CSCI 455: Lab #4 — MPI Timing Model and Comparison

Darwin Jacob Groskleg

Winter 2020

Contents

Summary Lab 4 consists of a series of tests to measure the timing of a single send/recv communication using the ping-pong method. Based on the tests, you can estimate the t_startup and t_data with the least square regression method.

Part 1: Estimating Communication Latency of MPI Send/Recv

```
        tags
 ⚙darwingroskleg@starbuck  ~/Dropbox/Documents/Terms/2020-01 - Winter/CSC
I455/Lab4-Timing Model Comparison  make run1
cc -std=c99 -Wall -Wextra -g -D_GLIBCXX_DEBUG -O0   -c -o slr.o slr.c
mpicc -std=c99 -Wall -Wextra -g -D_GLIBCXX_DEBUG -O0  -lmpi  mpi_latency.c
slr.o   -o mpi_latency
Platform: Darwin (4 cpu cores recognized)
PMIX_MCA_gds=hash mpirun --host localhost --mca btl_vader_backing_directory
 /tmp --mca btl ^tcp --oversubscribe  -np 4 ./mpi_latency
INFO: Number of processes = 4
INFO: Only executing 2 tasks - extra cluster processes will be ignored
task 1 has started...
task 0 has started...
Beginning latency timing test:
        Number of reps = 10
        Data Size      = 10
****************************************
Rep#       T0          T1         deltaT
   1        0.00       52.00       52.00
   2       60.00       86.00       26.00
   3       92.00      104.00       12.00
   4      109.00      112.00        3.00
   5      116.00      119.00        3.00
   6      123.00      125.00        2.00
   7      129.00      132.00        3.00
   8      135.00      137.00        2.00
   9      141.00      143.00        2.00
  10      148.00      150.00        2.00
****************************************
*** Avg round trip time = 10.700000 microseconds
*** Avg one way latency = 5.350000 microseconds
Beginning latency timing test:
        Number of reps = 10
        Data Size      = 50
****************************************
```

Figure 1: First part of output

```
Beginning latency timing test:
        Number of reps = 10
        Data Size      = 5000
****************************************
Rep#        T0          T1          deltaT
   1     1104.00     1451.00      347.00
   2     1458.00     1473.00       15.00
   3     1483.00     1494.00       11.00
   4     1498.00     1505.00        7.00
   5     1509.00     1517.00        8.00
   6     1521.00     1529.00        8.00
   7     1533.00     1540.00        7.00
   8     1544.00     1552.00        8.00
   9     1555.00     1563.00        8.00
  10     1567.00     1575.00        8.00
****************************************
*** Avg round trip time = 42.700000 microseconds
*** Avg one way latency = 21.350000 microseconds
SUMMARY STATISTICS/ESTIMATIONS:
t_comm(1) = 5.060000 microseconds
t_startup = 5.060000 microseconds
t_data    = -0.000000 microseconds
 darwingroskleg@starbuck  ~/Dropbox/Documents/Terms/2020-01 - Winter/CSC
 darwingroskleg@starbuck  ~/Dropbox/Documents/Terms/2020-01 - Winter/CSC
I455/Lab4-Timing Model Comparison
```

Figure 2: Last part of output

mpi_latency.c

```c
/*****************************************************************************
* FILE: mpi_latency.c
* AUTHORS:
*   Darwin Jacob Groskleg (2020)
*   Sazzad (02/11/18)
*   Laurence T. Yang
* DESCRIPTION:
*   MPI Latency Timing Program - C Version
*   In this example code, a MPI communication timing test is performed.
*   MPI process 0 will send "reps" number of 1 byte messages to MPI process 1,
*   waiting for a reply between each rep. Before and after timings are made
*   for each rep and an average calculated when completed.
*
* NOTES
*   - Ping-pong method sends same size data back and forth.
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include <stdbool.h>
#include <math.h>
#include <assert.h>

#include <sys/time.h>
#include <mpi.h>

#include "slr.h"

#define NUMBER_REPS 10
#define DATA_SIZE 5000

/* send/receive process designators, maps task label to expected rank */
enum TaskRanks {
    Master = 0,
    Worker = 1
};

double sample_latency_with_n_bytes(int bytes_of_traffic);
void run_sampling_responder(int bytes_of_traffic);

/*
 * Estimate the t_startup and t_data with the least square regression method:
 *  y: AvgT/2 (one way latency)
 *  let K = 10 (number of runs, different message sizes)
 * t_startup = m * 0 + b            (time to send msg with no data)
 * t_comm = m*1 + b                 (time to startup and send 1 data word)
 * t_data = t_comm - t_startup   (time to send one data word)
 */
int main (int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int cluster_size;           /* number of MPI processes */
    MPI_Comm_size(MPI_COMM_WORLD, &cluster_size);
    int rank;                   /* my MPI process number */
```

```c
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == Master && cluster_size != 2) {
        printf("INFO: Number of processes = %d\n", cluster_size);
        printf("INFO: Only executing 2 tasks - extra cluster processes ");
        printf("will be ignored\n");
    }
    MPI_Barrier(MPI_COMM_WORLD);

    if (rank < 2) {
        printf("task %d has started...\n", rank);
    }
    MPI_Barrier(MPI_COMM_WORLD);

    int x_byte_counts[] = {10, 50, 100, 200, 500, 1000, 2000, 3000, 4000, 5000};
    size_t k_tests = sizeof(x_byte_counts)/sizeof(x_byte_counts[0]);
    double y_timings[k_tests];

    for (size_t i=0; i<k_tests; i++){
        if (rank == Worker) {
            run_sampling_responder(x_byte_counts[i]);
        }
        else if (rank == Master) {
            y_timings[i] = sample_latency_with_n_bytes(x_byte_counts[i]);
        }
    }

    if (rank == Master) {
        slr_equation_t eqt = slr_find_line(k_tests, x_byte_counts, y_timings);
        double t_startup = slr_predict(eqt, 0);
        double t_comm    = slr_predict(eqt, 1);
        double t_data    = t_comm - t_startup;
        printf("SUMMARY STATISTICS/ESTIMATIONS:\n");
        printf("t_comm(1) = %f microseconds\n", t_comm);
        printf("t_startup = %f microseconds\n", t_startup);
        printf("t_data    = %f microseconds\n", t_data);
    }

    MPI_Finalize();
    exit(0);
}

/* Returns: time in microseconds (t_comm)
 *      - the one-way latency (round-trip / 2)
 * Takes:
 *  bytes_of_traffic = N, the amount of data to be sent
 */
double sample_latency_with_n_bytes(int bytes_of_traffic) {
    assert(bytes_of_traffic <= DATA_SIZE);

    char msg[DATA_SIZE];        /* buffer containing DATA_SIZE byte message */
    MPI_Status status;          /* MPI receive routine parameter */
    int tag = 1;                /* MPI message tag parameter */
    int reps = NUMBER_REPS;     /* number of samples per test */
    /* round-trip latency timing test */
    printf("Beginning latency timing test:\n");
```

```c
    printf("\tNumber of reps = %d\n", reps);
    printf("\tData Size      = %d\n", bytes_of_traffic);
    printf("****************************************\n");
    /*      <  >  <      . >  <      . >  <      . >                     */
    printf("Rep#       T0           T1          deltaT\n");

    const double MSPerSecond = pow(10, 6);
    const double ClockResolution = 1;//MPI_Wtick(); // seconds per clock tick
    double T0, T1;              /* start/end times per rep in ms */
    double deltaT;              /* time for one rep in ms */
    double sumT = 0;            /* sum of all reps times in microseconds */
    int error;
    for (int n = 1; n <= reps; n++) {
        /* start time */
        T0 = MPI_Wtime() * MSPerSecond * ClockResolution;

        /* send message to worker - message tag set to 1.  */
        error = MPI_Send(
                &msg,
                bytes_of_traffic,
                MPI_BYTE,   // for char
                Worker,     // destination
                tag,
                MPI_COMM_WORLD
            );
        /* If return code indicates error quit */
        // DARWIN GROSKLEG:
        //  this is unecessary and already properly handled
        //  by the MPI error handler, which will abort the MPI job in most
        //  cases.
        if (error)
            MPI_Abort(MPI_COMM_WORLD, error);

        /* Now wait to receive the echo reply from the worker  */
        // "echoes" the same data back???
        error = MPI_Recv(
            &msg,
            bytes_of_traffic,
            MPI_BYTE,   // for char
            Worker,     // source
            tag,
            MPI_COMM_WORLD,
            &status);

        /* If return code indicates error quit */
        // Redundant step is skipped because the MPI already handles it.
        if (error)
            MPI_Abort(MPI_COMM_WORLD, error);

        /* end time */
        T1 = MPI_Wtime() * MSPerSecond * ClockResolution;

        /* calculate round trip time and print */
        deltaT = T1 - T0;
        sumT += deltaT;
        /* print statement for each to keep each column right justified */
```

```
        printf("%4d   ", n);
        printf("%10.2f   ", T0);
        printf("%10.2f   ", T1);
        printf("%10.2f\n", deltaT);
    }

    /* average time per rep in microseconds */
    double avgT = sumT / reps;
    printf("*****************************************\n");
    printf("*** Avg round trip time = %f microseconds\n", avgT);
    printf("*** Avg one way latency = %f microseconds\n", avgT/2);

    return avgT/2;
}

void run_sampling_responder(int bytes_of_traffic) {
    assert(bytes_of_traffic <= DATA_SIZE);
    char msg[DATA_SIZE];         /* buffer containing DATA_SIZE byte message */
    MPI_Status status;           /* MPI receive routine parameter */
    int tag = 1;                 /* MPI message tag parameter */
    int reps = NUMBER_REPS;      /* number of samples per test */

    while (reps--) {
        // ping
        MPI_Recv(
            &msg,
            bytes_of_traffic,
            MPI_BYTE,
            Master,
            tag,
            MPI_COMM_WORLD,
            &status);
        // pong
        MPI_Send(
            &msg,
            bytes_of_traffic,
            MPI_BYTE,
            Master,
            tag,
            MPI_COMM_WORLD);
    }
}
```

slr.h

```c
/* slr.h
 * -----
 * Authors: Darwin Jacob Groskleg (2020)
 *
 * Purpose: do simple linear regression.
 */
#ifndef SLR_H_INCLUDED
#define SLR_H_INCLUDED

typedef struct {
    double a; /* intercept */
    double b; /* slope */
} slr_equation_t;

slr_equation_t slr_find_line(int n, int X[], double Y[]);

double slr_predict(slr_equation_t eqt, int x);

#endif /* SLR_H_INCLUDED */
```

slr.c

```c
/* slr.c
 * -----
 * Authors: Darwin Jacob Groskleg (2020)
 *
 * Purpose: do simple linear regression.
 */
#include "slr.h"

#include <assert.h>
#include <math.h>

double regression_coefficient(int, double, double, double, double);
double y_intercept(int, double, double, double);


double slr_predict(slr_equation_t eqt, int x) {
    /*  intercept  +  slope * x    */
    return  eqt.a  +  eqt.b * x;
}

/* let K = n
 * for each (x, y) of K:
 *       x^2, xy
 *   calc slope:
 *       m =  (K * SUM(xy) - SUM(x)*SUM(y))
 *          / (K * SUM(x^2) - SUM(x)^2)
 *   calc intercept:
 *       b = (SUM(y) - m*SUM(x))/K
 *
 * passes as pointer, NOT COPY
 */
slr_equation_t slr_find_line(int n, int X[], double Y[]) {
    //assert() len(X) == len(Y) == n
    slr_equation_t eqt;
    double sigma_x  = 0;
    double sigma_xx = 0;
    double sigma_y  = 0;
    double sigma_xy = 0;
    for (int i=0; i<n; i++) {
        sigma_x  += X[i];
        sigma_xx += pow(X[i], 2);
        sigma_y  += Y[i];
        sigma_xy += X[i] * Y[i];
    }
    double slope = regression_coefficient(n, sigma_x, sigma_xx,
                                                    sigma_y, sigma_xy);
    eqt.a = y_intercept(n, slope, sigma_x, sigma_y);
    eqt.b = slope;
    return eqt;
}

/* Slope aka "Regression Coefficient"
 *
 * Biostatistical Analysis 5ed, JH Zar, Pages 330-337
 *  Yi = a + BXi for best fit using least squares linear regression.
```

```
 *
 */
double regression_coefficient(
        int    data_points,
        double sum_of_x,        // SUM x
        double sum_sqr_of_x,    // SUM xx
        double sum_of_y,        // SUM y
        double sum_of_xy)       // SUM xy
{
    double sum_of_cross_products = sum_of_xy - sum_of_x * sum_of_y/data_points;
    double sum_of_squares_x = sum_sqr_of_x - pow(sum_sqr_of_x, 2)/data_points;
    return sum_of_cross_products / sum_of_squares_x;
}

double y_intercept(
        int    data_points,
        double slope,
        double sum_of_x,
        double sum_of_y)
{
    return (sum_of_y - slope * sum_of_x) / data_points;
}
```

Part 2: The Efficient Processor Scaling of MPI_Bcast Over Send/Recv

Program Outputs and Results

compare_bcast.c

```
//
// Comparison of MPI_Bcast with the my_bcast function
//
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#include <mpi.h>

/* send/receive process designators, maps task label to expected rank */
enum TaskRanks {
    Master = 0,
    Worker = 1
};

void my_bcast(void* data, int count, MPI_Datatype datatype, int root,
              MPI_Comm communicator)
{
    int world_rank;
    MPI_Comm_rank(communicator, &world_rank);
    int world_size;
    MPI_Comm_size(communicator, &world_size);

    if (world_rank == root) {
        // If we are the root process, send our data to everyone


    } else {
        // If we are a receiver process, receive the data from the root


    }
}

int main(int argc, char** argv) {
    if (argc != 3) {
        fprintf(stderr, "Usage: compare_bcast num_elements num_trials\n");
        exit(1);
    }

    int num_elements = atoi(argv[1]);
    int num_trials = atoi(argv[2]);

    MPI_Init(NULL, NULL);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    double total_my_bcast_time = 0.0;
    double total_mpi_bcast_time = 0.0;
    int i;
    int* data = (int*)malloc(sizeof(int) * num_elements);
    assert(data != NULL);

    for (i = 0; i < num_trials; i++) {
```

```c
        // Time my_bcast
        // Synchronize before starting timing



        // Synchronize again before obtaining final time



        // Time MPI_Bcast



    }

    // Print off timing information
    if (world_rank == 0) {
        printf("Data size = %d, Trials = %d\n", num_elements * (int)sizeof(int),
            num_trials);
        printf("Avg my_bcast time = %lf\n", total_my_bcast_time / num_trials);
        printf("Avg MPI_Bcast time = %lf\n", total_mpi_bcast_time / num_trials);
    }

    free(data);
    MPI_Finalize();
    return 0;
}
```

hiding_latency.c

```
/* hiding_latency.c
 * ----------------
 * Authors: Darwin Jacob Groskleg
 * Date:    Saturday, May 16, 2020
 *
 * Taken from "Intro to Parallel Computing (2018), Roman Trobec et al.",
 *  Page 120
 *
 * Overlapping communication and computation
 */
#include <mpi.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
int i; double a;
for (i = 0; i < 100000000/numproc; i++) {
a = sin(sqrt(i)); //different amount of calculation return a;
main(int argc, char* argv[]) //number of processes must be > 1
int p, i, myid, tag=1, proc, ierr;
double start_p, run_time, start_c, comm_t, start_w, work_t, work_r; double *buff = nullptr;
MPI_Request request; MPI_Status status;
MPI_Init(&argc, &argv);
start_p = MPI_Wtime(); MPI_Comm_rank(MPI_COMM_WORLD, &myid); MPI_Comm_size(MPI_COMM_WORLD, &p);
#define master 0
#define MSGSIZE 100000000 //5000000 //different sizes of ←
messages
buff = (double*)malloc(MSGSIZE * sizeof(double)); //allocate
if (myid == master) {
for (i = 0; i < MSGSIZE; i++) { //initialize message
buff[i] = 1;
}
start_c = MPI_Wtime();
for (proc = 1; proc<p; proc++) {
MPI_Irecv(buff, MSGSIZE,
MPI_DOUBLE, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &←
      1
2
3
4 5{ 6
   double other_work(int numproc)
   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 37 38 39
40 41 42 43
44 45 46 47 48 49 50 51
}
int {
                            #if 1
 //non-blocking receive
  request); #endif
  #if 0
); #endif
}
```

```
MPI_Recv(buff, MSGSIZE, //blocking receive
MPI_DOUBLE, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status↩
     }
comm_t = MPI_Wtime() - start_c; start_w = MPI_Wtime();
work_r = other_work(p);
work_t = MPI_Wtime() - start_w; MPI_Wait(&request, &status);
//block until Irecv is done

```