**CS 426–Parallel Computing Project 1 Report**

**Part A: Find the Max**

**Serial**



**MPI V1**

1 Process:



2 Processes:



3 Processes:



4 Processes:



8 Processes:

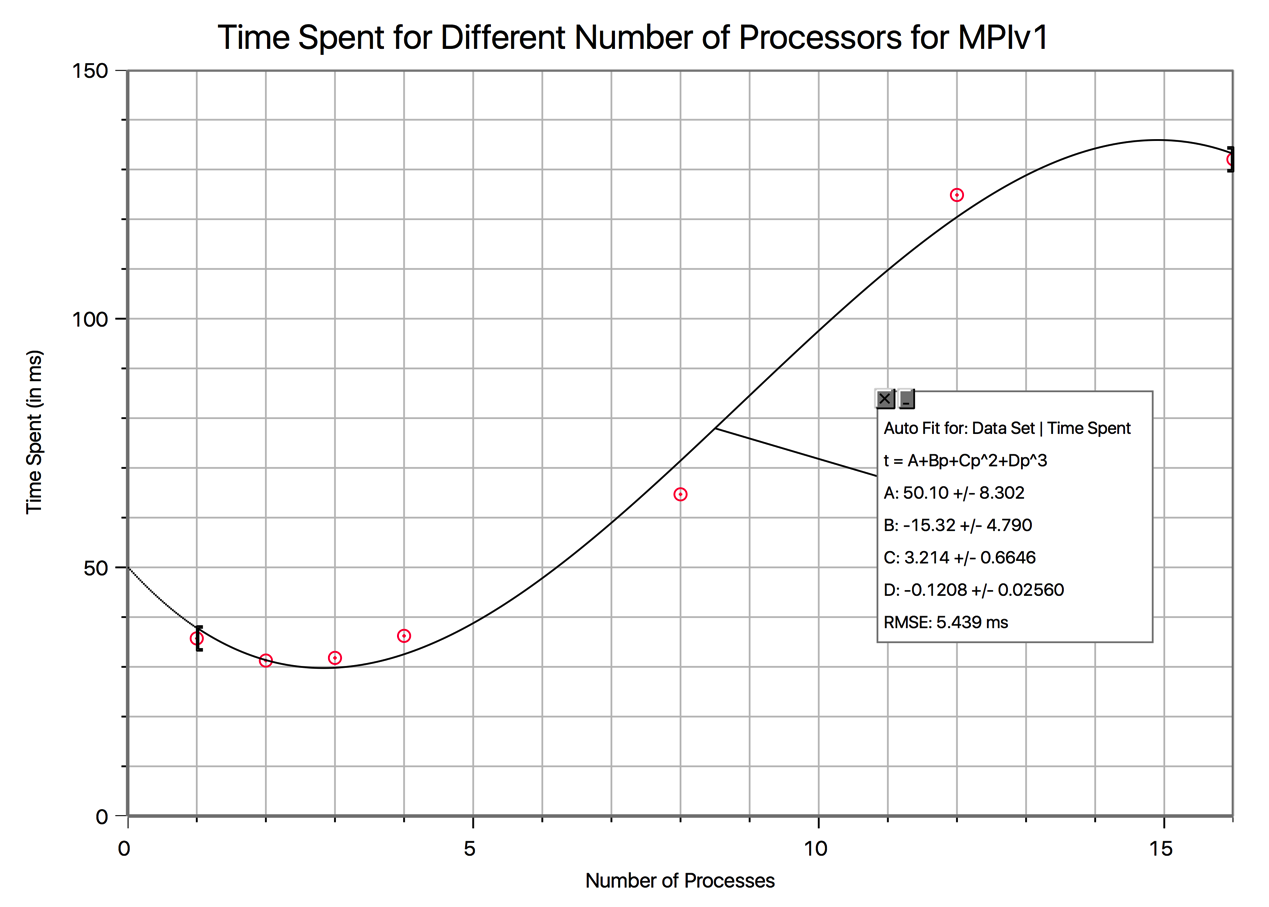


12 Processes:



16 Processes:



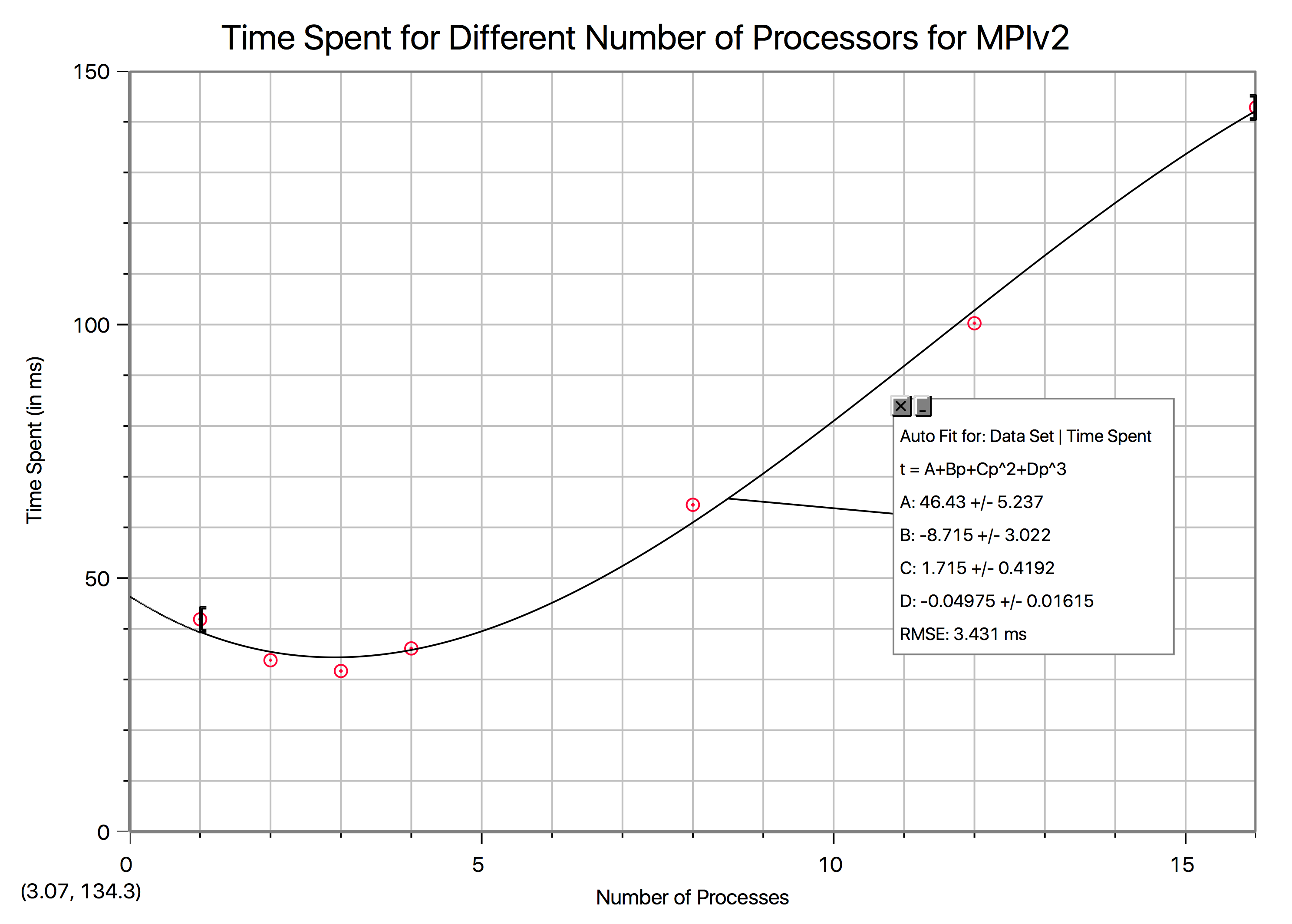


In this part of the project, I dispatched the lines of input as equal as possible by distributing the last remainder part of the lines to previous processors. To do this, I prepared an algorithm that checks whether if the rank of the current processor is smaller than the remainder. If it is, one more the normal size of the portion is given to that processor. This way, the remainder part is distributed equally and the last part is not very small than the previous parts. Besides sending the array parts to the processors, I also sent the array sizes to them. The child processes sent their own local maxima to the master process and the master process found the overall maximum from the child maxima and its own maximum.

As seen in the images, the fastest implementation of this part was obviously executing it serially without any usage of MPI, because making the computer do calculations this simple, does not require a lot of processing and having multiple processes work spends more time to communicate than to make local computations. After the serial, the smallest elapsed time is with two processes instead of one. This shows that there’s an optimal process number, but not less or more. Increasing the process number more actually caused an increase in the execution time, since (first of all I was oversubscribing after 4 processes) there is an overhead of communication between child processes and the master process and memory allocation.

**MPI V2**

|  |  |
| --- | --- |
| Number of Processes | Elapsed Time (ms) |
| 1 | 41.863918 |
| 2 | 33.787966 |
| 3 | 31.675100 |
| 4 | 36.128044 |
| 8 | 64.455032 |
| 12 | 100.25692 |
| 16 | 142.830133 |



In this part, different from V1, I used MPI\_Allreduce function to ensure that all of the processes have the mean while the mean is being calculated through this function. This function caused the 3-process case to win over all of the other number of processors. Increasing the process number caused a higher increase in the overhead, therefore execution time. Also, the function caused the execution times to increase, but the optimal execution time is the same with MPIv1.

**Part B: Image Smoothing**

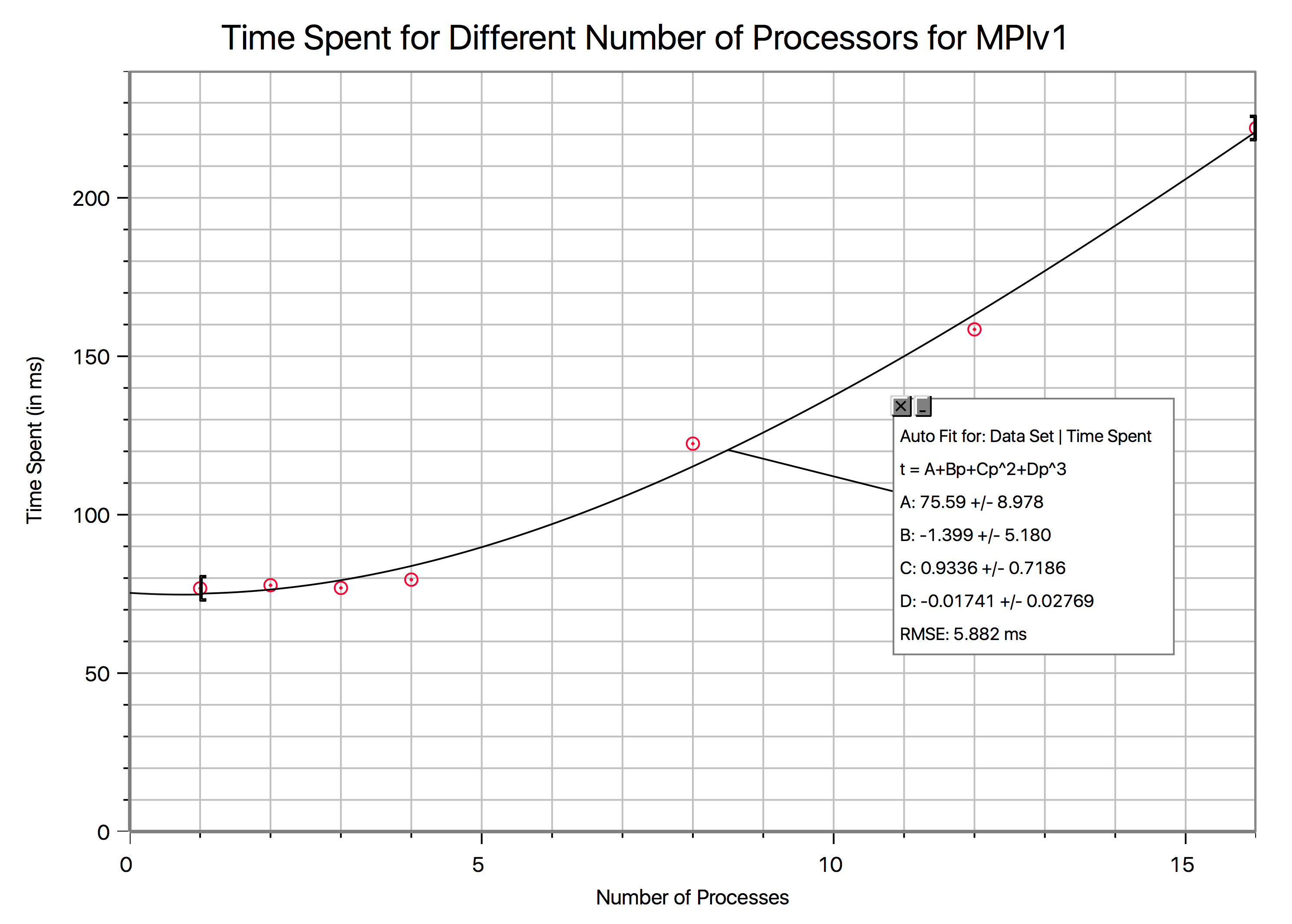
**Serial**

Smooth Image (Output of smooth-serial.c)

**MPI V1**

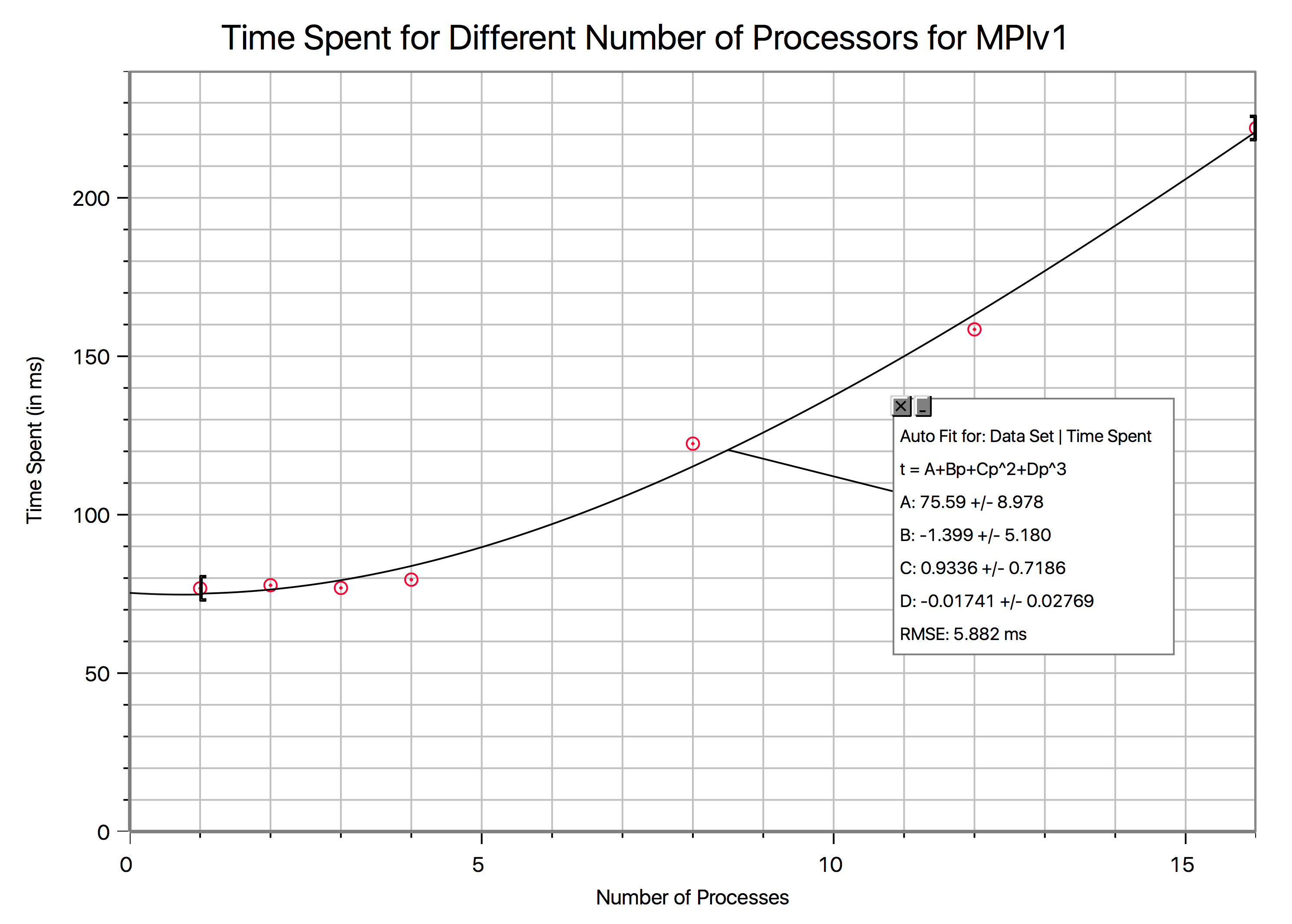
|  |  |
| --- | --- |
| Number of Processes | Elapsed Time (ms) |
| 1 | 76.771021 |
| 2 | 77.746868 |
| 3 | 76.895952 |
| 4 | 79.524040 |
| 8 | 122.434855 |
| 12 | 158.497810 |
| 16 | 222.032070 |



In this part, I dispatched the rows of the image by using the algorithm that I did for the first part. This way, the rows of the image were distributed as equal as possible to processes. A design choice that I made was to send the arrays to processes as one-dimensional (flattened two-dimensional) arrays, since MPI\_Send and MPI\_Receive do not support sending 2 dimensional arrays. I made each process filter its own image portion by taking care of the overlapping indices by having three cases when choosing the local image sizes and calculating local outputs. The first row, middle rows and last row were the three cases. For the first row case, I sent one row more from the second processes local image, for the middle rows, I sent one row from previous local image and one row more from next local image’s first row and for the last case, I sent one row more from the previous local image. This way, the edge cases were handled to overlap pixels. As seen above from the table and the graph, the optimal elapsed time was when I had two processes to do the processing of the image. However, after having more than 4 processes, the elapsed time increased significantly, emphasizing the overhead and showing the consequences of oversubscribing.

**MPI V2**

|  |  |
| --- | --- |
| Number of Processes | Elapsed Time (ms) |
| 1 | 76.771021 |
| 2 | 77.746868 |
| 3 | 76.895952 |
| 4 | 79.524040 |
| 8 | 122.434855 |
| 12 | 158.497810 |
| 16 | 222.032070 |



In this part, I dispatched the rows of the image by using the algorithm that I did for the first part. This way, the rows of the image were distributed as equal as possible to processes. A design choice that I made was to send the arrays to processes as one-dimensional (flattened two-dimensional) arrays, since MPI\_Send and MPI\_Receive do not support sending 2 dimensional arrays. I made each process filter its own image portion by taking care of the overlapping indices by having three cases when choosing the local image sizes and calculating local outputs. The first row, middle rows and last row were the three cases. For the first row case, I sent one row more from the second processes local image, for the middle rows, I sent one row from previous local image and one row more from next local image’s first row and for the last case, I sent one row more from the previous local image. This way, the edge cases were handled to overlap pixels. As seen above from the table and the graph, the optimal elapsed time was when I had two processes to do the processing of the image. However, after having more than 4 processes, the elapsed time increased significantly, emphasizing the overhead and showing the consequences of oversubscribing.