# On Optimizing Collective Communication

Marcel Heimlich
Department of Computer Sciences
The University of Texas at Austin
1 University Station C0500
Austin, Texas 78712
heimlich@cs.utexas.edu

May 13, 2003

**Abstract**

It has long been thought that research into collective communication algorithms on distributed memory parallel computers has been exhausted. This paper demonstrates that the implementations available as part of widely-used libraries are still suboptimal. We demonstrate this through the implementation of the "reduce-scatter" collective communication and comparison with the MPICH implementation of MPI. Performance on a large cluster is reported.

## 1 Introduction

Extensive research over the past decade has been reported in regards to collective communication and the implementations of algorithms for distributing data between processors. It has been shown that effective communication algorithms can be implemented using sophisticated techniques [6, 1, 2, 4, 5, 3, 8, 9]. Even though these algorithms have been extensively researched, public domain and vendor implementations are frequently still suboptimal. In this paper another step is taken to achieve even higher performance than in past presentations.

In this paper a set of collective communication algorithms implemented on distributed-memory multidimensional-mesh systems are presented. These already efficient algorithms are reconstructed to improve performance. Methods involving changing the type of communication between processors are used to increase performance. Using the methods detailed in this paper it is possible to more than double the performance of commercial and public domain provided implementations.

The remainder of the paper is organized as follows: In Section 2 we provide some basic assumptions that are made for the purpose of presenting this paper.

From that follows a section on the interface we use for passing messages between processors found in Section 3. In Section 4 we discuss the communication algorithm. Performance results are given in Section 5. In Section 6 a summary of the paper is given in a conclusion.

# 2 A model of parallel computation

In order to give an analysis on the performance of the algorithms it is necessary to present a model of parallel computation. The following assumptions are made in this report:

1. **Target architectures**
   The target architectures are distributed-memory multidimensional-mesh systems.

2. **Indexing** This paper assumes a parallel architecture with p nodes. The nodes are indexed from $P_0$ to $P_{n-1}$

3. **Communicating between nodes**
   At any given time each node can send one message to any of its neighbors. The communication network uses cut-through worm-hole routing. The result is that multi-hop messages do not take noticeably longer than a single hop message.[3]

4. **Cost of communication**
   For each message of length $n$ we model the cost of sending that message by $\alpha + n\beta$ where $\alpha$ and $\beta$ respectively represent the message startup time and per data transmission time. If no network conflicts occur then the time to send a message between **any** two neighbors is modeled by $\alpha + n\beta$.

5. **Cost of computation** The cost required to perform an arithmetic operation is denoted with a $\gamma$.

6. **Network conflicts**
   It is assumed that the path between two communicating nodes is completely occupied. Therefore if a communication path is being shared by more than two nodes then network conflicts occur. This extra cost is modeled with $\alpha + nk\beta$ where $k$ is the maximum number of conflicts associated with the links between the processors.

The above assumptions are useful when conducting an analysis of communication costs on actual architectures.

# 3 Message-Passing Interface (MPI)

The Message-Passing Interface (MPI) is a standardized and portable message-passing system designed by a group of researchers from academia and industry.

The purpose of MPI was to function on a wide variety of parallel computers. The MPI standard was developed in the early 1990's [7].

Sharing data between processors requires a send operation on the part of the processor that has the data and a corresponding receive by the processor that needs the data. The following discussion details the type of sends and receives used in the implementations discussed herein.

In MPI the send routine it called by the sending node and the receive routine is called by the receiving node.

## 3.1 Standard mode, Blocking Send and Receive

This is the basic type of send and receive that can be used and thus the least efficient yet sometimes necessary. No assumption should be made as to whether a guarantee can made that there is a certain amount of space in the outgoing buffer

To send a message, a call to `MPI_Send` is used:

```
MPI_Send(buf, count, datatype, dest, tag, comm )
```

- `buf - initial address`
- `count - number of entries to send`
- `datatype - datatype of each entry`
- `dest - rank of destination`
- `tag - message tag`
- `comm - descriptor of the processors involved in the communication`

This operation does what you would expect it to, which is it sends data from one processor to another.

To receive a message, a call to `MPI_Recv` is required:

```
MPI_Recv(buf, count, datatype, source, tag, comm, status )
```

- `buf - initial address`
- `count - number of entries to send`
- `datatype - datatype of each entry`
- `source - rank of destination`
- `tag - message tag`
- `comm - descriptor of the processors involved in the communication`
- `status - information regarding the received message`

The operation will receive the data which the send operation has posted.

## 3.2 Posting Nonblocking Send and Receive

The standard MPI nonblocking posting send syntax is -

`MPI_Isend(buf, count, datatype, dest, tag, comm, request)`

- `buf - initial address`
- `count - number of entries to send`
- `datatype - datatype of each entry`
- `dest - rank of destination`
- `tag - message tag`
- `comm - communicator`
- `request - request handle`

If you replace `Send` with `Recv` then you also have the syntax for a nonblocking post receive. These operations are also called "posting standard-mode, nonblocking sends and receives". Notice here that the post send and receive calls have the same names as the nonblocking calls except that they have an 'I' preceding their name which denotes "immediate". This signifies that the call is nonblocking. By using nonblocking communication one can increase the performance of certain algorithms because both the sends and the receives can be posted without the matching receive and send, respectively. The use of nonblocking receives allows one to post receives early and so achieve lower communication overheads without blocking the receiver while it waits for the send [7]. The difference when using nonblocking and blocking communication is that nonblocking communications use tags to allow identification of the communication operations. For example, a receiving process posts it's receive with tag $Q$ while the sending operation posts it's send with tag $Q$. This allows the two operations to match up with one another.

A non-blocking send indicates that data may be copied out of it's buffer where a non-blocking receive indicates that data may be written to it's buffer. The send and receive operations are unable to access any part of these buffers until the calls have been completed. The prohibition of read access to the send buffer actually allows better performance on some systems.

## 3.3 Synchronous, Blocking Send

The standard MPI nonblocking posting send syntax is -

`MPI_Ssend(buf, count, datatype, dest, tag, comm)`

- `buf - initial address`
- `count - number of entries to send`

- `datatype - datatype of each entry`
- `dest - rank of destination`
- `tag - message tag`
- `comm - communicator`

This type of send, synchronous mode, can be initiated whether a matching receive has been posted. The send only completes, successfully, when a matching receive is posted. Therefore, the completion of a synchronous send indicates the completion of a message transfer and that both the sender and the receiver have finished execution.

Notice that in the reconstructed algorithms detailed in this paper an initial `MPI_Ssend` is used after all the receives have been posted. This is to ensure that the communication between the nodes is synchronized. If one were to use all Ready Sends, described in Section 3.4 then it could not be ensured that all the receives have been posted and one may blatantly send to a process that has not posted all of it's receives. Therefore by using Ssend synchronization can be ensured.

The completion of a synchronous send indicates that the send buffer can be reused[7]

## 3.4 Ready-mode, Blocking Send

The standard MPI nonblocking posting send syntax is -

`MPI_Rsend(buf, count, datatype, dest, tag, comm, request)`

- `buf - initial address`
- `count - number of entries to send`
- `datatype - datatype of each entry`
- `dest - rank of destination`
- `tag - message tag`
- `comm - communicator`
- `request - request handle`

This type of communication is called a ready-mode blocking send. It is so denoted because a ready mode send may only be initiated if the matching receive has been posted. In essence this allows the removal of a hand-shake between the receive and send operation since the receive is assumed to already have been posted. Since the hand-shake between the communicators is removed this allows an increase in performance [1]. Therefore, every Rsend can be replaced by

[1] On some systems the increased performance is not visible.

a "standard-mode" blocking send without any adverse effects on the program other than performance. It should be noticed that a significant amount of performance increase in the collective communication algorithms discussed in this paper attain their increase in performance from this type of replacement, i.e. replacing the standard mode blocking sends with their respective non-blocking send. The other increase in performance is attained from the early posting of the receives discussed in Section 3.2.

### 3.5 Collective Communication

Frequently, communications involving all processors are required. Examples of this include simpler collective communications like a broadcast and more complex ones like reduce-scatter. These operations are implemented by a collection of individual messages. It is the algorithm chosen to orchestrate these messages that determines the time required for completion.

For the reduce-scatter operation, MPI supports

```
MPI_Reduce_scatter(sendbuf, recvbuf, recvcounts, datatype,
op, comm )
```

- `sendbuf - starting address of the send buffer`
- `recvbuf - starting address of the receive buffer`
- `recvcounts - integer array containing the number of items to be sent and received`
- `datatype - data type of the input buffer`
- `op - the operation to be performed during the reduce operation`
- `comm - descriptor of the processors involved in the communication`

This operation (for the case where the reduce is a simple summation) can be described as follows: Initially all processors have vectors of length $n$ items. Upon completion, these vectors have been added element-wise, and each processor owns approximately $1/p$ of the total result vector, where $p$ equals the number of processors involved.

Let us illustrate this on four processors:

- Before:

| Processor 0 | Processor 1 | Processor 2 | Processor 3 |
|---|---|---|---|
| $x_0^{(0)}$ | $x_0^{(1)}$ | $x_0^{(2)}$ | $x_0^{(3)}$ |
| $x_1^{(0)}$ | $x_1^{(1)}$ | $x_1^{(2)}$ | $x_1^{(3)}$ |
| $x_2^{(0)}$ | $x_2^{(1)}$ | $x_2^{(2)}$ | $x_2^{(3)}$ |
| $x_3^{(0)}$ | $x_3^{(1)}$ | $x_3^{(2)}$ | $x_3^{(3)}$ |

- After:

| Processor 0 | Processor 1 | Processor 2 | Processor 3 |
|---|---|---|---|
| $\sum_{j=0}^{3} x_0^{(j)}$ | $\sum_{j=0}^{3} x_1^{(j)}$ | $\sum_{j=0}^{3} x_2^{(j)}$ | $\sum_{j=0}^{3} x_3^{(j)}$ |

# 4 Communication Algorithms

The collective communication algorithms discussed in this paper entail those that allow communication of short and long vectors of information. The specific algorithm is the reduce-scatter discussed in Section 4.1 for the short vector and long vector cases. It turns out that, depending on the amount of data involved in a collective communication, a different algorithm is more efficient. Let us describe commonly used algorithms.

## 4.1 Short-vector algorithm

Short vector algorithms are so named due to the nature of the size of the data being transmitted. Basically, short vector algorithms transmit the data in a small packets where long vector algorithms need to transmit their data in several small packets. Short vector algorithms use what is called a Minimum Spanning Tree (MST) approach. MST algorithms incur $\lceil \log_2(p) \rceil$ startups due to their tree like nature. On hypercubes a minimum spanning tree is embedded from the node originating the communication which is denoted as the root.

The name of the reduce-scatter collective communication suggests implementing this operation as a reduce-to-one (`MPI_Reduce`) followed by a scatter (`MPI_Scatter`), each of which is a collective communication supported by MPI.

**MST Reduce-to-one** :

Let us illustrate this on four processors:

- Before:

| Processor 0 | Processor 1 | Processor 2 | Processor 3 |
|---|---|---|---|
| $x_0^{(0)}$ | $x_0^{(1)}$ | $x_0^{(2)}$ | $x_0^{(3)}$ |
| $x_1^{(0)}$ | $x_1^{(1)}$ | $x_1^{(2)}$ | $x_1^{(3)}$ |
| $x_2^{(0)}$ | $x_2^{(1)}$ | $x_2^{(2)}$ | $x_2^{(3)}$ |
| $x_3^{(0)}$ | $x_3^{(1)}$ | $x_3^{(2)}$ | $x_3^{(3)}$ |

- After:

| Processor 0 | Processor 1 | Processor 2 | Processor 3 |
|---|---|---|---|
| $\sum_{j=0}^{3} x_0^{(j)}$<br>$\sum_{j=0}^{3} x_1^{(j)}$<br>$\sum_{j=0}^{3} x_2^{(j)}$<br>$\sum_{j=0}^{3} x_3^{(j)}$ | | | |

1. Initially all processors have vectors of length $n$ items. Upon completion, these vectors have been added element-wise where one processor owns the entire result vector.
2. Reduce-to-one is an operation where the vectors of information are initially located on every processor. At the completion of the algorithm all of the data is on one processor.
3. The common algorithm used for a reduce-to-one the MST approach which leads to a cost of

$$\lceil \log_2(p) \rceil (\alpha + n\beta + n\gamma).$$

[MST Scatter]: Let us illustrate this on four processors:

- Before:

| Processor 0 | Processor 1 | Processor 2 | Processor 3 |
|---|---|---|---|
| $x_0^{(0)}$<br>$x_1^{(0)}$<br>$x_2^{(0)}$<br>$x_3^{(0)}$ | | | |

- After:

| Processor 0 | Processor 1 | Processor 2 | Processor 3 |
|---|---|---|---|
| $x_0^{(j)}$ | $x_1^{(j)}$ | $x_2^{(j)}$ | $x_3^{(j)}$ |

1. Initially, one processor holds the entire vector of length n items. Upon completion, each processor owns approximately $1/p$ of the total result vector, where $p$ equals the number of processors involved.
2. The Scatter operation is implemented using an MST approach. The cost associated with it is:

$$\lceil \log_2(p) \rceil (\alpha + \frac{p-1}{p} n\beta)$$

One would expect the estimated cost of a reduce-scatter to be that of a reduce-to-one plus that of a scatter.

$$\lceil \log_2(p) \rceil (\alpha + n\beta + n\gamma) + \lceil \log_2(p) \rceil \alpha + \frac{p-1}{p} n\beta$$

## 4.2 Long-vector Algorithms

When implementing long vector algorithms the goal is to achieve the optimal $\beta$ term. The algorithms presented herein may not be the most efficient implementations for all architectures but the simple approach used guarantees that there are no network conflicts.

### Reduce-scatter long vector

At every step the contribution to the local data has to be added before the contents can, then, be passed along. If all nodes receive the same amount of data at the end of the reduce-scatter, the cost is

$$(p-1)(\alpha + \frac{n}{p}\beta + \frac{n}{p}\gamma) = (p-1)\alpha + \frac{p-1}{p} n\beta + \frac{p-1}{p} n\gamma.$$

An illustrated picture is given in Figure 1. In that figure a notation of $\sum_{k=i:j} x_n^{(k)}$ is introduced. Here, $n$ is the data number and $k$ is the processor number. The $i$ is the beginning index into the processors and $j$ is the ending index. Notice that there may be some wrapping around of the numbers. i.e $i =$ 3, and $j = 0$ which indicates 3,.., $p$-1 , 0.

## 4.3 Comparing the two approaches

Recall that the cost of the short-vector algorithms has a cost of

$$2\lceil \log_2(p) \rceil \alpha + (\lceil \log_2(p) + \frac{p-1}{p}) n\beta + \lceil \log_2(p) \rceil n\gamma$$

where the long-vector algorithm has a cost of

$$(p-1)(\alpha + \frac{n}{p}\beta + \frac{n}{p}\gamma) = (p-1)\alpha + \frac{p-1}{p} n\beta + \frac{p-1}{p} n\gamma.$$

It is important to note that there is a critical choice to be made when choosing between the two different approaches. The MST algorithm is intended for use with short vectors of data where the bucket algorithm is intended for use with long vectors of data. Of course one could blindly use one algorithm or another but the consequence of this is one that leads to very poor performance.

| Proc. 0 | Proc. 1 | Proc. 2 | Proc. 3 |
|---|---|---|---|
| $x_0^{(0)}$ | $x_0^{(1)}$ | $x_0^{(2)}$ | $\leftarrow$ $x_0^{(3)}$ |
| $\leftarrow$ $x_1^{(0)}$ | $x_1^{(1)}$ | $x_1^{(2)}$ | $x_1^{(3)}$ |
| $x_2^{(0)}$ | $\leftarrow$ $x_2^{(1)}$ | $x_2^{(2)}$ | $x_2^{(3)}$ |
| $x_3^{(0)}$ | $x_3^{(1)}$ | $\leftarrow$ $x_3^{(2)}$ | $x_3^{(3)}$ |

| Proc. 0 | Proc. 1 | Proc. 2 | Proc. 3 |
|---|---|---|---|
| $x_0^{(0)}$ | $x_0^{(1)}$ | $\leftarrow \sum_{k=2:3} x_0^{(k)}$ | |
| | $x_1^{(1)}$ | $x_1^{(2)}$ | $\leftarrow \sum_{k=3:0} x_1^{(k)}$ |
| $\leftarrow \sum_{k=0:1} x_2^{(k)}$ | | $x_2^{(2)}$ | $x_2^{(3)}$ |
| $x_3^{(0)}$ | $\leftarrow \sum_{k=1:2} x_3^{(k)}$ | | $x_3^{(3)}$ |

| Proc. 0 | Proc. 1 | Proc. 2 | Proc. 3 |
|---|---|---|---|
| $x_0^{(0)}$ | $\leftarrow \sum_{k=1:3} x_0^{(k)}$ | | |
| | $x_1^{(1)}$ | $\leftarrow \sum_{k=2:0} x_1^{(k)}$ | |
| | | $x_2^{(2)}$ | $\leftarrow \sum_{k=3:1} x_2^{(k)}$ |
| $\leftarrow \sum_{k=0:2} x_3^{(k)}$ | | | $x_3^{(3)}$ |

| Proc. 0 | Proc. 1 | Proc. 2 | Proc. 3 |
|---|---|---|---|
| $x_0^{(0)}$ | $\leftarrow \sum_{k=1:3} x_0^{(k)}$ | | |
| | $x_1^{(1)}$ | $\leftarrow \sum_{k=2:0} x_1^{(k)}$ | |
| | | $x_2^{(2)}$ | $\leftarrow \sum_{k=3:1} x_2^{(k)}$ |
| $\leftarrow \sum_{k=0:2} x_3^{(k)}$ | | | $x_3^{(3)}$ |

| Proc. 0 | Proc. 1 | Proc. 2 | Proc. 3 |
|---|---|---|---|
| $\sum_{k=0:3} x_0^{(k)}$ | | | |
| | $\sum_{k=1:0} x_1^{(k)}$ | | |
| | | $\sum_{k=2:1} x_2^{(k)}$ | |
| | | | $\sum_{k=3:2} x_3^{(k)}$ |

Figure 1: Bucket algorithm for long-vector Reduce-scatter

### 4.4 Further Optimizations

#### Without Preposting Receives

As described above the short-vector algorithms has a cost of

$$2\lceil \log_2(p) \rceil \alpha + (\lceil \log_2(p) + \frac{p-1}{p}) n\beta + \lceil \log_2(p) \rceil n\gamma$$

. This cost is reduced by preposting receives. A discussion follows.

#### Preposting Receives

By preposting the receives we decrease the number of handshakes that is required to perform each send operation. If we have $v$ send operations then we have $3v$ handshakes if we do not prepost receives. Since in our improved implementations of the algorithms we do prepost the receives then there is only a cost incurred of $v$ handshakes per operation. This leaves us with a $2v$ handshake improvement. This $2v$ handshake improvement is gained from Sections 3.2 and 3.4 as was discussed previously. Modeling the formulas presented above we can account for this improved performance, where $\alpha_{\text{light}}$ is the new improved startup cost associated with the decreased handshakes. Therefore the cost of the short-vector algorithm is now:

$$2\lceil \log_2(p) \rceil \alpha_{\text{light}} + (\lceil \log_2(p) + \frac{p-1}{p}) n\beta + \lceil \log_2(p) \rceil n\gamma$$

and the cost of the long vector algorithm is now:

$$(p-1)(\alpha_{\text{light}} + \frac{n}{p}\beta + \frac{n}{p}\gamma) = (p-1)\alpha_{\text{light}} + \frac{p-1}{p} n\beta + \frac{p-1}{p} n\gamma.$$

Notice that given two different data distributions there is a critical choice to be made to determine which algorithm is to be used. The choice is dependent on the data volume, i.e whether there is a short-vector of data or a long-vector of data.

## 5 Performance

Here we describe the performance of the implementation of the algorithms. The architecture on which the implementation was ran is presented followed by the results attained from that architecture. The results are presented in two types of graphs both linear and loglog.

### 5.1 Target Architecture

The primary target architecture is a 300 node system consisting of the following.

1. Primary 2650 2-Way Compute nodes

- Vendor Dell
- Architecture 2650 Server
- Number of processors per node 2
- Operating System: Linux (RedHat 7.3, 2.4 kernel)
- Processor 2-2.4GHz Pentium 4 Processors
  (a) Main memory size: 1024M-bytes
  (b) Instruction cache size: 16K-bytes
  (c) Data cache size: 16K-bytes
  (d) Secondary unified instruction/data cache size 512K-bytes
- Myrinet 2000 M3F-PCI-64C Network Card(2Gb/s)

2. Secondary 6650 4-way SMP nodes

   - Vendor Dell
   - Architecture 6650 Server
   - Number of processors per node 4
   - Operating System: linux (RedHat 7.3, 2.4 Kernel)
   - Processor Description 4 1.6 Ghz Pentium 4 Processors, Main memory size: 4096M-bytes, Instruction cache size: 16K-bytes, Data cache size: 16K-bytes, Secondary unified instruction/data cache size: 256 K-bytes, Off chip level 3 cache size: 1MB
   - Myrinet 2000 M3F-PCI-64C Network Card(2Gb/s)

## 5.2 Results

Here we describe the graphs presented in Figures 2 through 7.

### Long-vector

In Figure 2 through Figure 7 the results of the long-vector algorithm are shown. Figure 2 presents the results for 128 processors and Figure 6 presents the results for 32 processors. The results are shown in the figures for the number of processor being that of 128, 64, and 32. In the odd numbered figures, i,e, 1, 3, and 5, the results are presented linearly. In the even numbered figures, i.e. 2, 4, and 6, the results are presented in a loglog graph. Notice the performance in the algorithms are much better demonstrated in the even numbered ( loglog ) graphs.

- The send-min line is the time for a "bounce". This is the shortest possible time for communication between two nodes using the blocking send and receive calls.
- The line labeled 'MPI' is for the MPICH public domain implementation. This is the implementation that was used in comparison of our implementations of the algorithms.

- The line labeled 'Bucket' is for the implementation used from previous research conducted by Dr. Robert van de Geijn. The implementation was used as a base for our research in this paper.
- The line labeled 'Bucket, optimized' is the line signifying the improved implementation.

Looking at Figure 7 one will notice the significant difference between the 'Bucket'(1) and the 'Bucket-optimized'(2) lines. The difference between (1) and (2) for very short vector lengths is approximately a factor of 2. This directly corresponds with our discussion of the decrease in the number of handshakes. As the vector length grows towards the limit one will notice that the algorithms tend to converge toward one another, because the $\alpha$ term becomes less significant. Also notice, around message length of $10^{2.5}$ there is an jump in the graph for (1). This is due to the fact that MPI, for short message lengths, allocates a buffer space that holds messages of the requested length. Therefore, the jump is caused from the fact that a new buffer space has to be allocated to hold the longer message lengths. There is not a jump in (2). This is in direct correspondence with the fact that by preposting the receives that the buffer space is already assumed to be allocated. At message length of $10^3$ the performance increase is now twice as much as before.

It should be noticed that by reducing the number of handshakes in the implementations that there is a factor of two difference. This corresponds to the $\alpha_{\text{light}}$ term given in our analysis. In the limit one would expect for all implementations to have the same performance.

# 6 Conclusion

Our research has shown that by taking an already existent algorithm and modifying the types of sends and receives one can attain a significant performance increase. The research also shows that the algorithms which we have implemented are better than vendor supplied algorithms. The algorithms presented are part of a large class of communication algorithms for which there have been shown efficient techniques for implementing these algorithms. Some of these techniques have been described. The improvement in the performance of these algorithms was attained through a change in the way the processing of the data was performed. Therefore, it has been shown that given a set of efficient algorithms implemented on a distributed-memory multidimensional-mesh architecture, a significant improvement in performance can be attained through the use of special types of send and receive messages. It should be noted that the vendor-supplied algorithms have still been unable to attain the same performance as the algorithms without the adjustments made. This is true even though a significant amount of papers on the subject have been published over the past decade.