Latency and Bandwidth Timing and Methodology:

The goal of the 'ping pong' test is to measure the time cost of sending messages of various sizes between nodes in a system. The test is accomplished by having one node send a packet to another node, and then having the second node return the packet back. The time taken to accomplish this task is measured.

There are two main variables to measure in the ping pong: latency and bandwidth. The latency of a system is the time it takes between the source sending a packet and the destination receiving the packet. The bandwidth term depends on the size of the packet. That is, once the start of the packet is received, at what rate does the rest of the packet transfer.

Latency is measured using small packet sizes to avoid the bandwidth term from interfering with the measurement. It is determined by finding the y-intercept in the results of the ping pong test. The bandwidth term is determined using large packet sizes so the latency term is negligible in the measurements. Bandwidth is the inverse of the slope of the line for larger packet sizes. Equation 1 below shows the equation for bandwidth:

Equation 1: $$T_c = \frac{1}{b}$$

where b is the slope of the line and T_c is the bandwidth in (Bytes/second). Using latency T_s and bandwidth T_c, the equation for packet transmission time of n size (bytes) can we written as follows in Equation 2:

Equation 2: $$TransferTime(s) = T_S + T_C * n$$

With some additional work Equation 2 can be transformed to find the number of cycles it takes to send n bytes. The additional information needed is the a, the latency term in the cycles per second equation. To find a, it is necessary to look up instructions/second of the processor. The processors Janus use are Intel Xeon X5660 at 2.8Ghz with 2 instructions cleared per cycle. Thus the instructions cleared per second are $2.8 * 10^9 * 2 = 5.6 * 10^9 \frac{instructions}{second}$. Equation 3 below shows the equation to find a.

Equation 3: $$\frac{instructions}{second} * T_s = a$$

Finally, Substituting in Equations 1 and 3 into Equation 2 yields Equation 4:

Equation 4: CyclesToSend = a + b*n

Equation 4 is typically more useful for determining efficiency as different systems will have different clock speeds.

The ping pong test was run using MPI (openmpi) on Janus. The ping-pong exchange was run 60 times in the code. The first 10 were warmup runs and the following 50 were timed. It should be noted that the transfer time desired is half of the time measured, as the packet was sent twice.

The goal of the first test was to determine the latency and bandwidth for Janus. For this test, three different configurations were used. Each configuration was run with message sizes ranging from 1B to 4MB increasing by powers of 2. The first configuration was the single node configuration, which was to send information to different processors on the same node. The second configuration used was the leaf node configuration. In this configuration, two nodes on the same leaf switch exchanged information. The final configuration was the random node configuration. In this configuration two nodes NOT on the same leaf switch exchanged information.

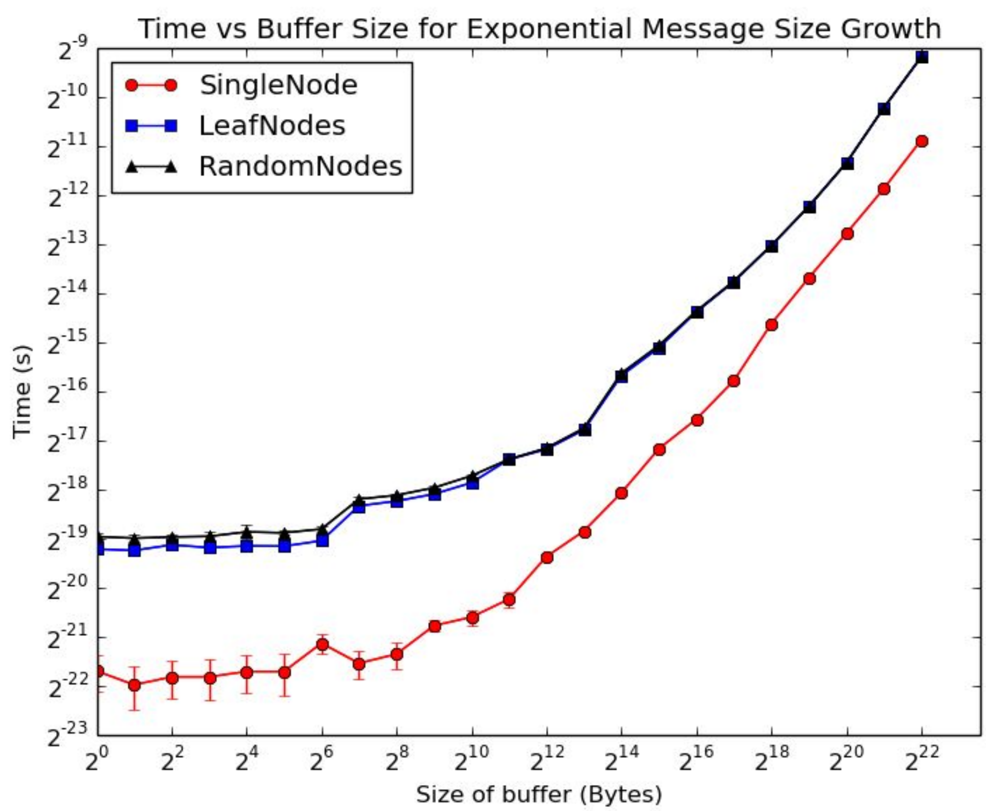

Figure 1: Time to transmit packets of various sizes using different topologies on Janus.

Figure 1 shows the results of the first test. The error bars in Figure 1 are the 95% confidence interval for that point (the 95% confidence interval of the 50 timed runs). The points themselves are the average for the 50 timed runs. As can be seen, the 95% confidence interval is fairly small after using 50 trials. The single node configuration appears to have a larger interval, but that is only because of the logarithmic scales used.

In Figure 1, it is clear that the single node configuration is by far the fastest. This makes sense as the physical distance between two processors on the same node is far less than two processors on different nodes. The leaf node and random node tests are almost identical except for a difference in latency. This is explained by the physical distance between the nodes as well as well as the random nodes passing through an additional switch. The difference in latency is better viewed in Figure 2 later on.

In Figure 1, the latency dominated area can be seen as the flat line for the first 6 points. After that there is a transition period where both the latency and bandwidth contribute. The last 8 points where the line slope is positive is the bandwidth dominated area. Using these features and the above equations the latency and bandwidth were calculated.

Table 1: Latency and Bandwidth Results on Janus for Various Topologies

	Latency (us)	Throughput (MB/s)	Alpha	Beta
Single Node (various nodes)	0.17333704	7,957.8435	970.6874	1.2566215e-10
Leaf Nodes (node0208,0209)	0.87749731	2,483.9663	4,913.9849	4.0258199e-10
Random Nodes (node0207,0610)	1.0439532	2,486.5452	5,846.1379	4.0216447e-10
OSU Micro Benchmarks 5.0 (node0432,0433)	~1.57	~3300	~8792	~3.0e-10

Table 1 shows the results from the first test as well as a the results from the official OSU Micro Benchmarks 5.0 latency and bandwidth tests. The OSU tests were run only once on leaf nodes as a sanity check, so only approximate values are given. Full output can be viewed in Figures 10 and 11 in the appendix.

As seen in Table 1, there is only a slight difference in latency between leaf and random nodes. The throughput is almost negligible. The biggest difference is between the single node test and the multiple node tests. The latency for the single node test is smaller by a factor of 5 and the throughput is higher by a factor of 3.

The OSU tests yield a higher latency but a lower throughput. The difference between the official benchmark suite the leaf node test is likely in the efficiency of implementation, not the topology. Both implementations are reasonably close for latency and throughput.

The second test was set up similar to the first except that the message sizes were different. The range was 1B to 4KB increasing linearly. The goal of this test was to try and find if MPI switches delivery protocols at a certain message size.

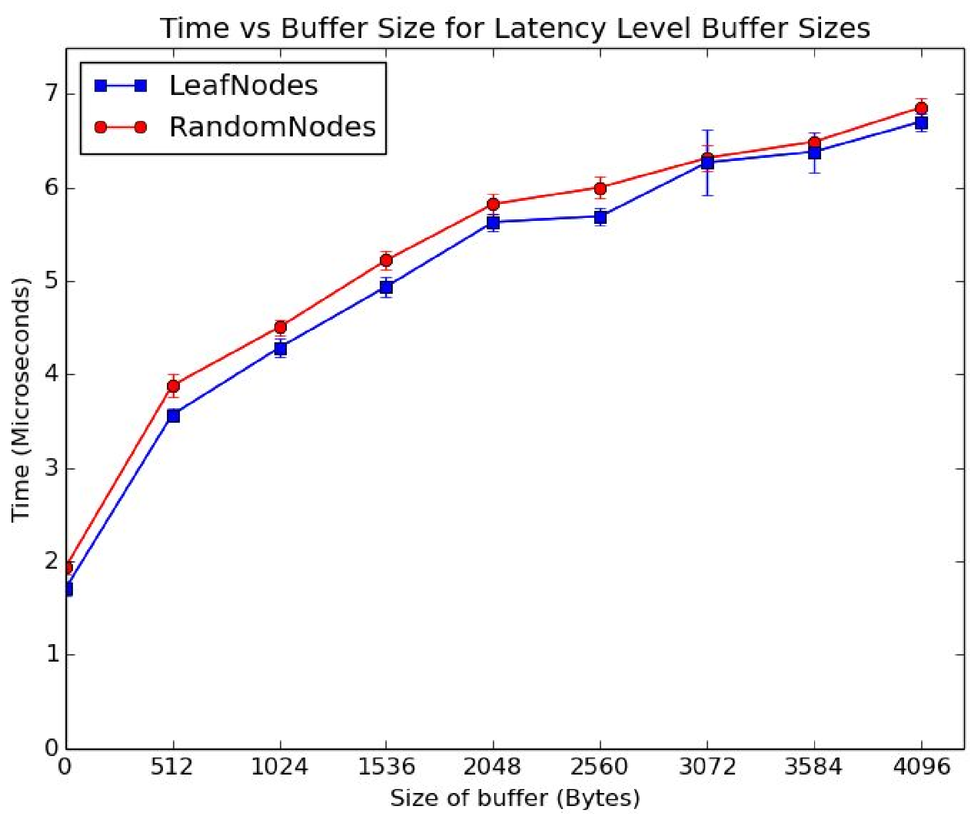

Figure 2: Time to transmit packets of ‘latency level’ sizes using different topologies on Janus.

Figure 2 shows the results of the second test. Similar to the first test, the error bars represent the 95% confidence interval. The confidence intervals are again relatively small. The one exception is the leaf node point at 3072B, which has an interval of +-0.5us. Looking at the data the values for that point have a larger range.

The latency difference of about 0.1us between leaf nodes and random nodes can be seen in Figure 2. Effectively, the difference between sending data between leaf nodes and random nodes is almost negligible. This is especially true for larger packet sizes.

Looking at Figure 2 it is unclear whether or not MPI switches protocols in the given range. If there is a switch it's between 0B and 512B where the slope of the line is much steeper than anywhere else.

Dense Linear Transpose:

The object of the dense linear transpose function is to perform a matrix transpose between processes. An example is shown below in Figure 3.

<table>
<tr><td>Process 0</td><td>Data Flow</td><td>Process 1</td></tr>
<tr><td>Matrix A
<table><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>0</td><td>1</td><td>2</td></tr><tr><td>0</td><td>1</td><td>2</td></tr></table></td><td>--------------------->
---------------------></td><td>Matrix A^T
<table><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr><tr><td>2</td><td>2</td><td>2</td></tr></table></td></tr>
<tr><td colspan="3">Figure 3: An example of a dense linear transpose between two processes.</td></tr>
</table>

This transpose is accomplished in MPI using three steps. As there is no generic datatype to store a column in C, the first step is to create a special MPI derived datatype. The derived column datatype is effectively a struct with datatypes and offsets. This allows MPI to map the data correctly after it has been sent. The second step to use MPI_Send to send the column datatype from Process 0 to Process 1, one column at a time. Finally, Process 1 uses MPI_Recv with the address of a regular matrix (with rows that have the size of the columns sent). Process 1 has to make sure to put the columns in the correct rows. For a transpose, column 1 from Process 0 becomes row 1 for Process 1, column 2 from Process 0 becomes row 2 for Process 1, etc.

Appendix: Proof of functionality, topology. OSU benchmark results. Code.

```
aaron@AaronPC: ~/Documents/MPI/HW3

aaron@AaronPC:~/Documents/MPI/HW3$ mpiexec -np 2 ./transpose.out 9
Initial matrix, world_rank = 0
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

Final matrix, world_rank = 1
0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5
6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8
aaron@AaronPC:~/Documents/MPI/HW3$ mpiexec -np 2 ./transpose.out 4
Initial matrix, world_rank = 0
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3

Final matrix, world_rank = 1
0 0 0 0
1 1 1 1
2 2 2 2
3 3 3 3
aaron@AaronPC:~/Documents/MPI/HW3$ mpiexec -np 2 ./transpose.out 2
Initial matrix, world_rank = 0
0 1
0 1

Final matrix, world_rank = 1
0 0
1 1
aaron@AaronPC:~/Documents/MPI/HW3$
```

Figure 4: Dense linear transform output for matrices of sizes 9x9, 4x4, 2x2.

JobName|Nodelist
pp_sn_1|node0274
pp_sn_2|node0312
pp_sn_4|node0379
pp_sn_8|node0947
pp_sn_16|node1210
pp_sn_32|node0829
pp_sn_64|node0830
pp_sn_128|node0831
pp_sn_256|node0207
pp_sn_512|node0208
pp_sn_1024|node0209
pp_sn_2048|node0210
pp_sn_4096|node0211
pp_sn_8192|node0212
pp_sn_16384|node0403
pp_sn_32768|node0404
pp_sn_65536|node0405
pp_sn_131072|node0406
pp_sn_262144|node0407
pp_sn_524288|node0408
pp_sn_1048576|node1343
pp_sn_2097152|node1344
pp_sn_4194304|node1345

Figure 5: Nodes used for single node ping pong performance for power of 2 message sizes.

JobName|Nodelist
pp_rn_1|node[0207,0610]
pp_rn_2|node[0207,0610]
pp_rn_4|node[0207,0610]
pp_rn_8|node[0207,0610]
pp_rn_16|node[0207,0610]
pp_rn_32|node[0207,0610]
pp_rn_64|node[0207,0610]
pp_rn_128|node[0207,0610]
pp_rn_256|node[0207,0610]
pp_rn_512|node[0207,0610]
pp_rn_1024|node[0207,0610]
pp_rn_2048|node[0207,0610]
pp_rn_4096|node[0207,0610]
pp_rn_8192|node[0207,0610]
pp_rn_16384|node[0207,0610]
pp_rn_32768|node[0207,0610]
pp_rn_65536|node[0207,0610]
pp_rn_131072|node[0207,0610]
pp_rn_262144|node[0207,0610]
pp_rn_524288|node[0207,0610]
pp_rn_1048576|node[0207,0610]
pp_rn_2097152|node[0207,0610]
pp_rn_4194304|node[0207,0610]

Figure 6: Nodes used for ping pong test with power of 2 message sizes for random nodes (nodes not on same leaf switch).

JobName|Nodelist
pp_ln_1|node[0208-0209]
pp_ln_2|node[0208-0209]
pp_ln_4|node[0208-0209]
pp_ln_8|node[0208-0209]
pp_ln_16|node[0208-0209]
pp_ln_32|node[0208-0209]
pp_ln_64|node[0208-0209]
pp_ln_128|node[0208-0209]
pp_ln_256|node[0208-0209]
pp_ln_512|node[0208-0209]
pp_ln_1024|node[0208-0209]
pp_ln_2048|node[0208-0209]
pp_ln_4096|node[0208-0209]
pp_ln_8192|node[0208-0209]
pp_ln_16384|node[0208-0209]
pp_ln_32768|node[0208-0209]
pp_ln_65536|node[0208-0209]
pp_ln_131072|node[0208-0209]
pp_ln_262144|node[0208-0209]
pp_ln_524288|node[0208-0209]
pp_ln_1048576|node[0208-0209]
pp_ln_2097152|node[0208-0209]
pp_ln_4194304|node[0208-0209]

Figure 7: Nodes used for ping pong test for power of 2 message sizes with nodes on a leaf switch.

JobName|Nodelist
pp_ln_small_1|node[0208-0209]
pp_ln_small_512|node[0208-0209]
pp_ln_small_1024|node[0208-0209]
pp_ln_small_1536|node[0208-0209]
pp_ln_small_2048|node[0208-0209]
pp_ln_small_2560|node[0208-0209]
pp_ln_small_3072|node[0208-0209]
pp_ln_small_3584|node[0208-0209]
pp_ln_small_4096|node[0208-0209]

Figure 8: Nodes used for ping pong test with small messages for leaf nodes.

JobName|Nodelist
pp_rn_small_1|node[0207,0610]
pp_rn_small_512|node[0207,0610]
pp_rn_small_1024|node[0207,0610]
pp_rn_small_1536|node[0207,0610]
pp_rn_small_2048|node[0207,0610]
pp_rn_small_2560|node[0207,0610]
pp_rn_small_3072|node[0207,0610]
pp_rn_small_3584|node[0207,0610]
pp_rn_small_4096|node[0207,0610]

Figure 9: Nodes used for ping pong test with small messages for random nodes (not on a leaf switch).

 # OSU MPI Bandwidth Test v5.0

# Size	Bandwidth (MB/s)
1	2.43
2	4.86
4	9.78
8	19.54
16	39.08
32	73.22
64	146.12
128	274.15
256	407.47
512	884.93
1024	1539.50
2048	2263.02
4096	2818.10
8192	3045.36
16384	3125.83
32768	3180.59
65536	3203.59
131072	3216.93
262144	3101.84
524288	3242.94
1048576	3321.37
2097152	3361.51
4194304	3382.09

Figure 10: osu_bw test for node0432, node0433

# OSU MPI Latency Test v5.0	
# Size	**Latency (us)**
0	1.55
1	1.58
2	1.58
4	1.58
8	1.58
16	1.57
32	1.77
64	1.77
128	1.96
256	3.12
512	3.29
1024	3.71
2048	4.83
4096	5.78
8192	7.73
16384	10.78
32768	16.02
65536	26.26
131072	47.81
262144	86.32
524288	163.37
1048576	317.57
2097152	625.74
4194304	1245.50

Figure 11: osu_latency test for node0432, node0433

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

//Aaron Holt
//HPSC
//MPI Ping Pong
//compile with mpicc hw3.1-holtat.c -o timeit.o
//run with mpiexec -np 2 ./timeit.o BufferSize

const char *argp_program_version =
    "argp-ex3 1.0";
const char *argp_program_bug_address =
    "<bug-gnu-utils@gnu.org>";

/* Program documentation. */
static char doc[] =
    "Argp example #3 -- a program with options and arguments using argp";

/* A description of the arguments we accept. */
static char args_doc[] = "BufferSize(bytes)";

/* The options we understand. */
static struct argp_option options[] = {
    {"verbose",  'v', 0,      0,  "Produce verbose output" },
    { 0 }
};

/* Used by main to communicate with parse_opt. */
struct arguments
{
    char *args[1];                /* buffer size */
    int verbose;
};

/* Parse a single option. */
static error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
    /* Get the input argument from argp_parse, which we
     know is a pointer to our arguments structure. */
    struct arguments *arguments = state->input;

    switch (key)
        {
        case 'v':
            arguments->verbose = 1;
            break;

        case ARGP_KEY_ARG:
            if (state->arg_num >= 1)
            /* Too many arguments. */
            argp_usage (state);

            arguments->args[state->arg_num] = arg;

            break;

        case ARGP_KEY_END:
            if (state->arg_num < 1)
            /* Not enough arguments. */
            argp_usage (state);
            break;

        default:
            return ARGP_ERR_UNKNOWN;
        }
    return 0;
}

/* Our argp parser. */
static struct argp argp = { options, parse_opt, args_doc, doc };

int
main (int argc, char **argv)
{
    struct arguments arguments;

    /* Parse our arguments; every option seen by parse_opt will
       be reflected in arguments. */
    argp_parse (&argp, argc, argv, 0, 0, &arguments);

    // printf ("Buffer Size (bytes) = %s\n"
    //         "VERBOSE = %s\n",
    //         arguments.args[0],
    //         arguments.verbose ? "yes" : "no");

    //buffer size from input char* to int
    int size;
    size = 1; //default
    if (sscanf (arguments.args[0], "%i", &size)!=1) {}


    //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);

    //Bail if incorrect
    if (world_size > 2)
    {
        if (world_rank == 0)
        {
            printf("World size greater than 2, exiting\n");
        }
        exit(0);
    }
    if (world_size < 2)
    {
        if (world_rank == 0)
        {
            printf("World size less than 2, exiting\n");
        }
        exit(0);
    }
```

```
    if (world_rank == 0 && arguments.verbose == 1)
    {
        printf("Buffer size (bytes) = %d\n", size);
    }

    //Timing variables
    double total_time;
    total_time = 0;
    double starttime, endtime;
    double alltime[50] = {0};

    //Dynamically allocate arrays
    char *buffer;     //buffer to send
    buffer = (char*) malloc(size*sizeof(char)+1);
    buffer[size] = '\0';

    int j = 0;
    if (world_rank == 0)
    {
        for (j=0; j<size; j++)
        {
            buffer[j] = (char)11.0;
        }
        if ( arguments.verbose == 1)
        {
            for (j=0; j<3; j++)
            {
                printf("Initial data in buffer[%d]: %d ", j, buffer[j]);
                printf("\n");
            }
        }
    }

    //Timing
    //10 warmup, 40 test
    int kk;
    for (kk=0; kk<60; kk++)
    {
        for (j=0; j<size; j++)
        {
            buffer[j] = (char)11.0;
        }

        MPI_Barrier(MPI_COMM_WORLD);

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

        //Time bcast
        if (world_size > 1)
        {
            if (world_rank == 0)
            {
                MPI_Send(buffer, size, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
                MPI_Recv(buffer, size, MPI_CHAR, 1, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
            }
            else if (world_rank == 1)
            {
                MPI_Recv(buffer, size, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
                    MPI_STATUS_IGNORE);
                MPI_Send(buffer, size, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
            }
        }
        if (kk>=10)
        {
            endtime = MPI_Wtime();
            total_time = total_time + endtime - starttime;
            alltime[kk-10] = endtime - starttime;
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);
    // printf("Data %d, world rank %d\n", buffer[0], world_rank);

    if (world_rank == 0)
    {
        int i;
        for(i=0; i<50; i++)
        {
            if (i < 49)
            {
                printf("%2.9f,", alltime[i]);
            }
            else
            {
                printf("%2.9f", alltime[i]);
            }
        }
        printf("\n%2.9f\n", total_time/100);
    }
    MPI_Barrier(MPI_COMM_WORLD);


    //Free malloc'ed data
    free (buffer);

    MPI_Finalize();
    exit (0);
}
```

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

//Aaron Holt
//HPSC
//MPI Dense Matrix Transpose
//compile with mpicc hw3.2-holtat.c -o dense_transpose.o
//run with mpiexec -np 2 ./dense_transpose.o SquareMatrixSize

const char *argp_program_version =
    "argp-ex3 1.0";
const char *argp_program_bug_address =
    "<bug-gnu-utils@gnu.org>";

/* Program documentation. */
static char doc[] =
    "Argp example #3 -- a program with options and arguments using argp";

/* A description of the arguments we accept. */
static char args_doc[] = "MatrixSize";

/* The options we understand. */
static struct argp_option options[] = {
    {"verbose",  'v', 0,      0,  "Produce verbose output" },
    { 0 }
};

/* Used by main to communicate with parse_opt. */
struct arguments
{
    char *args[1];                /* m x m */
    int verbose;
};

/* Parse a single option. */
static error_t
parse_opt (int key, char *arg, struct argp_state *state)
{
    /* Get the input argument from argp_parse, which we
     know is a pointer to our arguments structure. */
    struct arguments *arguments = state->input;

    switch (key)
        {
        case 'v':
            arguments->verbose = 1;
            break;

        case ARGP_KEY_ARG:
            if (state->arg_num >= 1)
            /* Too many arguments. */
            argp_usage (state);
            arguments->args[state->arg_num] = arg;
            break;

        case ARGP_KEY_END:
            if (state->arg_num < 1)
            /* Not enough arguments. */
            argp_usage (state);
            break;

        default:
            return ARGP_ERR_UNKNOWN;
        }
    return 0;
}

/* Our argp parser. */
static struct argp argp = { options, parse_opt, args_doc, doc };


void matrix_transpose(int m, double matrix[m][m], int from, int to, int world_rank)
{
    int i,j;

    //Create new column derived datatype
    MPI_Datatype column;
    //count, blocklength, stride, oldtype, *newtype
    MPI_Type_hvector(m, 1, m*sizeof(double), MPI_DOUBLE, &column);
    MPI_Type_commit(&column);

    //Send columns, 1 at a time
    if (world_rank == from)
    {
        for(i=0; i<m; i++)
        {
            //*data,count,type,to,tag,comm
            MPI_Send(&matrix[0][i], 1, column, to, 0, MPI_COMM_WORLD);
        }
    }
    //Receive as rows, 1 at a time
    else if (world_rank == to)
    {

        for(i=0; i<m; i++)
        {
            //*data,count,type,from,tag,comm,mpi_status
            MPI_Recv(&matrix[i][0], m, MPI_DOUBLE, from, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
        }
    }

    return;
}



int main (int argc, char **argv)
{
    struct arguments arguments;

    /* Parse our arguments; every option seen by parse_opt will
       be reflected in arguments. */
    argp_parse (&argp, argc, argv, 0, 0, &arguments);

    //matrix size, mxm
    int m;
    m = 5;
    if (sscanf (arguments.args[0], "%i", &m)!=1) {}

    //verbose?
    int verbose;
    verbose = arguments.verbose;

    // printf("m x n = %d x %d\n", m, m);

    // 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);

    int i, j, from, to;
    double matrix[m][m];

    //initialize matrices
    if (world_rank == 0)
    {
        printf("Initial matrix, world_rank = %d\n", world_rank);
        for(i=0;i<m;i++)
        {
            for(j=0;j<m;j++)
            {
                matrix[i][j] = j;
                printf("%d ", j);
            }
            printf("\n");
        }
        printf("\n");
    }
    else
    {
        for(i=0;i<m;i++)
        {
            for(j=0;j<m;j++)
            {
                matrix[i][j] = -1;
            }
        }
    }


    //Call matrix transpose function
    from = 0;
    to = 1;
    matrix_transpose(m, matrix, from, to, world_rank);

    //Print final matrix
    if (world_rank == to)
    {
       printf("Final matrix, world_rank = %d\n", world_rank);
        for (i=0; i<m; i++)
        {
            for(j=0; j<m; j++)
            {
                printf("%d ", (int)matrix[i][j]);
            }
            printf("\n");
        }
    }


    MPI_Finalize();
    exit (0);
}
```