Jonathan Alexander Gibson
jagibson44, T00198998
Program 5 (MPI Broadcast Comparison)

The purpose of the ***mpi_bcast*** program was to compare the results of a custom MPI library broadcast implementation with the default MPI library implementation, in terms of execution time. A C++ library was created to reflect the custom library implementation and is defined in ***custom_bcast.h*** and ***custom_bcast.cpp***. The goal was to run the custom and default library broadcast implementations, on an HPC cluster, using 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 26, and 28 processes while measuring the execution time for each. Each implementation was executed on one node and two nodes independently (with half of the processors on each node). The values were then averaged across three equivalent runs, for robustness. The data to be broadcasted is an array of 100,000 random elements of type ***double***. The hope was to gain enough useful data to inquire about the differences, scalability, and performances for the two implementations and the potential overhead associated with using two nodes, instead of just one.

The file ***mpi_bcast.cpp*** is the core of this program. It starts by including the ***custom_bcast*** library, defining the broadcast array length, and doing error checking for the root (included as a command line argument) and the MPI world size. Three different arrays are then dynamically allocated using ***calloc()***. Array ***org_bcast_arr*** is the array initialized (with all 0s), populated with random values, and broadcasted only by the root process. Array ***cust_bcast_arr*** is the array that is initialized (with all -1s) and collected by all processes, other than root, after ***org_bcast_arr*** is broadcasted, via custom library implementation. Array ***mpi_bcast_arr*** is the array that is initialized (with all -2s) and collected by all processes, other than root, after ***org_bcast_arr*** is broadcasted, via default library implementation.

Sufficient print statements are included in ***mpi_bcast.cpp*** to confirm proper operation of the code, complete with the first and last elements of the initialized, original, custom, and default arrays, before and after the broadcast functions. Each print statement is prefixed with the MPI process rank for confirmation. The process labeled as root is printed before broadcasting, while all relevant execution times are printed after broadcasting. Functions (like ***MPI_Barrier()***) and ***if*** statements are placed strategically in an attempt to preserve proper print order and to capture the most accurate wall time, for each section of code.

The output data from the HPC cluster was produced and collected very quickly, because the execution time for all runs was in the range of milliseconds. Despite the small runtimes, there were some significant changes in the results, between different runs, that accounted for important implications. The output data and Excel calculations are shown in Table 1 and Table 2 below.

Single Node Stats:

	Processors:	Single Node Time (s):			AVG (s):	Mult Increase:		
Custom:	2	0.000302	0.000267	0.000326	0.000298	1.00		
	4	0.000643	0.000730	0.000643	0.000672	2.25		
	6	0.000886	0.000999	0.000935	0.000940	3.15		
	8	0.001386	0.001421	0.001303	0.001370	4.59		
	10	0.001893	0.001938	0.002207	0.002013	6.75		
	12	0.002566	0.002417	0.002686	0.002556	8.57		
	14	0.003691	0.003506	0.003803	0.003667	12.29		
	16	0.006443	0.005832	0.006473	0.006249	20.95		
	18	0.005843	0.006292	0.006860	0.006332	21.22		
	20	0.006124	0.006508	0.007220	0.006617	22.18		
	26	0.008156	0.007790	0.008289	0.008078	27.08	AVG Mult:	
	28	0.008912	0.010121	0.010869	0.009967	33.41	14.77	
								AVG Diff:
								2.23602
Default:	2	0.000155	0.000181	0.000169	0.000168	1.00		
	4	0.000481	0.000430	0.000445	0.000452	2.69		
	6	0.000484	0.000549	0.000481	0.000505	3.00		
	8	0.000546	0.000551	0.000565	0.000554	3.29		
	10	0.000566	0.000606	0.000618	0.000597	3.54		
	12	0.000677	0.000636	0.000734	0.000682	4.05		
	14	0.001108	0.001014	0.001083	0.001068	6.35		
	16	0.001642	0.001474	0.001631	0.001582	9.40		
	18	0.001460	0.001603	0.001757	0.001607	9.54		
	20	0.001452	0.001530	0.001593	0.001525	9.06		
	26	0.001856	0.001748	0.001715	0.001773	10.53	AVG Mult:	
	28	0.001901	0.001883	0.001868	0.001884	11.19	6.60	

Table 1: *Execution results and multiplicative time increases for single node collective broadcast.*

Double Node Stats:

	Processors:	Double Node Time (s):			AVG (s):	Mult Increase:		
Custom:	2	0.000573	0.000564	0.000565	0.000567	1.00		
	4	0.020308	0.023572	0.017142	0.020341	35.85		
	6	0.020697	0.024334	0.031949	0.025660	45.23		
	8	0.057372	0.717812	0.054570	0.276585	487.52		
	10	0.051003	0.047130	0.039006	0.045713	80.58		
	12	0.084431	0.124270	0.112471	0.107057	188.70		
	14	0.064372	0.097563	0.051499	0.071145	125.40		
	16	0.140639	0.133796	0.120253	0.131563	231.90		
	18	0.092824	0.108171	0.101844	0.100946	177.93		
	20	0.158570	0.143854	0.136287	0.146237	257.76		
	26	0.216563	0.209156	0.190567	0.205429	362.10	AVG Mult:	
	28	0.196862	0.205125	0.220036	0.207341	365.47	214.40	
								AVG Diff:
								14.17449
Default:	2	0.000243	0.000240	0.000238	0.000240	1.00		
	4	0.001472	0.001477	0.001538	0.001496	6.22		
	6	0.000603	0.000694	0.000626	0.000641	2.67		
	8	0.014677	0.025128	0.015623	0.018476	76.88		
	10	0.001346	0.001330	0.001269	0.001315	5.47		
	12	0.001166	0.001129	0.001156	0.001150	4.79		
	14	0.001021	0.000985	0.000869	0.000958	3.99		
	16	0.014248	0.013028	0.010481	0.012586	52.37		
	18	0.000731	0.000781	0.000794	0.000769	3.20		
	20	0.000888	0.000788	0.000751	0.000809	3.37		
	26	0.000935	0.000865	0.000897	0.000899	3.74	AVG Mult:	
	28	0.000894	0.000886	0.000888	0.000889	3.70	15.13	

Table 2: *Execution results and multiplicative time increases for double node collective broadcast.*

There are a few things that are apparent from this dataset.

1. In general, the execution time for broadcasting increases with the number of processes.
2. The default MPI broadcast implementation is more efficient than the custom library implementation. This result can be observed via the ***AVG Mult*** values associated with each of the four test cases.
3. When the broadcast program was ran on one node, the multiplicative increase of execution time (for both custom and default MPI implementations) was relatively linear, consistent, predictable, non-erratic. In contrast, when the broadcast program was ran on two nodes, the multiplicative increase of execution time (for both custom and default MPI implementations) was non-linear, inconsistent, unpredictable, and erratic. This result can be observed via the ***AVG Diff*** values associated with the single and double node runs (where ***AVG Diff*** = ***AVG Mult***$_{Custom}$ / ***AVG Mult***$_{Default}$).

I produced data plots with MATLAB to show the relative differences for the four test cases. The plots are shown in Figure 1, below.

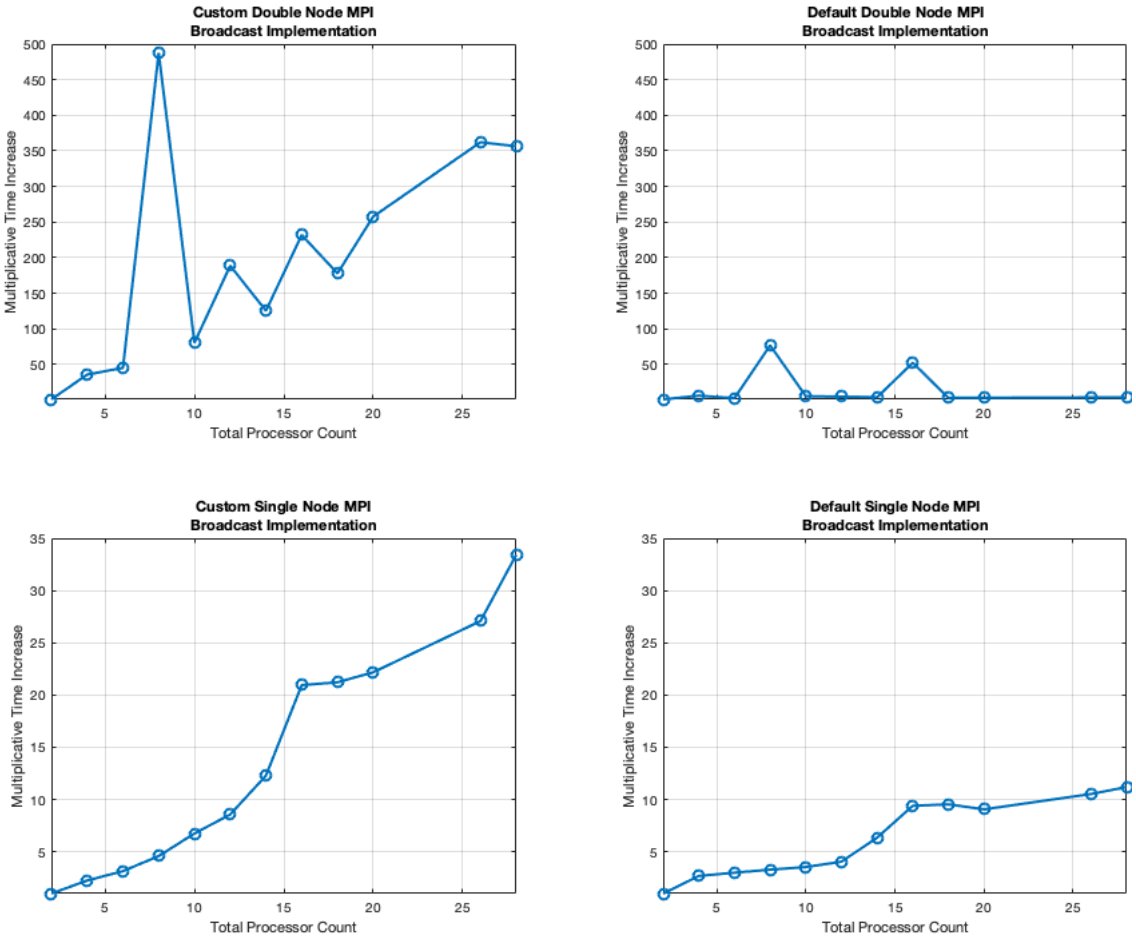

Figure 1: *Multiplicative time increases for the four different test cases.*

In addition to the statements listed above, there are some things that we can infer from these results.

1. The default library broadcast implementation (***MPI_Bcast()***) might have a mechanism to account for the increase in processes participating in a broadcast. I hypothesis that this is the case because, the increases in processor count have very small effects on the multiplicative execution time increases. I imagine that the default library implementation could use the other processes (other than root) to send data, that they have received from root, simultaneously, to alleviate some work from the root. This could account for the relatively low time increases (and overhead) associated with increases in processor count.
2. The custom library broadcast implementation has a similar linear increase to the default library implementation, associated with the increase in processor count, but the slope is much more steep. This implies that the execution time increases are almost directly related to the number of processes (the number of communications necessary for root to complete a broadcast).
3. The double node runs have very erratic and unpredictable behavior as the processor count increases. The spikes in the plots represent cases in which the average runtime took significantly longer, for a certain process count. These spikes are visible in both the custom and default library broadcast implementations. However, these spikes are much less apparent for the default library broadcast implementation. This fact furthers my suspicions towards the default library broadcast implementation having a mechanism to account for the number of participating processes, despite the processor node locations.
4. The single node runs have very little erratic or unpredictable behavior, as the processor count increases. This implies that there is an innate overhead that is associated with setting up the MPI environment and using communications between two or more nodes, rather than using just one.

Concerning the scalability, using the custom library implementation does not seem like a good option, for an ever increasing number of processes (or increasing buffer size). The scalability gets even worse when considering the custom library implementation with communications between two or more nodes. Fortunately, both cases for the default library implementation seem scalable for increasing an processor count and buffer size.

Because of the above inferences, I would desire to use collective communications with MPI on a single node, rather than between nodes. However, because these values are representative of millisecond range communications, this choice *should* have negligible effects on the program as a whole, especially when considering the runtime of an entire MPI application, with any level of complexity. But, just because these communication results *should* have negligible effects on an MPI application, doesn't mean they will; for robustness, these results should not be completely ignored.

The MATLAB code used to generate the plots, shown in Figure 1, is shown below.

```
%JONATHAN ALEXANDER GIBSON
%CSC 6740
%PARALLEL DISTRIBUTED ALGORITHMS
%DR. GHAFOOR
%PROGRAM 5 (MPI BCAST COMPARISON)

clear %clear all saved variable values
clc %clear the command window
close all %close all figures
format long %long variable format
%--------------------------------------------------------------------------

procs = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 26, 28]; %x-axis for all graphs

%(i) Custom Double Node Multiplicative Time Increase Per Total Processor Count
yMult1 = [1,35.85,45.23,487.52,80.58,188.7,125.4,231.9,177.93,257.76,362.1,356.47];
figure;
subplot(2, 2, 1);
plot(procs, yMult1, 'o-', 'LineWidth', 2, 'MarkerSize', 8);
grid on;
ylim([1, 500]);
```

```
xlim([2, 28]);
ylabel('Multiplicative Time Increase');
xlabel('Total Processor Count');
title({'Custom Double Node MPI', 'Broadcast Implementation'});
%--------------------------------------------------------------------------

%(ii) Default Double Node Multiplicative Time Increase Per Total Processor Count
yMult2 = [1,6.22,2.67,76.88,5.47,4.79,3.99,52.37,3.2,3.37,3.74,3.7];
subplot(2, 2, 2);
plot(procs, yMult2, 'o-', 'LineWidth', 2, 'MarkerSize', 8);
grid on;
ylim([1, 500]);
xlim([2, 28]);
ylabel('Multiplicative Time Increase');
xlabel('Total Processor Count');
title({'Default Double Node MPI', 'Broadcast Implementation'});
%--------------------------------------------------------------------------

%(iii) Custom Single Node Multiplicative Time Increase Per Total Processor Count
yMult3 = [1,2.25,3.15,4.59,6.75,8.57,12.29,20.95,21.22,22.18,27.08,33.41];
subplot(2, 2, 3);
plot(procs, yMult3, 'o-', 'LineWidth', 2, 'MarkerSize', 8);
grid on;
ylim([1, 35]);
xlim([2, 28]);
ylabel('Multiplicative Time Increase');
xlabel('Total Processor Count');
title({'Custom Single Node MPI', 'Broadcast Implementation'});
%--------------------------------------------------------------------------

%(iv) Default Single Node Multiplicative Time Increase Per Total Processor Count
yMult4 = [1,2.69,3,3.29,3.54,4.05,6.35,9.4,9.54,9.06,10.53,11.19];
subplot(2, 2, 4);
plot(procs, yMult4, 'o-', 'LineWidth', 2, 'MarkerSize', 8);
grid on;
ylim([1, 35]);
xlim([2, 28]);
ylabel('Multiplicative Time Increase');
xlabel('Total Processor Count');
title({'Default Single Node MPI', 'Broadcast Implementation'});
%--------------------------------------------------------------------------
```

Example output for an MPI job run with 12 processes, on a single node, is shown below.

Comparing the execution time between the default library version of "MPI_Bcast()" and a custom implementation using only point-to-point communication...

2 : Original broadcast array[0 1 ... 99998 99999] = [0 0 ... 0 0]
2 : Initialized custom broadcast array[0 1 ... 99998 99999] = [-1 -1 ... -1 -1]
2 : Initialized MPI broadcast array[0 1 ... 99998 99999] = [-2 -2 ... -2 -2]
1 : Original broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]

Using process 1 as the root to broadcast data to all other processes...
0 : Custom broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
1 : Custom broadcast array[0 1 ... 99998 99999] = [0 0 ... 0 0]
2 : Custom broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
3 : Custom broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
4 : Custom broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
5 : Custom broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
6 : Custom broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
7 : Custom broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]

8 : Custom broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
9 : Custom broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
10 : Custom broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
11 : Custom broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]

0 : Default MPI broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
1 : Default MPI broadcast array[0 1 ... 99998 99999] = [0 0 ... 0 0]
2 : Default MPI broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
3 : Default MPI broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
4 : Default MPI broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
5 : Default MPI broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
6 : Default MPI broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
7 : Default MPI broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
8 : Default MPI broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
9 : Default MPI broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
10 : Default MPI broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]
11 : Default MPI broadcast array[0 1 ... 99998 99999] = [7.2892e+307 6.48761e+306 ... 7.79705e+307 8.53471e+307]

The custom and default broadcast implementations are equivalent.

The original broadcast array (org_Bcast_arr[]) was allocated and initialized with random double values in 0.005437 seconds.
The array was broadcasted to all other processes using "custom_Bcast()" in 0.002417 seconds.
The array was broadcasted to all other processes using "MPI_Bcast()" in 0.000636 seconds.

Number of nodes:
SLURM_JOB_NUM_NODES = 1.
Tasks per node:
SLURM_CPUS_ON_NODE = 12.