MPI Reduce: From Current Approaches to a Greedy Algorithm

Ryan Tanaka

December 13, 2017

Abstract

Collective communication functions defined by the Message Passing Interface are often used in high performance computing workflows to orchestrate collective actions amongst groups of processes. The reduce operation is useful when developers need to combine data stored at each MPI process. Current reduce implementations are highly optimized to minimize the operation's execution time while maximizing network and process utilization. This paper explores three basic approaches: binomial, pipelined, and pipelined binary tree reductions. Both theoretical and empirical running times are discussed regarding these algorithms. Modern reduce algorithms utilize these basic approaches and captilize on their strengths. This paper also analyzes a new greedy pipelined reduction algorithm and empirically benchmarks it against current approaches. Findings show that binomial reductions are faster for smaller messages while pipelined and pipelined binary tree implementations are faster for larger messages. Furthermore, the greedy pipelined algorithm can be faster in all situations, however requires that an optimal message segment size be chosen.

1 Introduction

The Message Passing Interface, or MPI, is an established library standard that defines message passing behavior. It provides an agreed upon model for moving data from one process to another, which is useful for developers working on distributed memory environments. Two important types of communication patterns, point to point, and collective communications, are defined by MPI. The focus of this paper is on one specific collective communication operation, *reduce*. The reduce operation is used when individual MPI processes have data that needs to be combined via a commutative or associative operation, then returned to a designated root process where further computation may take place. This operation involves the scheduling of $(p-1)$ send/receive pairs where p is the number of processes involved in the operation. In the event that the message being transmitted as q segments, $(p-1)q$ send/receive pairs must be scheduled. A correct reduce algorithm properly applies the reduction to the data by a designated operation, does not deadlock (send/receive pairs are properly scheduled), and returns the globaly reduced message to the root process. If any of these conditions are not met, the algorithm is not correct. When a small number of processes are involved in a reduce operation, the effects of a naively implemented reduce algorithm may not be inherently visible as sheer computation power, high bandwidth, and low latencies may mask its suboptimal performance, however modern workloads demand collective communications to involve hundreds, even thousands of processes to be working on larger and larger message sizes through different physical network topologies and thus it is crucial that the right reduce algorithm is chosen

to avoid bottleknecks.

The goal of this study was to implement and compare basic MPI Reduce approaches with the greedy pipelined algorithm proposed by Bradley Lowery and Julien Langou [1]. Current MPI Reduce algorithms include binomial, pipelined , and pipelined binary tree reductions. These algorithms were implemented exactly as described by Lowery and Langou [1], Kumar, Sameh, and Nysal [2], and Thakur, Rabenseifner, and Gropp [3]. All algorithms, including the greedy algorithm, consist of MPI defined point to point communications: MPI Send and MPI Recv. These algorithms were then tested in a simulated environment using SMPI and SimGrid. This paper is laid out as follows. First, I introduce the communication and cost models used to evaluate the three reduction algorithms. Next, I provide an overview of how the greedy pipelined algorithm works. Then I describe the benchmark and subsequent findings.

2 Models

Reduce algorithms are executed in parallel by participating processes, therefore various models are needed to evaluate algorithm performance. These models provide a way to constrain process behavior, as well as provide good theoretical estimates for algorithm running time. For this study, I am assuming a *Unidirectional Communication Model* and use a *Linear Cost Model* to model theoretical performance.

Unidirectional Communication Model. This means that a process can only be in one of two states : sending or receiving. For example, a process may not partake in any point to point communication operation asynchronously such as sending while receiving data. For simplicity, algorithms are first developed assuming this model, then are extended to a *Bi-direction Communication Model* where a process may partake in the asynchronous action mentioned above.

Linear Cost Model. The *Linear Cost Model* takes into account three important variables which affect message passing: latency, bandwidth, and computation time. The time it takes for a message to be reduced and sent to another process is modeled by the following equation:

$$T = (\alpha + \beta m + \gamma m)$$

α denotes latency, or the time it takes for a packet to travel through the network before it reaches its destnation. βm, denotes the inverse of the system's bandwidth multiplied by the message size, m. βm represents the time it takes for message of size m to be pushed out onto the network before it can be sent off. γ denotes the time it takes to compute one unit of m. The γm term is important for evaluating a reduce because some type of operation needs to be applied to the data. However, in the case of a broadcast operation (effectively the inverse of a reduce), this term can be ignored. The γm term is also subject to various optmization techniques such as multi-threading, but for this evaluation I will assume γm to be 0 since processing power is much greater than bandwidth and latency.

Almost all reduction algorithms can be evaluated using the Linear Cost Model, however there are circumstances that require empirical analaysis to verify the performance of an algorithm. This is the case with the greedy pipelined reduction algorithm. According to Lowery and Langou [1], there is no closed form model to represent their algorithm and therefore the Linear Cost model will be used only in the analysis of the binomial, pipelined, and pipelined binary tree reduction algorithms.

3 Current Approaches

Binomial Reduction. The binomial reduction works as follows: half of the processes left to be reduced are assigned as sending processes while the other half are assigned as receiving processes. After the receiving processes have received and reduced their data, half of those receiving processes then become sending processes while the other half of the processes that had just received and reduced data from the previous step become receiving processes again. This repeats until there is one processes left (the root process). The flow of this algorithm resembles a binomial tree. In the case of a non power of two number of processes participating in the reduce, the algorithm will still resemble a binomial tree except the root node will have one subtree that is not a binomial tree. The following equation represents the theoretical running time of a binomial reduction:

$$T_{binomial} = \lceil \log_2(p) \rceil (\alpha + \beta m + \gamma m)$$

At most 2 processes can be reduced at a time (one processes sends to one receiving process), therefore given p processes, the most that can be reduced at one time in parallel is $p/2$. Lowery and Langou [1] point out that latency is much higher than bandwidth in practice and therefore we can expect a binomial reduction to be fast when $\alpha \gg \beta m$. This is because $\lceil \log_2(p) \rceil \alpha$ is optimal when performing a reduce. My implementation only works on a power of 2 number of processes and process 0 must be selected as the root node. In practice binomial reductions are implemented to work without those two contstrains.

Pipelined Reduction. The pipelined reduction works by splitting up the message into equal sized segments (last segment may be smaller) and sending them down a chain of processes arranged in a logical topology resembling a singly linked list. One end of the linked list is the tail which only sends segments. The other end of the linked list would be the root which will only be receiving and reducing. Processes in between both ends will be receiving, reducing, then sending its reduced message to the next process in the chain. The following equation represents the theoretical running time of a pipelined reduction:

$$T_{pipeline} = ((p-1) + 2(q-1))(\alpha + \beta m + \gamma m)$$

Given p processes, it will take $(p-1)$ steps for the root to finish reducing the first segment and $2(q-1)$ steps to finish reducing the remaining segments, where q is the number of segments. The constant factor of 2 takes into account the fact that there will be unused links in between the nodes of the linked list like logical topology since we are assuming a Unidirectional Communication Model. When $\alpha \ll \beta m$, pipelining helps to reduce the βm term by using smaller messages. For my implementation, the root does not need to be 0.

Pipelined Binary Tree Reduction. The pipelined binary tree reduction is similar to the pipelined reduction, however instead of arranging the processes in a single line, a binary tree topology is used instead. This has the benefit of reducing the amount of steps it takes for the root node to finish reducing the first segment, however takes more time than the pipelined reduction in terms of the amount of time it takes for the root to finish reducing the remaining segments. With this implementation, leaf nodes send segments to their parents. The root node receives from its left child, reduces, receives from its right child, then reduces. The nodes in between behave like the root node, except after reducing both of its children, it then sends the reduced segment off to its parent. The following equation represents the theoretical running time of a pipelined binary tree reduction:

$$T_{binary} = 2(\lceil \log_2(p+1) \rceil - 1)(\alpha + \beta m + \gamma m) + 4(q-1)(\alpha + \beta m + \gamma m)$$

After the root node has finished reducing its first segment, it has to wait $4(q-1)(\alpha+\beta m+\gamma m)$ to finish reducing its next segment. This is twice as bad as the pipelined reduction, however the pipelined binary tree reduction will start working on subsequent segments after the first much faster than the pipelined reduction. As the number of processes gets very large, the $2(\lceil \log_2 (p+1) \rceil - 1)(\alpha+\beta m+\gamma m)$ factor in T_{binary} keeps this startup time minimized. It is important to note that the above formula slightly overestimates the time it takes for T_{binary}. Lowery and Langou [1] provide this forumula, and the $4(q-1)(\alpha+\beta m+\gamma m)$ term is true in that subsequent segments after the first take that amount of time to reach the root, however there is a unit of time that overlaps when observing the algorithm in practice. For example, as the root receives a segment from its right child, the root's left child's starts receiving from its left child in parallel (this does not violate the Unidirectional Communication Model). Nonetheless, the formula above gives a good estimate of T_{binary}'s theoretical running time. The root does not need to be 0 in my implementation and p is not restricted to being $2^h - 1$ for any h.

In the case of a pipelined k-ary tree reduction, I would expect its theoretical running time to be bounded by the following formula:

$$T_{k-ary} = k(\lceil \log_k (p+1) \rceil - 1)(\alpha + \beta m + \gamma m) + 2k(q-1)(\alpha + \beta m + \gamma m)$$

The number of steps it would take for the root to finish the first segment would be further minimized, however the constant factor k introduced by doing so might outweigh the benefits of the $k(\lceil \log_k (p+1) \rceil - 1)$ term.

4 Greedy Algorithm by B. Lowery and J. Langou

The greedy pipelined reduction algorithm presented by Lowery and Langou [1] uses ideas from both pipelined and binomial reduction algorithms. Their algorithm is claimed to outperform the previously mentioned algorithms when tested theoretically for medium sized messages. As the number of processes increases, the range of message sizes where this algorithm exceeds also increases. For this study, I only examined the algorithm with q = 1,2,4,8 or 16, however in their study, the authors conducted a number of experiments on a 64 process platform to determine optimal segment sizes.

When the greedy algorithm is used with $q = 1$ (no segmentation/pipelining), it behaves exactly like a binomial reduction. Empirical findings from my tests confirmed that this is true. When segmentation is used, the algorithm does a binomial reduction on the first segment. During this time, a greedy choice is made by the algorithm to assign the sending processes of the first segment engage in a binomial type reduction on the second segment amongst themselves. This process continues until all segments have been reduced. The algorithm also is restricted to reducing segments in order. For example, a process cannot reduce segment s_2 before it has completed its reduction of segment s_1 . In order to make the greedy choice of which segment to send to what process, each process has a history array allocated on the heap at the start of the reduction. The size of this array is equal to the number of segments, and the values in the array are initialized to the number of processes participating in the reduction. Start and end counters are also initialized to keep track of values in the array. Values in this array will indicate what segment has been sent to what process, and therefore the process can make a decision as to which segment to send to what process at a given point in time. Once a value in the history array is decremented to 0, that segment is said to be completely done. When the last value in the array is 0, the process is done with the reduce operation. In their proof of optimality, the authors refer to these values in the history array as "state vectors"

where the ending state of one vector can be used as the initial state vector of the following segment [1].

The advantage that this algorithm has over other implementations is that the time it takes for the first segment to be reduced is $\lceil \log_2(p) \rceil (\alpha + \beta s_1 + \gamma s_1)$, therefore subsequent segments can be started on by the root process faster than with the pipelined and pipelined binary tree reduction. In a nutshell, this algorithm is a clever way of using binomial reductions and pipelining together in a way that may lead to performance improvements given the right segment sizes.

5 Benchmark

All algorithms were tested in a simulated environment using SMPI and SimGrid. The simulated cluster consisted of 64 nodes (1 MPI process per node) connected by a crossbar switch. Latency was set to 20 microseconds and bandwidth set to 10Gbps. Each reduce algorithm was called and had its execution time measured using the function MPIWtime(). The operation used in this test was a SUM on INT values. Message sizes used in the test ranged from 64 bytes to about 67 megabytes and were incremented in powers of 2 ($2^6, 2^7..2^{26}$ bytes). Messages transmitted by the greedy, pipeline, and binary tree implementations can be segmented into q equi segments and therefore were tested using 1, 2, 4, 8, and 16 equi segments.

6 Findings

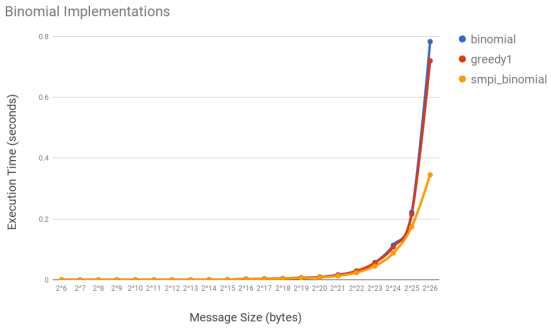

Figure 1: Binomial reduction execution times.

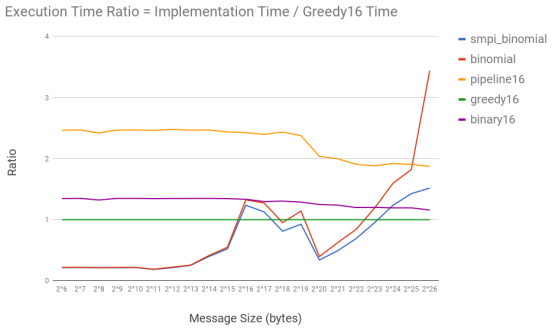

Figure 2: Comparison of other algorithms against the greedy algorithm with $q = 16$

Figure 1 illustrates execution times of the binomial reduction, greedy reduction with 1 segment, and SMPI binomial reduction tested on message sizes mentioned in the Benchmark section. This demonstrates that the greedy reduction with one segment (no pipelining) does in fact behave just like a standard binomial reduction. The SMPI implementation seems to behave the same, however as message sizes increase, it appears to perform much better than both the standard and greedy single segment implementations. A brief look at the source code hosted on Github leads me to believe that this may be due to the fact that the implementation uses memcpy in the function Datatype::copy(), however cannot say for certain until tested.

Figure 2 uses the greedy algorithm with 16 equi segments as a baseline and compares its performance to the other algorithms. These results were computed by taking the implementation time of the other algorithms and dividing it by the implementation time of the greedy algorithm with 16 equi segments. 16 equi segments were also used for the pipelined and pipelined binary tree reductions. The pipelined algorithm performed the worse for smaller message sizes as expected but does better for larger message sizes. I would expect it to approach the binary pipelined algorithm eventually due to the fact that the pipelined algorithm has a $2(q-1)(\alpha+\beta m+\gamma m)$ value in its theoretical running time while the binary pipelined algorithm has a $4(q-1)(\alpha+\beta m+\gamma m)$ value. The binary tree pipelined algorithm performed better than the pipeline algorithm because of the reduced startup time for the root to start reducing segments after the first. In regards to the binomial algorithm implementations, they performed better on the platform for smaller message sizes but worse for larger message sizes as expected. The greedy algorithm shows performance improvements over the other implementations for message sizes greater than 2^{23} bytes. Based on the graph, it is clear that the greedy algorithm could perform just as good or better than all other implementations if optimal segment sizes were chosen for each message size. For example, using a single segment for smaller messages would clearly cause the greedy algorithm to perform just as good as the binomial implementations.

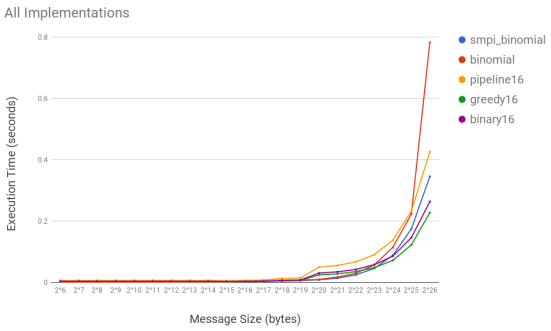

Figure 3: All execution times.

Figure 3 displays the execution times of all implementations (data from 1, 2, 4, and 8 equi segments can be found at the end of the paper). HPC systems are so fast that differences in performance may only become visible for very large message sizes. It can more insightful to use any single algorithm as a baseline to determine performance instead of just evaluating execution times.

7 Conclusion

This paper illustrated the theoretical and empirical running times of current MPI Reduce implementations as well as a new greedy pipelined approach. Each implementation performed well in different scenarios as demonstrated in the empirical findings section. Theoretically, it is also possible to predict the expected performance of these algorithms using the linear cost model. For complicated approaches such as the greedy pipelined algorithm, it is necessary to evaluate its performance based on empirical findings. To summarize my findings, a binomial reduction is an ideal choice for smaller message sizes while a pipelined approach is ideal for larger message sizes. With algorithms that use pipelining, segment sizes need to be considered carefully in order to perform well. A bad choice of

segment sizes will result in a poorly performing reduction. The findings of this paper provide some validation for the claims made by Lowery and Langou in regards to the performance of their greedy algorithm.

The empirical findings in this paper were limited in that the benchmark was run with only 64 MPI processes. Ideally, a larger (and smaller) number of processes need to be evaluated to better understand these algorithms. Furthermore, pipelined algorithms were tested using only equi segments where q = 1, 2, 4, 8, and 16, thus covering a minute set of possible segment arrangements. Its clear that variable sized segments can make the most out of these algorithms and therefore should be tested as well. Additionally, a Unidirectional Communication Model was used for simplicity, however in practice asynchronous sends and receives may be used to further increase performance. Lowery and Langou [1] address this in their paper by extending the algorithm to assume a Bidirectional Communication Model.

This paper merely scratched the surface of only one aspect of MPI collective communications, however provides some insight in regards to the trade offs that are taken into account when using one reduction technique over another.

References

[1] B. R. Lowery and J. Langou, "A greedy algorithm for optimally pipelining a reduction," Oct. 2013.

[2] S. Kumar, S. Sharkawi, and N. J. K. A, "Optimization and analysis of mpi collective communication on fat-tree networks," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*, (Chicago Illinois), pp. 1031–1040, IEEE, 2016.

[3] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *International Journal of High Performance Computing Applications*, vol. 19, Feb. 2005.