Utilizing New Communication Features in Compilation for Private-Memory Machines

Susan Hinrichs
Thomas Gross

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract
The communication system of some 3^{rd} generation private-memory machines provides long-lived connections (which reserve communication resources like buffers between nodes) as well as direct access by the computation unit(s) of the node to the communication system. These features allow a compiler to find innovative solutions when compiling data-parallel programs for a private-memory machine. In this paper, we discuss some optimizations that can be included in a compiler for the iWarp system, an example private-memory parallel system with a novel communication architecture.

1 Introduction

Traditional private-memory systems (e.g, the 1^{st} and 2^{nd} generation systems like the Intel iSC/1 and iSC/2, the NCube1 and NCube2, etc.) are based on the processor pool model: the parallel system provides a pool of processors (nodes), and these processors communicate via messages without any explicit knowledge of the interconnection network. In such systems, the communication system reserves communication resources (e.g., buffers) to send a message and gives up those resources after the message has been sent. Multiple messages sent between the same pair of processors may pass over different routes.

One advantage of this model is that the programmer does not need to worry about the details of the communication system. As long as programs are explicitly parallelized by programmers, such a simplified model of the communication system may be necessary. However, this model also hides the communication details from the parallel compiler, thereby depriving the compiler of the opportunity to exploit properties of the communication system. With a good understanding of a program's communication patterns, a parallel compiler can take advantage of features of the communication system in 3^{rd} generation parallel systems like iWarp, Paragon, and CM-5.

Some parallel systems provide communication at the connection level by supporting *long-lived connections*[1]. For long-lived connections, the communication resource reservations are not necessarily released after the message has been sent. Instead several messages can be sent over the same long-lived connection without contending for communication resources. Messages sent between a pair of processors over the same long-lived connection will always follow the same path and therefore be received in order. Since the compiler has control over what happens on the long-lived connection, it has more options for communication optimization.

Newer parallel systems also provide support for finer grained communication. As communication hardware has improved, the communication hardware overhead has decreased, so it is feasible to send smaller messages than could profitably be sent in earlier systems. This change in feasible message size means that some communication optimizations employed by compilers for 1^{st} and 2^{nd} generation system may need to be rethought.

2 Problem background

The data parallel programming paradigm provides a simple, singled-threaded model for the programmer of a parallel machine. While compiling data parallel programs to private-memory machines, the compiler must insert communication to implement the shared namespace model advanced by the data parallel paradigm. Minimizing the communication cost is essential to efficiently target such programs to private-memory machines. In many cases, communication costs from a poorly distributed problem can overwhelm any parallelism performance gains. To minimize communication costs, it is vital that the back end of the compiler has a good understanding of the underlying communication hardware and has the ability to control that hardware.

There are two basic types of communication optimizations that are valid for a private-memory system: minimizing the amount of unnecessary communication and minimizing the execution time of necessary communication. By wise placement of data, the compiler can avoid unnecessary communication, but most data parallel programs will still require some communication. The optimizations discussed in this paper are concerned with making this necessary communication run fast.

We are taking the data parallel approach by writing a compiler for programs written in a dialect of Fortran targetted to iWarp, a fine-grained, private-memory machine[2]. The compiler takes advantage of the long-lived connections and fine-grained communication supported by iWarp. Our dialect of Fortran is based on Fortran 77 and uses the array extensions of Fortran 90 and data mapping directives in the style of those provided by High Performance Fortran [4]. See [7] for an overview of this project.

2.1 Compiler model

One basic execution model of shared namespace programs on private memory machines follows the *owner computes rule*. In this model, an array is assigned over a set of processors, so each

[1]The benefits of long-lived connections have been recognized in computer networks for a long time, and novel network designs blur the boundary between private-memory parallel systems and distributed systems[13].

processor owns a portion of the parallel array. When executing a statement, the processors that own portions of the left hand side array are responsible for fetching the portions of the right hand side arrays needed to evaluate the new values of the left hand side array. This simple model removes confusion about which processors are responsible for each statement computation. Our compiler follows this rule conceptually, but for efficiency the implementation may stray from this rule.

For a particular statement, there is an inherent *alignment* between the arrays in the statement. In the following statement, corresponding elements i of A, B, and C should be on the same processor before this statement can be executed:

```
A(1:n) = B(1:n) + C(1:n)
```

For this statement, arrays A, B, and C are *aligned* if if `A(i)`, `B(i)`, and `C(i)` are allocated to the same processors. If they are not on the same processor when the statement is executed, data must be communicated to preserve the alignment constraint.

Our compiler executes a data alignment phase that produces a data alignment graph following the algorithm developed for the Connection Machine Fortran compiler[12]. This algorithm tries to determine which array dimensions should be placed on processors together to satisfy the alignment constraints without communication. In the data alignment graph, nodes represent aligned array dimensions that should be placed together on the processors, and arcs represent communication that must occur to maintain the alignment constraints at run time.

From the data alignment and distribution information, some of the communication patterns can be calculated at compile time. With sufficient compile time information, data to processor maps can be calculated for each array reference. These maps can be used to calculate processor to processor communication maps between array references. The communication maps contain the communication patterns used in the later compiler phases. In many cases, these communication maps can be automatically calculated and stored using a matrix notation[11].

2.2 iWarp: the target system

iWarp communication hardware provides many opportunities for communication optimization. Each iWarp processor consists of a pair of tightly integrated computation and communication agents. An iWarp array is made of iWarp nodes (sometimes called "cells") connected in a torus by high speed physical busses. Each bus operates at 40Mbytes/sec. Since the communication and computation agents are tightly integrated, the communication overhead is low, so fine-grained communication is possible. The cost of sending a word is the same as the cost of writing a register.

iWarp supports DMA-style, asynchronous communication in addition to the program directed, fine-grained communication. The communication hardware supports this asynchronous communication model through a limited number of *spools*. Each spool can be set up to send or receive data over the network in the background while the computation agent is performing other calculations. Spools can be used to build an efficient, general message passing library.

The iWarp processor provides multiple *logical channels* between adjacent nodes, so multiple high speed connections can be set up using the same physical busses[1]. On each node there is a finite number of queues that buffer the data sent over the logical channels. A logical channel is assigned a queue on the source node and another queue on the destination node. These logical channels can be chained together to form a *pathway* by using the destination queue of one logical channel as the source queue of another logical channel. Since the communication and computation agents on a node are separate, they can operate asynchronously, so the communication agents on intermediate nodes can forward data through the node without interrupting the computation agent on that node. Pathways enable high speed communication to more nodes than just the nearest neighbors in the physical torus. Multiple messages can be sent over a pathway before its resources are freed, so pathways can be treated as long-lived connections.

Using iWarp's communication software[10], a user can group connections into *networks* between nodes[6]. These networks are mapped onto dedicated pathways at run time. The user's node program accesses these networks at run time through distinct ports and can concurrently exchange multiple messages over these networks. A network can have multiple sending and/or receiving ports. In this case, the nodes communicating over this network use it like a unidirectional bus or ring. Nodes "upstream" can send messages to nodes "downstream" on the network. Nodes that communicate over the same bus can also share messages. A node can read part of a message and *forward* the rest of the message on to another node, without reading the rest of the message into memory. Nodes can also add data to a message without reading the whole original message into memory.

Preserving the expected order of messages is a problem of mapping data parallel programs to private-memory machines. Since iWarp supports long-lived connections, it provides more possibilities for solving this message ordering problem. iWarp's fine-grained communication also merits exploring different communication optimization choices. The remainder of this paper discusses optimization issues related to the design of the message ordering and communication optimization phases of our data parallel Fortran compiler.

3 Message ordering

In the SIMD model, explicit program synchronization is not necessary to maintain the order of communication. The single stream of control implicit in this model automatically runs in the processors in lock step, so they do not need to wait for communication. No processor will communicate before the other processors are ready.

When mapping programs under the MIMD model, explicit program synchronization is necessary because the processors are not running in lock step. With shared memory, processors must synchronize while waiting for communication through shared memory. With private memory, this communication synchronization is unnecessary. Since all communication is explicit in the form of messages, the private memory processors wait for communication by waiting for messages to arrive. However, with private memory, an effort must be made to preserve the expected order of these messages. Consider the example shown in Figure 1. Given two parallel statements executed over three processors, the following message passing pattern could occur. In the first statement (S1), processor P1 sends to processor P2. In the second statement (S2), processor P3 sends to P2. If the two statements are executed without observation of the statement boundary, P3 may speed ahead to S2 and send its data to P2, and this message may arrive at P2 before the message from P1 arrives.

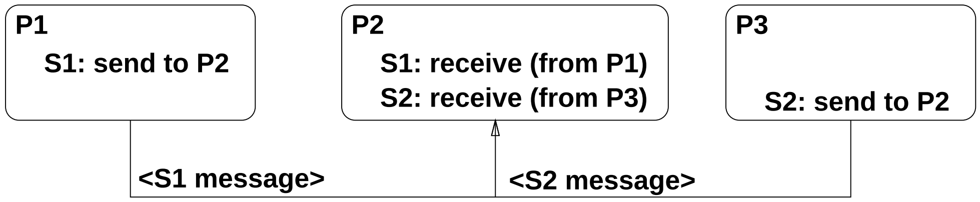


Figure 1: Processor P2 may see the messages from statements S1 and S2 in an unexpected order.

Of course this particular example can be solved by labeling messages and buffering out of order messages. However, this will not cleanly solve the general problem. Given a constant amount of buffering space, an example can always be created that needs more buffering space. Consider the example in Figure 2. At each iteration of the loop, the owner of that row of B sends it to processor P1. Without statement separation, the processors (other than one) will go ahead and send their data to processor one. The amount of buffer space needed to deal with out of order message arrival will be a function of the number of iterations `n`.

```
for i = 1 to n
    A(1:n) = f(B(i, 1:n), A(1:n))
```

Figure 2: Example of potential bad message buffering case. Assume A is stored on processor P1, and the rows of B are spread over all other processors.

3.1 Barrier synchronization

Another straightforward solution is to emulate the SIMD model by inserting barrier synchronizations between each adjacent pair of statements. Then the compiler must still ensure that there is sufficient buffer space for each single communication step, but it does not need to consider the buffer needs of multiple communication steps together.

However, barriers are not strictly needed between each pair of statements. When is it safe to remove a barrier? A barrier can be removed when it is not possible for a message sent after the barrier to be received before the barrier.

Some statement terminology is needed to explore various synchronization cases. For a statement B, there is a set of S processors that send data, R processors that receive data, and NC processors that communicate no data. So $proc(B) = S \cup R \cup NC$. There is a function M that maps senders to receivers for communication in statement B. Consider two adjacent statements B_1 and B_2, where $proc(B_1) = S_1 \cup R_1 \cup NC_1$ and $proc(B_2) = S_2 \cup R_2 \cup NC_2$. No synchronization is necessary in the following cases:

- $M_1 = M_2$ because the send/receiver pairs are identical in each case. Assume M_1 and M_2 are injective. The hardware guarantees that messages sent between a particular pair of nodes will be received in the order sent.

- $S_1 \cup R_1 = S_2 \cup R_2 = \emptyset$ because there will be no communication.
- $R_1 \cap R_2 = \emptyset$ because receiving sets are disjoint.

It is not sufficient to just try to remove barriers pairwise. All statements before the last barrier must be considered when determining if the next barrier is necessary. Consider a case where $R_1 \cap R_2 = \emptyset$, so that the barrier is removed. Then we consider whether the barrier between B_2 and B_3 is necessary. $R_2 \cap R_3 = \emptyset$ too, but $R_1 \cap R_3 \neq \emptyset$, so it is not safe to remove this barrier.

The communication maps calculated in a previous compiler phase can be used to determine whether array occurrences are aligned or have the same communication maps. Consider two pairs of array occurrences (A_1, A_2) and (B_1, B_2) that must communicate to maintain their alignment constraints. If the communication maps for A_1 and B_1 are equivalent, no buffering or additional synchronization is needed between these two communication steps, because the sending and receiving pairs of processors are the same in both cases. More information is needed to determine if array locations that are not aligned in the data alignment graph actually have disjoint receiving processor sets. In this case, the size of the target processor array is needed to determine absolute array locations.

It may be possible to reduce the cost of synchronization by replacing barrier synchronizations with fuzzy barriers[9]. A fuzzy barrier synchronizes processors over a region rather than at a single location in the code. If a set of processors is almost synchronized, the loosened restrictions of a fuzzy barrier may prevent any processor from actually blocking.

The fuzzy barriers can be inserted by viewing the mapped data parallel program as executing alternating phases of communication and computation. For each original data parallel statement, there must be a communication phase to move data to the correct processors. Then the computation phase performs the calculation specified by the original statement. To maintain message order, it is important that at any particular point in time all processors are either in the communication phase for a single statement or are working on any computation phase. It is acceptable for some processors go on to the next computation phase while other processors are still in the communication phase, but before any processor can continue onto the communication phase for the following statement, all processors must have left the communication phase for the previous statement. Fuzzy barriers can enforce that only one communication phase is active at any point in the execution without forcing all processors to synchronize at the end of each communication phase. If fuzzy barriers do not add too much barrier overhead, this loosening of synchronization restrictions may improve performance by taking greater advantage of the multiple instruction streams.

3.2 Using long-lived connections

For a private-memory machine that only supports message level communication, labeling and buffering out of order messages and adding barrier synchronization are the only message ordering options. The previous discussion assumes only simple message passing and barrier synchronization is available. Systems that support long-lived connections have additional message ordering choices.

The discussion above assumes that all messages arrived on a single long-lived connection. If the nodes communicate over different long-lived connections, fewer ordering restrictions are necessary because the nodes use distinct ports to communicate over each long-lived connection. Communications over disjoint sets of ports do not collide. Assume each node has a limited number of ports p_1 to p_n to use for communication. For each statement the compiler knows which port will be used for the communication, so no barrier is needed between communication steps operating on separate ports.

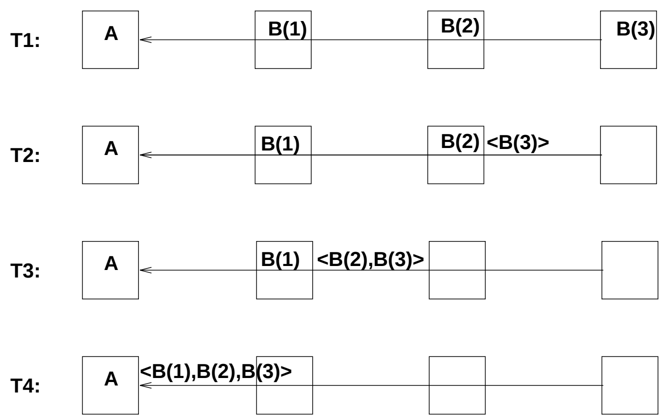


Figure 3: A small example of building a message in order by prepending and forwarding messages.

As described in Section 2.2, messages can be shared over networks with multiple senders and receivers by *forwarding* messages to nodes "downstream". Using forwarding, multiple nodes can work together to compose or build up a message, and no single node must know how many nodes are working together. Also, parts of a single message can be consumed by multiple nodes. Each node can read a portion of the message and forward on the remaining message.

This shared construction and consumption of messages can be used to help with the message ordering problem. By sharing construction of a message, the nodes can impose an order on their portions of the message. The message is started at a known node. Other downstream nodes sharing in the message construction can control where they insert their message sections.

Consider again the example in Figure 2. The nodes that own the rows of B can build up one message to processor P1 with the rows of B in the expected order as shown in Figure 3. The most upstream processor on the network starts the message, and every other processor prepends the row(s) of B it owns. When the message reaches P1, the message contains all the rows of B in order.

Sharing construction of a message may be slower for individual processors than using message passing to send the data directly. A sending node must wait for the shared message to arrive before the node can send its data. However, directly sending messages to a single node will result in network congestion as many messages compete for limited communication resources, and the receiving processor can receive the data more efficiently if it arrives in a predefined order.

Most likely some combination of message buffering, barrier synchronization, and multiple ports must be used to maintain the order of messages. Message buffering is limited by the amount of buffering space available, and making extra buffer copies reduces the amount of memory bandwidth available for "real" work. Barrier synchronization does not add any load on the memory system and is not resource limited, but emulating a SIMD machine wastes the benefits of the MIMD model in the cases where strict synchronization is not necessary. Multiple ports really just provide hardware support for message labeling. However, the hardware labeling lets the program see several messages at the same time without buffering the messages in memory, so it is superior to software message labeling. Unfortunately, there are only a limited number of hardware ports available, so they cannot solve the ordering problem alone.

3.2.1 Assigning barrier synchronization

With a good compile time understanding of the communication patterns, the compiler can make decisions about where barrier synchronizations are needed based on the rules in Section 3.1. Using these rules, the barrier assignment problem can be posed as a data flow calculation. Each array reference, A_i, has the following sets

- $P_{in}(A_i)$ = set of communication patterns reaching A_i
- $P_{out}(A_i)$ = set of communication patterns passing A_i

A communication pattern "reaches" A_i if there is no barrier synchronization between the source of the communication and A_i. A communication pattern "passes" A_i if it reaches A_i and there is no barrier needed for A_i.

If the communication that A_i requires does not collide with the communication patterns in $P_{in}(A_i)$ (i.e. it meets the conditions described in Section 3.1), then no barrier is needed before the communication for A_i can take place. In this case all communication passes A_i.

$$P_{out}(A_i) = P_{in}(A_i) \cup comm(A_i)$$

If one of the communication patterns collides, a barrier is needed before the communication for A_i can safely proceed. Only the communication pattern directly related to A_i passes.

$$P_{out}(A_i) = comm(A_i)$$

At each step P_{in} is the union of all P_{out} sets that precede it. The calculation of the P_{in} and P_{out} sets should continue until a fixed point is reached.

4 Communication optimizations

iWarp's unique communication features provide communication performance enhancements by providing for a direct coupling between the computation and communication units of a processor and by supporting logical channels between adjacent processors. Since the communication and computation units in each processor are directly coupled, the computation unit has direct access to the communication system, so reading the data of a single-word message is as fast as reading a word of a larger message. iWarp's logical channels support direct, long-lived connections between nodes that are not physically adjacent. Long-lived connections remove the need to re-establish connections during the execution of a program and so reduce the overhead associated with communication. iWarp's communication architecture is sufficiently different from earlier designs so that some communication "optimizations" discussed in the literature may not be optimizations on iWarp.

4.1 Specialized connections

The limited number of logical channels requires that in the most general case communication must be performed by a message passing system, since there are not enough logical channels to form direct connections between all nodes. However, frequently used regular communication patterns may benefit from using long-lived special networks. Long-lived networks reserve communication bandwidth, so they avoid potential communication congestion, and communication over long-lived networks will not encounter the overhead of reserving communication resources for each message, making it practical to send shorter messages. As described in Section 3.2 long-lived connections can help solve the message ordering problem.

Regular communication patterns can be detected in the contracted data alignment graph and array distribution information. The remaining arcs in the data alignment graph represent shifts and transposes needed to maintain array alignment at run time. If there are sufficient resources, simple, long-lived networks can be defined for these communication patterns.

4.1.1 Assigning long lived networks

iWarp supports a range of lifetimes for networks. A communication network can exist for one message up to the lifetime of the array program. Assigning resources to communication patterns is analogous to assigning registers to data values with different live ranges. By restricting the program to use either networks that last for one message or dedicated networks that last for the duration of a program phase, a simple algorithm can be used to assign communication resources to the most "worthy" communication patterns.

The compiler can use the communication pattern information to pick possible dedicated networks. Then it chooses a subset of these candidates that provides the optimal performance increase. The compiler can assign each candidate network a value (V) based on the improvement over the message passing library and the number of resources used. Given k candidate networks, $N_1 \ldots N_k$, and p logical channels, the compiler must pick the networks that fit and optimally use the resources. This problem is analogous to bin packing, and the same dynamic programming solution can be used.

Fill in a k by p table where entry T[i,j] equals the maximum sum of values over networks $N_1, .., N_i$ using j logical channels.

- T[1, j] = V_1: if network 1 fits in j logical channels;
  T[1, j] = 0: otherwise;
- T[i, j] = MAX(T[i-1, j], T[i-1, j - logical_channel_count(N_i)] + V_i): choose the maximum value either including the i^{th} network or not

After the table is completed entry T[k][p] contains the maximum sum of networks values that fit in p logical channels. By back tracing the decisions leading to T[k][p], the compiler can find the networks that contribute to this value.

Note that this formulation of a dynamic programming solution depends on choosing good, representative network values (V's). Issues like network congestion and resource contention are difficult to quantify. Experience is needed to determine whether the straightforward issues like connection set up overhead and barrier reduction are enough of a force to make the network values reasonably useful.

4.2 Blocking communication

Many compilers for private-memory machines spend a lot of effort combining communication into larger blocks, because in many multicomputers each communication step has a large start up overhead[5, 3, 14, 8]. In this model, it is better to have fewer larger messages than more smaller messages. In newer multicomputers, this is no longer the case. The communication start up overhead has become a much smaller percentage of the cost of communication. In fact, blocking up communication can make performance worse in machines like iWarp and CM-5 that can read and operate on data from the network without first storing it in memory. If communication is in the loop, the data can *stream* through a node, and the program can use the data directly from the network without storing it in memory. After moving communication out of the loop, the program must store the data in memory possibly performing more memory accesses.

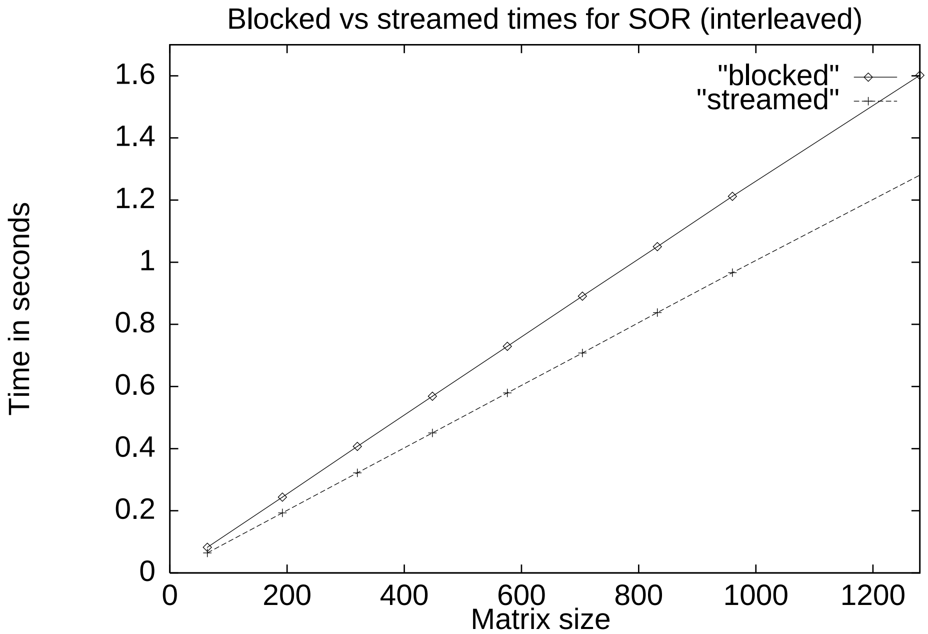
Consider the sequential loop in Figure 5a. Assume `A(i)` and `B(i)` are allocated on the same processor, and the arrays are distributed over N/n processors in a one dimensional array using a cyclic data distribution. Then blocking up all the communication before the loop results in code for each node as shown in Figure 5b. Leaving the communication in the loop results in code for each node as depicted in Figure 5c.

The blocked version of Figure 5b performs $2n$ communication steps and $4n$ memory accesses. The streaming version of Figure 5c (with communication in the loop) also performs $2n$ communication steps but only $2n$ memory accesses. Preliminary measurements of both styles for SOR and LU decomposition on iWarp are shown in Figure 4. There figures show that there is some benefit to leaving communication in the loop for programs like SOR (successive over relaxation). Since LU decomposition spends proportionately more time in computation than communication, the streaming versus blocking choice does not effect its performance as much. Only programs that follow a particular pattern will experience a performance boost from leaving communication in the loop. Programs that eventually store the data to memory or perform many computations for each communicated word will still benefit from blocking up communication.

Blocking data may still seem like an advantage for situations that perform communication in background concurrently with computation even if the blocked version performs more memory accesses. However, background communication is not free. The number of memory accesses will be the same, so the use of the available memory bandwidth will be the same regardless of whether the communication is performed in the background or not.

5 Preliminary results

We have a compiler that generates robust, general communication code and are now implementing the communication optimizations described in Sections 3 and 4 to optimize this general communication. To estimate how effective these optimizations can be, we hand-optimized several programs: matrix vector multiplication, SOR, LU decomposition, and an N–body simulation.



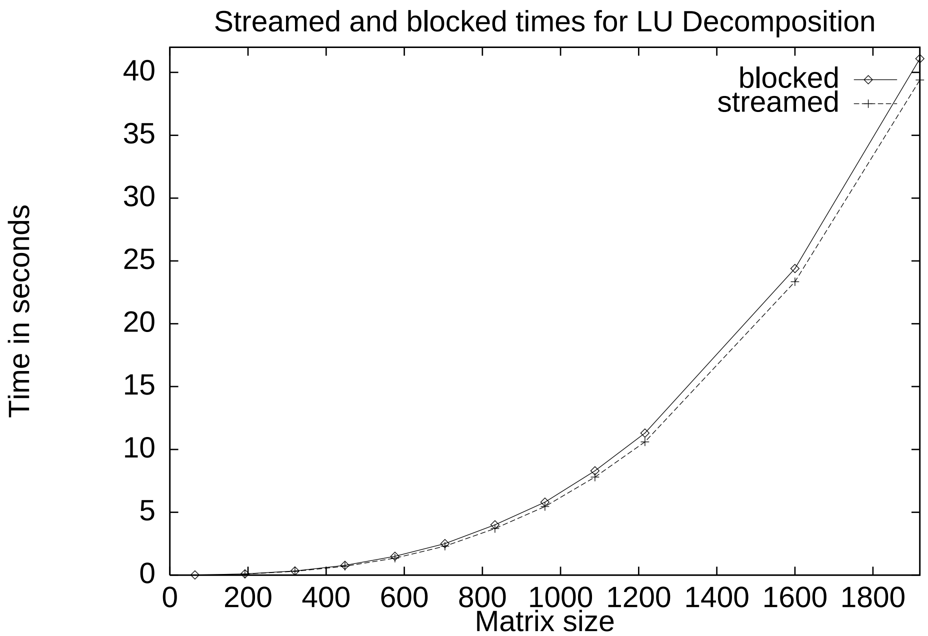


Figure 4: Graphs of run times for SOR and LU decomposition blocking communication and leaving communication in the loop (streaming).

(a)

```
for i = 1 to N
     B(i) = f(A(i+1))
```

(b)

```
send_block(left, A, n)
recv_block(right, ashift, n)
for i = 1 to n
     B(i) = f(ashift(i))
```

(c)

```
for i = 1 to n
      send(left, A(i))
      B(i) = f(receive(right))
```

Figure 5: Blocked versus streamed communication.

Our compiler is a source to source compiler. It takes in a data parallel program and produces a parallelized node program. To hand-optimize the programs, we looked through the generated node program and performed the optimizations described in the previous sections by hand. This resulted in eliminating unnecessary barrier synchronizations, replacing general message passing communication with communication over dedicated networks, and eliminating unnecessary communication checks.

In all cases we checked, the communication optimizations provided substantial execution improvements. Figure 6 shows the results from the SOR example. The input code for the SOR example is listed in Figure 7. The data array `A` is distributed over two processor dimensions, so both array dimensions can be executed in parallel. The data is block distributed to minimize communication. The main source of communication is the shift communication to the nearest neighbor grid processors. In the base case, this communication is performed by a general message passing library. In the optimized case, this communication is supported by long-lived networks that form a nearest neighbor grid.

The other examples we tested showed similar performance improvements. Although these are simple examples, the preliminary results show that these communication optimizations have performance improvement potential.

6 Concluding remarks

Compiler research for earlier private-memory machines often ignored the details of the communication hardware. Newer distributed memory machines offer more opportunities to obtain high performance from the communication system and call for appropriate compiler optimizations to realize the performance potential. At the same time, the communication system also provides

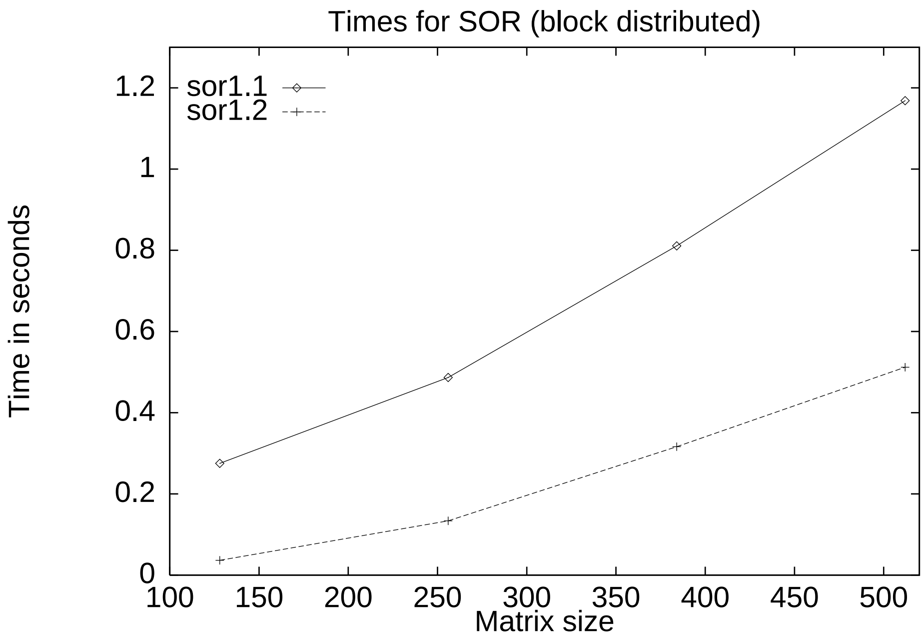


Figure 6: Run times of all SOR cases. Sor1.1 shows the times of the base compiler code. Sor1.2 shows the times of the communication optimized code.

```
 do t = 1, nt
    A(2:nx-1,2:ny-1) = c1*A(1:nx-2,2:ny-1) +
$ c2*A(3:nx,2:ny-1) + c3*A(2:nx-1,1:ny-2) +
$ c4*A(2:nx-1, 3:ny) + c5*A(2:nx-1,2:ny-1)
 enddo
```

Figure 7: Data parallel code for SOR.

support for tasks that typically arise in the compilation of data parallel programs onto distributed memory machines. For example, long-lived connections give the compiler control over a subset of the communication networks, so it can better control message ordering and communication efficiency. Direct coupling between the communication unit and the computation units as well as efficient fine-grained communication make smaller messages feasible by reducing the overhead associated with communication.

References

[1] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. Supporting systolic and memory communication in iwarp. In *Proc. 17th Intl. Symposium on Computer Architecture*, pages

70–81. ACM, May 1990. A revised version has appeared as technical report CMU-CS-90-197.

[2] S. Borkar et al. iWarp: An Integrated Solution to High-Speed Parallel Computing. In *Proceedings of the Supercomputing Conference*, pages 330–339, 1988.

[3] D. Callahan and K. Kennedy. Compiling Programs for Distributed-Memory Multiprocessors. *The Journal of Supercomputing*, October 1988.

[4] High Performance Fortran Forum. *High Performance Fortran Language Specification Version 1.0.*, May 1993.

[5] H. M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, Rheinische Friedrich-Wilhelms-Universitaet, Bonn, Germany, 1989.

[6] T. Gross. Communication in iWarp Systems. In *Proceedings Supercomputing '89*, pages 436–445. ACM/IEEE, November 1989.

[7] T. Gross, D. O'Hallaron, S. Hinrichs, G. Lueh, J. Stichnoth, and J. Subhlok. Compiling Task and Data Parallel Programs for iWarp. In *Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, 1992.

[8] M. Gupta and P. Banerjee. Demonstration of Automatic Data Partitioning Techiniques for Parallelizing Compilers on Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, 1992.

[9] R. Gupta. The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*, pages 54–63, April 1989.

[10] S. Hinrichs. *Programmed Communcation Service Tool Chain User's Guide*. Carnegie Mellon University, release 2.8 edition, 1991. Now part of Intel RTS 3.0.

[11] S. Hinrichs. Communication Optimizations for Private Memory Machines. Thesis propsal, December 1992.

[12] K. Knobe, J. D. Lukas, and G. L. Steele, Jr. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[13] H. T. Kung. Gigabit local area networks: A systems perspective. *IEEE Communications Magazine*, 30(4):79–89, Apr 1992.

[14] A. Rogers and K. Pingali. Process Decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 69–80, June 1989.