Peyman Shobeiri & Victor Cruz

IDs: 40327586 & 40136204

Fall 2024

## Table of Contents

## Question 4

**An approximate algorithm for calculating the value of π (PI) and its parallel version using the Master-Worker paradigm are provided towards the end of the tutorial that we discussed in the first class. Here is a link to the tutorial:**

**https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial**

**The pseudo-code is in the tutorial and the C code that uses MPI is available in the following link:**

**https://hpc-tutorials.llnl.gov/mpi/examples/mpi_pi_reduce.c**

**Here is what you are required to do:**

**A ) Write a sequential version of the algorithm and execute it on a single node of the cluster. You can take the parallel code and change it to sequential by removing the parallel components. Measure the execution time. (Note: For fairness of performance comparison, a sequential and parallel version must do equal amount of "total computational work", ignoring any other overheads.)**

The sequential version of the code can be run on only one processor. We have added the MPI_Wtime() function in order to compute execution time and removed redundant calls in the parallel version. To compile the code the following command is used:

```
⇒ mpicc sequential_pi.c -o sequential_pi.o
```

In order to run the code, use the mpirun command as follows:

```
⇒ mpirun -np 1 --hostfile /media/pkg/apini-scripts/apini_hostfile sequential_pi.o
```

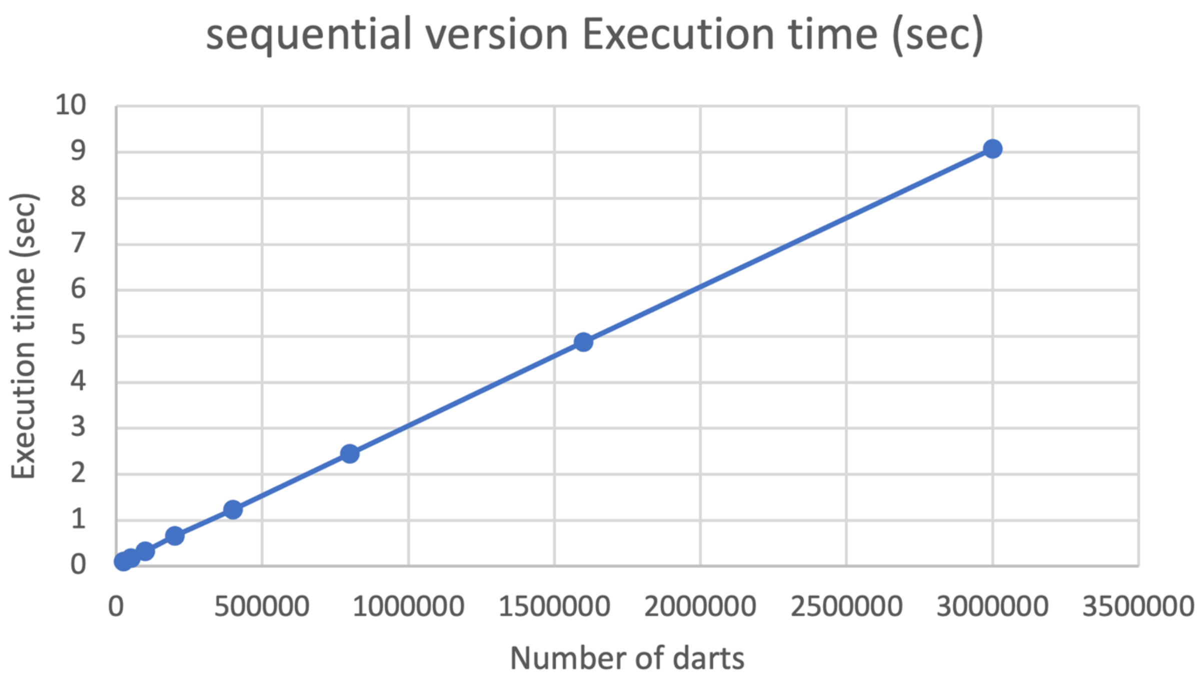
Note that since this code runs sequentially there is only need for one worker which is specified with -np 1. Although this code is in the sequential_pi.c file you can achieve the same result by running the parallel version with only one worker. You can do this by compiling the parallel code and running it as follows:

```
⇒ mpicc mpi_reduce.c -o mpi_reduce.o
```

```
⇒ mpirun -np 1 --hostfile /media/pkg/apini-scripts/apini_hostfile mpi_reduce.o
```

The execution times of running this program using different numbers of darts are shown in the following table and chart.

| Number of darts | Iterations (Rounds) | Execution time (sec) |
|---|---|---|
| 25000 | 100 | 0.101 |
| 50000 | 100 | 0.176 |
| 100000 | 100 | 0.325 |
| 200000 | 100 | 0.664 |
| 400000 | 100 | 1.234 |
| 800000 | 100 | 2.442 |
| 1600000 | 100 | 4.880 |
| 3000000 | 100 | 9.081 |



**B ) Execute the parallel version of the given program with varying number of workers (e.g. 2, 4, 6, etc.) and measure the parallel execution time in each case. Ideally, workers should be mapped to distinct nodes of the cluster.**

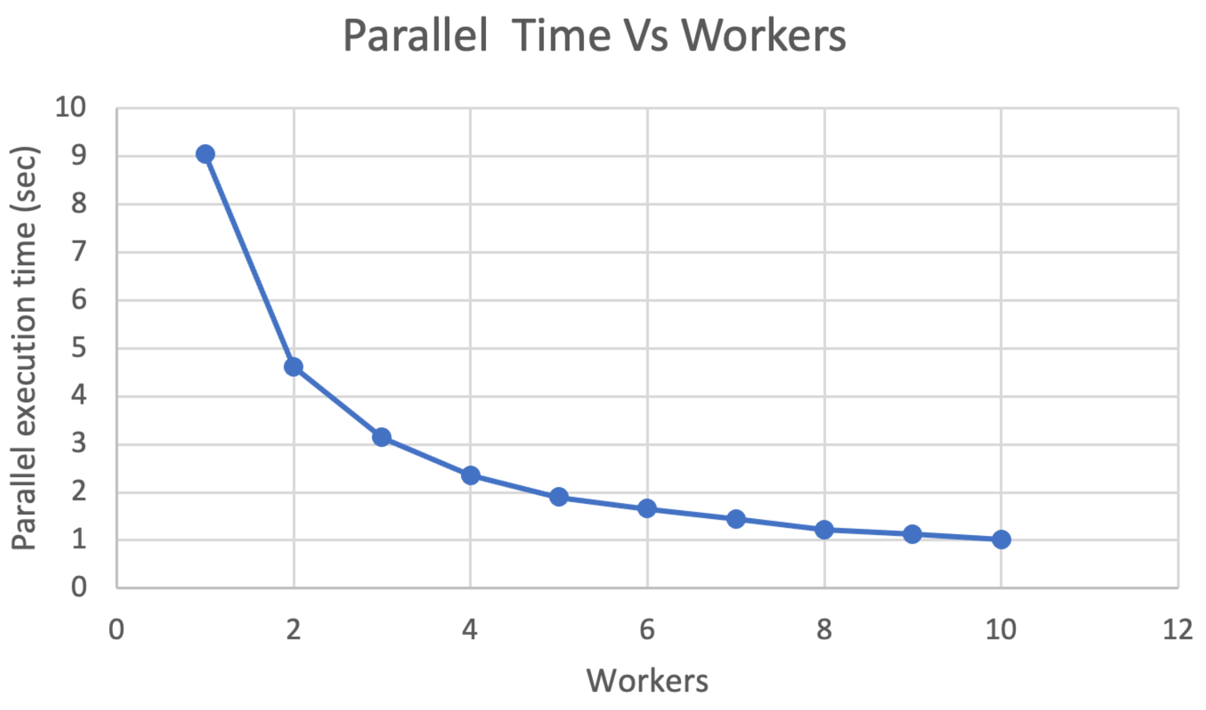
The parallel algorithm outlined in the tutorial can be found in the mpi_reduce.c file. We calculated the execution time for different numbers of workers using the following commands:

```
⇒ mpicc mpi_reduce.c -o mpi_reduce.o
```

```
⇒ mpirun -np 2 --hostfile /media/pkg/apini-scripts/apini_hostfile mpi_reduce.o
```

We increased the number of workers up to 10, as the cluster consists of only 5 nodes, and each node can support a maximum of 2 workers. Thus, 10 workers is the maximum configuration possible. The darts and Iterations (rounds) remained constant, and the tasks were divided among the workers. The parallel execution times for this version are summarized in the following table and chart.

| Number of darts | Iterations (Rounds) | Workers | Parallel execution time (sec) |
|---|---|---|---|
| 3000000 | 100 | 1 | 9.036 |
| 3000000 | 100 | 2 | 4.609 |
| 3000000 | 100 | 3 | 3.135 |
| 3000000 | 100 | 4 | 2.347 |
| 3000000 | 100 | 5 | 1.898 |
| 3000000 | 100 | 6 | 1.651 |
| 3000000 | 100 | 7 | 1.435 |
| 3000000 | 100 | 8 | 1.219 |
| 3000000 | 100 | 9 | 1.120 |
| 3000000 | 100 | 10 | 1.012 |



**C ) In the given program b), the master and the worker processes are statically spawned. Another way to implement the program is to first create the Master process, which dynamically spawns the workers through explicitly calling *MPI_Comm_Spawn*. Modify the given program accordingly, to use this dynamic**

**spawning mechanism available in the current versions of MPI and solve the PI calculation problem. Everything else remains the same as the original parallel version, except the dynamic spawning and communicators used between master and workers. Execute the parallel version of the given program with varying number of workers (e.g. 2, 4, 6, etc.) as before and measure the parallel execution time in each case. Important: for each run, spawn all the workers at one time, and not multiple times which can crash the system.**

In this version, we create the master process and pass the number of worker processes as a parameter. First, we compile the master and worker code by running the following commands in the terminal:

```
⇒ mpicc pi_master_code.c -o pi_master_code.o
```

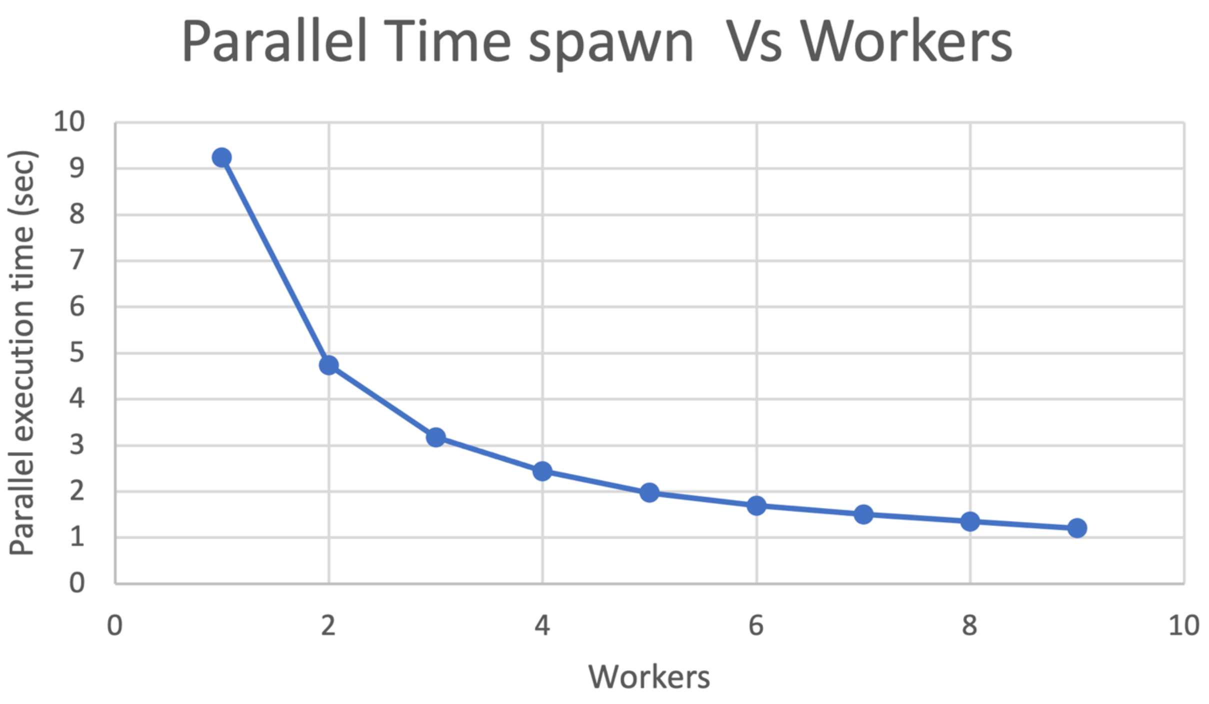
```
⇒ mpicc pi_slave_code.c -o pi_slave_code.o
```

Then we run the code by:

```
⇒ mpirun -np 1 --hostfile /media/pkg/apini-scripts/apini_hostfile pi_master_code.o 2
```

In the above command, 1 is the number of master processes, and 2 is the number of worker processes that is passed as an argument. Note that a master itself is a node, so for this code maximum number of workers is 9 (since we can have 10 nodes max). A table and chart for execution times for this version are shown below.

| Number of darts | Iterations (Rounds) | Workers | Parallel execution time |
|---|---|---|---|
| 3000000 | 100 | 1 | 9.249 |
| 3000000 | 100 | 2 | 4.743 |
| 3000000 | 100 | 3 | 3.180 |
| 3000000 | 100 | 4 | 2.445 |
| 3000000 | 100 | 5 | 1.975 |
| 3000000 | 100 | 6 | 1.693 |
| 3000000 | 100 | 7 | 1.509 |
| 3000000 | 100 | 8 | 1.347 |
| 3000000 | 100 | 9 | 1.204 |



D ) **Plot a *speedup* versus *number of workers* curve based on your experiments in (b) and (c) above. Explain any unusual behavior, e.g., slow down, sub-linear speedup, etc.**

As we could see in the charts it could be obtained that both graphs result in a sub-linear speedup. On the other hand, if we could increase the number of our workers, then we would see that the speed up will decrease after a certain number of workers. This behavior is explained by Amdahl's Law, which says the maximum speedup achievable through parallelization is limited by that part of the computation that can't be parallelized. For an infinite number of processors, the sequential part of the computation will always impose an upper limit on the achievable speedup. This limitation is further exacerbated by the communication overhead that increases as the number of processors grows.

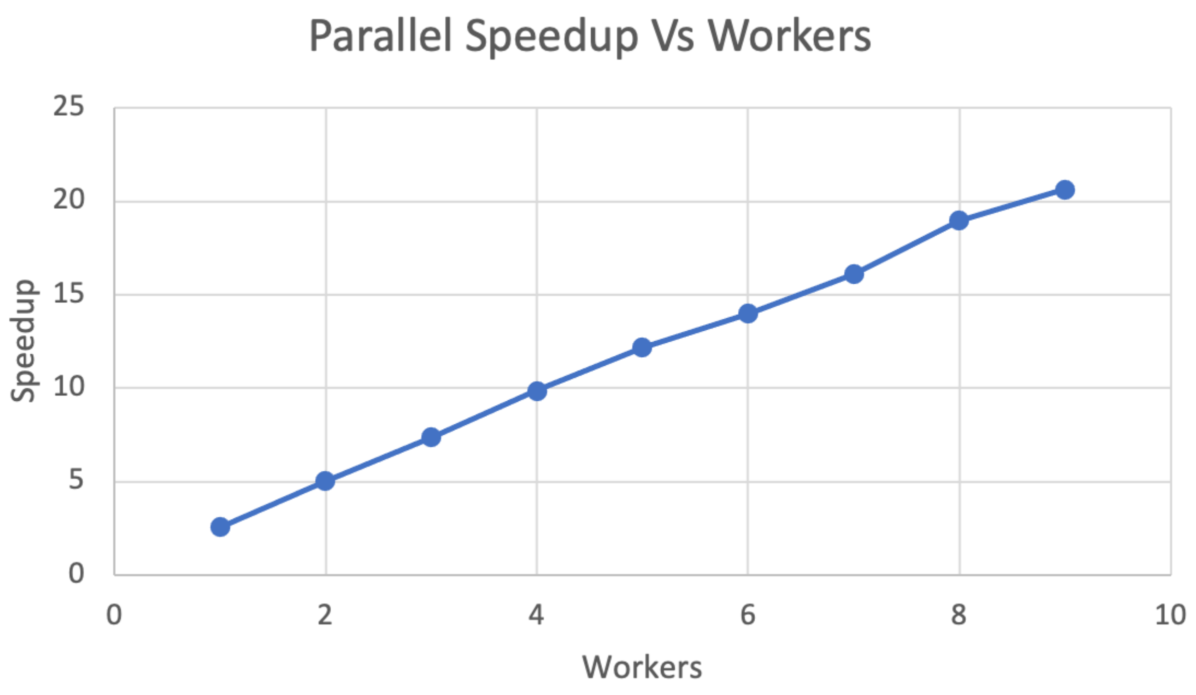
Besides, another reason can be the additional overhead created in the dynamic version, since during the spawning of the workers, more time is consumed compared to the static parallel version. In the case of the dynamic version, the time used to spawn the workers forms part of the total execution time which obviously increases the runtime measured. On the other hand, the static version already has all its processes ready from the beginning.

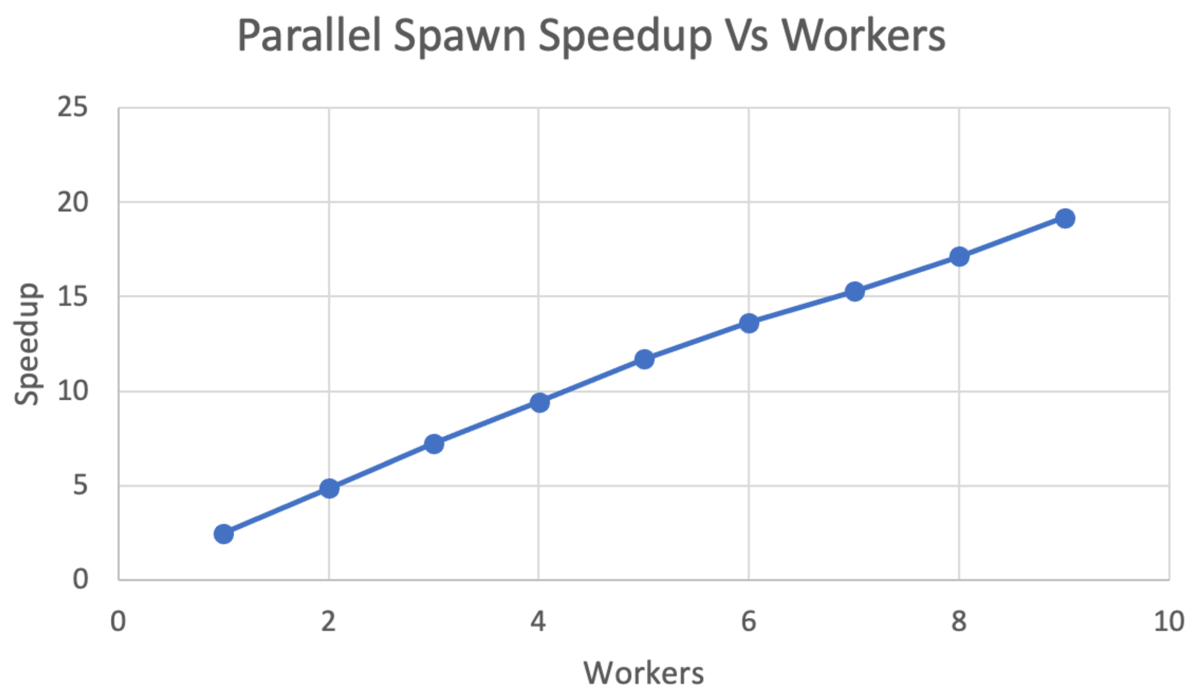
We have calculated the speedup using the following formula by the following command:

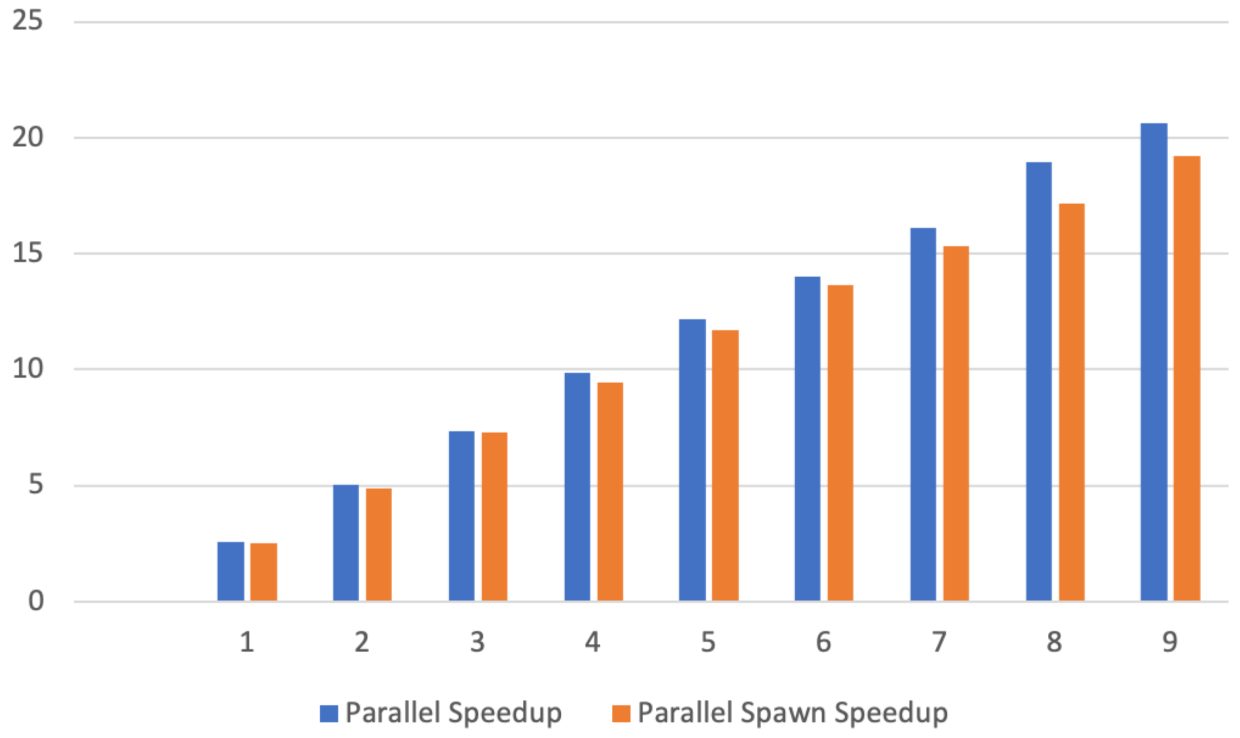
$$\text{Speedup} = \frac{\text{T sequential}}{\text{T parallel}}$$

A summary of the results is presented below. It is worth mentioning that the plot compares the speedup between both the standard parallel version and the dynamically spawned version.

| Number of darts | Iterations (Rounds) | Workers | Sequential Time | Parallel Time (sec) | Parallel Spawn Time (sec) | Parallel Speedup | Parallel Spawn v |
|---|---|---|---|---|---|---|---|
| 3000000 | 100 | 1 | 23.098 | 9.036 | 9.249 | 2.556 | 2.497 |
| | | 2 | | 4.609 | 4.743 | 5.011 | 4.87 |
| | | 3 | | 3.135 | 3.180 | 7.368 | 7.264 |
| | | 4 | | 2.347 | 2.445 | 9.841 | 9.447 |
| | | 5 | | 1.898 | 1.975 | 12.17 | 11.7 |
| | | 6 | | 1.651 | 1.693 | 13.99 | 13.643 |
| | | 7 | | 1.435 | 1.509 | 16.096 | 15.306 |
| | | 8 | | 1.219 | 1.347 | 18.948 | 17.148 |
| | | 9 | | 1.120 | 1.204 | 20.623 | 19.184 |




Parallel Spawn Speedup Vs Workers
Speedup
Workers
0
5
10
15
20
25
0
2
4
6
8
10


0
5
10
15
20
25
1
2
3
4
5
6
7
8
9
Parallel Speedup
Parallel Spawn Speedup