Delta Send-Recv: Compiler and Run-time Support for Dynamic Pipelining of Coarse-grained Computation and Communication

Bin Bao, Chen Ding
University of Rochester
bao, cding@cs.rochester.edu

Yaoqing Gao, and Roch Archambault
IBM Toronto Software Lab
ygao, archie@ca.ibm.com

Abstract

This paper presents compiler and run-time support for improving MPI programs that use coarse-grained computation and communication. It uses compiler or virtual-memory support to divide computation and communication into pieces we call "deltas" and interleave them. Delta-send overlaps data production and data send. Delta-receive, previously known as early release, overlaps data receive and data use. When used together in a program, the two primitives are dynamically chained to produce the effect of sender-receiver pipelining and can gain more than the maximal speedup possible with just sender or just receiver side overlapping. The paper presents a design and an analysis of delta send-recv and measures the effect of such dynamic pipelining on a set of kernel tests with some contrived computation and representative communication patterns including pairwise and collective communication.

1. Introduction

Computation and communication overlapping is a basic method in improving the performance of distributed code. Non-blocking send and receive permits overlapping between communication and independent computation. We present an extension to MPI non-blocking send-recv called *delta send-recv*. Using compiler and virtual-memory paging support, delta send-recv divides a data message and its computation into pieces that we call *deltas* or increments. The communication starts as soon as the first increment is computed, and the communication of early increments is overlapped with the computation of later ones. Similar to*early release* [4], which begins the use of data as soon as a page has arrived, delta-recv begins the use of data when the increment is received.

The most important benefit comes from combining incremental send and receive, where dependent data is being computed, communicated, and consumed in a distributed processing pipeline. For coarse-grained computation and communication, the pipelining by delta send-recv greatly increases the amount of parallelism between the sender and the receiver and tolerates the communication cost. In addition, the pipelining of successive delta sender-receiver pairs is dynamically chained to produce a cascading effect, in which dependent computation in all involved tasks proceeds in parallel. As a result, delta send-recv is capable of performance improvement linear to the number of tasks. In comparison, non-blocking communication can improve performance by at most a constant factor.

Compiler analysis has been studied for explicitly parallel programs, including dataflow analysis [2, 5, 8, 9] and abstract interpretation [3]. Previous compiler research has used loop strip-mining and tiling to enable similar sender-receiver pipelining [6, 10]. Automatic transformation requires precise send-receive matching. Recent advances including the notion of task groups [2] and techniques for matching textually unaligned barriers [11]. However, static send-receive matching is not yet fully solved in a general system with a dynamically created tasks. In comparison, the pipelining by delta send-recv is formed dynamically without the need for static send-receive matching. It is easier to use since it needs program analysis and transformation only at the sender side.

[copyright notice will appear here]

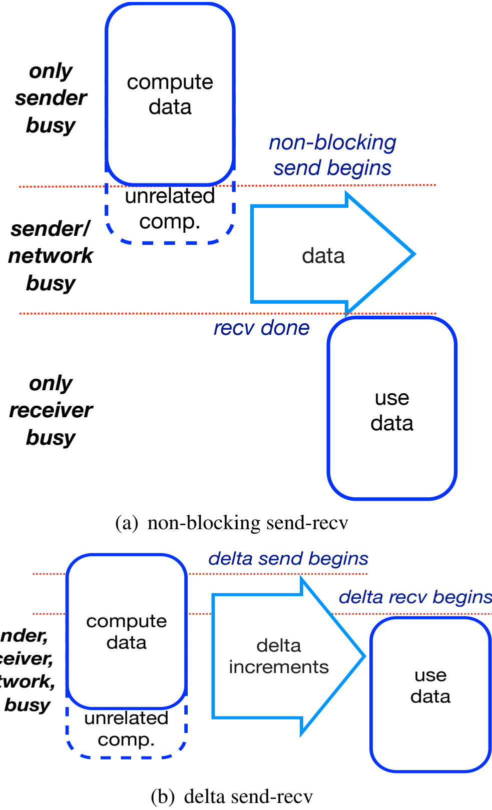
2. Dynamic Pipelining Through Incremental Communication

Delta send-recv provides a simple extension to the MPI interface. MPI_Delta_send_begin is called before the computation to the send buffer, which indicates the runtime system that the following computation will consecutively write to the send buffer, and then MPI_Delta_send_end marks the end of such computation. MPI_Delta_wait is used to finish our non-blocking style delta send, similar to its counterpart in the MPI non-blocking send. For the delta-receive operation, we only provide one function, called MPI_Delta_recv. Although the delta-receive is non-blocking, we can save the corresponding wait operation because the wait is implicit at the use (or def) point of the received data.

With the page protection mechanic, MPI_Delta_send_begin can monitor that computation. First it puts the send buffer under page protection. When a later write to the send buffer triggers a page fault, the signal handler can determine which *delta* the computation is trying to write to, non-blockingly sends out the previous *delta* if the current one is not the first *delta*, and then opens the page protection to the current *delta* to allow the computation writes data into it. MPI_Delta_send_end is used to send out the last *delta* since no page fault will be triggered from send buffer after it. MPI_Delat_wait waits for all issued non-blocking sendings of *delta*.

An important fact that guarantees the correctness of our delta-send is the computation consecutively writes data into the send buffer. The compiler can analyze whether such computation pattern exists, and inserts the MPI_Delta_send_begin right before the computation.

Similar to early release in [4], MPI_Delta_recv first turns on the page protection to the receive buffer and then let program continue its execution. While early release uses alias memory pages, we create a duplication of the receive buffer to perform the backgroud receiving. Once the system sees a page fault from an access to the receive buffer, it can figure out which *delta* the access belongs to,



(a) non-blocking send-recv

(b) delta send-recv

Figure 1. Comparison between the effect of non-blocking send-recv and delta send-recv in a parallel execution.

and then wait for its corresponding receive to the duplicate buffer to finish. When that receive finishes, the system copies the received *delta* from the duplicate buffer to the acture receive buffer. Early release incurs a page fault for every received page. Delta-recv, like delta-send, is parameterized by the delta size Δx and incurs a page fault every Δx pages.

Dynamic pipelining When used together, delta send and receive enable the parallelism between sender and receiver computation. We call this *dynamic pipelining*. The pipelining effect is illustrated by an example in Figure 1. In the example, the sender and the receiver have dependent computations on a set of data. The sender also has some independent computation. Figure 1(a) shows the effect of non-blocking communication, which overlaps the independent computation and the communication.

The execution of delta send-recv is shown in Figure 1(b). The sender begins to execute. After computing the first increment, the communication starts. The receiver starts once the first increment is arrived. At this time, the sender, the network, and the receiver are all running in parallel. The process is a form of 3-stage pipelining where data of each increment is computed, transferred, and used. In addition to enabling parallel execution of dependent computations, the use of incremental send may improve network efficiency. If it takes a program longer time to compute an increment than to communicate one, the communication calls will be spaced out, and the lower rate of communication may alleviate network congestion if there is any.

Pipeline chaining Delta send-recv may be chained together to enable parallelism between more than two tasks. If we extend the example in Figure 1 such that when it finishes processing, the second task sends the data to a third task. The same type of pipelining happens between them in the exact same fashion as the pipelining between the first two tasks. Given enough computation, all three tasks will execute in parallel after an initial period.

The benefit of chaining is important for MPI aggregate communication such as broadcast and reduce. Such communication is often carried out on a tree topology so it takes $O(\log n)$ steps to reach n tasks. Chaining would enable parallelism between these steps.

Overhead There are several sources of overhead when using delta send-recv. It incurs a higher cost because it transfers a series of messages instead of a single message. It includes processor time in setting up and receiving a message and communication overhead in storing its meta data and acknowledging its status. Delta send-recv must be non-blocking, which can be more costly to implement than blocking communication because of the need to manage simultaneous transfers. If paging support is used to monitor program data access, there will be the cost of one page fault when sending and the receiving each increment.

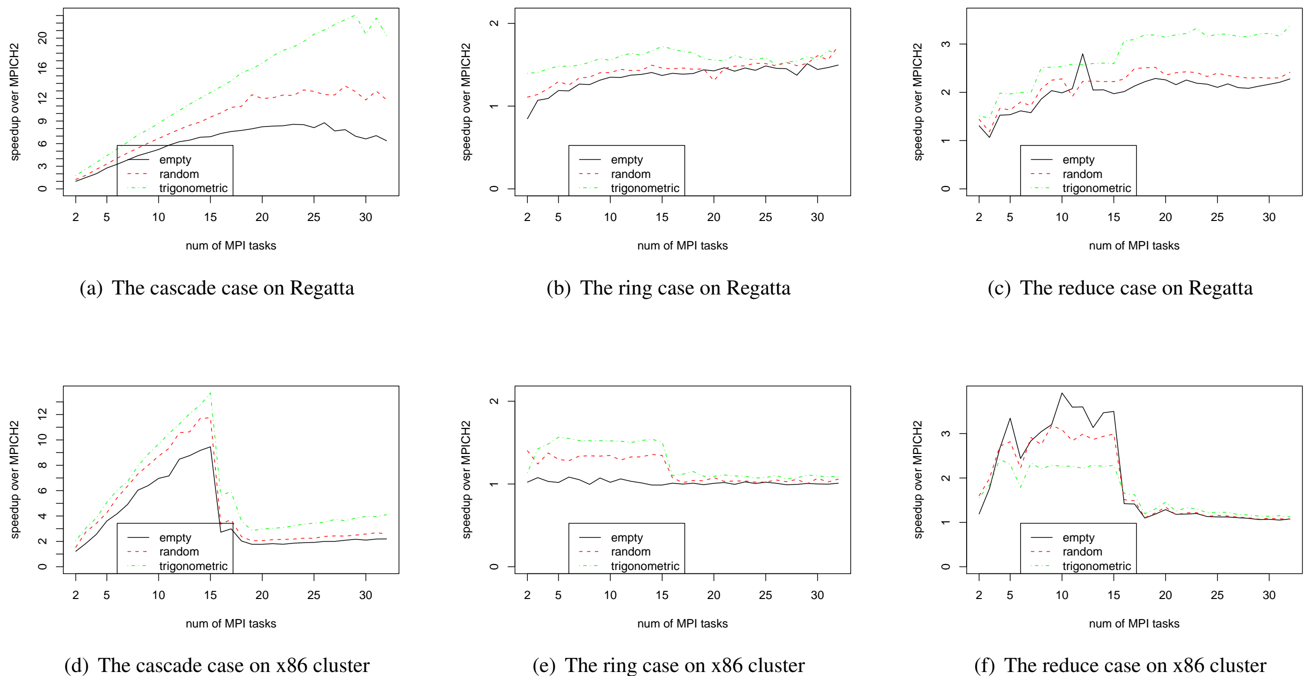
3. Evaluation

We use two machines with different characteristics to evaluate delta send-recv. The first is a 32-processor IBM p690 "Regatta" multiprocessor machine. The second is an Ethernet switched homogeneous cluster with over 40 PC nodes. We use MPICH2 1.2.1 [7] on both machines.

We use four kernel benchmarks that can represent common communication patterns in distributed applications. The test *pair* has two processes that sender computes and then sends an array of data, while the receiver receives the data and does the same computation to check them. *Cascade* is a series of send-recv pairs, where each node receives the data from its left neighbor, checks them, and then pass them to its right neighbor. *Ring* performs communication in a virtual ring topology. Unlike cascade, there is no dependence between tasks in their computation. *Reduce* contains a set of processes are organized in a virtual tree topology. The reduce operation is performed on each element of an array, and finally the root process holds an array in which each element is the sum of values from all other processes. For each of the four tests, we alter the amount of computation using the following three types, *empty*, *random*, and *trigonometric*.

We have studied the effect of different delta sizes and send-recv sizes in [1]. In general, the best increment size is within a range, from 3 to 8 memory pages. And the granularity for delta send-recv to be beneficial is 10 pages (40KB). Figure 2 shows the speedup of using delta send-recv over using the normal MPI send and receive when running a different number of MPI tasks (processes). The results are collected with setting the data array to one million element (4MB) and the delta size to 5 memory pages.

Figures 2(a) and 2(d) show the performance of the cascade benchmark measured on the Regatta system and the cluster system respectively. In figure 2(a), the improvement increases proportionally as we increase the number of tasks. This confirms our analysis in Section 2 that pipeline chaining can achieve arbitrarily high speedup over base MPI implementation. The improvement boosts as the amount of computation increases, since having more computation leaves more room for overlapping communication. The limiting factor on both machines is communication bandwidth. On Regatta, the bandwidth appears saturated at 27 tasks for empty computation, 29 for random, and 30 for trigonometric. On the cluster, the saturation happens at 16 tasks in all three cases. The other four graphs in Figure 2 show same saturation point in *ring* and *reduce* tests on the cluster. The bandwidth on Regatta seems sufficient in these two tests.



(a) The cascade case on Regatta

(b) The ring case on Regatta

(c) The reduce case on Regatta

(d) The cascade case on x86 cluster

(e) The ring case on x86 cluster

(f) The reduce case on x86 cluster

Figure 2. Performance of delta send-recv over the number of MPI tasks

The improvement for the *ring* test increases with the number of tasks to around 50% in all three computation types on Regatta, as shown by Figure 2(b). On the cluster, the improvement is constant for random and trigonometric, although the empty computation shows no improvement, as shown by Figure 2(e). There is no cascading in this case. The improvement comes from the higher efficiency between each pair of neighbors, compared to non-blocking send-recv. The improvement for the *reduce* test increases at a logarithmic scale, as shown by Figures 2(c) and 2(f). The reason is that the length of the longest dependent task chain in k-task reduce is $\log_2 k$.

4. Summary

We have presented delta send-recv, an extension to MPI non-blocking send-recv, for overlapping dependent computation with communication, using compiler and operating system support. It enables dynamic pipelining and pipeline chaining and requires program analysis and transformation only at the sender side. Empirical evaluation on an SMP multi-processor and a PC cluster shows that delta send-recv consistently improves parallel performance for coarse-grained send-recv. The largest improvement comes from pipeline chaining, which has led to up to 23 times faster running time on 29 MPI tasks.

References

[1] B. Bao, T. Bai, C. Ding, Y. Gao, and R. Archambalt. Delta send-recv: Compiler and run-time support for dynamic pipelining of coarse-grained computation and communication. Technical Report URCS #953, Department of Computer Science, University of Rochester, 2010. *to appear*.

[2] G. Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2009.

[3] C. Colby. Analyzing the communication topology of concurrent programs. In *Proceedings of the SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 202–213, 1995.

[4] J. Ke, M. Burtscher, and E. Speight. Tolerating message latency throuth the early release of blocked receives. In *Proceedings of the Euro-Par Conference*, 2005.

[5] J. Lee, D. A. Padua, and S. P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1999.

[6] J. M. Mellor-Crummey, V. S. Adve, B. Broom, D. G. Chavarría-Miranda, R. J. Fowler, G. Jin, K. Kennedy, and Q. Yi. Advanced optimization strategies in the rice dhpf compiler. *Concurrency and Computation: Practice and Experience*, 14(8-9):741–767, 2002.

[7] Mpich2: an implementation of the message-passing interface (MPI). version 1.0.5 released on December 13, 2006, available at http://www-unix.mcs.anl.gov/mpi/mpich/.

[8] J. H. Reif and S. A. Smolka. Data flow analysis of distributed communicating processes. *International Journal of Parallel Programming*, 19(1):1–30, 1990.

[9] M. M. Strout, B. Kreaseck, and P. D. Hovland. Data-flow analysis for mpi programs. In *Proceedings of the International Conference on Parallel Processing*, pages 175–184, 2006.

[10] A. Wakatani and M. Wolfe. A new approach to array redistribution: Strip mining redistribution. In *Proceedings of PARLE*, pages 323–335, 1994.

[11] Y. Zhang and E. Duesterwald. Barrier matching for programs with textually unaligned barriers. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 194–204, 2007.