The Parallel Computation of Pi

Alexander S. Cadigan
Department of Computer Science
Kalamazoo College
Kalamazoo, Michigan
alexander.cadigan15@kzoo.edu

Jennifer W. Cho
Department of Computer Science
Kalamazoo College
Kalamazoo, Michigan
jennifer.cho15@kzoo.edu

Abstract—**Pi's numeric value is involved in calculations across a broad range of academic studies and has been computed using computer algorithms. In this paper, we introduce and implement three parallel methods of approximating pi which are the interval, integral, and Monte Carlo method. The implementation was done in C++ using both OpenMP and MPI parallelizing approaches. The algorithms divide calculation tasks among different processes to approximate the value of pi. Our experimental results show that for OpenMP algorithm implementations, the integral method had the fastest runtime, highest speedup ratio, and greatest efficiency while the interval method had the slowest runtime, lowest speedup ratio and least efficiency. For MPI algorithm implementations, however, the interval method resulted in fastest runtime, highest speedup, and greatest efficiency. The Monte Carlo method implementation resulted in slowest runtime, lowest speedup, and least efficiency in both OpenMP and MPI implementations. In terms of scalability, we found that none of the implementations are scalable. Though, our interval and integral method approximation of pi resulted in zero percent errors while the Monte Carlo resulted in a very small percent error most likely due to its random method of calculating pi.**

I. Introduction

The numerical constant pi (π) is applicable and instrumental to a wide range of calculations that describe universal phenomenon. From the normal probability distribution, which describes natural probabilistic occurrences, to the Fourier Transform that converts a signal to a frequency spectrum, the value of pi is involved. With its occurrence in numerous field of studies and its property of never ending approximation, the calculation of pi has been a task adopted by numerous mathematicians since its discovery. Furthermore, due to the constant's great influence in both theoretical and realistic applications, it's only natural to analyze a preferable way of its approximation.

In this paper, the implementation of existing algorithms for computing pi in C++ is described in detail. Specifically for the purpose of this research, three methods for computing pi, which are the interval, integral, and Monte Carlo method, were implemented in both Open-MP and MPI to compare the parallel algorithms on two different parallel architectures and programming environments. We are interested in analyzing and fully understanding how the difference in the communication and combination of the results of parallel tasks affect the computation of pi. That said, our ultimate goal was to evaluate our algorithms' speed, efficiency, and accuracy for its approximation of pi.

II. Implementation

During the course of our research project, we tested three different algorithms used to approximate pi. These algorithms are referred to as the interval, integral, and Monte Carlo methods. We tested sequential, OpenMP, and MPI versions of these three pi calculation techniques on the Jigwé cluster, which contains 52 Intel(R) Xeon(R) Platinum 8164 CPU @ 2.00GHz CPU cores and has 196 GB of memory. The OpenMP implementations were tested using 2 to 2048 threads (increasing by a factor of 2), and the MPI implementations were tested using 2 to 128 Processors (increasing by a factor of 2). All three algorithms involve executing a calculation over multiple iterations with more iterations resulting in a more precise value of pi. As such, we tested our algorithms using 100 to 1,000,000,000 iterations (increasing by a factor of 10).

A. Interval

The interval method of calculating pi can be reduced to a relatively simple summation formula, given by

$$\pi = 4\sum_{i=0}^{N} \frac{\sqrt{1-\left(\frac{i-0.5}{N}\right)^2}}{N}$$

where N is the number of iterations to carry out. When implementing a parallel version of the interval algorithm, we divided the calculations up between the processes and then summed the resulting N sub-interval areas.

B. Integral

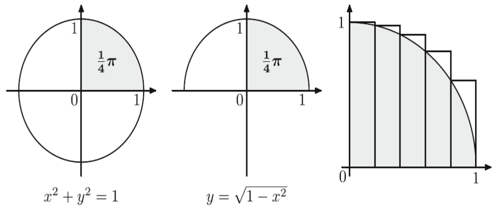

Fig. 1. Numerical Integration on interval [0,1]

The integral method is similar to the interval method in which it divides the number of calculation between the processes and the resulting N sub-intervals areas are summed. The algorithm takes the integral of a unit circle, $x^2 + y^2 = 1$, from the interval $[0, 1]$ which is split into N sub-intervals. By finding the explicit formula of the unit circle equation we can find that the integral equation for approximating pi evaluates to the equation below.

$$\pi = \sqrt{1 - x^2}dx$$

For simplification, the left Riemann sum integration method is chosen in our algorithm which means that the area of each rectangle a processor computes is equal to the width of a rectangle, $\frac{1}{N}$ multiplied by the function value computed in the left-end point of the interval. Figure 1 shows the shaded area of the circle that is being calculated and the intervals in which the integration occurs. That said, the integration equation can be summarized to the summation below.

$$\sum_{i=0}^{N-1} (\frac{1}{N}\sqrt{1 - (\frac{1}{N}i)^2})$$

C. Monte Carlo

The Monte Carlo implementation of approximating pi relies on a random shooting method. This computation shoots randomly into a square $[0, 1]$ x $[0, 1]$ while counting how many of the shots hit inside the unit circle and how many don't. The ratio of the number of hits and the total number of shots gives the approximation of the area of the unit circle. Figure 2 visualizes how the ratio is calculated. Since each shot is independent, the task of random shooting is distributed among different processes.

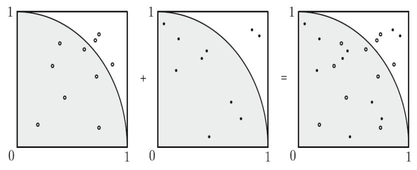

Fig. 2. Random Shooting Method on interval [0,1] x [0,1]

To generate random shots, the rand_r function was used because the usual random generating functions (rand or random) are not thread-safe. The function takes a seed, modifies it, and returns random values in the interval [0,1]. A distinct seed is used for each processes to with the value of the number of processes. This allows the algorithm to use one distince random generator for each thread.

III. Results

The figures below showcase the results from testing our three algorithms. It should be noted that we have only included results for when the number of iterations equals 1 billion. This is because as the number of iterations increases the approximation of pi will become more precise, and we wanted to display the data from the most accurate version of pi that we calculated. Figure 3 shows that the interval algorithm had the slowest runtime of the three methods. The Monte Carlo and integral methods had very similar runtimes, with the integral algorithm running slightly faster. We see

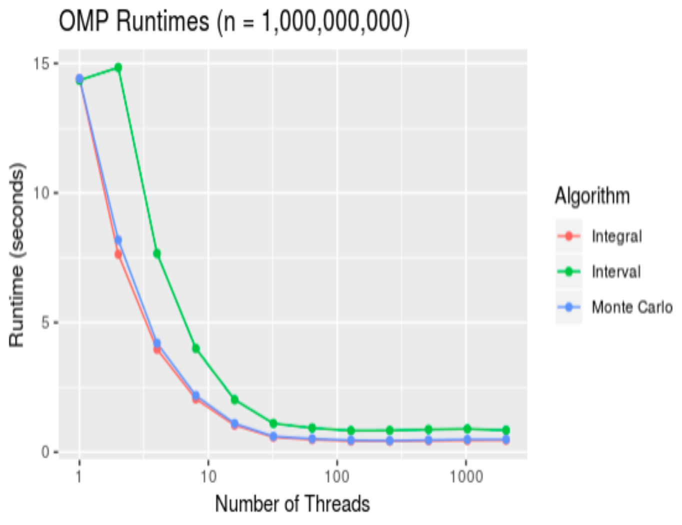

Fig. 3. OMP Algorithm Runtimes

a dramatic decrease in algorithm runtime as the number of threads increases, however, the runtime starts to level off once we hit 64 threads. Figure 4 shows that the Monte Carlo

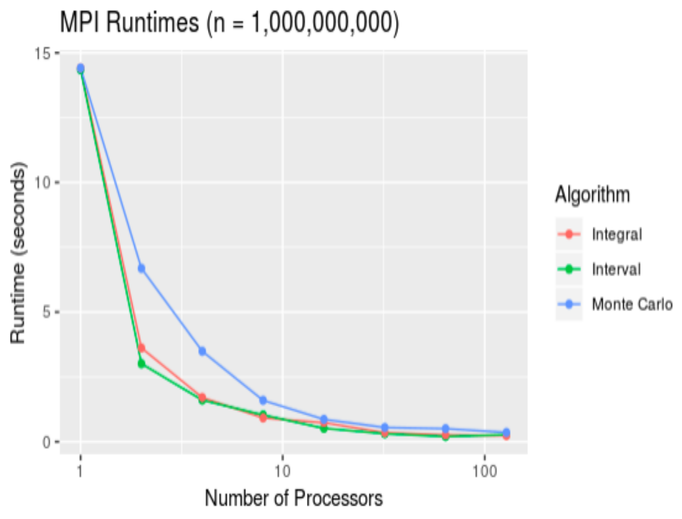

Fig. 4. MPI Algorithm Runtimes

algorithm had the slowest runtime of the three methods. The interval and integral methods had very similar runtimes, with the interval algorithm running slightly faster. Similar to the OMP algorithm runtimes, we see a dramatic decrease in runtime as the number of processors increases. The runtime begins to level off as we reach around 32 processors. Figure 5 shows that the integral algorithm had the greatest speedup ratio of the three methods. The Monte Carlo method speedups lag slightly below, and the interval method has much lower speedups than the other two. As the number of threads reaches 128, we start to see the speedup ratio level off and then begin to fall. Figure 6 shows that the interval algorithm had the greatest speedup ratio of the three methods. The integral

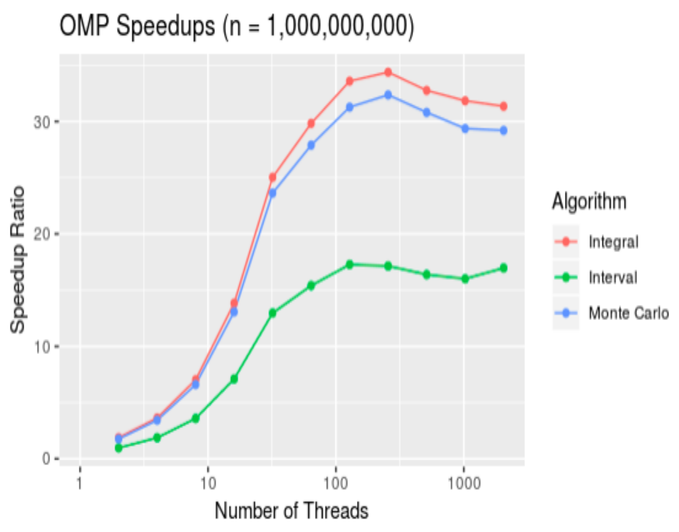

Fig. 5. OMP Algorithm Speedups

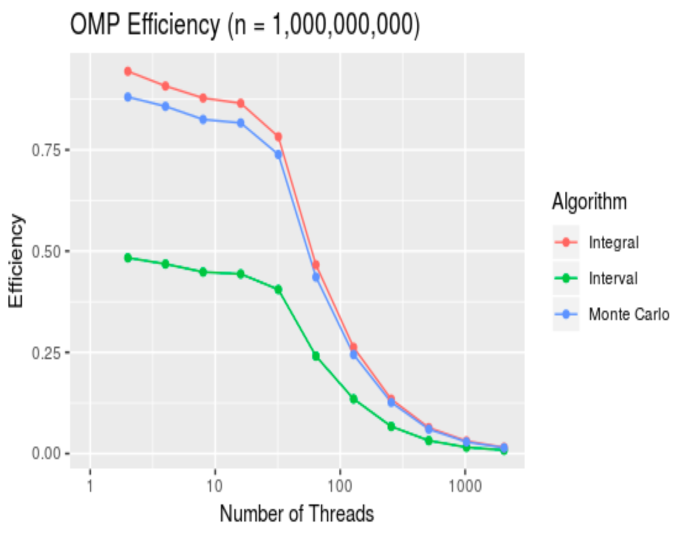

Fig. 7. OMP Algorithm Efficiency

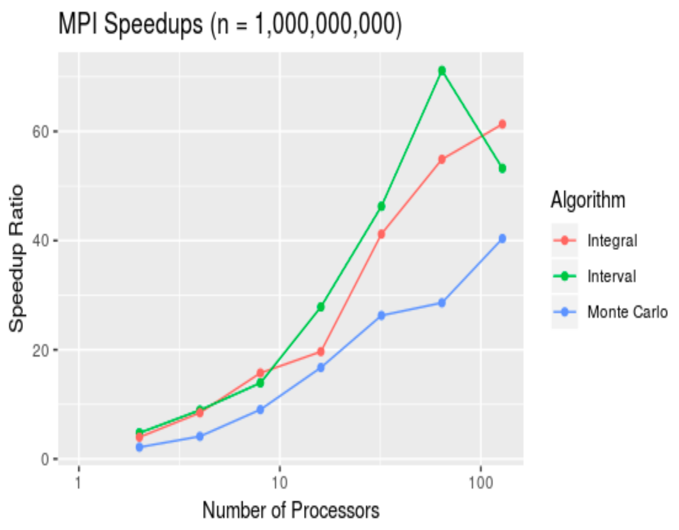

Fig. 6. MPI Algorithm Speedups

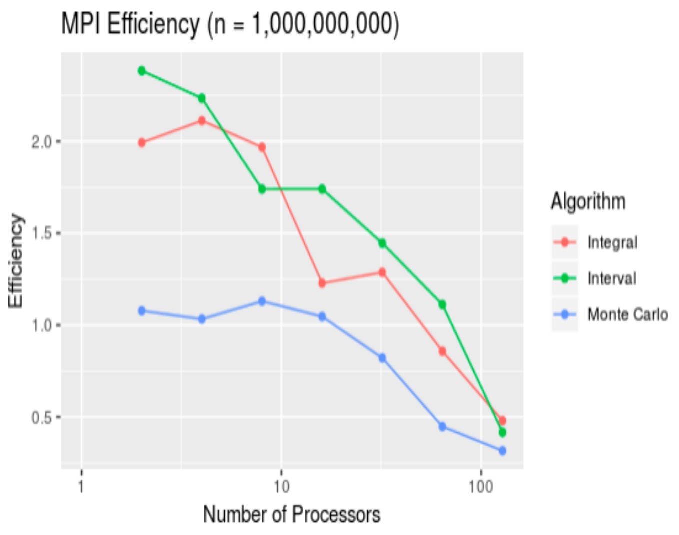

Fig. 8. MPI Algorithm Efficiency

method had the second greatest speedups, and the Monte Carlo method had the lowest speedups. We can see that the speedup ratio increases as the number of processors increases. Figure 7 shows that the integral algorithm had the greatest efficiency of the three methods. The Monte Carlo method was a close second, and the interval method had a much lower efficiency than the other two. We can see that the efficiency dramatically decreases as the number of threads increases, and the efficiency values for 2048 threads is close to 0. Figure 8 shows that the interval algorithm had the greatest efficiency on average of the three methods. The integral method was a close second, while the Monte Carlo method clearly had a lower efficiency than the other two. Like the OMP algorithms, the efficiency dramatically decreases as the number of processors increases. It should also be noted that our MPI algorithms produced absurd efficiency values of greater than 1. This likely means that the MPI algorithms do not scale well, which will be discussed later in greater detail. Figure 9 shows that both the interval and integral methods show a 0% error when run with 1 billion iterations. However, this does not mean that these methods calculated pi to an exact value. Instead, this indicates that these two methods were able to calculate pi correctly to the number of decimal places to which we were able to store (in this case the standard c++ double variable type). The Monte Carlo method showed a very small percent error, and we found no correlation between the number of threads and the percent error. Figure 10 shows that both the interval and integral methods show a 0% error when run with 1 billion iterations, just like the results for the OMP algorithms. We also see a very small percent error for the Monte Carlo method, and we found no correlation between the number of processors and the percent error.

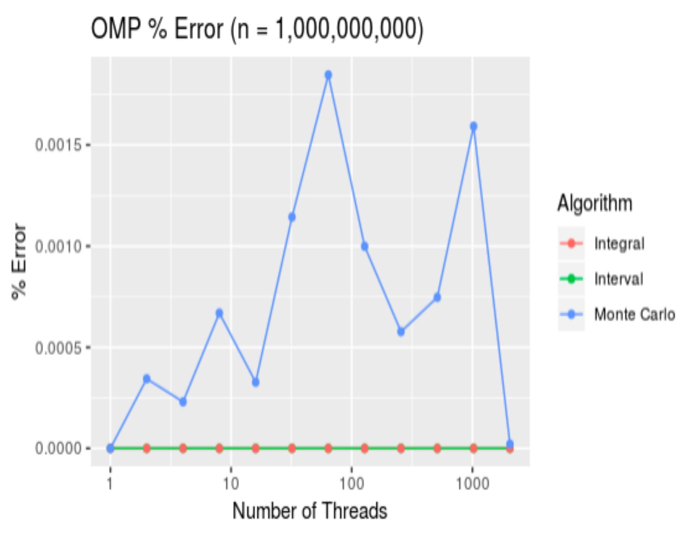

Fig. 9. OMP Algorithm Percent Error

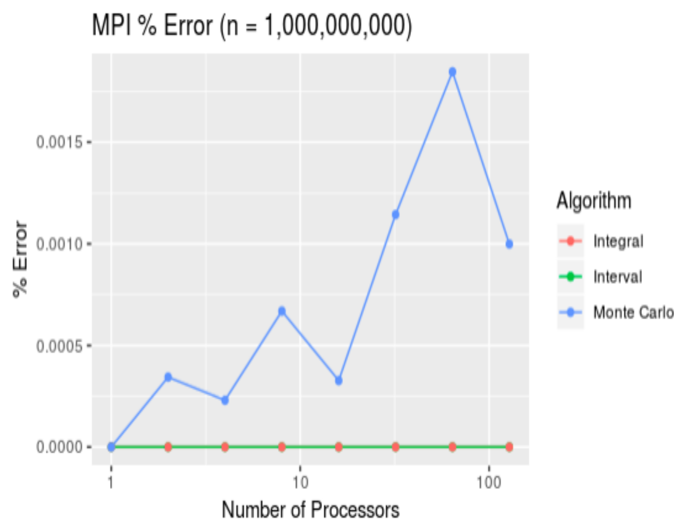

Fig. 10. MPI Algorithm Percent Error

IV. Conclusion

We found that for the OMP algorithm implementations the integral algorithm had the fastest runtime, highest speedup ratio, and greatest efficiency, while the interval algorithm had the slowest runtime, lowest speedup ratio, and least efficiency. However, we found that for the MPI algorithm implementations the interval algorithm had the fastest runtime, highest speedup ratio, and greatest efficiency, while the Monte Carlo algorithm had the slowest runtime, lowest speedup ratio and least efficiency. We also found that the interval and integral methods produced a 0% error when run with 1 billion iterations. As it turns out, the interval method produced a 0% error when run with 100,000 or more iterations, thus making the interval method slightly more precise than the integral method. Although the Monte Carlo method showed some percent error, the values are very small and are almost unnoticeable.

We also wanted to explore the scalability of the algorithms we had implemented. An algorithm is strongly scalable if the number of threads/processors can be increased without changing the problem size and the efficiency remains constant. Figure 11 shows the efficiency values for the OMP algorithms

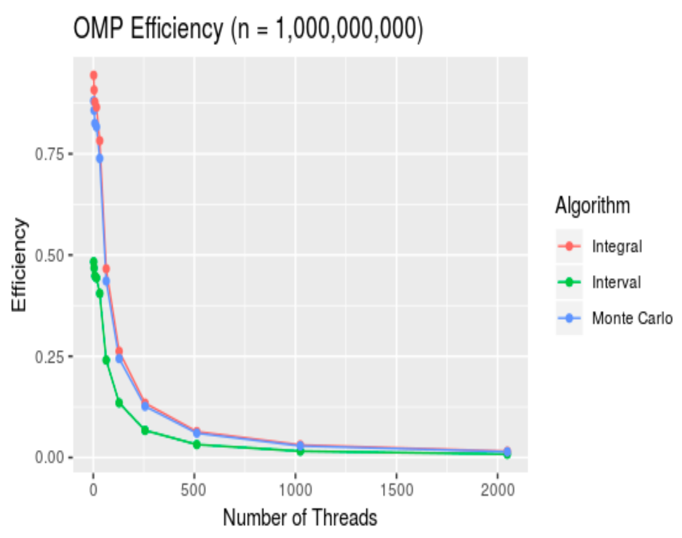

Fig. 11. OMP Algorithm Strongly Scalable Analysis

without using a logarithmic x-axis scale as to avoid distorting the figure. It is clear that the efficiency does not remain constant as the number of threads increases. Therefore, the OMP algorithms are not strongly scalable. Figure 12 shows the

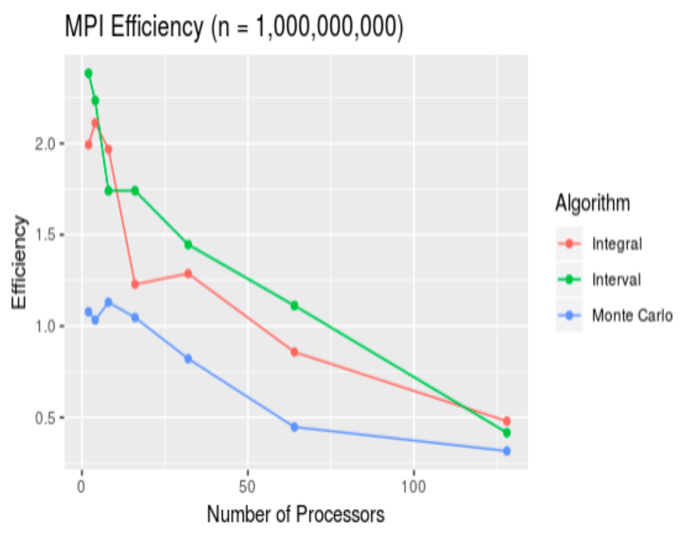

Fig. 12. MPI Algorithm Strongly Scalable Analysis

efficiency values for the MPI algorithms without a logarithmic x-axis scale. It it clear that the efficiency does not remain constant as the number of processors increases. Therefore, the MPI algorithms are not strongly scalable. An algorithm is weakly scalable if increasing the number of threads/processors and the problem size by the same amount results in constant efficiency values. Figure 13 shows the efficiency values for the

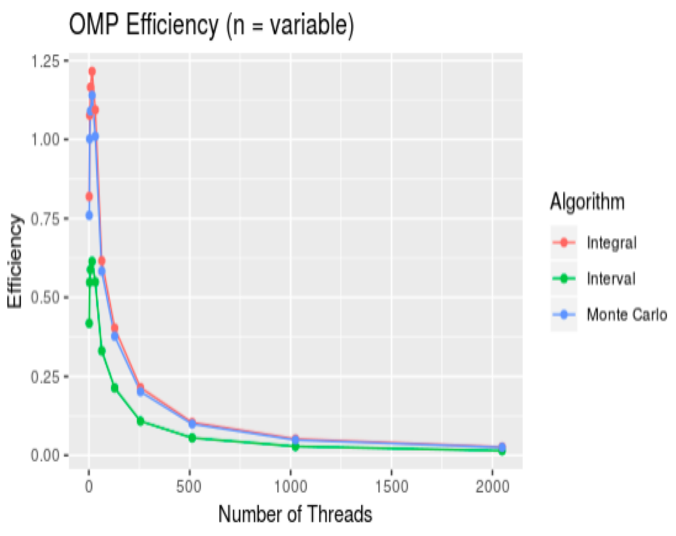

Fig. 13. OMP Algorithm Weakly Scalable Analysis

OMP algorithms when the number of iterations increases at the same rate as the number of threads. For example, when the number of threads is 2048, the number of iterations is 1 billion. However, when the number of threads is 1024, the number of iterations is 500 million. It is clear that the efficiency does not remain constant as the number of threads and iterations increase at the same rate. Therefore, the OMP algorithms are not weakly scalable. Figure 14 shows the efficiency values for

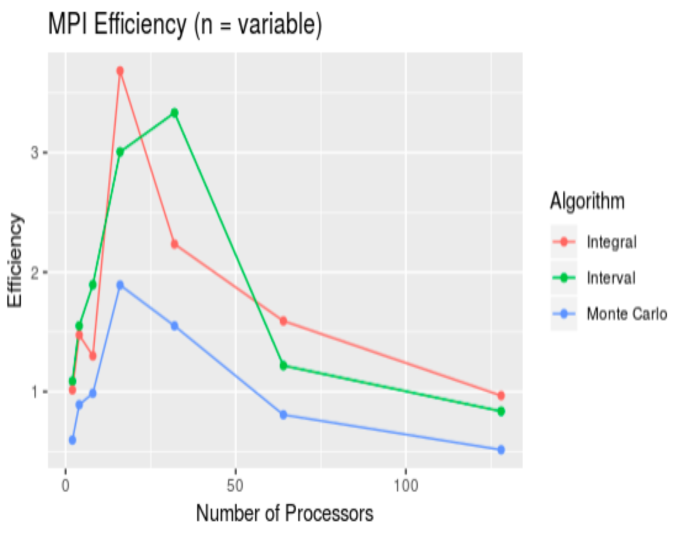

Fig. 14. MPI Algorithm Weakly Scalable Analysis

the MPI algorithms when the number of iterations increases at the same rate as the number of processors. For example, when the number of processors is 128, the number of iterations is 1 billion. However, when the number of processors is 64, the number of iterations is 500 million. It is clear that the efficiency does not remain constant as the number of processors and iterations increase at the same rate. Therefore, the MPI algorithms are not weakly scalable. In conclusion, we have determined that none of the algorithms we implemented are scalable. However, we can only say that they are not scalable under our testing environment. Perhaps if they were tested on a more powerful cluster with access to additional resources we would see a difference in our results.

With our implementation in C++, we were only able to approximate pi with up to 15 decimal places. However, a long double would allow for an approximation up to 21 decimal places depending on the compiler used. Computing pi with greater amount of decimal places will allow for a more accurate approximation. In addition, the interval and integral method would potentially not have a zero percent error.

For the purpose of studying implementing parallel algorithms using OpenMP and MPI, we could modify the algorithms experimenting with critical sections, atomic sections, and different message passing techniques. These modifications would not necessarily make the algorithm more efficient. Instead, it would be used for research purposes in terms of exploring techniques of parallelizing algorithms.

Additionally, for future directions, implementation of other methods of approximating pi will be considered. As mentioned earlier, due to pi's never ending property, the calculation of pi is a popular task among mathematicians across the world. Other methods to be considered for implementing as a parallel algorithm are the Lui Hui method, Ramanujan method, and Euler's method.

V. Acknowledgements

The authors would like to thank Dr. Sandino Vargas-Peréz for providing critical feedback in our work throughout the course and for teaching us the necessary foundations to implement parallel algorithms. Additionally, the authors would like to thank him for allowing them access to Jigwé for testing purposes.

References

[1] Trobec, R., Slivnik, B., Bulic, P., Robic, B. (2018). Introduction to parallel computing: from algorithms to programming on state-of-the-art platforms . Cham, Switzerland: Springer. https://doi.org/10.1007/978-3-319-98833-7
[2] Are There Any Practical Applications of Pi, aside from Calculating the Area or Circumference of a Circle? — The Curtis Center @ UCLA. n.d. Accessed March 13, 2019. http://curtiscenter.math.ucla.edu/content/pi$_{applications}$.