MPI k-means

Amirou Sanoussy

April 2024

1 Farthest First Conceptual and Demonstration Task

In implementing the calc_arg_max function, I aimed to parallelize the task of finding the index of the point in the dataset that maximizes the cost, which is the squared distance to the nearest center. To achieve this, I distributed the data among MPI processes, with each process independently handling a subset of the dataset. Within a loop iterating over the data points, I ensured that each process processed data starting from its rank with a stride equal to the total number of processes, ensuring an approximately equal workload distribution. After computing my local maximum cost and its corresponding index, I used an MPI_Allreduce operation with MPI_MAXLOC, facilitating synchronization and communication to obtain the global maximum cost and its index across all processes. By leveraging MPI collective communications and parallel computation, I aimed to achieve scalability and performance while accurately identifying the farthest point in the dataset.

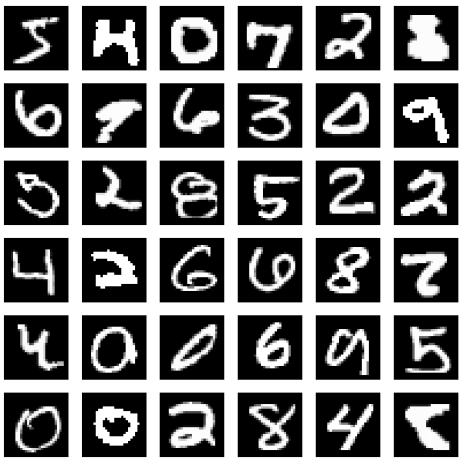

Figure 1: mnist_36_0.png

Below is the *kmeans* timing output:

```
[afroamir@tinkercliffs2 PSA04]$ sbatch mpi_kmeans_timing.sh 25 0
Submitted batch job 2367466
[afroamir@tinkercliffs2 PSA04]$ cat mpi_kmeans_timing.out
(1,52.1996),(2,26.1755),(4,13.0861),(8,6.5954),(16,3.3251),(32,1.7833),
```

Using the *mpi_kmeans_timing.out* values, I can then calculate the speedup from using 1 core, to 32 cores with the following formula: $S_P = \frac{T_S}{T_P}$ where

$$T_S = 52.1996 seconds$$
$$T_P = 1.7833 seconds$$
$$S_P = \frac{T_S}{T_P}$$
$$S_P = \frac{52.1996}{1.7833}$$
$$S_P = 29.1895095901$$

When comparing the performance of using 32 cores versus one core, the calculated speedup is approximately 29. This indicates that the parallelized implementation significantly improves performance compared to the serial execution. However, it's important to note that this speedup needs to catch up, which would be equal to the number of cores initialized, in this case, 32. The ideal speedup represents perfect linear scalability, where the execution time decreases linearly with the number of cores utilized. Though the code may need further optimizations and considerations for minimizing overheads, it could bridge the gap between the achieved speedup and the ideal speedup, leading to even more efficient parallel execution. The speedup discussion above can be seen visually

Below in a strong scaling study plot.

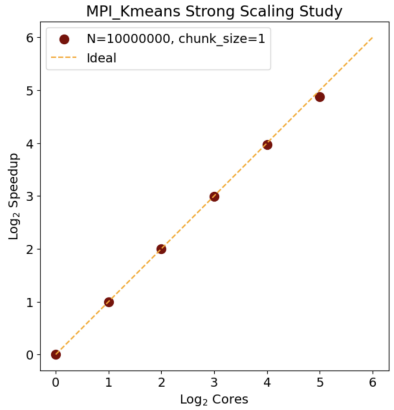

Figure 2: Farthest First Strong Scaling Study

2 k-means Conceptual and Demonstration Tasks

When implementing the calc_kmeans_next function, I aimed to efficiently update the centroid positions (kmeans_next) in a parallel environment. To achieve this, I distributed the dataset among MPI processes using a simple data parallelism approach, allowing each process to handle a subset of the data independently. After each process computed its local sums for kmeans_next and cluster_size, I utilized MPI collective communications, specifically MPI_Allreduce with MPI_SUM, to perform global summation across all processes. This ensured that all processes had access to the globally updated centroid positions and cluster sizes, facilitating accurate calculation of the new centroid positions. By leveraging MPI collective communications, I aimed to minimize communication overhead and ensure the correctness and scalability of the parallel implementation. Overall, by integrating MPI collective communications, the calc_kmeans_next function efficiently updated centroid positions in parallel, enabling improved performance and scalability in the parallel execution of the K-means algorithm.

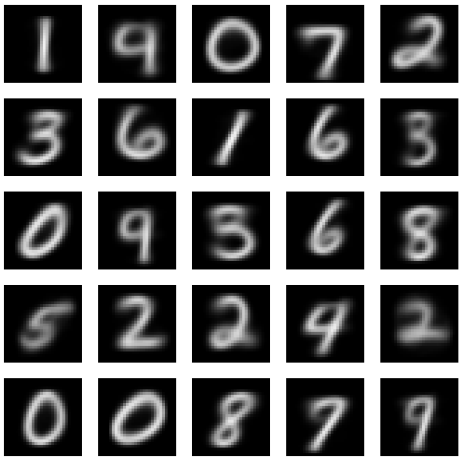

Figure 3: mnist_25_55.png

Below is the *kmeans* timing output:

```
[afroamir@tinkercliffs2 PSA04]$ sbatch mpi_kmeans_timing.sh 25 4
Submitted batch job 2368518
[afroamir@tinkercliffs2 PSA04]$ cat mpi_kmeans_timing.out
(1,70.2372),(2,35.3114),(4,17.7119),(8,8.8716),(16,4.6015),(32,2.2756),
```

Using the *mpi_kmeans_timing.out* values, I can then calculate the speedup from using 1 core, to 32 cores with the following formula: $S_P = \frac{T_S}{T_P}$ where

$$T_S = 70.2372 seconds$$

$$T_P = 2.2756 seconds$$

$$S_P = \frac{T_S}{T_P}$$

$$S_P = \frac{70.2372}{2.2756}$$

$$S_P = 30.8653541923$$

When comparing the performance of using 32 cores versus one core, the calculated speedup is approximately 31. This indicates that the parallelized implementation significantly improves performance compared to the serial execution. This is a lot better performance compared to the Farthest First performance. The speed up is almost equal to be proportional to the number of cores used in the parallelization of the code. However, the code can be tweaked to further optimize the codes performance so as to get as close to the ideal speedup. The

speedup discussion above can be seen visually Below in a strong scaling study plot.

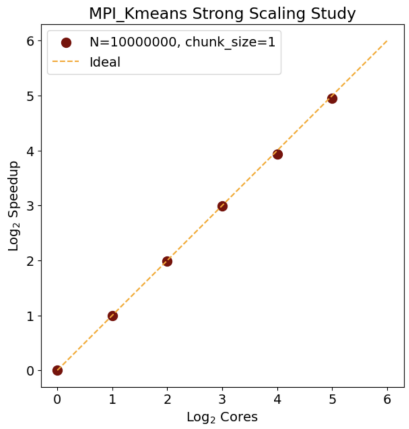

Figure 4: k-means scaling study plot