Delta Send-Recv for Dynamic Pipelining in MPI Programs

Bin Bao, Chen Ding
Department of Computer Science,
University of Rochester
Rochester, NY, USA
{*bao, cding*}*@cs.rochester.edu*

Yaoqing Gao, Roch Archambault
IBM Toronto Software Lab
Markham, ON, Canada
{*ygao, archie*}*@ca.ibm.com*

***Abstract*—Pipelining is necessary for efficient do-across parallelism but the use is difficult to automate because it requires send-receive analysis and loop blocking in both sender and receiver code. The blocking factor is statically chosen.**

This paper presents a new interface called delta send-recv. Through compiler and run-time support, it enables dynamic pipelining. In program code, the interface is used to mark the related computation and communication. There is no need to restructure the computation code or compose multiple messages. At run time, the message size is dynamically determined, and multiple pipelines are chained among all tasks that participate in the delta communication. The new system is tested on kernel and reduced NAS benchmarks to show that it simplifies message-passing programming and improves program performance.

***Keywords*-MPI; communication-computation overlapping; dynamic pipelining**

I. Introduction

Computation and communication overlapping is a basic method in optimizing distributed programs. A straightforward way is non-blocking send and receive, which overlaps communication with unrelated computation. For dependent computation that either produces the outgoing message or consumes the incoming data, overlapping still can be done in a finer grain, known as pipelining. Pipelining cannot be easily automated for complex code because it requires exact send-receive pairing to perform matching transformations in both the sender and the receiver code. Manual transformation, on the other hand, makes code harder to understand and maintain. In addition, a static solution is not sufficient if the send-receive relation is not completely known at compile time.

In this paper, we present *delta send-recv*, an extension of the MPI send/receive interface and its run-time support. It divides a data message at run time into pieces which we call *deltas* or increments. On the sender side, the communication starts as soon as the first increment is computed. On the receiver side, the data can be used as soon as the first increment arrives. Delta receive is similar to early release [1]. When combined with delta send, it forms pipelining dynamically. Multiple senders and receivers may be dynamically chained to produce cascading in a task group, improving performance by a factor linear to the number of tasks.

For example, delta send-recv supports efficient coarse-grained reduce. Pipeline cascading reduces the cost by $O(\log k)$ for k tasks in a tree topology, compared to using MPI non-blocking send-recv.

In terms of programmability, delta send-recv enables pipelining without having to reorganize the computation code. It supports variable-size communication, where the size is unknown until the complete message is generated. In addition, it can be implemented using virtual-memory support which does not need access to program source code. It allows communication optimization for interpreted languages such as Matlab and R, whose use of separately compiled or dynamically loaded libraries makes manual transformation impractical.

The current design has several shortcomings.

- Pipelining is useful only when the amount of dependent computation is significant.
- The interface is applicable only when the computation follows the certain pattern, i.e. computation writes to the send buffer only once.
- The run-time support incurs overheads, which include the cost of monitoring message data usage and sending and receiving multiple messages. We will study the costs both analytically and empirically.

The main contributions of the paper are as follows:

- A new interface for delta-send/recv for pipelining communication and dependent computation.
- Two implementation schemes based on compiler and the OS support respectively and a common run-time system.
- Evaluation of the benefits and overheads. The benefits include both performance and programmability.

The rest of the paper is organized as follows. Section II describes the interface and the implementation. In Section III, we evaluate performance using three sets of tests mainly on a PC cluster and show the effects of the amount of computation, the cost of communication, the overhead of delta send-recv, and the minimal size needed for a delta increment. Finally, we discuss related work in Section IV

```
MPI_Delta_send_begin( ... );

/* dependent computation */
... ...
MPI_Delta_send_end( ... );

/* independent computation */
... ...
MPI_Delta_wait( ... );
```

(a) Delta-send to overlap both dependent and independent computation with the receive

```
MPI_Delta_recv( ... );

/* independent computation */
... ...

/* dependent computation */
... ...
```

(b) Delta-recv to overlap both dependent and independent computation with the receive

```
/* dependent computation */
... ...
MPI_Isend( ... );

/* independent computation */
... ...
MPI_Wait( ... );
```

(c) Non-blocking send to overlap independent computation with the send

```
MPI_Irecv( ... );
/* independent computation */
... ...

MPI_Wait( ... );
/* dependent computation */
... ...
```

(d) Non-blocking recv to overlap independent computation with the receive

Figure 1. The delta communication primitives used to pipeline communication and dependent computation. It subsumes the use of non-blocking send and receive, also shown in the figure for comparison, to overlap communication with independent computation. Delta send needs one more call than non-blocking send. Delta receive uses one fewer call than non-blocking receive.

and summarize in Section V.

II. Delta Send-Recv

A. The Interface

We show the interface by the use of the communication primitives in the context of computation. The computation has two parts: the dependent computation that produces and consumes the communicated data and the independent computation that does not use the communicated data.

On the sender side, as in Figure 1(a) shows, MPI_Delta_send_begin is called before the dependent computation that produces the sent message. It has the same parameters as a normal MPI nonblocking send, which includes 7 parameters including the buffer address, size, data type, destination task, tag, MPI communicator, and request handle. MPI_Delta_send_end is called after the dependent computation. It has just one parameter, which is the request handle. MPI_Delta_wait is called after the independent computation, taking the request handle and returning a status pointer. MPI_Delta_wait implies MPI_Delta_send_end, so the latter can be omitted if there is no independent computation before the wait.

The purpose of these primitives is to enable pipelining by the sender, where a chunk, i.e. a delta, of message is produced and sent while the next chunk is being computed. The production of the message data may be sequential or not depending on the implementation which we will describe next.

On the receiver side, as Figure 1(b) shows, MPI_Delta_recv is called before the message data is used. The parameter list is the the same as a MPI_Recv, including the buffer address, size, data type, source task, tag, MPI communicator, and status. Data may be received out of order. There is no need for an MPI_Wait.

Figure 1(c,d) show the use of non-blocking send/receive as a comparison. Non-blocking communication can overlap communication with independent computation. MPI_Delta_wait is like MPI_Wait for a non-blocking send. It blocks the sender until the message has been delivered. MPI_Delta_send_begin and MPI_Delta_recv take the same list of parameters, but they are placed before (rather than after) the dependent computation. As a result, delta-send/receive combines goes one step further and overlaps the communication with both the dependent computation and the independent computation.

The run-time effect is illustrated by an example in Figure 2. The left-side figure shows the effect of non-blocking communication, which overlaps the independent computation and communication. The right-side figure shows that delta send and receive improves processor utilization by pipeline parallelism and network utilization by incremental communication.

B. The Implementation

The implementation has two parts. The first is access monitoring. The second is dynamic messaging. In the general case, the message data can be produced and consumed in any order. To simplify the presentation, we first show a limited design which assumes the sequential order and then discuss the extensions needed to remove the limitation.

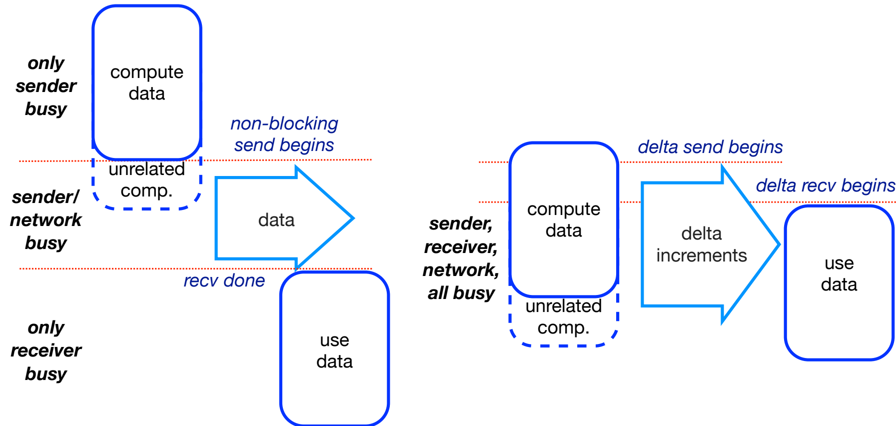


Figure 2. (Left) MPI non-blocking send-recv cannot overlap communication with dependent computation. (Right) Delta send-recv enables pipelining between the sender and the receiver and better utilizes the processors and the network.

```
comment: sent, requested are two integers initialized to 0
proc delta_send_data(s) ≡
  comment: Adding another s bytes to the message
  requested = requested + s
  if requested > deltasize
    then MPI_Isend(sent, requested − sent)
  fi
end
```

```
comment: received, nextmsg are two integers initialized to 0
proc delta_wait_data(base, size) ≡
  comment: Waiting for message data at base for size bytes
  while base + size > received do
        MPI_Wait(nextmsg)
        s = MPI_getsize(nextmsg)
        received = received + s
        nextmsg = nextmsg + 1
  od
end
```

Figure 3. Simplified algorithms for the delta send/receive. Access monitoring is done by calling these two functions, which instigates dynamic messaging.

As the sender computes the message data, it calls delta_send_data when it finishes computing a piece of the data. The pseudo code is shown in Figure 3. The function waits until the finished pieces amount to a threshold, $deltasize$, and sends these data in a message. The code shows the connection between access monitoring, done by calling delta_send_data, and dynamic messaging, done by executing the function. The message size can be determined and adjusted at run time.

On the receiver side, the program calls delta_wait_data before it uses a piece of message data. The pseudo code is also shown in Figure 3. The function keeps waiting for the next increment until the waited data has arrived.

The send/wait functions in Figure 3 form the core of the run-time support. The interface calls are implemented based on them. The delta send/receive calls are used to initialize the internal parameters needed by the send/wait. It is possible that a sender does not write the entire message. MPI_delta_send_end is used to inform the run-time system that there will be no more calls to delta_send_data. The run-time system then sends all the remaining data, if any, in one (last) message. MPI_delta_wait is a blocking operation to ensure that all delta sends are finished.

Delta send-recv may be implemented inside an MPI library or as our prototype be built as a user-level library over the standard MPI interface. Like standard MPI, its interface can be used by C/C++/Fortran programs. Next we describe two solutions for access monitoring.

1) OS Triggering: Virtual memory support can be used to monitor data access at the sender and the receiver.

Delta-send: MPI_Delta_send_begin places the send buffer under page protection except for the first *delta* increment. A *delta* is a group of consecutive memory pages. The operation installs a custom page fault handler. When a write to the send buffer triggers a page fault, it invokes the signal handler. Since the fault address signals the completion of the previous *delta*, it takes the address range and calls delta_send_data (shown in Figure 3), which initiates dynamic messaging. The handler unprotects the pages in the next *delta*, and resumes the execution of the sender. In this fashion, the page fault handler sends all *delta*s except for the last one, which is sent out when the sender reaches MPI_Delta_send_end. Finally, MPI_Delta_wait waits for all (non-blocking) *delta* sends to finish. To be correct, delta-send requires sequential write, which has to be guaranteed by the user or compiler analysis.

Delta-recv: MPI_Delta_recv turns on the page protection for the receive buffer, creates a shadow buffer of the same size, and then lets the receiver task continue its execution. When the receiver accesses the message data, it

incurs a page fault if the page containing the data is not yet received. The page fault handler calls delta_wait_data and blocks until the page (or pages if the delta size is more than a page) is received. The handler copies the pages from the shadow to the receive buffer, unprotects them, and resumes the receiver. The receiver can access data in any order in the receiver buffer, not just in sequential order. Furthermore, there is no need for a wait operation to follow the delta receive. Any message data not accessed are not needed (by the program).

The receive process is similar to early release developed by Ke et al. [1] It protects the receive buffer and "releases" the receiver task to continue to execute. Early release uses alias memory pages, while delta recv creates a shadow receive buffer to permit receiving in the background. Early release incurs a page fault for every received page. Delta-recv, like delta-send, is parameterized by the delta size. As we will show later in Section III-C, the overhead is substantially lower when the delta size is 4 or 5 pages rather than 1 page.

2) Compiler annotation: A compiler can insert delta send and wait calls directly instead of leveraging page protection. The sender can write to the message data in any order, and there is no paging overhead.

Delta-send: The compiler analyzes the computation between MPI_Delta_send_begin and MPI_Delta_wait to identify the dependent computation and annotate the write statements by calling delta_send_data. Standard dependence analysis can be used [2]. If the write to a send buffer happens in a tight innermost loop, direct annotation will incur a high run-time cost. The compiler can strip-mine the loop and insert the annotation in the outer loop to amortize the cost.

Our current delta_send_data function uses a range tree data structure to merge data ranges. If a new range is merged with existing ones, and the combined size exceeds the threshold, the function calls MPI_Isend to send out the data chunk.

Delta-recv: Similar compiler analysis and loop transformation can be performed on the receiver code following MPI_Delta_recv and insert calls to delta_wait_data (shown in Figure 3). Initially, the receiver posts non-blocking receives for the maximal number of delta messages and each of them receives data into its own shadow buffer. Since the actual delta size may be larger, and the specified and actual size of the communication may differ, the receiver cancels all remaining receives by calling MPI_Cancel after the last delta message (marked by the sender) has arrived.

In the general case, the message is managed and communicated as a set of data sub-ranges. Each call of delta_send/wait_data adds a sub-range. Dynamic messaging allows sub-ranges to be sent and received out of order. The bookkeeping of these data sub-ranges can help to detect misuse of delta send where a program may write to a memory location that has already been sent. The run-time support can abort the program and notify the user.

C. Dynamic Pipelining

Delta send-recv forms computation-communication pipelines *dynamically*, which has benefits and overheads.

Send-receive de-coupling and one-sided transformation: Delta-send can be implemented in a way that the message can be properly received by any type of receive as non-blocking sends can. Similarly, delta-recv can support messages from any type of sends.

Since there is no need to transform send-recv pairs in tandem, a user can optimize sender and receiver code separately. We call it a one-sided transformation. A user can transform an MPI send without knowing all the possible matching receives. As a result, one-sided transformation may improve performance more than previously possible or reduce the amount of programming time.

Another benefit is adaptive control. For example, based on the amount of computation, the size of message, and the running environment, delta-send-recv can dynamically choose different increment sizes to maximize performance.

Cascading: Delta send-recv may be chained together between more than two tasks. If we extend the example in Figure 2 such that when it finishes processing, the second task sends the data to a third task. Then the second and the third tasks form a pipeline in the same fashion as the first two tasks do. With 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. Cascading happens between all tasks on the same path from the root.

Overheads: Dynamic pipelining incurs two additional costs. The first is the increased number of messages. More messages require processor time in sending, receiving, storing meta data and acknowledging. Delta messages must be non-blocking, which is more costly to manage than blocking communication because of simultaneous transfers. The second cost is access monitoring. OS triggering incurs one page fault for each increment. Compiler annotation invokes the run-time library. We must control the two costs so they do not outweigh the benefit of pipelining.

Comparison with message splitting: In current MPI libraries such as OpenMPI, a non-blocking send, if it sends a large message, is divided into "fragments" to avoid flooding the network [3]. The library-level message splitting does not interleave communication with dependent computation, but delta send-recv does. In implementation, delta send-recv may adjust the increment size based on not just the network but also the program computation.

Comparison with compiler optimization: Prior work has used loop strip-mining and tiling to enable sender-receiver pipelining [4], [5]. Automatic transformation requires precise send-receive matching, a difficult problem for explicitly parallel code [6], [7]. In comparison, dynamic pipelining does not need static send-receive matching.

III. Evaluation

A. Methodology

Implementation: Our system is implemented as a user-level library over the standard MPI interface. It is written in C and provides an interface for use by C/C++/Fortran programs. To optimize, we manually insert delta send-recv functions in a way that can be automated with the compiler support. The insertion of delta send end and wait calls is a matter of replacing the original send and wait calls. Delta send begin and delta receive require knowing the start of the dependent computation. Loop unrolling is need by compiler annotation (but not by OS triggering). For regular loop code, such analysis and transformation can be implemented with existing techniques. The run-time library uses a pre-set delta size.

Test Suite: We use kernel and simplified application benchmarks. The kernel tests are as follows:

1) **pair:** The sender computes and sends a data array to the receiver, which performs identical computation (and compares the two results).
2) **cascade:** p tasks connected as $p-1$ pairs.
3) **ring:** A virtual ring of p tasks, each computes and sends data to the right, gets from the left, and repeats.
4) **array reduce:** A virtual tree of p tasks, each gets data from each child (if any), adds them and its own, and forwards the result to its parent. Equivalent to n MPI reduces, where n is the array size.

As the base line, we use blocking send-recv in all kernel tests. For instance, in *pair*, after the computation, the sender calls MPI_Send to pass the data array to the receiver, and the receiver calls MPI_Recv to get the data before its computation starts. The only exception is *ring*, which needs non-blocking send-recv to avoid deadlock.

We let the kernel program compute a *trigonometric* operation, in particular, $sin(i) * sin(i) + cos(i) * cos(i)$ for each element of the integer array. We run each test 100 times and take the average as the result. The performance variance is negligible, thus we don't show it on our graphs.

A set of more realistic tests is due to Danalis et al. at University of Delaware, who explained "Each kernel is an actual segment of the original program ... [and invoked] from a custom driver ... [to execute] multiple times (over 100) to amortize random noise effects." [8] We use 3 Delaware tests:

5) **NAS LU btls():** LU is a complete computational fluid dynamics simulation program in the NAS suite [9]. The kernel *btls* performs wavefront sweeps where each task waits for tasks on the "north" and "west." The standard test size, input C (message size 1KB), is not large enough for pipelining with the OS triggering implementation. We use input E instead (message size 10KB). We embedded the packing and unpacking loops into the compute loop so the message is created during (instead of after) the computation.
6) **HYCOM xcsum():** HYCOM is an application for ocean modeling run daily at the Navy DoD Supercomputing Resource Center [10]. The kernel *xcsum* performs a reduce for each row of a 2D array.
7) **NAS MG psinv( ):** MG is a multigrid solver in the NAS suite [9]. The kernel *psinv* performs stencil computation on three-dimensional data. Each task communicates with up to 6 neighbors. We duplicated both send and receive buffers to allow simultaneous communication by delta send-recv.

Operations	Time (μs)
computation (per page)	91.2
communicating 1 page	137.6
communicating 100 pages	5614.7
a (minor) page fault	9.9

Table I
BREAKDOWN OF PROCESSOR AND NETWORK SPEED

In the three tests, we use array padding to avoid mixing message array with other data on the same page. There are two other kernels in the Delaware suite. One is the NAS LU btls pre-conditioning loop. The kernel has no dependent computation, so delta send-recv has no benefit. The other is a simplified HYCOM xcaget kernel. The communication is mostly redundant, so it is unclear what to optimize.

Test Machine: The test platform is an 1Gb-Ethernet switched homogeneous PC cluster with 40 nodes. Each node has two Intel Xeon 3.2GHz CPU and 6GB memory, installed with Fedora 15 and GCC 4.6.1.. We use MPICH2 1.4.1 [11] as the underlying MPI library. We also tested on an older 32-processor IBM p690 multiprocessor machine and observed similar improvements but will not include IBM machine results for lack of space.

Table I shows the performance characteristics of the PC cluster. Communicating a 400KB message is 2.45 times faster than communicating 100 4KB messages, showing the overhead of incremental communication. In OS triggering, a page fault costs about 9.9 microseconds. In addition, the table shows the time per page for the *trigonometric* computation used in the kernel tests. Our test machine represents a commodity cluster rather than an up-to-date HPC system, however, delta send-recv should be applicable to newer systems as long as the communication time is not negligible.

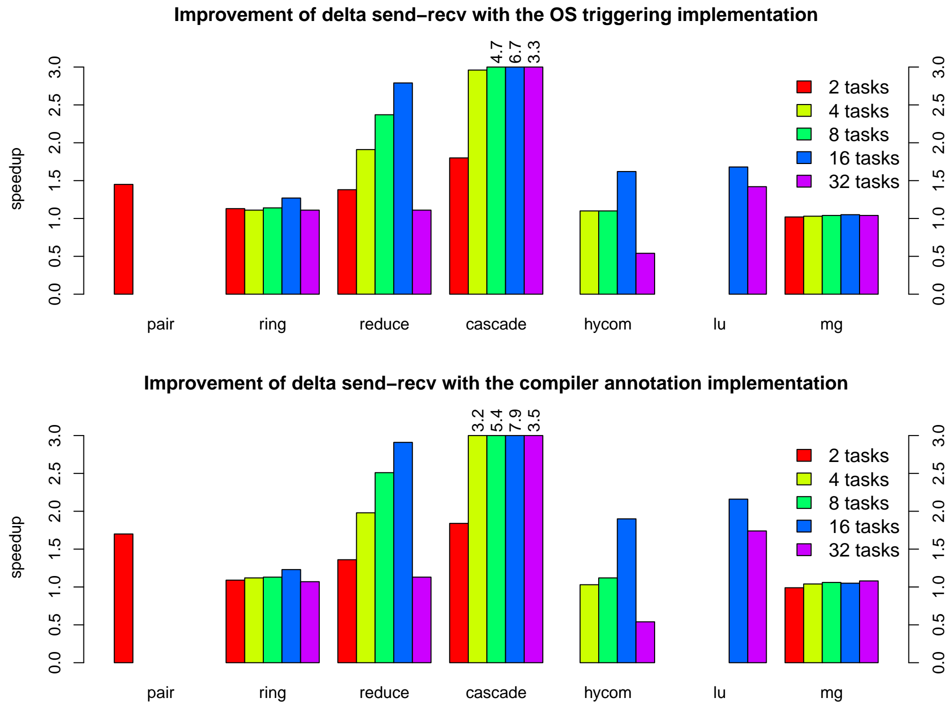


Figure 4. Performance improvements over the 2 to 32 task runs for the kernel and reduced application tests by delta send-receive with compiler annotation and OS triggering

B. Delta Send-Recv Performance

Figure 4 shows the improvement using delta send-recv over MPI send and receive when running 2 to 32 tasks.

Kernel tests: We use a message size of 400KB in these tests. The increment size is 16KB. We will evaluate different message and delta sizes later. With the OS triggering, *pair* shows 45% performance improvement. The base version of *ring* is usually considered highly efficient since the communication fully overlaps when each task sends and receives at the same time. Still, delta send-recv improves the performance by 13%, 11%, 14%, 27%, and 11% for 2, 4, 8, 16, and 32 tasks respectively. The baseline execution time of *ring* is 2.19s, 2.3s, 2.27s, and 2.31s for 2, 4, 8, and 16 tasks, but it increases to 17.8s when there are 32 tasks. Thus, the low improvement at 32 tasks does not show a limitation of delta send-recv. It reflects the limitation in our machine performance, such as the saturation of network bandwidth. We observe a similar drop at 32 tasks in all the tests on the PC cluster.

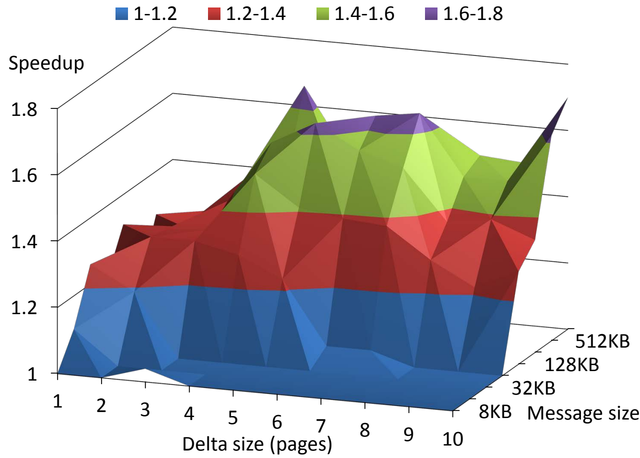
The test *reduce* is an MPI collective and similar to MPI_Bcast, MPI_Scatter, and MPI_Scatterv in that all implement one-to-many communication using a logical tree structure. An MPI library can internally pipeline the communication in a collective operation, as it was done in MPICH [12]. This test shows the effect of overlapping computation with communication, which cannot be implemented by current MPI collectives. In theory, the speedup increases as a logarithmic function of the number of tasks. On the cluster, we observe increasing speedups of 1.4, 1.9, 2.4, 2.8, and finally a drop to 1.1 due to insufficient bandwidth.

Cascade shows the largest speedups due to pipeline chaining, 1.8, 3.0, 4.7, 6.7, and 3.3. This test shows the best possible case for delta send-recv, in which the improvements increase linearly with the number of tasks in theory.

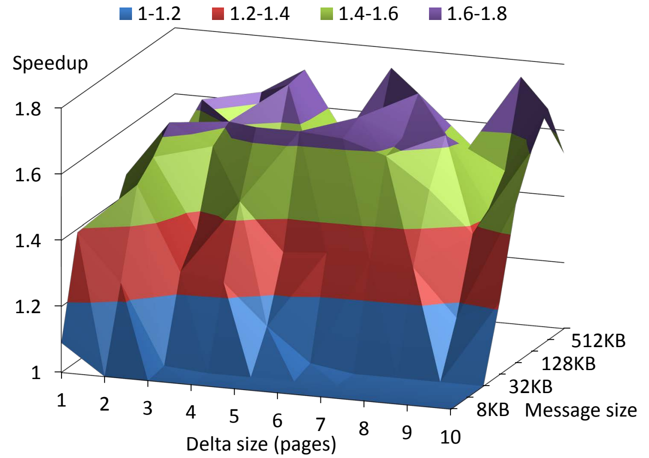
Compiler annotation is equally or more efficient. *Pair* shows 70% improvement. *Ring* is improved by 9%, 12%, 13%, 23%, and 7%, which are comparable to OS triggering. *Reduce* also shows similar speedups: 1.4, 2.0, 2.5, 2.9, and 1.1. *Cascade* shows higher improvements: 1.8, 3.2, 5.4, 7.9, and 3.5 times respectively.

Delaware tests: Not all task numbers are permitted in the *HYCOM* and *LU* kernels. With the OS triggering, the improvement for *HYCOM* is 10% for 4 and 8 tasks, 62% for 16 tasks, but a slowdown of 46% for 32 tasks due to the adverse effect of bandwidth contention. The improvement for *LU* is 68% for 16 tasks and 42% for 32 tasks. The improvements for *MG* are 2%, 3%, 4%, 5%, and 4%. The reason for the marginal improvement is that only 2 out of 6 communications can use delta send-recv.

Using compiler annotation, the performance improvements are similar or greater, as in the kernel tests. The largest enhancement happens at the 16-task run of *HYCOM*, 16-task



(a) OS triggering implementation



(b) Compiler annotation implementation

Figure 5. Improvement of *pair* for message sizes from 8KB to 1MB and delta increment sizes from 1 to 10 pages

and 32-task runs of *LU*. The improvements are 90%, 116%, and 74%, higher than 62%, 68%, and 42% by OS triggering.

We used different delta sizes and observed similar results except for *LU*. *LU* computes on a matrix and communicates the boundary data. The computation-to-communication ratio is high. We obtained a 2.52x speedup for 16 tasks when using 1KB delta versus 2.16x speedup in Figure 4.

C. Effect of Delta Size and Message Size

We evaluate delta send-receive in more detail using the *pair* test. We show the improvement as 3D plots as we vary the delta size from 1 page to 10 pages (4KB to 40KB) for the original message size from 8KB to 1MB, in Figure 5(a) for OS triggering and Figure 5(b) for compiler annotation.

Compiler annotation has two advantages. First, the speedup is higher in small message sizes: 9% vs 0% for 8KB messages, and 39% vs 10% for 16KB. Second, more combinations show high speedups, 19 vs 5 in the 1.6x-1.8x range.

Although not shown in Figure 5, we have evaluated some other combinations of larger delta sizes and message sizes. The general trend stays the same. The only difference is the performance drops to almost no speedup once the delta size reaches 32 pages. After looking into MPICH2 nemesis channel implementation, we find that a threshold on the cluster is MPIDI_CH3_EAGER_MAX_MSG_SIZE, which defines the switching point from eager protocol to rendezvous protocol, and its default value is 128KB. When a message is larger than this threshold, the communication will use the rendezvous protocol, and now in order to start a data transfer, the sender needs to wait for the acknowledgment from the receiver. As a result, when the delta size is equal to or larger than 32 pages, the latency almost doubles, and the effect of dynamic pipelining is diminished.

D. Comparison with Non-blocking Send-Recv

The Delaware tests were created to evaluate compiler techniques for overlapping communication with independent computation. In Figure 6, we compare the 16-task performance of their optimized code (tested on our machine) with delta send-recv.

In all three cases, pipelining communication with dependent computation is more profitable than overlapping with independent computation. The improvements by the two delta send-recv implementations and their code are 56%, 90% and 11% for HYCOM, 68%, 116% and 28% for LU, and 6%, 5% and -2% for MG.

We should note that their code was not optimized for our machines, so the performance may not represent the full capability of their techniques. In addition, they can improve communication when there is only independent computation

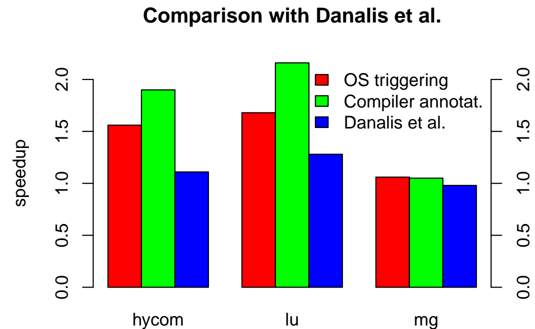


Figure 6. Comparing delta send-recv with Danalis et al. for overlapping communication with independent computation (16 tasks)

but delta send-recv cannot. This happens in one of the Delaware tests that we do not include here.

IV. Related Work

Virtual memory support: Virtual memory support has been used for incremental receive for messages in MPI [1] and bulk transfers in UPC [13]. Early release changes only one side of the communication. Our work adds incremental send to create dynamic pipelining. Delta receive, when using OS triggering, is similar to early release, except for the use of multi-page increments to amortize the paging overhead.

SBLLmalloc, a user-level memory allocator, uses virtual memory support to make MPI processes share read-only data (when they are executed on a single machine) and consequently reduces the memory consumption [14].

Compiler optimization: Danalis et al. developed a compiler to systematically convert all communication from blocking to nonblocking and automatically overlap communication with independent computation [8]. It uses loop distribution to move independent computations (so they could be overlapped) and variable cloning to enable such movements. Preissl et al. developed a hybrid approach, with trace analysis first to identify inefficient patterns in computation and communication and then compiler transformation to remove the inefficiency [15]. One pattern is a "late sender," for which a compiler converts a receive from blocking to non-blocking. Strout et al. developed dataflow analysis for MPI programs, which models information flow through communication edges [6]. Static techniques like these require send-recv matching. There are some recent advances in solving the matching problem, including the notion of task groups [7] and techniques for matching textually unaligned barriers [16]. However, static send-receive matching is not yet a fully solved problem especially with dynamically created tasks. Delta send-recv complements them by targeting dependent computation and by removing the requirement on static send-receive matching.

Pipelining is standard in compiler parallelized code, initially through message strip-mining [4] and later more systematically through give-n-take to place send as early as possible and the matching receive as late as possible [17]. The give-n-take framework uses a system of dataflow equations similar to those used to solve for lazy code motion in scalar compiler optimization [18]. It has been extended to consider the resource constraint [19]. These techniques are not designed for explicitly parallel code.

Manual programming: White and Dongarra developed and evaluated various form of computation and communication overlapping, but the transformations are done manually [20]. Marjanovic et al. studied a new programming model that combines MPI and SMPSs [21], [22]. The hybrid programming model achieves overlapping by marking communication and computation as SMPSs tasks and having the tasks scheduled properly by the runtime system. The new model requires a user to identify parallelism inside a task. In comparison, delta send-recv follows the traditional MPI programming paradigm.

MPI library design: Modern MPI libraries support non-blocking send-recv and also non-blocking collectives in libNBC [23] and the upcoming MPI-3 standard [24]. Non-blocking communication yields significant benefits in large-scale production MPI code [25]. It is generally useful for messages of all sizes. Delta send-recv is complementary, and the implementation is effective for mainly large messages.

MPI collectives such as irregular all-gather (MPI_Allgatherv) make use of communication pipelining [26]. It requires no changes to the user code but the effect is limited to a single MPI operation. Delta send-recv requires code changes but extends the benefit of pipelining beyond a single MPI call to include the user computation.

V. Summary

We have presented the delta send-recv interface and the two faceted implementation by OS triggering and compiler annotation. An MPI program can create pipelines dynamically between two or more tasks. We have evaluated the design using different communication topology, computation intensity, message and increment sizes, real benchmark kernels, different machines, networks, and MPI libraries. The results show that the best size is as small as 8KB for the message and 1KB for the delta increment. The improvement is up to 2.9 times for 16-task MPI reduce and 7.9 times for 16-task cascade.

Acknowledgment

The authors would like to thank Anthony Danalis at University of Tennessee Innovative Computing Laboratory and Oak Ridge National Laboratory for sharing the extracted NAS benchmark code. We thank Tongxin Bai for taking part in the initial design of the system. We also thank the anonymous CCGrid reviewers for their extremely valuable questions and suggestions. Bin Bao is supported by the IBM Center for Advanced Studies Fellowship. The research is also supported by the National Science Foundation (Contract No. CCF-1116104, CCF-0963759, CNS-0834566).

References

[1] J. Ke, M. Burtscher, and W. E. Speight, "Tolerating message latency through the early release of blocked receives," in *Proceedings of the Euro-Par Conference*, 2005, pp. 19–29.

[2] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers, Oct. 2001.

[3] T. Woodall, R. Graham, R. Castain, D. Daniel, M. Sukalski, G. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine, "TEG: A high-performance, scalable, multi-network point-to-point communications methodology," in *Proceedings of Euro PVM/MPI*, Budapest, Hungary, September 2004, pp. 303–310.

[4] A. Wakatani and M. Wolfe, "A new approach to array redistribution: Strip mining redistribution," in *Proceedings of PARLE*, 1994, pp. 323–335.

[5] 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*, vol. 14, no. 8-9, pp. 741–767, 2002.

[6] M. M. Strout, B. Kreaseck, and P. D. Hovland, "Data-flow analysis for MPI programs." in *Proceedings of the International Conference on Parallel Processing*, 2006, pp. 175–184.

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

[8] A. Danalis, L. L. Pollock, D. M. Swany, and J. Cavazos, "MPI-aware compiler optimizations for improving communication-computation overlap," in *Proceedings of the International Conference on Supercomputing*, 2009, pp. 316–325.

[9] D. Bailey, J. Barton, T. Lasinski, and H. Simon, "The NAS parallel benchmarks," NASA, Tech. Rep. 103863, Jul. 1993.

[10] E. P. Chassignet, L. T. Smith, G. R. Halliwell, and R. Bleck, "North atlantic simulation with the hybrid coordinate ocean model (HYCOM): Impact of the vertical coordinate choice, reference density, and thermobaricity," *Journal of Physical Oceanography*, vol. 32, pp. 2504–2526, 2003.

[11] "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/.

[12] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[13] C. Iancu, P. Husbands, and P. Hargrove, "HUNTing the overlap," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2005, pp. 279–290.

[14] S. Biswas, B. R. de Supinski, M. Schulz, D. Franklin, T. Sherwood, and F. T. Chong, "Exploiting data similarity to reduce memory footprints," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2011, pp. 152–163.

[15] R. Preissl, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan, "Transforming MPI source code based on communication patterns," *Future Generation Comp. Syst.*, vol. 26, no. 1, pp. 147–154, 2010.

[16] 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*, 2007, pp. 194–204.

[17] R. von Hanxleden and K. Kennedy, "A balanced code placement framework," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 5, pp. 816–860, 2000.

[18] K. Cooper and L. Torczon, *Engineering a Compiler, 2nd Edition*. Morgan Kaufmann, 2010.

[19] K. Kennedy and A. Sethi, "A constraint based communication placement framework," Center for Research on Parallel Computation, Rice University, Tech. Rep. CRPC-TR95515-S, Feb. 1995.

[20] J. B. White III and J. J. Dongarra, "Overlapping computation and communication for advection on hybrid parallel computers," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2011, pp. 59–67.

[21] J. M. Pérez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Proceedings of the IEEE International Conference on Cluster Computing*, 2008, pp. 142–151.

[22] V. Marjanovic, J. Labarta, E. Ayguadé, and M. Valero, "Overlapping communication and computation by using a hybrid MPI/SMPSs approach," in *Proceedings of the International Conference on Supercomputing*, 2010, pp. 5–16.

[23] T. Hoefler, A. Lumsdaine, and W. Rehm, "Implementation and performance analysis of non-blocking collective operations for MPI," in *Proceedings of the ACM/IEEE conference on Supercomputing*, 2007, p. 52.

[24] T. Hoefler and A. Lumsdaine, "Non-blocking collective operations for MPI-3," MPI Forum, 2008.

[25] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis, "Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications," in *Proceedings of the ACM/IEEE conference on Supercomputing*, 2006, p. 125.

[26] J. L. Träff, A. Ripke, C. Siebert, P. Balaji, R. Thakur, and W. Gropp, "A simple, pipelined algorithm for large, irregular all-gather problems," in *Proceedings of Euro PVM/MPI*, 2008, pp. 84–93.