a CFG node n (s cannot be hoisted through 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 .

Commonly every communication is associated with a communication buffer. Hoisting communication to the earliest possible point in a program can cause communication buffers problems [10]. In [7] we described how to make a program buffer-safe.

The SENDs of the code in Figure 1 are placed as early as possible considering buffer constraints. Note that SEND5 cannot be hoisted out of loop 4 since this loop carries a true dependence that affects the communication data of SEND5. SEND2 is hoisted through join node 10 which yields 2 copies of SEND2: SEND2₁, SEND2₂. Node 7 blocks SEND2₂ due to an exceeding communication buffer at this node. SEND2₂ is blocked at node 7 and is, therefore, placed at node 9. As a consequence, SEND2₁ cannot be hoisted to node 2 and is thus placed at node 3 of the left branch. All other buffer constraints are honored by every node of the program (see also [7]).

Message Coalescing

Once the SENDs of a program have been placed as early as possible we remove partially redundant communication by coalescing messages in order to reduce the number and volume of messages originating from the same array. Our techniques for message coalescing maintain buffer-safe communication placement and are described in detail in [7].

Latest RECV Placement

The analysis starts with the set of already placed SENDs. Every SEND implies a RECV which is moved in the direction

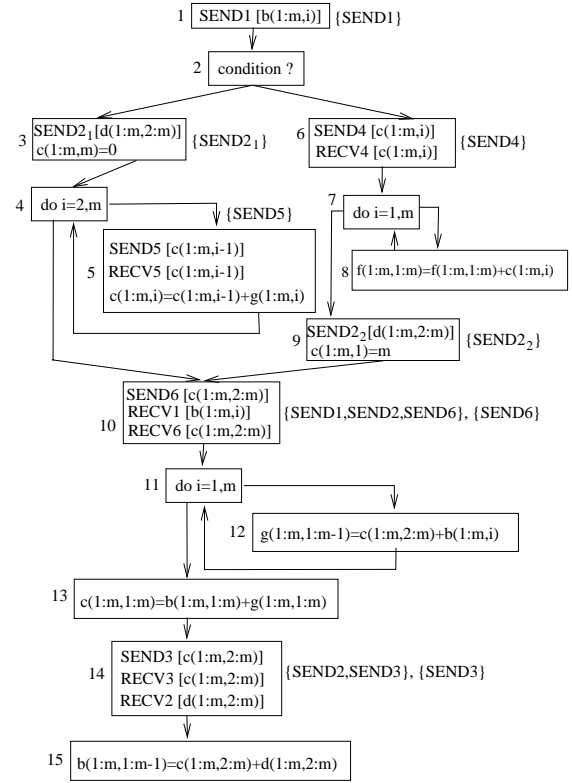


Figure 1: *ELB* (Earliest SEND - Latest RECV - Buffer-safe) communication placement. SENDs and RECVs are indexed by the access patterns which cause the communication.

of the control flow to the latest possible program point before the communication data is used. This results in maximizing latency hiding. The analysis for delaying RECVs is based on a global data flow analysis which is presented in [7].

Figure 1 displays our running code with RECVs appropriately inserted for their associated SENDs. Note that our approach implicitly guarantees balanced placement of SENDs with their associated RECVs.

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

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 maximizes 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

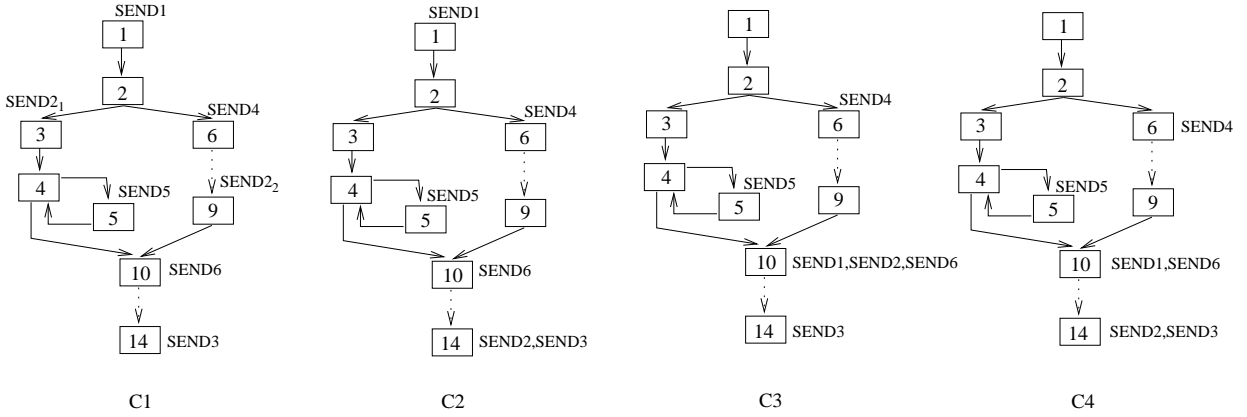


Figure 2: A variety of valid SEND combinations.

but do not try to examine the performance gains and losses of both strategies in order to find the best communication placement for a given problem and machine size.

Consider the code of Figure 1. Instead of placing SEND1, SEND2 and SEND6 as early as possible in order to maximize latency hiding, these SENDs could all be placed and aggregated at node 10 thus reducing the number of messages. An efficient performance estimator is required to determine which communication placement is better.

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. An efficient performance estimator is employed in order to determine the best communication placement created.

4.1 Determine SEND Placements and Maximum Matching SENDs

Firstly, for every $n \in N$ we determine $NS(n)$, the set of SENDs that can be placed in n such that every SEND placement is buffer-safe. Let $Elb(s)$, $s \in S$, denote the set of earliest buffer-safe positions (in terms of CFG nodes) of s in ELB (earliest SEND - latest RECV - buffer-safe placement according to Section 3). Note that there can be several SENDs for a specific non-local use u according to Section 2.1, for instance, $Elb(SEND2)$ is given by nodes 3 and 9 in Figure 1. 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. Note that all of the above placements of s are buffer-safe, since the nodes at which s is placed are dominated by a node in $Elb(s)$ which is guaranteed to be buffer-safe. If $NS(n) = \phi$ then there does not exist a SEND which can be placed in n .

Secondly, we determine the maximum *matches* of SENDs in every $n \in N$. We say that two SENDs s_1, s_2 *match* with each other iff s_1 and s_2 imply either the same sender-receiver relationships or subsets of it. Matching SENDs may be based on the same (message coalescing) or different arrays (message aggregation). For every $n \in N$ we create $Groups(n)$, a set of groups (subsets of SENDs), such that for every group $g \in Groups(n)$ the following holds: $g \subseteq NS(n)$ and all $s \in g$ mutually 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. Moreover, for each SEND s we create a separate group – containing only s – that is put in every node $n \in Elb(s)$. Note that SEND2₁ and SEND2₂ in Figure 1 are added as separate groups to nodes 3 and 9, respectively.

Finally, we traverse the CFG and delete all groups $g \in Groups(n)$ 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 SENDs that are included in g). This strategy tries to optimize both communication latency hiding and reducing number and volume of messages.

Note that our method is very flexible for adding additional SEND placements of interest or change the policy for placing SENDs into 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 and aggregation to SENDs that are placed at the same program point.

Figure 1 presents the sets of $Groups(n)$ in braces. For instance, two groups are created for node 14: $\{SEND2, SEND3\}$ and $\{SEND3\}$. SEND2 and SEND3 are put into the same group as they imply the same sender-receiver relationship. Node 14 is the earliest possible placement for $\{SEND2, SEND3\}$ which is not included in a previous node of the CFG. A separate group is created for SEND3 as node 14 is the earliest placement for SEND3 in ELB.

4.2 Determine SEND Combinations and Place RECVs

In the previous Section we identified various possibilities to place SENDs. In this section we describe how to determine a variety of promising SEND placements for various combinations of 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. A *valid SEND combination* C refers to a subset of $Groups_G$ that contains every $s \in S$ such that: If C contains a group with a copy of s such that there exist multiple copies of s then C must include one group for each copy of s . In all other cases s is included in exactly one group of C .

According to Section 2.1, multiple copies of a specific SEND s may be created by hoisting s through a join node

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

$$Overlap(g) = \frac{1}{|Paths(g)|} * \sum_{w \in Paths(g)} (CompTime_{path}(w) - CommTime(g)) * Prob(w) \quad (2)$$

$$CommCost(C) = \sum_{g \in C} |\gamma(Overlap(g))| \quad (3)$$

$$\gamma(r) = \begin{cases} 0 & : \text{if } r \geq 0 \\ r & : \text{otherwise} \end{cases} \quad (4)$$

Figure 3: Computation and communication cost functions

in the opposite direction of the control flow. For instance, a valid SEND combination of the code example in Figure 1 which contains {SEND2₁} must also contain {SEND2₂} since SEND2₁ and SEND2₂ are copies of SEND2.

Let VC specify the set of all valid SEND combinations of a CFG. Although 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.

In order to amortize message startup costs, we generate a single SEND (aggregate communication) for every group $g \in C$. For every g we have to place the associated RECVs as late as possible. For this purpose we reuse our analysis for placing RECVs (see Section 3) and apply it to groups. Aggregated SENDs and RECVs are again balanced.

Figure 2 shows the associated SEND placements of all four valid SEND combinations ($C1, \dots, C4$) for the running example. For the ease of presentation this figure displays only the node numbers which correspond to those in Figure 1. Nodes where no SENDs are placed are not shown (indicated by dotted arrows). For instance, SEND1 can either be placed as a single SEND in a group at node 1 ($C1$ and $C2$) or in a group at node 10 together with SEND2 and SEND6 ($C3$). 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 coalescing and 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.

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 3) 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. 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 that message aggregation overhead may vary for different SEND combinations.

Each group in C 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 is computed as the maximum communication time across all processors involved in g multiplied by the execution count of g (identical with the execution count of the associated SEND of 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 which is based on the computation time parameter of P^3T . $CompTime_a(i)$ (cf. Equation (1) of Figure 3) is the time required to execute all instances of i in a parallel program. $freq(i)$ is the 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 [4] 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 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. 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 (1). 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. The example of Figure 4 shows a SEND and all its associated RECVs of a group $g \in Groups_G$. The outgoing edges of branches are marked with their probability of being taken during execution of the program. We must examine three 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 covered by node 2. $Prob(w)$, the probability that a path w is actually executed, is computed by multiplying all edge probabilities along this path. Therefore, $Prob(w_1)=0.32$, $Prob(w_2)=0.48$, and $Prob(w_3)=0.2$.

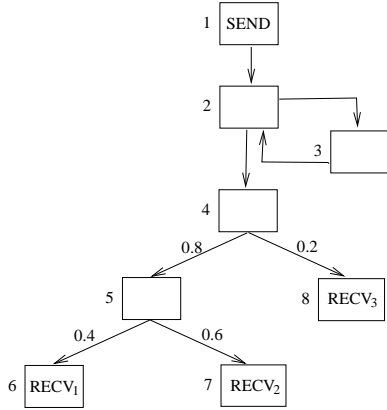


Figure 4: SEND/RECV paths of a group $g \in Groups_G$.

$Overlap(g)$, the degree of overlapping computation with communication of a specific group g of C is then computed – according to Equation (2) – as the average latency hiding across all possible paths of g weighted by the paths’ probability. Equation (3) 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 .

5 Experiments

We have implemented a pre-prototype of our communication optimization strategy as part of *VFCS* (Vienna Fortran Compilation System [2]), a compiler for distributed memory architectures. In the following we describe the results of our experiments performed to measure the potential benefits of our cost driven communication optimization.

In the first experiment, we examine the performance of all four communication placements $C1, \dots, C4$ as shown in Figure 2 for various problem sizes. 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 size of the array dimension). We report the results in Figure 5. In each bar-chart the x-axis specifies the problem size m and for each problem size 4 bars are plotted, one for each code version ($C1, \dots, C4$). 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 in the worst case of order $\Theta(m)$ and the data volume exchanged is of order $\Theta(m^2)$. Therefore, the effect 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. $C3$ and $C4$ are those versions that highly focus on message aggregation. Both of them 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. Therefore, 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 6 plots our communication cost function $CommCost$ according to Equation (3) of Figure 3 for the same set of problem sizes and code versions as shown in Figure 5. Again, the y-axis is normalized so that the code version with the largest $CommCost$ value 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, in Figure 5 the communication time of $C3$ for $m = 64$ accounts for approximately 65 % of the communication time of $C1$, whereas in Figure 6 it is close to 50 %. The reason for this loss in estimation accuracy is due to the difficulty to accurately model computation times [4] 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 each 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 significant 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 strategy across different problem sizes. In our experiment $C2$ seems to be the best compromise as it achieves the best average communication behavior across all problem sizes and communication placements considered. Note that $C2$ does not emphasize a specific communication optimization strategy but is a combination of several strategies. The authors believe that $C2$ 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 applied to a

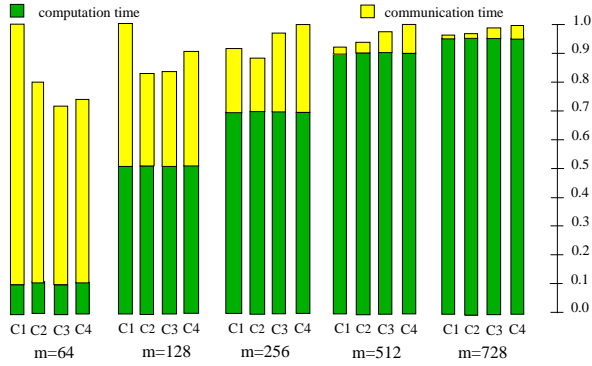


Figure 5: Normalized measured runtimes of code versions C_1, \dots, C_4 on a Meiko CS-2 with 16 processors

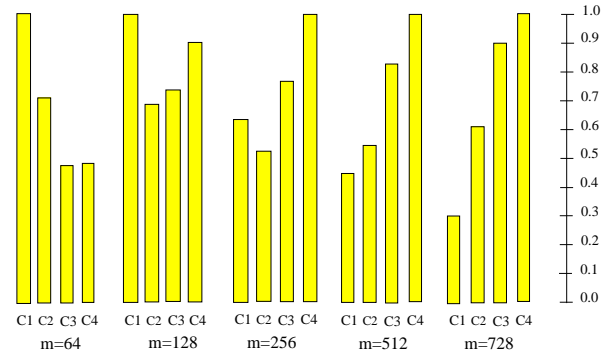


Figure 6: Normalized estimated communication cost function $CommCost$ for C_1, \dots, C_4 on a Meiko CS-2 with 16 processors

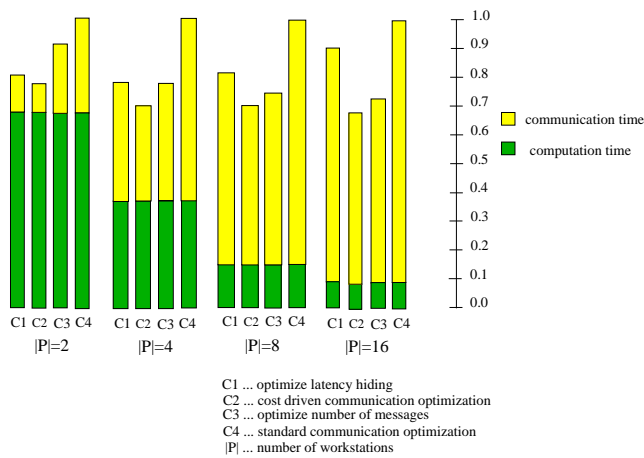


Figure 7: Normalized measured runtimes of four different PIC codes on a network of workstations

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 implemented a parallel PIC version (approximately 3500 lines of code) following a method described in [6] 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 program points where they can be aggregated with a maximum number of other messages. C4: communication is hoisted to outermost loops and messages are aggregated and coalesced 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 Ethernet (10 Mbits/sec bandwidth). Figure 7 shows the measurements. The x-axis of the bar-chart shows the number of workstations. 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 all code versions is fixed as the number of particles (1024) does not change. The communication overhead is the predominant factor of the overall runtime due to the slow network connecting the workstations and the fact that 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, for increasing number of workstations message aggregation is critical for reducing high message startup overheads on workstation networks. C3 outperforms C1 for 8 and 16 workstations by 9 % and 19 %, respectively. C2 is superior to all other code placements. The performance of C4 is inferior to all other versions examined as it is based only on standard communication optimizations.

The important observation of this experiment is the following: C2 is not drastically better than the next best version for every 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, C2 is 2.5 % - 22 % and 5.5 % - 14 % faster than C1 (latency hiding) and C3 (message aggregation), respectively. Hence, C2 is a sensible 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 considered. C2 has a reduced communication overhead ranging from 36 % - 70 % as compared to the standard communication placement C4.

6 Related Work

Communication optimization [2, 3, 8, 10, 1] has been extensively researched. Most existing compilers employ a combination of message aggregation and coalescing and collec-

tive communication which is based on single-loop analysis. Pipelined communication [10, 1] 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 [1] and across loops [8, 10, 9].

Von Hanxleden and Kennedy [9] optimized communication with respect to latency hiding for irregular programs. SENDs and RECVs are balanced. Arrays are treated as indivisible units which prevents exploiting compile-time knowledge about regular array subscript expressions.

Gupta, Schonberg, and Srinivasan [8] use a data flow and interval analysis to maximize latency hiding and to detect redundant communication. 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.

Chakrabarti, Gupta and Choi [3] 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 placing communication closer to the uses at the cost of latency hiding. SENDs and RECVs are not separately considered for placement as in most data flow analyses but always placed at the same node which prevents communication latency hiding. 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.

7 Conclusion

We have presented a novel approach to optimize communication based on global 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 by considering problem and machine characteristics. We start with the creation of an initial buffer-safe communication placement which maximizes latency hiding and message coalescing and serves as a basis for our problem and machine sensitive communication optimization. Based on the initial buffer-safe communication placement, 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 by using effective cost functions that model work distribution, computation and communication times, and the 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 losses among applicable optimization strategies which is not achievable with most existing approaches.

Preliminary performance results based on a pre-prototype implementation demonstrate that

- different communication optimization strategies become

the first order optimization effect for changing problem and machine characteristics.

- our method can achieve significant reduction in communication costs as compared to a fixed communication optimization strategy, and
- our analysis is very effective in improving the communication behavior and performance of parallel programs.

We are currently extending our approach to perform interprocedural analysis and to employ symbolic cost models for unknown machine and problem sizes.

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] S. Benkner et al. *Vienna Fortran Compilation System - Version 2.0 - User's Guide*, October 1995.
- [3] 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.
- [4] T. Fahringer. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Publishers, Boston, USA, ISBN 0-7923-9708-8, March 1996.
- [5] T. Fahringer. Efficient Symbolic Analysis for Parallelizing Compilers and Performance Estimators. *accepted for publication in the Journal of Supercomputing, Kluwer Academic Publishers*, to appear in 1998.
- [6] 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.
- [7] T. Fahringer and E. Mehofer. Buffer-Safe Communication Optimization based on Data Flow Analysis and Performance Prediction. In *Proc. of the 1997 International Conference on Parallel Architectures and Compilation Techniques (PACT'97)*, pages 189–200, San Francisco, USA, November 11-15 1997.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *ACM Sigplan Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Santa Barbara, CA, July 1995.