A Greedy Algorithm for Optimally Pipelining a Reduction

Bradley R. Lowery* and Julien Langou*
Mathematical and Statistical Sciences
University of Colorado Denver
Denver, CO, USA
{Bradley.Lowery, Julien.Langou}@ucdenver.edu

*Research of this author was fully supported by the National Science Foundation
grant # NSF CCF 1054864.

Abstract—Collective communications are ubiquitous in parallel applications. We present two new algorithms for performing a reduction. The operation associated with our reduction needs to be associative and commutative. The two algorithms are developed under two different communication models (unidirectional and bidirectional). Both algorithms use a greedy scheduling scheme. For a unidirectional, fully connected network, we prove that our greedy algorithm is optimal when some realistic assumptions are respected. Previous algorithms fit the same assumptions and are only appropriate for some given configurations. Our algorithm is optimal for all configurations. We note that there are some configuration where our greedy algorithm significantly outperform any existing algorithms. This result represents a contribution to the state-of-the-art. For a bidirectional, fully connected network, we present a different greedy algorithm. We verify by experimental simulations that our algorithm matches the time complexity of an optimal broadcast (with addition of the computation). Beside reversing an optimal broadcast algorithm, the greedy algorithm is the first known reduction algorithm to experimentally attain this time complexity. Simulations show that this greedy algorithm performs well in practice, outperforming any state-of-the-art reduction algorithms. Positive experiments on a parallel distributed machine are also presented.

Index Terms—reduction; reduce; collective; pipelining; message passing; communication;

I. INTRODUCTION

Applications relying on parallel distributed communication require the use of collective communications. In this paper, we focus on the reduction operation (MPI_Reduce) in which each process holds a piece of data and these pieces of data need to be combined using an associative operation to form the results on the root process. We present two new algorithms for performing a reduction. The operation associated with our reduction needs to be associative and commutative. The two algorithms are developed under two different communication models (unidirectional and bidirectional). Both algorithms use a greedy scheduling scheme.

Collective communications like reduction can often be the bottleneck of massively parallel application codes, therefore optimizing collective communications can greatly improve the performance of such application codes. The performance of a reduction algorithm is highly dependent on the machine parameters (e.g., network topology) and the underlying architecture; this makes the systematic optimization of a collective communication for a given machine a real challenge. For a unidirectional, fully connected network, we prove that our greedy algorithm is optimal when some realistic assumptions are respected. Previous algorithms fit the same assumptions and are only appropriate for some given configurations. Our algorithm is optimal for all configurations. We note that there are some configuration where our greedy algorithm significantly outperform any existing algorithms. This result represents a contribution to the state-of-the-art. For a bidirectional, fully connected network, we present a different greedy algorithm. We verify by experimental simulations that our algorithm matches the time complexity of an optimal broadcast (with addition of the computation). Beside reversing an optimal broadcast algorithm, the greedy algorithm is the first known reduction algorithm to experimentally attain this time complexity. Simulations show that this greedy algorithm performs well in practice, outperforming any state-of-the-art reduction algorithms. With respect to practical application, much work is still needed in term of auto-tuning and configuring for various architectures (nodes of multi-core, multi-port networks, etc.).

In the unidirectional context, we compared the greedy algorithm (uni-greedy) with three standard algorithms (binomial, pipeline, and binary) using a linear model to represent the point-to-point cost of communicating between processors. Unlike the standard algorithms, no closed-form expression exists for the completion time of the greedy algorithm; therefore, simulations are used to compare the algorithms. While we know that our algorithm is optimal in this context, these simulations
indicate the performance gain of the new algorithm over classic ones. In particular, for mid-size messages, the new algorithm is about 50% faster than pipeline, binary tree, or binomial tree.

In the bidirectional context, we again compare the greedy algorithm (bi-greedy) with the three standard algorithms as well as a reduce-scatter/gather (butterfly) algorithm. For the standard algorithms we see similar results as the unidirectional case. The butterfly algorithm performs well for mid-size messages, but has poor asymptotic behavior.

In the experimental context, we have implemented the binomial, pipeline, uni-greedy, and bi-greedy algorithms using OpenMPI version 1.4.3; all algorithms are implemented using point-to-point MPI functions: MPI_Send and MPI_Recv, or MPI_Ssendrecv. Moreover, the OpenMPI library provides a state-of-the-art implementation for a reduction. We also, compare with an implementation of the butterfly algorithm. Numerical comparisons of the greedy algorithm to OpenMPI’s built in function MPI_Reduce as well as the binomial and pipeline algorithms show the greedy algorithm is the best for medium size messages, confirming that the results found theoretically applies in our experimental context. However, the more simplistic algorithms (binomial and pipeline) perform better for small and large messages, respectively. Finally, the implementation of the butterfly algorithm exhibits similar results as in the theoretical context.

Finally, the idea of unequal segmentation is considered. Typically, when a message is split into segments during a reduction operation, the segments are assumed to be of equal size. We investigate if the greedy and pipeline algorithms can be improved by allowing for segments to have unequal sizes. It turns out that, for some parameter values, the greedy algorithm (in a unidirectional system) was optimized by unequal segmentations. This indicates that removing the equal segmentation assumption from our theory can lead to better algorithms. However, the gains obtained were marginal. We also note that the pipeline algorithm was always optimized by equal segmentation.

II. MODEL

Unidirectional and bidirectional systems are two ways to describe how processors are allowed to communicate with each other. In a unidirectional system, a processor can only send a message or receive a message at a given time, but not both. A bidirectional system assumes that at a given time a processor may receive a message from a processor and send a message to another (potentially different) processor simultaneously. We assume an unidirectional system for the initial sections and optimality proof (Sections IV and V). In Section VI we adapt the algorithm to a bidirectional system.

The parallel system contains \( p \) processors indexed from \( 0 \) to \( p - 1 \). Additionally, we will assume the system is fully connected (every processor has a direct connection to all other processors) and homogeneous. These assumptions are not verified in practice on current architecture, however they represent a standard theoretical framework and our experiments indicate that they are valid enough in practice to develop useful algorithms.

A linear model (Hockney [1]) is used to represent the cost of a point-to-point communication. The time required for each communication is given by \( \alpha + \beta m \), where \( \alpha \) is the latency (start up), \( \beta \) is the inverse bandwidth (time to send one element of the message), and \( m \) is the size of the message. Since we are performing a reduction, computations are also involved so we will also assume the time for computation follows a linear model and is given by \( \gamma m \), where \( \gamma \) is the computation time for one element of the message. We will investigate the theoretical case when \( \gamma = 0 \) (i.e., very fast processing units) as well as for nonzero \( \gamma \). Furthermore, we will assume that communication and computation can not overlap.

To achieve better performance the messages can be split into \( q \) segments of size \( s_1, \ldots, s_q \). In Section V we show the greedy algorithm is optimal for any segmentation. To obtain the best performance of a given algorithm an optimal segmentation is determined. In Section V-B we restrict the optimization to equi-segmentation (\( s_i = s, \forall i \)), except the last segment may possibly be smaller.

III. COLLECTIVE COMMUNICATIONS AND RELATED WORK

A collective communication operation is an operation on data distributed between a set of processors (a collective). Examples of collective communications are broadcast, reduce (all-to-one), reduce (all-to-all), scatter, and gather.

**Broadcast:** Data on one processor (the root) is distributed to the other processors in the collective.

**Reduce:** Each processor in a collective has data that is combined entry-wise and the result is stored on the root processor.

**All Reduce:** Same as reduce except that all processors in the collective contain a copy of the result.

**Scatter:** Data that is on the root processor is split into pieces and the pieces are distributed between the processors in the collective. The result is each processor (including the root processor) contains a piece of the original data.

**Gather:** The reverse operation of a scatter.
Optimizing the reduction operation is closely related to optimization of the broadcast operation. Any broadcast algorithm can be reversed to perform a reduction. Bar-Noy et al. [2] and Träff and Ripke [3] both provide algorithms that produce an optimal broadcast schedule for a bidirectional system. Here the messages are split into segments and the segments are broadcast in rounds. In both cases the optimality is in the sense that the algorithm meets the lower bound on the number of communication rounds. For the theoretical case when $\gamma = 0$, reversing an optimal broadcast will provide an optimal reduction. However, for $\gamma \neq 0$ this is no longer valid as an optimal schedule will most like take into account the computation.

Rabenseifner [4] provides a reduce-scatter/gather algorithm (butterfly) which provides optimal load balancing to minimize the computation. Rabenseifner does not predefine a segmentation of the message, but rather uses the techniques recursive-halving and recursive-doubling. The algorithm is done in two phases. In the first phase the message is repeatedly halved in size and exchanged among processes. At the end of the first phase the final result is distributed among all the processors. This phase is known as a reduce-scatter. The second phase gathers the results recursively doubling the size of the message. In [5] Rabenseifner and Träff improve on the algorithm for non-power of two number of processors.

Sanders et al. [6] introduce an algorithm to schedule to a reduction (or broadcast) using two binary trees. The authors notice that two binary trees could be use simultaneously and effectively reduce the bandwidth by a factor of two from a single binary tree. The time complexity approaches the lower bound for large messages. However, for small messages the latency term is twice larger than optimal. Also, the time complexity is never better than that of a reverse-optimal broadcast.

Other models have been used to describe more complex machine architectures. Heterogeneous networks, where processor characteristics are composed of different communication and computation rates, have been considered. Beaumont et al. [7], [8] consider optimizing a broadcast and Legrand et al. [9] consider optimizing scatter and reduce. In these papers, the problem is formulated as a linear program and solved to maximize the throughput (number of messages broadcasting per time unit). Also, higher dimensional systems have also been considered [10]. Here a processor can communicate with more than one processor at a time.

The machine parameters and architecture vary between machines and one algorithm may perform better on one machine versus another. Accurately determining machine parameters can be a difficult task. In practice, auto-tuning for each machine is required to obtain a well-performing algorithm. Vadhiyar et al. [11] discuss how to experimentally determine the optimal algorithm. Pjesivac-Grbović et al. [12] compare various models that can be used to auto-tune a machine. These models are more complex than the Hockney model and can provide a more accurate model for the communication cost. However, a linear model provides a good basis for theoretically comparing algorithms. Other models, such as LogP, LogGP, and PLogP, each provided useful information to help auto-tune a machine.

IV. STANDARD ALGORITHMS

Before introducing the greedy algorithms it is helpful to review three standard algorithms (binomial, pipeline, and binary).

Since we are using a unidirectional system, we will call a processor either a sending processor or a receiving processor. Sending and receiving processors are paired together to perform a reduction. After a receiving processor receives a segment it will combine its segment with the segment received. After a processor sends a segment it is said to be reduced for that segment. For the reduction to be completed each processor (except the root) must be reduced for each segment. The three algorithms are described below.

a) Binomial: At a given time suppose there are $n$ processors left to be reduced for a message, then $\lceil n/2 \rceil$ processors are assigned as sending processors and $\lfloor n/2 \rfloor$ are assigned as receiving processors. Afterwards there will be $\lceil n/2 \rceil$ processors left for reduction and the process is repeat until $n = 1$. Segmenting the message would only increase the latency of the communication time since all processors must be finished before the next segment can be started. Therefore, we do not consider segmentation.

b) Pipeline: At a given time suppose that processors $k + 1$ to $p - 1$ have been reduced for segment $s_i$. Processor $k$ is assigned as a sending processor for segment $s_i$ and processor $k - 1$ is assigned as a receiving processor for segment $s_i$.

c) Binary: At a given time suppose there are $2^n - 1$ processors left to be reduced for segment $s_i$, then $2^{n-1}$ processors are assigned as sending processors and $2^{n-2}$ processors are assigned as receiving processors. If $p \neq 2^n - 1$ for some $n$, then the initial step in the algorithm reduces the “extra” processors so that in the next step there is $2^n - 1$, for some $n$, processors left for reduction. Since there are twice as many sending processors as receiving processors, two processors will send to the same receiving processor and the time for one step in the binary algorithm is $2(\alpha + \beta s + \gamma s)$.

A. Lower bounds and optimality of the segment size $s$.

Given $p, \alpha, \beta, \gamma, m,$ and $s$, the time for a reduction can be calculated for each algorithm. Using these formulae
TABLE I: Time analysis for standard algorithms in a unidirectional system, where \( N = \lceil \log_2(p + 1) \rceil \), \( s_{\text{opt}} \) is the optimal equi-segment size, and \( T_{\text{opt}} \) is the time for the algorithm at \( s_{\text{opt}} \). Formulae are valid for \( p > 3 \).

| Method   | Time                                      | \( s_{\text{opt}} \)            | \( T_{\text{opt}} \)            |
|----------|-------------------------------------------|----------------------------------|----------------------------------|
| Binomial | \( (p - 1)(\alpha + \beta s + \gamma s) + 2(q - 1)(\alpha + \beta s + \gamma s) \) | \( \left( \frac{2\alpha}{(p - 3)(\beta + \gamma)} \right)^{1/2} \) | \( \left( (p - 3)(\alpha)^{1/2} + (2m(\beta + \gamma))^{1/2} \right)^2 \) |
| Pipeline | \( (p - 1)(\alpha + \beta s + \gamma s) + 2(q - 1)(\alpha + \beta s + \gamma s) \) | \( \left( \frac{2\alpha}{(p - 3)(\beta + \gamma)} \right)^{1/2} \) | \( \left( (N - 3)(\alpha)^{1/2} + (2m(\beta + \gamma))^{1/2} \right)^2 \) |
| Binary   | \( 2\left( \lceil \log_2(p + 1) \rceil - 1 \right)\left( \alpha + \beta s + \gamma s \right) + 4(q - 1)(\alpha + \beta s + \gamma s) \) | \( \left( \frac{2\alpha}{(N - 3)(\beta + \gamma)} \right)^{1/2} \) | \( 2 \left( (N - 3)(\alpha)^{1/2} + (2m(\beta + \gamma))^{1/2} \right)^2 \) |

TABLE II: Lower bounds for each term in the time for a reduction as well as the lower bounds for each standard algorithm in a unidirectional system, where \( N = \lceil \log_2(p + 1) \rceil \). Formulae are valid for \( p > 2 \).

| Method   | Latency  | Bandwidth | Computation |
|----------|----------|-----------|-------------|
| Reduce   | \( \lceil \log_2 p \rceil \alpha \) | \( 2m\beta \) | \( \frac{p - 1}{p} m\gamma \) |
| Binomial | \( \lceil \log_2 p \rceil \alpha \) | \( \lceil \log_2 p \rceil m\beta \) | \( \lceil \log_2 p \rceil m\gamma \) |
| Pipeline | \( (p - 1)\alpha \) | \( (p + 2m - 3)\beta \) | \( (p + 2m - 3)\gamma \) |
| Binary   | \( 2(N - 1)\alpha \) | \( 2(N + 2m - 3)\beta \) | \( 2(N + 2m - 3)\gamma \) |

The optimal segmentation size \( s_{\text{opt}} \) and optimal time \( T_{\text{opt}} \) can also be calculated when calculating the optimal segmentation size we assume that the segments are of equal size. A summary of these values is given in Table I. It is unknown what the optimal segmentation would be if the segments are of unequal size. The formulae for time are upper bounds in the case when \( m \) is not divisible by \( s \). These formulae differ from the formulae that appear in [3, 12] by a factor of 2 in the bandwidth term since the authors assume a bidirectional system. In a unidirectional system there is extra lag time.

To better understand the algorithms it is helpful to know the lower bounds for each term: \( \alpha \), \( \beta \), and \( \gamma \). The lower bounds for any reduction algorithm as well as other collective communications are discussed in [13]. The following rationales are used to obtain the lower bounds.

**d) Latency:** Every processor in a collective has at least one message that must be communicated. At any time step, at most half of the remaining processors can send their messages to another processor to perform the reduction. This gives the lower bound \( \lceil \log_2 p \rceil \alpha \).

**e) Bandwidth:** Consider the case when \( p = 3 \). Each processor must be involved in a communication for each element of the message. When two processors are involved in a communication the third processor is idle. Therefore, there must be two communications performed for each element of the message, which cannot occur simultaneously. This gives the lower bound \( 2m\beta \). For \( p > 3 \), the lower bound holds since a message of the same size on more processors can not be communicated faster. \( p = 2 \) is a special case with a bandwidth term of \( m\beta \).

**f) Communication:** Each element of the message must be involved in \( p - 1 \) computations. Distributing the communications evenly among all the processors gives the lower bound \( \frac{p - 1}{p} m\gamma \).

The lower bounds for the standard algorithms can be obtained using the following rationale. The latency term will be minimized when no segmentation is used \( (q = 1) \), since any segmentation will only increase the number of time steps and hence increase the latency. The bandwidth and computation terms will be minimized when \( s = 1 \), since this will maximize the time when multiple processors are working simultaneously. Table II shows the lower bounds for any reduction algorithm and the lower bounds for each of the standard algorithms.

Given machine parameters \( \alpha \), \( \beta \), and \( \gamma \), one can select the algorithm that provides the best performance. In practice, the latency is much larger than the bandwidth \( (\alpha \gg \beta) \). For small messages, \( \alpha \gg m \), the time for a reduction is dominated by the latency. The binomial algorithm minimizes the latency term and will provide the best performance for small messages. For large messages, \( \alpha \ll \beta m \), the time for a reduction is dominated by the bandwidth. The pipeline algorithm provides the smallest bandwidth term and will therefore give the best performance for large messages. For medium size messages, the binary algorithm will provide the best performance. The binary algorithm has both qualities that give binomial and pipeline good lower bounds for latency and bandwidth, respectively, only differing by a
factor of two.

The values of \( m \) that are considered “small” and “large” depend on the machine parameters (\( \alpha, \beta, \gamma \), and \( p \)). Figure 1 shows which algorithm is best for \( \alpha = 10, \beta = 1, \gamma = 0 \) as a function of the number of processors \( p \) and the message size \( m \). Here we are assuming that there is no time required for the computation (\( \gamma = 0 \)).

\[
\frac{\text{Minimum Standard Algorithm}}{1024} \quad \frac{\text{Binomial}}{512} \quad \frac{\text{Binary}}{256} \quad \frac{\text{Pipeline}}{128} \quad 64 \quad 32 \quad 16 \quad 8 \quad 4 \quad 2 \quad 1
\]

Since we assume that each processor must reduce the segments in order, then a reduction can be scheduled considering one segment at a time. Let \( I \) be a set of \( n \) processors that remain to send a given segment (or for the root, receive the final message). Let \( x_i \) be the time that processor \( i \) completed its previous task. We define the state of these processors as

\[
X^{(n)} = \{ x_i | i \in I \}.
\]

The state of the processors that have sent the segment is \( S^{(n)} \). The permutation \( P(X^{(n)}) \) orders the elements of \( X^{(n)} \) from smallest to largest. The time complexity of an algorithm is given by the final state of the root processor. A schedule for a send/receive pair is given by

\[
(X_k^{(n-1)}, S_k^{(n-1)}) = \text{reduce}(X_k^{(n)}, S_k^{(n)}, a, b, t).
\]

Processor \( a \) sends segment \( k \) to processor \( b \) starting at time \( t \). \( X_k^{(n-1)} \) is the updated state of the processors that remain to send segment \( k \) and \( S_k^{(n-1)} \) is the state of the processors that have sent segment \( k \). We will assume that successive send/receive pairs are scheduled in the order that the communications are started. This way the state of a processor that is not a member of the send/receive pair, but whose current state is less than \( t \) is set equal to \( t \).

**Lemma 1.** \( \text{reduce}(X^{(n)}, S^{(n)}, a, b, t) \), where \( P(X^{(n)}) = (x_{g_1}, x_{g_2}, \ldots, x_{g_n}) \), is a valid send/receive pair if and only if

(i) \( x_a \leq t \) and \( x_b \leq t \);
(ii) \( a \neq 0 \); and
(iii) \( t \geq x_{g_2} \).

**Proof:** (i) requires that processor \( a \) and \( b \) have completed their previous task before \( t \). (ii) states that the root can never be a sending processor. And (iii) implies (iii).

A greedy algorithm is obtained by scheduling every send/receive pair using \( t = x_{g_2} \) in Lemma 1. When a segment is done being scheduled then the final state of the processors are used as the initial state for the next segment. Algorithm 1 produces the unidirectional greedy (uni-greedy) schedule. A proof of optimality is given in Section V-A.

We give the C code for the uni-greedy algorithm on the next page in Figure 2. In this code, we are able to generate the greedy schedule dynamically during the reduction rather than precomputing the schedule by Algorithm 1. The algorithm assumes that MPI\_Op is commutative (and associative as well, of course). The algorithm allocates two extra buffers of size segment size. The variable \( s \) is the size of a segment and needs to be initialized (if possible “tuned”) in advance. The implementation is restricted to root being 0,
#include "reduce.h"

int global_s;

int Reduce_greedy(void *sendbuf_notype, void *recvbuf_notype, int m,
        MPI_Datatype mpi_datatype, MPI_Op mpi_op, int root, MPI_Comm mpi_comm){
    int pool_size;
    int my_rank;
    int j, i, q, s2, s3,qstart, qend, snext;
    int s=global_s;
    MPI_Status status;
    int *tempbuf, *tempbuf2;
    int *sendbuf, *recvbuf;

    sendbuf = (int *) sendbuf_notype;
    recvbuf = (int *) recvbuf_notype;

    if (root != 0) MPI_Abort( mpi_comm, 512);
    MPI_Comm_size(mpi_comm, &pool_size); MPI_Comm_rank(mpi_comm, &my_rank);

    if (my_rank != 0) tempbuf = (int*)malloc(s*sizeof(int));
    tempbuf2 = (int*)malloc(s*sizeof(int));

    if ( my_rank == 0)
        for(i=0;i<m;i++) recvbuf[i] = sendbuf[i];
    else
        for(i=0;i<s;i++) tempbuf[i] = sendbuf[i];

    q = m/s;
    if((m % s) != 0){ s2 = m % s; q++;
    } else
        s2 = s;

    hist = (int*)malloc(q*sizeof(int));
    for(i = 0; i < q; i++) hist[i] = pool_size;
    qstart = 0;
    qend = 0;

    while(hist[q-1] > 1){
        if(qstart == q-1 && hist[qstart-1]-hist[qstart] > 1){ qstart++; }
        for(i = qstart; i >= qend; i--){
            z = hist[i]/2;
            if(my_rank < hist[i] - z && my_rank >= hist[i] - 2*z) {
                MPI_Send(tempbuf, s2, mpi_datatype, my_rank-z,
                mpi_op, 512, mpi_comm);
            }
            else {
                MPI_Recv( tempbuf2, s2, mpi_datatype, my_rank+z,
                mpi_op, 512, mpi_comm, &status );
            }

            if(my_rank < hist[i] - z && my_rank >= hist[i] - 2*z) {
                recvbuf[i*s] += tempbuf2[j];
            }
            if(my_rank == 0) {
                for(j = 0; j < s; j++ ) tempbuf[i*s + j] += tempbuf2[j];
            }
            else {
                for( j = 0; j < s; j++ ) tempbuf[j] += tempbuf2[j];
            }
        }
        hist[i] -= z;
        if(hist[i] == 1){ hist[i] = 0; qend++; }
    }

    free( hist ); if(my_rank != 0) free( tempbuf ); free( tempbuf2 );
    return 0;
}

Fig. 2: C code for the uni-greedy reduction algorithm. The algorithm assumes that MPI_Op is commutative (and associative as well, of course). The algorithm allocates two extra buffers of size segment size. The global variable global_s is the size of a segment and needs to be initialized (if possible “tuned”) in advance. The implementation is restricted to root being 0, MPI_Datatype being int, and MPI_Op being +. These restrictions are not a consequence of the algorithm and can be removed.
In this section, we prove that the uni-greedy algorithm is optimal. The proof is a variation of the proof in [14]. Using the notation as in Lemma 1, we define

\[
\text{optreduce}(X^{(n)}) = \text{reduce}(X^{(n)}, S^{(n)}, g_1, g_2, x_{g_2}).
\]

We also define the partial ordering on \(\mathbb{R}^n\):

\[
X \leq Y \iff x_i \leq y_i, \ 1 \leq i \leq n.
\]

**Lemma 2.** Let \(X \in \mathbb{R}^n\) and \(Y \in \mathbb{R}^n\) be nondecreasing such that \(X \leq Y\) and let \(a \in \mathbb{R}\) and \(b \in \mathbb{R}\) such that \(a \leq b\). If \(x_k \leq a \leq x_{k+1}\) and \(y_l \leq b \leq y_{l+1}\), then

\[
(x_1, \ldots, x_k, a, x_{k+1}, \ldots, x_n) \leq (y_1, \ldots, y_l, b, y_{l+1}, \ldots, y_n).
\]

**Proof:** \(\forall i \text{ s.t. } i \leq \min(k, l),\)

\[
x_i \leq y_i.
\]

\(\forall i \text{ s.t. } i \geq \max(k, l) + 1,\)

\[
x_i \leq y_i.
\]

If \(k = l\), then \(a \leq b\) is the only other comparison needed.

If \(k < l\), then

\[
a \leq x_{k+1} \leq y_{k+1},
\]

\[
x_l \leq y_l \leq b,
\]

and \(\forall i \text{ s.t. } k < i < l,\)

\[
x_i \leq y_i \leq y_{i+1}.
\]

If \(k > l\), then

\[
a \geq x_k \geq x_{l+1} \geq y_{l+1} \geq b.
\]

By assumption \(a \leq b\), therefore \(a = b\). Hence,

\[
x_{l+1} = a = b,
\]

\[
a = x_k \leq y_k,
\]

and \(\forall i \text{ s.t. } l + 1 < i \leq k,\)

\[
x_i = a = b \leq y_{i-1}.
\]

Hence, for all cases the inequality is satisfied. \(\blacksquare\)
Lemma 3. Let 
\[(X^{(n-1)}, S^{(n-1)}) = \text{optreduce}(X^{(n)})\]
and 
\[(Y^{(n-1)}, R^{(n-1)}) = \text{reduce}(Y^{(n)}), c, d, t_n)\].
If \(P(X^{(n)}) \leq P(Y^{(n)})\), then
(i) \(P(X^{(n-1)}) \leq P(Y^{(n-1)})\); and
(ii) \(P(S^{(n-1)}) \leq P(R^{(n-1)})\).

Proof: Let \(P(X^{(n)}) = (x_g_1, x_g_2, \ldots, x_g_n)\) and 
\(P(Y^{(n)}) = (y_g_1, y_g_2, \ldots, y_g_n)\). The receiving processors will be updated to 
\(x_g + t_{\text{comp}} + t_{\text{comm}}\) and \(t + t_{\text{comp}} + t_{\text{comm}}\), respectively. Let
\[X^* = P(X^{(n-1)} \setminus \{x_g + t_{\text{comp}} + t_{\text{comm}}\})\]
and 
\[Y^* = P(Y^{(n-1)} \setminus \{t + t_{\text{comp}} + t_{\text{comm}}\})\].

We will show that \(X^* \leq Y^*\). Let \(x_g_i\) be the smallest state greater than \(x_g\) and \(y_i\), be the smallest state greater than \(t\). Then
\[X^* = (x_g_2, \ldots, x_g_k, \ldots, x_g_n)\]
and 
\[Y^* = (t, \ldots, t, y_h_1, \ldots, y_h_n)\].
\(\forall i \text{ s.t. } i < \min(k, l),\)
\[x_g_i \leq y_h_2 \leq t.\]
\(\forall i \text{ s.t. } i \geq \max(k, l),\)
\[x_g_i \geq y_h_i.\]
If \(k < l\), then \(\forall i \text{ s.t. } k \leq i < l,\)
\[x_g_i \leq y_h_i \leq t.\]
If \(k > l\), then \(\forall i \text{ s.t. } l < i < k,\)
\[x_g_i = x_g_2 \leq y_h_2 \leq t \leq y_h_i.\]
We have shown \(X^* \leq Y^*\) and \(x_g_2 + t_{\text{comp}} + t_{\text{comm}} \leq t + t_{\text{comp}} + t_{\text{comm}}\). Therefore, by Lemma 2, \(P(X^{(n-1)}) \leq P(Y^{(n-1)})\).

Now we will prove the second claim by induction. The state of the sending processors are updated to \(x_g + t_{\text{comm}}\) and \(t + t_{\text{comm}}\) and clearly, \(x_g + t_{\text{comm}} \leq t + t_{\text{comm}}\). For \(n = p\), we have \(S^{(p-1)} \leq R^{(p-1)}\). Assume that \(P(S^{(n)}) \leq P(R^{(n)})\), by Lemma 2, \(P(S^{(p)}) \leq P(R^{(p)})\).

We can now define an iteration on segment \(k\) with initial state \(X_k^{(p)}\) as repeatedly applying \(p - 1\) send/receive pairs. Let \(X_k^{(p)} = \text{optiter}(X_k^{(p)})\) be the iteration that applies optreduce at every step. The output is the collection of states of the sending processor from each step and the state of the root processor after finishing the final computation. Let \(X_k^{(p)} = \text{iter}(X_k^{(p)})\) be an iteration that applies any set of reduce operations.

Lemma 4. If \(P(X_k^{(p)}) \leq P(Y_k^{(p)})\) and \(X_k^{(p)} = \text{optiter}(X_k^{(p)})\) and \(Y_k^{(p)} = \text{iter}(Y_k^{(p)})\), then \(P(X_k^{(p)}) \leq P(Y_k^{(p)})\).

Proof: By Lemma 3, the state vectors of the remaining processors at every step will satisfy the conditions of Lemma 2. Hence, we can conclude that \(S_k^{(1)} \leq R_k^{(1)}\) and \(X_k^{(1)} \leq Y_k^{(1)}\), where \(X_k^{(1)} = \text{optreduce}(X_k^{(2)})\) and \(Y_k^{(1)} = \text{reduce}(Y_k^{(2)})\). The initial states for segment \(k + 1\) are \(X_k^{(p)} = S_k^{(1)} \cup X_k^{(1)}\) and \(Y_k^{(p)} = R_k^{(1)} \cup Y_k^{(1)}\). By Lemma 2, \(P(X_k^{(p)}) \leq P(Y_k^{(p)})\).

A reduction algorithm is obtained by repeatedly applying an iteration for every segment using the ending state vector of one segment as the initial state vector of the following. We obtain the greedy algorithm by always applying optiter. Any reduction algorithm is obtained by repeatedly applying any iteration.

Theorem 5. In a unidirectional, fully connected, homogeneous system the time complexity of the uni-greedy algorithm minimal among all reduction algorithms that reduce segments in order.

Proof: \(X_k^{(p)}\) is the state of the processors at the start of segment \(k\) when applying the greedy algorithm and \(Y_k^{(p)}\) is the state of the processors at the start of segment \(k\) when applying another reduction algorithm. We assume that the initial state of the processors before any reduction is done is zero. That is, \(X_k^{(1)} = Y_k^{(1)} = \{0, \ldots, 0\}\). By Lemma 3, \(P(X_k^{(p)}) \leq P(Y_k^{(p)})\) and by induction \(X_k^{(p)} \leq Y_k^{(p)}\). Hence, \(x_0 = \max(X_k^{(p)}) \leq \max(Y_k^{(p)}) = y_0\).

B. Theoretical Results

In Section VII-A we showed that given any segmentation of a message the uni-greedy algorithm is optimal among algorithms that reduce the segments in order on a processor. However, selecting the optimal segmentation over all possible segmentations is difficult. To avoid this difficulty, we restrict ourselves to considering only equi-segmentation in this section. In Section VIII we will investigate unequal segmentation.

We compare the optimal segmented uni-greedy algorithm with optimally segmented binomial, binary, and pipeline algorithms. For pipeline and binary, the optimal segment size are obtained by the formulae in Table IV. For binomial, the messages where always reduced with a single segment. The parameters used for the theoretical experiments were: \(\alpha = 0, 1, 10, 100, 1000, \beta = 1, \gamma = 0, 1, p = 2^k, \text{s.t. } k = 2, 3, \ldots, 10, \text{ and } m = 2^k \text{ bytes, s.t. } k = 2, 3, \ldots, 16\).
Figures 4a and 5 show the results for when \( \alpha = 10, \beta = 1, \) and \( \gamma = 0 \). In Figure 4a the algorithms are compared by plotting the ratio between the algorithms time and the time for the uni-greedy algorithm. For small messages the optimal segment size for uni-greedy is 1, and uni-greedy is exactly the binomial algorithm. So for small messages, \( (m < 64 \text{ bytes in this example}) \) uni-greedy and binomial have the same time. For large messages \( (m > 1.64 \times 10^4 \text{ bytes in this example}), \) the poor start up of the pipeline algorithm is negligible and the pipeline algorithm approaches the uni-greedy algorithm asymptotically. The middle region is where uni-greedy is the most relevant, seeing an increase in performance over the standard algorithms up to approximately 50% faster. Figure 4b plots the time for each algorithm.

Figure 5 gives the results for each value of \( p \) and \( m \), the uni-greedy algorithm is compared with the best of the standard algorithms by plotting

\[
\text{Ratio} = \frac{\text{Minimum time of the standard algorithms}}{\text{Time for the Greedy Algorithm}}.
\]

A ratio of 1 (white in the figure) indicates that a standard algorithm has the same time as the uni-greedy algorithm. Larger numbers indicate a more significant improvement. Again the region that uni-greedy provides the greatest improvement is for “medium” size messages. As the number of processors increase, the range of message sizes where uni-greedy provides the most improvement (ratio \( \approx 1.5 \)) increases. This is largely due to the difference in the number of time steps required to reduce the first segment of uni-greedy \( (\lceil \log_2 p \rceil) \) versus that of pipeline \( (p - 1) \). For larger \( p \), pipeline requires a larger message before it is able to “make up” for the extra time to reduce the first segment.

VI. BIDIRECTIONAL SYSTEM

In this section, we can adapt the uni-greedy algorithm presented in Section V under the unidirectional context to an algorithm more suited for a bidirectional system. We wish to compare the new algorithm to the current
state-of-the-art. We again have the standard algorithms: binomial, binary, and pipeline. We also consider a butterfly algorithm \([4, 5]\) which minimizes the computation term of the reduction. All of the algorithms mentioned so far work for non-commutative operations. If the operation is commutative, then any broadcast algorithm can be used (in reverse) for a reduction. Bar-Noy, et al. \([2]\) and Träff and Ripke \([3]\) provide broadcast algorithms that match the lower bound on the number of communication rounds for broadcasting \(q\) segments among \(p\) processors. The time complexity for such a reduction algorithm is\
\[
\left(\lceil \log_2 p \rceil + q - 1\right)(\alpha + \beta m + \gamma m).
\]
From theoretical simulations it is seen that our new algorithm (bi-greedy) has the same time complexity of a reverse optimal broadcast. The rest of this section is organized as follows: In Section VI-A we analyze the time complexity of the reduction algorithms. In Section VI-B we introduce the new algorithm (bi-greedy).

### A. Theoretical Results

We begin by summarizing the time complexity for each algorithm in Table III. Binomial and butterfly do not have a choice on the segment size, so the optimal segment/time rows are not included for those algorithms. Binomial, binary, pipeline, and bi-greedy, all work in the same way: the message is split into \(q\) segments of equi-segment size and segments are communicated in rounds, where the time for one round is \(\alpha + \beta s + \gamma s\). For these algorithms it is clear that bi-greedy is the best since it require the minimum number of rounds for any number of segments. The butterfly algorithm does not start with a fixed segmentation of the message, but rather recursively halves the message size and exchanges the message between processors. This method allows the computation to be distributed evenly among the processors and hence minimizing the computational term. If the computational rate is small (i.e. \(\gamma\) is large) than minimizing the computation term can be advantageous.

Figure 6a and 6b show the results for when \(p = 64\), \(\alpha = 50000\), \(\beta = 6\) and \(\gamma = 1\). In Figure 6a the ratio between the time of the algorithm and the time of bi-greedy is shown. When comparing bi-greedy with binomial, binary, and pipeline, we see similar results as in the unidirectional case. For small messages \((m < 10\ \text{KB in this example})\), bi-greedy and binomial have the same time. For large messages \((m > 10^5\ \text{KB in this example})\), pipeline approaches the bi-greedy algorithm asymptotically. For medium length messages \((200 < m < 4000\ \text{KB in this example})\), bi-greedy provides approximately a 50% increase in performance. In this example, the butterfly algorithm performs well for medium size messages \((m \approx 300\ \text{KB in this example})\) but does not provide good asymptotic behavior for large and small message sizes. This is not a surprise as Rabenseifner provides tuning experiments to switch to the binomial algorithm for small messages, and the pipeline algorithm for large messages \([4]\).

For some machine parameters the butterfly algorithm can perform better than bi-greedy for some message sizes. If the ratio, \(\beta/\gamma\), is small enough then for some message sizes the butterfly algorithm will be better than bi-greedy. To be more precise as to what constitutes a small ratio we consider Figure 7. Above the blue line bi-greedy will always be better than the butterfly algorithm. Below the blue line, there exists a message size, \(m\), such that the butterfly algorithm is better than bi-greedy (in Figure 6 butterfly would dip below bi-greedy). For up

| Algorithm            | Time       | Formula |
|----------------------|------------|---------|
| Binomial             | \(|\log_2 p|\alpha + \beta m + \gamma m|\)     |
| Pipeline             | \(|(p + q - 2)(\alpha + \beta s + \gamma s)|\) |
| Bi-greedy &          | \(|\log_2 p| + q - 1)(\alpha + \beta m + \gamma m)|\) |
| Optimal Broadcast    | \(|\log_2 p|\alpha + \beta m + \gamma m|\)     |

TABLE III: Time analysis for bidirectional reduction algorithms, where \(N = \left\lceil \log_2 (p+1) \right\rceil\), \(s_{opt}\) is the optimal equi-segment size, and \(T_{opt}\) is the time for the algorithm at \(s_{opt}\). Formulae are valid for \(p > 3\). The reduce-scatter/gather formula is only a lower bound when \(p\) is not a power of two.
to 1000 processors the ratio would have to be less than 5 for the butterfly algorithm to be better. In practice, a ratio larger than 10 is more likely, even for single core processors.

The bidirectional greedy (bi-greedy) algorithm adheres to the following restrictions: the message is split using equi-segmentation and the ports are filled giving priority to segments with smaller indices. This restriction differs from that of uni-greedy, in that a processor may receive a segment before sending a segment with a smaller index. For the uni-greedy algorithm, the smaller indexed segment must be sent first. A downfall of this relaxation is that extra storage must be allocated. However, other reduction algorithms also require a larger buffer (see Rabenseifner [4]).

Algorithm 2 on the next page gives the pseudo-code for the bi-greedy algorithm. $S$ and $R$ are $p \times 1$ arrays where entry $i$ gives the time when processor $i - 1$ has finished sending or receiving, respectively, the last scheduled communication. $C$ is a $p \times 2$ array where $C(i, 1)$ is the start time and $C(i, 2)$ is the end time of the last scheduled computation on processor $i - 1$. $M$ is a $p \times q$ array where entry $(i, j)$ gives the time when processor $i - 1$ has finished sending segment $j$ (except for the root ($i = 1$) which gives the time after finishing the final computation). In each inner loop the maximum number of send/receive pairs for segment $j$ is determined. The final entries of sendProc and recvProc are scheduled to send and receive segment $j$, respectively. These processors are paired entry-wise to obtain send/receive pairs. The algorithm assumes a discrete time step.

Figure 8 gives an example for 16 processors and 5
Algorithm 2: Bi-greedy Algorithm

\( S = \text{zeros}(p, 1); \)
\( R = \text{zeros}(p, 1); \)
\( C = \text{zeros}(p, 2); \)
\( t = 0; \)
\( M = \text{zeros}(p, q); \)
while \( \min\{M(i, j) \mid 1 \leq i \leq p \text{ and } 1 \leq j \leq q\} = 0 \) do
    \( \text{segStart} = \min\{j \mid M(1, j) = 0\}; \)
    \( \text{stop} = 1; \)
    \( j = \text{segStart} - 1; \)
    while \( \text{stop} \) do
        \( j = j + 1; \)
        \( \text{COMP} = \{i \mid C(i, 1) \leq t < C(i, 2) \text{ or } t + t_{\text{comm}}(j) > C(i)\}; \)
        \( I = \{i \mid M(i, j) = 0\}; \)
        \( \text{sendProc} = I \setminus \text{COMP} \setminus \{i \mid S(i) > t\}; \)
        \( \text{recvProc} = I \setminus \text{COMP} \setminus \{i \mid R(i) > t\}; \)
        \( \text{freeProc} = \text{sendProc} \setminus \text{recvProc}; \)
        \( \text{sendProc} \leftarrow \text{sendProc} \setminus \text{freeProc}; \)
        \( \text{recvProc} \leftarrow \text{recvProc} \setminus \text{freeProc}; \)
        \( s = |\text{sendProc}|; \)
        \( r = |\text{recvProc}|; \)
        \( f = |\text{freeProc}|; \)
        if \( s = r \) then
            \( y = \lfloor f/2 \rfloor; \)
            \( \text{sendProc} \leftarrow \text{sendProc} \cup \{\text{freeProc}(i) \mid f - y + 1 \leq i \leq f\}; \)
            \( \text{recvProc} \leftarrow \text{recvProc} \cup \{\text{freeProc}(i) \mid 1 \leq i \leq y\}; \)
        else if \( s < r \) then
            \( y = r - s; \)
            \( m = \min(f, y); \)
            \( x = \lfloor (f - m)/2 \rfloor; \)
            if \( m > 0 \) then
                \( \text{sendProc} \leftarrow \text{sendProc} \cup \{\text{freeProc}(i) \mid f - (m + x) + 1 \leq i \leq f\}; \)
            if \( x > 0 \) then
                \( \text{recvProc} \leftarrow \text{recvProc} \cup \{\text{freeProc}(i) \mid 1 \leq i \leq x\}; \)
        else if \( r < s \) then
            \( y = s - r; \)
            \( m = \min(f, y); \)
            \( x = \lfloor (f - m)/2 \rfloor; \)
            if \( m > 0 \) then
                \( \text{recvProc} \leftarrow \text{recvProc} \cup \{\text{freeProc}(i) \mid 1 \leq i \leq m + x\}; \)
            if \( x > 0 \) then
                \( \text{sendProc} \leftarrow \text{sendProc} \cup \{\text{freeProc}(i) \mid f - x + 1 \leq i \leq f\}; \)
        \( l = \min(|\text{sendProc}|, |\text{recvProc}|) \)
        if \( l = 0 \) then
            \( \text{sendProc} \leftarrow 0; \)
            \( \text{recvProc} \leftarrow 0; \)
        else
            \( \text{sendProc} = \{\text{sendProc}(i) \mid 1 \leq i \leq l\}; \)
            \( \text{recvProc} = \{\text{recvProc}(i) \mid 1 \leq i \leq l\}; \)
        \( M(i, j) = t + t_{\text{comm}}(j), \forall i \text{ s.t. } i \in \text{sendProc}; \)
        \( S(i) = t + t_{\text{comm}}(j), \forall i \text{ s.t. } i \in \text{sendProc}; \)
        \( R(i) = t + t_{\text{comm}}(j) + t_{\text{comm}}(j), \forall i \text{ s.t. } i \in \text{recvProc}; \)
        \( \text{C}(i, 1) = t + t_{\text{comm}}(j), \forall i \text{ s.t. } i \in \text{recvProc}; \)
        \( C(i, 2) = t + t_{\text{comm}}(j) + t_{\text{comm}}(j), \forall i \text{ s.t. } i \in \text{recvProc}; \)
        if \( \{\{i \mid M(i, j) = 0\} = 1 \) then
            \( M(i, j) = \max\{M(i, j) \mid 1 \leq i \leq p\} + t_{\text{comm}}(j); \)
        if \( \{\{i \mid S(i) \leq t\}\} + \{\{i \mid R(i) \leq t\}\} < 2 \) then
            \( \text{stop} = 0; \)
        else if \( j \geq q \) then
            \( \text{stop} = 0; \)
        \( t = t + 1; \)
equi-segments with $t_{comm} = 2$ and $t_{comp} = 1$. For this example we see that the time complexity matches the that of the optimal broadcast algorithm. From experiments we conjecture that for all $p$ and $q$ the bi-greedy algorithm has the same time complexity as a reverse optimal broadcast algorithm. We do not provide a proof at this time.

VII. Numerical Results

For the numerical experiments we implemented the uni-greedy algorithm developed under the unidirectional contexted (Figure 2) and the bi-greedy algorithm developed under the bidirectional context (Algorithm 2). The algorithms were implemented using OpenMPI version 1.4.3. All experiments were performed on the Janus supercomputer which consists of 1,368 nodes with each node having 12 cores. Since processors on a single node have access to shared memory, only one processor per node was used with a total of 64 nodes in use for the experiments. The nodes are connected using a fully non-blocking quad-data-rate (QDR) InfiniBand interconnect. The time for a reduction was calculated by taking the minimum time of 10 experiments.

We wish to compare the algorithms assuming that each is optimally tuned for the best segmentation. For a message of size $m$, segments of size $2^i$, s.t. $i = 0, \ldots, \lfloor \log_2 m \rfloor$, where tested and the best time was taken to be the time of an optimally tuned algorithm. Figures 9a and 9b compares the optimally tuned greedy algorithms with other algorithms for reduction.

Comparing bi-greedy with binomial and pipeline shows similar results as to those of the theoretical results. For small messages, the time for binomial and greedy are close, although binomial performs about twice as well as greedy in a few cases. For large messages, pipeline begins to out perform all algorithms and is a factor of two better than bi-greedy. However, for medium size messages ($100 < m < 3000$ KB) bi-greedy is better than both binomial and pipeline. The binary algorithm was not analyzed experimentally.

Comparing bi-greedy with the butterfly algorithm again shows similar results to the theoretical case. The bi-greedy algorithm outperforms butterfly for almost all message sizes. The butterfly algorithm performs very well for medium size messages (about the same time as bi-greedy for $m = 1000$ KB), but has poor performance for large and small messages. Comparing the bi-greedy algorithm with the built-in MPI function MPI_Reduce, shows that greedy, in most cases, is the best. Finally, the uni-greedy algorithm (although designed for a uni-directional system) performed fairly well in general, matching the performance of bi-greedy for small and large messages.

For small messages, greedy is equivalent to binomial, but there is a fairly large difference in the performance in this region. This is because the two algorithms implemented use different send/receive pairings and the performance differences indicate the system is actually heterogeneous.

Checking all of the possible segment sizes for a
reduce
(b) Message size versus time

| Time | Ratio |
|------|-------|
| 0    | 1     |
| 1    | 1.5   |
| 2    | 2     |
| 3    | 2.5   |
| 4    | 3     |
| 5    | 3.5   |

We prove that the unidirectional algorithm is optimal under certain assumptions. Our assumptions are

VIII. Unequal Segmentation

So far we have only optimized the greedy algorithm by splitting a message into equally sized segments (except possibly the last segment). An immediate question is then asked, what if the segments have unequal size? A question we have not found to be considered in the literature. Can the existing algorithms be improved?

To investigate these questions the message size was fixed to $m = 10$ and all possible segmentations were checked for $\alpha = 0, 1, \ldots, 10, 20, 30, \ldots, 100, 200, 300, \ldots, 1000$, $\beta = 1$, $\gamma = 0, 1$, and $p = 2^n$ s.t. $n = 2, \ldots, 10$ and $p = 3(2^n)$ s.t. $n = 1, \ldots, 9$. There a total of 512 possible segmentations. Examples of possible segmentations are $(9, 1), (3, 3, 3, 1), (3, 2, 4, 1), (2, 2, 1, 5), (6, 3, 1)$, etc. An equi-segmentation is said to be one that has the same segment size for all segments except possibly the last segment may be smaller.

For the greedy algorithm, 61 out of 986 experiments where optimized by an unequal segmentation. To compare unequal to equi-segmentation, we use the value

$$\text{Ratio} = \frac{\text{Best time for equi-segmentations}}{\text{Best time for all segmentations}}.$$

A ratio of 1 indicates that one of the optimal segmentations is an equi-segmentation. If there were multiple segmentations that were optimal and one of them was an equal segmentation, then the experiment was said to be optimized by an equi-segmentation. The maximum improvement over equi-segmentation was 7.3%. Of the 61 that did see an increase in performance the average improvement was 2.0%.

Table IV shows results for all experiments that were optimized by unequal segmentation. We note that, in all cases, one of the optimal segmentations was nearly the same as the best equi-segmentation. Actually, to obtain the optimal (unequal) segmentation from the best of the equi-segmentations, the size of only one or two segments had to be increased or decreased by a value of 1 in most cases.

Figures 10a and 10b show the results for all $\alpha$, $\gamma$, and $p$. For $\alpha > 20$, the greedy algorithm was always optimized by an equi-segmentation and therefore is not graphed. A pattern as to when unequal segmentation is optimal is not evident from the figures. For the pipeline algorithm, all experiments where optimized by equi-segmentation. The binary algorithm was not checked for unequal segmentation.

The experiment is limited to small values of $m$ since the number of possible segmentations grows exponentially as $m$ increases. The algorithms were not implemented for unequal segmentations since the small theoretical improvements most likely will not be obtained. Further investigation is required to determine if for larger mush sizes does the improvement of unequal segmentation become greater.

IX. Conclusion

Two new algorithms for reduction are presented. The two algorithms are developed under two different communication models (unidirectional and bidirectional).

We prove that the unidirectional algorithm is optimal under certain assumptions. Our assumptions are
fully connected, homogeneous system, and processing the segments in order. Previous algorithms that satisfy the same assumptions are only appropriate for some configurations. The uni-greedy algorithm is optimal for all configurations. The most improvement over standard algorithms is for messages of “medium” length, providing about 50% improvement when compared to the best of the standard algorithms in that region. The region of “medium” length messages become more prevalent as the number of processors increases.

We adapted the greedy algorithm that was developed in the unidirectional context to an algorithm that was more suited for a bidirectional system. With simulations we found that the bi-greedy algorithm matched the time complexity of optimal broadcast algorithms (scheduled in reverse order as a reduction). Similar performance improvements where found as in the unidirectional case.

Implementation of the algorithms confirm the theoretical results. Our implementation of the bi-greedy algorithm is was best among all algorithms implemented for “medium” size messages. For small and large messages, the more simplistic algorithms (binomial and pipeline), which are asymptotically optimal, outperformed the greedy algorithms.

Finally, the concept of unequal segmentation was discussed and analyzed for the greedy algorithm and the pipeline algorithm in a unidirectional system. For the greedy algorithm, unequal segmentation provided, in our sample test suite, the optimal segmentation 6.2% of the time with a maximum improvement of 7.3%. The pipeline algorithm was always optimized by equal segmentation.

ACKNOWLEDGMENT

This work utilized the Janus supercomputer, which is supported by the National Science Foundation (award number CNS-0821794) and the University of Colorado Boulder. The Janus supercomputer is a joint effort of the University of Colorado Boulder, the University of Colorado Denver and the National Center for Atmospheric Research.

REFERENCES

[1] R. Hockney, “The communication challenge for MPP: Intel Paragon and Meiko CS-2,” Parallel Computing, vol. 20, no. 3, pp. 389–398, 1994.
[2] A. Bar-Noy, S. Kipnis, and B. Schieber, “Optimal multiple message broadcasting in telephone-like communication systems,” Discrete Appl. Math., vol. 100, no. 1-2, pp. 1–15, Mar. 2000. [Online]. Available: http://dx.doi.org/10.1016/S0166-218X(99)00155-9
[3] J. L. Träff and A. Ripke, “Optimal broadcast for fully connected processor-node networks,” J. Parallel Distrib. Comput., vol. 68, no. 7, pp. 887–901, 2008.
[4] R. Rabenseifner, “Optimization of collective reduction operations,” in International Conference on Computational Science, 2004, pp. 1–9.
[5] R. Rabenseifner and J. L. Träff, “More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems,” in PVM/MPI, 2004, pp. 36–46.
[6] P. Sanders, J. Speck, and J. L. Träff, “Two-tree algorithms for full bandwidth broadcast, reduction and scan,” Parallel Computing, vol. 35, no. 12, pp. 581 – 594, 2009, selected papers from the 14th European PVM/MPI Users Group Meeting. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819109000957
[7] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert, “Pipelining broadcasts on heterogeneous platforms,” Parallel and Distributed Systems, IEEE Transactions on, vol. 16, no. 4, pp. 300 – 313, April 2005.
[8] O. Beaumont, L. Marchal, and Y. Robert, “Broadcast trees for heterogeneous platforms,” in 19th International Parallel and Distributed Processing Symposium IPDPS’2005. Society Press, 2005, pp. 3–8.

Fig. 10: Ratio for optimal segmentation versus the best equal segmentation.
[9] A. Legrand, L. Marchal, and Y. Robert, “Optimizing the steady-state throughput of scatter and reduce operations on heterogeneous platforms,” J. Parallel Distrib. Comput., vol. 65, no. 12, pp. 1497–1514, Dec. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2005.05.021

[10] E. Chan, R. A. van de Geijn, W. Gropp, and R. Thakur, “Collective communication on architectures that support simultaneous communication over multiple links,” in PPOPP, 2006, pp. 2–11.

[11] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra, “Automatically tuned collective communications,” in Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), ser. Supercomputing '00, Washington, DC, USA: IEEE Computer Society, 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=370049.370055

[12] J. Pjesivac-Grbović, T. Angskun, G. Bosilca, G. Fagg, E. Gabriel, and J. Dongarra, “Performance analysis of MPI collective operations,” in Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International, april 2005, p. 8 pp.

[13] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, “On optimizing collective communication,” in CLUSTER '04 Proceedings of the 2004 IEEE International Conference on Cluster Computing, 2004, pp. 145–156.

[14] M. Cosnard and Y. Robert, “Complexity of parallel QR factorization,” Journal of the A.C.M., vol. 33, no. 4, pp. 712–723, 1986.
| Parameters | Ratio | Equi-segmentation | Optimal Segmentation |
|------------|-------|-------------------|---------------------|
| $p = 6, \alpha = 1, \gamma = 1$ | 1.0048 | (4,4,2) | (3,5) |
| $p = 6, \alpha = 2, \gamma = 1$ | 1.0357 | (4,4,2) | (3,5) |
| $p = 8, \alpha = 0, \gamma = 1$ | 1.0020 | (4,4,2) | (3,5) |
| $p = 12, \alpha = 0, \gamma = 1$ | 1.0169 | (4,4,2) | (3,5) |
| $p = 12, \alpha = 1, \gamma = 1$ | 1.0160 | (4,4,2) | (3,5) |
| $p = 16, \alpha = 1, \gamma = 1$ | 1.0179 | (4,4,2) | (3,5) |
| $p = 24, \alpha = 1, \gamma = 1$ | 1.0190 | (4,4,2) | (3,5) |
| $p = 32, \alpha = 1, \gamma = 1$ | 1.0222 | (4,4,2) | (3,5) |
| $p = 32, \alpha = 2, \gamma = 1$ | 1.0191 | (4,4,2) | (3,5) |
| $p = 48, \alpha = 1, \gamma = 1$ | 1.0095 | (4,4,2) | (3,5) |
| $p = 48, \alpha = 2, \gamma = 1$ | 1.0084 | (4,4,2) | (3,5) |
| $p = 64, \alpha = 1, \gamma = 1$ | 1.0253 | (4,4,2) | (3,5) |
| $p = 64, \alpha = 2, \gamma = 1$ | 1.0136 | (4,4,2) | (3,5) |
| $p = 96, \alpha = 1, \gamma = 1$ | 1.0092 | (4,4,2) | (3,5) |
| $p = 96, \alpha = 2, \gamma = 1$ | 1.0092 | (4,4,2) | (3,5) |
| $p = 128, \alpha = 1, \gamma = 1$ | 1.0161 | (4,4,2) | (3,5) |
| $p = 192, \alpha = 1, \gamma = 0$ | 1.0069 | (4,4,2) | (3,5) |

TABLE IV: Optimal Segmentation compared to best of the equi-segmentations.