Lab 4: Distributed Parallel Computing with DHCP (Group 9's Perspective)

Group 9 – In collaboration with Group 10
Obadah Abozaid, Phoenix Daak, Aaron Phan

CSE 525

Abstract

The purpose of the project is to perform a demonstration of the concept called distributed parallel computing. In today's computing world, it is important to design ways to compute and maintain an excellent runtime within several calculations. An important takeaway about distributed parallel computing is that it alleviates workload and performs fast execution times with the use of a collection of autonomous computers that is connected to a network. In this project, distributed parallel programming will be created using the DHCP method. The experiment requires two Raspberry Pi (RPi) boards where one is the controller and the other being the worker. Although two are used to showcase this idea, there can be several workers executing the same program that can drastically lessen computational time. The goal is for the controller to execute processes to each worker that is connected in the same network and return the data to the controller. The worker will execute specific instructions based on the number of cores and return when finished. If successful, the controller was able to handle the returning data from all the workers and output them onto the controller's terminal.

Body

Updating the PI

Downloading the Raspberry Pi Imager with version 1.8.1 or higher allowed for the 64-bit version of Raspberry Pi OS Bookworm to be installed onto each PI board. Once the operating system was installed onto the PI, respected names were assigned to each Pi whether they were a worker node or the controller. For Group 9, the board's name was assigned to "controller9" acting as the controller in Lab 4. After the operating system was installed and the profiles were setup, updates for the boards were installed.

Running the following command: *sudo apt update,* updated the package information based on the most recent available packages. This ensures that the current packages installed in the operating system align with the rest of the Raspberry Pi's used as workers. Once the package information was updated, the package updates were installed to the systems. Running the following command: *sudo apt dist-upgrade* installed the updated packages to the system. The last command in the command segment, *dist-upgrade* ensured that the most updated packages for Raspberry Pi OS were installed. This command can remove old versions and replace with newly updated packages. Having the most updated packages installed onto the system the Raspberry Pi's were ready for the Message Passing Interface (MPI) setup.

MPICH Initialization

Message passing interfacing required a series of package installs and updates during the initialization in which both Raspberry Pi's followed. Running *sudo apt install mpich* installed all packages relevant to message passing interfacing. This included the compiler and necessary tools used to execute parallel programs between multiple systems which is known as the "mpiexec binaries". For this Lab, a distributed file storage system was not implemented. However, this would make it possible for the controller and all of the worker nodes to share a common file system, so files modified on one system does not need to be transferred to the other.

After the MPICH packages were installed, the Pi's were rebooted and the network for the boards were configured. The Pi's utilized a local WiFi network which allowed for IP addresses to easily be assigned to each board based on whether it was a controller or worker. This was made easy because of the known IP address range beginning with 192.168.1. Once the boards were connected the network, MPICH was ready to communicate through the IP addresses.

Configuring Controller and Worker

Since every Raspberry Pi went through the same initialization with MPICH and package updates, everything was on the same page and the boards were configured as controller and worker nodes. This process needed keys to be generated by the controller node. For this, *ssh-keygen -t rsa* was ran to generate respective keys for its worker nodes. To send the generated keys to the worker nodes, SSH was ran to copy the ID of the key to the worker node. This allowed for the MPICH to execute on the worker nodes without the use of password for the key coming from the controller node. This step was repeated for as many worker nodes there were, in this case, only once.

After distributing controller keys to the worker nodes, the controller node needed to know where its worker nodes are and how many resources it had to work with. First, the "hostname" of

the system was verified to read, “controller”. After the hostname was modified, the “hosts” file was modified to indicate that the current system was in fact a controller node. After the “controller” tag in the hosts file, the list of the worker nodes’ IP addresses was written to the file. Completing this, the file was saved and exited to then initialize the worker nodes in a similar but opposite fashion. Once the controller was initialized with all its worker nodes, the worker nodes needed information about who they were and who their controller was. For this the “hostname” of the worker node was modified ensuring that it read the name of the worker node. The name of the worker node for “controller9” was “worker10”. Once the “hostname” file was updated to correlate with the respected worker Pi, its needed to know who its controller was. For this, the “hosts” file was modified like the controller. Instead of listing the worker nodes IP addresses, the address of the controller node was listed, and the file was saved and exited.

Configuring the controller nodes to the worker nodes and vice versa, the nodes had were initialized to communicate with each other. The cores of the processors need to be configured now. For this, the “mpihost” file located in the controller node was modified to indicate the number of cores the processor could use. Both the controller and worker nodes had four cores to were ready to be with eight total cores for processes to be broken up into.

Dividing Work

Distributing work among the different cores between the controller and worker nodes meant for faster run times with processes. Some initial steps were taken to allow for this distribution. First, exact copies of the source code had to be placed in the same positions for each node. This made sure that when the controller called onto the worker nodes, they would have also the correct source code to complete the work it was assigned with. Once the source code was copied, it was compiled on the controller and the program was ran.

There are different functions within the source code in which indicates when a process should be forked out onto the different worker nodes. The first function which is responsible for forking process onto the worker nodes was “MPI_Scatter”. This function split up the data of the parent function and sent blocks of the data array to each worker node that the controller held. Once the individual worker nodes finished up their assigned work, the nodes waited until the other nodes were complete. After the completion of the processes, the second key function “MPI_Gather” was called. This function was responsible for gathering the data returned by the worker nodes. It rejoined the data that was originally scatter by the controller node back into the data array where the final solution could be interpreted. Another key function used in the source code was “MPI_bcast”. With “MPI_bcast” same data could be sent out to all the worker and controller nodes. This is important when dealing with data that is being modified. Having the most up to date data to work with ensured that each worker node was working with valid data and was seen to hold when analyzing the final data array gathered from the nodes. With the use of these key functions, processes were distributed to each individual cores of both the controller and worker nodes improving the overall runtime of a process.

Lab 4 Exercise 1

This part of the experiment involves outputting a set of prime numbers while utilizing MPI’s broadcast function that will allow the master process (the controller) to create load for the

workers to execute and process the numbers. Then, they are returned with MPI's reduce function that will combine the primes founded by workers and return to the controller that is outputted into the terminal.

Initializing the experiment requires the use of a file (under software code) called 'mpi_prime.c'. It is assumed that both the controller and the workers have obtained a copy of the code. 'mpi_prime.c' shows the setup of parallel computing with the use of the broadcast function and gathering the processes in preparation for computing all prime numbers of a given range. In this scenario, the tested number range was between 1 and 262144. Getting the number of processors requires calling 'MPI_Comm_size' and the rank of each process with 'MPI_Comm_rank'. The use of a constant called 'MPI_COMM_WORLD' is used to get the number of processes.

After recognizing the processes, a function called 'MPI_Bcast' is used to start the workers to execute another function called 'prime_number' that will determine if a given number is prime. 'MPI_Bcast' uses a standard collective communication technique called broadcasting where a process sends the same to other processes. In the scope of the project, the controller broadcasts specific data to connected workers that will start 'prime_number'. After determining a number as a prime, the code resumes and calls 'MPI_Reduce'. This is a function that contains a reduction calculation with a given operation that will handle the data which is returned to the master process.

After finishing the execution and bringing all primes into the master process (controller), the terminal will output all the prime numbers based on the given range within the program. Each prime number that has been found by the processes are given a timestamp that benchmarks the elapsed time from the start to the end of the execution. Depending on the number of processes, the runtime can radically change. For example, the use of eight cores gave a runtime execution of 44.474649 seconds when the range of finding primes was from 1 to 524288. However, changing the number of cores needed to run the program to four cores had a runtime of 87.776708 seconds, assuming there were no changes to the number range. According to the results, the runtimes can scale linearly but there are uncertainties to the consistency when running the program in four, six, and eight cores gives an erratic timestamp. The table below shows the elapsed time between then second and last primes before the program is finalized.

Cores	Runtime (in seconds)
4	67.486932
6	65.137937
8	32.779

Lab 4 Exercise 2

Moving forward, understanding the use of broadcasts and how the master process can utilize the workers in a network to perform calculations to determine if a number is prime allows another part of the project to continue exploring distributed parallel computing. Another concept to investigate is called MPI scatter and gather. MPI scatter is another collective routine that sends chunks of an array into different workers. Although broadcasting pays a similar feature where its concept is to send the same piece of data to all workers, scatter's functionality is key to its differentiation from broadcasting. Then, MPI gather is the inverse of scatter where all the workers' data are transmitted back to the master process and handled for output or to continue operation based on the processes' work.

In this part of the project, a program is created to demonstrate the use of scattering and gathering under MPI's functions which are 'MPI_Scatter' and 'MPI_Gather'. The goal of the program is to return the sum of a factorial, given a number that is depended on the number of cores used and the entries to scan for each core. For example, if an array size is given as an integer of 72 then the cores and the chunk size to scan must multiply to 72. Otherwise, overheads are introduced that can either hinder or idle the runtime execution. Then, for each entry in the array, an integer of 10 is included to calculate the factorial when 'MPI_Scatter' is called. When the processes finish the calculations, 'MPI_Gather' is called to return the values back to the controller and then the sum of the factorial is outputted onto the terminal with a timestamp of the runtime.

Finalizing the results, the table below shows the runtimes of each core with a chunk size that represents how much the process can calculate:

Cores	Chunk Size	Total Sum (calculating the factorial of 10 and then summed up)	Runtime (in seconds)
8	9	228614400	0.015542
6	12	217728000	0.010461
4	18	195955200	0.000947

Source Code (Software)

Lab 4 Exercise 1

```c
# include <math.h>
# include <mpi.h>
# include <stdio.h>
# include <stdlib.h>
# include <time.h>

int main ( int argc, char *argv[] );
int prime_number ( int n, int id, int p );
void timestamp ( );

int main ( int argc, char *argv[] )
{
  int id;
  int ierr;
  int n;
  int n_factor;
  int n_hi;
  int n_lo;
  int p;
  int primes;
  int primes_part;
  double wtime;

  n_lo = 1;
  n_hi = 262144;
  n_factor = 2;

  //Initialize MPI.
  ierr = MPI_Init ( &argc, &argv );
  if ( ierr != 0 ){
    printf ( "\n" );
    printf ( "PRIME_MPI - Fatal error!\n" );
    printf ( "  MPI_Init returns nonzero IERR.\n" );
    exit ( 1 );
  }

  //Get the number of processes.
  ierr = MPI_Comm_size ( MPI_COMM_WORLD, &p );

  //Determine this processes's rank.
  ierr = MPI_Comm_rank ( MPI_COMM_WORLD, &id );

  if ( id == 0 ){
    timestamp ( );
    printf ( "\n" );
    printf ( "PRIME_MPI\n" );
    printf ( "  C/MPI version\n" );
    printf ( "\n" );
    printf ( "  An MPI example program to count the number of primes.\n" );
    printf ( "  The number of processes is %d\n", p );
    printf ( "\n" );
```

```
 52.     printf ( "         N        Pi          Time\n" );
 53.     printf ( "\n" );
 54.   }
 55.
 56.   n = n_lo;
 57.
 58.   while ( n <= n_hi ){
 59.     if ( id == 0 ){
 60.       wtime = MPI_Wtime ( );
 61.     }
 62.     ierr = MPI_Bcast ( &n, 1, MPI_INT, 0, MPI_COMM_WORLD );
 63.     primes_part = prime_number ( n, id, p );
 64.     ierr = MPI_Reduce ( &primes_part, &primes, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD );
 65.     if ( id == 0 ){
 66.       wtime = MPI_Wtime ( ) - wtime;
 67.       printf ( "  %8d  %8d  %14f\n", n, primes, wtime );
 68.     }
 69.     n = n * n_factor;
 70.   }
 71.
 72.   //Terminate MPI.
 73.   ierr = MPI_Finalize ( );
 74.
 75.   //Terminate.
 76.   if ( id == 0 )
 77.   {
 78.     printf ( "\n");
 79.     printf ( "PRIME_MPI - Master process:\n");
 80.     printf ( "  Normal end of execution.\n");
 81.     printf ( "\n" );
 82.     timestamp ( );
 83.   }
 84.
 85.   return 0;
 86. }
 87.
 88. int prime_number ( int n, int id, int p )
 89. {
 90.   int i;
 91.   int j;
 92.   int prime;
 93.   int total;
 94.
 95.   total = 0;
 96.
 97.   for ( i = 2 + id; i <= n; i = i + p )
 98.   {
 99.     prime = 1;
100.     for ( j = 2; j < i; j++ )
101.     {
102.       if ( ( i % j ) == 0 )
103.       {
104.         prime = 0;
105.         break;
```

```
      }
    }
    total = total + prime;
  }
  return total;
}

void timestamp ( )
{
# define TIME_SIZE 40

  static char time_buffer[TIME_SIZE];
  const struct tm *tm;
  time_t now;

  now = time ( NULL );
  tm = localtime ( &now );

  strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );

  printf ( "%s\n", time_buffer );

  return;
# undef TIME_SIZE
}

```

Lab 4 Exercise 2

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#include <assert.h>

#define ARRAY_SIZE 72 //size of array

int *create_nums(int num_elements) { //fill array with the number 10
  int *numbers = (int *)malloc(sizeof(int) * num_elements);
  for(int i = 0; i < num_elements; i++){
    numbers[i] = 10;
  }
  return numbers;
}

int factorial ( int number ){ //take the factorial recursively
    if(number == 0){
            return 1;
    }
    else {
            int fact = (number * factorial(number - 1));
            return fact;
    }
}
```

```
26.
27. int compute_sum(int *array, int num_elements) { //compute sum of the factorials
that are sent to each core
28.   int i;
29.   int sum = 0;
30.   for (i = 0; i < num_elements; i++) {
31.     sum += factorial(array[i]);
32.   }
33.   return sum;
34. }
35.
36. int final_compute_sum(int *array, int num_elements) { //compute sum of the sums
returned from each core
37.   int i;
38.   int sum = 0;
39.   for (i = 0; i < num_elements; i++) {
40.     printf("Total sum: %d\n", sum);
41.     sum += array[i];
42.   }
43.   return sum;
44. }
45.
46. int main(int argc, char** argv) {
47.
48.   double wtime; //variable to record time
49.   int num_elements_per_proc = atoi(argv[1]); //number of elements to send to each
core
50.
51.   MPI_Init(NULL, NULL);
52.
53.   int world_rank; //define processor number through world rank
54.   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
55.   int world_size; //define number of processors
56.   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
57.
58.   int *numbers = NULL;
59.   if (world_rank == 0) { //if root node
60.     numbers = create_nums(ARRAY_SIZE); //create array of number 10
61.     wtime = MPI_Wtime ( ); //start recording time
62.   }
63.
64.   int *sub_nums = (int *)malloc(sizeof(int) * num_elements_per_proc); //buffer to
send array elements to each core
65.   assert(sub_nums != NULL);
66.
67.   MPI_Scatter(numbers, num_elements_per_proc, MPI_INT, sub_nums,
68.               num_elements_per_proc, MPI_INT, 0, MPI_COMM_WORLD); //scatter the
array elements
69.
70.   int sub_sum = compute_sum(sub_nums, num_elements_per_proc); //get sum of the
factorials of passed array section
71.
72.   int *sub_sums = (int *)malloc(sizeof(int) * world_size); //return buffer
73.   assert(sub_sums != NULL);
```

```
74.   MPI_Gather(&sub_sum, 1, MPI_INT, sub_sums, 1, MPI_INT, 0, MPI_COMM_WORLD);
//rejoin everything
75.
76.   if(world_rank == 0){
77.     int sum = final_compute_sum(sub_sums, world_size); //sum all returned values
78.     printf("Sum of all factorials from proc %d is %d\n", world_rank, sum);
//print final value
79.     wtime = MPI_Wtime ( ) - wtime; //find time difference
80.     printf("Time elapsed: %14f\n", wtime); //print elapsed time
81.   }
82.
83.
84.   if (world_rank == 0) {  // if root, free allocated memory
85.     free(numbers);
86.   }
87.   free(sub_sums);
88.   free(sub_nums);
89.
90.   MPI_Barrier(MPI_COMM_WORLD);
91.   MPI_Finalize();
92. }
93.
```

Schematics (Hardware)

None

Analysis

With the use of MPICH, execution of processes became more efficient. Combining multiple Raspberry Pi's to act as one system introduced more cores for the process to be split up into. For this process, the systems have the be on the same basis when it comes to communication. Version of the packages the systems are utilizing must be similar to each other. The files of both controller and worker nodes must correlate with their duties in the MPICH system. IP addresses must align with the controller and worker nodes, so the systems know exactly what other systems they are communicating with. Lastly, is the source code. If a share file system is not implemented, the controller and worker nodes must have the same source code when splitting up processes. This is because the MPICH has a forking nature where at the time of 'MPI_Scatter' the worker nodes start executing their program at the same level. If the source code does not align with that of the controller node, the worker nodes are doing unrelated work that was not specified by the controller node. With everything aligned throughout the nodes, the process was forked, and execution became faster.

This process is incorporated into many different areas in technology. In many scenarios, a singular system does not suffice the amount of work needed be completed by a program. This is where MPICH comes in, allowing for additional systems to be joined together and for the work to be distributed throughout the cores. This speeds up programs by allocating more resources to the system. There is no limit to how many worker nodes can be added however, the overhead of the computations could potentially be a limiting factor. Since the controller is scattering and gathering from multiple worker nodes, the time in which it takes to gather the information increases with the number of worker nodes. The more cores used in the MPICH means that more data must be gathered more times than if it were only a couple cores being used. To combat this, how the data is being distributed and the type of processes that are being assigned to the worker nodes can be closely monitored. Modifying the type of processes each worker node is assigned with could result in less overhead time.

The nature of MPICH also correlates with a group project compared to individual work. Having a group project takes pressure off the individual due to the work being divide among the different team members, like the process being divided among the different cores of multiple systems. Since more members of the team are contributing to a singular goal of completing the project, it is finished in more of an efficient and timely manner. The individual who takes on the project individually experiences more workload and time consumption with the project because the work cannot be divided and worked on at the same time.

Conclusion

The results effectively demonstrated the power and scalability of distributed parallel computing using the Message Passing Interface (MPI) with MPICH. Utilizing multiple Raspberry Pi nodes configured as controllers and workers, the lab highlighted the importance of synchronization, efficient resource allocation, and communication protocols. The results validated that parallelization significantly reduces computational time, yet also revealed the challenges of scaling, such as overhead and network latencies. These insights prove the value of distributed systems in modern computing, where tasks can be divided and conquered across multiple processors for modern performance. The results further reinforce the practical application of parallelism in solving computationally intensive problems, with very high potential to optimize performance.

Demos

Lab 4 Exercise 1: https://cardmaillouisville.sharepoint.com/:v:/s/CSE525-Groups910/EfDBNKK9YoVOt99KHJrVCigBPwPkQrsMH5oNaoO56aNX6A?e=V5gJpj&nav=eyJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJTdHJlYW1XZWJBcHAiLCJyZWZlcnJhbFZpZXciOiJTaGFyZURpYWxvZy1MaW5rIiwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsTW9kZSI6InZpZXcifX0%3D

Lab 4 Exercise 2: https://cardmaillouisville.sharepoint.com/:v:/s/CSE525-Groups910/EVFp8T6X8BJNnqwfr7XuS1ABBgLDfzf3Pr2nlRYhSQRvAA?e=t6vCd7&nav=eyJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJTdHJlYW1XZWJBcHAiLCJyZWZlcnJhbFZpZXciOiJTaGFyZURpYWxvZy1MaW5rIiwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsTW9kZSI6InZpZXcifX0%3D

References

MPI v4.1.7 Documentation: https://www.open-mpi.org/doc/v4.1/

MPI Tutorials: https://mpitutorial.com/tutorials/