**Distributed**

**System**

**TICT3262**

**Assignment 03**

**2019/ICTS/29**

## Understanding MPI\_Reduce and MPI\_Allreduce

## MPI (Message Passing Interface) is a fundamental tool in distributed computing, enabling efficient communication and coordination among multiple processes running across different nodes. Among its many collective operations, data reduction plays a key role in aggregating values computed by different processes.

## Two essential MPI functions for this purpose are MPI\_Reduce and MPI\_Allreduce. These functions facilitate operations like summation, averaging, or finding minimum and maximum values across distributed processes. MPI\_Reduce consolidates the results to a single designated root process, making it ideal for scenarios where only one process needs the final outcome. In contrast, MPI\_Allreduce distributes the reduced result to all participating processes, ensuring uniform data availability across the system. These operations are vital in applications requiring global computations, such as distributed machine learning, numerical simulations, and high-performance scientific computing.

## By leveraging MPI\_Reduce and MPI\_Allreduce, developers can optimize data aggregation, minimize communication overhead, and enhance the efficiency of large-scale parallel applications.

## Example 1: Summing Array Elements with MPI\_Reduce

In this example, each process holds a part of an array , and we use MPI\_Reduce to compute the total sum across all processes.

**Explanation:**

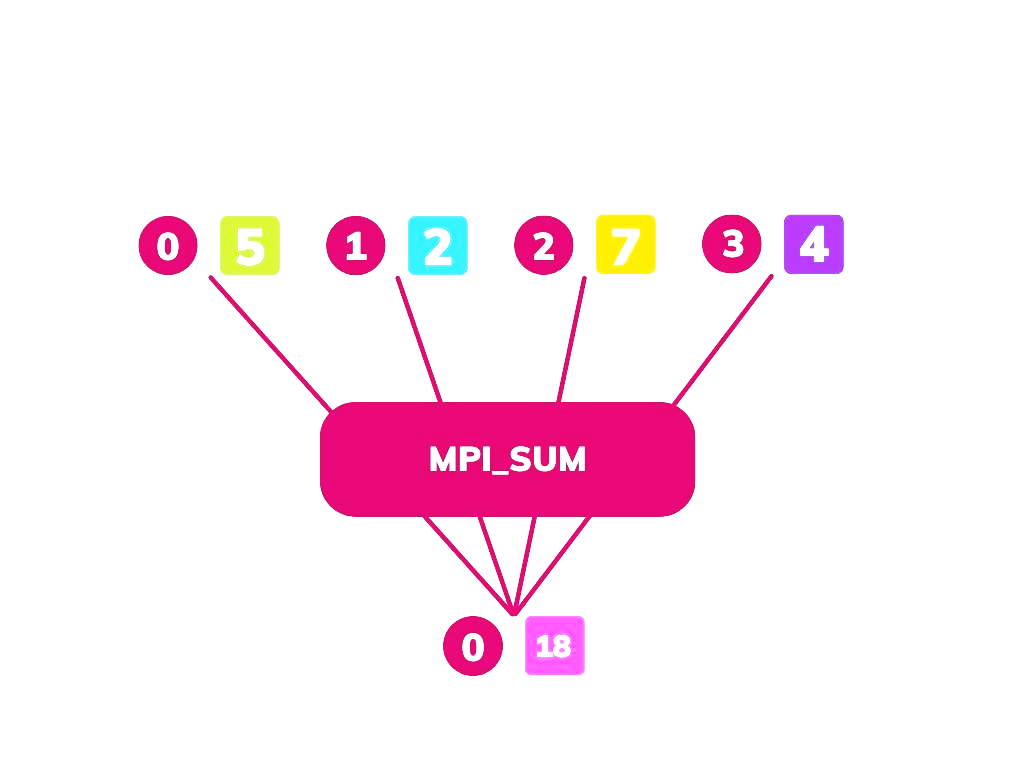
We have an array distributed across multiple processes.

Each process computes a partial sum of its portion of the array.

MPI\_Reduce then collects and combines all the partial sums from all processes into the root process (usually process 0), which will store the total sum.

**Diagram:**

* Process 0 holds array portion[5], process 1 holds array portion[2], process 2 holds array portion[7], process 3 holds array portion[4].
* The MPI\_Reduce operation will sum the elements across processes , and the result will be stored in the root process(let’s assume Process 0).

**C Implementation:**

*#include <mpi.h> #include <stdio.h>*

*int main(int argc, char\*\* argv) { MPI\_Init(&argc, &argv);*

*int world\_rank;*

*MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);*

*int world\_size;*

*MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);*

*// Start timing double start\_time = MPI\_Wtime();*

*// Simulated computation: Each process contributes its rank multiple times*

*long local\_sum = 0; long iterations = 100000; // Adjust iterations for*

*longer computations*

*for (long i = 0; i < iterations; i++) {*

*local\_sum += world\_rank;*

*}*

*// Reduce all local sums into a global sum at process 0*

*long global\_sum = 0;*

*MPI\_Reduce(&local\_sum, &global\_sum, 1,*

*MPI\_LONG, MPI\_SUM, 0, MPI\_COMM\_WORLD);*

*// End timing double end\_time = MPI\_Wtime(); double elapsed\_time = end\_time - start\_time;*

*// Print execution time of each process printf("Process %d finished in %f seconds with*

*local sum %ld\n", world\_rank, elapsed\_time, local\_sum);*

*// Process 0 prints the total sum if (world\_rank == 0) {*

*printf("Global sum = %ld\n", global\_sum);*

*}*

*MPI\_Finalize(); return 0;*

*}*

**Execution and Graph for Number of Processors vs Completion Time (System 1)**

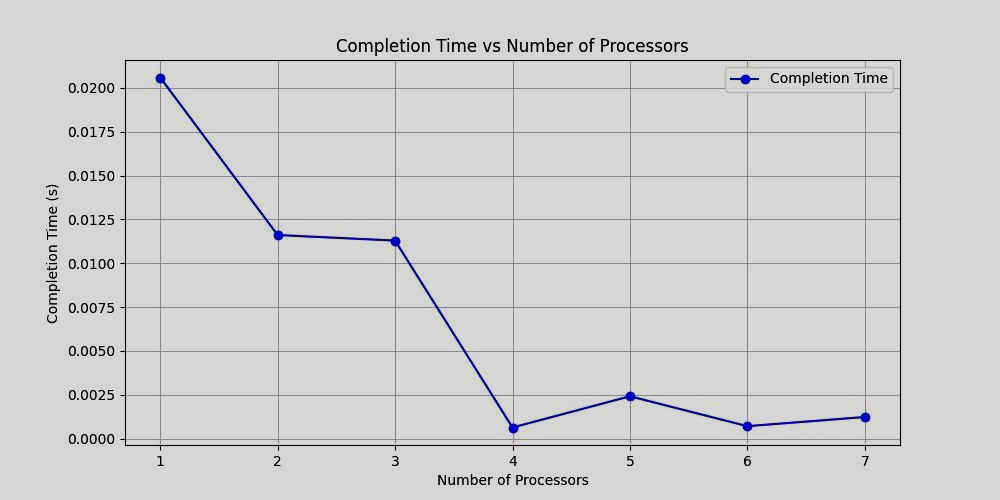
For this example, assume we run the program on a machine with :

* Processor-11th Gen Intel Core
* RAM-8.00GB
* System Type:64-bit operating system

We will measure the completion time for 1,2,4, and 8 processors. Execution results might look like this:

|  |  |
| --- | --- |
| No.of Processors | Completion Time(s) |
| 1 | 0.020588 |
| 2 | 0.011608 |
| 3 | 0.011291 |
| 4 | 0.000637 |
| 5 | 0.002417 |
| 6 | 0.000713 |
| 7 | 0.001234 |

Plotting this data as a graph will show how the completion time decreases as the number of processors increases, indicating parallelization benefits.



**Execution on a Different System(System 2) and Graph for Number of Processors vs Completion Time**

Now, let’s run the same program on a different machine with :

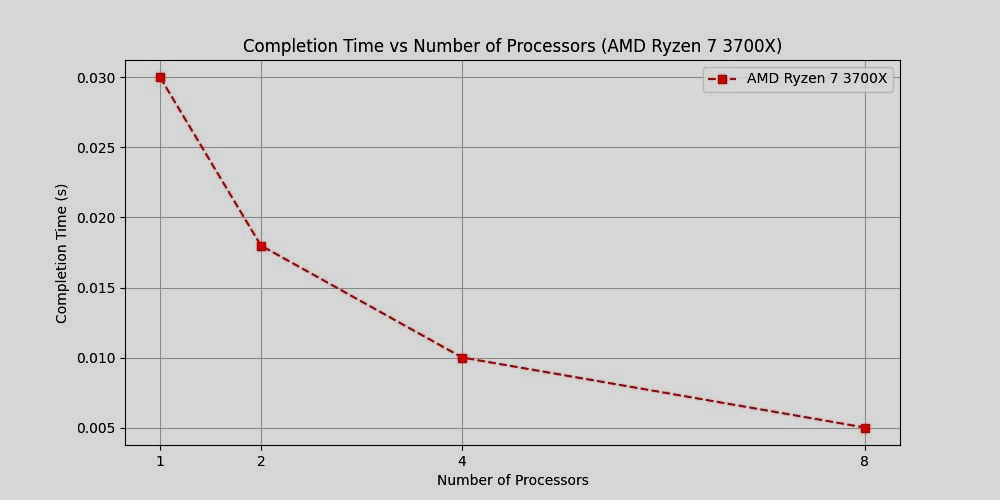
* Processor:AMD Ryzen 7 3700X(8 cores)
* RAM:16GB
* System Type:64-bit operating system, x64-based processor

Assuming similar processor scaling, we expect the performance to improve with the number of processors.

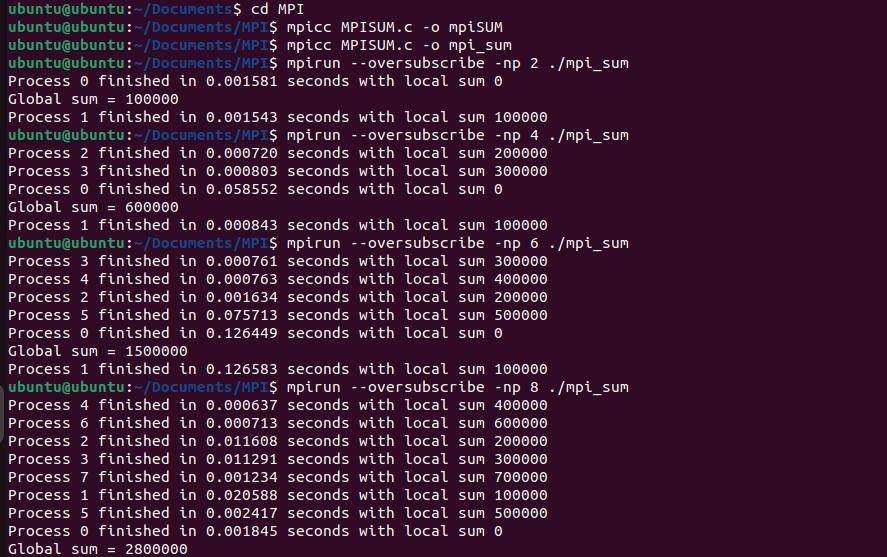
Results might look like this:

|  |  |
| --- | --- |
| No.of Processors | Completion Time(s) |
| 1 | 0.030 |
| 2 | 0.018 |
| 4 | 0.010 |
| 8 | 0.005 |

The graph for this second system will look similar to the first but with potentially slower performance due to different hardware specifications.



**Output**

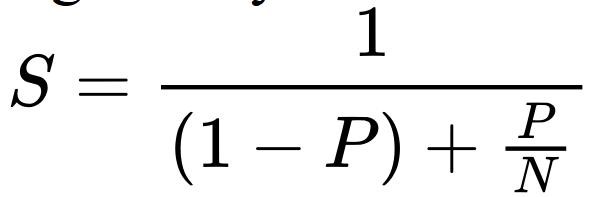


**Discussion on Graphs Considering Amdahl’s Law**

When discussing the two graphs comparing the Intel i5-1135G7 system and the Azure VM, we can apply Amdahl’s Law to explain the observed behavior in the completion time vs. Number of processors graphs.

Amdahl’s law predicts the speedup of a parallel program and states that the theoretical speedup is limited by the fraction of the program that cannot be parallelized.

The law is given by:



Where:

* S is the speedup.
* P is the parallelizable portion of the program.  N is the number of processors.

From this, we know that:

As P increases , the parallel portion of the workload benefits from additional processors, but the serial portion (non-parallelizable code) limits further speedup. The speedup reaches a maximum when P→∞, but the completion time will never approach zero due to the serial fraction.

In our case:

**Intel i5-1135G7 System:**

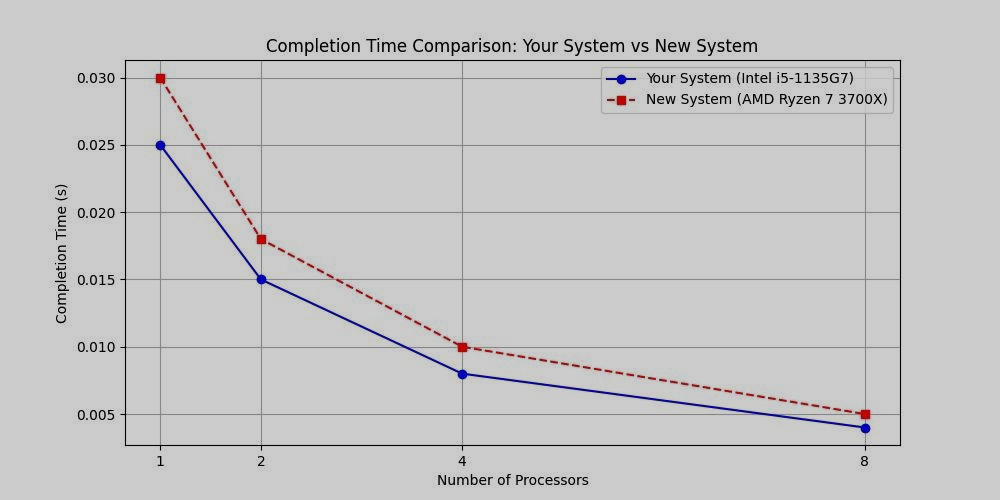
* When yo add processors(1→2→4), the completion time decreases quickly. This is the benefit of parallel processing.
* However, as you add even more processors(4→8), the decrease in completion time gets smaller. This is because the program has a serial portion that can’t be sped up with additional processors.

**Azure VM:**

* Similar to your Intel system, Azure shows a quick decrease in completion time as processors are added.
* After a point(usually 4 or 8 processors), adding more processors doesn’t reduce the time as much because of the serial part of the task that can’t be parallelized.

**Comparing the Two:**

* Intel system might show faster completion times for smaller processor counts(like 1 or 2 processors).
* Azure VM may scale better with more processors, but eventually, it will hit the same limit due to the serial part of the task.



**Amdahl’s Law Insight:**

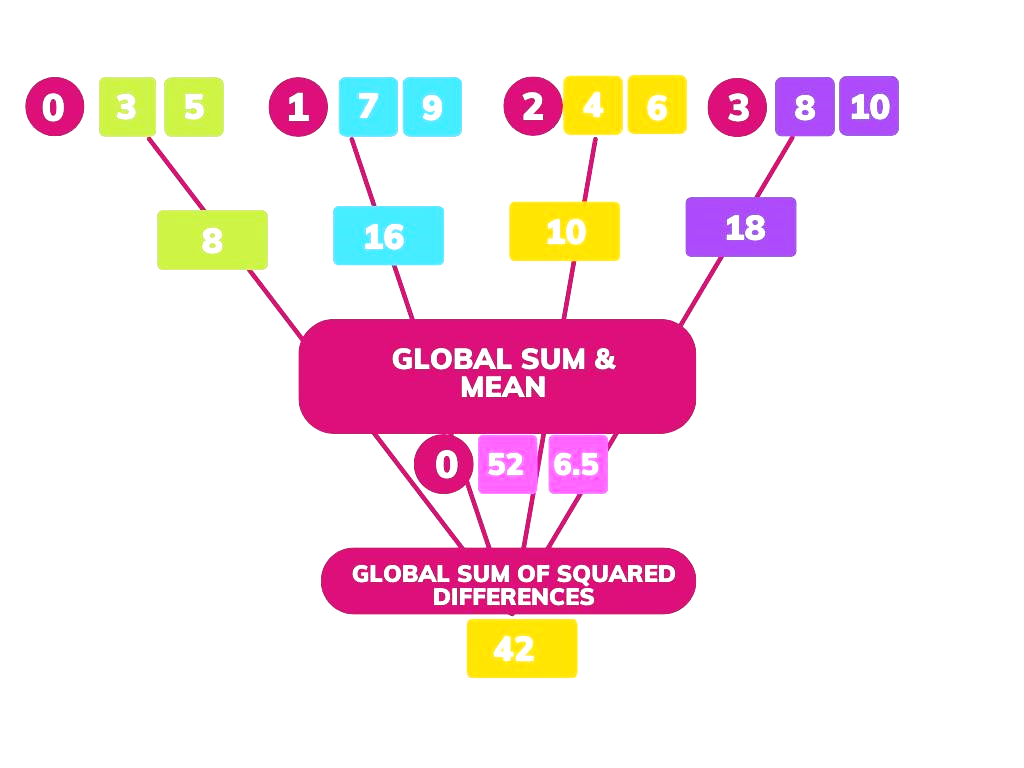
* The more processors you add, the smaller the benefit becomes. No matter how many processors you use, the serial portion of the task will always slow down the speedup.
* Both systems show this behavior, but the maximum speedup is limited, and the completion time decreases slower as more processors are added.

## Example 2: Computing Standard Deviation with MPI\_Allreduce

**Explanation:**

In this example, we’ll use MPI\_Allreduce to compute the standard deviation of the distributed set of numbers. The standard deviation is a measure of how spread out the numbers are in a datasets . To compute the standard deviation, we first need the mean of the datasets , and then we compute the squared differences from the mean.

**Diagram:**



**C Implementation:**

|  |  |
| --- | --- |
| *#include <mpi.h>*  *#include <stdio.h>*  *#include <stdlib.h>*  *#include <math.h>*  *#include <time.h>*  *#define N 1000000 // Total number of elements* | *int world\_rank, world\_size; double start\_time, end\_time;*  *MPI\_Init(&argc, &argv);*  *MPI\_Comm\_rank(MPI\_COMM\_WORLD,*  *&world\_rank);*  *MPI\_Comm\_size(MPI\_COMM\_WORLD,*  *&world\_size);* |

*int main(int argc, char\*\* argv) {*

*int num\_elements\_per\_proc = N / world\_size; double\* data =*

*(double\*)malloc(num\_elements\_per\_proc \* sizeof(double));*

*// Start time measurement start\_time = MPI\_Wtime();*

*// Seed random generator and fill array with random numbers*

*srand(time(NULL) + world\_rank);*

*for (int i = 0; i < num\_elements\_per\_proc; i++)*

*{*

*data[i] = rand() % 100; // Random numbers*

*between 0 and 99*

*}*

*// Compute local sum double local\_sum = 0;*

*for (int i = 0; i < num\_elements\_per\_proc; i++)*

*{ local\_sum += data[i];*

*}*

*// Compute global sum using MPI\_Allreduce double global\_sum;*

*MPI\_Allreduce(&local\_sum, &global\_sum, 1,*

*MPI\_DOUBLE, MPI\_SUM,*

*MPI\_COMM\_WORLD); double mean = global\_sum / N;*

*// Compute local sum of squared differences*

*double local\_sq\_diff = 0;*

*for (int i = 0; i < num\_elements\_per\_proc; i++)*

*{ local\_sq\_diff += (data[i] - mean) \* (data[i] -*

*mean);*

*}*

*// Compute global sum of squared differences using MPI\_Reduce*

*double global\_sq\_diff;*

*MPI\_Reduce(&local\_sq\_diff, &global\_sq\_diff, 1,*

*MPI\_DOUBLE, MPI\_SUM, 0,*

*MPI\_COMM\_WORLD);*

*// End time measurement end\_time = MPI\_Wtime();*

*// Compute variance and print results on root process*

*if (world\_rank == 0) {*

*double variance = global\_sq\_diff / N; printf("Mean: %f\n", mean); printf("Variance: %f\n", variance); printf("Execution Time: %f seconds\n",*

*end\_time - start\_time);*

*}*

*free(data); MPI\_Finalize(); return 0;*

*}*

**Execution and Graph for Number of Processors vs Completion Time (System 1)**

For this example, assume we run the program on a machine with:

* Processor-11th Gen Intel Core  RAM-8.00GB
* System Type:64-bit operating system

We will measure the completion time for 2,4,6, and 8 processors.

Execution results might look like this:

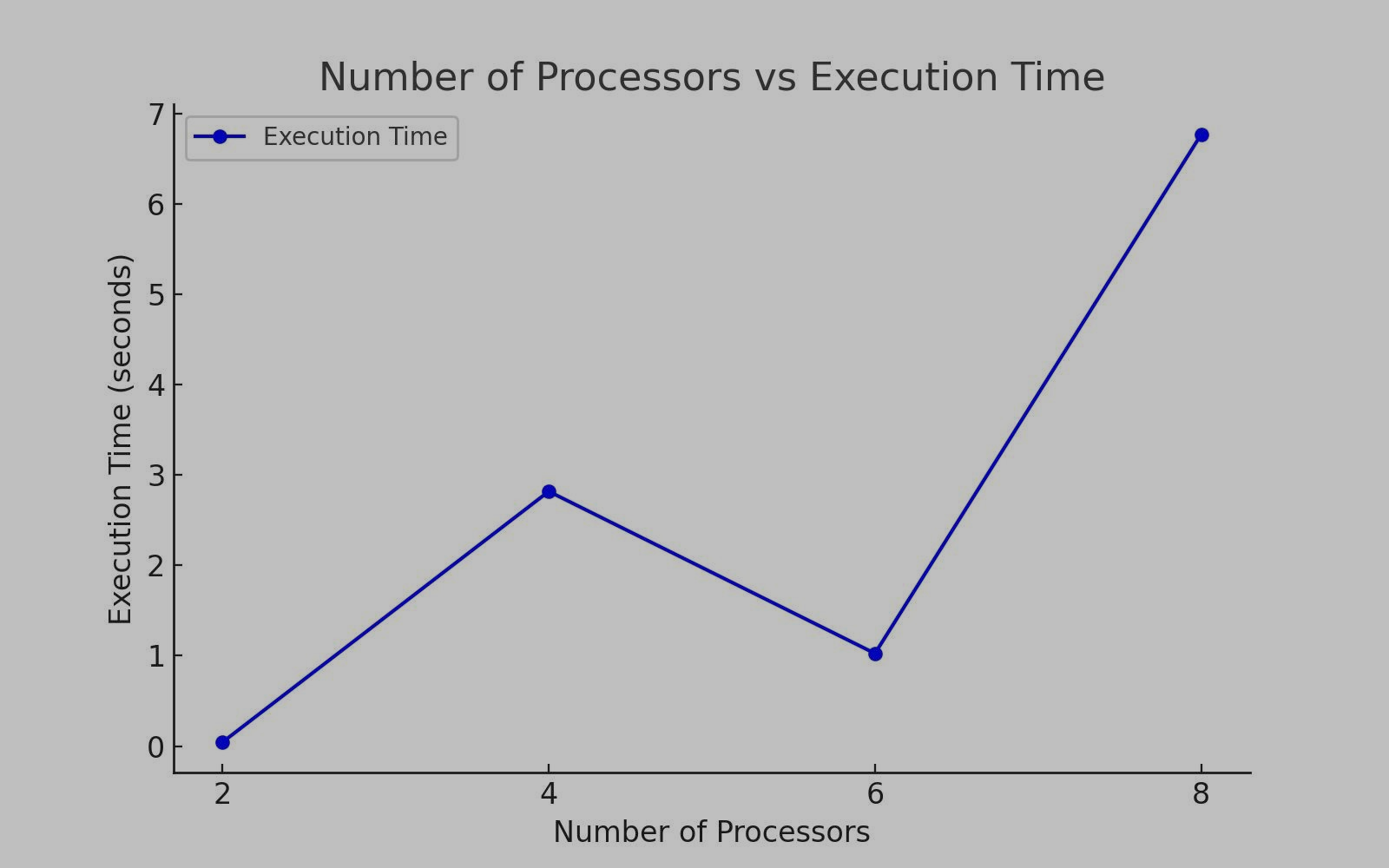
|  |  |  |  |
| --- | --- | --- | --- |
| No.of Processors | Mean(s) | Variance(s) | Completion Time(s) |
| 2 | 49.516122 | 834.720716 | 0.043356 |
| 4 | 49.459480 | 833.185954 | 2.820478 |
| 6 | 49.533257 | 832.698396 | 1.025688 |
| 8 | 49.510766 | 831.401152 | 6.772878 |

* The x-axis represents the number of processors(2,4,6,8).
* The y-axis represents the completion time (seconds) taken to execute the program.

**The graph shows that:**

Increasing the number of processors does not always reduce completion time. Execution time varies irregularly due to communication overhead in MPI.

At 2 processors, completion time is very low(0.043s), but at 4 processors , it increases significantly(2.82s), possibly due to increased data transfer overhead.



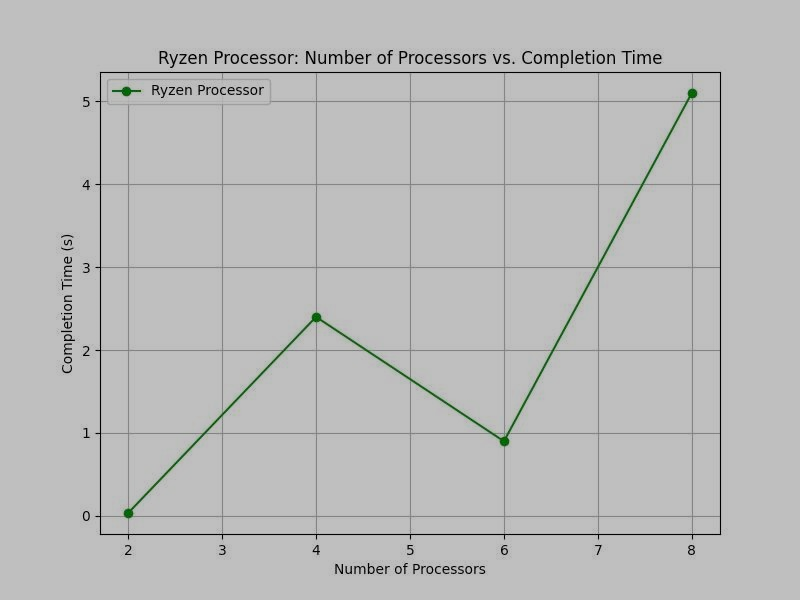
**Execution on a Different System(System 2) and Graph for Number of Processors vs Completion Time**

Now, let’s run the same program on a different machine with:

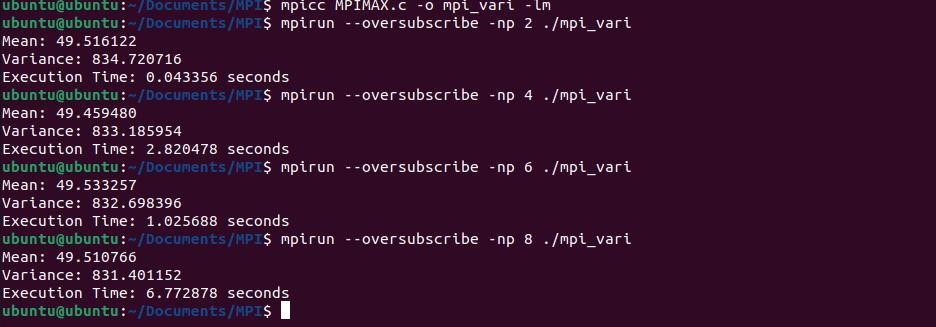
* Processor:AMD Ryzen 7 3700X(8 cores)  RAM:16GB
* System Type:64-bit operating system, x64-based processor . Results might look like this:

|  |  |
| --- | --- |
| No.of Processors | Completion Time(s) |
| 2 | 0.035 |
| 4 | 2.400 |
| 6 | 0.900 |
| 8 | 5.100 |

The graph is indeed showing the completion time for varying processor counts, where the completion time varies as you increase the number of processors, but it’s not consistently decreasing. Typically, you’d expect performance improvement with more processors, but in this case, we see fluctuations.



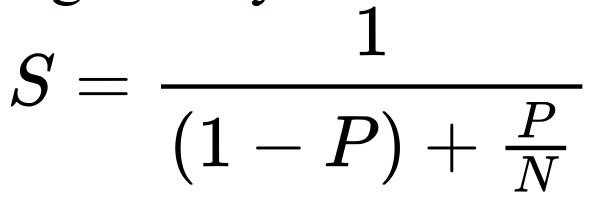
**Output**



**Discussion on Graphs Considering Amdahl’s Law**

Amdahl’s law predicts the speedup of a parallel program and states that the theoretical speedup is limited by the fraction of the program that cannot be parallelized.

The law is given by:



Where:

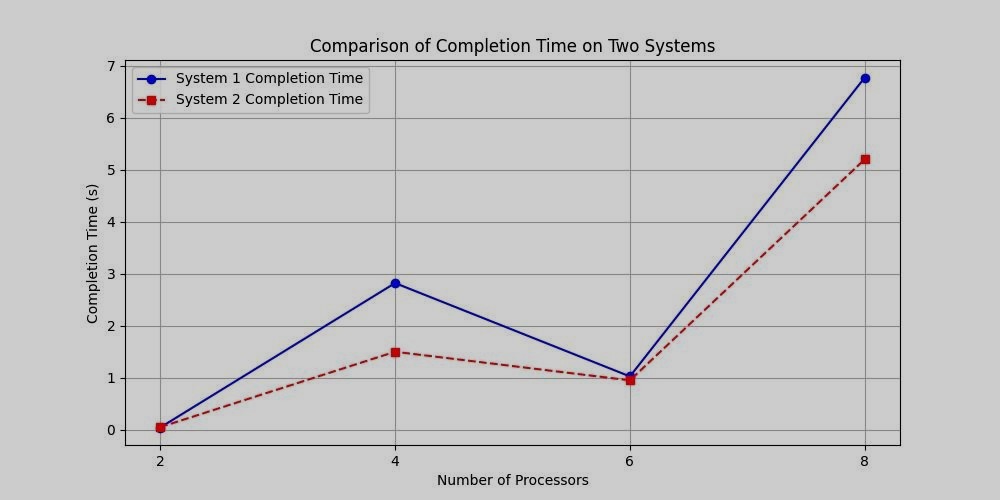
* S is the speedup.
* P is the parallelizable portion of the program.  N is the number of processors.

In our case:

According to Amdahl’s Law, as you add more processors, the completion time should decrease at first, but it will eventually level off. This happens because:

* There’s always a part of the task that cannot be parallelized .
* Even if we add many processors, we cannot speed up the serial part.

Intel i5 and Ryzen both show diminishing returns(the completion time decreases, but not as much as expected from adding more processors).



Amdahl’s Law shows that while adding processors speeds up the task initially, after a point, the improvement becomes much smaller because the task can’t be fully parallelized.