Implementation and Performance of Asynchronous and Synchronous Data Classification Algorithms

Adrian C. Moga and Michel Dubois

CENG Technical Report 95-07

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4475

March 1995

Implementation and Performance of Asynchronous and Synchronous Data Classification Algorithms [1]

Adrian C. Moga and **Michel Dubois**

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-2562
E-mail: {moga,dubois}@paris.usc.edu

1. Introduction

Data classification using cluster analysis techniques has a very broad domain of application, ranging from image processing to sociology [1], [2], [3], [4]. It is a main instrument in the process of revealing structure within complex and large bodies of data. Given a sample of data units, each unit is described by scores on a set of predefined features and the goal of the analysis is to group them into larger agglomerations or clusters. Units within a cluster have a high degree of association defined by the specific application, and the clusters are reasonably distinguishable from each other based on an application-specific metric.

The processing needs of data classification are quite demanding as the sets of data can reach very large sizes. Parallel algorithms are therefore a natural solution based on the association cluster-processing element. In addition to demands imposed by large inputs, some applications such as image processing may require real-time processing, thus reinforcing the need for parallel processing.

There are two broad categories of clustering methods: hierarchical and non-hierarchical methods and in this paper we focus on the latter. The objective of non-hierarchical clustering methods is to classify every data unit in one of k clusters, where k is fixed a priori or adjusted at

1. This research has been funded by the National Science Foundation Grant No. CCR-9222734

run-time. The procedure starts by choosing an initial partition and successively refines it into better partitions according to application-dependent criteria until it reaches an optimum. A parallel has been drawn between non-hierarchical methods and steepest descent algorithms used for unconstrained optimization. Therefore, they are iterative methods and can be implemented synchronously or asynchronously [5].

The implementations of non-hierarchical clustering algorithms for shared-memory multiprocessors manipulate very large sets of data. As the clusters become more and more well-defined, the association cluster-processor (plus cache and/or local memory) becomes stronger (i.e. with a higher hit ratio) because data units start to fall in place and stop migrating between clusters. This is true assuming static allocation. Self-scheduling implementations may incurring high start-up penalties at every iteration but offer better load balancing. Asynchronous implementations may affect this behavior for better or for worse, but they eliminate the need for barrier synchronizations between iterations and could also affect the convergence rate. As a side note, COMA machines could provide a good environments for clustering techniques, because of the natural association attraction memory-cluster.

This presents presents performance measurement taken on four implementations of synchronous and asynchronous versions of non-hierarchical clustering algorithms for shared memory multiprocessors using the ANL set of macros for parallel programming [7]. The performance of these implementations is analyzed in terms of ideal execution times and parallel speedups using the CacheMire simulator [8]. Additinally, the miss rates are evaluated for cache-coherent architectures with a directory-based protocol similar to Censier and Feautrier's [6]. The specific application for these implementations is a simplified image analysis.

Currently, there are no reports of asynchronous implementations of non-hierarchical clustering algorithms, nor of evaluations of the cache performance of any kind of implementation.

2. Implementations of Parallel Non-hierarchical Clustering Algorithms

Non-hierarchical clustering algorithms involve two phases, which are common to all iterative implementations:

- setting the initial configuration (establishing x_0)
- nearest centroid sorting (the successive approximation)

The first phase involves choosing a set of k cluster nuclei and has many known implementations. Possibilities include random assignment, synthetic assignment, picking the first k data samples or picking k data samples in the sequence of m with a stride of m/k. More elaborate techniques can be considered if necessary.

Several published algorithms exists for the second phase in two cases: fixed and variable number of clusters [1]. We have restricted ourselves to algorithms for fixed number of clusters. They are all iterative algorithms and include:

- Forgy's method and a variant due to Jancey.
- MacQueen's k-means method and a variant (only the latter has been considered for this study since it is more general).

Both algorithms iterate through two basic steps: allocating a data unit to the *closest* cluster and recomputing new centers for the clusters. Convergence is achieved when there are no more changes in the composition of the clusters. The first step involves computing distances (typically, Euclidean) from each data sample to every cluster center and comparing them to find the minimum. The second step replaces the center of every cluster by the mean of its current members.

Forgy's method is essentially of Jacobi type, as it first reallocates all the data and then recomputes the centers of the clusters. The convergent k-means method recomputes centers after every reallocation for the two clusters involved in the transfer and can be considered a Gauss-Seidel type of iteration.

The parallel implementations of the algorithms assign clusters to processing units such that multiple re-classifications of the clusters are handled simultaneously. In the synchronous version, a run through all the clusters must be completed (and new centers computed) before another

refinement step can be started. This involves a barrier at the beginning of each iteration. Synchronous iterations coupled with static scheduling can lead to dangerous load imbalances, as some of the processors can be assigned clusters which are considerably larger than others, thus lengthening the per-iteration execution time. Self-scheduling loops offer an alternative, but may affect the locality of accesses. Asynchronous implementations remove the barriers between iterations, but may exhibit different convergence properties.

The termination condition can be easily detected in synchronous implementations by using a shared counter of clusters that have changed in the course of an iteration. Termination is detected when this value is null at the end of an iteration. For asynchronous implementations a different solution must be devised since the time to check for termination is not clearly defined. Interestingly, a parallel can be drawn between termination and the absence of invalidations in the cache protocols (on the blocks containing cluster composition information).

In the following, we describe in detail four implementations of parallel algorithms for cluster analysis: synchronous Forgy's method (SF), synchronous Forgy's method with Jancey's variant (SFJ), synchronous convergent k-means method (SK) and an asynchronous version based on the convergent k-means method (A).

2.1 Synchronous Forgy's Method

Forgy's method clearly separates the two phases of the iterative process by first reallocating all the data units to the existing clusters and then recomputing the cluster centers. Because the cluster centers are fixed throughout the iteration, the outcome of any processing pass is independent of the particular sequence in which the data samples are input into the program or are processed in the course of one iteration.

The first phase is parallelized by assigning different processors to different data to compute the distances to clusters and, possibly, perform a reallocation. As the number of data items is several orders of magnitude larger than the number of available processors, starving is out of question. The scheduling is static because the jobs are highly homogeneous, leading to almost perfect load balancing. The overhead involved in self-scheduling would not be justified. During

data reallocation, which involves changes in the membership structure of the clusters, one lock protect each cluster and clusters are updated in mutual exclusion. The membership information consists of doubly-linked lists, one for each cluster, and back-pointers from a data item to the cluster it belongs to. The first phase ends with a barrier, separating it from the second phase.

The second phase updates the centers of the clusters in parallel by assigning clusters to different processors. It is a highly parallel phase, with no sharing involved. If the number of clusters is smaller than the number of processors, some processors will idle. One could further breakdown this phase into parallel updates of each coordinate of the cluster centers, which creates work for more processors without giving up any of the previous benefits. However, this phase is short enough, by comparison to the first phase, and therefore does not require special handling. The jobs are not as homogeneous anymore, as they depend on the number of nodes in a cluster. However, static scheduling is employed here, as well. This phase ends with another barrier, marking the beginning of the termination condition checking.

Termination checking is performed by reading a shared flag. The flag is set in the second phase anytime a cluster changes its center. The setting involves no locking. The termination checking ends with another barrier. If the flag was found to be set, all the processors proceed to another iteration. In addition, processor 0 resets the flag.

2.2 Synchronous Forgy's Method with Jancey's Variant

Jancey's variant to Forgy's methods consists in performing a different repositioning of the cluster center in the second phase. It works on the basis of an analogy with the steepest gradient search for unconstrained optimization. In the original Forgy's method, the new center is computed as the mean of all the members in the cluster. The displacement of the center from the old position to the new one can be viewed as a move toward the final position. However, since data was allocated to the cluster on the basis of distances with respect to the old center, the process of approaching the final position was possibly (hopefully) retarded. Hence, the new center is anticipated to be farther in the same direction. Conventionally, it is placed at twice the distance of the mean from the old center, in the same direction. This is equivalent to a mirroring of the old center through the mean. Scaling could be performed additionally, but no guidelines are available. Jancey's variant does not

involve the same data structures, synchronization patterns and allocation strategy as the implementation of Forgy's method described previously. The termination detection is different however. The criteria of unchanged cluster centers cannot be used anymore since centers can still move even when convergence is reached due to the mirroring effect after the cluster are well defined. Therefore, we need to track changes in the composition of each cluster and set a global flag after every such change to indicate that the iterative process must be continued.

2.3 Synchronous Convergent *k*-means Method

The convergent *k*-means method originates in MacQueen's method, which is not an iterative method. It performs readjustments of cluster centers after every reallocation, thus mixing the two phases of Forgy's method together. It has the benefit of making better allocations in each pass and, possibly, converges faster. The fact that cluster centers are updated more frequently is balanced to some extent by the fact that each recomputation of a center is simpler since only one item is added/removed at a time. Unlike Forgy's method, the outcome of one iteration is dependent on the order in which the data units are processed. In a parallel environment, the behavior is non-deterministic even in synchronous implementations.

In the parallel synchronous implementation of the convergent *k*-means method the data units are statically distributed to the available processors, which compute the distances to the centroids and if needed reallocate the units to clusters and re-computate the centers. All accesses to cluster-related information are protected by locks. By contrast to Forgy's method, lock-protected accesses are required in the computation of each distance from a node to a cluster center in order to avoid using partially updated center coordinates. This was not required in Forgy's method because the cluster centers remained fixed throughout the reallocation phase. The reallocation phase ends with a barrier, followed by termination checking.

For termination detection the same technique as in Forgy's method is employed. A shared flag is set every time a node is reallocated. Flag setting operations are required each time a data unit is reallocated, which is much more frequent than in Forgy's method. A center displacement is caused by at least one data reallocation. The flag is reset by processor 0 after all the processors have read it.

2.4 Asynchronous *k*-means Method

Our asynchronous implementation is based on the *k*-means method since Forgy's method was more restrictive. In this asynchronous implementation, the barrier marking the end of an iteration in the synchronous *k*-means method is removed. Thus, processors start new iterations independently of the others. Every processor keeps a local termination flag and only checks a global termination flag when the local flag is reset, which indicates that no changes occurred during the last pass. Checking the global flag consists of checking into a barrier. The barrier is raised by any processor that has performed changes during its most recent pass since these changes might affect the configuration for the processors that have halted. When leaving the barrier, different directions are taken, depending on whether a global termination flag is set. The flag can only be set by a processors that reaches the barrier and is the last active processor. Typically, a processor does not wait at the barrier more than twice until global termination. The implementation has also removed locked access to center coordinates for computing distances, thus allowing "approximative" distances to be used.

Of course, other allocation strategies can be applied for the asynchronous implementation. They will make the object of further study. Of particular interest is the allocation of clusters as the work units for processors. The non-interference radius of a cluster could then be defined as the double of the largest distances from the center. Unless another cluster places its center inside the non-interference area, the cluster can remain inactive once it reaches stability. This method could be of interest only if the number of clusters is large enough.

3. Simulation Experiments

The four implementations, described in the previous section, have been tested on an input consisting of 883 samples, having two coordinates each: *(x,y)* with values between 0 and 100. A graphic image of the test set is given in Figure 1. The samples can be thought of as pixels in a black-and-white image. Had the image been gray, a third coordinate would have been necessary.

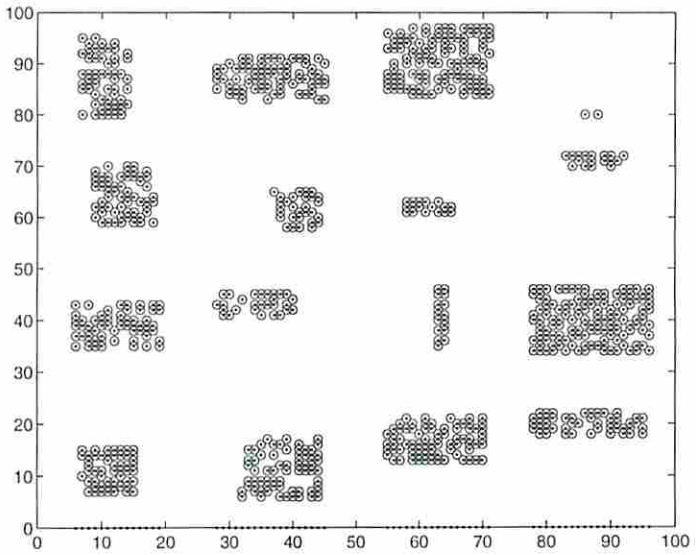

Figure 1. Graphic description of the test set of samples.

The number of clusters is set to 15. In all the runs, we have used the same strategy for seed selection for the centers: the first 15 samples. Since the samples were randomly mixed in the input, this was equivalent to using either random selection or selection with a stride. Synthetic seed points (equally spaced over the domain) were tried as well, but they were too biased and generated convergence in 1 or 2 steps.

The number of iterations for convergence is relatively small for all the synchronous implementations: 10-15 at most. The k-means method requires even fewer iterations. The clusters that are generated are mostly imposed by the set of seed points for the centroids. Many of the clusters visible in Figure 1 are recognized by the procedure. However, a limitation of the methods employing fixed number of clusters is that distinct clusters are lumped together in areas were insufficient seed points are initially allocated. In the end, the computed clusters have their centers somewhere in between two or even three of the clusters apparent in Figure 1. Conversely, large clusters can be split in areas with many seed points. Another observation regards Jancey's variant. It is unreliable and not very efficient in producing fast convergence. A problem is that clusters are sometimes emptied when the center is pushed too much in the "expected" direction. Probably more sophisticated methods of setting the seed points or a reduced "overshoot" when updating the centers would lead to a more predictable behavior.

4. Execution Time and Parallel Speedups

The ideal execution times and speedups obtained with a perfect memory system where all memory access penalties are zero are shown in Figures 2 and 3, respectively, for up to 64 processors. The execution times were collected using the CacheMire-2 simulator [8].

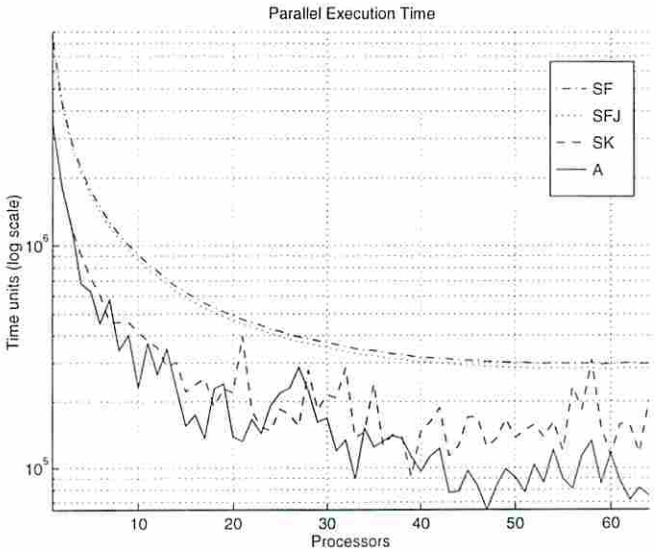

Figure 2. Ideal execution time for the four implementations.

The graphs show the clear superiority of the *k*-means method among the synchronous implementations and the fact that the asynchronous version can consistently provide additional improvement over the synchronous version.

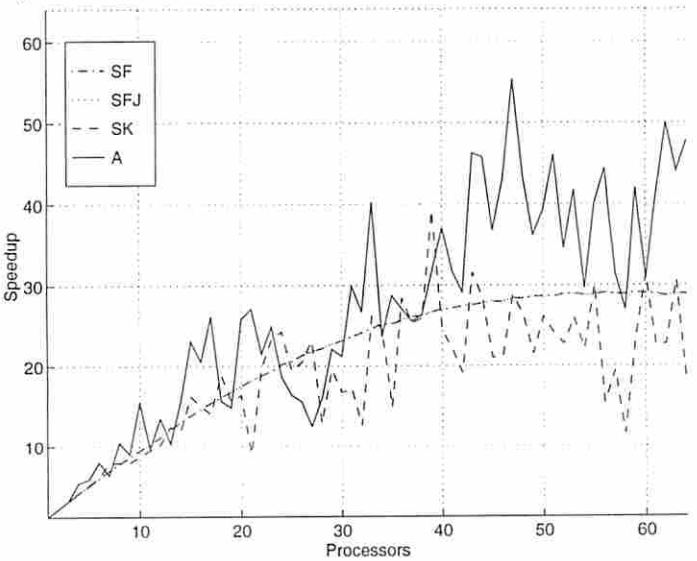

Figure 3. Ideal speedup for the four implementations.

The speedup graph is practically linear until about 20 processors. Forgy's method eventually saturates. Jancey's variant brings only minor improvements. The synchronous and asynchronous k-means methods have better scaling properties, although they seem to exhibit inconsistent behavior. This is the result of non-deterministic evolution of the clusters, as explained before. This may cause longer execution time with more processors. However, the trend is clearly upward although, occasionally, the speedup decreases. Also, it is interesting to observe that the asynchronous version can achieve superlinear speedup in some cases.

5. Cache Performance Evaluation Results

In this part of our work, we have obtained traces of executions of the four implementations using the CacheMire-2 simulator. The traces have been used for the performance analysis of a directory-based cache protocol with various cache configurations. Details of the coherence protocol can be found in [6].

We first describe the memory access pattern of each of the four implementations. In all implementations, the number of shared reads is considerably larger than the number of shared writes: 38 or 39% of all accesses are reads, but the fraction of writes varies from 0.3-0.5% for Forgy's method to 0.7-1.2% for the k-means methods. Also, the synchronization activity is quite low: from 0.17-0.3% of all accesses for the asynchronous method, up to about 0.56% for the other methods. The reason is that the changes in the cluster configurations involving write operations are preceded by the computation of distances between data units and the center of each of the k clusters, which exclusively involves read operations.

In the following, we report on the result of the simulations for each implementation, showing the miss rates and their breakdown for two cache sizes: 1 KB and infinite and for block sizes between 8 and 256 bytes. We also show the bus traffic. Comments on the results are included for each implementation.

Figure 4 shows the miss rates for the asynchronous implementation. The 1KB cache is clearly too small and the working set does not fit in it; consequently, replacement misses dominate. Cold misses are not significant in all cases. For infinite caches, the best block size is 8 bytes,

which is not surprising. The two coordinates of a data sample, which are read only, are 4 bytes each and therefore both exactly fit in an 8-byte block. Because of the type of data structures employed various writable data are included in each block, which leads to increased number of invalidations at larger block sizes (false sharing). This is true for the cluster data structures, as well. A coordinate is stored as a float (4 bytes) and two coordinates fit in a block. Better separation of different types of data could produce miss rate improvements. For example, the data samples coordinates are read-only and should be stored away from the information about the membership of the data units to clusters.

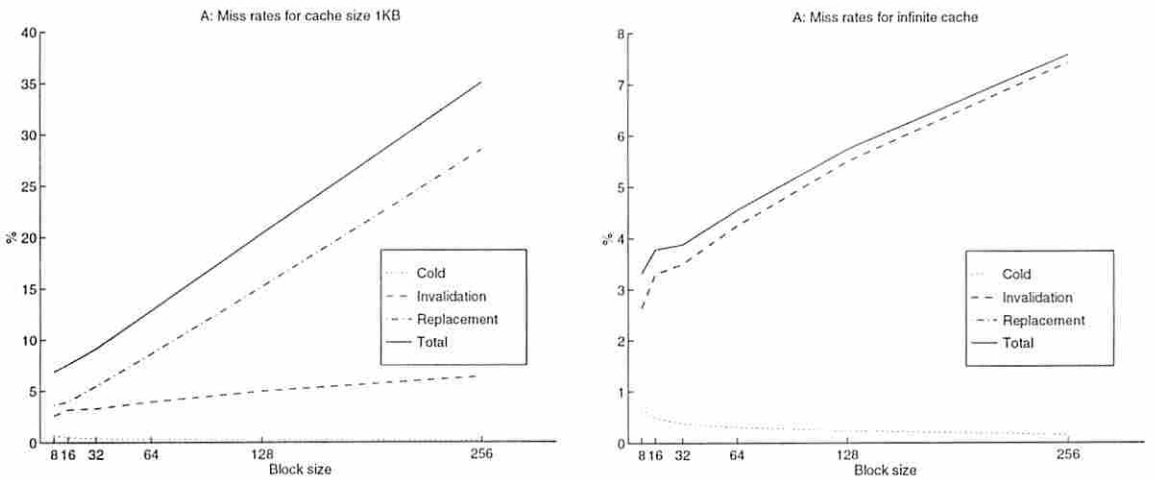

Figure 4: Miss rates for the asynchronous implementation.

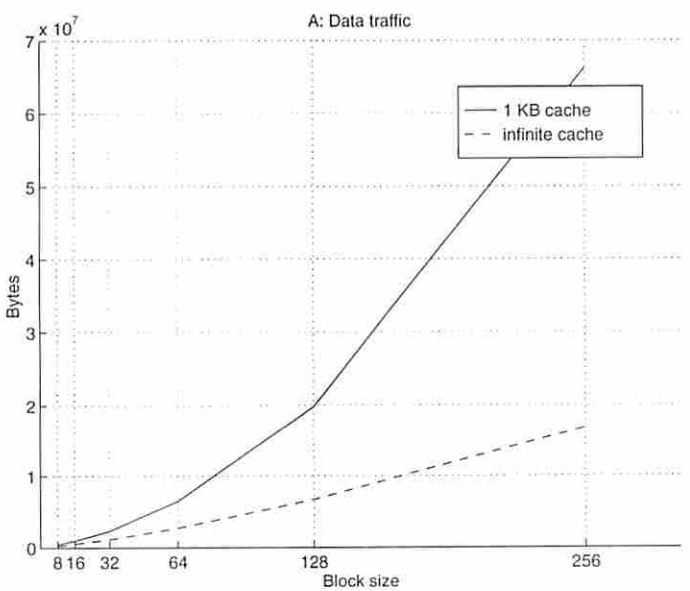

Figure 5: Data traffic for the asynchronous implementation.

Figure 5 shows the data traffic for the asynchronous implementation. Since both the miss rate and the block size grow, so does the traffic.

Figure 6 shows the miss rates for the synchronous k-means method. The trends are the same as in the asynchronous implementation. As the amount of locking is increased, more invalidations occur, leading to even higher miss rates. Figure 7 shows that the data traffic increases with the block size, like before, and is a little higher, due to the increased miss rate.

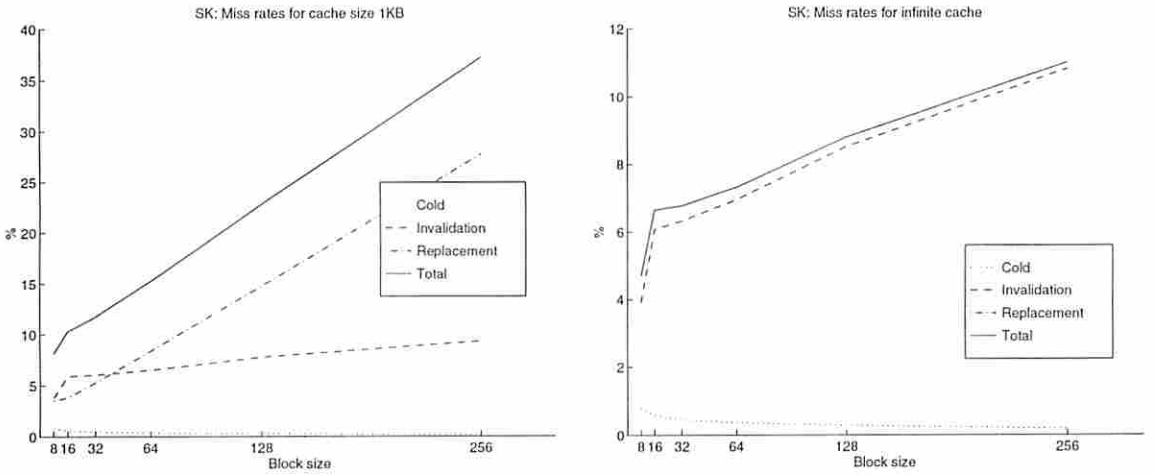

Figure 6: Miss rates for the SK implementation.

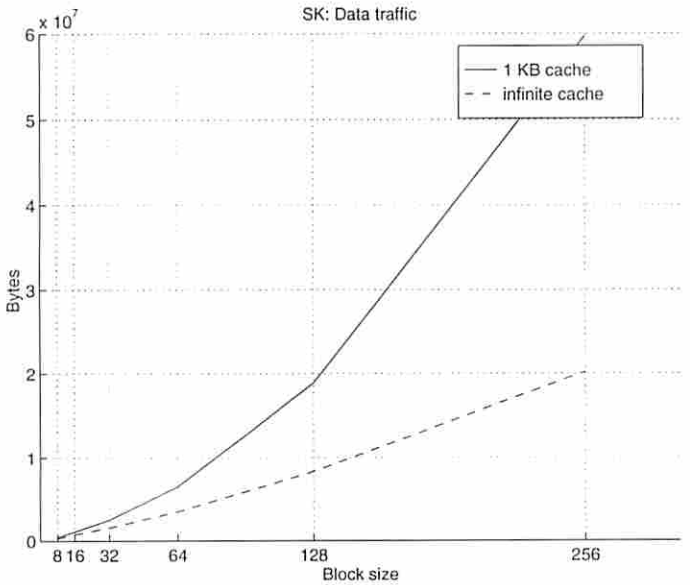

Figure 7: Data traffic for SK implementation.

Figure 8 shows the miss rates for the synchronous Forgy method. Miss rates are a little smaller, because the amount of writing is reduced, as noted before. The cluster center coordinates are mostly-read in Forgy's method, but often-written in the k-means method.

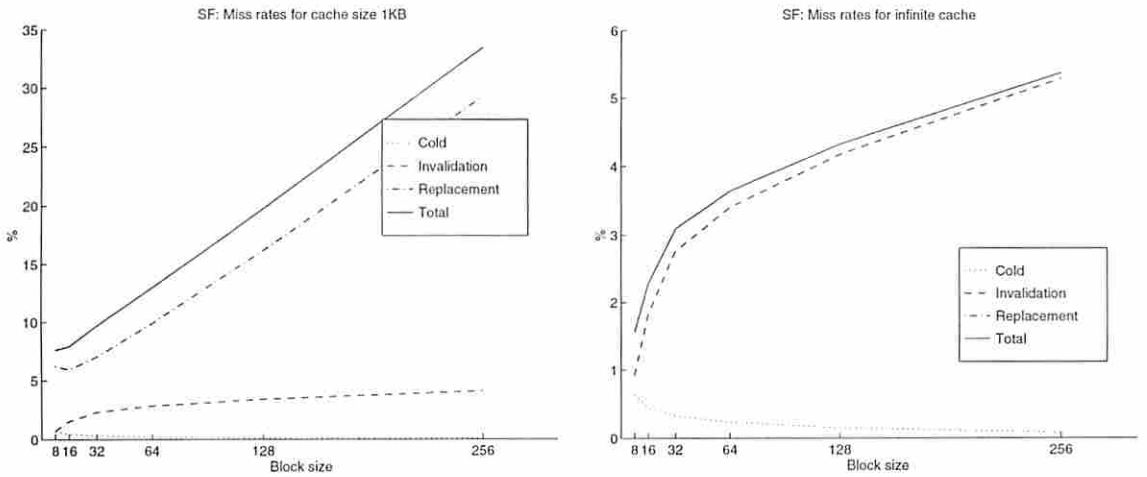

Figure 8: Miss rates for the SF implementation.

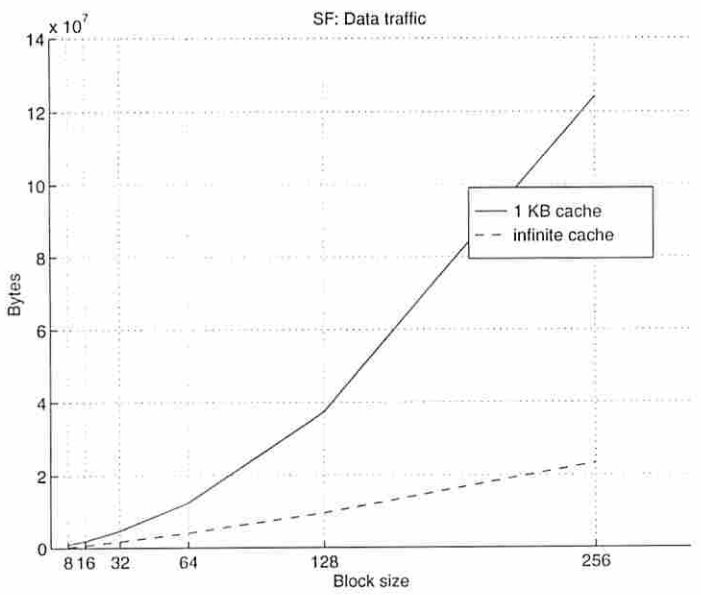

Figure 9: Data traffic for SF implementation.

The data traffic in Figure 9 shows the same increasing trend with larger block size.

Figure 10 shows the miss rates for the synchronous Forgy's method implementation using Jancey's variant. The miss rates are higher due to more write operations leading to invalidations. This is because the cluster centers continue oscillating even after the cluster membership is settled. Also, more frequent updates of the global termination flag are performed.

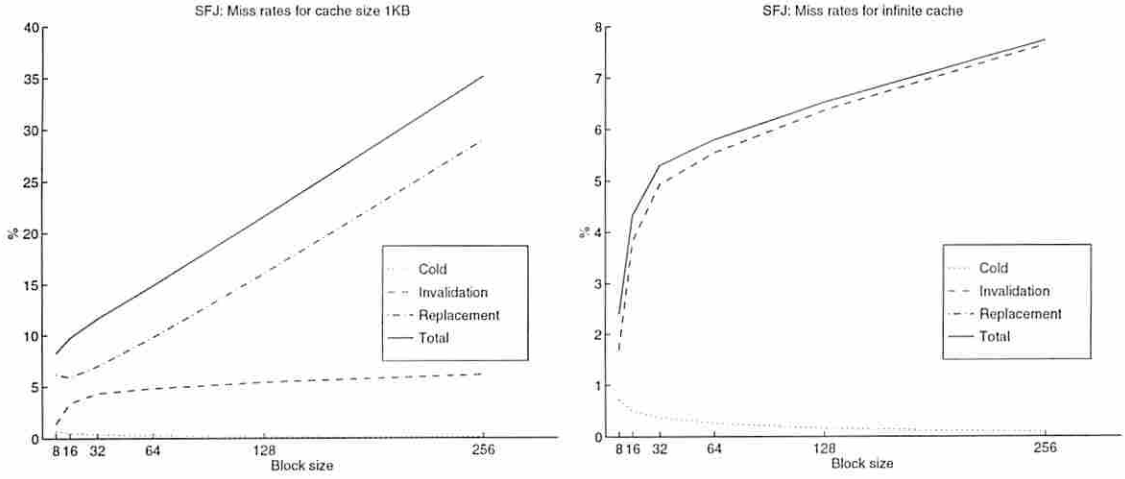

Figure 10: Miss rates for the SFJ implementation.

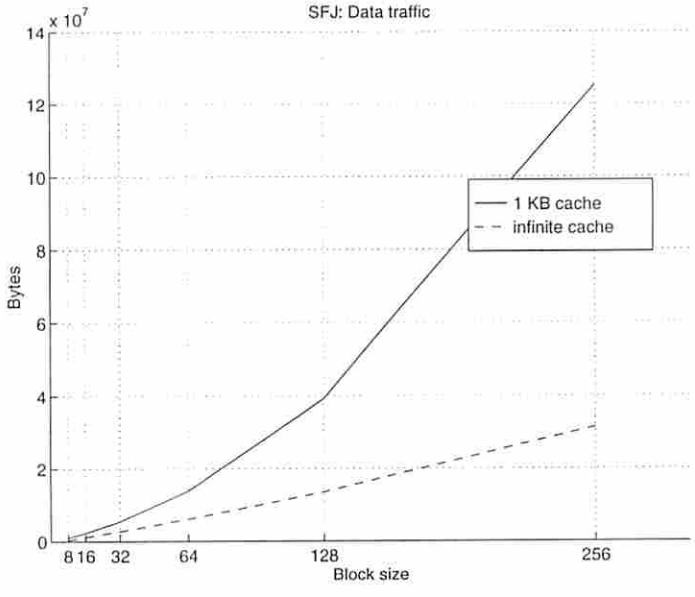

Figure 11: Data traffic for SFJ implementation.

The data traffic in Figure 11 is similar, only slightly higher, to the one in the SJ method

6. Conclusions

Cluster analysis algorithms are very suitable for parallel processing. They have good speedups and scale well with the number of processors, especially the *k*-means methods. On shared-memory machines, a good combination of data storage separation, based on the type of sharing, and small block sizes can lead to small miss rates. The asynchronous implementation was an overall winner.

References

[1] Michael R. Anderberg. *Cluster Analysis for Applications.* Academic Press, 1973.

[2] Brian Everitt. *Cluster Analysis.* SSRC, London, 1977.

[3] Benjamin S. Duran and Patrick L. Odell. *Cluster Analysis - A Survey.* Springer-Verlag, 1974.

[4] John A. Hartigan. *Clustering Algorithms.* John Wiley & Sons, 1975.

[5] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and distributed computation.* Prentice Hall, 1989.

[6] Lucien Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. IEEE Transactions on Computers, Dec. 1978, pp. 1112-1118.

[7] James Boyle et al. *Portable programs for parallel processors.* New York: Holt, Rinehart, and Winston, 1987.

[8] M. Brorsson et al. The CacheMire Test Bench - A Flexible and Effective Approach for Simulation of Microprocessors. In Proceedings of the 26th Annual Simulation Symposium, March 1993, pp.41-49.