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

Bin Bao  Chen Ding  Yaoqing Gao*  Roch Archambaul*

The University of Rochester
Computer Science Department
Rochester, NY 14627

Technical Report TR-953

October 2010

Abstract

The paper presents delta send-recv, an MPI extension for overlapping coarse-grained computation and communication. It provides an interface for marking the data computation and its communication. It automatically blocks computation and divides communication into increments. Delta sends and recvs are dynamically chained to effect sender-receiver pipelining, which is superior to pipelining only at the sender or the receiver side.

The evaluation uses kernel tests to find the best increment size for different MPI implementations and types of machines and networks. It shows 2 to 3 times performance improvement for large-volume data reduce involving 16 or 32 processors. In addition, the new interface enables computation and communication pipelining in an interpreted programming language, Rmpi.

* The research is supported by IBM CAS Student Fellowship, and IBM CAS Faculty Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding organizations.

IBM Toronto Software Lab, Markham, ON, Canada
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. However, it cannot be used when the communicated data is computed by the preceding computation or when the received data is used by the succeeding computation. In this paper, we present an extension called delta send-recv to enable overlapping between communication and dependent computation.

Delta send-recv divides a data message and its computation into pieces that we call deltas or increments. On the sender side, the communication starts as soon as the first increment is computed. The communication of early increments is overlapped with the computation of later ones. On the receiver side, the data can be used as soon as the first increment arrives. The use of early increments is overlapped with the communication of later ones. To coordinate computation and communication, delta send-recv uses virtual memory support to monitor data access at page granularity. Similar monitoring has been used to implement early release in an MPI receive (Ke et al., 2005).

When incremental send and receive are combined, data is computed, communicated, and consumed in a distributed processing pipeline we call a sender-receiver pipeline. Moreover, 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.

Delta send-recv improves both performance and programmability. One example is large volume data reduce, when many MPI tasks contribute values to a large set of data. Coarse-grained reduce, as shown by Google’s map-reduce, is useful for a wide range of data search and indexing problems. Existing MPI reduce is designed for communicating a single variable not a large data set. With delta send-recv and pipeline cascading, the cost of reduce can be reduced by $O(\log k)$ for $k$ tasks in a virtual tree topology, compared to using MPI non-blocking send-recv. In addition, the group cascading requires no global coordination, unlike MPI collection operations.

Delta send-recv simplifies the coding for communication optimization. It can be used by either a user or a compiler. It enables pipelining without having to reorganize the computation code. It can support variable-size communication, where the size is unknown until the complete message is generated. In addition, the use of virtual-memory paging support allows delta send-recv to monitor computation without having access to program source code. This 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 rest of the paper is organized as follows. Section 2 describes the interface and implementation. The effect of dynamic pipelining and cascading is analyzed, including the main factors such as the amount of computation, the cost of communication, the overhead of delta send-recv, and the size of the delta increment. In Section 3, we evaluate performance using a set of kernel tests, involving pair, ring, and tree based task groups, performing dependent and independent computations, running on different processors and networks, and using different MPI library implementations. We also show delta communication in
Rmpi, a language that is too high level to apply a similar transformation by hand. Finally, we discuss related work in Section 4 and summarize in Section 5.

2 Dynamic Pipelining Through Incremental Communication

2.1 Delta Send-Recv Interface

The interface for delta-send and delta-receive operations are shown in Figure 1. The sender uses three primitives shown in Figure 1(a). First, MPI_Delta_send_begin is called before a computation that produces a message. The message will be placed in the send buffer. The call implies that the writing to the send buffer is sequentially done from start to end. The interface, including the parameter for the send buffer, is identical to a normal MPI send. The use is different—MPI_Delta_send_begin comes before the computation for the message not after it. Also the second parameter, count, does not have to be the exact size of the sending data, but an upper bound. This enables variable-size communication as we shall discuss in Section 2.2.

The call to MPI_Delta_send_begin is paired with a call to MPI_Delta_send_end, which is used to mark the end of the computation. This is the place in code when the message has been completed. It is the conventional place where MPI send would have been inserted if the delta send is not used. Finally for the sender, MPI_Delta_wait is used to finish the delta send, similar to the MPI_wait for a MPI non-blocking send. It blocks the sender until the message has been delivered.

The interface for the receiver has one primitive as shown in Figure 1(b). MPI_Delta_recv has the same interface as a normal MPI receive. It should be placed before the communi-

```c
int MPI_Delta_send_begin(
    void *buf,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm,
    Delta_MPI_Request *req);

int MPI_Delta_send_end(
    Delta_MPI_Request *req);

int MPI_Delta_wait(
    Delta_MPI_Request *req,
    MPI_Status *status);

int MPI_Delta_recv(
    void *buf,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm comm,
    MPI_Status *status);
```

Figure 1: Delta send/recv interface in the C language
/* dependent computation */          MPI_Irecv(...);
... ...
MPI_Isend(...);
/* independent computation */          MPI_wait(...);
... ...

/* independent computation */          MPI_wait(...);
... ...
MPI_wait(...);
/* dependent computation */          MPI_wait(...);
... ...

(a) Non-blocking send to overlap independent
computation with communication
(b) Non-blocking recv to overlap independent
computation with communication

MPI_Delta_send_begin(...);
/* dependent computation */          MPI_Delta_recv(...);
... ...
MPI_Delta_send_end(...);
/* dependent computation */          /* independent computation */
... ...
MPI_Delta_wait(...);
/* independent computation */          ... ...
(MPI_Delta_send_end(...);
/* dependent computation */          /* independent computation */
... ...
MPI_Delta_wait(...);
/* independent computation */          (d) Delta-recv to overlap both dependent and in-
dependent computation with communication
... ...

Figure 2: Comparison between MPI non-blocking send and the delta send. Delta send needs
one more call on the sender side. Delta receive uses one fewer call on the receiver side.
cated data is needed, which is the same place for an MPI non-blocking receive. Because of
the run-time support (to be discussed next), there is no need for an MPI_wait.

Figure 2 compares the use of a delta-send with an MPI non-blocking send in a general
computation-communication setup. The computation has two parts: the dependent part
computing the message, and the independent part unrelated to the communication. MPI
non-blocking send inserts the send after the dependent computation and the wait after
the independent computation. In comparison, delta-send inserts the begin marker before
the dependent computation, the end marker after it, and the wait after the independent
computation. In this way, delta-send marks the code for the dependent computation for
overlapping with the communication.

2.2 Delta Send-Recv Implementation

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, it provides an
interface for use by C/C++/Fortran programs.
Delta-send  MPI_Delta_send_begin puts the entire send buffer under page protection except for the first delta. A delta is a group of consecutive memory pages. The operation also installs a custom page fault handler. When a write to the send buffer triggers a page fault, the signal handler in invoked. If the writes to the send buffer are sequential, the fault address should be the start of the next delta. The signal handler sends the previous delta using an MPI non-blocking send, unprotects the pages in the next delta, and resumes the execution of the sender.

The sequential write to the send buffer means that when the computation reaches a new delta, the previous one is ready to be sent. In the preceding fashion, the page fault handler sends all deltas 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 can 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 let receiver task continue its execution. It posts MPI non-blocking receives for all increments for the shadow buffer. The receiver incurs a page fault and the custom page fault handler when touching a protected page. The handler finds the corresponding page in the shadow buffer, identifies the increment message that includes that shadow page, and calls MPI wait for that message. Once the increment is arrived, the handler copies the pages from the shadow to the receive buffer, unprotects them, and resumes the receiver.

The run-time support guarantees the arrival of data upon the first access, so the receiver can access any data in the receiver buffer in any order. There is no need for a wait operation to request the arrival of all increments.

The receive process is similar to early release developed by Ke et al. (Ke et al., 2005) 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 perform the background receiving. More importantly, early release incurs a page fault for every received page. Delta-recv, like delta-send, is parameterized by the delta size $\Delta x$ and incurs a page fault every $\Delta x$ pages. As we will show later in Section 3.3, the overhead is substantially lower when $\Delta x$ is between 3 and 8 than when $\Delta x = 1$.

Communication placement  The placement of MPI_Delta_send_end and MPI_Delta_wait is the same as MPI_Isend and MPI_Wait for MPI non-blocking sends. The placement of MPI_Delta_recv is the same as MPI_Irecv for MPI non-blocking receives. The only one left is MPI_Delta_send_begin, which should be inserted before the dependent computation (that writes the send buffer in a sequential order). The insertion is done by hand in our experiments but can potentially be automated using compiler support.

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-receive pairs in tandem, a user can optimize sender code and receiver code separately. We call this a one-sided transformation. A user can apply message blocking for 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 characteristics of the execution environment, delta-send and delta-recv can dynamically choose different increment sizes to maximize performance or to switch off to avoid extra overhead.

**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 (Woodall et al., 2004). 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** Previous compiler research has used loop strip-mining and tiling to enable sender-receiver pipelining (Wakatani and Wolfe, 1994; Mellor-Crummey et al., 2002). Automatic transformation requires precise send-receive matching, a problem not yet fully solved for parallel code, despite some recent research progress on it (Bronevetsky, 2009; Zhang and Duesterwald, 2007). In comparison, pipelining by delta send-recv is formed dynamically without the need for static send-receive matching.

**Supporting variable-size communication** A data transfer is called variable-size communication if the size of the message is unknown until the end of the message is computed. One solution is to use two sends after the message is computed: the first to send the size, and the second to send the data. Another one is to have the receiver pre-allocate enough receive buffer and retrieve the length of the message once the message has arrived.

With MPI_Delta_send_end, the end of computation is explicitly marked. At that time, the run-time support takes the current increment as the last increment and informs the receiver side not to expect more data. Now the coding becomes simpler since the sender code does not have to explicitly extract the length of the message from the code. More importantly, delta send-recv eliminates the dependence on computing the messages size which makes computation and communication overlapping feasible for variable-size communication.

### 2.3 Sender-Receiver Pipelining

When used together, delta send and receive enable the parallelism between sender and receiver computation. The pipelining effect is illustrated by an example in Figure 3. In the example, the sender and the receiver have dependent computations on a set of data. The
sender also has some independent computation. Figure 3(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 3(b). The sender begins to execute. Communication starts after computing the first increment. The receiver starts upon receiving the first increment. At this time, the sender, the network, and the receiver are all running in parallel. The process is a 3-stage pipeline, where data of each increment is computed, transferred, and used. In addition to parallelism, the network utilization is also improved since the communication calls are spaced out.

**Overhead** There are several sources of overhead when using delta send-recv. It transfers a series of messages instead of a single message and requires processor time in setting up and receiving a message, 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. Since paging support is used to monitor program data access, there will be the cost of one page fault for each increment. Note that the increased parallelism still reduces the finish time of the receiver, even though the sender and the receiver each runs longer due to the overheads.

**Cascading** Delta send-recv may be chained together between more than two tasks. If we extend the example in Figure 3 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

---

Figure 3: MPI non-blocking send-recv cannot overlap communication with dependent computation, while delta send-recv does so through dynamic pipelining between the sender and the receiver.
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.

2.4 Analysis

To identify major factors in performance, we consider the sender-receiver pair shown in Figure 4(a).

**Delta-send** First, the time of the sender, \( T_{\text{sender}} \), includes the time of computing and transferring the data. There are four factors:

- *The computation size* \( x \), which is the time taken to compute the communicated data.
- *The number of increments* \( n \), which is the number we divide the computation and communication into.
- *The communication time* \( y(d) \), which is a *function* of the message size \( d \). The communication time is \( y(x) \) for all data and \( y\left(\frac{x}{n}\right) \) for each increment.
- *Per increment cost* \( z \), which is the extra cost due to creating an increment.

Without incremental sends, the total sender time is the sum of computation and transfer time, \( x + y(x) \). With delta send-recev, the total is computation time plus the last communication, assuming no contention on the network.

\[
T_{\text{sender}} = n\left(\frac{x}{n} + z\right) + y\left(\frac{x}{n}\right) = x + nz + y\left(\frac{x}{n}\right)
\]
The last two terms depend on $n$: the overhead $nz$ and the communication time $y(x/n)$. They change in opposite directions when $n$ changes. Take the common case where communication time is dominated by network latency, the volume of data has little effect on $y$, for example, $y(x) \approx y(x/n)$. Then $T_{sender}$ is minimized by setting $n = 1$, which means no delta send-receive. In fact, delta-send has almost no chance of improvement considering that the time without delta send-receive is $T_{sender} = x + y(x)$. The prospect changes, however, when we consider it in conjunction with delta-receive.

**Pipelining** Let time $T_{total}$ be the time from the start of the sender to the end of the receiver. For simplicity of illustration, we assume identical computation, $x$, by the receiver. Without delta send-receive, $T_{total} = 2x + y(x)$.

There are two cases in computing the time of delta send-receive. The first is computation bound, in which the time taken to compute an increment is no smaller than the time to communicate one, that is, $x/n + z \geq y(x/n)$. An example of this case is shown in Figure 4(b). In general, the total time is

$$T_{total} = (n + 1)(x/n + z) = x + x/n + (n + 1)z$$

The last two terms depend on $n$: the prologue time of the pipeline, $x/n$, and the overhead $(n + 1)z$. They change in opposite directions. Larger $n$ increases the overhead but reduces the prologue time (which means higher) parallelism. The optimal tradeoff can be computed for this simple case. Following the standard procedure, we can show that $T_{total}$ is lowest when $n_{opt} = \sqrt{x/z}$. Note that with pipelining, delta send-receive may reach a speedup as high as 2.

The second case is communication bound, where $x/n + z < y(x/n)$. An example is shown in Figure 4(c). The total time is

$$T_{total} = 2(x/n + z) + y(x) = 2z + 2x/n + y(x)$$

The variable term is minimized by maximizing $n$. The potential improvement from delta send-receive is low since the communication time $y$ dominates the total time. The potential improvement is much higher, however, when multiple pipelines are chained together.

**Cascading** Consider $k$ MPI tasks performing a series of dependent computations. An example is a reduce operation implemented using a virtual tree topology. Each intermediate tree node receives results from its children, updates them, and forwards them to its parent. The computation pattern on the path from a leaf to the root is a chain of send-receive pairs. When delta send-receive is used, the pipelining between each pair of nodes forms a cascade.

For simplicity, we assume that a sender and a receiver are identical. The total time without delta send-receive is $T_{total} = kx + (k - 1)y(x)$. The total time with delta send-receive is

$$T_{total} = \begin{cases} 
  x + (k - 1)x/n + (n + k)z & \text{if } x/n + z \geq y(x/n) \\
  kz + kx/n + y(x) & \text{otherwise}
\end{cases}$$
In either case, the speedup can be as high as $k - 1$. This exceeds maximal potential of computation and communication overlapping at just the sender side or just the receiver side. For each task, the overlapping can reduce execution time by a factor of two. Without cascading, the maximal improvement for $k$ tasks is bounded by 2. However, the bound is increased to $k - 1$ with cascading.

Choosing the increment size  Delta send-recv has one main parameter—the size of the increment. If page-protection is used for detecting data access, the increment is a multiple of the page size. We denote one increment as $\Delta x$. It is possible to estimate the various factors, $x, y, z$, and to choose a different size for $\Delta x$ for each delta-send and to use different sizes in the same delta-send. Such adaptive control can be used to estimate when delta send-receive is beneficial and how to maximize its benefit.

In theory, the best increment size depends on the amount of computation $x$, the cost of communication $y$, and the overhead of delta send-receive $z$. In previous analysis, we showed that in the case of symmetric, computation-bound sender-receiver, the optimal delta size is $n_{opt} = \sqrt{\frac{x}{z}}$. Through experiments that we will discuss in Section 3, we found the best increment size is fairly constant for two reasons. First, the overhead $z$ becomes insignificant once $\Delta x$ is 5 memory pages. The smallest $\Delta x$ is one page. It incurs a page fault for every page. When $\Delta x$ is five pages, the number of page faults is reduced by 80% and further reduction is not as significant. Second, when the message size increases over 5 memory pages, the communication time increases either because of more network contention or a large message being divided by the underlying MPI library.

3 Evaluation
3.1 Experimental Setup

Our prototype 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 test it we use two machines. The first is a 32-processor IBM p690 “Regatta” multiprocessor machine with 1.3GHz Power4 processors and with 32GB total memory. It is a shared memory machine, but we run parallel tests through MPI. The second is an Ethernet switched homogeneous cluster with over 40 nodes. Each node has two Intel Xeon 3.2GHz CPU and 6GB memory. We use MPICH2 1.2.1 (MPICH2) on both machines (An Ethernet aware implementation would be faster on the PC cluster (Karwande et al., 2003)). We run each test three times and use the average as the result.

The two systems have different characteristics. The IBM machine has relatively slow processors and an integrated, high performance network. The Intel cluster has fast processors but a relatively slow commodity network. We run tests under exclusive access on the IBM machine. The nodes of the Intel cluster and its network are shared with other users in the department. The two machines have completely different operating systems, AIX5.3 on IBM and Linux 2.6.30.10 on the cluster. The difference in system and hardware architecture helps to test the sensitivity of delta send-receive to processor or network speed or the design of the operating system.
We use four kernel benchmarks:

1. **pair**: The sender computes an array of data and sends them to the receiver. The receiver performs identical computation on the received data (and check whether the two results agree).

2. **cascade**: It consist of a series of communicating pairs.

3. **ring**: Tasks form a virtual ring. Each one computes the data, sends the data to the right neighbor, receives from the left neighbor, and verifies the received data. Unlike cascade, there is no dependence between the computation in these tasks.

4. **reduce**: Tasks form a tree. Reduce is performed from the leaves to the root. Every process receives one array of data from each of child, combines their values, adds its own values, and forwards the updated array to its parent. After the reduce, the root process holds an array in which each element is the sum of values from all children processes. For an array of size \( n \), this is equivalent of performing \( n \) MPI all-reduce operations (each on a single value).

The test **pair** shows the basic effect of sender-receiver pipelining. **Cascade** and **ring** show the effect of \( k \) senders and receivers, with dependent and independent computations respectively. **Reduce** contains a mix of dependent and independent computations. All four cases are common communication patterns in distributed applications.

The **reduce** test is representative of MPI collectives. MPI_Bcast, MPI_Scatter, and MPI_Scatterv—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 (Thakur et al., 2005). Since our test contains program processing at intermediate steps, it cannot be implemented entirely inside an existing MPI library. Delta send-recv provides the needed extension for MPI to pipeline communication and computation together.

For each of the four tests, we alter the amount of computation using the following three types:

1. **empty**: The program accesses one element on each memory page of the message array. This is the minimum required for triggering delta-sends.

2. **random**: The program calls the system random number generator to calculate each element of the message array.

3. **trigonometric**: The program computes \( \sin(i) \times \sin(i) + \cos(i) \times \cos(i) \) for each element of the array.

Table 1 lists the time of computing and transferring one memory page and the time of a minor page fault on the two systems. The computing time takes 12, 32, to 199 microseconds per page on IBM Regatta and 5, 27, and 91 microseconds per page on the PC cluster, showing that latter has twice the processor speed as the former. We expect a higher performance gain when a program performs a greater amount of computation per page.
Table 1: Unit Performance (microseconds)

<table>
<thead>
<tr>
<th></th>
<th>Regatta</th>
<th>Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty computation (1 page)</td>
<td>11.5</td>
<td>4.6</td>
</tr>
<tr>
<td>Random computation (1 page)</td>
<td>31.6</td>
<td>27.4</td>
</tr>
<tr>
<td>Trigonometric computation (1 page)</td>
<td>198.7</td>
<td>91.2</td>
</tr>
<tr>
<td>Communication (1 page)</td>
<td>54</td>
<td>137.6</td>
</tr>
<tr>
<td>Communication (100 pages)</td>
<td>8680.3</td>
<td>5614.7</td>
</tr>
<tr>
<td>A minor page fault</td>
<td>18.9</td>
<td>9.9</td>
</tr>
</tbody>
</table>

The table also shows the time to communicate small (1 page) and large (100 pages) messages. The time is collected based on a round-trip communication test. IBM Regatta is nearly 3 times faster for 1-page messages than the PC cluster but 60% slower for 100-page messages. The faster IBM time reflects that MPICH2 uses shared memory for small messages on that machine, which is effectively a different MPI implementation than the one used on the cluster. Finally, the table shows that a minor page fault costs 19 and 10 microseconds on the two systems. The overhead is not negligible.

We do not yet have a full-size C or Fortran application to demonstrate the benefit of delta send-recv. We have examined commonly used MPI tests including matrix multiply and the NAS benchmarks and found that communication consists of small messages except at the initial stage. When using large input sizes, the time of the initial stage is negligible in overall time. We plan to test delta send-recv when we have access to this type of MPI programs. It is possible that new delta send-recv may lead to new MPI applications that have coarse-grained computation and communication such as the map-reduce type. For this study, we have tested delta send-recv on a non-trivial software system, Rmpi, which we will discuss later.

### 3.2 Delta Send-Recv Performance

Figure 5 shows the improvement using delta send-recv over MPI send and receive when running a different number of MPI tasks (processes). In the base version, we use blocking send-recv in all tests except for ring, which needs non-blocking send-recv to avoid deadlock and to overlap the send operation with the (independent) computation on the received data. In the optimized version, we manually insert delta send-recv functions. For the results in Figure 5, we set the data array to one million element (4MB) and the delta size to 5 memory pages. The best delta size is 5, as we will discuss in Section 3.3. The improvement for smaller data sizes is similar, as we will show in Section 3.4.

Figures 5(a) and 5(b) show the performance of cascade measured on IBM Regatta and the PC cluster respectively. On the IBM system, the improvement increases proportionally with the number of tasks. This confirms our analysis in Section 2.4 on pipeline chaining. The limiting factor on both machines is communication bandwidth. On IBM 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
Figure 5: Improvement of performance by delta send-recv over MPICH2 send-recv for different numbers of MPI tasks
other four graphs in Figure 5 show same saturation point in *ring* and *reduce* tests on the cluster. The bandwidth on IBM Regatta seems sufficient in these two tests.

On IBM Regatta, the highest improvement is a factor of 8.8 at 26 tasks for *empty* computation, 13.7 at 28 tasks for *random*, and 23 at 29 tasks for *trigonometric*. On the cluster, the highest improvement is a factor of 9.5 for *empty*, 11.8 for *random*, and 13.7 for *trigonometric*, all at 15 tasks. The *empty* case shows that with only communication pipelining, the improvement even by a highly optimized MPI library is at most 10, lower than the improvement, up to 14 and 23, from the other two cases when computation pipelining and communication pipelining are combined.

The improvement for *ring* increases with the number of tasks to around 50% for all three computation types on IBM Regatta, as shown by Figure 5(c). On the cluster, the improvement is constant for random and trigonometric, although the empty computation shows no improvement, as shown by Figure 5(d). 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 5(e) and 5(f). The reason is that the length of the longest dependent task chain in *k*-task reduce is $\log_2 k$. One unusual result is that on the cluster, the improvement is in the reverse order of the amount of computation. We do not yet have an explanation for the anomaly.

### 3.3 Finding the Best Delta Size

We have tested delta sizes from 1 page to 40 pages (4KB to 160KB) for a message size of 1MB. The results are shown by the 8 graphs in Figure 6 (four test programs on two machines). In most cases, delta send-recv performance increases significantly when the delta size increases from 1 page to a few pages. In most cases, the performance drops when the delta size reaches a large enough threshold. After looking into MPICH2 nemesis channel implementation, we find that the threshold on IBM Regatta is controlled by MPID\_NEM\_CELL\_LEN, which specifies the default length of a cell as 64KB. According to MPICH2 design (Buntinas et al., 2006), a cell is a receive queue element, for the intra-node communication. If the communication message plus the header information is larger than the size of a cell, the message has to be divided into several cells. This explains the sudden performance drop on IBM Regatta when the delta size is 16 pages or more.

The corresponding 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, as predicted by our model in Section 2.4.

For the *pair* test on Regatta (Figure 6(a)), the best delta size is from 4 pages to 15 pages. On the cluster (Figure 6(b)), the effect is similar except that the drop happens twice at...
Figure 6: Improvement of performance by delta send-recv over MPICH2 send-recv for different delta sizes.
Figure 7: Performance of delta send-recv over send-recv size

around 8 pages and 32 pages. The performance improvement is higher since communication is more expensive in the cluster environment.

The results for the cascade test (Figures 6(c) and 6(d)) are for 4 MPI tasks. Besides the larger speedup, the shapes of the curves are similar to the pair results. The ring tests (Figures 6(e) and 6(f)) also use 4 tasks. The reduce tests (Figure 6(g) and 6(h)) use 15 tasks. All the figures show that the best delta size is insensitive to the type of tests, the amount of computation, and the choice of the machine platform, network, and operating system. Any delta size from 3 to 8 gives the best performance in most cases.

3.4 The Effect of Send-Rev Size

Delta send-recv is designed for coarse-grain computation and communication. We now test the performance by varying the size of delta send-recv. We use 5 pages for the delta size and vary the send-recv size from 20KB (one increment) to 2MB (100 increments). Only results for pair are shown. Other tests show a similar conclusion. Delta send-recv improves performance when data size is at least two increments. On IBM Regatta, shown in Figure 7(a), the improvement by delta send-recv is in fact much higher at smaller data sizes. The reason can be that when there are only a few increments, the communication of these increments can be highly efficient and parallelized. On the cluster, shown in Figure 7(b), the improvement at a size as small as 100KB is similar to that of larger sizes. Our results from the other three tests also show that delta send-recv is beneficial when send-recv size is 40KB or more.

3.5 Pipelining in Rmpi

R is a widely used scripting language in statistic computing. Rmpi (Yu, 2002) is an R package that allows individual R programs to communicate using an MPI library. We have added the support for delta send-recv into Rmpi. A compiler-based automatic transforma-
<table>
<thead>
<tr>
<th>Array Size (Bytes)</th>
<th>Speedup Over MPICH2</th>
</tr>
</thead>
<tbody>
<tr>
<td>80KB</td>
<td>0</td>
</tr>
<tr>
<td>500KB</td>
<td>0.5</td>
</tr>
<tr>
<td>1MB</td>
<td>1</td>
</tr>
<tr>
<td>1.5MB</td>
<td>1.5</td>
</tr>
<tr>
<td>2MB</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 8: Performance improvement of delta send-recv over send-recv size in the Rmpi trigonometric pair test

We have incorporated delta send-recv into Rmpi 0.5-8, which runs on R 2.9.2. The test case we use is similar to the trigonometric pair kernel benchmark, but in R. Vector write in R is not in-place, so we implemented the sender computation as a library function called from R. Delta-recv works with the vector operation directly.

We tested the R program on the cluster system. Figure 8 depicts the results. We set the delta size to be 5 pages, and when we change the array size, the speedup is up to 20%. The improvement is lower than the pair kernel test in C because R implementation incurs extra overheads.

4 Related Work

Virtual memory support has been used to support early release in MPI (Ke et al., 2005) and incremental receive in bulk transfer in UPC (Iancu et al., 2005). Early release overlaps receiver computation with network transfer. However, it changes only one side of the communication and cannot create sender-receiver pipelining. Delta-recv by itself would have had little positive effect, especially if the communication time is dominated by latency. A second difference is that delta send-recv uses multiple pages in each increment to amortize the page-fault overhead, while previous work uses single-page releases. Recently, virtual memory support is used in PSMalloc, an memory allocator that lets MPI processes share read-only data (when they are executed on a single machine) and consequently reduces the memory consumption (Biswas et al., 2009).

To enable computation-communication overlapping, modern MPI implementations provide non-blocking send-recv as well as non-blocking collectives such as libNBC (Hoefler et al., 2007) and the upcoming MPI-3 standard (Hoefler and Lumsdaine, 2008). Non-blocking MPI operations can hide communication costs with only independent computations. Despite of this limitation, significant benefits are shown to exist in large-scale pro-
duction MPI code (Sancho et al., 2006). Non-blocking communication is generally useful for messages of all sizes. Delta send-recv is complementary. It overlaps the communication with its dependent computation. The implementation is effective only for large messages.

Communication pipelining leads to large performance gains when used in implementing irregular all-gather (MPI_Allgatherv) and other MPI collective communication (Träff et al., 2008). It requires only a library improvement but the effect is limited to a single MPI operation. Delta send-recv extends the benefit of pipelining beyond a single MPI call by interleaving communication with the computation it depends on. In addition, global cascading by delta send-recv does not require global coordination as needed by MPI collectives. Independent groups of senders and receivers can cascade dynamically.

Compiler parallelization has been developed to generate parallel code from a sequential program, either automatically or using an annotation language such as HPF. Message strip-mining has been used for data redistribution, possibly with the support of an executor for irregular data (Wakatani and Wolfe, 1994). A systematic solution is to place send as early as possible and the matching receive as late as possible (von Hanxleden and Kennedy, 2000), taking the resource constraint of communication into account (Kennedy and Sethi, 1996).

For explicitly parallel code, Danalis et al. developed a compiler framework that converts blocking send/recv to nonblocking calls, analyzes the dataflow relation of MPI calls, and systematically overlaps communication with computation (Danalis et al., 2009). One technique is to use loop distribution to create independent computations that can be moved around communication calls to increase the benefit of non-blocking communication. Techniques such as variable cloning is used to enable such movements (Danalis et al., 2009). These techniques require static send-recv matching.

Compiler analysis has been studied for explicitly parallel programs (Strout et al., 2006; Lee et al., 1999). Recent advances including the notion of task groups (Bronevetsky, 2009) and techniques for matching textually unaligned barriers (Zhang and Duesterwald, 2007). However, static send-receive matching is not yet a fully solved problem especially with dynamically created tasks. Delta send-recv does not require static send-receive matching and can be used for programs that are not amenable to compiler analysis.

Marjanovic et al. studied a new programming model that combines MPI and SMPSs (Pérez et al., 2008; Marjanovic et al., 2010). The hybrid programming model achieves overlapping by marking communication and computation as SMPSs tasks and having the tasks scheduled properly by the runtime system. Delta send-recv is a simpler extension in comparison because it does not require explicitly parallelizing the sender or the receiver task.

5 Summary

We have presented delta send-recv, an extension to MPI for a user to express overlapping between communication and dependent computation. It dynamically forms sender-receiver pipelining between two tasks and pipeline cascades in more than two tasks. We have evaluated a prototype implementation using MPI task groups of different communication topology and computation intensity, using two machines with different processor and network speeds and system software. The results show that the best increment size is between
3 and 8 memory pages (12KB to 32KB), and the smallest message size to benefit from delta send-receive is 40KB to 100KB. In these kernel tests, the largest improvement comes from pipeline chaining, which has led to up to 23 times faster speed for 29 MPI tasks.

Delta send-receive decouples send operations from receive operations and allows one-sided transformation for an MPI program. It does not require changing the computation code. We demonstrate this benefit by pipelining a Rmpi program, for which manual transformation had been too difficult.

In conclusion, while delta send-receive does not replace careful manual coding needed for applications running on large scale machines, it may improve program productivity for ordinary users writing code for small cluster systems with modest staff support, commodity hardware, and limited programming expertise.

Acknowledgment

The authors thank Tongxin Bai for the helpful discussion, and Brandon Shroyer and Jonathan Carroll for the discussion about the applications and the idea of variable-size message support. We would also thank an anonymous reviewer for pointing out the threshold parameters in MPICH2 and for the comments we received at the First ACM SIGPLAN Workshop on Advances in Message Passing in Toronto in June 2010.

References


Sancho, José Carlos, Kevin J. Barker, Darren J. Kerbyson, and Kei Davis. 2006. Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *Proceedings of Supercomputing*, page 125.


