University of Houston

Parallelizing the k-means clustering algorithm using MPI

Homework 1

Course: Parallel Computations (COSC 6374)

Soumyottam[1] Chatterjee

Instructor: Dr. Edgar Gabriel

[1] schatterjee4@uh.edu, 313-413-7466

1 Problem Description

This project implements the parallelization of the *k-means clustering algorithm* using Message Passing Interface or MPI.

1.1 k-means clustering

Wikipedia defines k-means clustering as:

> k-means clustering is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining. k-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells.

1.1.1 Description

We quote from *Wikipedia*:

> Given a set of observations $(x_1, x_2, \ldots, x_n)$ where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into $k(\leq n)$ sets $S = \{S_1, S_2, \ldots, S_k\}$ so as to minimize the within-cluster sum of squares (WCSS).

1.1.2 Standard Sequential Algorithm or StdSeqAlg

We refer to *Wikipedia*:

> The most common algorithm uses an iterative refinement technique.
>
> Given an initial set of k means $m_1^{(1)}, m_2^{(1)}, \ldots, m_k^{(1)}$, the algorithm proceeds by alternating between two steps:
>
> 1. **Assignment step:** Assign each observation to the cluster whose mean yields the least within-cluster sum of squares (WCSS). Since the sum of squares is the squared Euclidean distance, this is intuitively the "nearest" mean.
> 2. **Update step:** Calculate the new means to be the centroids of the observations in the new clusters. Since the arithmetic mean is a least-squares estimator, this also minimizes the within-cluster sum of squares (WCSS) objective.
>
> The algorithm has converged when the assignments no longer change. Since both steps optimize the WCSS objective, and there only exists a finite number of such partitionings, the algorithm must converge to a (local) optimum. There is no guarantee that the global optimum is found using this algorithm.

1.1.3 Implementation of the StdSeqAlg

An Implementation of the StdSeqAlg using C can be found at A C-language Implementation of the k-means clustering algorithm by Dr. Edgar Gabriel.

1.2 Parallelization of the k-means clustering algorithm

The project is as follows (quoted from the following link).
Problem Description:

> Given the sequential code for an image processing algorithm, which performs a k-means clustering operation on pixels of an image,

1. Parallelize the algorithm using MPI using a 1-D block-row wise data distribution.
   (a) Develop the necessary code assuming each process holds the same number of rows of the image.
   (b) Measure the execution time for the k-means clustering for 1, 2, 4 and 8 processors for the 1024×1024 and 2048×2048 pixel images provided.
   (c) Determine the parallel speedup and the parallel efficiency of the code sequence on the whale cluster (Department of Computer Science, University of Houston) for the 2, 4, and 8 processor cases.
2. Consider the case of 1-D block column wise data distribution.
   (a) Briefly explain how the code would have to look for this data distribution.
   (b) Compare the communication requirements (number of messages, data transferred, number of collective operations) between the 1-D block column and block row wise data decomposition.

2 Solution Strategy

Suppose there are n processes that communnicate and coordinate among themselves to solve the k-means clustering problem on an $m \times m$ pixel image. We assume that m is divisible by n so that each process holds exactly the same amount of data.

1. Each of the n-processes reads their own portion of the input image file in parallel. In a block-row wise data distribution, each process reads $\frac{m}{n}$ rows. We implement this by requiring that the process with rank i reads from Row $\frac{i \cdot m}{n}$ to Row $\frac{(i+1) \cdot m}{n} - 1$.
2. Process 0 chooses k data points from its part of the image to serve as the initial values for the k centroids. Process 0 then "broadcasts" the centroid-values to all the other processes.
3. Each process has now their own data points (pixels) and the centroid values (which are the same across all processes throughout the course of the algorithm). So each process easily assigns each data point to their nearest clusters (centroids).
4. Now we need to compute the new centroid-values, which are the arithmetic means of their respective cluster-member-data-points. Each process computes the sum of the cluster-member-data-points for each cluster, and then those "local" sums are combined (with the help of the MPI_Allreduce operation) to calculate the global sum, from which we can easily determine the arithmetic mean of *all* the member-data-points for each cluster, and thus get the new centroid values for each cluster.
5. We go back to Step 3 and repeat until the sums of squares of distances of the member data points from their respective centroids no longer significantly change. At that point, the algorithm decides that it has found the local optimum.

3 Results

3.1 Resources Used

We run our code on the Whale Cluster at the Department of Computer Science at University of Houston. For an n-process job, the cluster allocates n-nodes, each of which has the following hardware configurations:

- two 2.2 GHz quad-core AMD Opteron processor (8 cores total)
- 16 GB main memory
- 4×DDR InfiniBand HCAs.

3.2 Measurements

Let $N = \{1, 2, 4, 8\}$ and $M = \{512, 1024, 2048\}$. $n \in N$ denotes the number of processes and $m \in M$ denotes the dimension of the input image fine. For each $(n, m) \in N \times M$, we find out the average execution time of our algorithm. We do this through the following methodology.

For a fixed (n, m), we iterate our algorithm for $NUM\_REPETITIONS$ times and then calculate the average execution time. We further repeat this measurement 5 times and calculate the average execution time.

Remark. So, in total, each average is found out from $5 \cdot NUM\_REPETITIONS$ actual runs of the algorithm.

The value of $NUM\_REPETITIONS$ is set as follows:

- $NUM\_REPETITIONS = 100$ when $m = 512$.
- $NUM\_REPETITIONS = 50$ when $m = 1024$.
- $NUM\_REPETITIONS = 20$ when $m = 2048$.

We measure the execution times with the help of the MPI function MPI_Wtime. We measure the execution times in microseconds.

Once we have measured all the execution times, we calculate the parallel speedups and the parallel efficiencies with the following formulas.

$$S(p) = \frac{T_s}{T_p(p)} \tag{1}$$

and

$$E(p) = \frac{S(p)}{p} \tag{2}$$

where, p is the number of processes, T_s is the execution time for the sequential algorithm and $T_p(p)$ is the execution time of the parallel algorithm using p processes. $S(p)$ is the parallel speedup and $E(p)$ is the parallel efficiency.

3.2.1 Tables and Graphs

Number Of Processes	Average Time (in Microseconds)		
	Image Dimension = 512 x 512	Image Dimension = 1024 x 1024	Image Dimension = 2048 * 2048
1	5277909.318	33928208.176	82479602.636
2	2948524.166	18038189.784	44879032.958
4	1843256.674	10123216.856	26339424.06
8	1435994.526	6329566.244	17068505.948

Figure 1: Time measurements - table

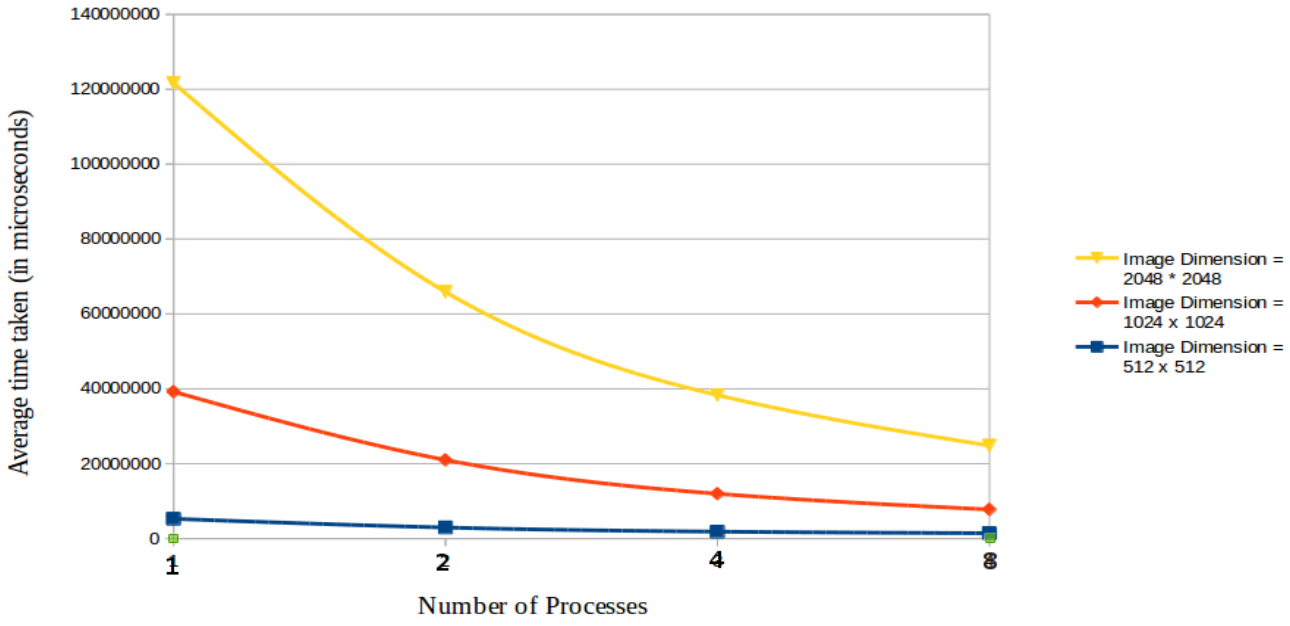

Figure 2: Time measurements - graph

Number Of Processes	Speedup (multiplied by 1000)		
	Image Dimension = 512 x 512	Image Dimension = 1024 x 1024	Image Dimension = 2048 * 2048
1	1000	1000	1000
2	1790	1881	1838
4	2863	3351	3131
8	3675	5360	4832

Figure 3: Speedup comparisons - table

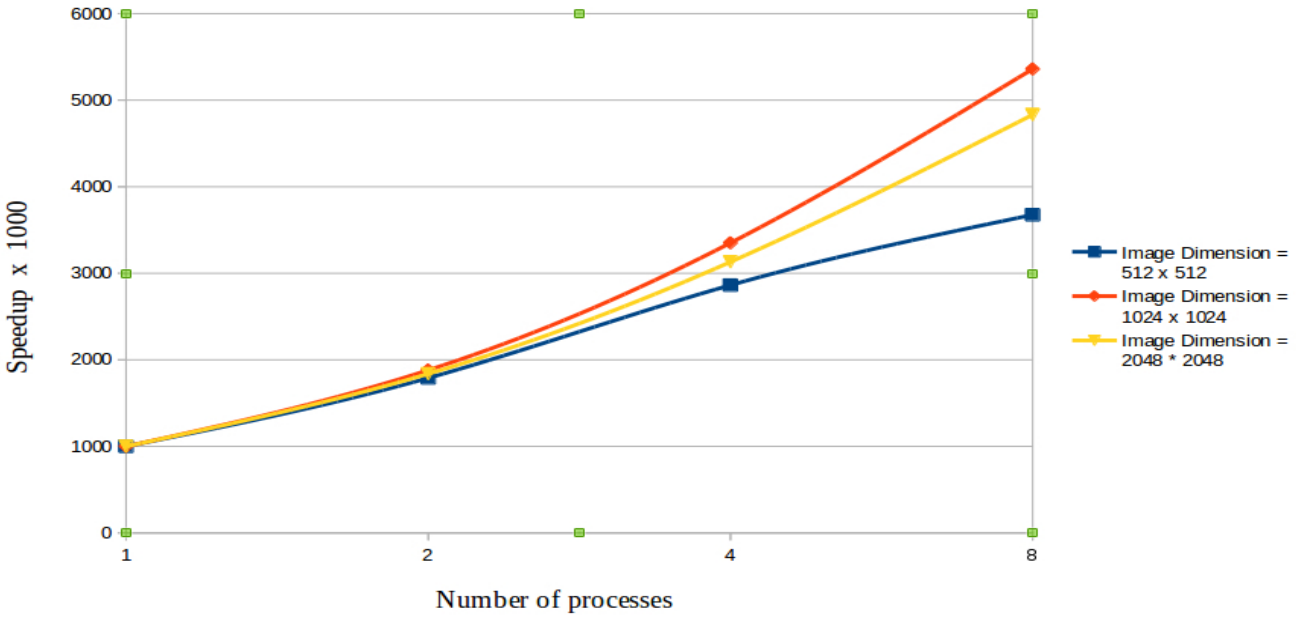

Figure 4: Speedup comparisons - graph

Number Of Processes	Efficiency (in percentage)		
	Image Dimension = 512 x 512	Image Dimension = 1024 x 1024	Image Dimension = 2048 * 2048
1	100	100	100
2	89.5	94.1	91.9
4	71.6	83.8	78.3
8	45.9	67	60.4

Figure 5: Efficiency comparisons - table

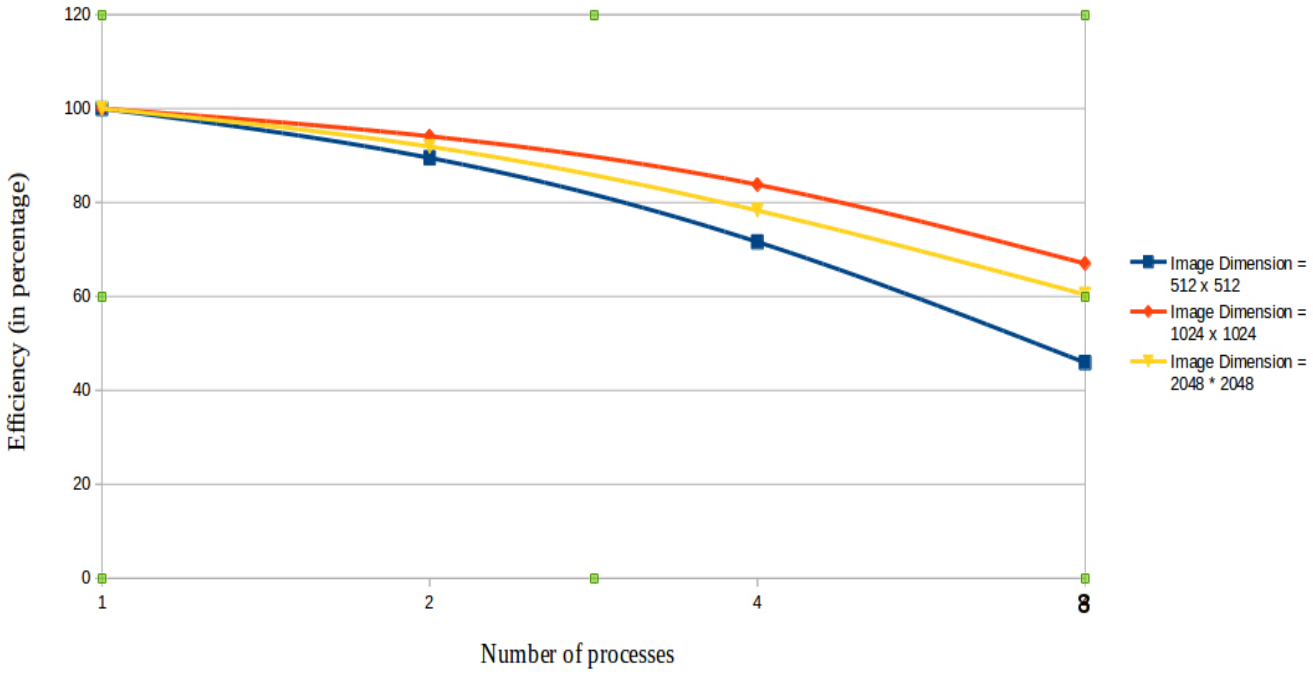

Figure 6: Efficiency comparisons - graph

4 Discussion of block-column wise data distribution

4.1 Modifications in the code

In this code, each process would hold entire columns but only portions of rows. In other words, each process would hold pixels from *non-contiguous* locations from the input image file. Thus the reading of the input and the writing of the output would have to be performed differently.

In the block-row wise data distribution, a single "read" or "write" operation suffices, but in this case, we would need to run a loop over all the rows to perform the "read" and the "write" operations correctly.

But once the reading of the input has been done, the algorithm proceeds exactly as before, where each process computes the member data points for each clusters, and then combines their calculations, and so on.

4.2 Communication Requirements

Since the algorithm works in the exact same way, the communication requirements would be the same too. For example, in our implementation, there would be one MPI_Bcast operation, and three MPI_Allreduce operations. No other communication is involved.