CMPUT 681: PARALLEL AND DISTRIBUTED SYSTEMS

Final Project

Data Clustering on Distributed Memory Multiprocessors (k-means)

Name: Farrukh Ahmed
Student ID: 1438704
CCID: farrukh1

Name: Diego Serrano
Student ID: 1202391
CCID: serranos

Name: Varun Sapra
Student ID: 1439266
CCID: vsapra

Dec/12/2014

1. Introduction

Data Clustering is a computationally expensive task and one of the most popular clustering algorithms is k-means algorithm. K-means partitions *n* data points into *k* clusters where each data point is placed in a cluster with nearest mean. The time complexity of sequential k-means algorithm to cluster *d* dimensional vectors taking *i* iterations is *O(nkdi)*.

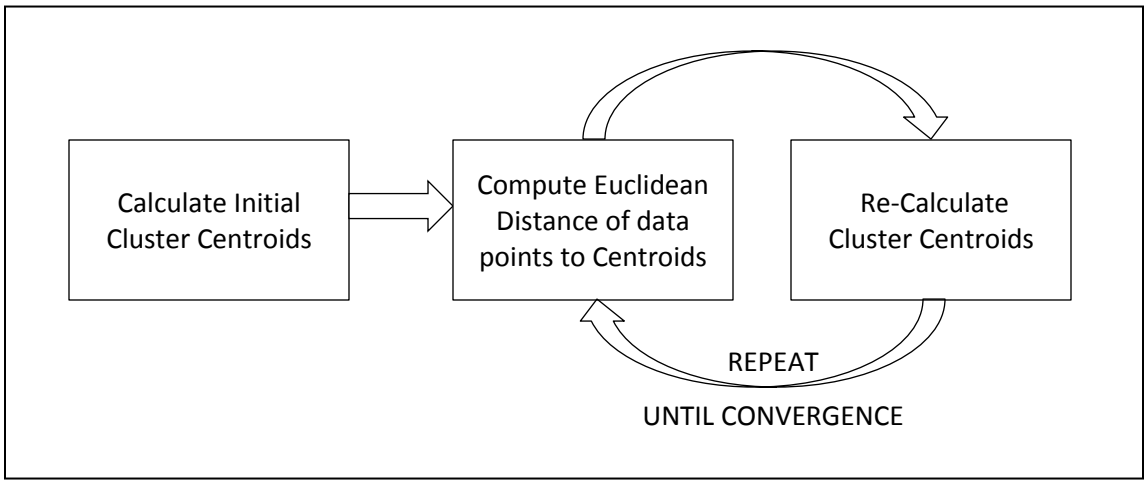


Figure 1: Sequential k-means Algorithm

Parallel algorithms can achieve better performance by dividing the problem into sub-tasks and spreading them over multiple processors (*p*). Along with the consideration of performance, paper mentions the need of parallelism over distributed memory processors due to huge datasets that cannot be processed in shared memory. The algorithm presented in the paper divides the task in a way that each processor is responsible for processing a contiguous block of *n/p* vectors. As all the processors (*p*) are similar, load is divided equally and each processor works on same number of vectors.

2. Empirical Analysis

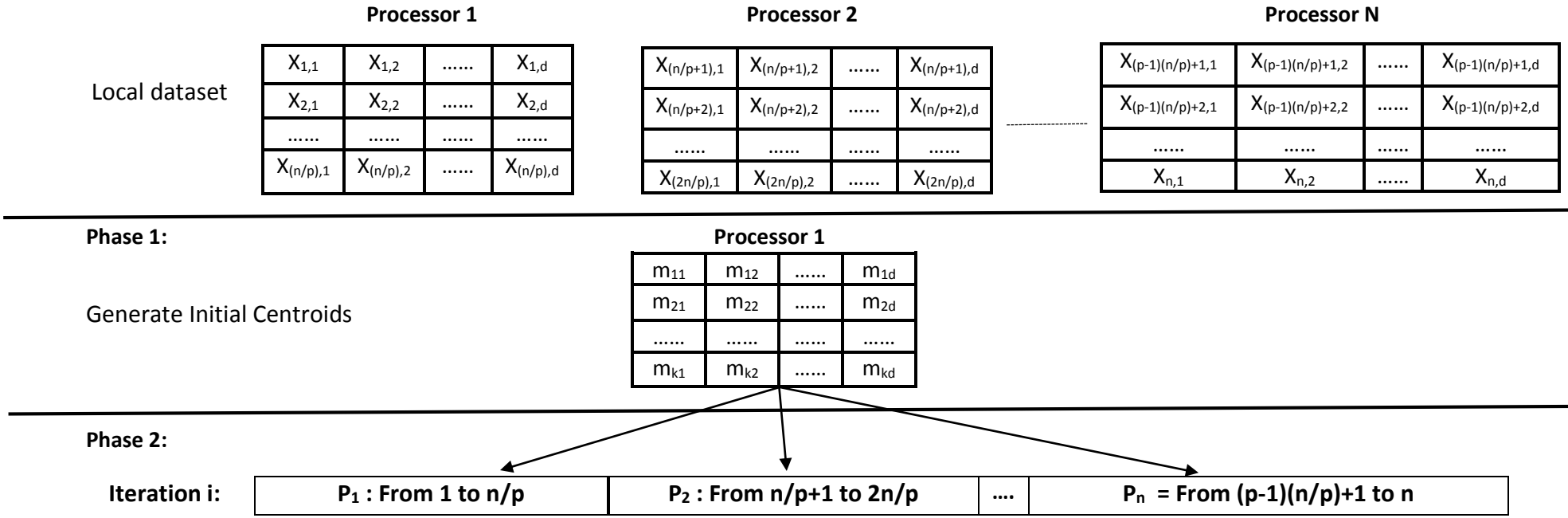
2.1 Implementation

The parallel algorithm in the paper generates initial cluster centroids and then divides work so that each processor has to find closest centroids for disjoint set of data points. After assigning local data points to clusters, processors communicate this information to recalculate centroids. This recalculation of centroids continues until convergence.

We implemented the algorithm ourselves as the implementation from the authors was not available. According to the description in paper, we have implemented the algorithm in three phases:

1. **Phase One: Initialization.** One processor generates initial cluster centroids randomly and broadcasts them to all the other processors.
2. **Phase Two: Distance Calculation.** Each processor calculates the Squared Euclidean distance for each of the data points $X_i, \mu * \left(\frac{n}{p}\right) + 1 \leq i \leq (\mu + 1) * \left(\frac{n}{p}\right), where\ \mu = 0\ to\ p - 1$ to the cluster centroids. These data points are added to a cluster with minimum distance. Additionally, local mean standard error *mse* is also updated.
3. **Phase Three: Centroid Recalculation.** Each processor communicates its local sum of data points and total number of data points in each cluster. The global sum and count is calculated at each processor and averaged to get new cluster centroids. Local *mse* is also aggregated in global *mse* to check convergence.

Our implementation is in C and MPI, following the pseudo-code shown in (1). The timing measurements were done using `MPI_Wtime()`. Following the convention of the original paper, timing measurements ignore the time required to read the dataset from disk.


Processor 1
Processor 2
Processor N
Local dataset
X1,1 X1,2 ..... X1,d
X2,1 X2,2 ..... X2,d
....
X(n/p),1 X(n/p),2 ..... X(n/p),d
X(n/p+1),1 X(n/p+1),2 ..... X(n/p+1),d
X(n/p+2),1 X(n/p+2),2 ..... X(n/p+2),d
....
X(2n/p),1 X(2n/p),2 ..... X(2n/p),d
X(p-1)(n/p)+1,1 X(p-1)(n/p)+1,2 ..... X(p-1)(n/p)+1,d
X(p-1)(n/p)+2,1 X(p-1)(n/p)+2,2 ..... X(p-1)(n/p)+2,d
....
Xn,1 Xn,2 ..... Xn,d
Phase 1:
Processor 1
Generate Initial Centroids
m11 m12 ..... m1d
m21 m22 ..... m2d
....
mk1 mk2 ..... mkd
Phase 2:
Iteration i:
P1 : From 1 to n/p
P2 : From n/p+1 to 2n/p
....
Pn = From (p-1)(n/p)+1 to n
For every processor
Compute Euclidean Distance
$D_1 = \sum_{t=1}^{d}(m_{1t} - X_{it})^2$
$D_2 = \sum_{t=1}^{d}(m_{2t} - X_{it})^2$
.....
$D_k = \sum_{t=1}^{d}(m_{kt} - X_{it})^2$
Find closest centroid distance
min_distancei = min (D1, D2,....., DK)
Find closest centroid
min_centeri = IndexOf(min_distancei)
Compute local mean squared error
local_mse += min_distancei

Add X_i to cluster with closest local centroid — index: min_center_i

……	……	……	……
……	……	……	……
X_{i1}	X_{i2}	……	X_{id}
……	……	……	……

Computed sum and count of data points assigned to local centroids

m'_{11}	m'_{12}	……	m'_{1d}
m'_{21}	m'_{22}	……	m'_{2d}
……	……	……	……
m'_{k1}	m'_{k2}	……	m'_{kd}

n'_1
n'_2
…..
n'_k

Phase 3:

Perform Allreduce to recalculate global centroids

$\sum_{\mu=1}^{p} m'_{11.\mu} / \sum_{\mu=1}^{p} n'_{1.\mu}$	……..	……..	$\sum_{\mu=1}^{p} m'_{1d.\mu} / \sum_{\mu=1}^{p} n'_{1.\mu}$
……	……..	……..	……
$\sum_{\mu=1}^{p} m'_{k1.\mu} / \sum_{\mu=1}^{p} n'_{k.\mu}$	……..	……..	$\sum_{\mu=1}^{p} m'_{kd.\mu} / \sum_{\mu=1}^{p} n'_{k.\mu}$

Aggregate MSE — MSE = $\sum_{\mu=1}^{P} local\_mse$

Repeat phase 2 and phase 3 until convergence: — while (MSE < OldMSE)

Figure 2: parallel k-means implementation

In order to test the correctness of the algorithm, our implementation was run on special error-free (or noise-free) datasets generated using the algorithm in (2). For these testing datasets, we pick the initial centroids in a way that the local minimum found by the algorithm is also the global minimum. From the generated datasets, we know the centroids that generated all the points, and we can compare that our solution is close to the generating centroids.

When testing the algorithm for a given dataset with a different number of processors, we noted the results were slightly different. After investigating the possible causes, we found in (3) that, floating-point arithmetic suffers from rounding errors and other inadequacies in bit representation with a limited number of bits. Consequently, floating-point operations (such as SUM or PRODUCT) are commutative but not associative. Thus, the order in which they are applied in collective operations makes a difference to the final outcome. When we tested the algorithm again on different number of processors, we found that the result is the same on different number of processors within a small margin of error.

Additionally, the algorithm's convergence condition is very sensitive to those small variations caused by floating-point operations, resulting in different number of iterations when the number of processors is changed. In order to have a consistent comparison, we fixed the number of iterations in the experiments for same data size and different number of processors, as inferred by the experiments presented in the original paper.

2.2 Experimentation

We ran all our experiments on the *coldlake* cluster, using nodes *cold04* to *cold08*. The nodes used have quad-core AMD A8-5545M APU, 8GB of memory, and are connected through a Gigabit Ethernet switch.

We conducted our experiments up to 8 processors. In the original paper (1), they also perform experiments with 16 processors; however, at the time of conducting our experiments, 16 processors were not available.

The datasets[1] were generated using the algorithm in (2) with the same parameters. Following the convention in (1), we randomly select the initial centroids and use the same centroids regardless of the number of processors. Other variations of k-means employ better strategies to pick initial centroids, such as k-means++ (4) and its parallel implementation (5), however, in our experiments we are concerned about performance and not the global minimum.

In order to smooth fluctuations, each data point reported is the result of the average of the last five runs, from a total of seven runs. During the executions, we tried to minimize the impact of other processes, users or traffic contending for the processor or network bandwidth, by examining the cluster monitoring tool before sending the tasks for execution.

In order to assess our implementation, we employ several performance metrics. **Speedup** measures the performance of parallel execution compared to the sequential execution. As in the original paper, we report 3 sets of speedup experiments, where we vary *n*, *d* and *k* respectively.

Data Size n	**Iterations**	**Execution Times (in seconds)**			
		1 PE	**2 PEs**	**4 PEs**	**8 PEs**
$\mathbf{2^{13}}$	47	0.393	0.274	0.264	0.231
$\mathbf{2^{15}}$	43	1.435	0.777	0.504	0.347
$\mathbf{2^{17}}$	33	4.362	2.220	1.198	0.667
$\mathbf{2^{19}}$	25	12.794	6.398	3.271	1.719
$\mathbf{2^{21}}$	12	24.522	12.265	6.170	3.160

Table 1: Execution Times in seconds for varying *n*, with fixed *d=8* and *k=8*.

[1] Datasets are available at http://we.tl/bNqjvkWtPP

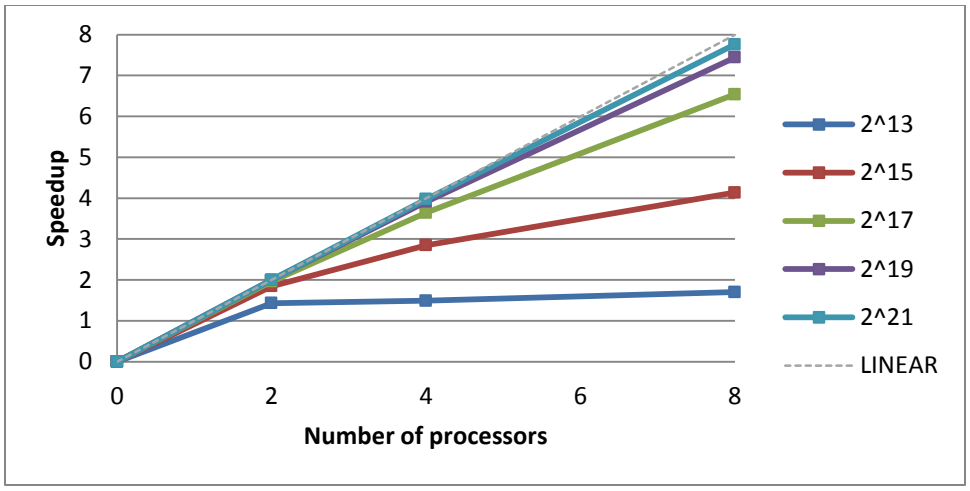


Figure 3: Speedup comparison for varying *n*, with fixed *d=8* and *k=8*.

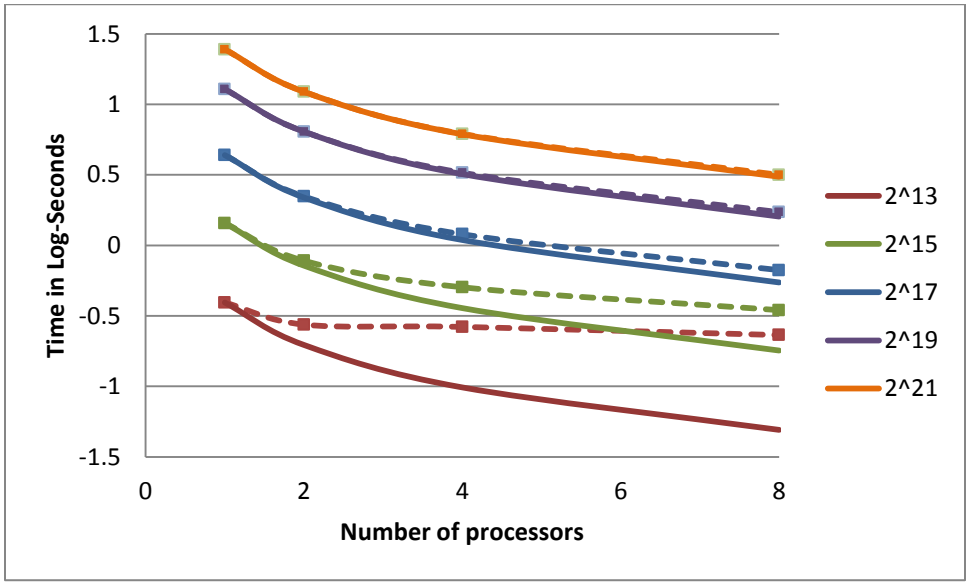


Figure 4**:** Log_{10}-seconds versus the number of processors for varying *n*, with fixed *d=8* and *k=8*. A dotted line connects the observed execution times, while a solid line represents ideal execution times obtained by dividing the observed execution time for *1* processor by the number of processors.

Varying *n*: We study how speedup varies when $n=2^{13}$, 2^{15}, 2^{17}, 2^{19} and 2^{21}, with fixed number of dimensions *d=8* and clusters *k=8*. The experiments show that increasing the number of processors for a specific problem size increases speedup. For example, the speedup for $n=2^{21}$

with *4* processors is *3.97* and speedup with *8* processors is *7.76*. For this largest dataset, we observe a relative speedup of 7.76 on 8 processors, for a speedup efficiency of 97%. However, for all sizes of *n*, diminishing returns for speedup can be seen when number of processors is increased. As the number of processors is increased, overhead of communicating local sum and count of data points is increased.

On the other side, increasing the data size for a fixed number of processors improves the speedup. In case of *8* processors speedups for $n=2^{13}$, 2^{15}, 2^{17}, 2^{19} and 2^{21} are *1.70, 4.13, 6.53, 7.44 and 7.76* respectively, which confirms the excellent size-up behavior. This effect is less obvious for 2 processors because the cost of communication is comparatively low. The effects of granularity are obvious from the results witnessed during experimentation.

Dimensions d	**Iterations**	**Execution Times (in seconds)**			
		1 PE	**2 PEs**	**4 PEs**	**8 PEs**
2	31	22.492	11.306	5.732	3.011
4	33	37.256	18.875	9.618	5.015
8	12	24.523	12.265	6.170	3.160

Table 2: Execution Times in seconds for varying *d*, with fixed $n=2^{21}$ and *k=8*.

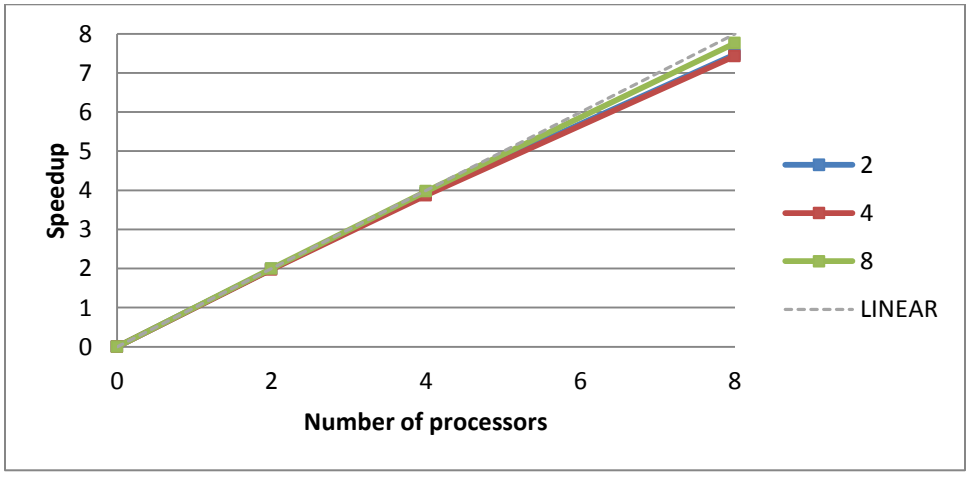


Figure 5: Speedup comparison for varying *d*, with fixed $n=2^{21}$ and *k=8*.

Varying *d*: For the dimensions, we study the speedup behavior when the dimensions vary as *d=2*, *4* and *8*, with fixed number of data points $n=2^{21}$ and clusters *k=8*. In Figure 5, we observe near linear speedup between *7.43* and *7.76* on *8* processors, because of high granularity due to large value of $n=2^{21}$. Similar to the original paper, it can be seen that varying *d* has a very little impact on speedup.

Clusters K	Iterations	Execution Times (in seconds)			
		1 PE	2 PEs	4 PEs	8 PEs
2	14	7.431	3.617	1.823	0.948
4	16	16.591	8.087	4.081	2.093
8	12	24.523	12.265	6.170	3.160
16	18	67.034	33.587	16.963	8.720

Table 3: Execution Times in seconds for varying *k*, with fixed $n=2^{21}$ and *d=8*.

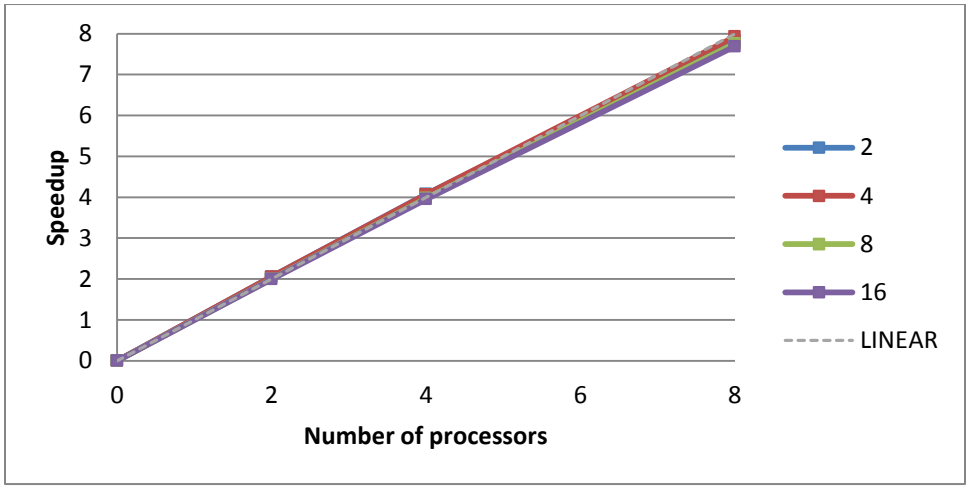


Figure 6: Speedup comparison for varying *k*, with fixed $n=2^{21}$ and *d=8*.

Varying *k*: Finally, we study how speedup varies when *k=2*, *4*, *8* and *16*, with fixed number of data points $n=2^{21}$ and dimensions *d=8*. In Figure 6, we also observe near linear speedups between 7.69 and 7.93 on 8 processors, because of high granularity due large value of $n=2^{21}$. Similarly as in the case of varying d, varying k has negligible impact on speedup like the results discussed in the original paper.

Although speedups in our experiments are almost linear, as in the original paper (1), the authors reported a speedup efficiency of 97.6% when $n=2^{21}$, $d=8$, $k=8$ and $p=16$, which is about the same efficiency, 97%, we achieved with the same parameters and $p=8$. If we follow the trend using a linear interpolation of the speedup efficiency, with diminishing returns we would achieve 94.6% efficiency. This decrease in speedup may be explained by the cluster architecture, which in the case of the original paper is a specialized system, the IBM POWERparallel SP2[2], with optimizations regarding compacting data structures to decrease communication overhead.

We also measure the **scale-up**, which is the ability to keep the performance levels when workload and resources increase proportionally. We report 3 sets of scale-up experiments, where we scale *n*, *d* and *k* respectively. Note that we use the time per iteration, because different datasets may require significantly different number of iterations that will affect raw time and scale-up analysis.

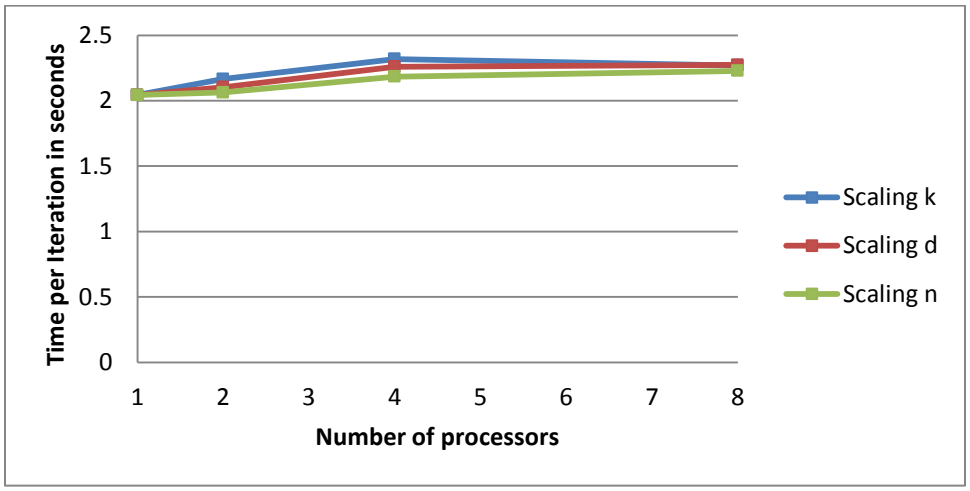


Figure 7: Scale-up curves. Time per iteration in seconds versus the number of processors.

[2] More information about IBM SP2 can be found at http://www.cac.cornell.edu/~slantz/what_is_sp2.html

Scaling *n*: To perform the scale-up with respect to *n*, we used datasets with $n=2^{21}*p$ on *p=1*, *2*, *4* and *8* processors respectively, with fixed number of dimensions *d=8* and desired clusters *k=8*. The execution times are reported in Figure 7, from where it can be seen that the parallel implementation delivers almost constant time, only perturbed by the time it takes to communicate and synchronize.

Scaling *d*: The scale-up on *d*, we used datasets with *d=8*p* on *p=1*, *2*, *4* and *8* processors respectively, with fixed number of data points $n=2^{21}$ and desired clusters *k=8*. The execution times are reported in Figure 7, with similar results as the scale-up on *n*.

Scaling *k*: The scale-up on *k*, we used datasets with *k=8*p* on *p=1*, *2*, *4* and *8* processors respectively, with fixed number of data points $n=2^{21}$ and dimensions *d=8*. The execution times are reported in Figure 7, with similar results as the scale-up on *n*.

In the scale-up experiments we could notice that when different machines have to communicate and synchronize, the time spent per iteration increases moderately. In our experiments, the increase of time per iteration from a single processor to 8 processors is approximately 10% of the time (0.230 seconds). Additionally, it can be seen that from 4 to 8 processors, it remains almost constant, since we use the same number of machines for the experiments with p=4 and p=8. In conclusion, the scale-up experiment shows that the algorithm achieves a good scale-up behavior, only adding minimum overhead for communication and synchronization, depending on the number of machines, since most message-passing operations are collective operations.

2.3 Phase-by-Phase Analysis

The Phase-by-Phase analysis presents more details about factors affecting the speedup. We have analysis with *2*, *4* and *8* processors as for single processor we use sequential algorithm.

Phases	Execution Times (in seconds)		
	2PEs	4PEs	8PEs
Phase 1	0.00003	0.00003	0.00002
Phase 2	0.197	0.103	0.059
Phase 3	0.076	0.160	0.171

(a) Data Size of $n=2^{13}$, No. of Iterations: 47

Phases	Execution Times (in seconds)		
	2Pes	4PEs	8PEs
Phase 1	0.00002	0.00003	0.00002
Phase 2	0.714	0.361	0.185
Phase 3	0.063	0.142	0.162

(b) Data Size of $n=2^{15}$, No. of Iterations: 43

Phases	Execution Times (in seconds)		
	2Pes	4PEs	8PEs
Phase 1	0.00002	0.00003	0.00002
Phase 2	2.161	1.083	0.549
Phase 3	0.058	0.113	0.117

(c) Data Size of $n=2^{17}$, No. of Iterations: 33

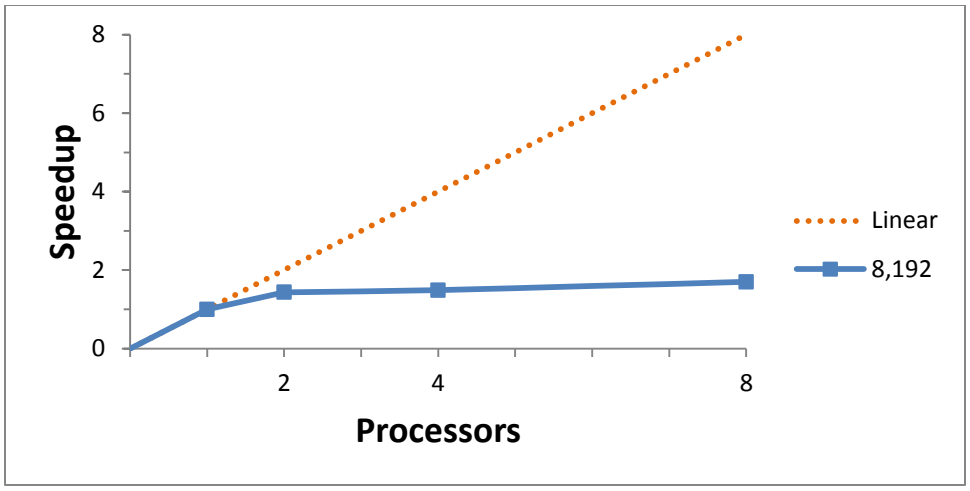
Phases	Execution Times (in seconds)		
	2Pes	4PEs	8PEs
Phase 1	0.00002	0.00003	0.00002
Phase 2	6.320	3.172	1.607
Phase 3	0.050	0.098	0.110

(d) Data Size of $n=2^{19}$, No. of Iterations: 25

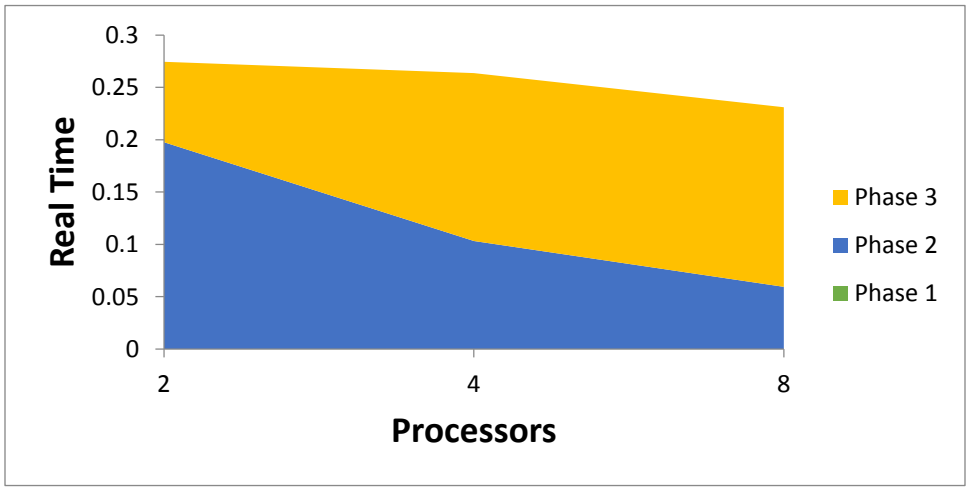
Phases	Execution Times (in seconds)		
	2Pes	4PEs	8PEs
Phase 1	0.00002	0.00003	0.00002
Phase 2	12.177	6.092	3.080
Phase 3	0.048	0.077	0.079

(e) Data Size of $n=2^{21}$, No. of Iterations: 12

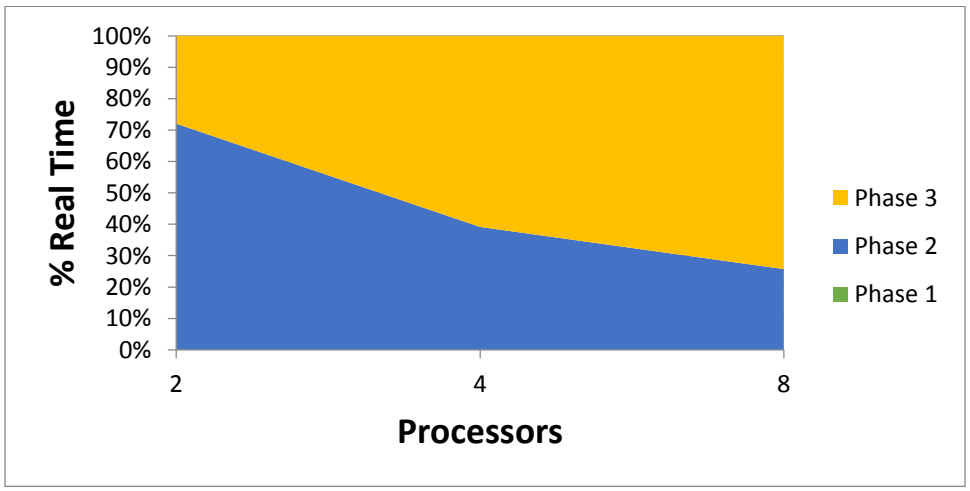
Table 4: Phase-by-Phase Execution times (in seconds) for varying *n,* with *d=8* and *k=8*.



(a) Speedup Curve

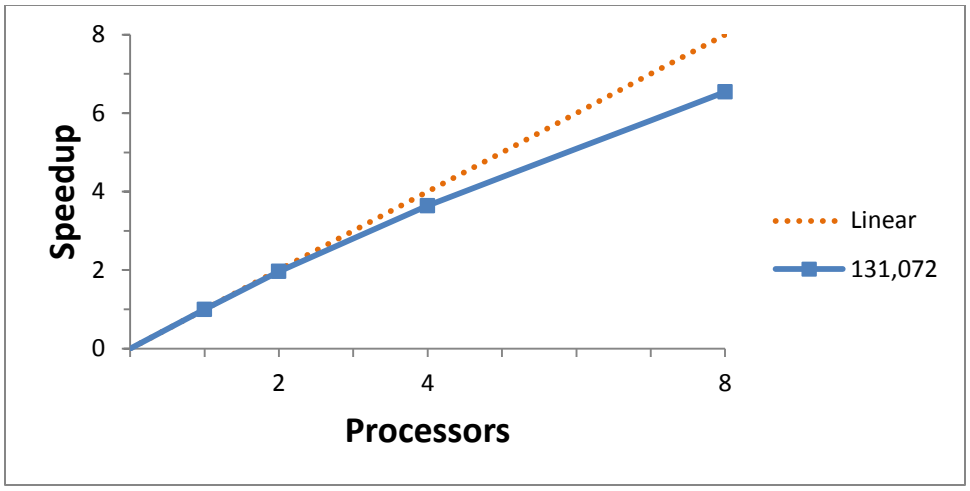


(b) Phase-by-Phase Analysis, real time (seconds)

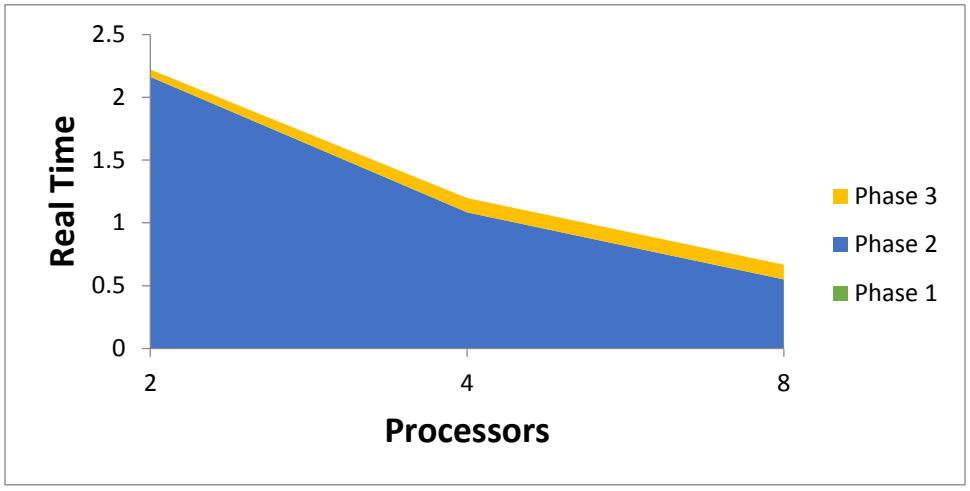


(c) Phase-by-Phase Analysis, percentage of real time

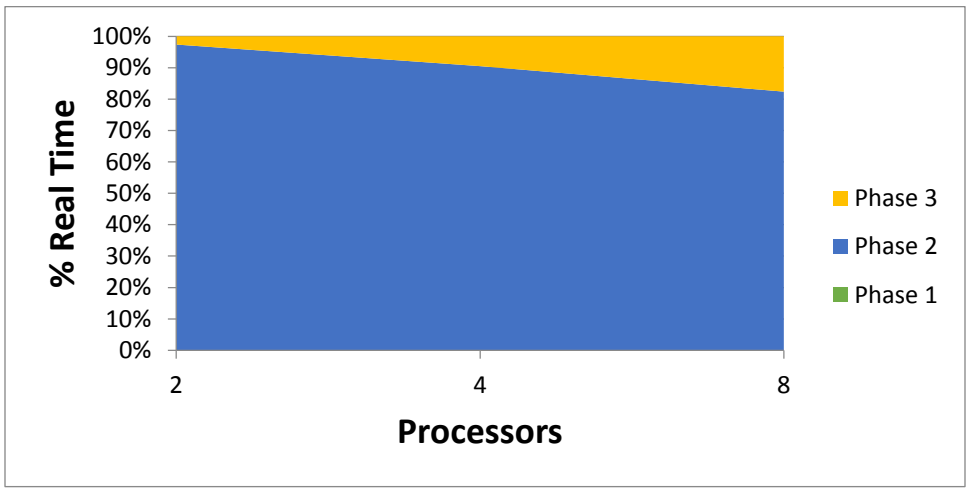
Figure 8: Phase-by-Phase Analysis for $n = 2^{13}$, with $d=8$ & $k=8$.



(a) Speedup Curve

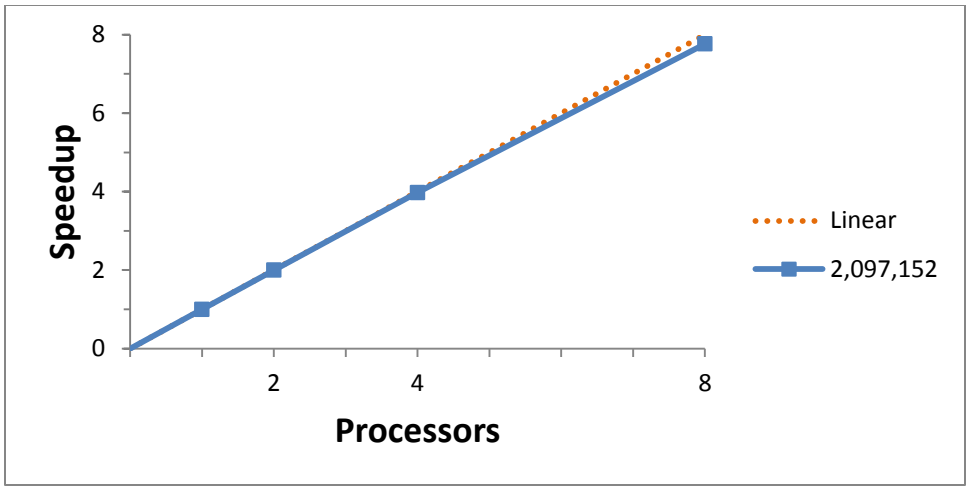


(b) Phase-by-Phase Analysis, real time (seconds)

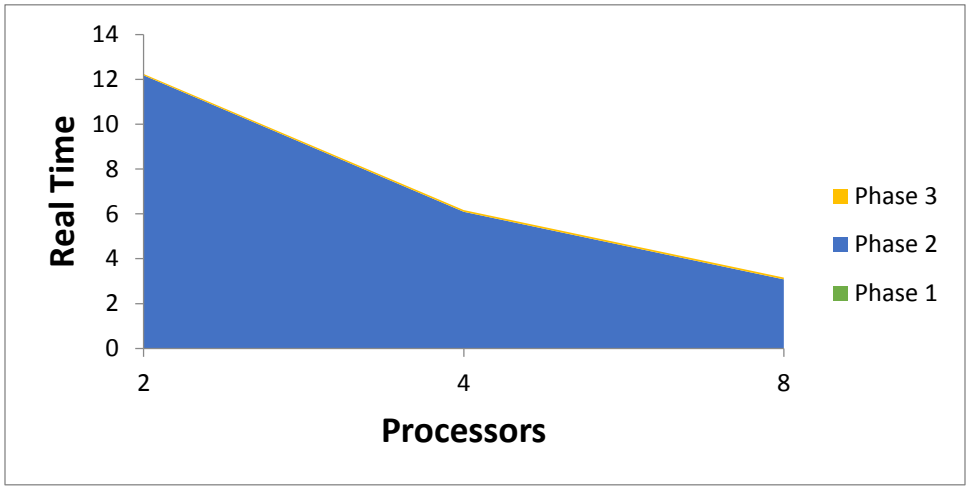


(c) Phase-by-Phase Analysis, percentage of real time

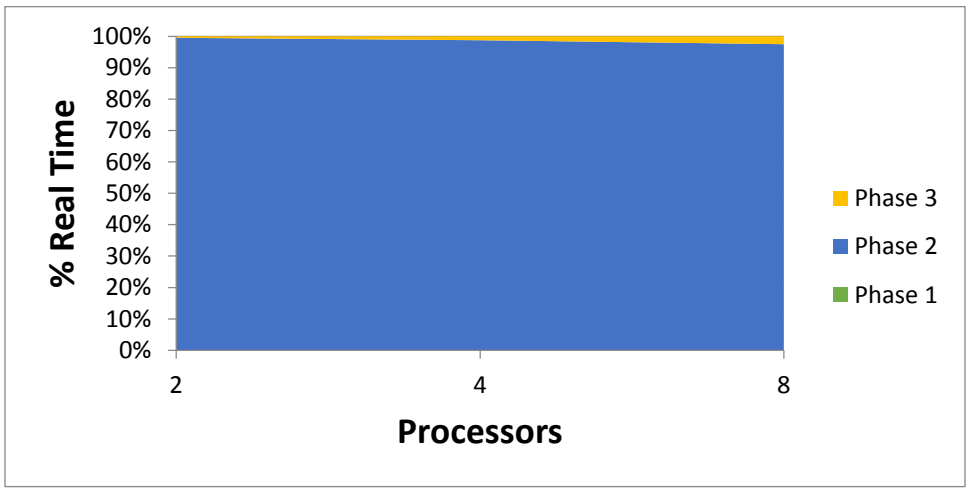
Figure 9: Phase-by-Phase Analysis for $n = 2^{17}$, with $d=8$ & $k=8$.



(a) Speedup Curve



(b) Phase-by-Phase Analysis, real time (seconds)



(c) Phase-by-Phase Analysis, percentage of real time

Figure 10: Phase-by-Phase Analysis for $n = 2^{21}$, with $d=8$ & $k=8$.

Varying *n*: In all the experiments, Phase 1 i.e. Initialization is not noticeable as the amount of computation involved in calculating initial centroids and broadcasting them to all processors is negligible when compared to Phase 2 and Phase 3. Although Phase 1 is sequential with only single processor working, experiments confirm that its sequential implementation does not hurt granularity due to its very short span.

Phase 2 is computation intensive and processors spend *d*k* iterations in calculating Euclidean distance between each data point and the cluster centroids. This phase has no communication and thus granularity is very high and it can be seen in Table 4 - (d), (e) that when we double the number of processors the execution time for this phase is almost halved.

Phase 3 is communication intensive. Each processor sends the local sum and count of all the data points assigned to each cluster and then computes their average. Additionally, reduction operation is performed to compute global Mean Squared Error *mse* value. One can observe from %Real time graph in Figure 9(c) that for a fixed value of n, as the number of processers increase, time spent in Phase 3 increases. Figure 8(c), 9(c) show that when *n* increases with fixed processors, communication decreases and granularity improves.

Figure 10(b)-(c) show that communication with $n=2^{21}$ is negligible and because of coarse granularity speedup is almost linear as shown in Figure 10(a). Figure 8(b)-(c) with $n=2^{13}$ show the effect of fine granularity on speedup where communication overhead for 4 and 8 processors is increased so much that speedup curve gets almost flat after 2 processors as observed in the Figure 8(a).

Phases	Execution Times (in seconds)		
	2PEs	4PEs	8PEs
Phase 1	0.00004	0.00005	0.00004
Phase 2	11.259	5.598	2.928
Phase 3	0.048	0.134	0.084

(a) Dimensions *d = 2*, No. of Iterations: 31

Phases	Execution Times (in seconds)		
	2PEs	4PEs	8PEs
Phase 1	0.00003	0.00003	0.00003
Phase 2	18.786	9.451	4.707
Phase 3	0.089	0.167	0.308

(b) Dimensions *d = 4*, No. of Iterations: 33

Phases	Execution Times (in seconds)		
	2PEs	4PEs	8PEs
Phase 1	0.00002	0.00003	0.00002
Phase 2	12.178	6.093	3.080
Phase 3	0.058	0.078	0.080

(c) Dimensions *d = 8*, No. of Iterations: 12

Table 5: Phase-by-Phase Execution times (in seconds) for varying *d,* with $n=2^{21}$ and *k=8*.

Phases	Execution Times (in seconds)		
	2PEs	4PEs	8PEs
Phase 1	0.00003	0.00003	0.00002
Phase 2	3.555	1.789	0.899
Phase 3	0.062	0.034	0.049

(a) Clusters k = 2, No. of Iterations: 14

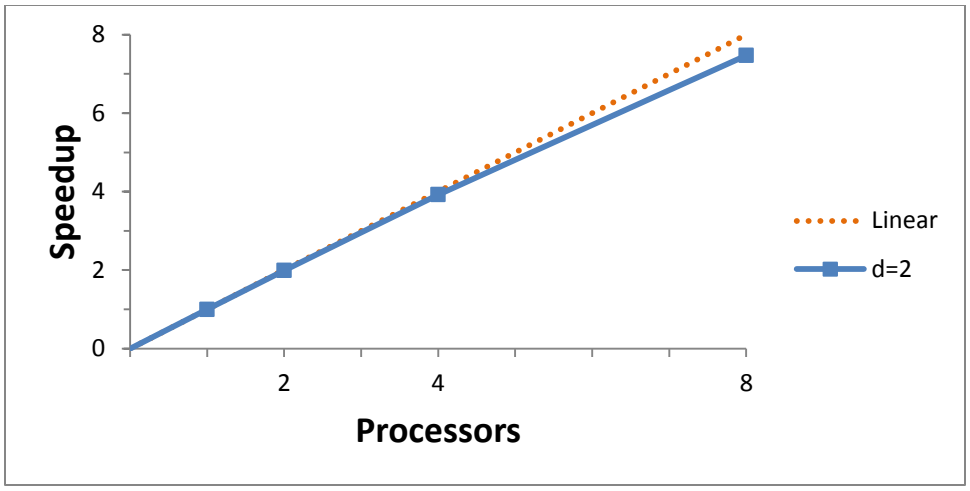
Phases	Execution Times (in seconds)		
	2PEs	4PEs	8PEs
Phase 1	0.00002	0.00003	0.00002
Phase 2	8.046	4.017	2.032
Phase 3	0.041	0.064	0.061

(b) Clusters k = 4, No. of Iterations: 16

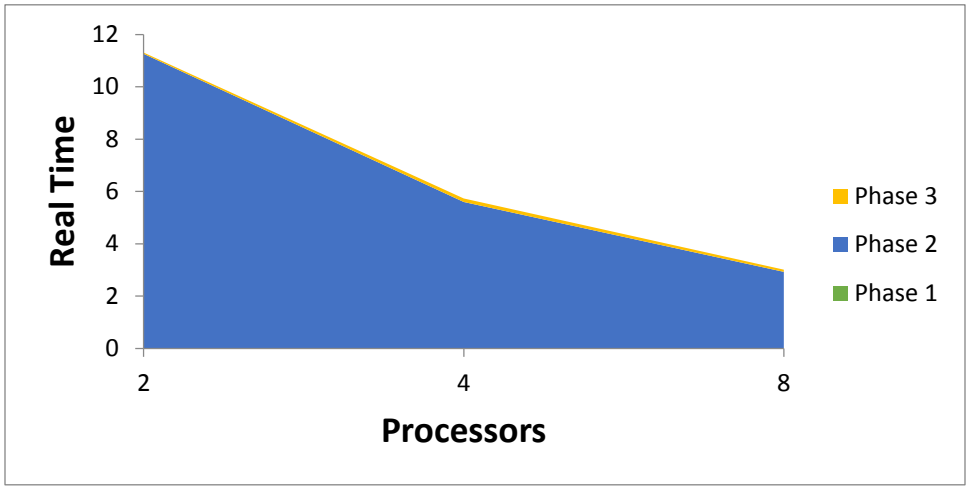
Phases	Execution Times (in seconds)		
	2PEs	4PEs	8PEs
Phase 1	0.00003	0.00003	0.00003
Phase 2	33.450	16.750	8.497
Phase 3	0.137	0.214	0.224

(c) Clusters k = 16, No. of Iterations: 18

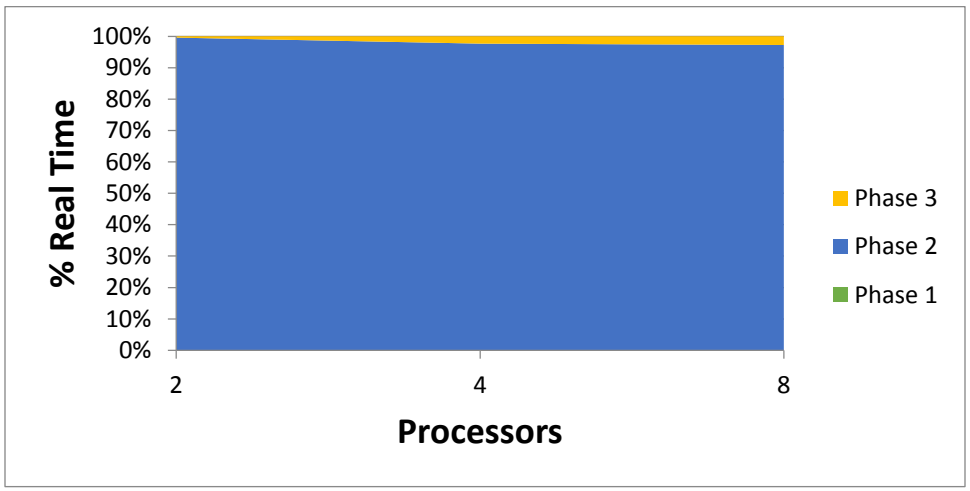
Table 6: Phase-by-Phase Execution times (in seconds) for varying *k,* with $n=2^{21}$ and *d=8*.



(a) Speedup Curve

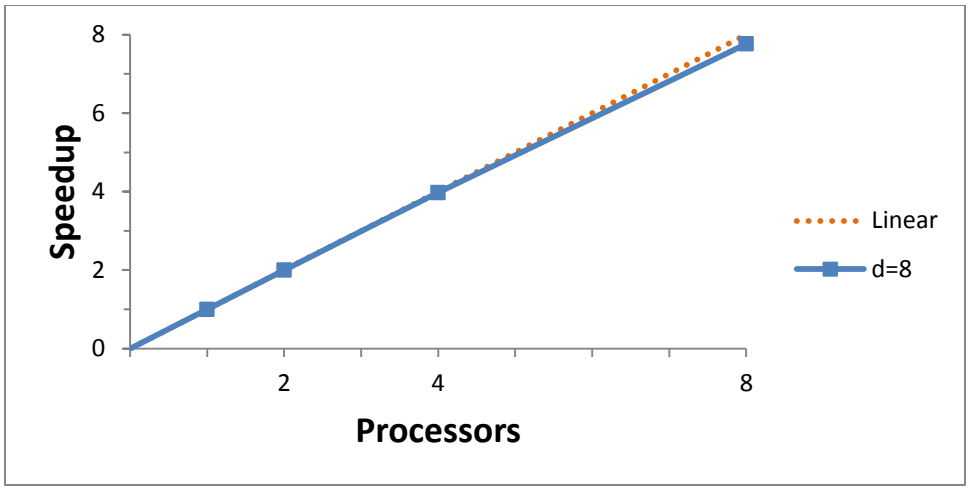


(b) Phase-by-Phase Analysis, real time (seconds)

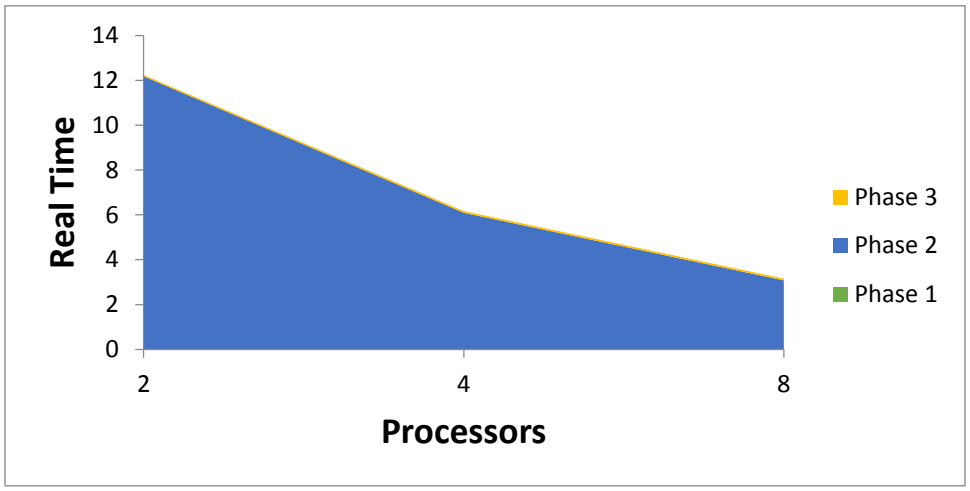


(c) Phase-by-Phase Analysis, percentage of real time

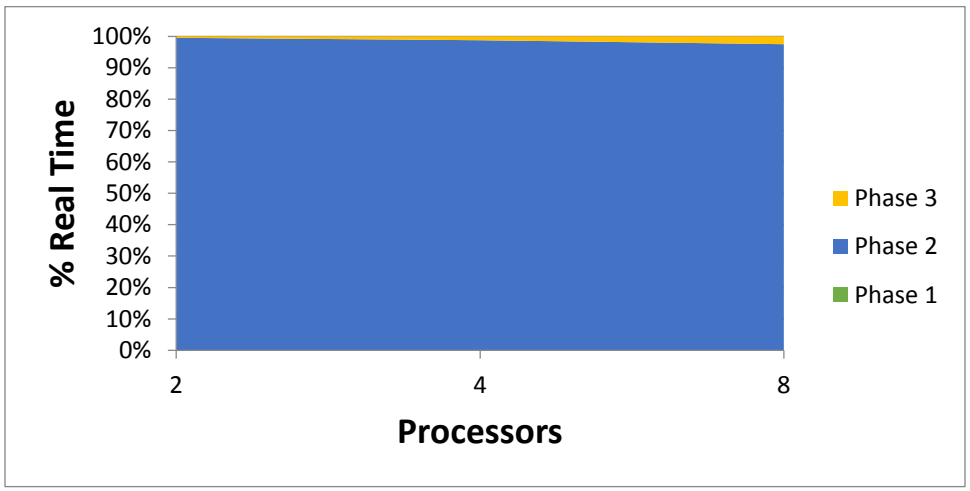
Figure 11: Phase-by-Phase Analysis for *d=2,* with $n = 2^{21}$ & *k=8*.



(a) Speedup Curve

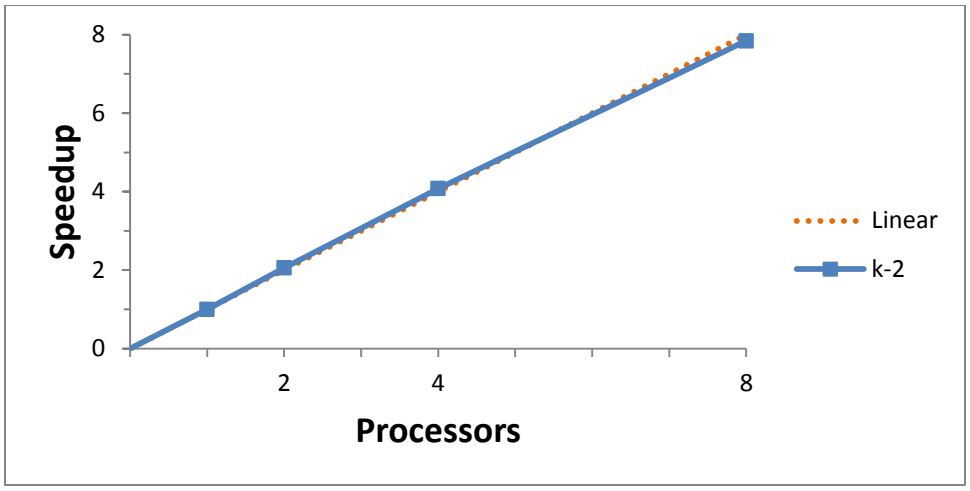


(b) Phase-by-Phase Analysis, real time (seconds)

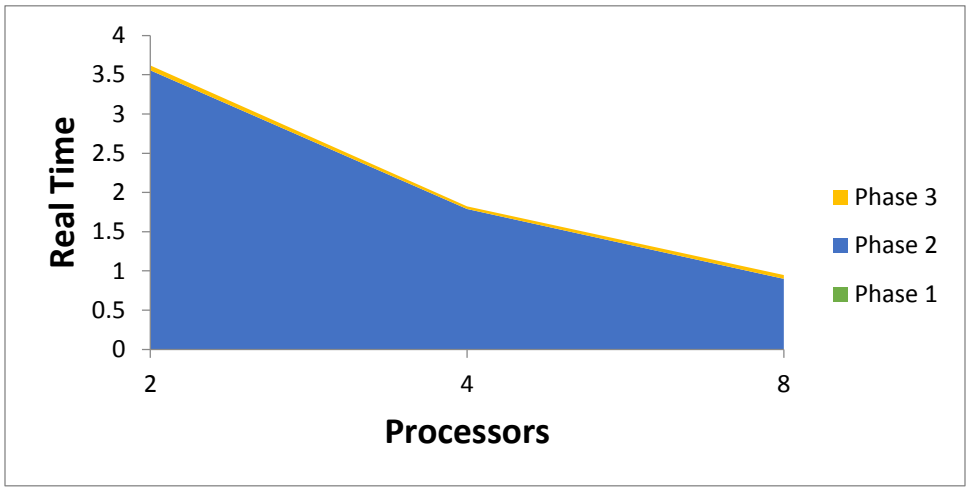


(c) Phase-by-Phase Analysis, percentage of real time

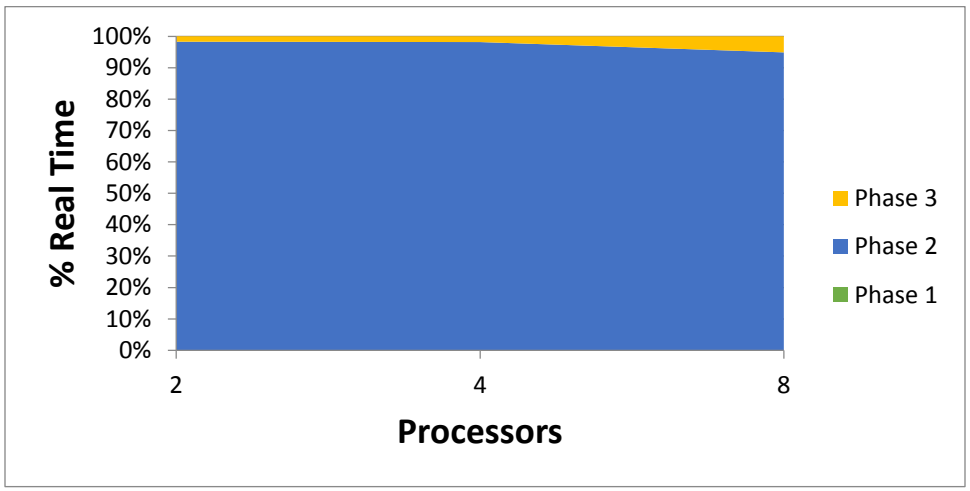
Figure 12: Phase-by-Phase Analysis for *d=8,* with $n = 2^{21}$ & *k=8*.



(a) Speedup Curve

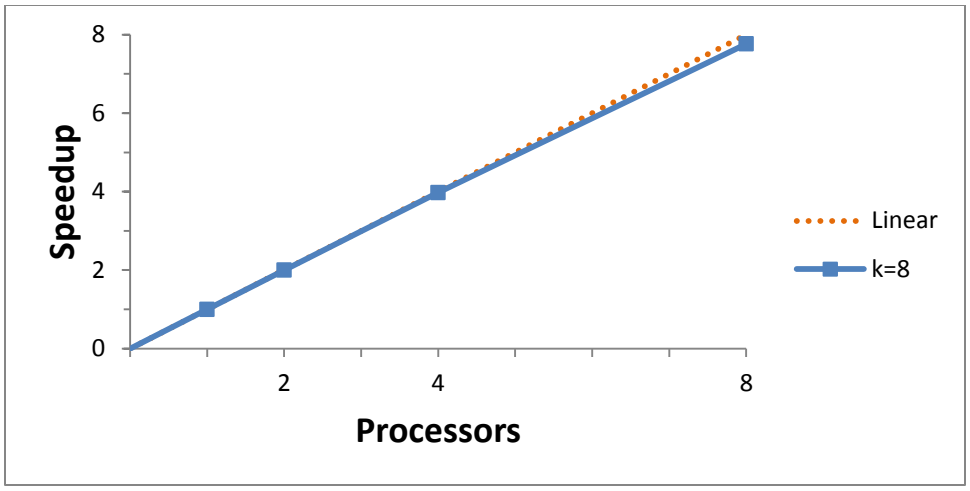


(b) Phase-by-Phase Analysis, real time (seconds)

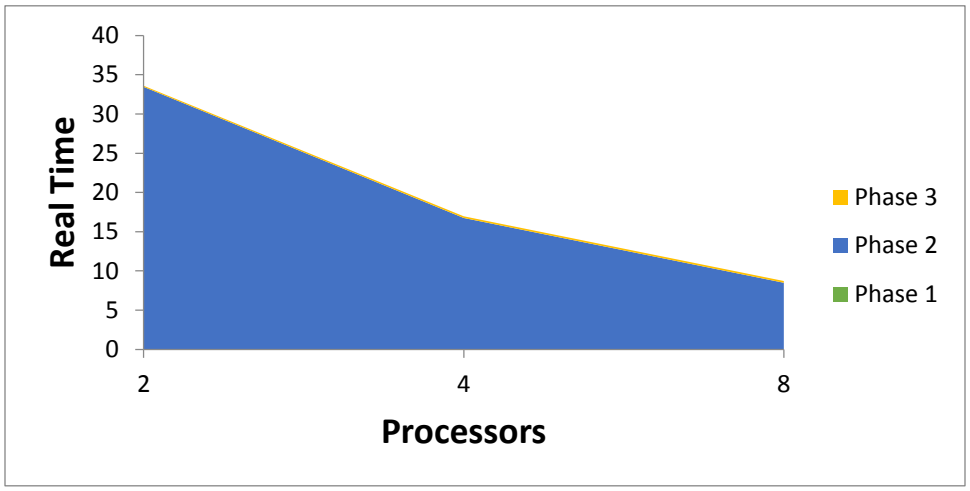


(c)Phase-by-Phase Analysis, percentage of real time

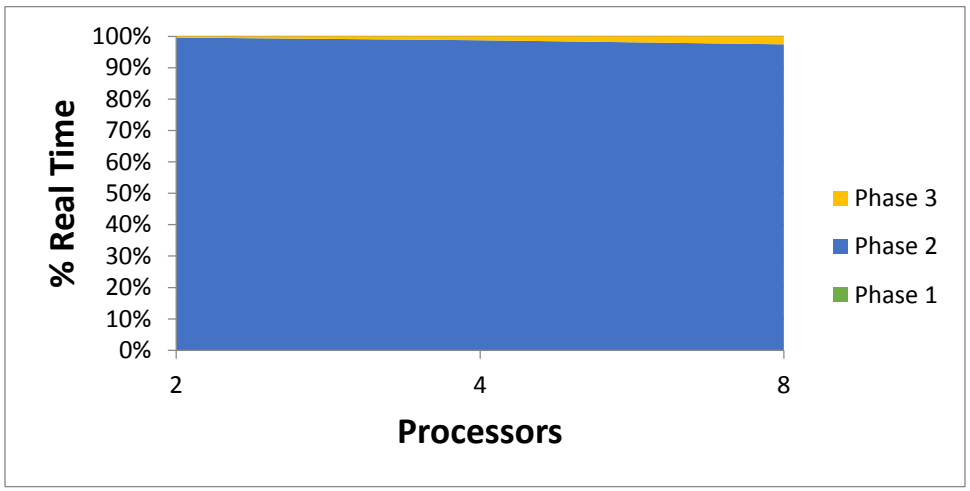
Figure 13: Phase-by-Phase Analysis for *k=2,* with $n = 2^{21}$ & *d=8*.



(a) Speedup Curve



(b) Phase-by-Phase Analysis, real time (seconds)



(c) Phase-by-Phase Analysis, percentage of real time

Figure 14: Phase-by-Phase Analysis for *k=8,* with $n = 2^{21}$ & *d=8*.

Varying *d* & *k*: The behavior for varying both *d* and *k* is almost similar. Phase 1 for both of them is even similar to varying *n* case where the amount of time spent is negligible. Thus, Phase 1 doesn't affect granularity and overall speedup.

Phase 2 for both *d* and *k* with $n=2^{21}$, takes almost all of the time in overall computation as seen in Fig 11-14(b)-(c). Phase 3 which is communication extensive is comparatively very small and this is because of high granularity of $n=2^{21}$. For varying *d* and *k* for fixed processors, we see that time spent on communication remains almost same which shows that *d* and *k* doesn't impact on performance. The overall granularity is affected only by varying *n* as observed in the original paper.

3. Conclusion

This paper presents a performance characterization of parallel k-means with MPI. We experimented on a four-node cluster using several datasets to inspect the behavior in three different dimensions: number of data points (*n*), dimensions (*d*) and desired clusters (*k*); and two kinds of metrics: speedup and scale-up.

The aspect of granularity can be witnessed in the results produced during experimentation. The evaluation of results show that small number of data points has a considerable negative impact on speedup, due to fine granularity. On the other hand, large datasets increase granularity and better offset these communication overheads introduced by adding processors.

In the general case, we proved that parallel k-means scales almost linearly, only minimally affected by synchronization and communication. The impact of such communication is small due to the fact that it only has to exchange *k*d*i* numbers, which in practical scenarios is small amount of data ($k, d, i \ll n$).

References

1. *A Data-Clustering Algorithm on Distributed Memory Multiprocessors.* **Dhillon, Inderjit S. and Modha, Dharmendra S.** London, UK : Springer-Verlag, 2000. Revised Papers from Large-Scale Parallel Data Mining, Workshop on Large-Scale Parallel KDD Systems, SIGKDD. pp. 245-260.

2. *An algorithm for generating artificial test clusters.* **Milligan, Glenn W.** s.l. : Springer-Verlag, 1985. Psychometrika. Vol. 50, pp. 123-127. 1.

3. *On the Reproducibility of MPI Reduction Operations.* **Balaji, P and Kimpe, D.** Zhangjiajie, China : s.n., 2013. High Performance computing and Communications HPCC 2013.

4. *K-means++: The Advantages of Careful Seeding.* **Arthur, David and Vassilvitskii, Sergei.** New Orleans, Louisiana : Society for Industrial and Applied Mathematics, 2007. Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 1027-1035.

5. *Scalable K-means++.* **Bahmani, Bahmani, Bahman, et al., et al.** s.l. : Proc. VLDB Endow., 2012. pp. 622-633.