A Critical Performance Study of Memory Mapping on Multi-Core Processors: An Experiment with k-means Algorithm with Large Data Mining Data Sets

S. N. Tirumala Rao
JNTUK
Kakinada
A.P, India

E. V. Prasad
JNTUK
Kakinada
A.P, India

N. B. Venkateswarlu
AITAM
Tekkali
A.P. India

ABSTRACT
Increased availability of Multi-Core processors is forcing us to re-design algorithms and applications so as to exploit the available computational power from multiple cores. It is not un-common to employ memory mapping of files in applications involving huge I/O bandwidth to improve the response/service times. This paper mainly focuses on performance of memory mapped files on Multi-Core processors. Experiments are carried out with k-means algorithm, a popular Data mining (DM) clustering algorithm, to explore the potential of Multi-Core hardware under OpenMP API and POSIX threads. Observations are made both with static and dynamic threads of OpenMP. Experiments are also conducted with both simulated and real data sets. Experiments indicate that memory mapping of files gives considerable benefit on Multi-Core processors also. In addition, the benefit increased with increased physical memory size. Also, the benefit of memory mapping with the selected algorithm is increasing with number of cores.

Categories and Subject Descriptors
C.1 processor architectures , C.1.2 Multiple Data Stream Architectures (Multiprocessors) , Parallel processors ,C.1.4 Parallel Architectures H. information Systems H.3.3 Information Search and Retrieval ,Clustering, D. Software ,D.1 Parallel programming.

General Terms
Performance, Design, Experimentation and Verification.

Keywords
OpenMP, mmap(), fread(), POSIX threads, scalability, Multi-Core and k-means.

1. INTRODUCTION
In the recent years, many network and other applications, which demand huge I/O(Input/Output) overhead, are reported to be using a special I/O feature known as mmap() to improve their performance. For example, John et.al., [10] studied the performance of Apache Server. In addition, Avadis Tevanian et al., [1] also reported that there would be the CPU time benefit by making use of memory mapping rather than conventional I/O in Mach Operating Systems. Nevertheless, Joseph, et al., [3] had also revealed that there was clear cut performance advantage of mmap over fread and iostream, because of the huge datasets to be accessed. Most data mining algorithms also must consider the I/O actions carefully, as they would minimize their effects. The well known commands in LINUX like *grep, fgrep, egrep* and *find* also use memory mapping concept for large data files. Most of the commercial data mining tools and public domain tools such as Clusta, Xcluster, Rosetta, FASTLab, Weka etc., support DM algorithms, which accepts data sets in flat file form or CSV (comma separated values) form. Thus, they use standard I/O functions such as fgetc(), fscanf(). However, fread () is also in wide use with many DM algorithms [14, 15].

Reading a byte from a file by using fread() incur three I/O operations such as (1)Removing a cache block to accommodate new disk block. (2)Reading this disk block into the buffer cache. (3) Copying the same block into the process address space from buffer. If the removed cache block is dirty, it needs another I/O to update dirty block to disk. However in the memory mapping, entire file is mapped into the process address space such that the file is treated as an extension of virtual address space of the process. Here, when we need a byte from the mapped file, the block (Having the byte) is directly copied to memory. So the major amount of difference comes only before applying the algorithm on data. The percentage of benefit of mmap() in algorithm time is less compared to the percentage benefit in reading the file. S.N.Tirumala Rao et al.,[13] studied the benefit of memory mapping with popular data clustering algorithm, k-means. They have reported that on serial computers, use of memory mapped() files reduce the CPU time requirements of the k-means algorithm.

Also, in the literature we may find efforts to parallelize the k-means and other DM algorithms to reduce the CPU time requirements. Manasi N. Joshi [7] describes clustering large data sets as time consuming and processor intensive. The implementation of the parallel version of a popular clustering algorithm exploits the inherent data parallelism in the *k*-means algorithm and makes use of the message-passing model. Dhillon I.S et al.[5] proposes a parallel implementation of k-means clustering algorithm based on message passing model. It also proves that the scale up of algorithm is with the increase of the number of data points.

More over, the scenario is changing in the recent years with the availability of Multi-Core processors. Obviously, one may be interested to get the maximum benefit from the HW (Hardware). The amount of improvement in performance by the use of a Multi-Core processor is dependent on the software algorithms and their implementation. In particular, the possible gains are limited by the part of the software that can be "parallelized" to run on multiple cores simultaneously; this effect is proposed by Amdahl's law. Software benefits are from Multi-Core architecture, where code can be executed in parallel. Under most operating systems, this requires a code to execute in separate threads or processes. Each application runs on a system of its own process, so multiple applications will be benefited from Multi-Core architectures. Each application may also have multiple threads but, it must be specifically mentioned to utilize multiple threads. As of now, the above stated popular DM tools both commercial/open-source have not been utilized Multi-Core architecture effectively so far.

In this paper, we propose to study the benefit of memory mapping on Multi-Core processors. For the purpose of experimentation, we have used k-means [6] algorithm which is reported to be second widely used DM algorithm. Two widely used multi-threading utilities known as POSIX threads and OpenMP threads are used while mapping the selected algorithm on Multi-Core processors.

2. MAPPING THE ALGORITHMS TO MULTI-CORE PROCESSORS

In the literature, we may find researchers presently using OpenMP and POSIX threads to exploit the available parallelism with Multi-Core machines.

2.1 OpenMP

OpenMP is an API which allows the user to introduce parallelism to the program with minimal time to modify source code. Of course, every one will be interested to achieve efficiency and scalability without much development time. Thus, OpenMP is planned to reduce development time. In a nutshell, OpenMP API contains some pragmas with which one can specify the compiler that a section of code shall be run in parallel. More over if the compiler does not support OpenMP pragmas, the program need not be modified, and simply it will run in serial mode [9]. OpenMP automatically parallelizes of some parts of code. Hence, it introduces some overhead in the computation that makes it significantly heavier. To expect some improvements, the granularity of the threads created by OpenMP has to be the coarsest possible. Otherwise, the total computational time is dominated by the time needed for creating the threads. OpenMP allows user only to control the number of threads which are created in a loop or for dealing with a part of the program [11].

In OpenMP, there is no global way to control the *total* number of threads that are run on the machine. Controlling the total number of created threads is more difficult; it may become important which would induce a great loss of efficiency. Thus, to avoid this, we have programmed two functions, one that effectively has a parallel section and the other is purely sequential. The recursive calls need not create new threads. They are all done calling the purely sequential function. A shared counter on the number of threads allows controlling it. In OpenMP there is no need for synchronization for accessing particular shared variable. OpenMP provides implicit locking facility. This technique allows significant speed up in most cases; it induces significant changes in the source code. It distributes the execution of the associated statement among the members of the team that encounter it. 1) Work sharing construct do not launch new threads. 2) There is no barrier upon entry to work-sharing construct. 3) There is an implied barrier at the end of a work sharing construct. OpenMP assigns one task per processor. Each user thread is mapped one system thread [15].

2.1.1 Scheduling

OpenMP supports loop level scheduling. This defines how loop iterations are assigned to each participating thread**.** Scheduling types include:

- *static*
  - Each thread is assigned a *chunk* of iterations in fixed fashion (round robin).
- *dynamic*
  - Each thread is initialized with a *chunk* of threads, then as each thread completes its iterations, it gets assigned the next set of iterations.
- *guided*
  - Iterations are divided into pieces that successively decrease exponentially, with *chunk* being the smallest size. This is a form of load balancing that is less.
- *runtime*
  - Scheduling is deferred until run time. The schedule type and chunk size can be chosen by setting the environment variable **OMP_SCHEDULE**.

2.2 Posix threads

The POSIX threads API does not aim at automatic parallelization. It just allows the programmer to create new threads. As the programmer has to decide what will be performed in parallel, it allows a full control on what occurs in each thread. The main drawback of using directly POSIX thread API inside the arithmetic is that it leads to perform many system calls that may hinder the performances on small examples. The programmer has to adopt synchronization mechanism to access a particular shared variable. POSIX applications are not portable as OpenMP. Posix threads provide explicit locking facility. POSIX threads assign many short tasks for few processors. Several user threads are mapped to one system thread. It does not have a dedicated kernel thread [15].

In our experimentations, we have used first two options.

2.3 Implementing Parallel K-Means Algorithm

This section mainly focuses on implementing a parallel k-means with fread() and mmap().

2.3.1 The k-means Algorithm

0. Select some number of clusters, k.

1. Select initial cluster points.

2. Cluster each record do_clustering() method.

3. Calculate new cluster centers. Compare the new cluster points with initial cluster points. If both are same, stop and print the new cluster centers as output. Otherwise, take new cluster centers as initial cluster centers and go to step 2. Computational complexity of k-means algorithm is O(kndI), where n=number of samples or records to be clustered, d=dimensionality, and I= number of iterations.

For fuller details of serial k-means algorithms, readers are advised to refer to any popular DM books [6]. The parallel versions of k-means were implemented with OpenMP API using data parallelism technique. Main computational burden in k-means algorithm is step 2. That is, we need to classify each record to a cluster by calculating the distance between itself and all the initial clusters. This is the component which we propose to parallelize using both OpenMP and POSIX threads to get the benefit of Multi-Core processors. In a nutshell, data parallelism divides the input data into independent sets and processes simultaneously. This is used in our algorithm also. The pseudo code for the implementation of parallel k-means is stated below.

2.3.2 Pseudo code

```
0.    Select some number of clusters, k.
1.    Select initial cluster points.
2.    omp_set_num_threads(here no of threads are 5);
3.    /* Start of parallel code    */
      #pragma omp parallel sections
      {
         #pragma omp section
          do_clustering(0);

         #pragma omp section
           do_clustering(1);

         #pragma omp section
           do_clustering(2);

         #pragma omp section
           do_clustering(3);

         #pragma omp section
            do_clustering(4);
      }
```

/*The data is evenly distributed among the entire do_clustering functions. The #pragma omp parallel sections, assign each do_clustering function to a separate thread. Thread wise cluster wise totals are calculated. */

4. Implicit barrier for the threads /* End of parallel code */

5. Sum up the cluster wise totals and cluster wise count.

6. Calculate the new cluster points. Compare the new cluster points with previous cluster points. If both are same, go to step 7, other wise Set new cluster points as current cluster points and go to step 3.

7. Display final cluster points.

As mentioned above, OpenMP allows for some explicit control when parallelizing loops. In the examples cited above, OpenMP uses internal heuristics to allocate each iteration of the loop to a particular *thread*[1].

The omp_set_num_threads represents an OpenMP runtime library routine, used to set the required number of threads. In this method, the number of threads is set as 5. An OpenMP program begins as a single thread of execution, called Master thread. The master thread executes sequentially, when it encounters a parallel construct, the master thread creates a team of specified number of (in this example it is 5) threads and master thread becomes a member of the team. When a master thread encounters the line *#pragma omp* parallel sections, a team of threads are created to execute parallel sections. Each section is identified by an OpenMP #pragma omp section. The do_clustering(0) function, with parameter zero, clusters first 1/5 of given samples. The steps to distribute the records are 1) Number of records to clustered = Total number of records/ Number of threads. 2) Starting row = Number of records to be clustered * Thread number. 3) Ending row = Starting row + Number of records to be clustered. The remaining samples are equally distributed among the remaining threads.

3. EXPERIMENTAL WORK

For our experimentation, we have used the following computers.

1. Intel(R) Core(TM)2 Quad CPU Q6600 @2.40GHz processor, 1 MB Cache memory.
2. Intel Pentium Dual-Core 2.80 GHz processor with 1 GB RAMS, 1 MB Cache.
3. Intel Single Core processor Speed 2.2GHz with 1 MB cache.

In order to standardize our observations, we have used same RAM's, HD(Hard Disk), and Cache with all the selected computers. Fedora 9 Linux (Kernel 2.6.25-14, Red Hat version 6.0.52) equipped with GNU C++ (*gcc* version 4.3) is used in our study. We have used both OpenMP and POSIX threads to study their performance on Multi-Core processors.

In our experiments, Pocker hand data set [12] and randomly generated data set are used. Random data set is generated to have 20 million records with the dimensionality of 20. Pokers set have 1 million records with ten attributes (dimensions). All the algorithms are tested with a maximum data file size of 2 GB. In all the graphs, number of clusters as K, dimensions as d, number of records as N.

In a nutshell, the following forms of selected algorithm are used.

1. Parallel implementation of k-means with fread () (**PFk-means**)

[1] http://www.nersc.gov/nusers/help/tutorials/openmp/do_clause.html

2. Parallel implementation of k-means with mmap () (**PMk-means**)
3. Serial implementation of **k-means** with mmap() (**SMk-means**)
4. Serial implementation of **k-means** with fread() (**SFk-means**)

In addition, we have used the following notations in our figures which convey type of machine, threads package used, data set used and OS.

1. QORL signifying Quad-Core - OpenMP –Random Data Set – Linux OS
2. DORL signifying Dual-Core - OpenMP- Random Data Set – Linux OS
3. QOPL signifying Quad-Core – OpenMP – Pokers Data Set – Linux OS
4. DOPL signifying Dual-Core – OpenMP - Pokers Data Set – Linux OS
5. DPsRL signifying Dual-Core – Posix Threads – Random Data Set – Linux OS
6. QPsRL signifying Quad-Core – Posix Threads – Random Data Set – Linux OS
7. QRL Signifying Quad-Core – OpenMP and POSIX threads - Random Data Set – Linux OS

The ***PFk-means*** and *PMk-means* are parallelized using OpenMP API and POSIX Threads. In this section, the focus is laid on analyzing the performance improvement in PMk-means clustering algorithm over PFk-means clustering, with respect to its execution times(User-time, System-time and Real-time). The observations of SFk-means and SMk-means are also recorded.

We have used time command available in Linux to measure the times consumed by our algorithms. When the program completes its execution, time command reports user time, system time, and real time (elapsed time). The user time is the amount of time the CPU spends in executing the users program. The system time is, the amount of time the CPU spends in kernel mode. The real time is the time gap between the invocation of the program and termination of the program. Primarily, we have experimented our algorithms by bringing the Linux system in single user mode to avoid effect other programs. However, some more experiments were carried out while few tens of people are logged in the machine. Our conclusions are observed to be independent of number of users.

S.N.Tirumala Rao et al.,[13] reported the benefit of memory mapping over fread(), read(), fgetc() based solutions with serial k-means algorithm on single core processor. In order to investigate its benefit in the selected Dual-Core and Quad-Core systems, we have conducted numerous experiments by varying different parameters such as number of records, number of clusters and dimensionality. Figures shows benefit of memory mapping concept with conventional fread() on Dual-Core and Quad-Core machines with single thread, i.e., serial K-Means.

Observations indicate that memory mapping of data files gives speed up on Multi-Core processors also and the speed up increases in a number of dimensions, number of records and number of clusters. Also, this supports the scalability of the concept on Multi-Core processors like serial processors which is reported in [13]. This paper also analyses the performance of parallel k-means with static threads and dynamic threads of OpenMP. In POSIX thread based k-means also data is divided into five equal parts for five threads. The same algorithms run under Dual-Core and Quad-Core architectures.

3.1 Quad-Core and Dual-Core and OpenMP API

The observations of parallel k-means algorithm using five OpenMP static threads were analyzed. In this method, data is divided into five equal parts among the static threads. That is, if we happened to have N records, each thread is made to run the k-means algorithm on N/5 records. That is, first thread carries k-means algorithm on first N/5 records; the second thread works on next N/5 records, and vice versa. For all the threads, initial cluster centres will be taken as same. We have experimentally found that our algorithms are consistently giving better results for five numbers of static threads. Thus, we have given the reports of our results for the same. We believe that this may not be the case with when number of cores increases.

OpenMP based PMk-means algorithm consistently gave benefit in real time, user time and system time over PFk-means algorithm, irrespective of the number of records, dimensions, clusters and size of RAM in both Dual-Core and Quad-Core machines ***(see Figure 1 to 3)***. The percentage of mmap() benefit in parallel k-means algorithm is higher compared to serial k-means algorithm which were depicted in the ***Figure 4 to 5***. The effect of RAM size is observed by varying size of the RAM in steps of 0.5GB from 0.5GB to 3GB i.e., 0.5GB, 1GB, 1.5GB, 2GB, 2.5GB, 3GB. The percentage of PMk-means real time and user time benefit decreases as the size of RAM increases in both the Dual-Core and Quad-Core. The percentage of PMk-means system time increases as RAM increases which were depicted in *Table 1 and 2* in annexure. Our observations corroborates that memory mapped versions of our experiments take less time compared to equivalent fread() based versions.

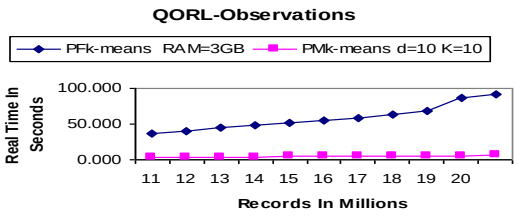

Figure 1: Observations of *k*-means on Quad-Core machine

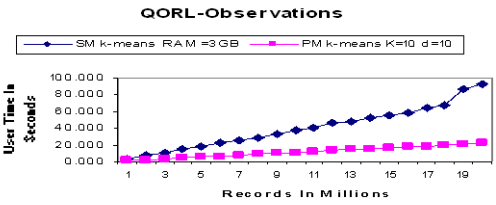

Figure 2: Observations of k-means on Quad-Core machine.

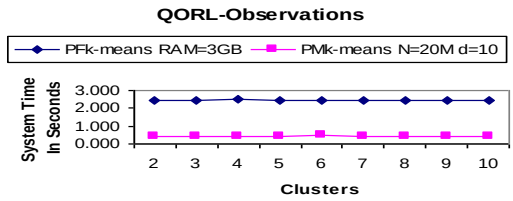

Figure 3 Observations of k-means on Quad-Core machine.

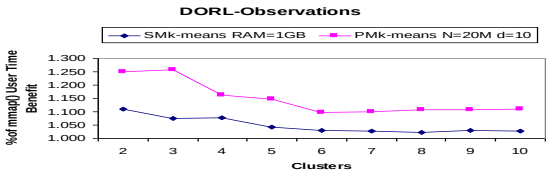

Fig 4: Observations of k-means on Dual-Core.

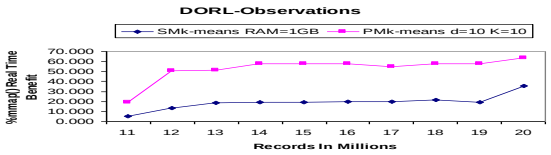

Figure 5: Observations of k-means on Dual-Core.

In OpenMP k-means performance with static threads also compared to the dynamic threads by varying number of threads and distributing data equally among threads. The number of threads is varied and observations are noted. It could be observed that PMk-means gives more real time, user time, system time benefit over PFk-means with either static threads or dynamic threads of OpenMP under both the architectures, which can be observed from *Figure 6 and 7*.

The OpenMP based PMk-means real time benefit increases as RAM increases where as the speed variation is minor in system time and user time in both Dual-Core and Quad-Core. In order to explore the reasons, we have carried out separate timing measurements for only processing (i.e., carrying out k-means clustering algorithm) and initial data reading. This indicates that major benefit is coming from the second one which is also reported in [13]. Also, the OpenMP based PFK-means real time and system time benefit increases with RAM size. However, the PFk-means real time benefit is very high which may be also attributed to the same reason mentioned above. The OpenMP based PMk-means real time, user time and system time and the benefit increases with the number of cores increases irrespective of number of records, clusters and dimensions. Also, the OpenMP based PFk-means real time and user time speed increases with the number of cores irrespective of number of records, clusters and dimensions. As the number of cores increases the number of threads creation, termination and scheduling time increases, so the system time benefit may automatically be reduced as number of cores increases. The observations of this section are drawn from *Table 3 and 4***.** These observations support that both memory mapped() based and fread() based algorithms are getting benefited by using multiple cores which is shown from ***Figure 8 to 9***.

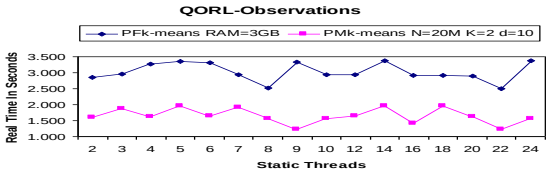

Figure 6: Observations of k-means on Quad-Core machine.

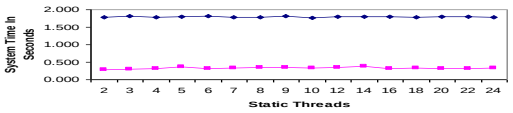

Figure 7: Observations of k-means on Quad-Core machine.

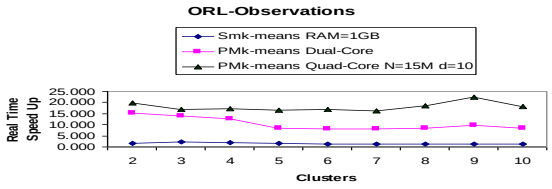

Figure 8: Speed up observations of k-means.

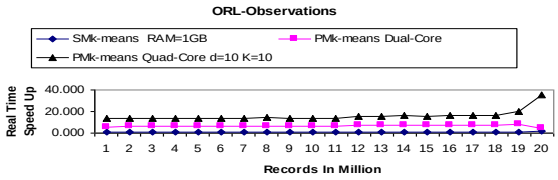

Figure 9: Speed up Observations of k-means.

3.2 Quad-Core and Dual-Core and POSIX API

We have conducted a series of experiments with our algorithm using POSIX threads also. POSIX based PMk-means algorithm gives more real-time and system-time benefit compared to PFk-means algorithm, irrespective of the number of records, dimensions, clusters and size of RAM in both Dual-Core and Quad-Core architectures, which could be observed in ***Figure 10***. This can be attributed to the POSIX thread creation. In POSIX based threading, one process is created to each thread and assigned to a core. Thus, the benefit in real time is falling considerably. But the percentage of PMk-means real time benefit decreases and system time benefit increases as the size of RAM increases which could be observed from *Table 5 and 6*.Also, *Figure 11* indicated that, POSIX based PFk-means algorithm gives more user-time benefit compared to PMk-means algorithm irrespective of the number of records, dimensions, clusters,

except at the size of RAM equal to 0.5GB. The user time benefit decreases as the size of RAM increases, the maximum percentage of benefit is four only, which could be observed from the *Table 7 and 8*.

The POSIX based PMk-means, real time, system time and user time speed increases as the number of cores increases irrespective of number of records, clusters, dimensions and size of RAM. The Quad-Core real time speed is very high as compared to Dual-Core real time speed. The POSIX based PMk-means, real time and system time speed is high compared to PFk-means real time speed and system speed respectively, irrespective of number of records, dimensions and clusters. These conclusions are drawn from observations given *Table 9 and 10*.

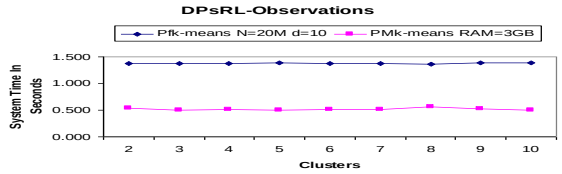

Figure 10: Observations of k-means on Dual-Core machine.

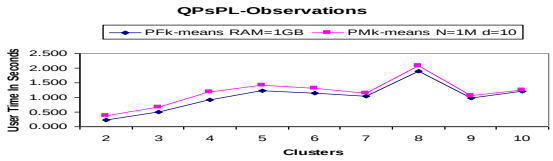

Figure 11: Observations of k-means on Quad-Core machine

3.3 OpenMP VS POSIX Comparison

We have also compared OpenMP and POSIX based algorithms. OpenMP based PMk-means implementation takes less real time, user time and system time compared to POSIX based PMk-means irrespective of the number of records, dimensions and clusters for real data (pokers) and simulated data in both Dual-core and Quad-Core ***(see Figure 12 and 13)***.

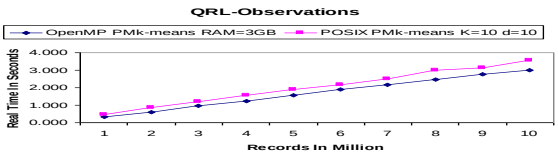

Figure 12: Observations of k-means on Quad-Core machine.

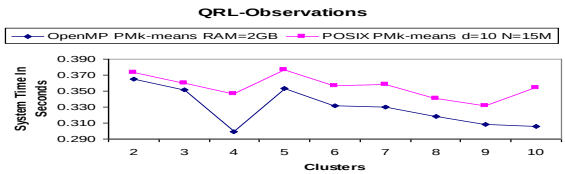

Figure 13: Observations of k-means on Quad-core machine.

4. CONCLUSION

The concept of memory mapping supported by Linux is studied in comparison to commonly used fread() based I/O with k-means clustering algorithm. The computational benefit of mmap() over fread() based algorithm is independent of number of samples, dimensions and number of clusters on Quad-Core and Dual-Core shared memory architecture. The percentage of mmap() benefit is more in parallel k-means than serial k-means in selected Multi-Core machines. It is also observed that the static thread based implementations result in better performance over dynamic threads under Quad-Core architecture with OpenMP. The percentage of system time benefit increases in PMk-means algorithm with the increase of RAM size. The percentage of user time and real time benefit decreases in PMk-means algorithm as size of RAM increases even though considerable amount of benefit could be observed. Parallel k-means with mmap() is more scalable compared to the parallel k-means with fread() irrespective of the number of cores, number of records, number of dimensions, clusters and the size of the RAM. Also, OpenMP based PMk-means is more scalable compared to the POSIX based PMk-means.

5. REFERENCES

[1] "A UNIX interface for shared memory and memory mapped files under mach" , www.72.14.235.104

[2] ECE 222 System Programming Concepts lecturer notes on system calls, www.parl.clemson.edu

[3] fread/ifstream, read/mmap performance results www.lastmind.net.

[4] Gray, A. and Moore, A. (July.-2004), "Data structures for fast statistics", Tutorial presented at the International Conference on Machine Learning, Banff, Alberta, Canada.

[5] I. S. Dhillon and D. S. Modha, "*A Data Clustering Algorithm on Distributed Memory Multiprocessors In Large-Scale Parallel Data Mining*" , Lecture Notes in Artificial Intelligence, vol. 1759, Springer-Verlag, pp 245-260, March 2000.

[6] Jiawei Han and Micheline Kamber (2006),"Data Mining concepts and Techniques", 2nd edition Morgan Kaufmann Publishers, San Francisco.

[7] Manasi N. Joshi, "Parallel *K* - Means Algorithm on Distributed Memory Multiprocessors", Project Report**,** Computer Science Department University of Minnesota, Twin Cities, Spring 2003.

[8] N.B.Venkateswarlu, M.B.Al-Daoud and S.A Raberts (1995), "Fast k-means Clustering *Algorithms*", University of Leads School of Computer Studies Research Report Series Report 95.18.

[9] OpenMP Architecture , "OpenMP C and C++ Application Program Interface", http://www.openmp.org/

[10] Optimized performance analysis of Apache-1.0.5 server, www.isi.edu

[11] "Parallel Programming In OpenMP" by Rohit Chandra , Ramesh Menon , Leo Dagum , David Kohr , Dror Maydan

and Jeff McDonald, Academic press, A Harcourt Science and Technology Company, USA,2001.

[12] Rabert cattral and Franz Oppacher Carleton University, Department of Computer Science Intelligent systems research unit, Canada, http://archive.ics.uci.edu/ml/datasets/Poker+Hand

[13] S .N. Tirumala Rao, E. V. Prasad, N. B. Venkateswarlu and B. G. Reddy, "*Significant performance evaluation of memory mapped files with clustering algorithms*", IADIS International conference on applied computing, Portugal pp .455-460, 2008.

[14] Tuba Islam,"An unsupervised approach for Automatic Language dentification", Master Thesis, Bogaziqi University, Istambul, Turkey,2003.

[15] Yen-Yu chen, Dingquing Gasu, Torsten Suel (2002), "*I/O Efficient Techniques for computing page rank*", Department of computer and information science, Polytechnique university, Brooklyn, Technical Report -CIS-2002-03.

ANNEXURE

Table 1: Quad-Core Real time observations of k-means algorithm by varying RAM with random data for dimensionality 10 and clusters 10 under Linux environment.

Records in Million	<-Quad-Core –OpenMP--PFk-means-> Real-Time				←Quad-Core-OpenMP -PMk-means -> Real-Time				%of mmap() Benefit	
	RAM 0.5GB	RAM 1.5GB	RAM 2.5GB	RAM 3GB	RAM 0.5GB	RAM 1.5GB	RAM 2.5GB	RAM 3GB	RAM 0.5GB	RAM 3GB
11	253.763	4.162	4.065	4.066	67.208	3.462	3.400	3.387	73.52	16.70
13	425.436	4.863	4.861	4.841	91.207	4.058	4.069	4.015	78.56	17.06
15	533.361	6.281	5.696	5.582	112.951	4.642	4.645	4.620	78.82	17.23
19	812.045	13.901	7.094	7.065	157.193	5.948	5.883	5.775	80.64	18.26
20	862.789	17.658	7.441	7.580	164.806	6.229	6.321	6.386	80.90	15.75

Table 2: Quad-Core System time observations of k-means algorithm by varying RAM with random data for 10 million records and dimensions 10 under Linux environment.

No Of Clusters	<-Quad-Core –-OpenMP---PFk-means---> System-Time				←Quad-Core--OpenMP --PMk-means -> System-Time				%of mmap() Benefit	
	RAM 0.5GB	RAM 1.5GB	RAM 2.5GB	RAM 3GB	RAM 0.5GB	RAM 1.5GB	RAM 2.5GB	RAM 3GB	RAM 0.5GB	RAM 3GB
2	4.206	1.303	1.211	1.192	0.899	0.089	0.227	0.214	78.63	82.05
4	4.213	1.297	1.216	1.179	0.959	0.095	0.200	0.212	77.24	82.02
6	5.611	1.325	1.209	1.188	1.993	0.152	0.181	0.223	64.48	81.23
8	5.078	1.297	1.230	1.201	1.841	0.183	0.209	0.204	63.75	83.01
10	5.756	1.306	1.209	1.183	1.985	0.203	0.217	0.188	65.51	84.11

Table 3: k-means Real-Time Speed Up Comparison of single-core, Dual-Core and Quad-Core at 15 Million records and dimensions 10 at 1GB RAM under Linux environment.

No Of Clusters	←-Single-Core-OpenMP---→ Real-Time 1GB RAM			<-----Dual-Core----------OpenMP------> Real-Time 1GB RAM				<-----Quad-Core-------OpenMP---------> Real-Time 1GB RAM			
	SFk-means(S1)	SMk-means	Speed-UP	PFk-means(P1)	PMk-means(P2)	S1/P1	S1/P2	PFk-means(P3)	PMk-means(P4)	S1/P3	S1/P4
2	24.104	16.259	1.483	15.420	1.568	1.563	15.372	11.156	1.225	2.161	19.677
4	28.954	14.789	1.958	16.777	2.271	1.726	12.749	11.574	1.687	2.502	17.163
6	54.688	39.933	1.369	20.967	6.674	2.608	8.194	13.381	3.215	4.087	17.010
8	52.203	37.728	1.384	20.521	6.085	2.544	8.579	12.818	2.838	4.073	18.394
10	84.258	60.487	1.393	24.054	10.048	3.503	8.386	14.464	4.673	5.825	18.031

Table 4: k-means Real-Time Speed up Comparison of single-core, Dual-Core and Quad-Core at 15Million records and dimensions 10 at 3GB RAM under Linux environment

No Of Clusters	←-Single-Core—OpenMP---→ Real-Time 3GB RAM			<-----Dual-Core----------OpenMP------> Real-Time 3GB RAM				<-----Quad-Core-------OpenMP---------> Real-Time 3GB RAM			
	SFk-means(S1)	SMk-means	Speed-UP	PFk-means(P1)	PMk-means(P2)	S1/P1	S1/P2	PFk-means(P3)	PMk-means(P4)	S1/P3	S1/P4
2	10.617	9.279	1.144	2.863	1.614	3.708	6.578	2.304	0.963	4.608	11.02
4	16.485	14.790	1.115	3.480	2.337	4.737	7.054	2.387	1.441	6.906	11.44
6	41.808	39.993	1.045	7.513	6.477	5.565	6.455	4.343	3.322	9.627	12.59
8	39.258	37.861	1.037	7.248	5.990	5.416	6.554	3.795	2.804	10.345	14.00
10	62.946	60.769	1.036	11.246	10.073	5.597	6.249	5.526	4.879	11.391	12.90

Table 5: Quad-Core Real time observations of k-means algorithm by varying RAM with random data for 10 million records and dimensions 10 under Linux environment

No Of Clusters	<--Quad-Core –POSIX--PFk-means--> Real-Time				←Quad-Core--POSIX -PMk-means -> Real-Time				%of mmap() Benefit	
	RAM 0.5GB	RAM 1.5GB	RAM 2.5GB	RAM 3GB	RAM 0.5GB	RAM 1.5GB	RAM 2.5GB	RAM 3GB	RAM 0.5GB	RAM 3GB
2	36.033	1.187	1.225	1.298	25.031	0.992	0.851	0.896	30.53	30.97
4	30.237	1.381	1.512	1.496	17.196	1.045	1.104	0.999	43.13	33.22
6	67.185	2.614	2.592	2.614	43.920	2.536	2.318	2.316	34.63	11.40
8	46.982	2.507	2.610	2.589	32.668	2.150	2.226	2.187	30.47	15.53
10	75.982	4.013	3.941	3.933	64.240	3.774	3.497	3.519	15.45	10.53

Table 6: Quad-Core System time observations of k-means algorithm by varying RAM with random data for 10 million records and dimensions 10 under Linux environment.

No Of Clusters	<--Quad-Core --POSIX--PFk-means-----> System-Time				←Quad-Core---POSIX --PMk-means -> System-Time				%of mmap() Benefit	
	RAM 0.5GB	RAM 1.5GB	RAM 2.5GB	RAM 3GB	RAM 0.5GB	RAM 1.5GB	RAM 2.5GB	RAM 3GB	RAM 0.5GB	RAM 3GB
2	2.555	0.615	0.614	0.634	1.465	0.234	0.233	0.249	42.66	60.73
4	2.364	0.609	0.616	0.628	0.744	0.257	0.205	0.264	65.91	60.99
6	3.670	0.618	0.620	0.626	1.721	0.226	0.198	0.244	58.66	67.57
8	3.101	0.625	0.625	0.642	1.351	0.224	0.225	0.226	43.53	65.89
10	3.862	0.605	0.646	0.637	2.909	0.236	0.213	0.215	52.49	68.45

Table 7: Quad-Core User time observations of k-means algorithm by varying RAM with random data for 10 million records and dimensions 10 under Linux environment

No Of Clusters	←--Quad-Core –-POSIX---PFk-means--> User-Time				←--Quad-Core---POSIX --PMk-means -> User-Time				%of mmap() Benefit	
	RAM 0.5GB	RAM 1.5GB	RAM 2.5GB	RAM 3GB	RAM 0.5GB	RAM 1.5GB	RAM 2.5GB	RAM 3GB	RAM 0.5GB	RAM 3GB
2	2.116	1.937	2.157	2.360	1.993	3.257	2.507	2.931	5.81	-24.19
4	2.997	2.668	3.126	2.984	2.792	3.357	3.696	3.121	6.84	-4.59
6	8.570	7.035	7.270	7.285	7.770	8.840	8.026	8.254	9.33	-13.30
8	7.781	6.739	6.881	6.694	7.849	7.674	7.488	7.620	-0.87	-13.83
10	13.193	11.935	11.812	11.717	12.897	12.678	12.284	12.195	2.24	-4.08

Table 8: Dual-Core User time observations of k-means algorithm by varying RAM with random data for 15 million records and clusters 10 under Linux environment.

No Of Dimensions	<--- Dual--Core –POSIX---PFk-means---> User-Time				←--Dual-Core –POSIX---PMk-means---> User-Time				%of mmap() Benefit	
	RAM 0.5GB	RAM 1.5GB	RAM 2.5GB	RAM 3GB	RAM 0.5GB	RAM 1.5GB	RAM 2.5GB	RAM 3GB	RAM 0.5GB	RAM 3GB
2	20.348	20.342	20.049	20.018	20.566	20.709	20.529	20.699	-1.07	-3.40
4	22.017	22.156	22.028	21.733	22.231	22.680	22.609	22.441	-0.97	-3.26
6	35.681	30.528	30.529	30.272	33.427	31.213	32.013	30.887	6.32	-2.03
8	29.696	24.854	24.805	24.806	27.544	25.293	25.403	25.264	7.25	-1.85
10	26.393	22.027	22.930	22.734	25.608	22.434	22.455	22.491	2.97	1.07

Table 9: k-means Real-Time Speed Up Comparison of single-core, Dual-Core and Quad-Core at 15 Million records and dimensions 10 at 1GB RAM under Linux environment.

No Of Clusters	←-Single-Core--POSIX------→ Real-Time 1GB RAM			<-----Dual-Core----------POSIX------> Real-Time 1GB RAM				<-----Quad-Core-------POSIX---------> Real-Time 1GB RAM			
	SFk-means(S1)	SMk-means	Speed-UP	PFk-means(P1)	PMk-means(P2)	S1/P1	S1/P2	PFk-means(P3)	PMk-means(P4)	S1/P3	S1/P4
2	24.104	16.259	1.48	15.694	12.129	1.54	1.99	9.052	7.412	2.66	3.25
4	28.954	14.789	1.96	16.699	3.504	1.73	8.26	10.070	1.608	2.88	18.01
6	54.688	39.933	1.37	21.209	8.397	2.58	6.51	12.061	3.525	4.53	15.51
8	52.203	37.728	1.38	20.357	7.531	2.56	6.93	11.816	3.349	4.42	15.59

10	84.258	60.487	1.39	24.687	11.639	3.41	7.24	13.964	9.387	6.03	8.98

Table 10: k-means Real-Time Speed Up Comparison of single-core, Dual-Core and Quad-Core at 15 Million records and dimensions 10 at 3GB RAM under Linux environment.

	←-Single-Core—POSIX----→ Real-Time 3GB RAM			<-----Dual-Core----------POSIX------> Real-Time 3GB RAM				<-----Quad-Core-------POSIX---------> Real-Time 3GB RAM			
No Of Clusters	**SFk-means(S1)**	**SMk-means**	**Speed-UP**	**PFk-means(P1)**	**PMk-means(P2)**	**S1/P1**	**S1/P2**	**PFk-means(P3)**	**PMk-means(P4)**	**S1/P3**	**S1/P4**
2	10.617	9.279	1.14	3.196	2.536	3.32	4.19	2.032	1.409	5.22	7.54
4	16.485	14.790	1.11	4.190	3.466	3.93	4.76	2.093	1.678	7.88	9.82
6	41.808	39.993	1.05	8.841	8.394	4.73	4.98	3.911	3.338	10.69	12.52
8	39.258	37.861	1.04	7.865	7.595	4.99	5.17	3.760	3.268	10.44	12.01
10	62.946	60.769	1.04	12.303	11.646	5.12	5.40	5.869	5.626	10.73	11.19