2021 Raspberry Pi Programming Competition Report

Taylor Powell

October 28, 2021

Table of Contents

Page

1 What is Parallelization? 1

2 Building the Cluster 2

3 Common Parallelization Techniques in C++ 3

4 Programs for Testing the Cluster 4

5 Results 8

6 Conclusion 11

SECTION 1

What is Parallelization?

For my entry into ODU's 2021 Computer Science Department's Raspberry Pi Programming Competition, I decided to examine the benefits and methods of parallelization in programming. This topic is particularly relevant as we examine the utility of cluster computing, especially on modern supercomputers.

ODU has a cluster, Wahab, which is available for use by students, faculty, and affiliated researchers for computational tasks. However, in order to take advantage of the power offered by such a system, one must first develop a programming methodology which takes advantage of the hundreds and hundreds of processing nodes available on the cluster.

In most computationally-intensive programs, the time spent computing is usually consumed by some process which is repeated over and over again. Let's take a particular example: numerical integration of a function over a definite domain. The concept of integration is taught beginning with a Riemann sum definition,

$$\int_a^b f(x)dx = \lim_{N\to\infty}\sum_{i=1}^{N} f(x_i)\Delta x \qquad a \leq x_i \leq b$$

Visually, we can visualize this integration process as breaking the function into N intervals, evaluating the function at these points, and approximating the area under the curve as the sum of rectangles of width dx and height $f(x_i)$. This approximation becomes more accurate as we take larger and larger values of N, since the rectangles get smaller and the error from over and underestimates decreases.

Implementing this technique through a program, we can make a computer do all the computational work, allowing us to examine results of millions and billions of subintervals with relative ease, a task which is

arduous and time-consuming for even a large team of humans. However, while the time required for each computation is minuscule, it is non-zero, and increasing the number of calculations will eventually leave the user waiting long periods of time to obtain a result.

This is where parallelization comes in handy. Since each task is the same as another, we can instead break the job into sub-tasks, assigning a group of tasks to different processors and summing the total result once the tasks are done. Now, instead of one processor doing this computation N times, a team of 10 processors can each perform $N/10$ computations, leading to a commensurate reduction in computational time. Assuming no overhead and perfectly even distribution of work, the time savings for adding P processors would follow the relation,

$$T_P = \frac{T_1}{P}$$

Where T_1 is the time the job takes on a single processor. In real-world applications, there is also an associated overhead cost for the algorithm to allocate resources, communicate data across nodes, and distribute tasks appropriately, which reduces the overall time-savings for short jobs. However, this overhead becomes less important as the numbers of jobs and the complexity of each task increases.

SECTION 2

Building the Cluster

For this project, I built a cluster using four Raspberry Pi 3 Model B boards. In order to create a cluster out of these boards, several key tasks must be completed:

1. Obtain a Micro SD card for each board, install an operating system, and install the necessary software.
2. Provide each board with power
3. Network the boards together (in this case I used an unmanaged switch connected by Ethernet cables).
4. Establish communication between each of the nodes.

Since each Raspberry Pi 3 board hosts a quad-core processor, my proposed cluster would allow me to run up to 16 parallel processes at once across the four boards.

2.1 Assembly and Installation

I followed a basic list of required items and purchased them to assemble the cluster:

Product	Pricing	Link
Four Raspberry Pi 3 Model B Boards	$203.96	Link
Raspberry Pi Cluster Case	$21.99	Link
Micro SD Cards	$35.40	Link
Ethernet Switch	$14.24	Link
Micro USB Power Supply	$10.99	Link
Micro SD Card Reader	$16.95	Link
Ethernet Cables	$9.99	Link
Micro USB Power Cables	$7.99	Link

Table 1: Component and pricing data

After assembling the cluster case and installing the boards, I downloaded two standard operating systems. I made one Raspberry Pi the nominal "master" node and installed a full desktop version of Raspian (a Linux distribution specific to Raspberry Pis). For the other three "slave" nodes, I installed a headless (no desktop or installed software) version of Raspian.

I then hooked the assembly up to power and Ethernet. The first task was updating each node's system, installing some of the basic software, and installing useful libraries (i.e., Eigen3).

Figure 1: (Left) Components of the Cluster case pre-assembly. (Right) Assembled Cluster.

2.2 Networking and Communication

In order to communicate between the nodes without manually logging in to each node when performing tasks, I generated and exchanged RSA keys (the main encryption technique for modern systems which relies on the difficulty of factoring the product of large prime numbers) between each slave node and the master node, which replaces the password requirement for logging into the system. I also assigned each node a static IP address on my home network so they knew where to locate one another.

While having the nodes able to communicate with one another was a major step, I still had to dedicate one of my computer monitors to the Raspberry Pi project. I decided to then do the same process to get the Raspberry Pi's to communicate with my desktop PC (which runs Windows 10) through SSH so I could run the entire cluster headlessly.

Now that I could run the cluster from my desktop PC through the terminal, I then began the work of making and implementing various programs with or without parallelization in order to determine the efficiency of this process numerically.

SECTION 3

Common Parallelization Techniques in C++

In order to examine parallelization, we need to understand the various methods available and how they are implemented in `C++`. In order to tie this discussion to a specific implementation, I will specifically look at how to parallelize a `for` loop.

3.1 Serial Programming

When writing a standard program with a `for` loop in `C++`, the processor will do each iteration of the loop in sequence (serially). Recalling the earlier example of numerical integration using Riemann sums, we could

loop over the interval from left to right and sum the contributions from each iteration. The processor will compute $f(x_i)$ at each x_i value, multiply by dx, and add the result to the running total.

3.2 OpenMP

The first (and simplest) method of parallelizing a program is through a free library called `OpenMP`. `OpenMP` works by creating a local cache of shared memory space on a single device, but then distributes the tasks to all of the threads available on that device. On a single Raspberry Pi board, for example, since the Raspberry Pi 3 Model B has a quad-core processor, implementing `OpenMP` results four available threads. On my personal PC, I have a six-core processor, but each processor is capable of hyper-threading (enables 2 threads per node), thus I can run 12 processes through `OpenMP`.

In this library, there is a very simple way of parallelizing a `for` loop: before the line initializing the loop, include a statement of the type,

```
#pragma parallel for
```

Which is a preprocessor command informing the compiler that the following for loop will be locally parallelized. While this library is powerful for its simplicity and ability to speed up computational work on a single device, it will be unable to take advantage of the greatest speedups available across computing clusters.

3.3 MPI

The industry standard for parallelization across multiple separate processors is known as MPI (message passing interface). MPI does not create a shared memory space as in `OpenMP`, but instead each node creates its own local memory and initialization of the program. The user must then explicitly build in lines of communication between the nodes in order to aggregate computations on a single node for output.

SECTION 4

Programs for Testing the Cluster

Since my cluster is on a Linux distribution and is running headlessly, there are a number of files I created to properly compile and run `C++` programs.

The first file is a `CMakeLists.txt` file, which contains the necessary compilation instructions for CMake to generate an executable for my program. The `CMakeLists.txt` file I made lists all the files to include in the project, and I also includes a `-O2` flag to instruct the compiler to take advantage of optimizations where applicable. Since I am also using MPI in these projects, I also include the MPI library through the `CMakeLists.txt` script. In programs where I use it, I also include the Eigen3 library with a call inside the `CMakeLists.txt` script.

The next item I use is a plaintext file to provide all the static IP addresses for the nodes in the cluster named `machines.txt`. When entering the command to execute a program using MPI, including this plaintext file allows me to skip manually typing in each of the node's IP addresses by hand.

I also wrote three script files to help me do file management across the cluster:

1. `runme.sh` - Creates a build directory and calls CMake to generate an executable file for the program inside the build folder.
2. `copyit.sh` - Copies the executable file to the main folder, deletes the build folder, and copies the executable across all nodes in the cluster.
3. `deleteit.sh` - Wipes the executable from all nodes in the cluster, returning the file system to the original state before `runme.sh` was initially called.

4.1 Basics for MPI

For any `.cpp` file which is MPI-compatible, they have a few commonalities:

```
#include <mpi.h>

int main(int argc, char **argv)
{
    int node, nproc;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &node);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);

    MPI_Finalize();
    return 0;
}
```

From this dummy code, we see a few of the basic commands to work with MPI. The process is initialized with `MPI_Init()` and ended with `MPI_Finalize()`. In between, we can find the total number of processors allocated for the job with `MPI_Comm_size()` and the ID number for this thread with `MPI_Comm_rank()`. In order to do more interesting tasks, there are a number of other communications-related commands. In particular, for this project I primarily made use of `MPI_Barrier()` and `MPI_Reduce()`, which work together to allow basic communication between threads.

4.2 Checking Communication with Each Node

The first task to check whether the cluster is working properly is the MPI equivalent of a "Hello world!" program. In this case, I made two simple script files:

1. A test program which has each node print its node number.
2. A test program to have each thread print its process ID.

These are the simplest possible programs to ensure the cluster is working properly and performing as expected.

4.3 Simple Integration Program

The idea of this program is to integrate a function using a naïve Riemann sums approach. In order to make the program suitable for parallelization, the integration range is split evenly between all the threads allocated for the task. If there are `nproc` processors and `N` intervals, then each processor has to complete `partition = N / nproc` tasks.

The interval can then be split across each of the threads by modifying a `for` loop as,

```
for(int i=node*partition; i < node*partition+partition; i++)
```

After each thread computes a local sum for its subinterval using the `MPI_Reduce()` function. In this instance, since I am seeking to sum the contribution from each thread and store the value into `trueValue`, the reduction takes place as,

```
MPI_Reduce(&value, &trueValue, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Then, I finally output the results on whichever node is given rank 0 through a simple if statement of the form: `if(node == 0) {cout << trueValue;}`

4.4 Adaptive Integration Using Gauss Quadrature

Now we come to the interesting application of parallel programming I examine where we can examine the consequences of non-=uniform task distribution. Specifically, I used an implementation of n-point Gauss quadrature with an adaptive integration algorithm to compute several integrals of interest.

4.4.1 Gauss-Legendre Quadrature

For the standard Gauss-Legendre quadrature (Ref. [1]), we modify an integral over the region $-1 \leq x \leq 1$ in terms of a series of weights (w_i) as,

$$\int_{-1}^{1} f(x)dx \simeq \sum_{i=1}^{n} w_i f(x_i)$$

Where x_i are the i-th zeroes of the corresponding n-th order Legendre polynomials. For any region which is not integrated from $[-1, 1]$, we can perform a change of integral simply by,

$$\int_a^b f(x)dx = \int_{-1}^{1} f\left(\frac{b-a}{2}x + \frac{a+b}{2}\right)\left(\frac{b-a}{2}\right)dx \simeq \frac{b-a}{2}\sum_{i=1}^{n} w_i f\left(\frac{b-a}{2}x_i + \frac{a+b}{2}\right)$$

4.4.2 Computational Approach

For my implementation of Gauss quadrature, I wrote the program to be flexible for any desired number of points for Gauss quadrature (n-point). Since the order of the Gauss rule changes the number and value of the weights used in the computation, I also wrote functions to quickly compute the roots of n-th order Legendre polynomials and compute the weights from them. The Legendre polynomials can be computed at a point quickly to arbitrary precision using the recurrence relation (Ref. [2]),

$$P_n(x) = \frac{(2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x)}{n}$$

Where $P_0(x) = x$ and $P_1(x) = x$. The roots are computed using a function named *LegendreRootFinder* and the weights with *LegendreWeightFinder*.

LegendreRootFinder: Since the roots are symmetric in the interval $[-1, 1]$ about $x = 0$, and the number of roots is equal to the order of the Legendre polynomial, we use a modified brute-force algorithm. I search in the interval $[-1, -\epsilon]$ (with $\epsilon = 10^{-9}$) for all roots. The interval is split into a region 40 times more fine than the number of roots on the interval (Legendre polynomials are generally well-behaved and roots are decently spaced), and each of them are checked for a sign change. Is a sign change is found, bisectional root finding is run to a precision of 10^{-15}. Then, for all computed roots, they are mirrored to the positive interval $[\epsilon, 1]$. Finally, for odd-order Legendre polynomials, the guaranteed root at $x = 0$ is added in manually.

LegendreWeightFinder: Taking the roots found previously, each weight is computed using the relation,

$$w_i = \frac{2}{(1 - x_i^2)[P_n'(x_i)]^2}$$

Where the derivative of the Legendre polynomial is computed using the recurrence relation,

$$\frac{x^2 - 1}{n}\frac{d}{dx}P_n(x) = xP_n(x) - P_{n-1}(x)$$

And these weights are passed back to the Gauss quadrature function to be applied over the interval of integration.

4.4.3 Adaptive Integration

I followed a standard adaptive integration scheme (roughly adapted from Ref. [3]):

1. Call integration function over $[x_L, x_R]$ with tolerance ϵ
2. Compute approximate integral using Gauss n-point quadrature
3. Break integral into two halves and compute approximate integral on each subinterval.
4. Check if error is below tolerance. If not, recursively call integration function on each subinterval with the acceptable tolerance halved ($\epsilon' = \epsilon/2$).
5. Return value to calling environment

It just remains to develop a methodology for testing error on over the interval. I do this in with two conditions:

1. Compare the value of the previous integral to the value of the integral approximated over each of the two subintervals. If their value differs by less than the desired tolerance, error is satisfactory.
2. Check if the difference between x_L and x_R is approaching machine precision for subtraction. If so, error is "as good as it gets" for this interval.

These two conditions are compared with an *OR* condition before the approximation is accepted and returned to the calling function.

4.4.4 Why Is Adaptive Integration a Good Test for Parallelization?

For this project, I wanted to explore the costs and benefits of parallelization. For most simple programs where the time-complexity arises from repeated calculations inside a looped structure, simply subdividing the iterations for the loop across the available threads is sufficient to ensure significant computational speedup.

Adaptive integration is a good test case for when this approach may not be good enough. For the computational scheme outlined in the previous section, the program may not need to uniformly sample a region to get an acceptable approximation if the function in one region varies more rapidly than in another area. For a concrete example, lets take a look at three sinusoidal functions: $\sin(10x)$, $\sin(10x^2)$, and $\sin(10x^3)$.

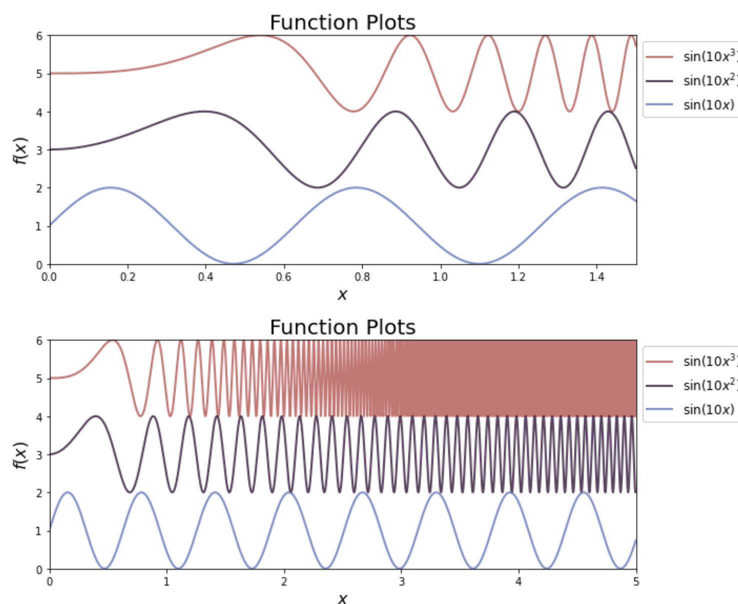

Figure 2: (Top) Each function plotted over the domain $0 \leq x \leq 1.5$. (Bottom) Each function plotted over the domain $0 \leq x \leq 5$.

From Figure 2, we see the behavior of each function. Particularly, the function $\sin(10x)$ is well-behaved and uniformly varying across the interval. Since sine has a linear dependence on x, its period of oscillation remains the same. By contrast, the other two functions have polynomial dependence on x, and thus their periods shorten drastically as x increases.

For an adaptive integration routine, it will need to sample the right side of each function's interval more finely in order to generate a sufficiently accurate approximation for the area under the curve. This means, if we naïvely split the region evenly between the various threads and call adaptive integration, we would expect the threads working in regions further from the origin to take more computational time than those nearer to the origin. Thus, some threads may finish early and be idle while others are still working, which is an undesirable quality for a parallelized program. We will seek to quantify this discrepancy in this simple case and suggest possible algorithmic changes to ensure the workload is shared more evenly for added speedup.

SECTION 5

Results

5.1 Checking Communication with Each Node

```
pi@node1:~/Documents/TestProjects/test004 $ mpiexec -n 16 -hostfile machines.txt hostname
node1
node1
node1
node1
node4
node4
node4
node4
node3
node3
node3
node3
node2
node2
node2
node2
```

```
pi@node1:~/projects $ mpiexec -hostfile machines.txt -n 16 test004
I am 1 of 16
I am 2 of 16
I am 3 of 16
I am 4 of 16
I am 11 of 16
I am 6 of 16
I am 12 of 16
I am 7 of 16
I am 8 of 16
I am 9 of 16
I am 10 of 16
I am 5 of 16
I am 13 of 16
I am 15 of 16
I am 16 of 16
I am 14 of 16
```

Figure 3: (Top) Checking where each thread lives. (Bottom) Retrieving thread IDs.

For this straightforward test of the cluster's communication, we see the expected results. There are four nodes, and each node carries four threads. In the bottom screenshot, we see the threads are numbered $0-15$ and there are a total of 16 threads. For the command line, we execute the program using,

```
mpiexec -n 16 -hostfile machine.txt test004
```

In this case, `mpiexec` is a command specifying a MPI-based execution of the program. `-n 16` informs MPI to generate 16 individual threads, and `-hostfile machine.txt` provides the computer with the static IP addresses for each of the nodes on the cluster to use for this application. Finally, `test004` happens to be the name of the compiled executable for this simple program.

5.2 Simple Integration Program

For the simple integration program, I computed the integral of $\ln(x^2 + 0.0000000001)$ over the domain $-0.1 \leq x \leq 0.1$. I chose this function since it is narrowly spiked near the origin and requires very fine sampling to obtain a decent estimate of the true value of the integral. To accomplish this, I used 1.6 million sample points over the interval. The interval is evenly subdivided and sent to each of the nodes allocated for the process, then the result is outputted on the thread with a thread ID of 0.

I ran this method then on the cluster and varied the number of allocated threads for each run between 1 and 16 and recorded the time it took for the result. The results are shown in Figure 4, where I have scaled time in terms of percent of maximum time. From our naïve expectation for an ideal parallelization routine, a curve fit to this data should be $T_P = P^{-1}$. When fitting to this function, we find a slightly modified coefficient of $T_P \propto P^{-0.978}$, which is very close to the expectation of -1. Therefore, we can be confident that the program is efficiently distributing the workload across each of the the nodes and there is little time lost to computational overhead.

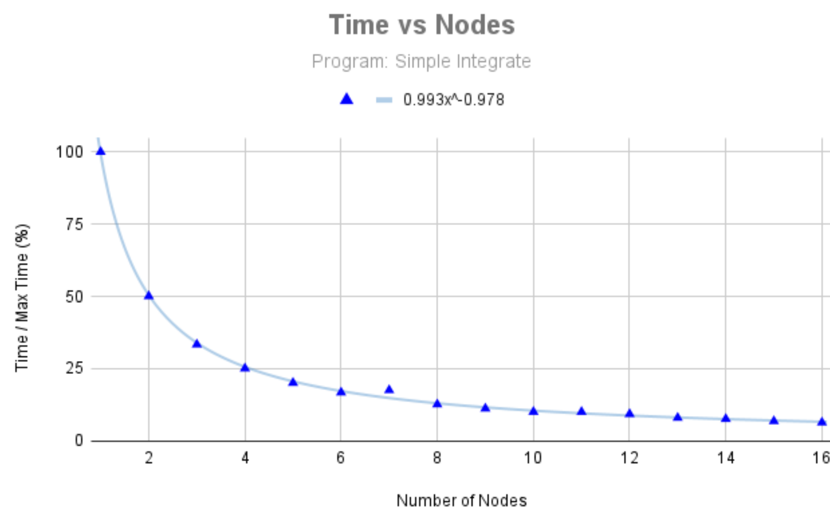

Figure 4: Plot of the time taken to compute a simple integral versus number of nodes used.

5.3 Adaptive Integration Using Gauss Quadrature

For the adaptive integration program implementing n-point Gauss quadrature, I fixed the order to 10-point Gauss quadrature and examined integrals of the three functions previously discussed. Additionally, with this program, I am primarily interested in quantifying the difference in speedup between the slowest threads and the fastest threads (which for these functions will be the right-most and left-most intervals, respectively). I therefore plot the timing data for the fastest and slowest threads for each run (normalized to the longest run) versus the number of threads.

In Figure 5, we see results for integrating the smoothly-varying function $f(x) = \sin(10x)$ over the domain $x \in \{0, 1000\}$. As expected, there is still a power-law relationship with a slope of nearly $m \approx -1$ since the work is relatively evenly distributed. This also manifests as very little "gap" between the two curves as the number of threads increases.

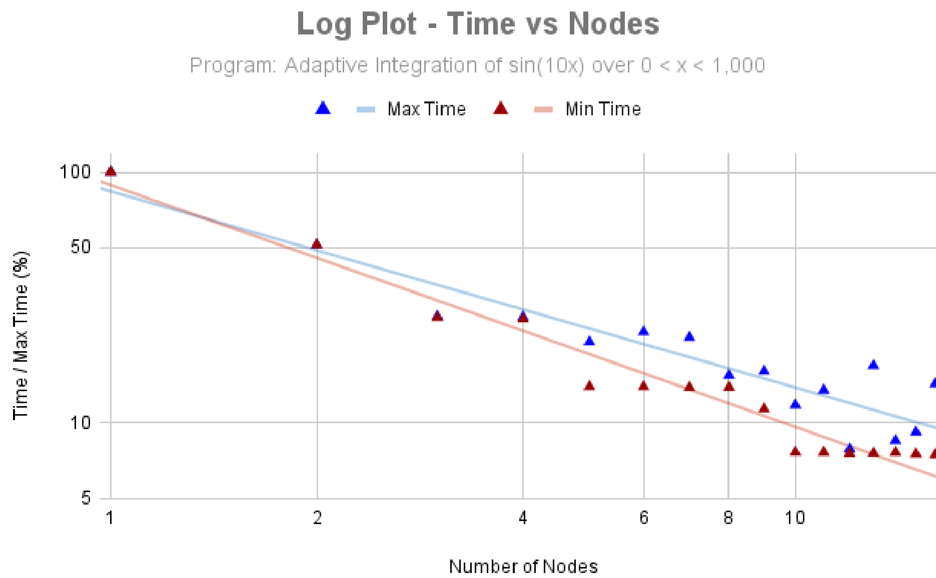

Figure 5: Log-log plot of time savings as the number of threads increases for the integral of $\sin(10x)$ over $x \in \{0, 1000\}$. Time is normalized to the maximum time spent.

In Figure 6, we see results for integrating $\sin(10x^2)$ over $x \in \{0, 100\}$ as well as $\sin(10x^3)$ over $x \in \{0, 20\}$. In each case, we see an obvious divergence in speedup between the maximum and minimum cases. For $f(x) = \sin(10x^3)$, the minimum curve follows a relation $T_{Pmin} \propto P^{-2}$ while the maximum curve follows $T_{Pmax} \propto P^{-0.717}$. Thus, some threads experience even faster speedups (as the more rapidly varying regions are removed from their domain) while the overall program runs sub-optimally.

We can also examine this in terms of the raw timing data. In the cubic case, the fastest thread is **55 times faster** than the slowest. If this program was running on a large cluster, improper distribution of the workload like this results in computational resources going completely unused for more than 90% of the run time, which is incredibly inefficient. This test emphasizes the importance of understanding that while parallelization offers huge advantages for speeding up computations, it is very important to understanding the way work is distributed across threads to ensure efficient use of resources.

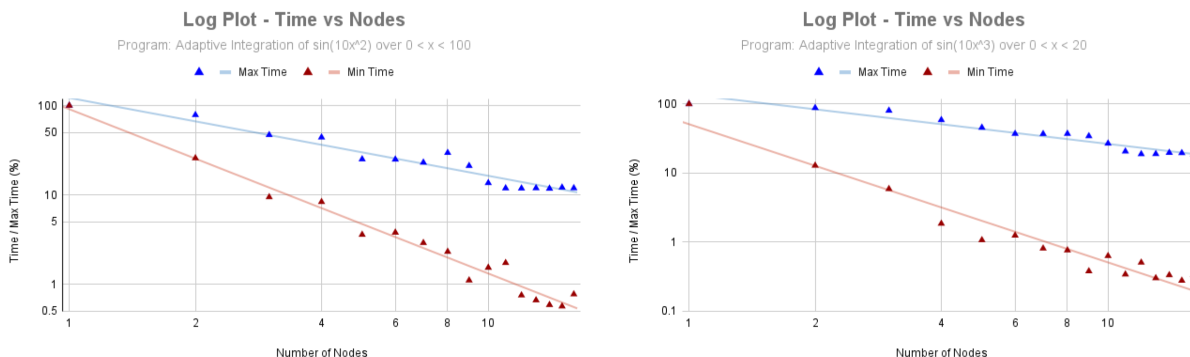

Figure 6: Log-log plots of time savings as the number of threads increases for the integral of (Left) $\sin(10x^2)$ over $x \in \{0, 100\}$ and (Right) $\sin(10x^3)$ over $x \in \{0, 20\}$. Time is normalized to the maximum time spent.

SECTION 6

Conclusion

This project was extremely productive for my own understanding and exploration of how to build and manage parallelized programs. I began using the Wahab cluster at ODU during the summer of 2021 for my research in the physics department because my programs could take days to complete on my personal PC. I taught myself how to use OpenMP and MPI to rewrite my code in a parallel way that was scalable. Suddenly, jobs that could take 48 or more hours when executed serially took a matter of 1 or 2 hours across a couple nodes on Wahab.

I was extremely intrigued by the operation of the cluster. after a brief email exchange with Dr. Elmesalami, I decided to build my own miniature version of a computing cluster to explore the architecture and development necessary to get separate nodes to communicate with one another.

This project was certainly non-trivial to complete, and I owe much of my success to tiny bits and pieces of information I gathered from online guides, message boards, and well-written program documentation from countless resources.

References

[1] K. F. Riley, M. P. Hobson, and S. J. Bence. 2006. *Mathematical Methods for Physics and Engineering.* Third Edition. Cambridge (UK): Cambridge University Press.

[2] G. Arfken, H. Weber, and F. Harris. 2012. *Mathematical Methods for Physicists.* New York: Elsevier Inc.

[3] J. N. Lyness. 1969. Notes on the Adaptive Simpson Quadrature Routine. Journal of the ACM. 16(3):483-495.

[4] P. J. Evans. 2019. "Building a Raspberry Pi cluster computer." The MagPi Magazine, available online at https://magpi.raspberrypi.com/articles/build-a-raspberry-pi-cluster-computer.

[5] S. Kenlon. 2020. "A beginner's guide to SSH for remote connection on Linux." OpenSource.com, available online at https://opensource.com/article/20/9/ssh.