Implementation Issues in the Design of I/O Intensive Data Mining Applications on Clusters of Workstations

R. Baraglia[1], D. Laforenza[1], Salvatore Orlando[2],
P. Palmerini[1] and Raffaele Perego[1]

[1] Istituto CNUCE, Consiglio Nazionale delle Ricerche (CNR), Pisa, Italy
[2] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy

Abstract This paper investigates *scalable* implementations of out-of-core I/O-intensive Data Mining algorithms on affordable parallel architectures, such as clusters of workstations. In order to validate our approach, the K-means algorithm, a well known *DM Clustering* algorithm, was used as a test case.

1 Introduction

Data Mining (DM) applications exploit huge amounts of data, stored in files or databases. Such data need to be accessed to discover patterns and correlations useful for various purposes, above all for guiding strategic decision making in the business domain. Many DM applications are strongly I/O intensive since they need to read and process the input dataset several times [1,6,7]. Several techniques have been proposed in order to improve the performance of DM applications. Many of them are based on parallel processing [5]. In general, their main goals are to reduce the computation time and/or reduce the time spent on accessing out-of-memory data.

Since the early 1990s there has been an increasing trend to move away from expensive and specialized proprietary parallel supercomputers towards clusters of workstations (COWs) [3]. Historically, COWs have been used primarily for science and engineering applications, but their low cost, scalability, and generality provide a wide array of opportunities for new domains of application [13]. DM is certainly one of these domains, since DM algorithms generally exhibit large amounts of data parallelism. However, to efficiently exploit COWs, parallel implementations should be adaptive with respect to the specific features of the machine (e.g. they must take into account the memory hierarchies and caching policies adopted by modern hardware/software architectures).

Specific *Out-of-Core* (OoC) techniques (also known as External Memory techniques) [3,14] can be exploited to approach DM problems that require huge amounts of memory. OoC techniques are useful for all applications that do not completely fit into the physical memory. Their main goal is to reduce memory hierarchy overheads by bypassing the OS virtual memory system and explicitly

managing I/O. Direct control over data movements between main memory and secondary storage is achieved by splitting the dataset into several small blocks. These blocks are then loaded into data structures which will certainly fit into physical memory. They are processed and, if necessary, written back to disks. The knowledge of the patterns used by the algorithm to access the data can be exploited in an effective way to reduce I/O overheads by overlapping them with useful computations. The access pattern exploited by the DM algorithm discussed in this paper is simple, since read-only datasets are accessed sequentially and iteratively. Note that the general purpose external memory mechanism provided by the operating system – in our case, the Unix `read()` system call – is specifically optimized for this kind of data access.

This paper investigates *scalable* implementations of I/O-intensive DM algorithms on affordable parallel architectures, such as clusters of PCs equipped with main memories of a limited size, which are not sufficiently big to store the whole dataset (or even a partition of it). The test case DM application used to validate our approach is based on the on-line K-means algorithm, a well known *DM Clustering* algorithm [8,10]. The testbed COW was composed of three SMPs, interconnected by a 100BaseT switched Ethernet, where each SMP was equipped with two Pentium II - 233 MHz processors, 128 MB of main memory, and a 4GB UW-SCSI disk. Their OS was Linux, kernel version 2.2.5-15. The paper is organized as follows. Section 2 discusses implementation issues related to the design of I/O-intensive DM applications. Section 3 deals with the K-means algorithm and its parallel implementation. Finally, Section 4 discusses the results of our experiments and draws some conclusions.

2 Implementation of I/O Intensive DM Applications

As mentioned above, we are interested in DM algorithms that access sequentially the same dataset several times. The repeated scanning of the whole dataset entails good spatial locality but scarce temporal locality. The latter can only be exploited if the whole dataset entirely fits into the physical memory. In general, however, this condition cannot be taken for granted because "real life" datasets are generally very large. Moreover, the physical memory is of limited size, and other running processes contend for its usage. The adoption of an OoC algorithm, which takes advantage of possible prefetching policies implemented by both software drivers and disk controllers [11], and which allows to exploit *multitasking* or *multithreading* strategies in order to overlap I/O latencies with useful computations, is thus mandatory.

The best policy might thus appear to be to adopt OoC algorithms only if a dataset does not fit into the physical memory. When the memory is large enough, an in-core approach might seem more efficient, since all the dataset is read once from disk, and is repeatedly accessed without further I/O operations. Clearly such an in-core strategy might fail when other processes use the main memory, thus causing swapping on the disk. We believe that "smart" OoC approaches are always preferable to their in-core counterparts, even when datasets are small

with respect to memory size. This assertion is due to the existence of a *buffer cache* for block devices in modern OSs, such as Linux [2]. The available physical memory left unused by the kernel and processes is dynamically enrolled in the buffer cache on demand. When the requirement for primary memory increases, for example because new processes enter the system, the memory allocated to buffers is reduced. We conducted experiments to compare in-core and out-of-core versions of a simple test program that repeatedly scans a dataset which fits into physical memory. We observed that the two versions of the program have similar performances. In fact, if we consider the OoC version of this simple program, at the end of the first scan the buffer cache contains the blocks of the whole dataset. The following scans of the dataset will not actually access the disk at all, since they find all the blocks to be read in the main memory, i.e. in the buffer cache. In other words, due to the mechanisms provided by the OS, the actual behavior of the OoC program becomes in-core.
We also observed another advantage of the OoC program over the in-core solution. During the first scan of the dataset, the OoC program takes advantage of OS prefetching. In fact, during the processing of a block the OS prefetches the next one, thus hiding some I/O time. On the contrary, I/O time of in-core programs cannot be overlapped with useful computations because the whole dataset has to be read before starting the computation.

In summary, the OoC approach not only works well for small datasets, but it also scales-up when the problem size exceeds the physical memory size, i.e., in those cases when in-core algorithms fail due to memory swapping. Moreover, to improve scalability for large datasets, we can also exploit multitasking techniques in conjunction with OoC techniques to hide I/O time. To exploit multitasking, non-overlapping partitions of the whole dataset must be assigned to distinct tasks. The same technique can also be used to parallelize the application, by mapping these tasks onto distinct machines. This kind of data-parallel paradigm is usually very effective for implementing DM algorithms, since computation is generally uniform, data exchange between tasks is limited, and generally involves a global synchronization at the end of each scan of the whole dataset. This synchronization is used to check termination conditions and to restore a consistent global state. Consistency restoration is needed since the tasks start each iteration on the basis of a consistent state, generating new local states that only reflect their partial view of the whole dataset.
Finally, parallel DM algorithms implemented on COWs also have to deal with load imbalance. In fact, workload imbalance may derive either from different capacities of the machines involved or from unexpected arrivals of external jobs. Since the programming paradigm adopted is data parallel, a possible solution to this problem is to dynamically change partition sizes.

3 A Test Case DM Algorithm and its Implementation

There is a variety of applications, ranging from marketing to biology, astrophysics, and so on [8], that need to identify subsets of records (clusters) present-

ing characteristics of *homogeneity*. In this paper we used a well known clustering algorithm, the **K-means** algorithm [10] as a case study representative of a class of I/O intensive DM algorithms. We deal with the on-line formulation of K-means, which can be considered as a competitive learning formulation of the classical K-means algorithm. K-means considers records in a dataset to be represented as *data-points* in a high dimensional space. Clusters are identified by using the concept of proximity among data-points in this space. The K-means algorithm is known to have some limitations regarding the dependence on the initial conditions and the shape and size of the clusters found [9,10]. Moreover, it is necessary to define *a priori* the number K of clusters that we expect to find, even though it is also possible to start with a small number of clusters (and associated centers), and increase this number when specific conditions are observed. The three main steps of the on-line K-means sequential algorithm are: (1) start with a given number of centers randomly chosen; (2) scan all the data-points of the dataset, and for each point p find the center closest to p, assign p to the cluster associated with this center, and move the center toward p; (3) repeat step 2 until the assignment of data-points to the various clusters remains unchanged. Note that the repetition of step 2 ensures that centers gradually get attracted into the middle of the clusters. In our tests we used synthetic datasets and we fixed a priori K.

Parallel Implementation. We implemented the OoC version of the algorithm mentioned above, where data-points are repeatedly scanned by sequentially reading small blocks of 4 KBytes from the disk. The program was implemented using MPI according to an SPMD paradigm. A non overlapping partition of the input file, univocally identified by a pair of boundaries, is processed by each task of the SPMD program. The number of tasks involved in the execution may be greater than the number of physical processors, thus exploiting multitasking. This parallel formulation of our test case is similar to those described in [12,4], and requires a new consistent global state to be established once each scan of the whole dataset is completed. Our global state corresponds with the new positions reached by the K centers. These positions are determined by summing the vectors corresponding with the centers' movements which were separately computed by the various tasks involved. In our implementation, the new center positions are computed by a single task, the root one, and are broadcast to the others. The root task also checks the termination condition.

The load balancing strategy adopted is simple but effective. It is based on past knowledge of the bandwidths of all concurrent tasks (i.e. number of points computed in a unit of time). If a load imbalance is detected, the size of the partitions is increased for "faster" tasks and decreased for "slower" ones. This requires input datasets to be replicated on all the disks of our testbed. If complete replication is too expensive or not possible, file partitions with overlapping boundaries can be exploited as well. Let NP be the total number of data-points, and $\{p_1, \ldots, p_n\}$ the n tasks of the SPMD program. At the first iteration $np_i^1 = NP/n$ data-points are assigned to each p_i. During iteration j each p_i measures the elapsed time T_i^j spent on elaborating its own block of np_i^j points,

so that $t_i^j = T_i^j / np_i^j$ is the time taken by each p_i to elaborate a single point, and $b_i^j = 1/t_i^j = np_i^j / T_i^j$ is its bandwidth. In order to balance the workload, the numbers np_i^{j+1} of data-points which each p_i has to process in the next iteration are then computed on the basis of the various b_i^j ($np_i^{j+1} = \rho_i^j \cdot NP$, where $\rho_i^j = b_i^j / \sum_{i=1}^{n} b_i^j$). Finally, values np_i^{j+1} are easily translated into partition boundaries, i.e. a pair of offsets within the replicated or partially replicated input file.

4 Experimental Results and Conclusions

Several experiments were conducted on the testbed with our parallel implementation of K-means based on MPI. Data-parallelism, OoC techniques, multitasking, and load balancing strategies were exploited. Note that the successful adoption of multitasking mainly depends on (1) the number of disks with respect to the number of processors available on each machine, and (2) the computation granularity (i.e., the time spent on processing each data block) with respect to the I/O bandwidth. In our experiments on synthetic datasets, we tuned this computational granularity by changing the number K of clusters to look for. Another important characteristics of our approach is the size of the partitions assigned to the tasks mapped on a single SMP machine. If the sum of these sizes is less than the size of the physical main memory, we guess that the behavior of the OoC application will be similar to its in-core counterpart, due to a large enough buffer cache. Otherwise, sequential accesses carried out by a task to its dataset partition will entail disk accesses, so that the only possibility of hiding these I/O times is to exploit, besides OS prefetching, some form of moderated multitasking.

Figure 1 shows the effects of the presence of the buffer cache. On a single SMP we ran our test case algorithm with a small dataset (64 MB) and small computational granularity (K=3). Bars show the time spent by the tasks in computing (`t_comp`), in doing I/O and being idle in some OS queue (`t_io + t_idle`), and in communication and synchronization (`t_comm`). The two bars on the left hand side represent the first and the second iterations of a sequential implementation of the test case. The four bars on the right hand side regard the parallel implementation (2 tasks mapped on the same SMP). Note that in both cases the `t_io` and `t_idle` are high during the first iteration, since the buffer cache is not able to fulfill the read requests (cache misses). On the other hand, these times almost disappear from the the second iteration bars, since the accessed blocks are found in the buffer cache (cache hits).

Figure 2 shows the effects of multitasking on a single SMP when the disk has to be accessed. Although a small dataset was used for these experiments, the bars only refer to the first iteration, during which we certainly need to access the disk. Now recall that our testbed machines are equipped with a single disk each. This represents a strong constraint on the I/O bandwidth of our platform. This is particularly evident when several I/O-bound tasks, running in parallel on an SMP, try to access this single disk. In this regard, we found that our test case has different behaviors depending on the computational granularity.

For a fine granularity (K=8), the computation is completely I/O-bound. In this condition it is better to allocate a single task to each SMP (see Figure 2.(a)). When we allocated more then one task, the performance worsened because of the limited I/O bandwidth and I/O conflicts. For a coarser granularity (K= 32), the performance improved when two tasks were used (see Figure 2.(b)). In the case of higher degrees of parallelism the performance decreases. This is due to the overloading of the single disk, and to noises introduced by multitasking into the OS prefetching policy.

Figure 3 shows some speedup curves. The plots refer to 20 iterations with K=16. We used at most two tasks per SMP. Note the super-linear speedup achieved when 2 or 3 processors were used. These processors belong to distinct SMPs, so that this super-linear speedup is due to the exploitation of multiple disks and to the effects of the buffer cache. In fact, when moderately large datasets were used (64 MB or 128 MB) the data partitions associated with the tasks mapped on each SMP fit into the buffer caches. Overheads due to communications, occurring at the end of each iteration, are very small and do not affect speedup.
In the case of a larger dataset (384 MB), whose size is greater than the whole main memory available, when the number of tasks remains under three, linear speedups were obtained. For larger degrees of parallelism, the speedup decreases. This is still due to the limited I/O bandwidth on each SMP.

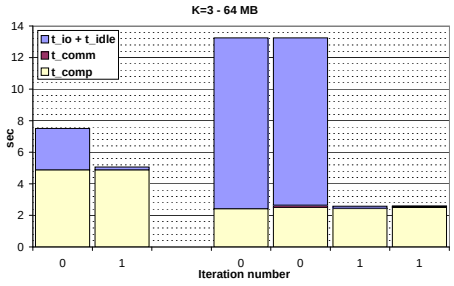

Figure1. Execution times of two iterations of the test case on a single SMP.

Figure 4 shows the effectiveness of the load balancing strategy adopted. Both plots refer to experiments conducted using all the six processors of our testbed with the 64MB dataset and K = 32. The plot in the left hand side of the figure shows the number of blocks dynamically assigned to each task by our load balancing algorithm as a function of the iteration index. During time interval $[t1, t3]$ ($[t2, t4]$) we executed a CPU-intensive process on the SMP `A` (`M`) running tasks `A0` and `A1` (`M1` and `M1`). As it can be seen, the load balancing algorithm quickly detects the variation in the capacities of the machines, and correspondingly adjusts the size of the partitions by narrowing partitions assigned to slower

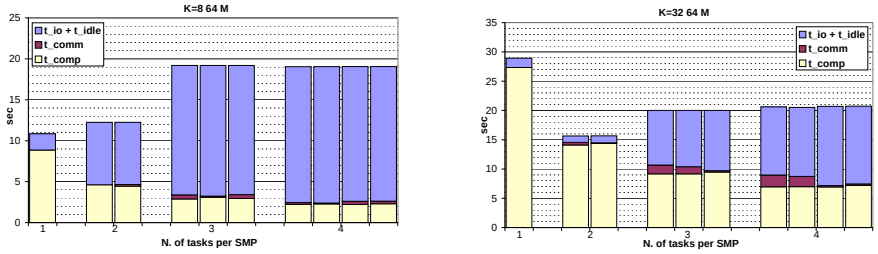

Figure2. Execution times of the first iteration on a single SMP by varying the number of tasks exploited and the computational granularities: (a) K=8 and (b) K=32.

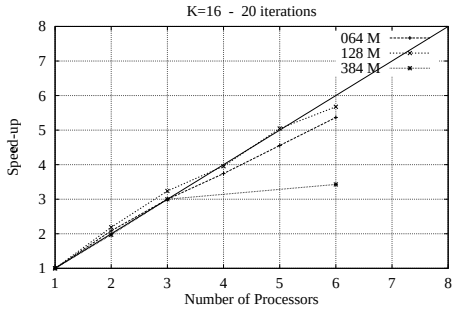

Figure3. Speedup curves for different dataset sizes (20 iterations, $K = 16$).

machines and enlarging the others. The plot in the right hand side compares the execution times obtained exploiting or not our load balancing strategy as a function of the external load present in one of the machines. We can see that in the absence of external load the overhead introduced by the load balancing strategy is negligible. As the external load increases, the benefits of exploiting the load balancing strategy increase as well.

In conclusion, this work has investigated the issues related to the implementation of a test case application, chosen as a representative of a large class of DM I/O-intensive applications, on an inexpensive COW. Effective strategies for managing I/O requests and for overlapping their latencies with useful computations have been devised and implemented. Issues related to data parallelism exploitation, OoC techniques, multitasking, and load balancing strategies have been discussed. To validate our approach we conducted several experiments and discussed the encouraging results achieved. Future work regards the evaluation of the possible advantages of exploiting lightweight threads for intra-SMP parallelism and multitasking. Moreover, other I/O intensive DM algorithms have to be considered in order to define a framework of techniques/functionalities useful for efficiently solving general DM applications on COWs, which, unlike homo-

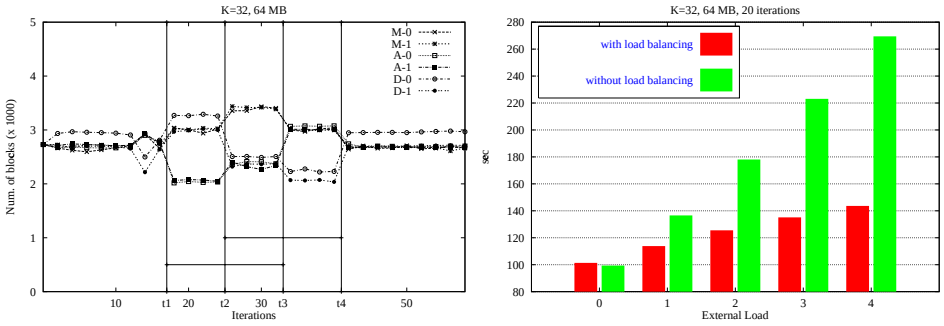

Figure4. Effectiveness of the load balancing strategy.

geneous MPPs, impose additional issues that must be addressed using adaptive strategies.

References

1. Jain A.K. and Dubes R.C. *Algorithms for Clustering Data.* Prentice Hall, 1988.
2. M. Beck et al. *Linux Kernel Internals, 2nd ed.* Addison-Wesley, 1998.
3. Rajkumar Buyya, editor. *High Performance Cluster Computing.* Prentice Hall PTR, 1999.
4. I. S. Dhillon and D. S. Modha. A data clustering algorithm on distributed memory machines. In *ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 1999.
5. A. A. Freitas and S. H. Lavington. *Mining Very Large Databases with Parallel Processing.* Kluwer Academin Publishers, 1998.
6. V. Ganti, J. Gehrke, and R. Ramakrishnan. Mining Very Large Databases. *IEEE Computer*, 32(8):38–45, 1999.
7. E. Han, G. Karypis, and V. Kumar. Scalable Parallel Data Mining for Association Rules. *IEEE Transactions on Knowledge and Data Engineering.* To appear.
8. J.A. Hartigan. *Clustering Algorithms.* Wiley & Sons, 1975.
9. G. Karypis, E. Han, and V. Kumar. Chameleon: Hierarchical Clustering Using Dynamic Modeling. *IEEE Computer*, 32:68–75, 1999.
10. Mac Queen, J.B. Some Methods for Classification and Analysis of Multivariate Observation. *5^{th} Berkeley Symp. on Mathematical Statistics and Probability*, pages 281–297. Univ. of California Press, 1967.
11. Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.
12. K. Stoffel and A. Belkoniene. Parallel k-means clustering for large datasets. *EuroPar'99 Parallel Processing*, Lecture Notes in Computer Science, No. 1685. Springer-Verlag, 1999.
13. Sterling T.L., Salmon J., Becker D.J., and Savarese D.F. *How to Build a Beowulf. A guide to the Implementation and Application of PC Clusters.* The MIT Press, 1999.
14. J. S. Vitter. External Memory Algorithms and Data Structures. In *External Memory Algorithms (DIMACS Series on Discrete Mathematics and Theoretical Computer Science).* American Mathematical Society, 1999.