

Diagrams:

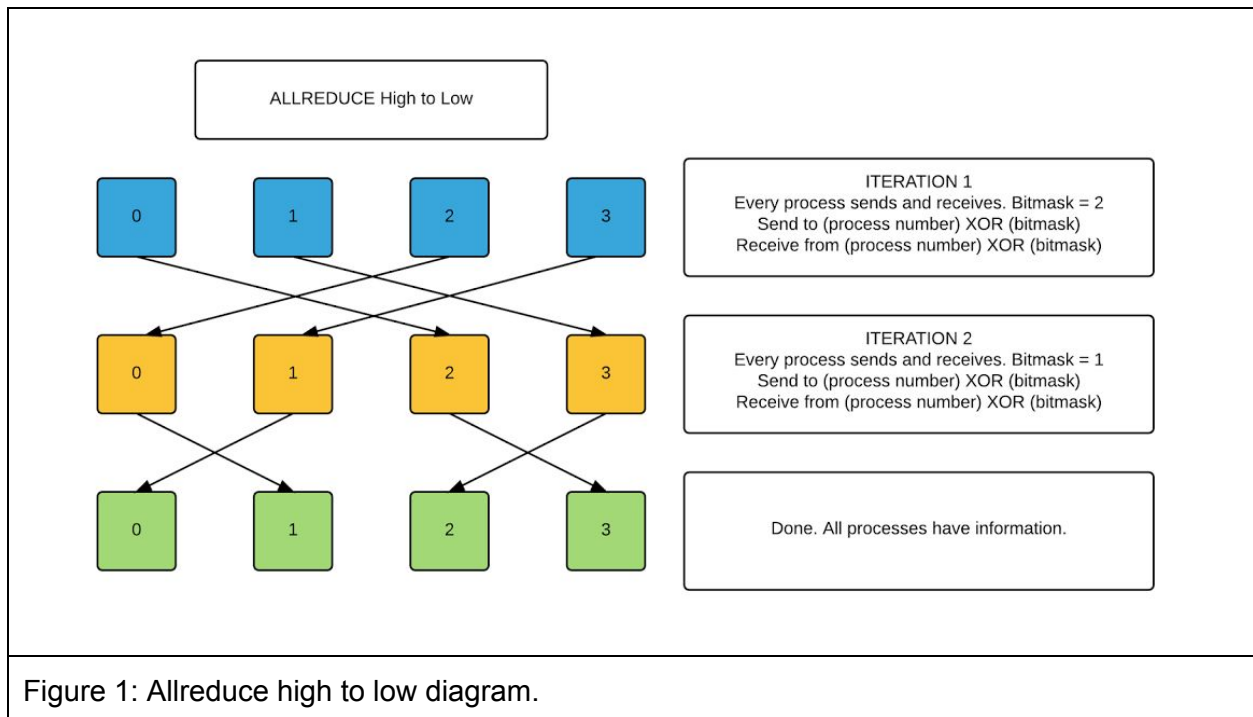


Figure 1: Allreduce high to low diagram.

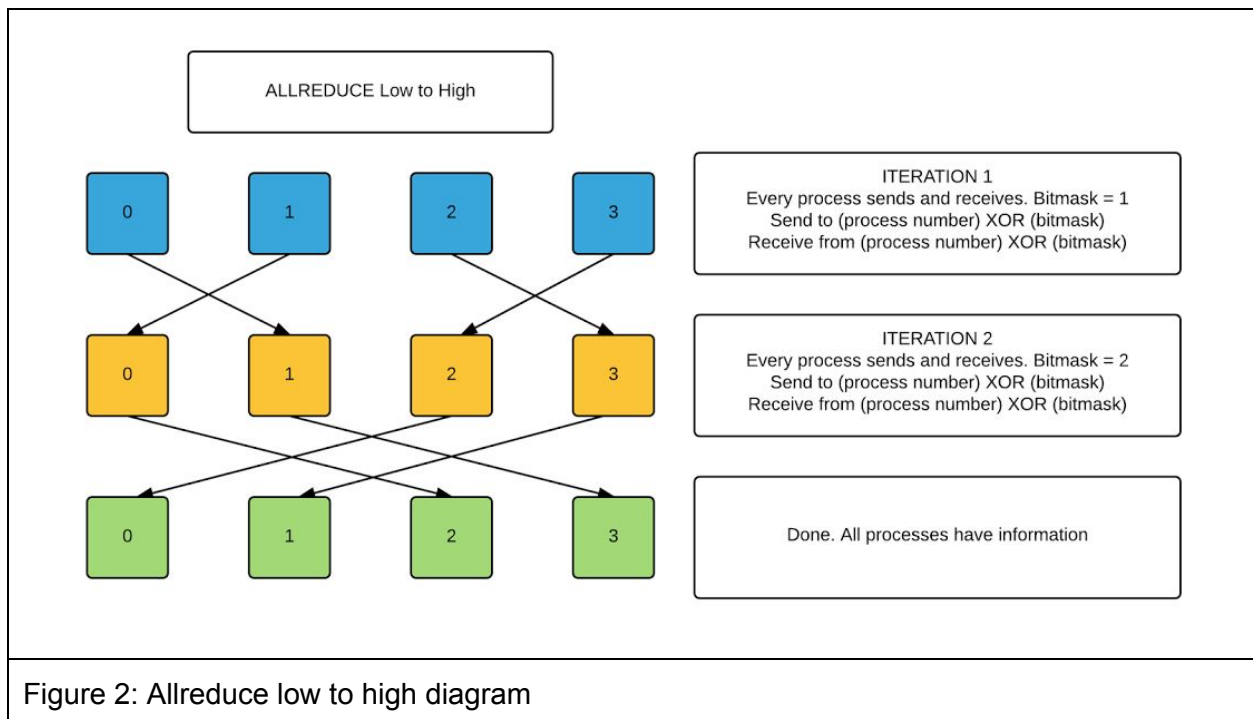
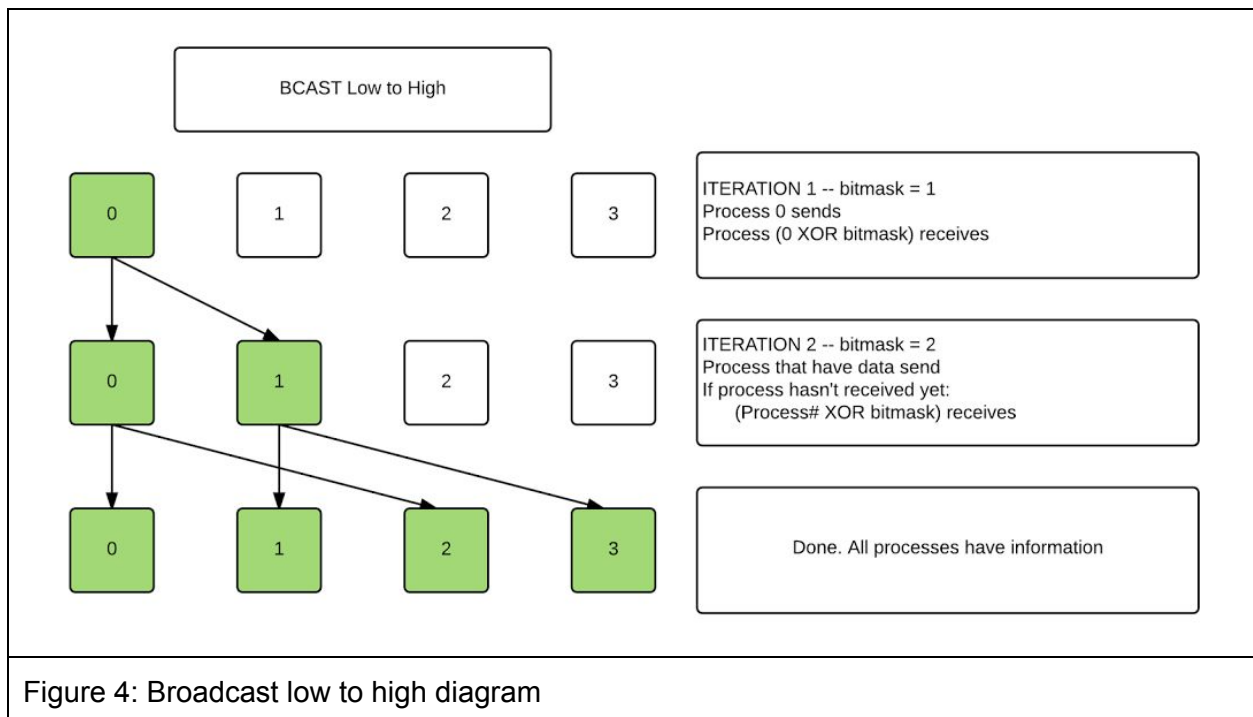
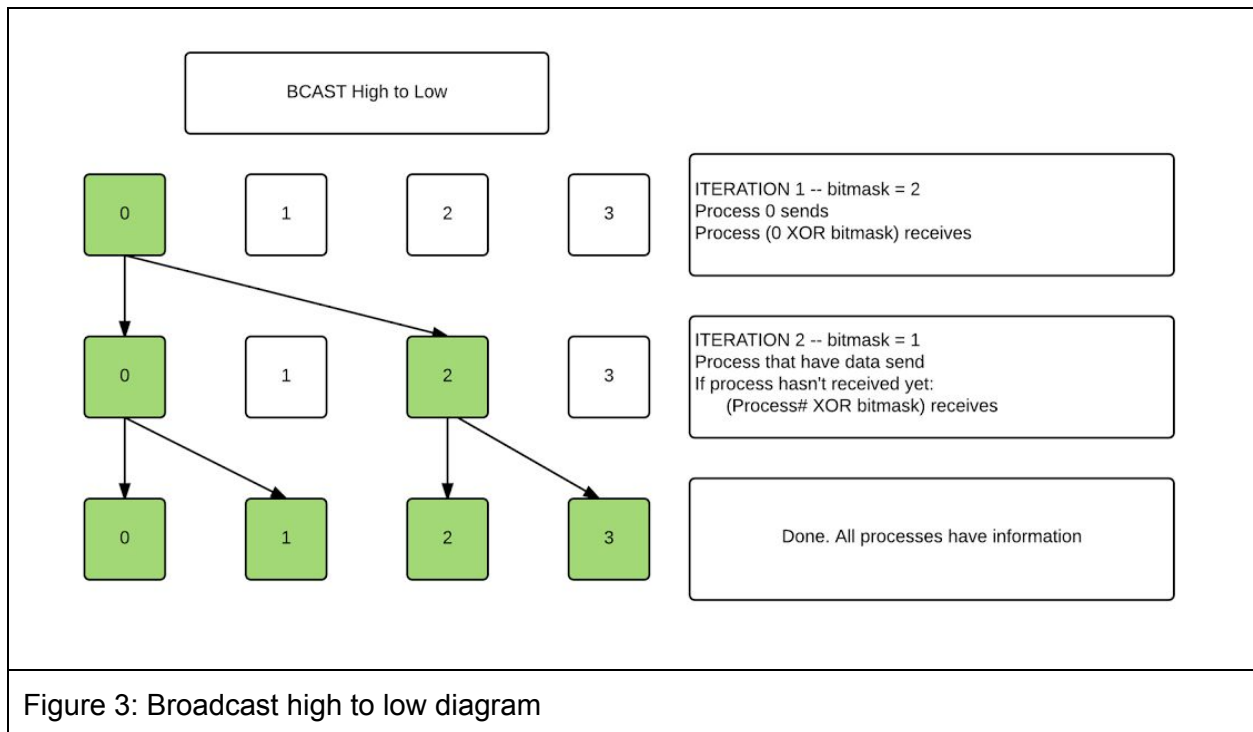
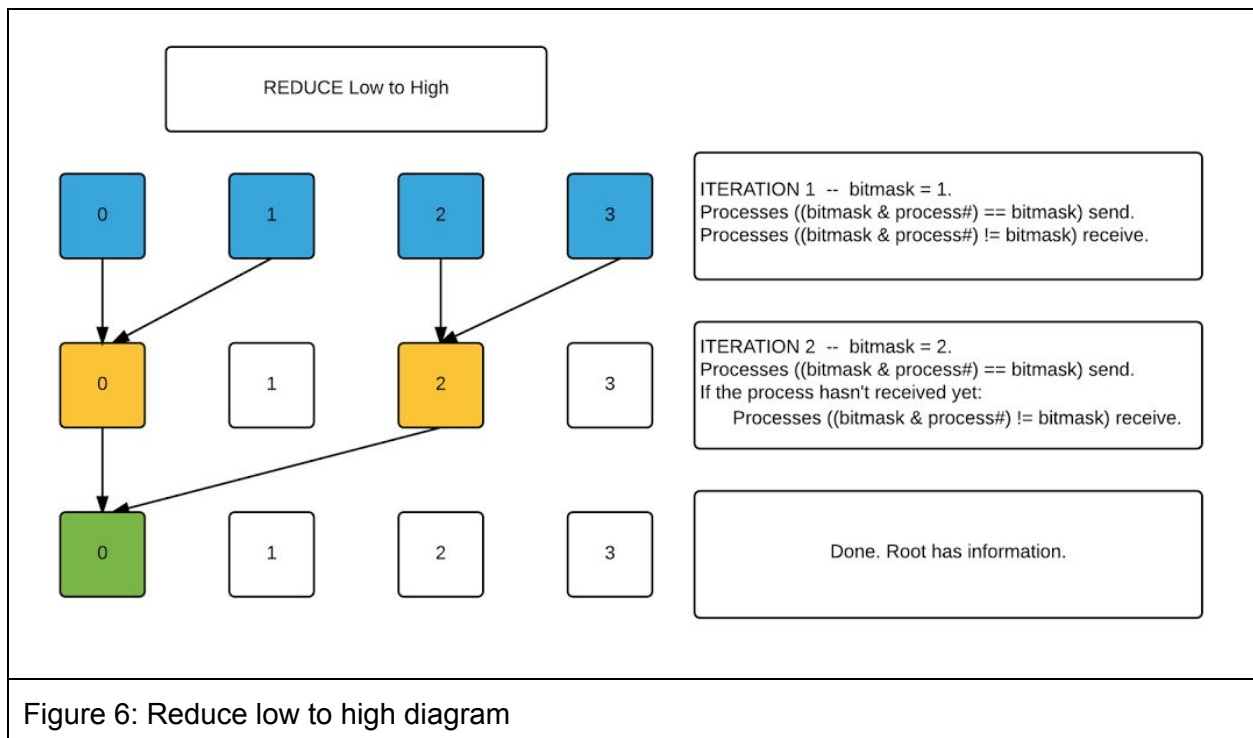
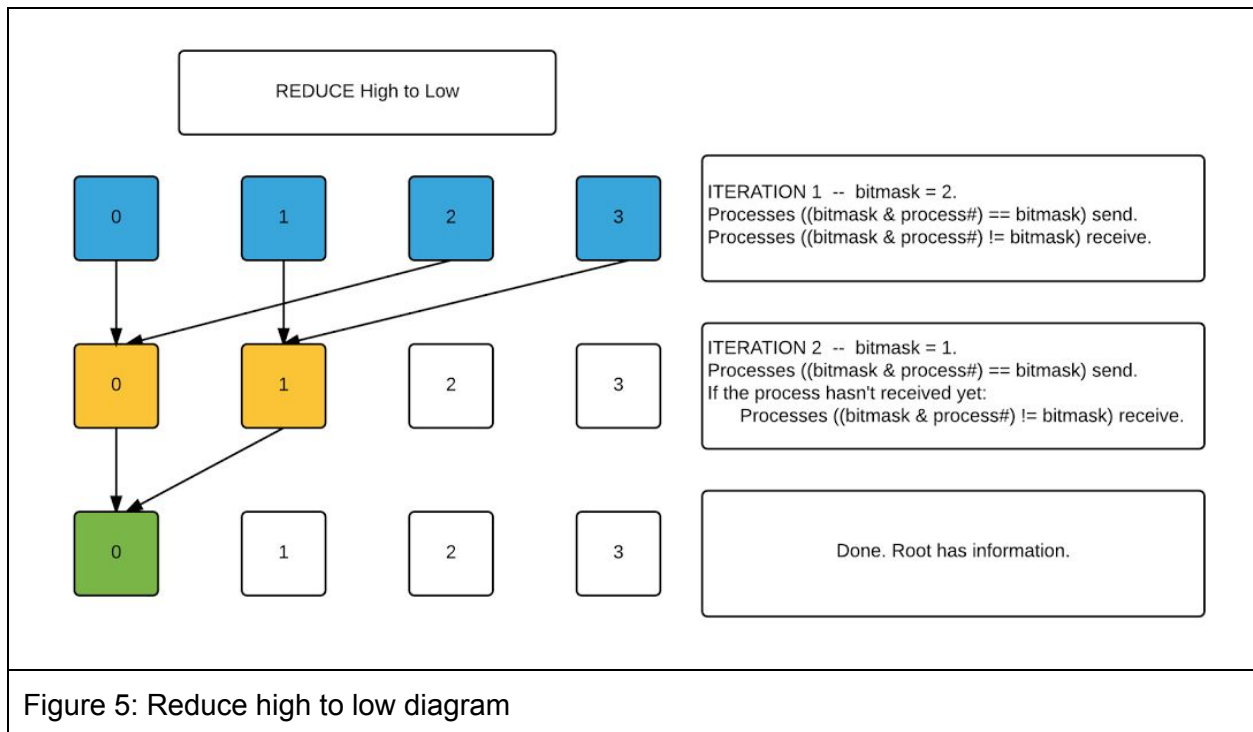


Figure 2: Allreduce low to high diagram





Graph Results

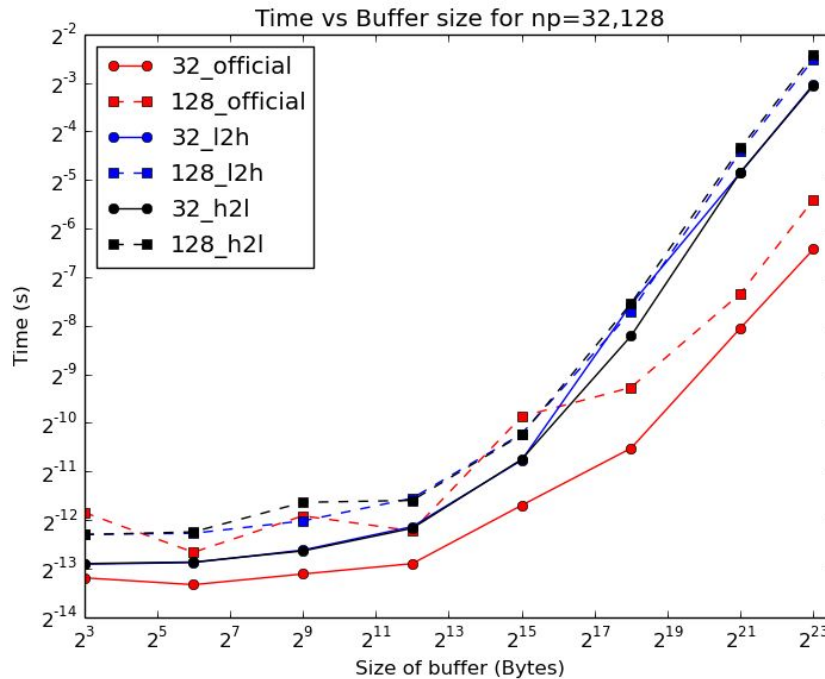


Figure 7: Time taken for official all reduce, my butterfly allreduce using high to low and low to high. Test was run with 32 and 128 nodes. X axis is $\log_2(\text{buffer size})$. The buffer size ranges from 8 Bytes to 8 Megabytes. Results were generated by running code on Janus. Timing was done using MPI_Wtime. 20 iterations total: 10 warmup followed by 10 timed and averaged. This procedure is true of following graphs as well.

For all cases except the 128 node official allreduce, the results barely change until 2^6 . Then they exponentially increase until around 2^{15} bytes. They linearly increase after that. The 128 node official allreduce goes up and down until 2^{13} , at which point it trends up. This is likely different based on what nodes the job was run on. Sometimes the job was on one rack whereas others it was on six or more racks.

In theory this area should be a flat line as any mpi message has can send roughly a kb of data without increasing bandwidth usage. It should effectively follow the line $y = a + bN$, where a is the latency, b is the bandwidth term and N is the size of the buffer. As seen in Figure 7, the results found roughly follow the $y = a + bN$ theory.

Performance Vs Official Allreduce

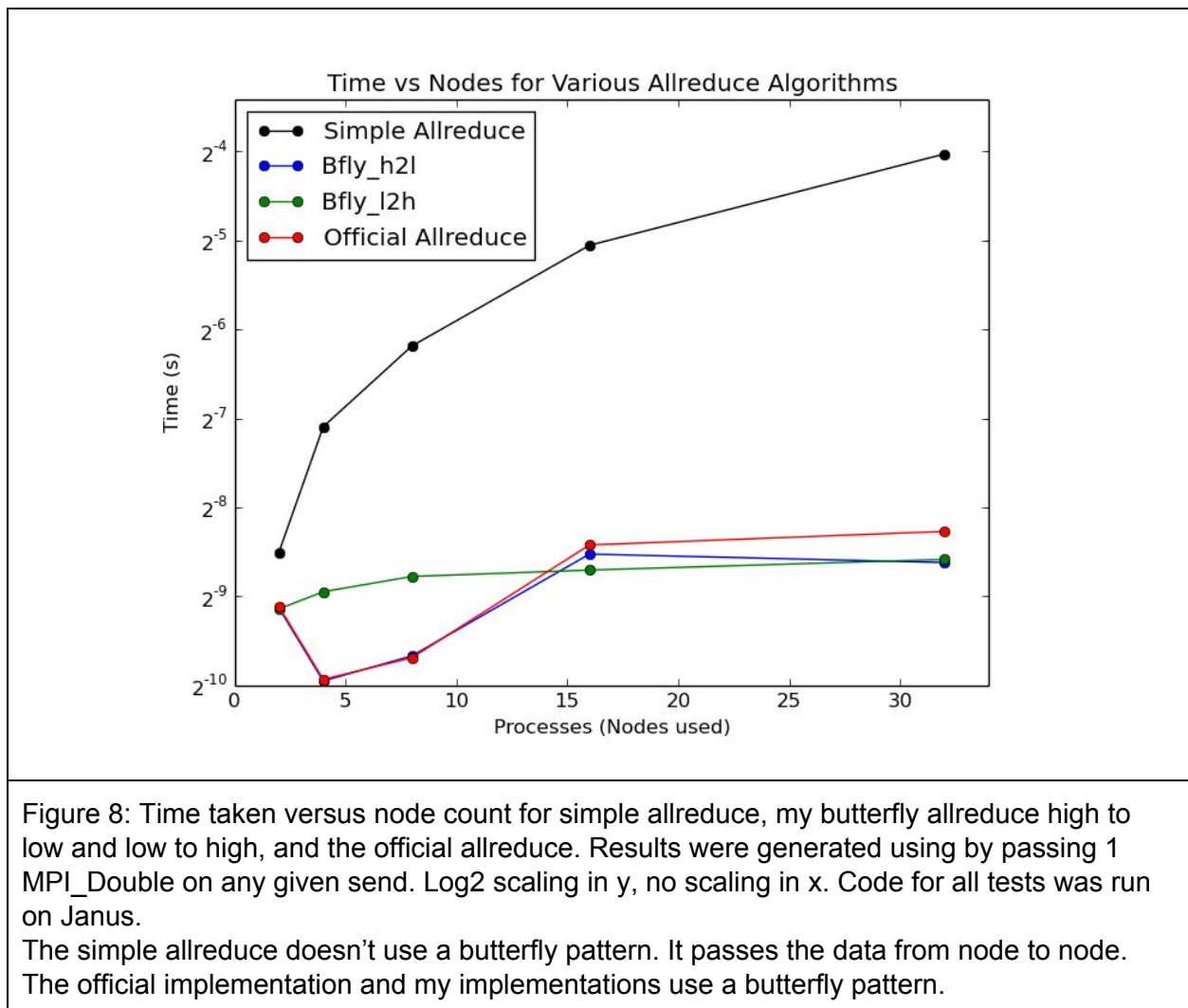
The official allreduce is far more efficient than the new implementations, especially at higher buffer sizes. In the latency dominated range, the new implementations are close, but still slower by a factor of 2. When bandwidth begins to dominate, this increases to a factor of 8 or more. It is likely that the official allreduce uses more efficient sends, and has more efficient single thread performance. The new implementations had simple loops to operate on the larger buffers which were not optimized.

Bit Traversal Order

Looking at figure 7, the bit traversal order seems to have little to no impact on the Janus system.

Match which traversal for broadcast with reduce

It would make for sense to compare a high to low broadcast with a low to high reduce. Looking at figures 3 and 6, it can be seen that the pattern is simply reversed. This is similar of the low to high broadcast and high to low reduce, which can be seen in figures 4 and 5. However, based on the butterfly results where bit traversal order didn't matter on Janus, this is likely a system specific problem.



Latency domain

This graph is supposed to show results from the latency domain. As can be seen from the graph, the official allreduce, and my implementations saw only a slight increase in time with an increase in nodes. The simple allreduce saw a linear increase in time with respect to the number of nodes used.

Physical topology

If one knew the physical topology, one could optimize the algorithm even further. By passing information between nodes that are physically closer, the latency could be decreases.

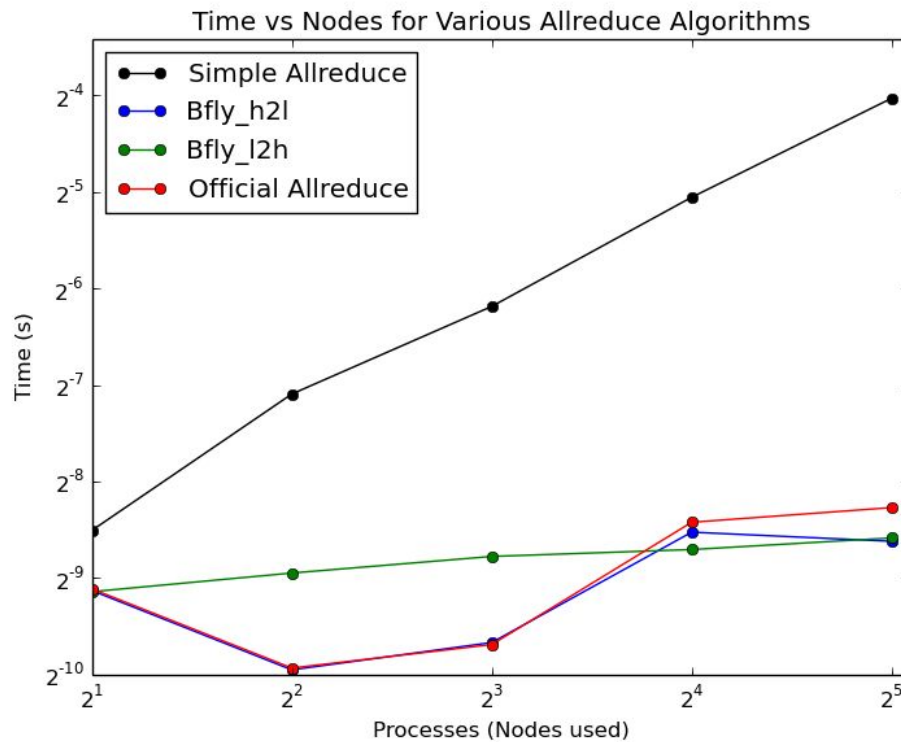


Figure 9: Time taken versus node count for simple allreduce, my butterfly allreduce high to low and low to high, and the official allreduce with log2 scaling. Results were generated using by passing 1 MPI_Double on any given send. Code for all tests was run on Janus.

Regression Results

With log2 scaling the official all_reduce as well as my implementations can be seen better. The official allreduce as well as my high to low follow some sort of curve whereas my low to high appears linear. Regression results found using Python's polyfit are shown below:

Regression Results -> $y = m + ax + bx^2$			
Algorithm	m	a	b
Official Allreduce	-1.07163188e-06	1.06260278e-04	1.01331373e-03
My low to high	-1.22296015e-06	6.60872865e-05	1.72894118e-03
My high to low	-1.79467109e-06	1.07510911e-04	1.01245098e-03
Simple Allreduce	1.52324478e-06	1.89160531e-03	-8.62647059e-04

Aside from the simple allreduce which appears to be very close to linear, the curves don't seem to follow a common pattern.

Program Explanations/Instructions:

For all programs compile with: `$mpicc hw2-PROGRAM.c -lm -o PROGRAM.out`

run with `$mpiexec -np NP ./PROGRAM.out -type SIZE`

where NP is process number

-type is -h or -l for high to low and low to high

SIZE is buffer size

Butterfly:

The first butterfly program relied on buffering and as such failed spectacularly on sizes close to 1 MB. It had all processes send and receive at the same time rather than matching sends and receives.

The second time the program used an alternating method to match sends and receives. The algorithm pseudocode is below:

```
deliver_to = world_rank XOR bitmask
```

```
If even iteration:
```

```
    if deliver_to > world_rank
```

```
        MPI_Send
```

```
if odd iteration:
```

```
    if deliver_to < world_rank
```

```
        MPI_Send
```

```
receive_from = world_rank XOR bitmask
```

```
if even iteration:
```

```
    if receive_from < world_rank
```

```
        MPI_Recv
```

```
if odd iteration:
```

```
    if receive_from > world_rank
```

```
        MPI_Recv
```

This matches all sends with receives for cases where the number of processors is a power of 2. Additional logic was added to account for process counts of 3 or 5, where additional sends are needed on some iterations.

Broadcast:

The broadcast algorithm was simpler than the butterfly as the edge cases were easier to deal with. The general pattern was to send from a node if it has information. Pseudocode is shown below.

```
if information_received:
```

```
    deliver_to = world_rank XOR bitmask
```

```
    MPI_Send
```

```
if not_received
```

```
    receive_from = world_rank XOR bitmask
```

```
    MPI_Recv
```

Logic was added such that information wasn't send to processes that didn't exist.

Reduce:

Reduce was broadcast in reverse. The general idea is that every process except process zero should deliver once. Again bit shifting patterns are used to determine which processes send. Pseudocode for high to low:

```
if not delivered
    if world_rank AND bitmask == bitmask
        deliver_to = world_rank XOR bitmask
        MPI_Send

if world_rank < bitmask
    receive_from = world_rank XOR bitmask
    if receive_from < world_size
        MPI_Recv
```

Fan in fan out reduce:

The fan in fan out reduce was the result of copying and pasting the main broadcast code after the the reduce code.

Simple Allreduce:

The simple allreduce had data start at process 0, send and combine in a line to process N. At this point process N then sent the final value back down the line to process 0.

General Testing:**Functionality:**

To test Butterfly, all nodes started out with a double array containing the 1.0. Every Thus the end result after adding all the data should be the number of nodes. Every node printed out what data they had at the end of the program. This was tested for various array sizes, low to high and high to low, and different node sizes (including odd node sizes).

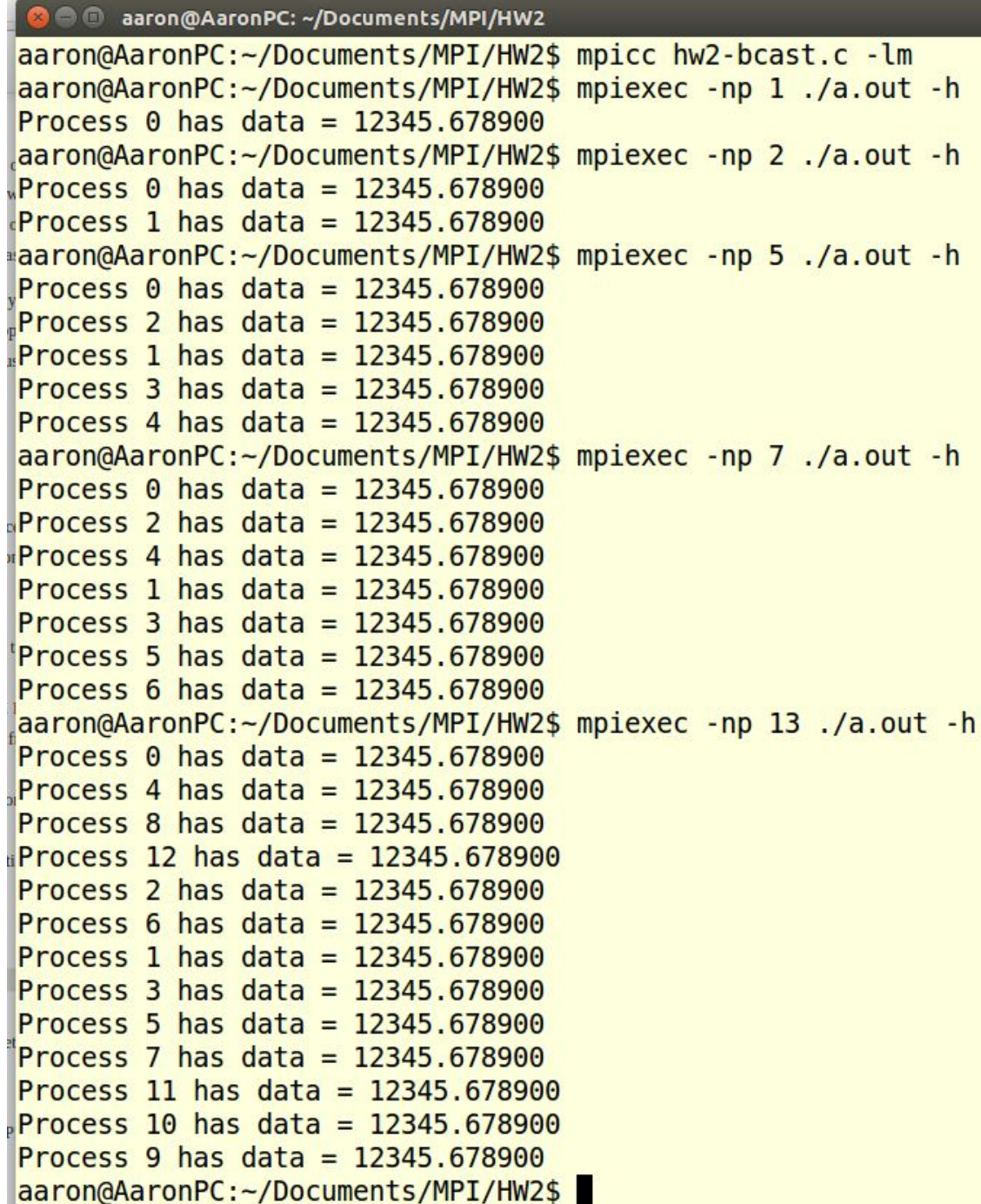
To test broadcast, node 0 started with data 12345.6789 in a double and all other nodes started with 0. At the end of the program each process printed out the data they ended up with. This was tested for different node sizes (including odd node sizes), low to high and high to low.

To test Reduce, all nodes started out with a double array containing the 1.0. Every Thus the end result after adding all the data should be the number of nodes. Process 0 printed out its value after the program ended. This was tested for different node sizes (including odd node sizes), low to high and high to low.

Timing:

MPI_Wtime was used to time the programs. 20 iterations were used. 10 warmup and 10 timed and averaged into 1 data point.

Appendix A: Program Outputs

A terminal window titled 'aaron@AaronPC: ~/Documents/MPI/HW2' displays the execution of an MPI program. The user runs 'mpicc hw2-bcast.c -lm' to compile the program. Then, they run 'mpiexec -np 1 ./a.out -h', which outputs 'Process 0 has data = 12345.678900'. Next, they run 'mpiexec -np 2 ./a.out -h', showing 'Process 0' and 'Process 1' both with data '12345.678900'. Then, 'mpiexec -np 5 ./a.out -h' shows processes 0 through 4, all with the same data. Finally, 'mpiexec -np 7 ./a.out -h' shows processes 0 through 6, all with the same data. The last command shown is 'mpiexec -np 13 ./a.out -h', with the first four processes (0, 4, 8, 12) listed and all showing the same data value. The terminal ends with the prompt 'aaron@AaronPC:~/Documents/MPI/HW2\$' and a cursor.

```
aaron@AaronPC: ~/Documents/MPI/HW2
aaron@AaronPC:~/Documents/MPI/HW2$ mpicc hw2-bcast.c -lm
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 1 ./a.out -h
Process 0 has data = 12345.678900
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 2 ./a.out -h
Process 0 has data = 12345.678900
Process 1 has data = 12345.678900
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 5 ./a.out -h
Process 0 has data = 12345.678900
Process 2 has data = 12345.678900
Process 1 has data = 12345.678900
Process 3 has data = 12345.678900
Process 4 has data = 12345.678900
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 7 ./a.out -h
Process 0 has data = 12345.678900
Process 2 has data = 12345.678900
Process 4 has data = 12345.678900
Process 1 has data = 12345.678900
Process 3 has data = 12345.678900
Process 5 has data = 12345.678900
Process 6 has data = 12345.678900
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 13 ./a.out -h
Process 0 has data = 12345.678900
Process 4 has data = 12345.678900
Process 8 has data = 12345.678900
Process 12 has data = 12345.678900
Process 2 has data = 12345.678900
Process 6 has data = 12345.678900
Process 1 has data = 12345.678900
Process 3 has data = 12345.678900
Process 5 has data = 12345.678900
Process 7 has data = 12345.678900
Process 11 has data = 12345.678900
Process 10 has data = 12345.678900
Process 9 has data = 12345.678900
aaron@AaronPC:~/Documents/MPI/HW2$
```

Figure 10: Broadcast high to low

```
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 1 ./a.out -1
Process 0 has data = 12345.678900
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 2 ./a.out -1
Process 0 has data = 12345.678900
Process 1 has data = 12345.678900
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 5 ./a.out -1
Process 0 has data = 12345.678900
Process 1 has data = 12345.678900
Process 2 has data = 12345.678900
Process 3 has data = 12345.678900
Process 4 has data = 12345.678900
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 32 ./a.out -1
Process 0 has data = 12345.678900
Process 16 has data = 12345.678900
Process 24 has data = 12345.678900
Process 8 has data = 12345.678900
Process 2 has data = 12345.678900
Process 26 has data = 12345.678900
Process 1 has data = 12345.678900
Process 3 has data = 12345.678900
Process 27 has data = 12345.678900
Process 4 has data = 12345.678900
Process 17 has data = 12345.678900
Process 10 has data = 12345.678900
Process 20 has data = 12345.678900
Process 28 has data = 12345.678900
Process 9 has data = 12345.678900
Process 25 has data = 12345.678900
Process 12 has data = 12345.678900
Process 14 has data = 12345.678900
Process 22 has data = 12345.678900
Process 30 has data = 12345.678900
Process 18 has data = 12345.678900
Process 6 has data = 12345.678900
Process 7 has data = 12345.678900
Process 11 has data = 12345.678900
Process 19 has data = 12345.678900
Process 21 has data = 12345.678900
Process 5 has data = 12345.678900
Process 23 has data = 12345.678900
Process 13 has data = 12345.678900
Process 15 has data = 12345.678900
Process 31 has data = 12345.678900
Process 29 has data = 12345.678900
```

Figure 11: broadcast low to high


```

aaron@AaronPC:~/Documents/MPI/HW2$ mpicc hw2-bfly2.c -lm
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 4 ./a.out -h 1
2 3 -1 -1 iteration 0
-1 -1 0 1 iteration 1
1 -1 3 -1 iteration 2
-1 0 -1 2 iteration 3
Process 0 has data 4.000000 on iteration 4
Process 1 has data 4.000000 on iteration 4
Process 2 has data 4.000000 on iteration 4
Process 3 has data 4.000000 on iteration 4
aaron@AaronPC:~/Documents/MPI/HW2$
aaron@AaronPC:~/Documents/MPI/HW2$
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 7 ./a.out -h 1
4 5 6 -1 -1 -1 -1 iteration 0
-1 -1 -1 -1 0 1 2 iteration 1
2 3 -1 -1 6 -1 -1 iteration 2
-1 -1 0 1 -1 -1 4 iteration 3
1 -1 3 -1 5 -1 -1 iteration 4
-1 0 -1 2 -1 4 -1 iteration 5
Process 0 has data 7.000000 on iteration 6
Process 1 has data 7.000000 on iteration 6
Process 4 has data 7.000000 on iteration 6
Process 6 has data 7.000000 on iteration 6
Process 2 has data 7.000000 on iteration 6
Process 3 has data 7.000000 on iteration 6
Process 5 has data 7.000000 on iteration 6
aaron@AaronPC:~/Documents/MPI/HW2$ █

```

Figure 12: Butterfly output with debugging information. The array shows which process is sending where, with -1 being no send. Not printed are the extra sends needed on some iterations.

[illegible]

Figure 13: Butterfly high to low with full array output. This shows changing buffer size

```

aaron@AaronPC:~/Documents/MPI/HW2$ mpicc hw2-bfly2.c -lm
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 1 ./a.out -h 1048576
Process 0 has data 1.000000 on iteration 0
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 2 ./a.out -h 1048576
Process 0 has data 2.000000 on iteration 2
Process 1 has data 2.000000 on iteration 2
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 5 ./a.out -h 8048576
Process 0 has data 5.000000 on iteration 6
Process 1 has data 5.000000 on iteration 6
Process 2 has data 5.000000 on iteration 6
Process 3 has data 5.000000 on iteration 6
Process 4 has data 5.000000 on iteration 6
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 13 ./a.out -h 1048576
Process 0 has data 13.000000 on iteration 8
Process 8 has data 13.000000 on iteration 8
Process 9 has data 13.000000 on iteration 8
Process 12 has data 13.000000 on iteration 8
Process 10 has data 13.000000 on iteration 8
Process 11 has data 13.000000 on iteration 8
Process 2 has data 13.000000 on iteration 8
Process 4 has data 13.000000 on iteration 8
Process 6 has data 13.000000 on iteration 8
Process 3 has data 13.000000 on iteration 8
Process 1 has data 13.000000 on iteration 8
Process 5 has data 13.000000 on iteration 8
Process 7 has data 13.000000 on iteration 8
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 16 ./a.out -h 1
Process 0 has data 16.000000 on iteration 8
Process 4 has data 16.000000 on iteration 8
Process 5 has data 16.000000 on iteration 8
Process 8 has data 16.000000 on iteration 8
Process 6 has data 16.000000 on iteration 8
Process 9 has data 16.000000 on iteration 8
Process 7 has data 16.000000 on iteration 8
Process 12 has data 16.000000 on iteration 8
Process 14 has data 16.000000 on iteration 8
Process 1 has data 16.000000 on iteration 8
Process 13 has data 16.000000 on iteration 8
Process 10 has data 16.000000 on iteration 8
Process 2 has data 16.000000 on iteration 8
Process 15 has data 16.000000 on iteration 8
Process 11 has data 16.000000 on iteration 8
Process 3 has data 16.000000 on iteration 8
aaron@AaronPC:~/Documents/MPI/HW2$ █

```

Figure 14: Butterfly high to low


```
aaron@Aaron-PC:~/Documents/MPI/HW2$ mpiexec -np 1 ./hw2-bfly.out -l 1048576
P0 has result 1.000000
Average Time = 0.000000
aaron@Aaron-PC:~/Documents/MPI/HW2$ mpiexec -np 2 ./hw2-bfly.out -l 1048576
P0 has result 2.000000
Average Time = 0.043855
aaron@Aaron-PC:~/Documents/MPI/HW2$ mpiexec -np 4 ./hw2-bfly.out -l 128
P0 has result 4.000000
Average Time = 0.025198
aaron@Aaron-PC:~/Documents/MPI/HW2$ mpiexec -np 32 ./hw2-bfly.out -l 1024
P0 has result 32.000000
Average Time = 0.327194
aaron@Aaron-PC:~/Documents/MPI/HW2$ mpiexec -np 16 ./hw2-bfly.out -l 1
P0 has result 16.000000
Average Time = 0.114000
aaron@Aaron-PC:~/Documents/MPI/HW2$ █
```

Figure 15: Butterfly low to high


```
aaron@AaronPC: ~/Documents/MPI/HW2
aaron@AaronPC: ~/Docum... x aaron@AaronPC: ~/Docum... x aaron@AaronPC: ~/Docum... x aaron@AaronPC: ~/Docum... x
[holtat@login04 HW2]$ python autoscript.py
sbatch script_l_1

sbatch script_l_8

sbatch script_l_64

sbatch script_l_512

sbatch script_l_4096

sbatch script_l_32768

sbatch script_l_262144

sbatch script_l_1048576

sbatch script_l_4
sbatch: error: Batch script is empty!

[holtat@login04 HW2]$ watch squeue -u holtat
[holtat@login04 HW2]$ head -n 2 results_128_l*
==> results_128_l_1048576.out <==
P0 has result 128.000000
Average Time = 0.174552

==> results_128_l_1.out <==
P0 has result 128.000000
Average Time = 0.000199

==> results_128_l_262144.out <==
P0 has result 128.000000
Average Time = 0.046847

==> results_128_l_32768.out <==
P0 has result 128.000000
Average Time = 0.004774

==> results_128_l_4096.out <==
P0 has result 128.000000
Average Time = 0.000835

==> results_128_l_512.out <==
```

Figure 16: Butterfly with results on Janus

```
aaron@AaronPC:~/Documents/MPI/HW2$ mpicc hw2-reduce.c -lm
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 1 ./a.out -h
Final Result is 1.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 2 ./a.out -h
Final Result is 2.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 5 ./a.out -h
Final Result is 5.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 7 ./a.out -h
Final Result is 7.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 13 ./a.out -h
Final Result is 13.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 16' ./a.out -h
> ^C
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 16 ./a.out -h
Final Result is 16.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 32 ./a.out -h
Final Result is 32.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 128 ./a.out -h
Final Result is 128.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 129 ./a.out -h
Final Result is 129.000000
aaron@AaronPC:~/Documents/MPI/HW2$ █
```

Figure 17: Reduce high to low

```
aaron@AaronPC: ~/Documents/MPI/HW2
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 1 ./a.out -l
Final Result is 1.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 2 ./a.out -l
Final Result is 2.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 3 ./a.out -l
Final Result is 3.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 4 ./a.out -l
Final Result is 4.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 5 ./a.out -l
Final Result is 5.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 17 ./a.out -l
Final Result is 17.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 32 ./a.out -l
Final Result is 32.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 64 ./a.out -l
Final Result is 64.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -np 137 ./a.out -l
Final Result is 137.000000
aaron@AaronPC:~/Documents/MPI/HW2$ █
```

Figure 18: Reduce low to high

```
aaron@AaronPC:~/Documents/MPI/HW2$ mpicc hw2-fanin-fanout.c -lm
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -n 1 ./a.out -h
Result after reduce on rank 0 is 1.000000
After bcast, process 0 has data = 1.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -n 4 ./a.out -h
Result after reduce on rank 0 is 4.000000
After bcast, process 0 has data = 4.000000
After bcast, process 1 has data = 4.000000
After bcast, process 2 has data = 4.000000
After bcast, process 3 has data = 4.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -n 7 ./a.out -h
Result after reduce on rank 0 is 7.000000
After bcast, process 0 has data = 7.000000
After bcast, process 1 has data = 7.000000
After bcast, process 2 has data = 7.000000
After bcast, process 5 has data = 7.000000
After bcast, process 3 has data = 7.000000
After bcast, process 6 has data = 7.000000
After bcast, process 4 has data = 7.000000
aaron@AaronPC:~/Documents/MPI/HW2$ mpiexec -n 16 ./a.out -h
Result after reduce on rank 0 is 16.000000
After bcast, process 0 has data = 16.000000
After bcast, process 2 has data = 16.000000
After bcast, process 4 has data = 16.000000
After bcast, process 8 has data = 16.000000
After bcast, process 6 has data = 16.000000
After bcast, process 14 has data = 16.000000
After bcast, process 10 has data = 16.000000
After bcast, process 12 has data = 16.000000
After bcast, process 1 has data = 16.000000
After bcast, process 9 has data = 16.000000
After bcast, process 3 has data = 16.000000
After bcast, process 11 has data = 16.000000
After bcast, process 7 has data = 16.000000
After bcast, process 5 has data = 16.000000
After bcast, process 13 has data = 16.000000
After bcast, process 15 has data = 16.000000
aaron@AaronPC:~/Documents/MPI/HW2$
```

Figure 19: Fan in fan out all reduce

Appendix B: Code

Butterfly:

```
#include "mpi.h"
#include "stdio.h"
#include "stdlib.h"
#include "math.h"
#include "string.h"
#include <time.h>

//Aaron Holt
//instructions
/*
Compile with $mpicc hw2-fly2.c -lm
run with $mpiexec -np NP ./a.out -type SIZE
where NP is process number
-type is -h or -l for high to low and low to high
SIZE is buffer size
*/

int main(int argc, char *argv[])
{
    //n is nearest power of 2 processes
    //if 3 processes are used, n will be 4.
    int n;
    //number of iterations needed to reduce
    int iterations;
    //Size of buffer
    int size;

    //High-Low or Low-High
    //1 = h->l
    //0 = l->h default
    int type = 1;
    //counters
    int i, j, k, jj, kk;

    char option_h2l[] = "-h";
    char option_H2L[] = "-H";
    char option_l2h[] = "-l";
    char option_L2H[] = "-L";
```

```

//Argument parsing.
for (j=1; j<argc; j++)
{
    //Get tag (operation type) value if present
    if (sscanf (argv[j], "%i", &size)!=1) {}

    //High to low or low to high
    if ((strcmp(argv[j], option_h2l)==0) || (strcmp(argv[j], option_H2L)==0)) {
        // printf("%s\n", argv[j]);
        type = 1;
    }
    if ((strcmp(argv[j], option_l2h)==0) || (strcmp(argv[j], option_L2H)==0)) {
        // printf("%s\n", argv[j]);
        type = 0;
    }
}

// printf("Size = %d\n", size);

//For now, hardcode tag (operation)
int tag = 0; //tag = 0 => addition

// Initialize the MPI environment
MPI_Init(NULL, NULL);

// Get the number of processes
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Get the rank of the process
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

// Get the name of the processor
char processor_name[MPI_MAX_PROCESSOR_NAME];
int name_len;
MPI_Get_processor_name(processor_name, &name_len);

//first masks to be used with l-h and h-l
int bitmask;

//Find out how many iterations are needed
//Find bitmask

```



```

    n = pow(2, ceil(log(world_size)/log(2)));
    iterations = log(n)/log(2);

//Timing
double total_time;
total_time = 0;
double starttime, endtime;

int deliver_to;    //who to send to
int receive_from; //who to receive from

//Dynamically allocate arrays
double *buffer_1;    //buffer to send
double *buffer_local; //store local results
buffer_1 = (double*) malloc(size*sizeof(double)+1);
buffer_local = (double*) malloc(size*sizeof(double)+1);
buffer_1[size] = '\0';
buffer_local[size] = '\0';
signed int *receive;    //Which processes should send
signed int *send;        //Which processes should receive
send = malloc(n*sizeof(int)+1);
receive = malloc(n*sizeof(int)+1);

// printf("%f\n", (double)sizeof(buffer_1));

// Account for iterations where 1 process needs to send twice
    signed int extra_send[2];
    extra_send[0] = -1;
    extra_send[1] = -1;

//fill arrays
for (j=0; j<n; j++)
{
    send[j] = (int)0;
    receive[j] = (int)0;
    // printf("%d ", send[j]);
    // printf("\n");
}
for (j=0; j<size; j++)
{
    buffer_1[j] = (double)1.0;
    buffer_local[j] = (double)1.0;
    // printf("%f ", buffer_1[j]);
    // printf("\n");
}

//10 warmup + 10 timed
for (kk=0; kk<20; kk++)

```

```

{
    //Starting bitmasks
    if (type == 1)
    {
        bitmask = n/2;
    }
    else
    {
        bitmask = 1;
    }

    for (k=0; k<size; k++)
    {
        buffer_1[k] = 1.0;
    }
    for (k=0; k<size; k++)
    {
        buffer_local[k] = 1.0;
    }

    if (kk>=10)
    {
        starttime = MPI_Wtime();
    }

    //Run to 2*iterations to account for deadlock
    //if everything sends/receives at once
    //Alternate sending and receiving
    for (i=0; i<2*iterations; i++)
    {
        // if (world_rank == 0)
        // {
        //     printf("i=%d, imod2=%d, bitmask=%d\n", i, i%2, bitmask);

        // }

        // printf("Here TOP, world rank %d, iteration %d!\n", world_rank, i);
    // MPI_Barrier(MPI_COMM_WORLD);

    //zero arrays
    for (j=0; j<world_size; j++)
    {
        send[j] = -1;
        receive[j] = -1;
    }
}

```



```

    }
    extra_send[0] = -1;
    extra_send[1] = -1;

//Calculate who sends
if (type == 1 || type == 0)
{
    for (jj = 0; jj<world_size; jj++)
    {
        deliver_to = jj^bitmask;

        //On even count, only deliver if deliver_to>your_rank
        if ((deliver_to > jj) && (i%2 == 2 || i%2 == 0))
        {
            if (deliver_to<world_size)
            {
                send[jj] = deliver_to;
            }
        }

        //On odd count, only deliver if deliver_to<your_rank
        if ((deliver_to < jj) && (i%2 == 1) && (deliver_to>=0))
        {
            send[jj] = deliver_to;
        }
        else if (i%2 == 1)
        {
            send[jj] = -1;
        }
    }
}

//Take care of odd process numbers (ie np=5...)
//Only do these after 1st iteration, on even count
if (i>1)
{
    // printf("HERE, iteration %d\n", i);
    if (world_rank+2*bitmask>=world_size &&
        world_rank+bitmask<world_size &&
        (i%2 == 2 || i%2 == 0))
    {
        // printf("HERE22, iteration %d\n", i);
        extra_send[0] = world_rank; //from
        extra_send[1] = world_rank + bitmask; //to
    }
}

```

```

        int normal_deliver;
        normal_deliver = world_rank ^ bitmask;
        if (extra_send[1]==normal_deliver)
        {
            extra_send[0] = -1;
            extra_send[1] = -1;
        }

        if (i>1)
        {
            int normal_receive;
            normal_receive = world_rank ^ bitmask;
            if (normal_receive>=world_size)
            {
                normal_receive = world_rank - bitmask;
                extra_send[0] = normal_receive;
                extra_send[1] = world_rank;
            }
        }
    }

    //Debug
    // if (world_rank == 0)
    // {
    //     for (k = 0; k<world_size; k++)
    //     {
    //         printf("%d ", send[k]);
    //     }
    //     printf(" iteration %d \n", i);
    // }
    // printf("Extra send from %d, to %d, iteration %d \n",
    //         extra_send[0], extra_send[1], i);

    // printf("Here BEFORE SEND, world rank %d, iteration %d!\n",
world_rank, i);

    // MPI_Barrier(MPI_COMM_WORLD);

    //Send
    if (send[world_rank]>=0)
    {
        // printf(" Process %d delivers to %d on iteration %d \n",
        //         world_rank, send[world_rank], i);
        // MPI_Send(buffer_1, size, MPI_DOUBLE, send[world_rank], tag,
MPI_COMM_WORLD);
    }

```

```

//receive
for (j=0; j<world_size; j++)
{
    if (send[j]==world_rank)
    {
        //      printf("Process %d receives from %d on iteration %d \n",
        //                                     world_rank, j, i);
        MPI_Recv(buffer_local, size, MPI_DOUBLE, j, tag,
MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        for (k=0; k<size; k++)
        {
            buffer_local[k] = buffer_local[k] + buffer_1[k];
        }
    }
}

if (i%2 == 0 && type == 1)
{
    if (extra_send[0]==world_rank)
    {
        // printf("  Process %d delivers to %d on iteration %d \n",
        //                                     world_rank, extra_send[1], i);
        MPI_Send(buffer_1, size, MPI_DOUBLE, extra_send[1],
tag, MPI_COMM_WORLD);
    }

    if (extra_send[1]==world_rank)
    {
        // printf("Process %d receives from %d on iteration %d \n",
        //                                     world_rank, j, i);
        MPI_Recv(buffer_local, size, MPI_DOUBLE,
extra_send[0], tag, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        for (k=0; k<size; k++)
        {
            buffer_local[k] = buffer_local[k] + buffer_1[k];
        }
    }
}

//      printf("Here AFTER SEND, world rank %d, iteration %d!\n", world_rank, i);
//      MPI_Barrier(MPI_COMM_WORLD);

```

```

//Update bitmask, buffer_local on evens
if (i%2 == 1)
{
    if (type == 1)
    {
        bitmask = bitmask / 2;
    }
    else
    {
        bitmask = bitmask * 2;
    }

    for (k=0; k<size; k++)
    {
        buffer_1[k] = buffer_local[k];
    }
    // MPI_Barrier(MPI_COMM_WORLD);
    // printf("Process %d has data %f on iteration %d \n",
    //       world_rank, buffer_local[0], i);
}
}
if (kk>=10)
{
    endtime = MPI_Wtime();
    total_time = total_time + endtime - starttime;
}
}

```

```

MPI_Barrier(MPI_COMM_WORLD);
if (world_rank == 0)
{
    printf("P0 has result %f\n", buffer_local[0]);
    printf("Average Time = %f\n", total_time/10);
}
MPI_Barrier(MPI_COMM_WORLD);

```

```

// MPI_Barrier(MPI_COMM_WORLD);
// // printf("Process %d has data %f on iteration %d \n",
// //       world_rank, buffer_local[0], i);
// if (world_rank == 0)
// {
//     printf("P0 ARRAY : ");
//     for (i=0; i<size; i++)
//     {

```

```
        //      printf("%f\n", buffer_local[i]);  
        //    }  
    // }
```

```
    // MPI_Barrier(MPI_COMM_WORLD);
```

```
    //Match malloc with free  
    free (buffer_1);  
    free (buffer_local);  
    free (send);  
    free (receive);  
  
    return 0;  
  
    MPI_Finalize();  
}
```

Broadcast:

```
#include "mpi.h"
#include "stdio.h"
#include "stdlib.h"
#include "math.h"
#include "string.h"
#include <time.h>
```

```
//Aaron Holt
//instructions
/*
Compile with $mpicc hw2-fly2.c -lm
run with $mpiexec -np NP ./a.out -type
where NP is process number
-type is -h or -l for high to low and low to high
*/
```

```
int main(int argc, char *argv[])
{

    //n is nearest power of 2 processes
    //if 3 processes are used, n will be 4.
    int n;
    //number of iterations needed to reduce
    int iterations;
    //tag to determine operation
    int tag;

    //High-Low or Low-High
    //1 = h->l
    //0 = l->h default
    int type = 0;
    int j; //counter in arg parse loop

    char option_h2l[] = "-h";
    char option_H2L[] = "-H";
    char option_l2h[] = "-l";
    char option_L2H[] = "-L";

    //Argument parsing.
    for (j=1; j<argc; j++)
```

```

{
    //Get tag (operation type) value if present
    if (sscanf (argv[j], "%i", &tag)!=1) {}

    //High to low or low to high
    if ((strcmp(argv[j], option_h2l)==0) || (strcmp(argv[j], option_H2L)==0)) {
        // printf("%s\n", argv[j]);
        type = 1;
    }
    if ((strcmp(argv[j], option_l2h)==0) || (strcmp(argv[j], option_L2H)==0)) {
        // printf("%s\n", argv[j]);
        type = 0;
    }
}

//For now, hardcode tag (operation)
tag = 0; //tag = 0 => addition

// Initialize the MPI environment
MPI_Init(NULL, NULL);

// Get the number of processes
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Get the rank of the process
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

// Get the name of the processor
char processor_name[MPI_MAX_PROCESSOR_NAME];
int name_len;
MPI_Get_processor_name(processor_name, &name_len);

double data;

if (world_rank == 0)
{
    data = 12345.6789;
}

//first masks to be used with l-h and h-l
int bitmask;

//Find out how many iterations are needed
//Find bitmask
n = pow(2, ceil(log(world_size)/log(2)));

```

```

        iterations = log(n)/log(2);
        //find bitmask
        if (type == 1)
        {
            bitmask = n/2;
        }
        else
        {
            bitmask = 1;
        }

        // if (world_rank == 0)
// {
//     printf("n = %d, iterations = %d, mask = %d, type = %d \n", n,
//         iterations, bitmask, type);
// }

// Variables
int i; //counter
int deliver_to; //who to send to
int receive_from; //who to receive from
int received;

if (world_rank == 0)
{
    received = 1; //In bcast, each process should receive only once
}
else
{
    received = 0;
}

for (i = 0; i<iterations; i++)
{
    //////////// High to Low//////////
    if (type == 1)
    {
        // Check which processes should be sending
        //can only send if you've received
        if (received == 1)
        {
            deliver_to = world_rank + pow(2,i);
            if (deliver_to < world_size)
            {
                // printf("Process %d delivers to %d on iteration %d \n",
                //     world_rank, deliver_to, i);
                MPI_Send(&data, 1, MPI_DOUBLE, deliver_to, tag,

```



```

MPI_COMM_WORLD);
    }
}
else
{
    //Check which processes should be receiving
    if (received == 0)
    {
        //Account for a processes that doesn't need to do anything
        //for current iteration
        if (world_rank < pow(2,i+1))
        {
            receive_from = world_rank - pow(2,i);
            // printf("Process %d receives from %d \n", world_rank,
receive_from);

            MPI_Recv(&data, 1, MPI_DOUBLE, receive_from, tag,
MPI_COMM_WORLD,

                    MPI_STATUS_IGNORE);
            received = 1;
        }
    }
}
}
//////////Low to high//////////
else if (type == 0)
{
    // Check which processes should be sending
    //can only send if you've received
    if (received == 1)
    {
        deliver_to = world_rank + (int)pow(2, iterations-i-1);
        if (deliver_to < world_size)
        {
            // printf("Process %d delivers to %d on iteration %d \n",
// world_rank, deliver_to, i);
            MPI_Send(&data, 1, MPI_DOUBLE, deliver_to, tag,
MPI_COMM_WORLD);
        }
    }

    //Check which processes should be receiving
    else if (received == 0)
    {
        //Account for a processes that doesn't need to do anything
        //for current iteration
        if (world_rank % (int)pow(2, iterations-i-1) == 0)
        {
            double local_result;

```

```

        receive_from = world_rank - (int)pow(2, iterations-i-1);

        if (receive_from >= 0)
        {
            // printf("Process %d receives from %d on iteration %d. \n",
            // world_rank, receive_from, i);
            MPI_Recv(&local_result, 1, MPI_DOUBLE, receive_from,
tag, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
            data = local_result;
            received = 1;
        }
    }
}

//Debug
// if (world_rank == 0)
// {
//     printf("Type = %d, pow = %d, iteration = %d, n/2 = %d \n",
//           type, (int)pow(2, iterations-i-1), i, n/2);
// }

printf("Process %d has data = %f \n", world_rank, data);

MPI_Finalize();
}

```

Reduce:

```
#include "mpi.h"
#include "stdio.h"
#include "stdlib.h"
#include "math.h"
#include "string.h"
#include <time.h>
```

```
//Aaron Holt
```

```
int main(int argc, char *argv[])
{
```

```
    //n is nearest power of 2 processes
    //if 3 processes are used, n will be 4.
    int n;
    //number of iterations needed to reduce
    int iterations;
    //tag to determine operation
    int tag;
```

```
    //High-Low or Low-High
    //1 = h->l
    //0 = l->h default
    int type = 0;
    int j; //counter in arg parse loop
```

```
    char option_h2l[] = "-h";
    char option_H2L[] = "-H";
    char option_l2h[] = "-l";
    char option_L2H[] = "-L";
```

```
    //Argument parsing.
    for (j=1; j<argc; j++)
    {
```

```
        //Get tag (operation type) value if present
        if (sscanf (argv[j], "%i", &tag)!=1) {}
```

```
        //High to low or low to high
        if ((strcmp(argv[j], option_h2l)==0) || (strcmp(argv[j], option_H2L)==0)) {
            // printf("%s\n", argv[j]);
```

```

        type = 1;
    }
    if ((strcmp(argv[j], option_l2h)==0) || (strcmp(argv[j], option_L2H)==0)) {
        // printf("%s\n", argv[j]);
        type = 0;
    }
}

//For now, hardcode tag (operation)
tag = 0; //tag = 0 => addition

//data to add up, each process will have its own data
double data;

// Initialize the MPI environment
MPI_Init(NULL, NULL);

// Get the number of processes
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Get the rank of the process
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

// Get the name of the processor
char processor_name[MPI_MAX_PROCESSOR_NAME];
int name_len;
MPI_Get_processor_name(processor_name, &name_len);

//Initialize data on all processes
data = 1;

//first masks to be used with l-h and h-l
int bitmask;

//Find out how many iterations are needed
//Find bitmask
n = pow(2, ceil(log(world_size)/log(2)));
iterations = log(n)/log(2);
//find bitmask
if (type == 1)
{
    bitmask = n/2;
}
else
{

```

```

        bitmask = 1;
    }

    if (world_rank == 0)
    {
        // printf("n = %d, iterations = %d, mask = %d, type = %d \n", n,
        //     iterations, bitmask, type);
    }

    // Variables
    int i; //counter
    int deliver_to;    //who to send to
    int receive_from;  //who to receive from
    int delivered = 0; //In reduce, a process should only deliver once

    for (i = 0; i < iterations; i++)
    {
        //////////// High to Low ////////////
        if (type == 1)
        {
            // Check which processes should be sending
            if ((world_rank & bitmask) == bitmask)
            {
                if (delivered == 0)
                {
                    deliver_to = world_rank ^ bitmask;
                    // printf("Process %d delivers to %d on iteration %d \n",
                    //     world_rank, deliver_to, i);
                    MPI_Send(&data, 1, MPI_DOUBLE, deliver_to, tag,
MPI_COMM_WORLD);
                    delivered = 1;
                }
            }
            else
            {
                //Check which processes should be receiving
                if (world_rank < bitmask)
                {
                    double local_result;
                    local_result = data;
                    receive_from = world_rank ^ bitmask;

                    //Account for a process that doesn't need to do anything
                    //during a send
                    if (receive_from < world_size)
                    {
                        // printf("Process %d receives from %d \n", world_rank,
receive_from);

```

```

MPI_COMM_WORLD,
MPI_Recv(&data, 1, MPI_DOUBLE, receive_from, tag,
MPI_STATUS_IGNORE);
// Sum data
data = local_result + data;
}
}
}
}
//Low to high
else if (type == 0)
{
// Check which processes should be sending
if ((world_rank & bitmask) == bitmask)
{
if (delivered == 0)
{
deliver_to = world_rank ^ bitmask;
// printf("Process %d delivers to %d on iteration %d \n",
// world_rank, deliver_to, i);
MPI_Send(&data, 1, MPI_DOUBLE, deliver_to, tag,
MPI_COMM_WORLD);
delivered = 1;
}
}
else
{
//Check which processes should be receiving
if ((world_rank % (int)pow(2,(i+1))) == 0)
{
double local_result;
local_result = data;
receive_from = world_rank ^ bitmask;

//Account for a process that doesn't need to do anything
//during a send
if (receive_from < world_size)
{
// printf("Process %d receives from %d on iteration %d \n",
// world_rank, receive_from, i);
MPI_Recv(&data, 1, MPI_DOUBLE, receive_from, tag,
MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
// Sum data
data = local_result + data;
}
}
}
}
}

```

```

    }

    //Debug
    // if (world_rank == 0)
    // {
    //     printf("Type = %d, Mask = %d, iteration = %d \n", type, bitmask, i);
    // }

    //Update bitmask
    if (type == 1)
    {
        bitmask = bitmask / 2;
    }
    else
    {
        bitmask = bitmask * 2;
    }
}

if (world_rank == 0)
{
    printf("Final Result is %f \n", data);
}

MPI_Finalize();
}

```

Simple Allreduce

```
#include "mpi.h"
#include "stdio.h"
#include "stdlib.h"
#include "math.h"
#include "string.h"
#include <time.h>

//Aaron Holt

int main(int argc, char *argv[])
{
    //n is nearest power of 2 processes
    //if 3 processes are used, n will be 4.
    int n;
    //number of iterations needed to reduce
    int iterations;
    //tag to determine operation
    int tag;

    //For now, hardcode tag (operation)
    tag = 0; //tag = 0 => addition

    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    //Timing
    // clock_t t;
    // t = clock();
    // double total_time;
```



```

// total_time = 0;
double total_time;
total_time = 0;
double starttime, endtime;
starttime = MPI_Wtime();

for (int j=0; j<=11; j++)
{

    /// REDUCE
    double data, local_result;
    int received;
    data = 1.0;
    received = 0;

    if (world_rank == 0)
    {
        received = 1;
        MPI_Send(&data, 1, MPI_DOUBLE, world_rank+1, tag, MPI_COMM_WORLD);
    }

    while (1)
    {
        if (received == 0)
        {
            local_result = data;
            MPI_Recv(&data, 1, MPI_DOUBLE, world_rank-1, tag, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
            data = local_result + data;
            if (world_rank == world_size-1)
            {
                break;
            }
            MPI_Send(&data, 1, MPI_DOUBLE, world_rank+1, tag, MPI_COMM_WORLD);
            break;
        }
        break;
    }
    // printf("Process %d has data = %f \n", world_rank, data);

    /// BCAST
    received = 0;
    if (world_rank == world_size-1)
    {
        received = 1;
        MPI_Send(&data, 1, MPI_DOUBLE, world_rank-1, tag, MPI_COMM_WORLD);
    }
    while (1)

```

```

{
    if (received == 0)
    {
        MPI_Recv(&data, 1, MPI_DOUBLE, world_rank+1, tag, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        if (world_rank == 0)
        {
            break;
        }
        MPI_Send(&data, 1, MPI_DOUBLE, world_rank-1, tag, MPI_COMM_WORLD);
        break;
    }
    break;
}
// printf("Process %d has data = %f \n", world_rank, data);

// t = clock() - t;
// double time_taken = ((double)t)/CLOCKS_PER_SEC;
// double global_time;
// MPI_Allreduce(&time_taken, &global_time, 1,
//              MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
// total_time += global_time;
// if (world_rank <= 1)
// {
//     printf("Iteration %d, My time %f, total_time %f\n",
//           j, endtime-starttime, total_time);
// }

}
//Timing
endtime = MPI_Wtime();
total_time = total_time + endtime - starttime;
total_time = total_time / 11.0;
if (world_rank == 0)
{
    printf("%f\n\n", total_time);
}

// printf("Process %d has data = %f \n", world_rank, data);

MPI_Finalize();

```

```
}
```

Fan in fan out

```
#include "mpi.h"
#include "stdio.h"
#include "stdlib.h"
#include "math.h"
#include "string.h"
#include <time.h>
```

```
//Aaron Holt
```

```
int main(int argc, char *argv[])
{
```

```
    //n is nearest power of 2 processes
    //if 3 processes are used, n will be 4.
    int n;
    //number of iterations needed to reduce
    int iterations;
    //tag to determine operation
    int tag;
```

```
    //High-Low or Low-High
    //1 = h->l
    //0 = l->h default
    int type = 0;
    int j; //counter in arg parse loop
```

```
    char option_h2l[] = "-h";
    char option_H2L[] = "-H";
    char option_l2h[] = "-l";
    char option_L2H[] = "-L";
```

```
    //Argument parsing.
    for (j=1; j<argc; j++)
    {
```

```
        //Get tag (operation type) value if present
        if (sscanf (argv[j], "%i", &tag)!=1) {}
```

```
        //High to low or low to high
        if ((strcmp(argv[j], option_h2l)==0) || (strcmp(argv[j], option_H2L)==0)) {
            // printf("%s\n", argv[j]);
```

```

        type = 1;
    }
    if ((strcmp(argv[j], option_l2h)==0) || (strcmp(argv[j], option_L2H)==0)) {
        // printf("%s\n", argv[j]);
        type = 0;
    }
}

//For now, hardcode tag (operation)
tag = 0; //tag = 0 => addition

//data to add up, each process will have its own data
double data;

// Initialize the MPI environment
MPI_Init(NULL, NULL);

// Get the number of processes
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Get the rank of the process
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

// Get the name of the processor
char processor_name[MPI_MAX_PROCESSOR_NAME];
int name_len;
MPI_Get_processor_name(processor_name, &name_len);

//Initialize data on all processes
data = 1;

//first masks to be used with l-h and h-l
int bitmask;

//Find out how many iterations are needed
//Find bitmask
n = pow(2, ceil(log(world_size)/log(2)));
iterations = log(n)/log(2);
//find bitmask
if (type == 1)
{
    bitmask = n/2;
}
else
{

```

```

        bitmask = 1;
    }

    if (world_rank == 0)
    {
        // printf("n = %d, iterations = %d, mask = %d, type = %d \n", n,
        //     iterations, bitmask, type);
    }

// Variables
int i; //counter
int deliver_to;    //who to send to
int receive_from;  //who to receive from
int delivered = 0; //In reduce, a process should only deliver once

for (i = 0; i < iterations; i++)
{
    //////////// High to Low ////////////
    if (type == 1)
    {
        // Check which processes should be sending
        if ((world_rank & bitmask) == bitmask)
        {
            if (delivered == 0)
            {
                deliver_to = world_rank ^ bitmask;
                // printf("Process %d delivers to %d on iteration %d \n",
                //     world_rank, deliver_to, i);
                MPI_Send(&data, 1, MPI_DOUBLE, deliver_to, tag,
MPI_COMM_WORLD);
                delivered = 1;
            }
        }
        else
        {
            //Check which processes should be receiving
            if (world_rank < bitmask)
            {
                double local_result;
                local_result = data;
                receive_from = world_rank ^ bitmask;

                //Account for a process that doesn't need to do anything
                //during a send
                if (receive_from < world_size)
                {
                    // printf("Process %d receives from %d \n", world_rank,
receive_from);

```

```

MPI_COMM_WORLD,

MPI_Recv(&data, 1, MPI_DOUBLE, receive_from, tag,

MPI_STATUS_IGNORE);
// Sum data
data = local_result + data;
}
}
}
}
//Low to high////////////////////////////////////
else if (type == 0)
{
// Check which processes should be sending
if ((world_rank & bitmask) == bitmask)
{
if (delivered == 0)
{
deliver_to = world_rank ^ bitmask;
// printf("Process %d delivers to %d on iteration %d \n",
// world_rank, deliver_to, i);
MPI_Send(&data, 1, MPI_DOUBLE, deliver_to, tag,
MPI_COMM_WORLD);
delivered = 1;
}
}
else
{
//Check which processes should be receiving
if ((world_rank % (int)pow(2,(i+1))) == 0)
{
double local_result;
local_result = data;
receive_from = world_rank ^ bitmask;

//Account for a process that doesn't need to do anything
//during a send
if (receive_from < world_size)
{
// printf("Process %d receives from %d on iteration %d \n",
// world_rank, receive_from, i);
MPI_Recv(&data, 1, MPI_DOUBLE, receive_from, tag,
MPI_COMM_WORLD,

MPI_STATUS_IGNORE);
// Sum data
data = local_result + data;
}
}
}
}
}

```

```

    }

    //Debug
    // if (world_rank == 0)
    // {
    //     printf("Type = %d, Mask = %d, iteration = %d \n", type, bitmask, i);
    // }

    //Update bitmask
    if (type == 1)
    {
        bitmask = bitmask / 2;
    }
    else
    {
        bitmask = bitmask * 2;
    }
}

if (world_rank == 0)
{
    printf("Result after reduce on rank %d is %f \n", world_rank, data);
}

if (type == 1)
{
    bitmask = n/2;
}
else
{
    bitmask = 1;
}

    // if (world_rank == 0)
// {
//     printf("n = %d, iterations = %d, mask = %d, type = %d \n", n,
//         iterations, bitmask, type);
// }

```

int received; //In bcast, each process should receive only once

```

if (world_rank == 0)
{
    received = 1; //In bcast, each process should receive only once
}
else

```

```

{
    received = 0;
}

for (i = 0; i < iterations; i++)
{
    /////////////// High to Low////////////////////
    if (type == 1)
    {
        // Check which processes should be sending
        //can only send if you've received
        if (received == 1)
        {
            deliver_to = world_rank + pow(2,i);
            if (deliver_to < world_size)
            {
                // printf("Process %d delivers to %d on iteration %d \n",
                //       world_rank, deliver_to, i);
                MPI_Send(&data, 1, MPI_DOUBLE, deliver_to, tag,
MPI_COMM_WORLD);
            }
        }
        else
        {
            //Check which processes should be receiving
            if (received == 0)
            {
                //Account for a processes that doesn't need to do anything
                //for current iteration
                if (world_rank < pow(2,i+1))
                {
                    receive_from = world_rank - pow(2,i);
                    // printf("Process %d receives from %d \n", world_rank,
receive_from);
                    MPI_Recv(&data, 1, MPI_DOUBLE, receive_from, tag,
MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
                    received = 1;
                }
            }
        }
    }
    ///////////////Low to high////////////////////
    else if (type == 0)
    {
        // Check which processes should be sending
        //can only send if you've received

```



```

        if (received == 1)
        {
            deliver_to = world_rank + (int)pow(2, iterations-i-1);
            if (deliver_to < world_size)
            {
                // printf("Process %d delivers to %d on iteration %d \n",
                // world_rank, deliver_to, i);
                MPI_Send(&data, 1, MPI_DOUBLE, deliver_to, tag,
MPI_COMM_WORLD);
            }
        }

        //Check which processes should be receiving
        else if (received == 0)
        {
            //Account for a processes that doesn't need to do anything
            //for current iteration
            if (world_rank % (int)pow(2, iterations-i-1) == 0)
            {
                double local_result;
                receive_from = world_rank - (int)pow(2, iterations-i-1);

                if (receive_from >= 0)
                {
                    // printf("Process %d receives from %d on iteration %d. \n",
                    // world_rank, receive_from, i);
                    MPI_Recv(&local_result, 1, MPI_DOUBLE, receive_from,
tag, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
                    data = local_result;
                    received = 1;
                }
            }
        }
    }

    //Debug
    // if (world_rank == 0)
    // {
    //     printf("Type = %d, pow = %d, iteration = %d, n/2 = %d \n",
    //         type, (int)pow(2, iterations-i-1), i, n/2);
    // }
}

```

```
printf("After bcast, process %d has data = %f \n", world_rank, data);
```

```
MPI_Finalize();
```

```
}
```