Oliver Hoffman, Faiyam Rahman

CS 5460: Parallel and Distributed Lab 1

02/23/15

**Part 3**

When we run Hello World on the AWS Server with n processes, the sentence “Hello, I am *y* of *n* processes” is printed for every *y* from 0 to n-1. The interesting part is the order in which the sentences are printed. Specifically, the order is random. This is probably because the length of time it takes for a process to finish is dependent on what else the processor its running on has to deal with, and that appears random to us.

As we vary the number of processes from 4 all the way up to 100, the pattern is more or less consistent. The only thing to note is that since our AWS Server has 8 actual processors on which to do work, when we run the file with more than 8 processes, we actually expected that the first 8 would finish in random order, then the next 8 would finish in random order, etc. What we found, however, is that while lower number processes tend to finish earlier, the overall order is still mostly random. This is probably because a processor does not wait for other processors to finish their work before picking up another process.

**Part 4**

Here we present a profile of our AWS Server consisting of its message latency, its message bandwidth as a function of packet size, and its total floating point operations per second as a function of the number of processors employed.

*Latency:* ~350-520 nanoseconds. The precise number varied every time we ran the code.

*Bandwidth:* The bandwidth linearly increased to around 8100 Megabytes/second, when the packet size was around 30,000 doubles. It then decreased linearly all the way to 4500 Megabytes/second with a packet size of 990,000 doubles. We suspect this is because messages have a size limit of around 30,000 doubles. When we increased past that size, sending the payload required multiple messages, which increased overhead.

*Floating Point Operations Per Second:* The total number of FLOPs increased linearly as we varied the number of processors employed. We did not push past 8 processors since we believe our AWS server only has 8 virtual CPUs.

**Explanation for *latency.c***

See the end of this report for the code.

After initializing the MPI library, retrieving the global size and retrieving the rank of each process, we first make sure that we have enough processes to calculate the latency (2 processes minimum). After that, we set up a SENDER process and a RECEIVER process.

SENDER Process

For a given number of tests (which we set by a macro), it:

1. Logs the time
2. Sends the time to the receiver
3. Waits for an acknowledgement before starting over

RECEIVER Process

The Receiver initializes a message\_latency variable to 0, and then for a given number of tests (which we set by a macro), it:

1. Receives the message from SENDER

2. Logs the time at which it received it

3. Calculates the time elapsed between sending and receiving

4. Adds the elapsed time to the latency variable

5. Sends an acknowledgement message to the SENDER

At the end, the receiver divides the total time elapsed by the number of tests, and

then prints the value to standard output. This is the latency.

**Explanation for *bandwidth.c***

See the end of this report for the code.

After initializing the MPI library, retrieving the global size and retrieving the rank of each process, we first make sure that we have enough processes to calculate the bandwidth (2 processes minimum). After that, we set up a SENDER process and a RECEIVER process.

SENDER process

For a given packet size (number of doubles), it:

1. For a given number of tests (which we set by a macro) it:
   1. It creates a message with the given number of doubles, initializing each double to 0.0
   2. It logs the time it sends the message, and saves it to the 0th index of the message it is about to send
   3. Sends the message to the receiver
   4. Waits for acknowledgement from the receiver before starting over

RECEIVER process

It initializes a results array of duples, where each duple will be (packet size,

average time to send such a packet).

For a given packet size (number of doubles), it:

1. For a given number of tests (which we set by a macro), it:
   1. It initializes a message\_latency variable to 0.0
   2. It then calculates the time it takes to send a packet of the given size as follows:
      1. Receives the message from SENDER
      2. Logs the time at which it received it
      3. Calculates the time elapsed between sending and receiving
      4. Adds the elapsed time to the latency variable
      5. Sends an acknowledgement message to the SENDER
2. It calculates the average time to send a message with the given packet size by dividing the total time elapsed to send all messages (message\_latency) by the number of messages sent (NUM\_MSGS)
3. It updates the results array with a (packet size, average time to send such a packet) duple.

After all the tests have been run for each packet size, it iterates over the results array and prints out (packet size, bandwidth) duples. Note that the bandwidth is calculated by a function called megabytesPerSecond.

Both the SENDER and RECEIVER increase the packet size from 1000 to (1000 \* 100) in steps of 1000 in our specific implementation. However, the packet size step and number of tests (in this case, 100), can be varied via macros.

**Explanation for *flops.c***

See the end of this report for the code.

We initialize the MPI library, retrieve the global size and retrieve the rank of each process. At a high level we set up master-worker architecture, in which the master initializes the vectors for which the workers will calculate a dot product and then splits that array into chunks that are sent to the workers. The workers calculate the dot product for their part and send it back to the master, which then adds up all the results.

In the case that we run the code on just 1 process, the code runs slightly differently. The master does not chunk up the work or send any messages. Instead, it simply initializes the arrays, calculates the dot product, and returns the FLOPS.

To actually calculate the FLOPS, it logs the time at which it sends the first work message and then once it receives the last result message it again logs the time. In the 1 processor case, the start and end times are logged before and after the dot product calculation. We subtract the former from the latter to get the elapsed time in seconds. We calculate the total floating point operations per second as follows:

FLOPS = (size of vector \* 2) / elapsed time

We multiply by 2 because for each element in the arrays we do one multiplication and one addition.

The master then prints the total FLOPS to the standard output.

*Latency.c*

|  |
| --- |
| #include <stdio.h> |
|  | #include <mpi.h> |
|  |  |
|  | #ifndef DEBUG |
|  | #define debug\_print(M, ...) |
|  | #else |
|  | #define debug\_print(M, ...) fprintf(stderr, M , ##\_\_VA\_ARGS\_\_) |
|  | #endif |
|  |  |
|  | #define NUM\_TESTS 10 // num iterations of test |
|  | #define SEND\_ID 0 // process sending message |
|  | #define RECV\_ID 1 // process receiving message |
|  | #define TAG\_TEST 4 // Indicates this message is being timed |
|  | #define TAG\_ACK 5 // Indicates this message is to acknowledge receipt of a test message |
|  |  |
|  |  |
|  | int main (int argc, char \*\*argv) |
|  | { |
|  | // Initialization of parameters |
|  | int sz, myid; |
|  | MPI\_Init (&argc, &argv); |
|  | MPI\_Comm\_size (MPI\_COMM\_WORLD, &sz); |
|  | MPI\_Comm\_rank (MPI\_COMM\_WORLD, &myid); |
|  |  |
|  |  |
|  | // Program begins for each process |
|  | if(sz < 2) |
|  | { |
|  | printf("Too few processes to run latency testing. Exiting program.\n"); |
|  | MPI\_Finalize (); |
|  | return 0; |
|  | } |
|  |  |
|  | if(myid == SEND\_ID) |
|  | { |
|  | int i; |
|  |  |
|  | for(i = 0; i < NUM\_TESTS; ++i) |
|  | { |
|  | // Log start time of message send |
|  | double current\_time = MPI\_Wtime(); |
|  |  |
|  | // Send test message |
|  | MPI\_Send(&current\_time, 1, MPI\_DOUBLE, RECV\_ID, TAG\_TEST, MPI\_COMM\_WORLD); |
|  | debug\_print("SENDER: Message %d sent - %f.\n", i, current\_time); |
|  |  |
|  | // Receieve acknowledgement of successful test message |
|  | int acknowledgement; |
|  | MPI\_Status status; |
|  | MPI\_Recv(&acknowledgement, 1, MPI\_INT, RECV\_ID, TAG\_ACK, MPI\_COMM\_WORLD, &status); |
|  | } |
|  |  |
|  | debug\_print("SENDER: Execution finished.\n"); |
|  | } |
|  | else if(myid == RECV\_ID) |
|  | { |
|  | int i; |
|  | double message\_latency = 0.0; |
|  |  |
|  | for(i = 0; i < NUM\_TESTS; ++i) |
|  | { |
|  |  |
|  | // Receieve message |
|  | double send\_time; |
|  | MPI\_Status status; |
|  | MPI\_Recv(&send\_time, 1, MPI\_DOUBLE, SEND\_ID, TAG\_TEST, MPI\_COMM\_WORLD, &status); |
|  |  |
|  | debug\_print("RECEIVER: Message %d received - %f.\n", i, send\_time); |
|  |  |
|  | // Compare how much time the message took between send and receive time |
|  | double receive\_time = MPI\_Wtime(); |
|  | double result\_time = receive\_time - send\_time; |
|  | debug\_print("RECEIVER: Time elapsed - %f seconds.\n", result\_time); |
|  | message\_latency += result\_time; |
|  |  |
|  | // Send acknowledgement to sender |
|  | int acknowledgement = 1; |
|  | MPI\_Send(&acknowledgement, 1, MPI\_INT, SEND\_ID, TAG\_ACK, MPI\_COMM\_WORLD); |
|  | debug\_print("RECEIVER: Acknowledgement sent.\n"); |
|  |  |
|  | } |
|  |  |
|  | debug\_print("RECEIVER: Final message latency - %f.\n", message\_latency/NUM\_TESTS); |
|  | debug\_print("RECEIVER: Execution finished.\n"); |
|  |  |
|  | #ifndef DEBUG |
|  | printf("Final message latency - %f.\n", message\_latency/NUM\_TESTS); |
|  | #endif |
|  | } |
|  |  |
|  | MPI\_Finalize (); |
|  | return 0; |
|  | } |

*Bandwidth.c*

|  |
| --- |
| #include <stdio.h> |
|  | #include <stdlib.h> |
|  | #include <mpi.h> |
|  |  |
|  | #ifndef DEBUG |
|  | #define debug\_print(M, ...) |
|  | #else |
|  | #define debug\_print(M, ...) fprintf(stderr, M , ##\_\_VA\_ARGS\_\_) |
|  | #endif |
|  |  |
|  | #define NUM\_TESTS 100 // num iterations of test |
|  | #define PACKET\_STEP 1000 // step size of tests (i.e 5 packets, then 10 packets, etc) |
|  | #define NUM\_MSGS 10 // num messages to send for every iteration of the test |
|  | #define SEND\_ID 0 // process sending message |
|  | #define RECV\_ID 1 // process receiving message |
|  | #define TAG\_TEST 4 // Indicates this message is being timed |
|  | #define TAG\_ACK 5 // Indicates this message is to acknowledge receipt of a test message |
|  |  |
|  | void print\_array(double \*array, int size, char \*array\_name) { |
|  | int i; |
|  | for(i = 0; i < size; i++) { |
|  | printf("%s elem %d: %f\n", array\_name, i, array[i]); |
|  | } |
|  | } |
|  |  |
|  | // Given an array [packet\_size in doubles, average time to send the packet], |
|  | // outputs bandwidth in megabytes/second |
|  | double megabytesPerSecond(double \*array) { |
|  | double num\_doubles = array[0]; |
|  | double ave\_time = array[1]; |
|  | double result = ((num\_doubles \* sizeof(double)) / 1000000) / ave\_time; |
|  | return result; |
|  | } |
|  |  |
|  | int main (int argc, char \*\*argv) |
|  | { |
|  | // Initialization of parameters |
|  | int sz, myid; |
|  | MPI\_Init (&argc, &argv); |
|  | MPI\_Comm\_size (MPI\_COMM\_WORLD, &sz); |
|  | MPI\_Comm\_rank (MPI\_COMM\_WORLD, &myid); |
|  |  |
|  |  |
|  | // Program begins for each process |
|  | if(sz < 2) |
|  | { |
|  | printf("Too few processes to run bandwidth testing. Exiting program.\n"); |
|  | MPI\_Finalize (); |
|  | return 0; |
|  | } |
|  |  |
|  | if(myid == SEND\_ID) |
|  | { |
|  | int packet\_size; |
|  |  |
|  | // for(packet\_size = PACKET\_STEP; packet\_size <= PACKET\_STEP \* NUM\_TESTS; packet\_size += PACKET\_STEP) |
|  | for(packet\_size = PACKET\_STEP \* NUM\_TESTS; packet\_size >= PACKET\_STEP; packet\_size -= PACKET\_STEP) |
|  | { |
|  | int i; |
|  | for(i = 0; i < NUM\_MSGS; ++i) |
|  | { |
|  |  |
|  | // create test message |
|  | double \*time\_array = (double \*)malloc(packet\_size \* sizeof(double)); |
|  | if (!time\_array) { |
|  | printf("Failed to allocate time\_array on processor %d", myid); |
|  | MPI\_Finalize(); |
|  | return 0; |
|  | } |
|  | int j; |
|  | for(j = 0; j < packet\_size; j++) { |
|  | time\_array[j] = 0.0; |
|  | } |
|  |  |
|  | // Log start time of message send |
|  | time\_array[0] = MPI\_Wtime(); |
|  |  |
|  | // Send test message |
|  | MPI\_Send(time\_array, packet\_size, MPI\_DOUBLE, RECV\_ID, TAG\_TEST, MPI\_COMM\_WORLD); |
|  | debug\_print("SENDER: Message %d sent - %f.\n", i, current\_time); |
|  |  |
|  | // clean heap |
|  | free(time\_array); |
|  |  |
|  | // Receieve acknowledgement of successful test message |
|  | int acknowledgement; |
|  | MPI\_Status status; |
|  | MPI\_Recv(&acknowledgement, 1, MPI\_INT, RECV\_ID, TAG\_ACK, MPI\_COMM\_WORLD, &status); |
|  | } |
|  | } |
|  |  |
|  |  |
|  | debug\_print("SENDER: Execution finished.\n"); |
|  | } |
|  | else if(myid == RECV\_ID) |
|  | { |
|  | double results\_array[NUM\_TESTS][2]; |
|  | int packet\_size; |
|  |  |
|  | // for(packet\_size = PACKET\_STEP; packet\_size <= PACKET\_STEP \* NUM\_TESTS; packet\_size += PACKET\_STEP) |
|  | for(packet\_size = PACKET\_STEP \* NUM\_TESTS; packet\_size >= PACKET\_STEP; packet\_size -= PACKET\_STEP) |
|  | { |
|  | int i; |
|  | double message\_latency = 0.0; |
|  |  |
|  | // Calculate latency for this packet size |
|  | for(i = 0; i < NUM\_MSGS; ++i) |
|  | { |
|  | // Receive message |
|  | double \*send\_time\_array = (double \*)malloc(packet\_size \* sizeof(double)); |
|  | if (!send\_time\_array) { |
|  | printf("Failed to allocate send\_time\_array on processor %d", myid); |
|  | MPI\_Finalize(); |
|  | return 0; |
|  | } |
|  | MPI\_Status status; |
|  | MPI\_Recv(send\_time\_array, packet\_size, MPI\_DOUBLE, SEND\_ID, TAG\_TEST, MPI\_COMM\_WORLD, &status); |
|  |  |
|  | // Check that we got the right thing |
|  | debug\_print("RECEIVER: Message %d received - %f.\n", i, send\_time\_array[0]); |
|  | #ifdef DEBUG |
|  | print\_array(send\_time\_array, packet\_size, "send\_time\_array"); |
|  | #endif |
|  | // Compare how much time the message took between send and receive time |
|  | double receive\_time = MPI\_Wtime(); |
|  | double result\_time = receive\_time - send\_time\_array[0]; |
|  | debug\_print("RECEIVER: Time elapsed - %f seconds.\n", result\_time); |
|  | message\_latency += result\_time; |
|  | free(send\_time\_array); |
|  |  |
|  | // Send acknowledgement to sender |
|  | int acknowledgement = 1; |
|  | MPI\_Send(&acknowledgement, 1, MPI\_INT, SEND\_ID, TAG\_ACK, MPI\_COMM\_WORLD); |
|  | debug\_print("RECEIVER: Acknowledgement sent.\n"); |
|  | } |
|  |  |
|  | debug\_print(" Packet Size, Message latency: %d, %f\n", packet\_size, message\_latency/NUM\_MSGS); |
|  |  |
|  | // Update results\_array |
|  | int num\_iter = (packet\_size / PACKET\_STEP) - 1; |
|  | results\_array[num\_iter][0] = packet\_size; |
|  | results\_array[num\_iter][1] = message\_latency/NUM\_MSGS; |
|  |  |
|  | debug\_print("RECEIVER: Final message latency for packet size=%d - %f.\n", |
|  | packet\_size, message\_latency/NUM\_MSGS); |
|  | } |
|  |  |
|  | #ifndef DEBUG |
|  | printf("Packet size, Message Latency\n"); |
|  | int i; |
|  | for(i = 0; i < NUM\_TESTS; i++) |
|  | { |
|  | printf("%d, %f\n", (int)results\_array[i][0],megabytesPerSecond(results\_array[i])); |
|  | } |
|  | #endif |
|  |  |
|  | debug\_print("RECEIVER: Execution finished.\n"); |
|  | } |
|  |  |
|  | MPI\_Finalize (); |
|  | return 0; |
|  | } |

*flops.c*

|  |
| --- |
|  |
| #include <stdio.h> |
|  | #include <stdlib.h> |
|  | #include <mpi.h> |
|  |  |
|  | #ifndef DEBUG |
|  | #define debug\_print(M, ...) |
|  | #else |
|  | #define debug\_print(M, ...) fprintf(stderr, M , ##\_\_VA\_ARGS\_\_) |
|  | #endif |
|  |  |
|  | #define MASTER 0 // Master process ID |
|  | #define ARRAY\_SIZE 1000000 // Size of arrays for dot product |
|  | #define WORK\_MSG\_1 1 // Array to be sent |
|  | #define WORK\_MSG\_2 2 // Array to be sent |
|  | #define RESULT\_MSG 3 // The result of a dot product calculation |
|  |  |
|  | double dotProduct(double \*a1, double \*a2, unsigned long long length) { |
|  | unsigned long long i; |
|  | double result = 0.0; |
|  | for(i = 0; i < length; i++) |
|  | { |
|  | result += (a1[i] \* a2[i]); |
|  | } |
|  |  |
|  | return result; |
|  | } |
|  |  |
|  | int main (int argc, char \*\*argv) |
|  | { |
|  | // Initialization of parameters |
|  | int sz, myid; |
|  | MPI\_Init (&argc, &argv); |
|  | MPI\_Comm\_size (MPI\_COMM\_WORLD, &sz); |
|  | MPI\_Comm\_rank (MPI\_COMM\_WORLD, &myid); |
|  |  |
|  | // master process sets up calculation |
|  | if(myid == MASTER) |
|  | { |
|  | // Initalize vectors |
|  | double \*array1 = (double \*)malloc(ARRAY\_SIZE \* sizeof(double)); |
|  | double \*array2 = (double \*)malloc(ARRAY\_SIZE \* sizeof(double)); |
|  | if (!array1 || !array2) |
|  | { |
|  | printf("Error: Array1 or Array2 failed to allocate with ARRAY\_SIZE %d\n", ARRAY\_SIZE); |
|  | MPI\_Finalize(); |
|  | return 0; |
|  | } |
|  | unsigned long long i; |
|  | for(i = 0; i < ARRAY\_SIZE; ++i) |
|  | { |
|  | array1[i] = 1.0; |
|  | array2[i] = 1.0; |
|  | } |
|  |  |
|  | // If there's only one process, have the master do the work |
|  | if (sz == 1) |
|  | { |
|  | printf("Executing dot product on the master. only 1 process!\n"); |
|  | double start\_time = MPI\_Wtime(); |
|  | double result = dotProduct(array1, array2, ARRAY\_SIZE); |
|  | double end\_time = MPI\_Wtime(); |
|  | double FLOPS = (ARRAY\_SIZE \* 2) / (end\_time - start\_time); |
|  | printf("FLOPS: %f\n", FLOPS); |
|  |  |
|  | MPI\_Finalize(); |
|  | return 0; |
|  | } |
|  |  |
|  | // Send each process the chunk they are going to calculate |
|  | int process; |
|  | for(process = 1; process < sz; process++) |
|  | { |
|  | int num\_elems = ARRAY\_SIZE / (sz - 1); |
|  | int offset = num\_elems \* (process - 1); |
|  |  |
|  | MPI\_Send(array1 + offset, num\_elems, MPI\_DOUBLE, process, |
|  | WORK\_MSG\_1, MPI\_COMM\_WORLD); |
|  | MPI\_Send(array2 + offset, num\_elems, MPI\_DOUBLE, process, |
|  | WORK\_MSG\_2, MPI\_COMM\_WORLD); |
|  | } |
|  |  |
|  | // Get the start time |
|  | double start\_time = MPI\_Wtime(); |
|  |  |
|  | // Receive the results from each process |
|  | double \*results = (double \*)malloc((sz - 1) \* sizeof(double)); |
|  | MPI\_Status status; |
|  | for(process = 1; process < sz; process++) |
|  | { |
|  | int offset = process - 1; |
|  | MPI\_Recv(results + offset, 1, MPI\_DOUBLE, process, |
|  | RESULT\_MSG, MPI\_COMM\_WORLD, &status); |
|  | } |
|  |  |
|  | // Calculate FLOPS |
|  | double end\_time = MPI\_Wtime(); |
|  | double flops = (ARRAY\_SIZE \* 2) / (end\_time - start\_time); |
|  |  |
|  | // Add up all the results |
|  | int j; |
|  | double result = 0.0; |
|  | for(j = 0; j < (sz - 1); j++) |
|  | { |
|  | result += results[j]; |
|  | } |
|  | debug\_print("Result of dot product: %f\n", result); |
|  | free(results); |
|  |  |
|  | // print results |
|  | printf("FLOPS: %f\n", flops); |
|  | } |
|  |  |
|  | // workers do calculation |
|  | else { |
|  |  |
|  | // get the work |
|  | int work\_length = ARRAY\_SIZE / (sz - 1); |
|  | double \*work\_array\_1 = (double \*)malloc(work\_length \* sizeof(double)); |
|  | double \*work\_array\_2 = (double \*)malloc(work\_length \* sizeof(double)); |
|  | if (!work\_array\_1 || !work\_array\_2) |
|  | { |
|  | printf("A Work array failed to allocate on process %d\n", myid); |
|  | MPI\_Finalize(); |
|  | return 0; |
|  | } |
|  | MPI\_Status status1, status2; |
|  | MPI\_Recv(work\_array\_1, work\_length, MPI\_DOUBLE, MASTER, WORK\_MSG\_1, |
|  | MPI\_COMM\_WORLD, &status1); |
|  | MPI\_Recv(work\_array\_2, work\_length, MPI\_DOUBLE, MASTER, WORK\_MSG\_2, |
|  | MPI\_COMM\_WORLD, &status2); |
|  |  |
|  | // do the work |
|  | double result = dotProduct(work\_array\_1, work\_array\_2, work\_length); |
|  | MPI\_Send(&result, 1, MPI\_DOUBLE, MASTER, RESULT\_MSG, MPI\_COMM\_WORLD); |
|  | free(work\_array\_1); |
|  | free(work\_array\_2); |
|  | } |
|  |  |
|  | MPI\_Finalize (); |
|  | return 0; |
|  |  |