Alpha-Beta Computer timing and Matrix Transposition

Adam Ross
adro4510@colorado.edu

Abstract

This report is written to examine and discuss how to experimentally determine the Ts, Tc, Alpha and Beta parameters of the two super computing systems Stampede and Comet using a ping ponging of messages ranging in size from 1 byte to 4 Mib. We will also be discussing some subtleties of the MPI implementation of Send and Receive and how MPI handles different message sizes. Additionally a dense matrix transposition program will be written and verified.

1. Introduction

The system architecture running your MPI code affects the performance of your algorithm as a whole, one such piece of the system being latency and bandwidth between processes. There exist many vehicles in which your MPI messages may travel such as ethernet, infiniband, or through buses locally in the system given one or more CPUs per board. As such it is valuable to determine the message latency, ts, and the overall bandwidth, tc, to once more optimize parallel code. I will first be timing and calculating tc and ts on the stampede and comet systems using messages of sizes 1 byte through 4 MiB increasing by a factor of 2. Then I will analyze the same MPI message passing parameters under a smaller scope from 1 byte to 16KiB to reveal subtleties in the MPI send and recv implementations.

2. Overview

To calculate Ts and Tc a program was written to send and receive messages between two processes in a ping pong style. Process 0 send a message of size x to process 1 and then process 1 sends the same back to process 0. In doing this multiple times we can gain understanding of the time it takes to send information of various sizes over two architecture cases; intra node communication and inter node communication. Code for sending the messages back and forth with reasonable timing accuracy is summarized below by figure 2.1.

```
for (size = 1; size <= FOUR_MB_BUFFER_SIZE; size *= 2) {
        while(error_margin > %5 && number_runs < 10) {
            if (my_rank == 0) {
                MPI_Barrier(MPI_COMM_WORLD);
                start = MPI_Wtime();
                 /* run multiple times to account for timer resolution */
                for (pass = 0; pass < max; pass++) {
                    MPI_Send(to 1);
                    MPI_Recv(from 1);
                }
                finish = MPI_Wtime();
                raw_time = (finish - start) / max;
                timing_data[n] = raw_time;
```

```
        /* calculate if we are within a decent error margin percentage */
         cont = Calc_Confidence_Interval_stop(timing_data, n, size);
         MPI_Barrier(MPI_COMM_WORLD);
         MPI_Send(&cont, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else { /* my_rank == 1 */
        MPI_Barrier(MPI_COMM_WORLD);
         /* run multiple times to account for timer resolution */
         for (pass = 0; pass < max; pass++) {
             MPI_Recv(from 0);
             MPI_Send(to 0);
        }
        /* If process 0 gets the data it needs; stop */
         MPI_Barrier(MPI_COMM_WORLD);
         MPI_Recv(&cont, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    }
    n++;
}
```

Figure 2.1 - Implementation for ping ponging

3. Verification/ Procedure

Below are the graphs generated by the code above.

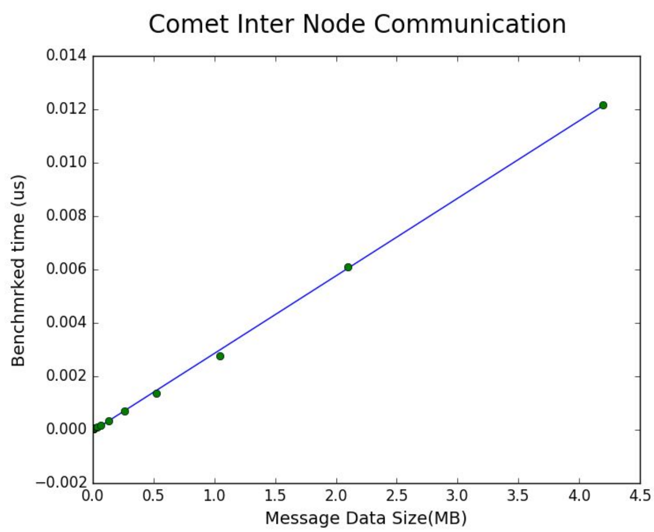

Figure 3.1 - Comet Inter Node Communication Plot

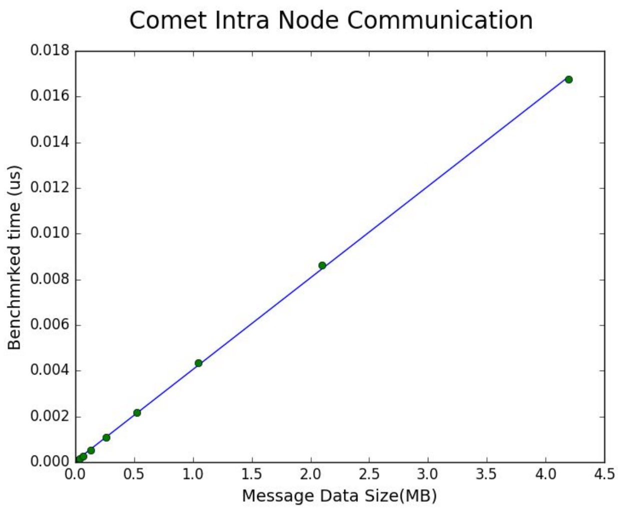

Figure 3.2 - Comet Intra Node Communication Plot

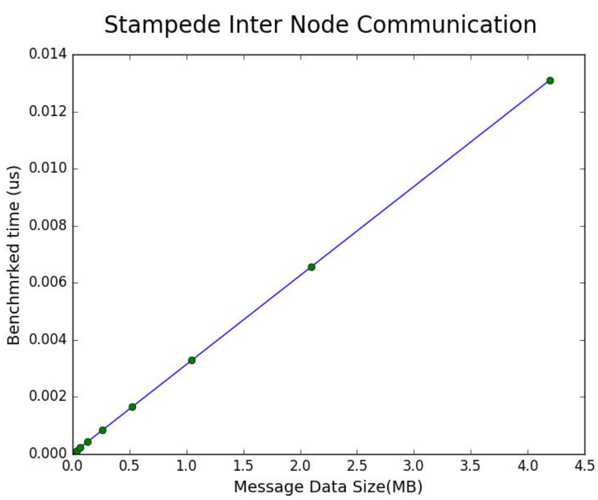

Figure 3.3 - Stampede Inter Node Communication Plot

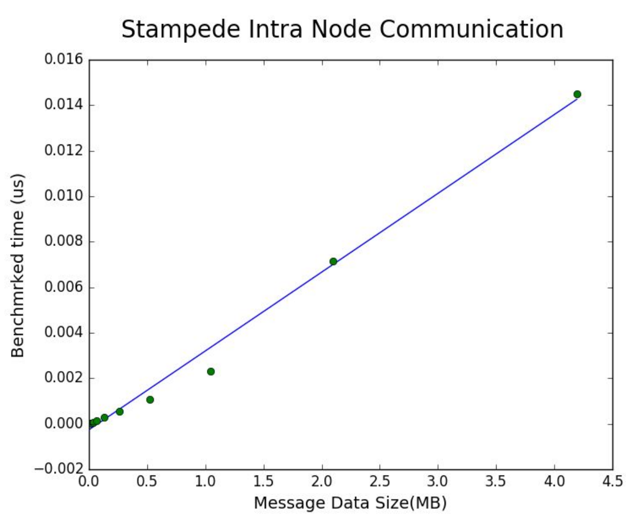

Figure 3.4 - Stampede Intra Node Communication Plot

After running the above code on two processes between two computing nodes and two processes in the same computing node we have enough data to calculate Tc, Ts, Alpha and Beta for both inter node communication and intra node communication. Ts was found simply by the time is took to send a single byte across each system. Tc was found by fitting a line to our data via least square fit, the slope of our line determined the additional time it took to send each additional byte across the various systems.

Alpha(cycles) was then calculated as the number of cycles it takes to send a single message, which was done by multiplying Ts (seconds) by the processor's clock frequency (cycles/second). Beta(cycles/byte) was then calculated by multiplying Tc (seconds/byte) by the processor's clock frequency (cycles/seconds). The processor clock frequency was determined by looking up the frequency of the Processor listed on the xsede website being Intel Xeon E5-2680 v3 for Comet and Intel Xeon E5-2680 for Stampede. These calculated numbers are represented in the Figure 2.6.

	ts (sec)	tc (sec/byte)	Alpha (cycles)	Inverse Beta (bytes/cycle)	Beta (cycles/byte)	IPC	CPU Freq (Hz	Ta (operations/sec)	Alpha (Flops)	Beta (Flops/Byte)
Comet Inter	0.00000282577289	0.000000002904314	7629.586803	0.1275242176	7.8416478	16	2700000000	43200000000	122073.3888	125.4663648
Comet Intra	0.000001166456489	0.0000000011009581	3149.43252	0.3364073259	2.97258687	16	2700000000	43200000000	50390.92032	47.56138992
Stampede Inter	0.000004315461736	0.000000003118037	10788.65434	0.1282858414	7.7950925	8	2500000000	20000000000	86309.23472	62.36074
Stampede Intra	0.000000281585298	0.000000001462788	703.963245	0.2734504248	3.65697	8	2500000000	20000000000	5631.70596	29.25576

Figure 3.5 - Comet and Stampede ts, tc, alpha and beta values.

Given the Data above we can determine the timing for sending a message(Tm(seconds) in general given by the equation:

Tm = (alpha + beta * message_size) / (CPU_Freq)

Which follows our units as:

(seconds) = ((cycles) + ((cycles/byte) * (bytes)) / (cycles/second)

We had also looked at small message latency and bandwidth. To do this we repeat the same process above from set our sample data size to increase linearly from 1 Byte to 16 KiB. In the figure 3.6 I have chose not to plot a line due to the discontinuity of the plot.

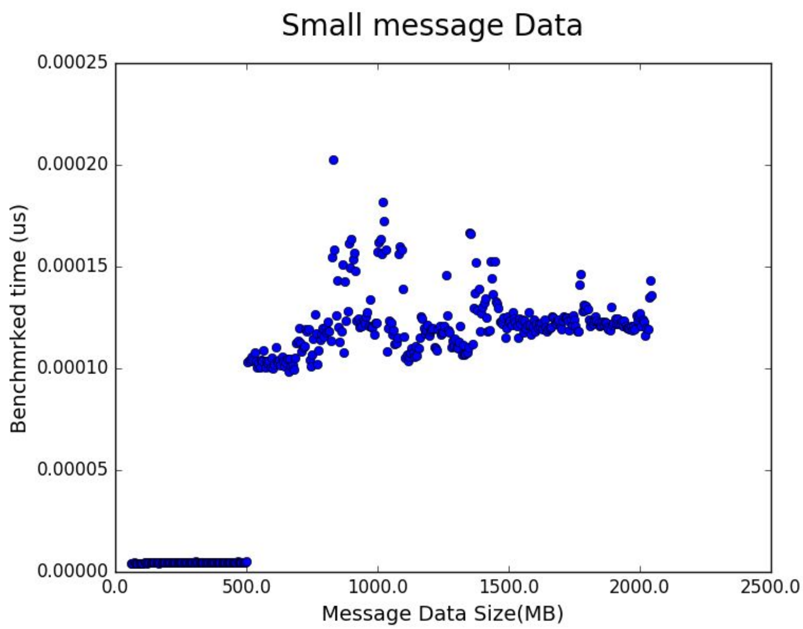

Figure 3.6 - Small message timings from 1 Byte to 16KiB

This discontinuity points to a shift in the MPI send/recv transportation protocol right around 512 Bytes. In my data the actual number of large timing change happened at 506 Bytes. I would believe this is due to message headers or additional information being passed such that the total of the message sent is 512 Bytes.

Next I shall briefly mention the correctness of my MPI matrix transposition program. Below is a summary of the code and a sample of the input and output which should speak to the correctness.

```
data = (int **)malloc(N * sizeof(int*));
    data[0] = malloc(N * N * sizeof(int));
    for (i = 1; i < N; i++) {
        data[i] = data[0] + (i * N);
    }

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            data[i][j] = i*N + j;
        }
    }

    if(my_rank == 0) {
        print_matrix(data);
    }

    MPI_Type_vector(N, 1, N, MPI_INT, &column);
    MPI_Type_commit(&column);
```

```
    for (i = 0; i < N; i++) {
        if (my_rank == 0) {
            MPI_Send(&data[0][i], 1, column, 1, 0, comm);
        } else { /* my_rank == 1 */
            MPI_Recv(&data[i][0], N, MPI_INT, 0, 0, comm, &status);
        }
    }

    if(my_rank == 1) {
        printf("\n");
        print_matrix(data);
    }
```

Figure 3.7 - Code sample from my Matrix transpose MPI program

```
mpiexec -np 2 RossAdam_transpose_HW5
Start - Rank: 0
0 1 2 3 4 5
6 7 8 9 10 11
12 13 14 15 16 17
18 19 20 21 22 23
24 25 26 27 28 29
30 31 32 33 34 35

End - Rank: 1
0 6 12 18 24 30
1 7 13 19 25 31
2 8 14 20 26 32
3 9 15 21 27 33
4 10 16 22 28 34
5 11 17 23 29 35
```

Figure 3.8 - Output from my MPI transpose program

4. Conclusion

The data above, when taken into account for optimization, be useful for deciding what message sizes to send to optimize code. For example you would not want to send a 530 Byte message in lieu of the doubling of message timing around 506 Bytes, Ideally would would design your program to not sit right above that increased timing discontinuity.

It would also be beneficial to design your node topography to try and send most messages intra node, given the alpha and beta parameters of intra node communication are smaller, and aggregate before sending it inter node if possible without adding extra calculation steps.

```
/* HW5 Alpha-Beta
 *
 *
 * Name: Adam Ross
 *
 * Input: none
 * Output: Data size, Timing and confidence information
 *
 *
 *
 *
 */

#include <stdio.h>
#include <unistd.h>
#include <math.h>
#include <stdlib.h>
#include "mpi.h"

#define MAX 10
#define FOUR_MB_BUFFER_SIZE 4194304

int Calc_Confidence_Interval_stop(double timing_data[10], int n, int size);

main(int argc, char* argv[]) {
    int         p;
    int         my_rank;
    double      *size_buffer;
    int         size;
    int         pass;
    MPI_Status  status;
    double      start, finish;
    double      raw_time;
    double      timing_data[10];
    int         max =               186;
    int         n =                 0;
    int         cont =              1;
    char        hostname[30];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    size_buffer = (double *)calloc(FOUR_MB_BUFFER_SIZE, sizeof(double));

    if (my_rank == 0) {
        printf("MPI timer resolution: %1.20f\n", MPI_Wtick());
    }

    // Print Host name to verify we are on different nodes
    // gethostname(hostname, 15);
    // printf("My rank: %d\t%s\n", my_rank, hostname);

    // MPI wamup before actual timings
    if (my_rank == 0) {
        MPI_Send(size_buffer, FOUR_MB_BUFFER_SIZE, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
        MPI_Recv(size_buffer, FOUR_MB_BUFFER_SIZE, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &status
);
    } else {
        MPI_Recv(size_buffer, FOUR_MB_BUFFER_SIZE, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status
);
        MPI_Send(size_buffer, FOUR_MB_BUFFER_SIZE, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    }

    // Iterate for each desired data size
    for (size = 1; size <= FOUR_MB_BUFFER_SIZE; size *= 2) {
        while(cont) {
            if (my_rank == 0) {
                // Barrier to kind of sync our timing
                MPI_Barrier(MPI_COMM_WORLD);
                // start timing
                start = MPI_Wtime();

                // Run the ping pong multiple times to get resolution away from the tick res
olution
                for (pass = 0; pass < max; pass++) {
                    MPI_Send(size_buffer, size, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
                    MPI_Recv(size_buffer, size, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &status);
                }
                // Finish timing
                finish = MPI_Wtime();
                raw_time = (finish - start) / max;
                timing_data[n] = raw_time;

                // Calculate the confidence error percentage to determine if we need to run
more iterations
                cont = Calc_Confidence_Interval_stop(timing_data, n, size);

                // Tell process 1
                MPI_Barrier(MPI_COMM_WORLD);
                MPI_Send(&cont, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
            } else { /* my_rank == 1 */
                // Barrier to kind of sync our timing
                MPI_Barrier(MPI_COMM_WORLD);

                // Run the ping pong multiple times to get resolution away from the tick res
olution
                for (pass = 0; pass < max; pass++) {
                    MPI_Recv(size_buffer, size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);
                    MPI_Send(size_buffer, size, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
                }
                MPI_Barrier(MPI_COMM_WORLD);
                MPI_Recv(&cont, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
            }
            n++;
        }
        // Do not need to run as many iterations for larger message sizes
        max -= 8;
        cont = 1;
        n = 0;
    }

    MPI_Finalize();
}  /* main */


/* Helper function calculate the confidence interval, error margins and determine
 * if we should keep looping.
 * Returns 1 or 0 for conintue or stop.
*/
int Calc_Confidence_Interval_stop(double timing_data[10], int n, int size) {
    double      sum =               0.0;
    double      mean =              0.0;
    double      std_dev =           0.0;
    double      marg_err =          0.0;
    double      marg_perc =         100.0;
```

```
    int         i;

    if (n > 2) {
        for (i = 0; i < n; i++) {
            sum += timing_data[i];
        }
        mean = sum / n;
        sum = 0.0;
        for (i = 0; i < n; i++) {
            sum += pow(timing_data[i] - mean, 2);
        }
        std_dev = sqrt(sum / n);
        marg_err = 1.96 * (std_dev / sqrt(n));
        marg_perc = (marg_err / mean) * 100;
    } else {
        return 1;
    }
    if (marg_perc > 5.0  && n < 20) {
        return 1;
    } else {
        printf("%d\t%1.20f\t%1.10f\t%1.10f\t%f\t%d\n", size, mean, std_dev, marg_err, marg_p
erc, n);
        return 0;
    }
}
```

```c
/* HW5 Alpha-Beta, Dense Matrix Transpose
 *
 *
 * Name: Adam Ross
 *
 * Input: none
 * Output: Byte timing data for 1 Byte - 16 KiB
 * on intervals of 5 bytes
 *
 *
 *
 */

#include <stdio.h>
#include <unistd.h>
#include <math.h>
#include <stdlib.h>
#include "mpi.h"

#define MAX 10
#define SIXTEEN_KB_BUFFER_SIZE 2048

int Calc_Confidence_Interval_stop(double timing_data[10], int n, int size);

main(int argc, char* argv[]) {
    int         p;
    int         my_rank;
    double      *size_buffer;
    int         size;
    int         pass;
    MPI_Status  status;
    double      start, finish;
    double      raw_time;
    double      timing_data[10];
    MPI_Comm    comm;
    int         max =               128;
    int         n =                 0;
    int         cont =              1;
    char        hostname[30];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_dup(MPI_COMM_WORLD, &comm);

    size_buffer = (double *)calloc(SIXTEEN_KB_BUFFER_SIZE, sizeof(double));

    if (my_rank == 0) {
        printf("MPI timer resolution: %1.20f\n", MPI_Wtick());
    }

    gethostname(hostname, 15);
    printf("My rank: %d\t%s\n", my_rank, hostname);

    // MPI wamup before actual timings
    if (my_rank == 0) {
        MPI_Send(size_buffer, SIXTEEN_KB_BUFFER_SIZE, MPI_DOUBLE, 1, 0, comm);
        MPI_Recv(size_buffer, SIXTEEN_KB_BUFFER_SIZE, MPI_DOUBLE, 1, 0, comm, &status);
    } else {
        MPI_Recv(size_buffer, SIXTEEN_KB_BUFFER_SIZE, MPI_DOUBLE, 0, 0, comm, &status);
        MPI_Send(size_buffer, SIXTEEN_KB_BUFFER_SIZE, MPI_DOUBLE, 0, 0, comm);
    }
    // Iterate for each desired data size
    for (size = 1; size <= SIXTEEN_KB_BUFFER_SIZE; size+= 5) {
        while(cont) {
            if (my_rank == 0) {
                // Barrier to kind of sync our timing
                MPI_Barrier(comm);
                // start timing
                start = MPI_Wtime();

                // Run the ping pong multiple times to get resolution away from the tick res
olution
                for (pass = 0; pass < max; pass++) {
                    MPI_Send(size_buffer, size, MPI_DOUBLE, 1, 0, comm);
                    MPI_Recv(size_buffer, size, MPI_DOUBLE, 1, 0, comm, &status);
                }
                // Finish timing
                finish = MPI_Wtime();
                raw_time = (finish - start) / max;

                // Store in array for confidence analysis
                timing_data[n] = raw_time;

                // Calculate the confidence error percentage to determine if we need to run
more iterations
                cont = Calc_Confidence_Interval_stop(timing_data, n, size);
                MPI_Barrier(comm);
                // Tell process 1
                MPI_Send(&cont, 1, MPI_INT, 1, 0, comm);
            } else { /* my_rank == 1 */
                // Barrier to kind of sync our timing
                MPI_Barrier(comm);

                // Run the ping pong multiple times to get resolution away from the tick res
olution
                for (pass = 0; pass < max; pass++) {
                    MPI_Recv(size_buffer, size, MPI_DOUBLE, 0, 0, comm, &status);
                    MPI_Send(size_buffer, size, MPI_DOUBLE, 0, 0, comm);
                }
                MPI_Barrier(comm);
                MPI_Recv(&cont, 1, MPI_INT, 0, 0, comm, &status);
            }
            n++;
        }
        // Do not need to run as many iterations for larger message sizes
        if (size % 65 == 0) {
            max -= 4;
            if (my_rank == 0) {
                printf("%d\n", max);
            }
        }
        cont = 1;
        n = 0;
    }

    MPI_Finalize();
}  /* main */


/* Helper function calculate the confidence interval, error margins and determine
 * if we should keep looping.
 * Returns 1 or 0 for conintue or stop.
*/
int Calc_Confidence_Interval_stop(double timing_data[10], int n, int size) {
```

```
    double      sum =               0.0;
    double      mean =              0.0;
    double      std_dev =           0.0;
    double      marg_err =          0.0;
    double      marg_perc =         100.0;
    int         i;

    if (n > 2) {
        for (i = 0; i < n; i++) {
            sum += timing_data[i];
        }
        mean = sum / n;
        sum = 0.0;
        for (i = 0; i < n; i++) {
            sum += pow(timing_data[i] - mean, 2);
        }
        std_dev = sqrt(sum / n);
        marg_err = 1.96 * (std_dev / sqrt(n));
        marg_perc = (marg_err / mean) * 100;
    } else {
        return 1;
    }
    if (marg_perc > 5.0  && n < 20) {
        return 1;
    } else {
        printf("%d\t%1.20f\t%1.10f\t%1.10f\t%f\t%d\n", size, mean, std_dev, marg_err, marg_p
erc, n);
        return 0;
    }
}
```

```c
/* HW5 Dense Matrix Transpose
 *
 *
 * Name: Adam Ross
 *
 * Input: none
 * Output: Printed Matricies to show correctness
 *
 *
 *
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <getopt.h>
#include <string.h>
#include "mpi.h"

#define MAX 10
#define BUFSIZE 20
#define FOUR_MB_BUFFER_SIZE 4194304
#define N 6

void print_matrix(int **matrix);

void print_usage() {
    printf("Usage: -f file containging an nxn dense matrix sperated by spaces.\n");
}

main(int argc, char* argv[]) {
    int                 option = 0;
    char                buf[BUFSIZE + 1];
    int                 p;
    int                 my_rank;
    MPI_Status          status;
    MPI_Comm            comm;
    int                 i;
    int                 j;
    int                 **data;

    MPI_Datatype        column;
    double              dense_matrix[6][6] = {
        {1, 2, 3, 4, 5, 6},
        {7, 8, 9, 10, 11, 12},
        {13, 14, 15, 16, 17, 18},
        {19, 20, 21, 22, 23, 24},
        {25, 26, 27, 28, 29, 30},
        {31, 32, 33, 34, 35, 36}
    };

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_dup(MPI_COMM_WORLD, &comm);

    // Malloc our 2d array
    data = (int **)malloc(N * sizeof(int*));
    data[0] = malloc(N * N * sizeof(int));
    for (i = 1; i < N; i++) {
        data[i] = data[0] + (i * N);
    }

    // initalize to 0-N^2
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            data[i][j] = i*N + j;
        }
    }

    // Print the initial array store in rank 0
    if(my_rank == 0) {
        printf("Start. My rank: %d", my_rank);
        print_matrix(data);
    }

    // Build MPI datatype vector of every Nth item - i.e. a oclumn
    MPI_Type_vector(N, 1, N, MPI_INT, &column);
    MPI_Type_commit(&column);

    // Send each column to rank 1
    for (i = 0; i < N; i++) {
        if (my_rank == 0) {
            MPI_Send(&data[0][i], 1, column, 1, 0, comm);
        } else { /* my_rank == 1 */
            MPI_Recv(&data[i][0], N, MPI_INT, 0, 0, comm, &status);
        }
    }

    // Print the end trans formed result
    if(my_rank == 1) {
        printf("\nEnd. My rank: %d", my_rank);
        print_matrix(data);
    }

    MPI_Finalize();
}  /* main */

/* Helped method to print the whol matrix */
void print_matrix(int **matrix) {
    int i;
    int j;

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}
```

```
#!/usr/bin/python

# Adam Ross - tstc.py
#
# A helper function to aggregate and plot the data from Stampede and Comet

import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
import numpy as np
import glob

name_map = {
    "new_data/comet_inter_data" : "Comet Inter Node Communication",
    "new_data/comet_intra_data" : "Comet Intra Node Communication",
    "new_data/stampede_inter_data" : "Stampede Inter Node Communication",
    "new_data/stampede_intra_data" : "Stampede Intra Node Communication",
    "new_data/small_message" : "Small message Data"
}

for data_file in glob.glob("new_data/*"):
    byte_size = []
    timing = []

    with open(data_file) as f:
        content = f.readlines()
    content = [x.strip() for x in content]


    for point in content:
        data = point.split("\t")
        byte_size.append(float(data[0]))
        timing.append(float(data[1]))

    # Ts is the time to send a single byte which is the first element in our data
    ts = timing[0]

    byte_size = byte_size[12:]
    timing = timing[12:]

    # Least square fit data
    n = len(byte_size)
    stdevx = np.std(byte_size)
    stdevy = np.std(timing)
    sumx = sum(byte_size)
    sumy = sum(timing)
    sumxy = sum([byte_size[i] * timing[i] for i in range(n)])
    sumx2 = sum([x ** 2 for x in byte_size])
    # Tc is the slope of our lease square
    tc = ((n*sumxy) - (sumx*sumy)) / ((n*sumx2) - (sumx**2))
    b = (sumy - (tc*sumx)) / n

    print "file ", data_file, "     ts: ", '{0:.15f}'.format(ts), "      tc: ", '{0:.15f}'.f
ormat(tc)

    fig = plt.figure()
    fig.suptitle(name_map[data_file], fontsize=20)
    ax = fig.add_subplot(1,1,1)

    x = [2**x for x in range(len(byte_size))]
    y = [(b + tc*a) for a in x ]

    # For our normal data nromalize the x and y axis
    if data_file != "new_data/small_message":
        x = [a/1000000.0 for a in x]
        byte_size = [a/1000000.0 for a in byte_size]
        ax.plot(x, y)
        ax.plot(byte_size, timing, "o")
        ax.xaxis.set_major_formatter(mtick.FormatStrFormatter('%1.1f'))
    else:
        #ax.plot(x, y)
        ax.plot(byte_size, timing, "o")
        #ax.set_xticks(np.arange(0, 2048, 256))
        ax.xaxis.set_major_formatter(mtick.FormatStrFormatter('%1.1f'))



    legend = ax.legend(loc='upper center', shadow=True)
    #ax.set_xscale("log", nonposy='clip')
    #ax.set_yscale("log", nonposy='clip')
    plt.xlabel('Message Data Size(MB)', fontsize=14)
    plt.ylabel('Benchmrked time (us)', fontsize=14)
    plt.show()
```