Suhas Kamath

DS 221: INTRODUCTION TO SCALABLE SYSTEMS

November 14, 2025

Parallel Programming - OpenMP Assignment

Theory & Methodology

K-means clustering is an iterative unsupervised learning algorithm that segregates a set of data points into K clusters. [1] [2] The diagram below shows how the algorithm iterates:

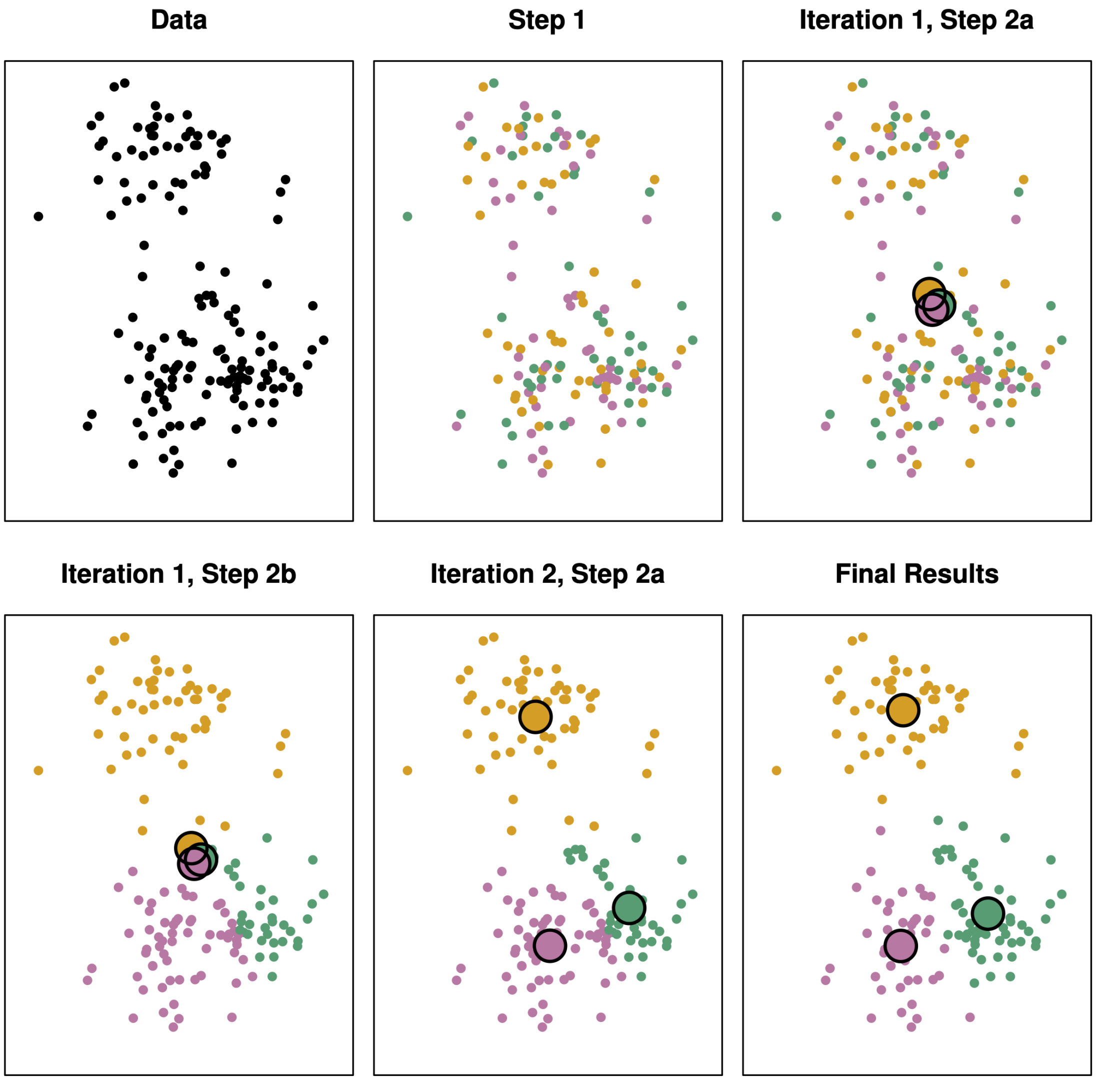

Figure 1. K-means iterative clustering algorithm. [3]

The basic (sequential) algorithm for K-means clustering is as follows:

1) **Initialization:**

- Choose the number of clusters, K. We use $K = 20$, as specified in the question.

- Initialize the K cluster centroids, $C_1, C_2, \ldots, C_K$. We simply use the first K data points as the centroids for simplicity.[1]

2) **Assignment Step (Cluster Assignment):**
   - For each data point P_i in the dataset:
   - Calculate its distance to each of the K centroids. We use Squared Euclidean distance:

$$d(P_i, C_j) = \|P_i - C_j\|^2$$

   - Assign the data point P_i to the cluster of its nearest centroid.

3) **Update Step (Centroid Recalculation):**
   - For each cluster j (from $j = 1$ to K):
   - Recalculate the position of its centroid C_j to be the arithmetic mean of all data points P_i currently assigned to it.

4) **Check for Convergence:**
   - After the update step, we check if the algorithm has converged. Convergence is reached if the cluster assignments (the sets S_j) did not change from the previous iteration.[2]
   - If not converged, return to Step 3 (Assignment Step).

We can parallelise a few aspects of this algorithm:

1) The assignment step, wherein we assign each point to its nearest centroid, can be fully parallelised.
2) The accumulation step, where the per-thread local sums are merged into global, can be parallelised.
3) To check for convergence, we calculate the distance of each new centroid from its previous position. This can also be parallelised using OpenMP's `reduce` feature.

Experimental Setup

Both the sequential and parallel programs were executed on the teaching cluster. The teaching cluster has the following specifications:

- **CPU:** Intel Xeon Gold 5318Y 24-core 2.1GHz
- **RAM:** 128 GB RAM
- **Storage (HDF5):** 2×12 TB SATA HDD, 1.92TB PCIe NVME M.2 Enterprise SSD
- **Operating System:** Cent OS

[1]We can also use `kmeans++` algorithm, but such an algorithm will only increase the complexity. Our aim is check the time saved by using OpenMP, not obtain a perfect clustering algorithm. [4]

[2]To accommodate for floating point errors, we assume convergence when the change in centroid positions between iterations is below a small threshold, ϵ. For the sake of this experiment, we take $\epsilon = 0.0001$.

- **Job scheduling software:** SLURM

The program was executed parallelly using OpenMP. For testing the benefits of OpenMP $1, 4, 8, 16, 32$ threads were used. When using 1 thread, the OpenMP program behaves exactly like a sequential program. To obtain more accurate readings, each experiment was repeated 5 times, and the execution time recorded.

The execution times (of each experiment) is then fed into Python, where the average corresponding to each thread count is calculated and output. The Python program also produces a graph using `matplotlib`.

Results

As mentioned above, we calculated the average time over 5 experiments per number of threads. These averages are displayed in the below table:

Threads	**Average Time with Dynamic (ms)**	**Average Time with Static (ms)**
1	54.375	27.880
4	95.743	8.482
8	92.402	5.635
16	102.869	4.125
32	119.260	3.651

The graph obtained by plotting the averages is as follows:

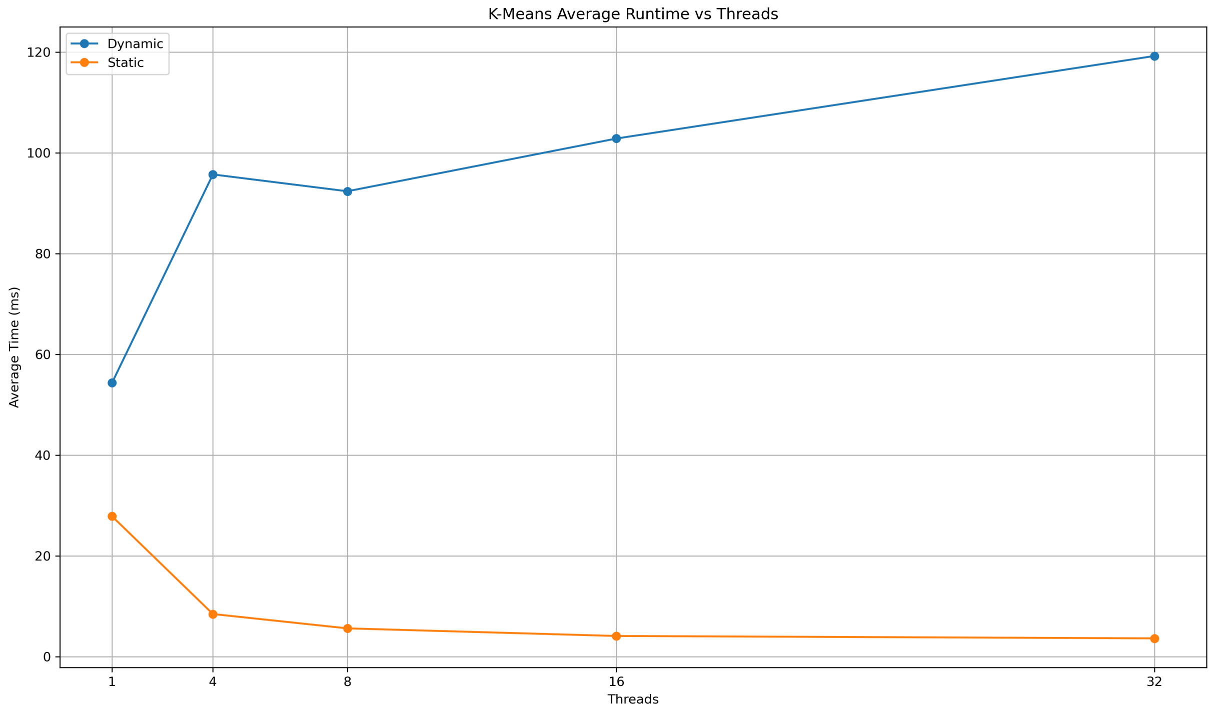

Figure 2. The graph depicting the average execution time against the number of threads.

The speed-up figures are as follows [3]:

Threads	Average Speedup with Dynamic	Average Speedup with Static
1	0.513	1.000
4	0.291	3.287
8	0.302	4.948
16	0.271	6.759
32	0.234	7.636

The plot is showing the speedups is as follows:

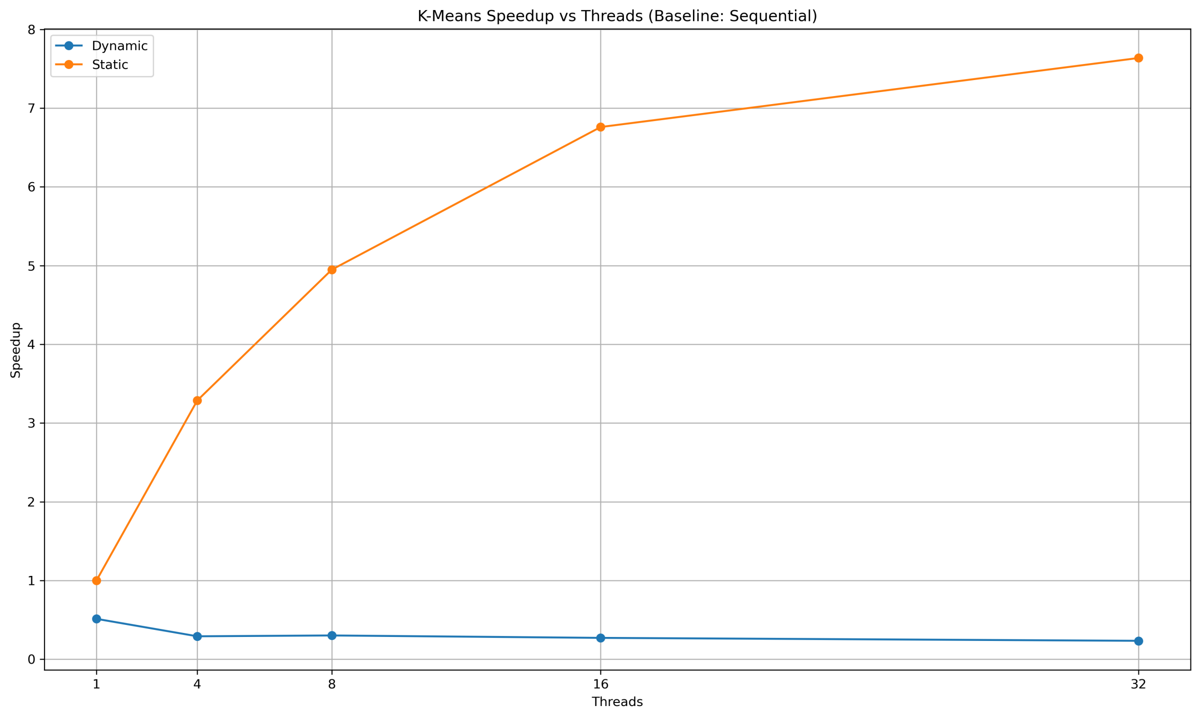

FIGURE 3. The graph depicting the average speed-up against the number of threads.

OBSERVATIONS

Dynamic Scheduling. In the dynamic scheduling section, the opposite of what we had expected is happening. We expected the execution time to decrease as we provided the program with more and more threads. On the contrary, the execution time is increasing with increasing number of threads. The following explanation may provide an insight into this observation.

When we use dynamic scheduling, it maintains a central work queue of loop chunks. When a thread becomes idle, it must acquire a lock on this queue, take the next available chunk, and then release the lock.

[3]The speed-ups are calculated with respect to the sequential program. The speed-ups are also compared across averages of sequential and parallel runs.

This strategy can be very efficient where the loops are non-uniform, i.e., where some iterations take much longer than others. In other words, dynamic scheduling works best when a thread has a chance of executing faster than other threads.

In our case, the K-Means assignment loop is a highly uniform workload; every iteration takes a virtually identical amount of time, because each iteration essentially performs the same number of mathematical operations.

Static Scheduling. As can be seen in the graph, when we go from a sequential program to a parallel program with 4 threads, it takes just 30% of the time of the sequential program. This correlates to a speed-up of around 3.29.

However, as we increase the number of threads, the speed-up reduces. For example, we we go from 4 threads to 8, the speedup is just 1.51 (which is less than half of the speed-up we obtained when going from a sequential program to 4 threads).

This is essentially the law of diminishing returns, wherein a higher ”investment” need not guarantee the same ”return on investment”. In parallel programming, this is called as Amdahl's law. Amdahl's law can be stated as follows:

“The overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used.” [5]

Amdahl's law can be visualised using the following graph:

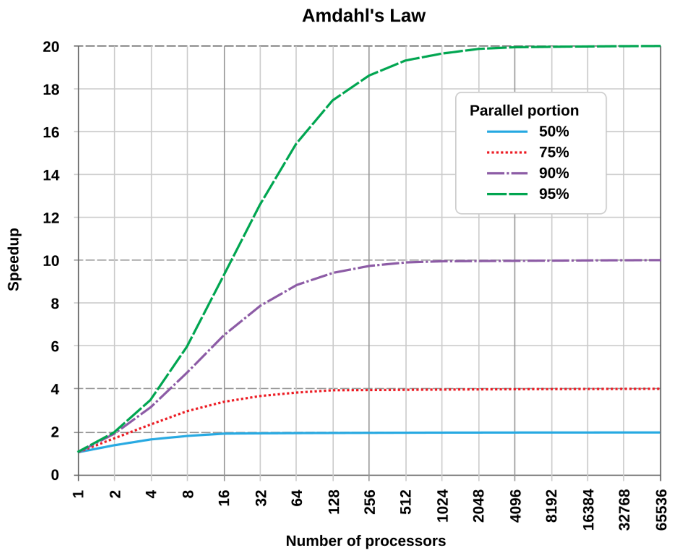

Figure 4. Visualisation of Amdahl's law. [6]

As can be seen in the graph, after a particular threshold, adding more processors (or threads) does not make any difference in the speed-up. What we have observed in our experiment corroborates Amdahl's law.

Dynamic vs. Static Scheduling. We can clearly see in the graph that the dynamic scheduling program required twice the execution of the static scheduling program, even for the program with a single thread. Ideally, both of these must have had the same execution time, due to them not being parallelised at all.

This is due to the unnecessary work of the scheduler's lock acquiring/releasing mechanism for every chunk, even with no other threads to compete with. In other words, when we attempt to dynamically schedule a program, there is a lot of overhead associated with it. In fact, the overhead can be so high that the execution time is doubled, as happened in this case.

Suhas Kamath

DS 221: INTRODUCTION TO SCALABLE SYSTEMS

November 14, 2025

Parallel Programming - MPI Assignment

Theory & Methodology

Matrix-vector multiplication is one of the most used methods with multiple applications, especially in the field of neural network training and image processing. [7]

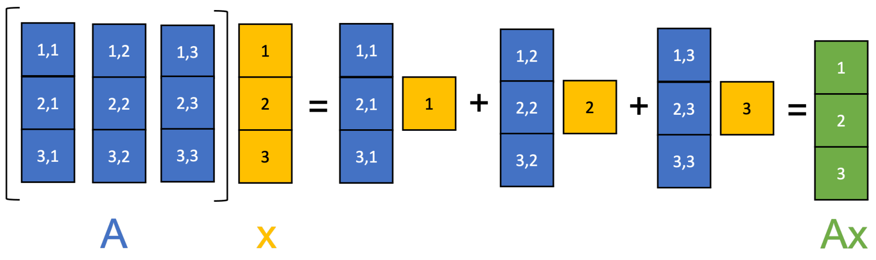

Figure 5. Pictorial representation of matrix-vector multiplication. [7]

Some of these times, we will have to deal with matrices with millions of rows and columns. However, in most of these applications, we have to deal with sparse matrices. Sparse matrices are matrices wherein most of the elements are zero. In such matrices, we can apply various compression techniques, such as the compressed-sparse-row (CSR) method, as is used in this experiment.

The CSR format stores a sparse matrix in row form using three one-dimensional arrays. The three arrays are as follows: [8] [4]

- `V[nnz]`: non-zero values in row-major order
- `C[nnz]`: column offset within the row group for a non-zero value
- `R[M + 1]`: offset to the start of nnz values for the i^{th} row in array `A`

When a matrix is stored in specialised CSR format, we can use specialised algorithms to perform matrix vector multiplications. In this case, what we do to efficiently parallelise the program. In such a case, the number of calculations is proportional to the number of non-zero elements. Hence, we divide the non-zero elements equally among the processes. This is a decent method of dividing the array into chunks for multiple processes.

Another aspect of this program is the IO. We read the matrix from a file in a shared folder on the cluster. One method to go about this is to get the master process (or rank 0 process) to read the entire matrix and then send chunks of it to all the sub-processes. However, this is very inefficient.

[4]Let NNZ denote the number of nonzero entries in M.

As we know, to read from a file, we only require a shared lock. As the name suggest, the shared lock can be shared by multiple processes. This implies that all processes can read the file at once. We should exploit this fact and get all the sub-processes to read from the same file in parallel. This ensures that a higher level of parallelism is reached, and that the processes are more independent. This helps to reduce the overall execution time, as the root node will not waste time in communicating the matrix data to all the other processes.

Each process also calculates its the row indexes for which it is responsible. This could have been done by the root process and then distributed among the other processes. However, by getting each process to calculate the indexes, we are saving on communication cost.

Experimental Setup

Similar to the OpenMP section of the assignment, both the sequential and parallel programs were executed on the teaching cluster. The teaching cluster has the following specifications:

- **CPU:** Intel Xeon Gold 5318Y 24-core (48-threads) 2.1GHz
- **RAM:** 128 GB RAM
- **Storage (HDF5):** 2×12 TB SATA HDD, 1.92TB PCIe NVME M.2 Enterprise SSD
- **Operating System:** Cent OS
- **Job scheduling software:** SLURM

To check the speedup provided by MPI, we used 8, 16, 32, 64, and 128 processes, alongside the sequential program. The computation time required by each process was output as a comma-separated values (CSV) file. The total execution time output directly to the terminal (and was recorded in the `output.log` file by `SLURM`). After the CSV timings data has been recorded, the data is fed into Python, and then analysed. The average time per number of processes, load factor, are output. The analysed data is plot using `matplotlib`. The results are shown in the section below.

Results

The table of results is as follows:

No. of Processes	**Avg. Computation Time (ms)**	**Avg. Execution Time (s)**	**Avg. Load Imbalance Factor**	**Speed-up w.r.t. 1-process**
1	378.16	1.871	-	-
8	80.70	1.100	0.0001	4.69
16	62.71	0.923	0.0002	6.03
32	59.06	1.559	0.0125	6.40
64	64.00	2.640	0.0006	5.91
128	76.82	5.814	2.4679	4.92

The graphs, as required by the assignment, are as follows:

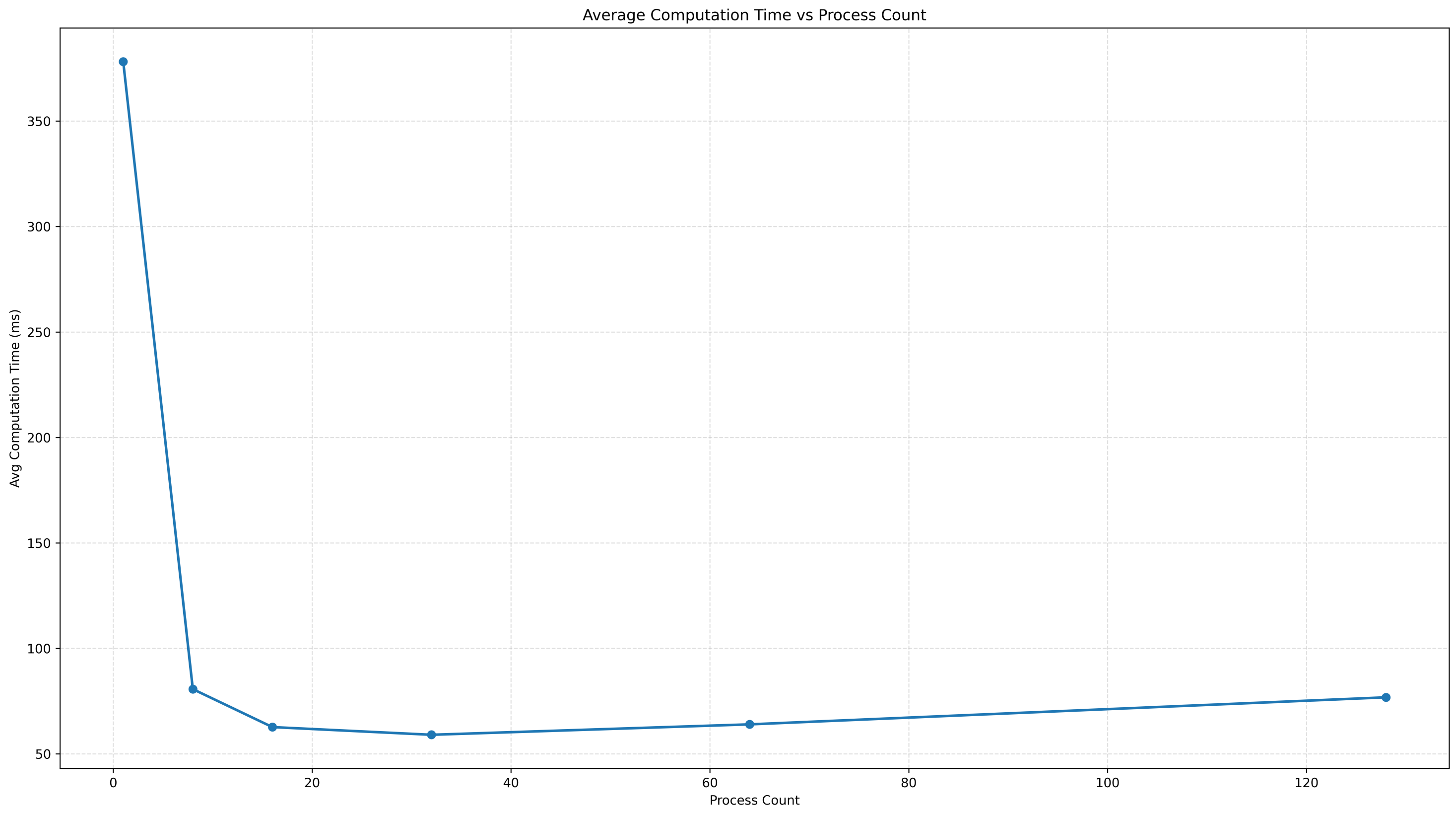

FIGURE 6. Plot of average computation time (across 5 runs) against number of processes.

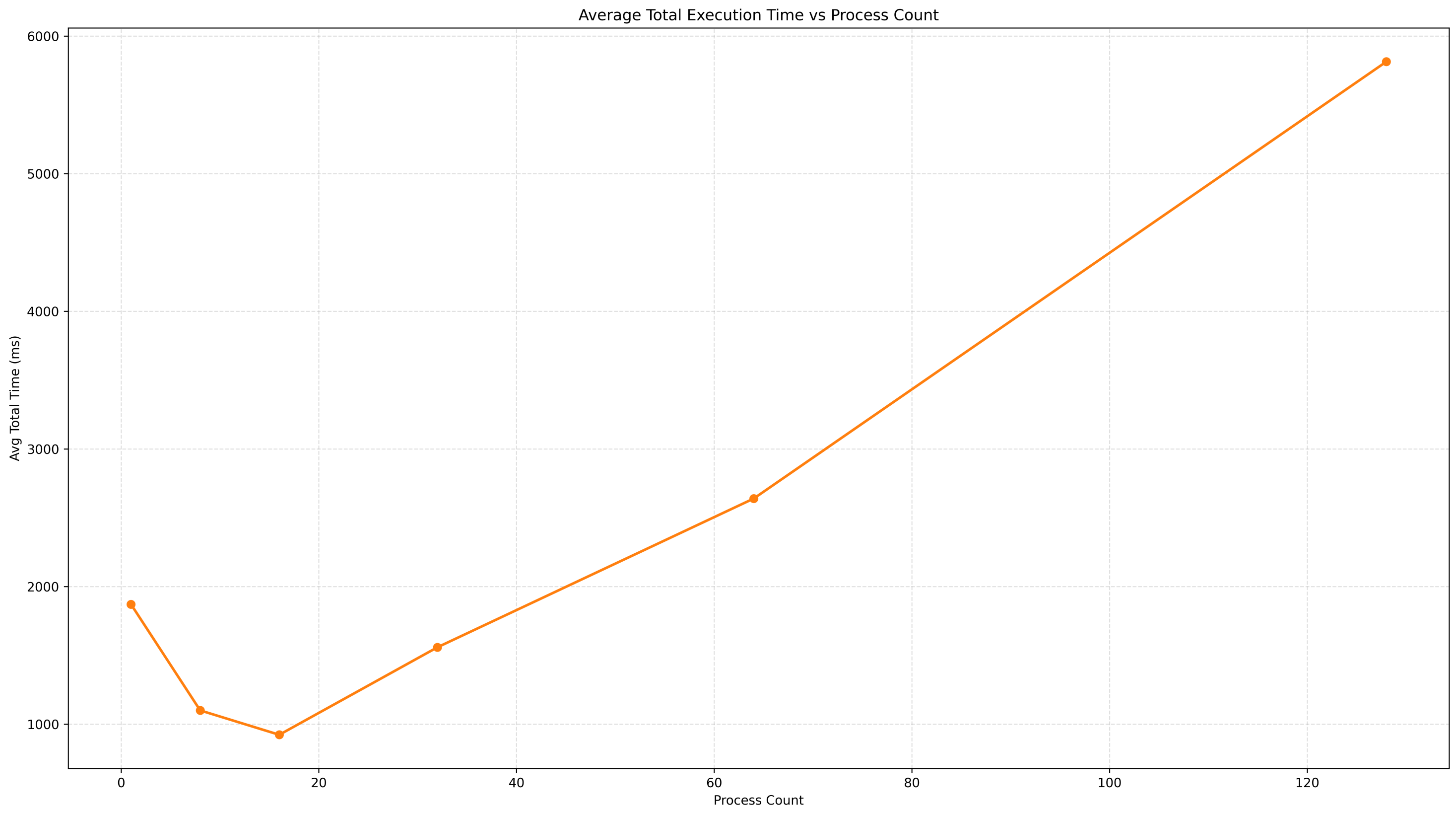

Figure 7. Plot of average execution (total) time (across 5 runs) against number of processes.

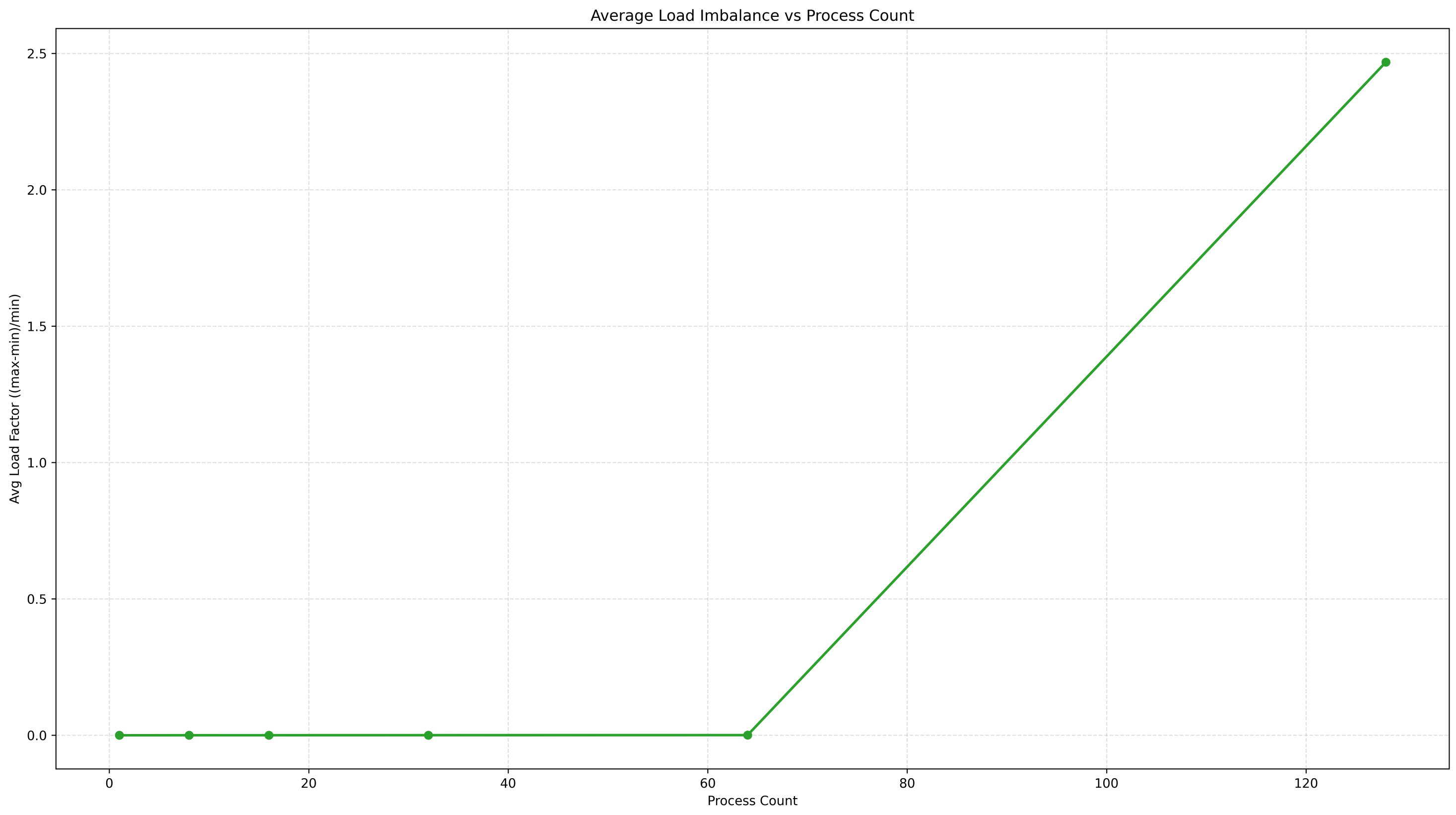

Figure 8. Plot of average load imbalance factor (across 5 runs) against number of processes.

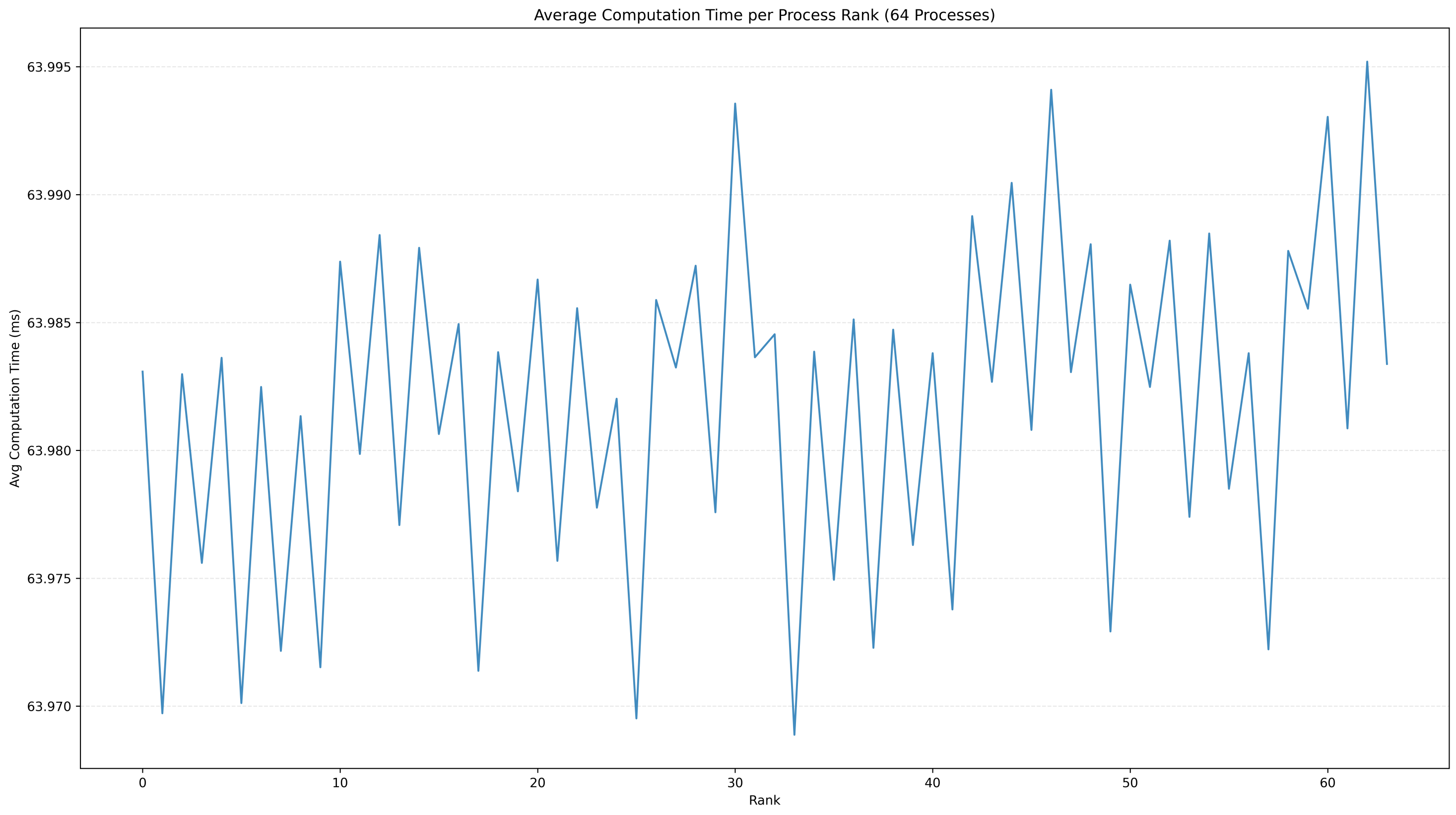

Figure 9. Time spent by each process against its rank, for 64 processes.

Observations

First, let us try to draw some conclusions from the table containing the results.

Firstly, let us see a trend in the We can see a huge reduction in computation time taken when we go from a sequential program to a 8-process parallel program. After this, the reduction in time is not as much. Again, as explained in the OpenMP section of the assignment, this is due to the law of diminishing returns, or as more popularly known in computer science, Amdahl's Law. A graph depicting this law is as follows:

As we go down the table, we also observe that the time taken by 64 or 128 processes is actually more than the time taken by 32 processes. Ideally, this should not happen. Even taking Amdahl's law into consideration, the time should not increase. It should either remain the same or decrease, as we increase the number of processes. The same can be seen in the above graph, where we can see that the graph eventually becomes an almost perfectly horizontal line, but does not increase.

The secret to this lies in the system specifications. As mentioned in the Experimental Setup section, the cluster on which this program is executed has only 48 threads. Creating more processes than cores essentially means that we will have to give a processor more than one more process to work with. This means that we shall have to employ CPU scheduling methods, which may result in context switches. A significant amount of

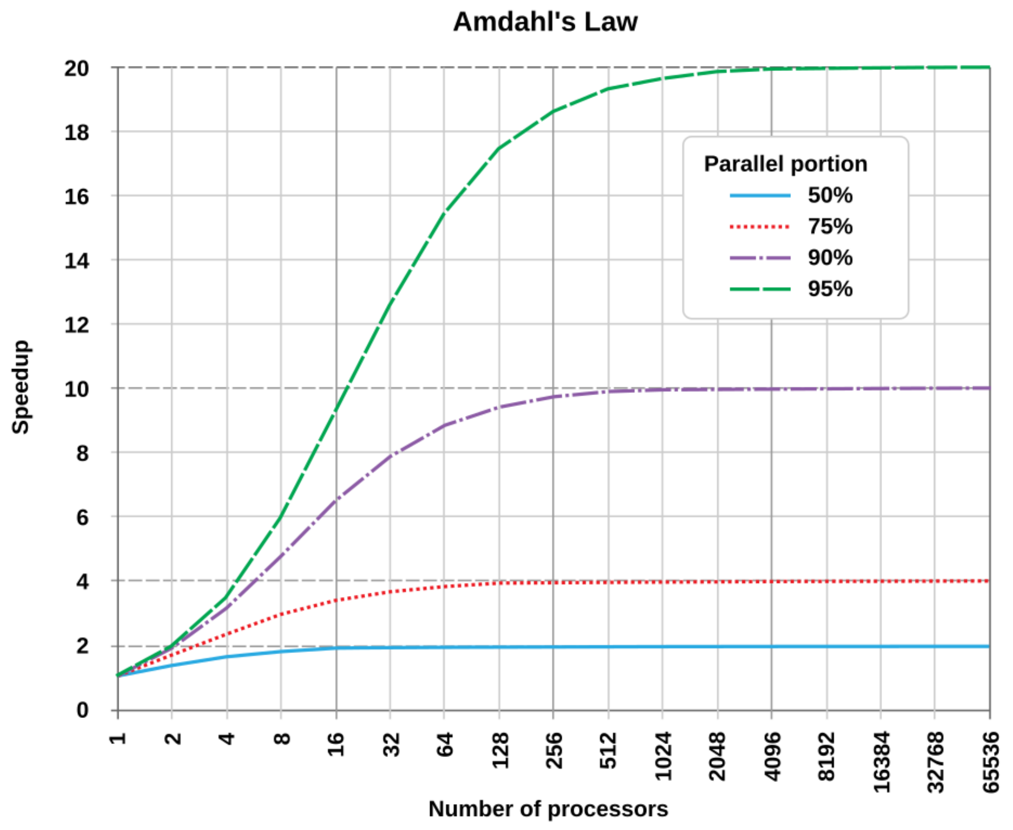

Figure 10. The graph depicting the average execution time against the number of threads. [6]

time is wasted in context switching, thereby increasing the total execution time. This is further exaggerated when we use 128 processes. Each core of the CPU will have to manage multiple processes.

This also mandates a higher load imbalance factor, which is very noticeable, especially when we are using 128 processes. As can be seen in the load imbalance graph, the graph is fairly horizontal until we get 128 processes, where it suddenly shoots up. Even when we are using 64 processes, the increase in load factor is not very noticeable in the graph. This change, however, can be noticed when the figures are quantified into numbers, as is done in the table.

Now, let us compare the average execution times. Contrary to the computation time, which quantifies only the time taken to perform the mathematical operations, the execution time measures the total time taken, including the time required to read the matrix and vector from disk, as well as the time taken for message passing.

We can see that the execution time decreases as we go from sequential to a 16-process execution. However, we can also notice that the execution time increases as we increase the number of processes above 16. This is due to disk contention. As mentioned earlier, each process reads the section of the matrix that it is responsible for parallelly with the other processes. Hence, all processes attempt to read from the same file at the same time. Furthermore, all of these processes want to read a mutually exclusive section of the file. In other words, every process want to read a unique set of values.

We know that a magnetic hard disk has a read-write head, and a set of disks that rotate. When we want to read some data from the disk, the cylinder rotates to the correct position. Then, the read-write heads finds the data in the section of the disk and returns it to the operating system. [9] Hence, if multiple processes want to read multiple sections of a file that may be span over multiple cylinders or tracks, the operating systems and HDD will not be able to service all these requests in a short amount of time. This is what is termed as disk contention.

References

[1] E. Kavlakoglu and V. Winland, "What is k-means clustering?,"

[2] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the Berkeley Symposium on Mathematics Statistics and Probability, 1967*, pp. 281–297, 1967.

[3] S. Bacallado and J. Taylor, "Stats202: K-means clustering," 2022.

[4] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '07)*, pp. 1027–1035, 2007.

[5] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[6] "Svg graph illustrating amdahl's law." Wikipedia, 2008.

[7] M. N. Bernstein, "Matrix-vector multiplication," 2020.

[8] S. Vadhiyar and C. Jain, "Ds221 (introduction to scalable systems): Algorithms and data structures," 2025.

[9] W. A. Goddard and J. J. Lynott, "Airect access magnetic disc storage device," 1954.