Quantifying the Effects of Communication Optimizations*

Sung-Eun Choi Lawrence Snyder

Department of Computer Science and Engineering
Box 352350
University of Washington
Seattle, WA 98195-2350

{*sungeun,snyder*}*@cs.washington.edu*

Abstract

Many papers describe compiler optimizations for communication, but most appear before any performance numbers are available, so it is not generally known how well these optimizations perform in practice. Further, since a compiler cannot apply all known communication optimizations to a given program because of incompatibilities between the optimizations, it is necessary to know how the optimizations compare to each other.

Using a specially constructed machine independent communication optimizer that allows control over optimization selection, we quantify the performance of three well known communication optimizations: redundant communication removal, communication combination, and communication pipelining. The numbers are shown relative to the base performance of benchmark programs using the standard communication optimization of *message vectorization*. The effects on static and dynamic communication call counts are tabulated. Though we consider a variety of communication primitives including those found in Intel's NX library, PVM and the T3D's SHMEM library, the majority of the experiments are run on the T3D using PVM and SHMEM. The results show substantial improvement, with two combinations of optimizations being most effective.

1 Introduction

There exists a rich body of work in optimizing communication for array languages and in parallelizing compilers [1, 2, 10, 15, 13, 21]. There are fewer studies empirically evaluating communication optimizations in the context of a specific compiler and target machine [4, 14, 19]. Moreover, detailed performance evaluations of communication optimizations for non-kernel applications are virtually nonexistent, particularly with respect to optimizations that are performed in a machine independent manner. This may be primarily due to implementation difficulties in achieving portability and performance on previous generation parallel computers.

In this paper, we quantify the effectiveness of three well-known communication optimizations: redundant communication removal, communication combination, and communication pipelining. In particular, each optimization is described in terms of how it improves performance. Though their descriptions are machine independent, the effectiveness of the optimizations can be influenced by machine specific characteristics. Consequently, we empirically evaluate these optimizations using four benchmark programs for two modern parallel machines, the Intel Paragon and the Cray T3D and two communication mechanisms, message passing and one-way communication (*i.e.*, T3D's SHMEM libraries). These benchmark programs are written in ZPL [17, 18], a portable data parallel array language similar to the array subset of Fortran 90. The compiler supports a instrumented compiler where optimizations are performed in a machine independent manner (ZPL source programs are compiled to SPMD ANSI C and linked with machine dependent libraries), allowing the same compiler output to be used for each set of experiments. Note that we are not assessing the effectiveness of the ZPL language or evaluating the compiler itself; the language semantics allow static detection of communication and thus no optimization opportunities are missed. Rather, we merely are looking to evaluate the *best case* behavior for a set of commonly used data parallel benchmark programs in the presence of standard communication optimizations. Thus the results of this study apply to parallel compilers of all forms – parallelizing compilers for sequential languages as well as compilers for parallel languages.

The paper is organized as follows. In Section 2, we review the goals of the three optimizations and comment on their effectiveness. In Section 3, we present an empirical evaluation of the optimiza-

*This research was supported by ARPA Grant N00014-92-J-1824

tions. Finally, in Section 4, we give directions for future work and conclusions.

2 Review of Optimizations

In this Section, we review three standard communication optimizations: redundant communication removal, communication combination, and communication pipelining. Our language context eliminates the need for *message vectorization*, the most common communication optimization where in communication of individual array elements is hoisted outside of a loop nest and combined into a single communication of a slice of the array. Parallelizing compilers for scalar languages such as Fortran 77 must perform message vectorization since the unit of representation, and thus the unit of communication, is a single scalar value. All respectable parallelizing compilers perform message vectorization. Compilers for array languages, on the other hand, can directly use arrays and array slices as the unit of representation [6], eliminating the need for message vectorization. Thus the baseline of comparison in our experiments will be optimization using only message vectorization.

Before describing the optimizations, we will briefly discuss the notation and assumptions. The examples in this section represent *single-program-multiple-data* (SPMD) code, though these optimizations can be applied to non-SPMD code. The notation is a simplified pseudo-code. For example,

$$A_{*,*} \leftarrow B_{*,*}$$

says that the array B is assigned to the array A for all index positions owned by a processor, *i.e.*, for all (i,j), $A_{i,j} \leftarrow B_{i,j}$. Similarly,

$$A_{*,*} \leftarrow B_{*,*+1}$$

says that the array B, shifted by one element in the second dimension, is assigned to the array A, *i.e.*, for all (i,j), $A_{i,j} \leftarrow B_{i,j+1}$. We assume that arrays are aligned and block distributed, therefore the statement above requires communication between neighboring processors. For convenience, we will assume the compiler generates message passing code by emitting message sends and receives, though this is a simplification of what the ZPL compiler does (we describe the actual implementation in Section 3.1). We omit the bounds information for the send and receive operations for clarity of presentation. We will also omit the source and destination of the send and receive operations. Recall that in SPMD code, for a given send/receive pair, a single processor is sending to processor p, while receiving from processor q where p and q are different processors, thus avoiding deadlock.

Figure 1(a) shows an example of naively generated communication. Notice that each non-local reference requires communication and the communication occurs immediately before the data is accessed. We now review the definition of the optimizations being considered and briefly describe the steps involved in their implementation.

Redundant communication removal. Communication is frequently not necessary because the non-local data has already been transmitted to the processor. Such *redundant* communication can be eliminated. Removing redundant communication reduces the number of messages sent and the volume of data sent. In Figure 1(a), the second communication of B is redundant and can be removed as in Figure 1(b). Specifically, if a processor has cached non-local data, a subsequent transfer of that same data is not necessary if the data has not been modified.

Communication combination. Several messages that are bound for the same processor may be *combined* into a single, larger message. Combining communication reduces the number of messages sent, but the volume of data sent remains the same. For example, in Figure 1(c), the communication for B and E will have the same source and destination processors and can therefore be combined.

Communication pipelining. Communication of messages may be *pipelined* such that the send is initiated earlier than the receive. This generally means that the receive is initiated immediately before the data is used while the send is initiated just after the last modification of the data. This optimization hides the communication latency by enabling computation to be performed during the data transfer. Pipelining does not affect the number of messages sent or the volume of data sent. Figure 1(d) illustrates an example of pipelined communication.

Notice that the goals of combining and pipelining are sometimes at odds. Specifically, combining may reduce the "distance" between sends and receives, where distance is a measure of how much of the exposed communication latency can be hidden by computation. Realize that this is no guarantee that the the latency will be hidden, as the communication mechanism on each target machine provide different opportunities. Figure 2 illustrates an example of two heuristics for combining communication in the presence of pipelining. When combining communication, a compiler may choose to *maximize combining* or *maximize latency hiding potential* or even a a hybrid solution based on machine *and application* characteristics. To maximize combining, messages are combined without regard for the distance between the send and receive (see Figure 2(b)). To maximize latency hiding, only messages that are completely nested are combined (see Figure 2(c)). In this way, messages are only

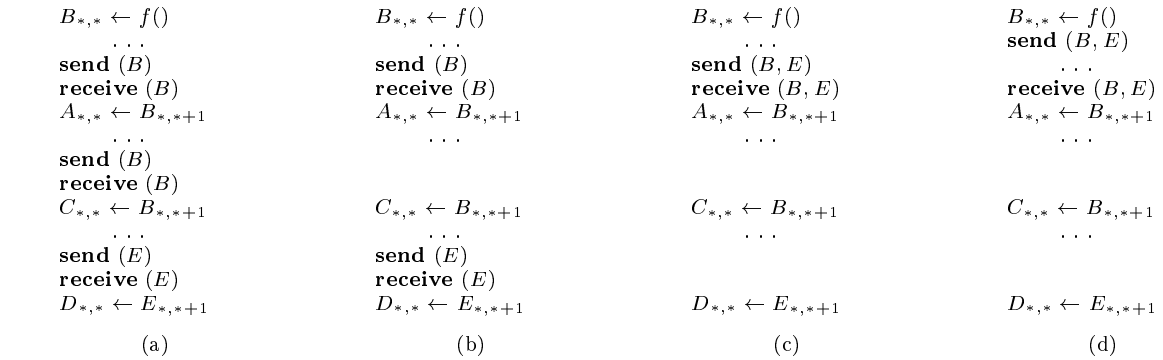


Figure 1: Example of communication optimizations: (a) naive communication generation, (b) redundant communication removal, (c) communication combination, (d) communication pipelining.

combined until the distance between the combined send and receives is no smaller than any of the distances of the uncombined communication. We will not discuss using a hybrid solution here, but tailoring communication combination for a particular machine and application is certainly worth investigating in the future.

3 Experimental Results

In this section, we present an evaluation of optimizations for communication generation. First, we discuss our methodology and the framework for evaluation. We then investigate the influence of machine characteristics on the optimizations using a synthetic benchmark program. Finally, we empirically evaluate the optimizations for four benchmark programs.

3.1 Methodology and Framework

Experiments were run on two platforms: the Intel Paragon [9] and the Cray T3D [3] (see Figure 3). On the Paragon, we use the native NX communication library routines. On the T3D we use a vendor optimized version of PVM [12] and the native SHMEM [3] library routines. The synthetic benchmark was run on two node dedicated partitions on the Paragon and the T3D. All benchmark programs were run on 64 node dedicated partitions on the T3D. Measured deviations were under 1% and therefore will not be reported with each experiment. All timings were taken using each machine's native timer.

The benchmark programs are written in ZPL, a portable data parallel array language developed at the University of Washington. ZPL can be used to solve a class of regular problems similar to those suitable for Fortran 90 and has been successfully used for real scientific and engineering applications

machine	*communication library*	*timer granularity*
Intel Paragon 50 MHz	NX (message passing)	~100 ns
Cray T3D 150 MHz	PVM (message passing) SHMEM (shared memory)	~150 ns

Figure 3: Machine parameters and communication libraries for the Paragon and the T3D.

[7, 11, 16, 20]. ZPL provides reductions, parallel prefix operators and other parallel operations, but for the purposes of this paper, we will concentrate on nearest-neighbor communication introduced by the shift operator, @. The language's semantics guarantee static detection of communication and allow us to concentrate on optimizations.

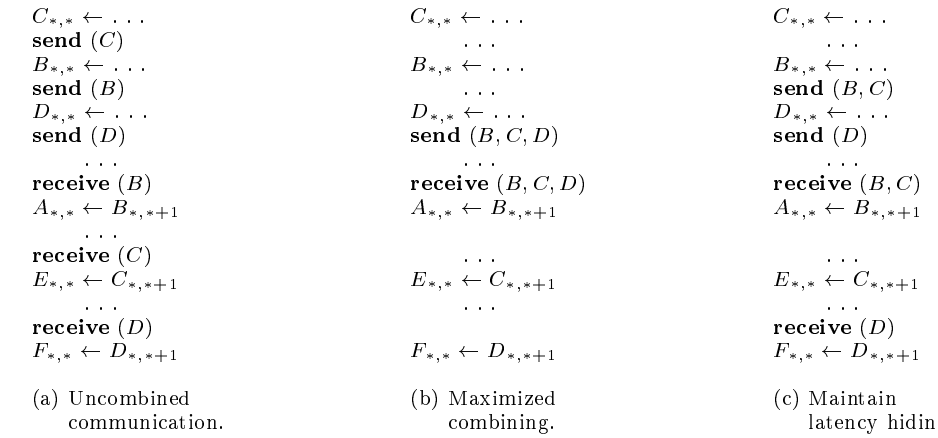
In ZPL, arrays are first class citizens. Operations are performed on whole arrays and indexing is not allowed.

```
A := 1.0;    -- assign the array A with 1.0
B := C;      -- assign the array B with
             -- corresponding elements of C
```

The @ operator allows shifted accesses to arrays. @ is a binary operator that take two arguments: an array operand (left operand) and a static offset vector (right operand). If A and B are two dimensional array, and east is the offset vector (0,1), then the following use of @ simply assigns all array elements of A with the corresponding elements of B, shifted by 1 in the second dimension.

```
A := B@east;  -- for all (i,j)
              --     A(i,j) ← B(i,j+1)
```

At runtime, all arrays are trivially aligned (*i.e.*, element (i,j) for all arrays resides on the same processor) and block distributed across a two dimensional



(a) Uncombined communication.

(b) Maximized combining.

(c) Maintain latency hiding.

Figure 2: Examples of combining communication: (a) communication of C, B, and D is pipelined; (b) all three communications are combined but the latency hiding ability is reduced; (c) only the communication of B and C are combined because doing so does not decrease the latency hiding ability.

virtual processor mesh. Consequently, the use of an @ implies the need for nearest neighbor communication. Figure 4 shows a code segment from the inner loop of the Tomcatv SPEC benchmark written in ZPL.

Our communication generation algorithm limits the scope of optimizations to a single *source-level* basic block, *i.e.*, a collection of whole array operations. For example, all the statements shown in Figure 4 are in the same basic block, bounded on one side by the start of the repeat loop, and therefore are considered at once for optimization. The analysis is much simpler than that for scalar languages where operations on arrays require loop nests and indexing. In our framework, the array statements are not expanded to loops nests until after communication generation, and as a result, they do not introduce control flow. This representation increases the granularity of analysis, thereby increasing the effectiveness of the algorithms. We now briefly explain the communication optimizations in the context of the ZPL compiler.

Redundant communication removal. Communication for @ expressions with the same array variable and same offset vector as a previous @ expression may be removed if the communication is redundant, *i.e.*, the required non-local values have not been modified since the communication. In the above example, the communication for X@east in line 9 is redundant because the non-local values have been cached earlier by the communication required for X@east in line 2.

Communication combination. Communication for @ expressions with the same offset vector but different array variable as a previous @ expression may be combined with that of the earlier expression if the neither array variable is modified after the communication is completed and before the data is used. All such communications have the same source and destination processors. For example, the communication for Y@east in line 3 may be combined with that of X@east in line 2.

Communication pipelining. Communication for @ expressions may be pipelined within a basic block by pushing the send operation of a communication up as far as the most recent modification of the required array values or the top of the basic block, whichever occurs later. For example, the send operation for X@se in line 9 can be initiated at the top of the repeat loop.

Though the compiler considers all optimizations simultaneously, the optimizations can be turned on and off individually. To isolate the effects of combining communication combining is maximized, unless otherwise noted.

The ZPL compiler generates machine independent SPMD ANSI C code. The C code is compiled using the target machine's native C compiler and linked with the ZPL runtime libraries to produce an executable. Programs written in ZPL may use any communication mechanism on any target machine with a single source compilation. This is achieved by compiling communication to the IRONMAN communication interface [5]. To make the paper self-contained, we will briefly discuss the interface here and refer the reader to the paper [5] for more details. A single data transfer is achieved by four li-

```
1  repeat
2     XX := X@east - X@west;
3     YX := Y@east - Y@west;
4     XY := X@south - X@north;
5     YY := Y@south - Y@north;

6     A := 0.250 * (XY * XY + YY * YY);
7     B := 0.250 * (XX * XX + YX * YX);
8     C := 0.125 * (XX * XY + YX * YY);

9     Rx := A*(X@east-2.0*X+X@west) + B*(X@south-2.0*X+X@north) - C*(X@se-X@ne-X@sw+X@nw);
10    Ry := A*(Y@east-2.0*Y+Y@west) + B*(Y@south-2.0*Y+Y@north) - C*(Y@se-Y@ne-Y@sw+Y@nw);

      ...
```

Figure 4: Code segment from Tomcatv SPEC benchmark written in ZPL.

brary calls: DR, SR, DN and SV. The calls themselves demarcate regions in the code where data transfer can occur. They are named for the state of the program on the source and destination processors: DR, destination ready to receive transmission; SR, source ready for transmission; DN, transmitted data needed at destination; and SV, transmission must be completed at the source since the data may become volatile. At link time, these calls are mapped to communication routines or no-ops on each platform. For example, when using the Paragon's csend/crecv, SR is mapped to csend, DN to crecv and DR and SV become no-ops.

```
DR(B,east);      -- this becomes a no-op
SR(B,east);      -- this call is mapped to csend
DN(B,east);      -- this call is mapped to crecv
SV(B,east);      -- this becomes a no-op
A := B@east;     -- this statement requires non-local values
```

Figure 5 describes the IRONMAN bindings for the Paragon and the T3D.

In the following sections, we use a synthetic benchmark to determine the machine characteristics that affect the optimizations and then perform whole program experiments on four benchmark programs.

3.2 Influence of Machine Characteristics

Our first experiment is simply to determine the machine dependent characteristics that affect the optimizations on the Paragon and the T3D. We measure the software overhead, *i.e.*, the *exposed* communication cost, for different communication primitives in our framework. On the Paragon, we use csend/crecv, basic message passing, isend/irecv, asynchronous message passing using the co-processor, and hsend/hrecv, message passing using callbacks. On the T3D, we use PVM for basic message passing and SHMEM for asynchronous shared memory operations.

Figure 6 shows the observed software overhead of communication in our framework. The synthetic benchmark program sends a message from one node to another 10000 times. Between any of the four parts that require communication, a busy loop is executed. The loop performs enough computation to hide the transmission time. The execution time of that loop is then subtracted from the total time.

The *knee* in the curves represent the message size for which combining messages begins to noticeably increase overhead. For both the Paragon and the T3D, the knee occurs at about 512 doubles (4K bytes). In other words, combining messages of 512 doubles or more does not improve performance. On these machines, combining messages smaller than 512 doubles is always better than sending several smaller messages.

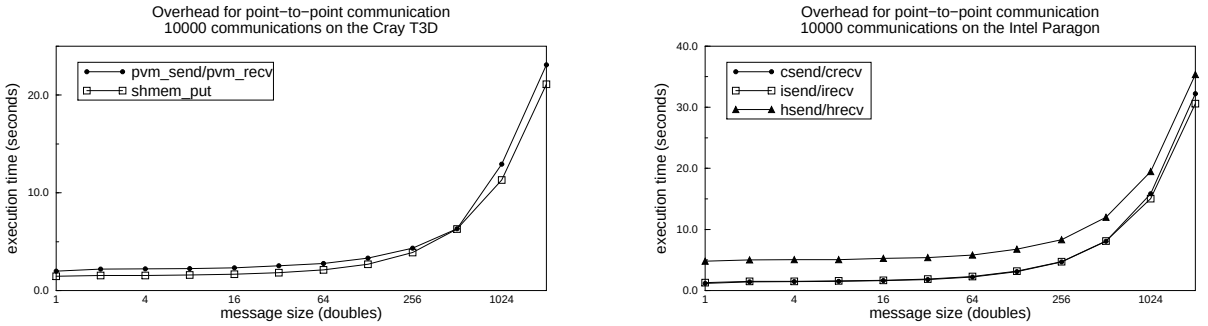
On the T3D, the SHMEM overhead is about 10% less than that of PVM. Though the difference is not as large as we hoped, this may be due to a limitation in our prototype implementation. In particular, the synchronizations are unnecessarily heavy-weight. We expect to see larger improvements with an optimized version currently under development. On the Paragon, we see that using the asynchronous primitives either does not reduce the exposed overhead, as in the case of isend/irecv, or increases it, as in case of hsend/hrecv. We also found that when we performed our full battery of test using the benchmark suite on the Paragon, the asynchronous primitives saw little performance improvement or, in most cases, performance degradation. Consequently, we will not present the Paragon results of experiments to follow.

3.3 Whole Program Experiments

For our experiments, we chose four benchmark programs (see Figure 7). Each benchmark is substantial and requires a significant amount of communication. The plotted numbers are scaled to our baseline: naive communication generation with message vectorization. Actual numbers are reported in Appendix A.

IRONMAN *interface*		*Intel Paragon*			*Cray T3D*	
program state	*call*	*message passing*	*asynchronous*	*callback*	*PVM*	*SHMEM*
destination ready	DR	no-op	irecv	hprobe	no-op	synch
source ready	SR	csend	isend	hsend	pvm_send	shmem_put
destination needed	DN	crecv	msgwait	hrecv	pvm_recv	synch
source volatile	SV	no-op	msgwait	msgwait	no-op	no-op

Figure 5: IRONMAN bindings on the Paragon and T3D.



(a) (b)

Figure 6: Exposed communication costs for various communication primitives on the Cray T3D and the Intel Paragon.

benchmark program	*description*	*line count*
TOMCATV	Thompson solver and grid generation (SPEC)	598
SWM	Weather prediction (shallow water model)	1570
SIMPLE	Hydrodynamics simulation (Livermore Labs)	2293
SP	CFD computation (NAS Application Benchmarks)	7866

Figure 7: Experimental benchmark programs. Line counts are given in terms of final output C code, excluding communication.

3.3.1 Effectiveness of Eliminating Communication

Figure 8 shows the reduction in the number of communications (where a communication refers to a set of calls to perform a single data transfer) due to eliminating communications. The static counts are simply the number of communications in the text of the SPMD program. The dynamic counts are the actual number of communications performed during the execution of the program on a single processor.

The dynamic counts achieve nearly the same reduction as the static counts. Statically, the number of communications is between 55% and 20% that of the baseline; dynamically, the number of communications is between 70% and 33% that of the baseline. This implies that most of the communication occurs within the *main loop* of the program. Redundant communication removal accounts for the majority of the static improvement, while dynamically, communication combination accounts for more of the reduction. This suggests that a significant portion of the redundant communication occurs in *set up* code while the combined communication primarily occurs within the main loop of the program.

3.3.2 Performance of Benchmark Programs

In this section, we present the performance results of running each benchmark program on a 64 node partition of the Cray T3D. Figure 9 provides a key for the experiments we performed. Each experiment adds an optimization. For example, "cc" is message vectorization with redundant communication removal *and* communication combination.

Performance using PVM. Figure 10(a) illustrates the reduction in execution times due to each optimization relative to the baseline. Execution times of fully optimized programs (pl) are as low as 72%

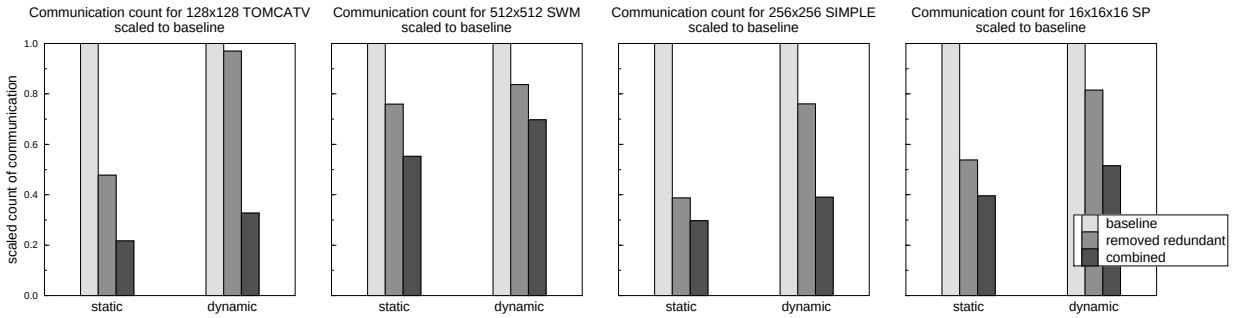


Figure 8: Reduction in the number communications due to redundant communication removal and communication combination. The static counts are the number of communications in the text of the program; the dynamic counts are the actual number of communications performed for our experiments.

experiment	*description*
baseline	message vectorization
rr	baseline with removing redundant
cc	rr with combing
pl	cc with pipelining
pl with shmem	pl using shmem_put
pl with max latency	pl with shmem, combining for maximum latency hiding

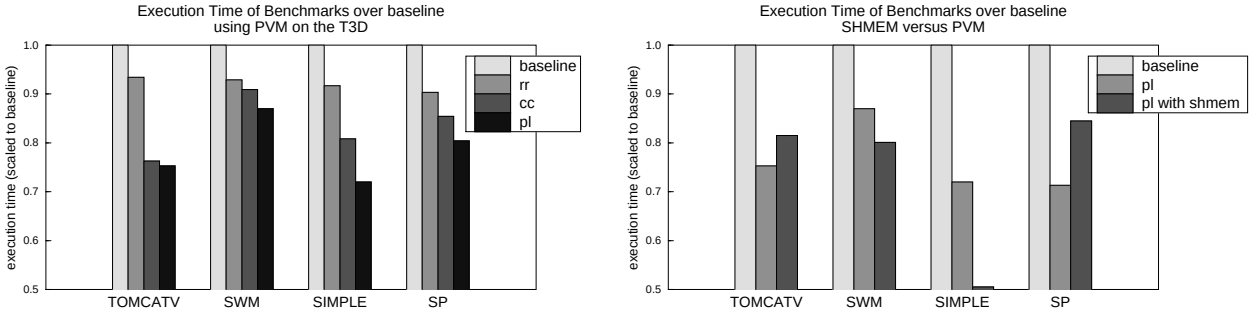
Figure 9: Key for experiments performed.

of that of the baseline. Eliminating communication alone, by redundant communication removal and communication combination (cc), reduced running times to as low as 76% that of the baseline. In the case of TOMCATV, pipelining affects performance very little. Examination of the code reveals that a large amount of time is spent in two small loops that implement a tri-diagonal solver. The opportunities for pipelining are limited by cross-loop dependences and the short code sequence itself. Compare this with the improvement for SIMPLE in which all communication occurs in the main body of the program. In general, each optimization impacts performance significantly.

Performance using SHMEM. Figure 10(b) illustrates the reduction in execution times of the fully optimized benchmark programs relative to the baseline using SHMEM's asynchronous shared memory primitives. The "pl with shmem" bar represents the performance of the same, fully optimized programs (pl) using shmem_put; for comparison, the "pl" bar is replicated from the Figure 10(a). For SWM and SIMPLE, performance is noticeably improved. The running time of SIMPLE is reduced to almost 50% that of the baseline. Recall that for SWM, the benefits of pipelining when using PVM are insignificant due to the limited space for exposing the communication latency. The reduced software overhead of shmem_put enables more of the latency to be hidden, resulting in a running time that is 80% that of the baseline.

Unfortunately, TOMCATV and SP experienced a degradation in performance. We have identified this as a limitation of our implementation, as mentioned above. The heavy-weight synchronization is particularly detrimental when parts of the computation are inherently sequential, as in TOMCATV and SP. The PVM version is highly optimized and the penalties for sequential computation are less severe. We expect to see the "pl with shmem" numbers drop below that of pl upon completion of the optimized version of the IRONMAN interface.

Comparing Combining Heuristics. In the previous section, we determined that the upper limit of message size for combining messages was 512 doubles. None of the above experiments combined messages as large as 512 doubles, so combining always improved performance. For those experiments, we use a heuristic that maximized combining. We now repeat the experiments for "pl with shmem" using the combining heuristic that maximizes latency hiding potential. Figure 11 shows the static and dynamic communication counts for the benchmarks when compiled using each of the combining heuristics. As expected, combining to maximize latency hiding potential does in some cases significantly increase the number of communications both statically and dynamically. For TOMCATV, the dynamic communication count is 97% that of the baseline, the same as for simply removing redundant communication (Figure 8). The only difference between the two versions is that the communication is pipelined. Figure 12 shows the scaled running times for this experiment. At runtime, the benchmark versions compiled for maximized combining



(a) (b)

Figure 10: Performance of optimized benchmark programs: (a) using PVM, (b) using SHMEM.

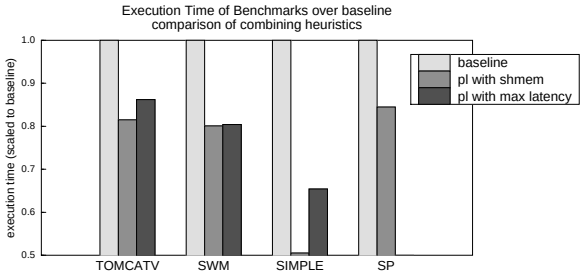


Figure 12: Comparison of combining heuristics. We were unable to run the "pl with max latency" version of SP due to a bug in the library code which will be fixed by the final paper.

always performed better than those compiled maximized latency hiding. Notice that the performance of TOMCATV when maximizing for latency hiding, effectively removing redundant communication and pipelining, is much better than that of simply removing redundant communication (Figure 10(a)), again showing that each optimization improves performance significantly.

4 Conclusions and Future Work

We have quantified the effectiveness of three communication optimizations: redundant communication removal, communication combination, and communication pipelining. We measured the software overheads on the Intel Paragon and the Cray T3D and found that combining message up to the size of 512 doubles (4K bytes) does not affect the overhead significantly. More importantly, the asynchronous communication primitives provided by Intel's NX libraries are extremely heavy-weight and showed no improvement over using the more traditional alternative, csend/crecv. T3D's asynchronous SHMEM operations show promise for performance improvement over the traditional message passing communication provided by PVM. The impact of each optimization is demonstrated using a suite of benchmark programs run on the T3D. We evaluated the effectiveness of each of the optimization and found that all three optimizations contribute significantly to decreasing running times versus the same benchmarks optimized using only message vectorization.

The instrumented ZPL compiler provides an excellent framework for evaluating sophisticated communication optimization. It is only natural to extend our study to include other communication optimizations, especially as performance bottlenecks shift. For example, we may want to employ a standard data flow analysis algorithm to apply optimizations across basic block boundaries. Also, we may want to *peel* off the iterations of a loop that access non-local data, hence exposing additional computation that may be overlapped with communication. Another simple method that may increase the opportunities for optimization is to use procedure inlining. Cooper *et al.* studied inlining in the context of scientific applications[8]. Though they found that it was almost always detrimental to performance, the presence of communication was not considered. In addition, we are currently investigating the interaction of communication optimizations with other array language optimizations, such as array contraction, and more traditional optimizations such as loop fusion. Finally, we plan to investigate the new issues that arise when machine specific characteristics can be incorporated into the compiler's optimization engine.

Acknowledgments. We would like to thank the ZPL compiler group at the University of Washing-

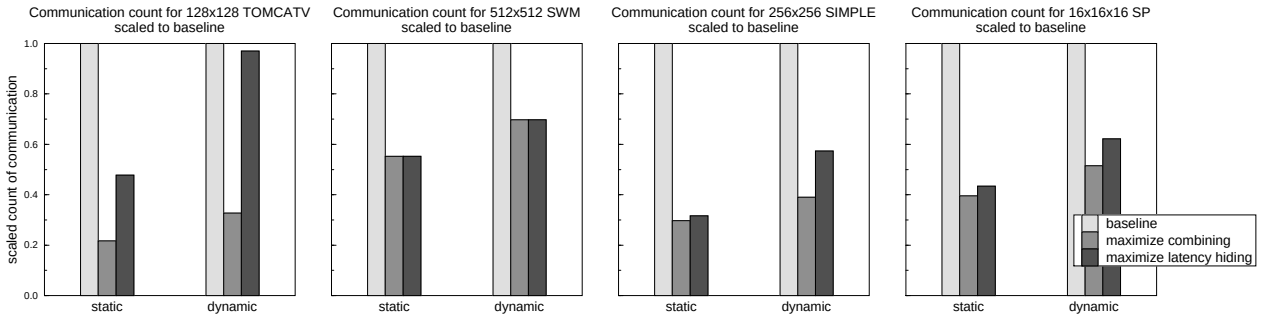


Figure 11: Reduction in the number communications due to different combining heuristics.

ton for their support of the compiler; Erin Aaron, Brad Chamberlain and E Lewis for additional encouragement; and the Arctic Region Supercomputing Center for allowing us access to their T3D.

References

[1] Gagan Agrawal and Joel Saltz. Interprocedural communication optimizations for distributed memory compilation. In *Workshop on Languages and Compilers for Parallel Computing*, pages 283–299, August 1994.

[2] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN'93 Conference on Program Language Design and Implementation*, June 1993.

[3] Ray Barriuso and Allan Knies. SHMEM user's guide for C. Technical report, Cray Research Inc., June 1994.

[4] Soumen Chakrabarti, Manish Gupta, and Jong-Deok Choi. Global communication analysis and optimization. In *SIGPLAN'96 Conference on Programming Language Design and Implementation*, 1996.

[5] Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. IRONMAN: An architecture indepedent communication interface for parallel computers. *submitted for publication*, 1996.

[6] Braford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. Factor-Join: A unique approach to compiling array languages for parallel machines. In *Workshop on Languages and Compilers for Parallel Computing*, August 1996.

[7] Sung-Eun Choi, Lawrence Snyder, Joachim Stadel, and Tom Quinn. A portable parallel planetary integrator. *in preparation*, September 1996.

[8] Keith D. Cooper, Mary W. Hall, and Linda Torczon. Unexpected side effects of inline substitution: A case study. *ACM Letters on Programming Languages and Systems*, 1(1):22–32, March 1992.

[9] Intel Corporation. *Paragon User's Guide*. 1993.

[10] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–478, September 1994.

[11] Marios D. Dikaiakos, Calvin Lin, Daphne Manoussaki, and Diana E. Woodward. The portable parallel implementation of two novel mathematical biology algorithms in ZPL. In *9^{th} International Conference on Supercomputing*, 1995.

[12] A. Belguelin et al. A user's guide to PVM. Technical report, Oak Ridge National Laboratories, 1991.

[13] Manish Gupta, Edith Schonberg, and Harini Srinivasan. A unified data-flow framework for optimizing communication. In *Workshop on Languages and Compilers for Parallel Computing*, pages 266–282, August 1994.

[14] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Evaluating compiler optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21(1):27–45, April 1994.

[15] Ken Kennedy and Nenad Nedeljković. Combining dependence and data-flow analyses to optimize communication. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 340–346, April 1995.

[16] E Christopher Lewis, Calvin Lin, Lawrence Snyder, and George Turkiyyah. A portable parallel n-body solver. In D. Bailey, P. Bjorstad, J. Gilbert, M. Mascagni, R. Schreiber, H. Simon, V. Torczon, and L. Watson, editors, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 331–336. SIAM, 1995.

[17] Calvin Lin. ZPL language reference manual. Technical Report 94–10–06, Department of Computer Science and Engineering, University of Washington, 1994.

[18] Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Workshop on Languages and Compilers for Parallel Computing*, pages 96–114. Springer-Verlag, 1993.

[19] Daniel J. Palermo, Ernesto Su, John A. Chandy, and Prithviraj Banerjee. Communication optimizations used in the PARADIGM compiler for distributed-memory multicomputers. In *International Conference on Paralle Processing*, pages II:1–10, August 1994.

[20] George Wilkey Richardson. Evaluation of a parallel chaos router simulator. Master's thesis, The University of Arizona, Department of Electrical and Computer Engineering, 1995.

[21] Reinhard von Hanxleden and Ken Kennedy. GIVE-N-TAKE – a balanced code placement framework. In *SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 107–120, June 1994.

A Experimental Results

experiment	*static count*	*dynamic count*	*execution time*
baseline	46	40400	2.491051
rr	22	39200	2.327301
cc	10	13200	1.901393
pl	10	13200	1.875820
pl with shmem	10	13200	2.029861
pl with max latency	22	39200	2.148066

Table 1: Results for 128x128 tomcatv on 64 processors.

experiment	*static count*	*dynamic count*	*execution time*
baseline	29	8602	6.809007
rr	22	7202	6.323369
cc	16	6002	6.191816
pl	16	6002	5.922135
pl with shmem	16	6002	5.454957
pl with max latency	16	6002	5.477305

Table 2: Results for 512x512 swm on 64 processors.

experiment	*static count*	*dynamic count*	*execution time*
baseline	266	28188	66.749756
rr	103	21433	61.193568
cc	79	10993	53.962579
pl	79	10993	48.077192
pl with shmem	79	10993	33.720775
pl with max latency	84	16143	43.637907

Table 3: Results for 256x256 simple on 64 processors.

experiment	*static count*	*dynamic count*	*execution time*
baseline	212	85982	22.572110
rr	114	70094	20.381131
cc	84	44286	19.274767
pl	84	44286	18.149760
pl with shmem	84	44286	19.079338
pl with max latency	92	53487	

Table 4: Results for 16x16x16 sp on 64 processors.