A Parallel Data Management Layer for Data Mining

M. Coppola[a], P. Pesciullesi[b], L. Presti[c], R. Ravazzolo[b], M. Vanneschi[b]

[a]Ist. di Scienza e Tecnologie dell'Informazione, CNR - Via Moruzzi 1, 56124 Pisa, Italy

[b]University of Pisa, Dip. di Informatica - Via Buonarroti 2, 56127 Pisa, Italy

[c]IMT Alti Studi Lucca, Via San Micheletto 3, 55100 Lucca, Italy

We propose the design of a data management abstraction level to implement a full set of parallel KDD applications with minimal performance overhead and greater scalability than conventional DBMS, providing a high-level parallel API to be exploited by parallel and out-of-core data mining algorithms. We describe an existing prototype and report examples and first test results with mining algorithms.

1. Introduction

The design of the data management level for Knowledge Discovery in Databases (KDD) involves several difficult trade-offs in choosing the right API. There are contrasting needs in the data management layer implementation, w.r.t. expressive power, flexibility, raw performance (e.g. I/O performance and computational overhead) which condition the overall performance of the KDD process.

Because of its iterative search nature, KDD highly benefits from the use of standard DBMS tools, exploiting their flexibility in the steps of data extraction and preparation. However, the size of data in real-life situations (and the number of attributes) often rules out algorithms with high accuracy just because of their complexity in terms of in-core and out-of-core operations [12]. To minimize this effect, data management support in the Data Mining (DM) step must achieve high efficiency and performance, leading to different specializations of DBMSs used.

Different solutions have been used in the practice, ranging from flat-file access, the development of special-purpose API to conventional DBMS [8], to RAM-based DB and OLAP approaches [9].

When developing parallel KDD systems we are confronted with even more complex issues, both as we have to solve larger and harder problem instances, and because of the need to efficiently exploit parallel I/O, and memory hierarchies made up of several stacked sequential and parallel architectural layers. These issues are not addressed by conventional DBMS systems, which fail to scale up to massively parallel architectures.

We propose the design of an intermediate abstraction level, the Parallel Data Repository (PDR), that provides enough flexibility to implement a full set of KDD applications, exposes performance critical choices to the application programmer hiding the messy details, and can be implemented with high performance on parallel architectures. We want to avoid

- the high overhead that standard DBMSs impose in order to support relational views, atomic transactions and concurrent access at the record level,
- the drawbacks of sequential/parallel low-level approaches to programming, which are complex, error prone and seldom portable
- the limits in scalability that conventional parallel approaches incur, being based on shared-memory DBMS servers and/or database replication.

We define an abstract software architecture for this data management support, which aims at exploiting out-of-core techniques from within structured parallel programs, reducing the complexity on implementation of parallel data mining applications without sacrificing the performance. The intended use of such a software layer is to ease parallel processing of data in the mining and validation steps of the KDD process. We thus assume that clean input data can be exported from conventional DBMS or Data Mart to the high-performance management system, in order to execute parallel mining algorithms on it.

The design of the PDR has been partially implemented, and it has been tested with several algorithms, showing promising results.

In section 2 we discuss the approach we have taken in designing the system, and compare with previous work. Sec.3 gives details on the PDR structure, and Sec.4 about those parts that have already been implemented. Sec.5 show simple examples of Mining algorithms that can be efficiently expressed using the API of the PDR and presents preliminary benchmark results. Sec.6 outlines future work directions.

2. Approach and Related Work

When applying parallel and distributed computing techniques to KDD systems and algorithms, we need to decompose data access and computation workload in parallel. Our aim is to reach *architecture portability* of DM computational cores, to be able to extend and reuse them as KDD modules or as the basis of different DM algorithms, without sacrificing their performance and parallel scalability. We define an abstraction level for data management, in order to optimize communication performance and workload distribution and, at the same time, to grant sufficient expressive power to code the algorithms independently from the details of data access. The approach we pursue has four key features

1. efficient parallel modularity/decomposability of computations exploiting the interface,
2. block-oriented, efficient exploitation of memory hierarchies,
3. DM tailored data management implementation and data semantics,
4. low overhead with respect to raw I/O.

The first point is addressed by exploiting the ASSIST structured parallel programming environment [1,11] for writing DM algorithms [4]. The parallel coordination approach allows to clearly express the parallel behaviour of the application, while sequential code performing the work doesn't deal with the issues of concurrent access to data. By decoupling the local (to each block or partition of the data) and global parts of the computation into different modules, we can control the flow of data in the algorithm structure itself, thus also avoiding the need for access control in the data management layer. In our view this requires

- independent concurrent operation on partitions of a file (a feature that is not provided by plain POSIX, but is needed for parallel I/O [10]),
- support for user-defined synopsis data structures linked to data blocks; each process/module in the DM application should be able to efficiently build/update/fetch the sufficient statistics needed to dispatch a data block within the algorithm and/or to a different processing node.

The structured approach to parallelism is coupled with a block-oriented data management level. We exploit the common structure of many DM algorithms, which are mainly data intensive, and can be written to work as much as possible on large blocks of data, improving the I/O performance. The theory of out-of core (OOC) algorithms has already shown that explicit secondary memory access control is needed to achieve optimal results. State-of-the-art

libraries for OOC programming like TPIE [2] are based on the load/unload paradigm. Block selection is specified by the user algorithm and is implemented by a block-moving engine, performing all I/O and related optimizations.

We concentrated on the parallel aspect of OOC, thus we do not have yet developed an API and an engine for prefetching strategies [2]. Recent works exist on the combination of the OOC and parallel aspects [5] into a single framework that organizes both issues (i.e. block fetching and parallel load-balancing). W.r.t. that, since ASSIST programs are not restricted to the pipeline pattern and can be run over the Grid as well as on cluster platforms, we are currently not going to integrate the expressive tools for OOC and parallelism computation. We support out-of-core parallel operations on a memory hierarchy by providing a block-oriented interface to the processes of a parallel application, which can then exploit block-aware algorithms to maximize the amount of in-memory computation.

Assuming the data has been cleaned and consolidated into a single large table, efficient support of parallel and secondary-memory block-oriented operation for DM algorithm is much easier to achieve than in the general case of DBMS applications, as we can focus on the problem of handling large bi-dimensional matrices with fixed row schema.

We thus advocate an intermediate approach between using DBMS tools and flat files, on the one hand providing only basic operations, with low computational cost, on the data tables. On the other hand, we improve w.r.t. relational databases and to flat-files in the ability to express data types routinely used in mining algorithms. Support of those basic data types used in DM algorithms that have a compact and efficient machine encoding, like small integers, sets of labels and booleans, also addresses the requirement of low overhead I/O.

Our approach differs from the prevalent one of developing special purpose API to conventional or parallel DBMS (Microsoft OLE-Db is an example [8]). The scalability of such an approach is limited by that of (most often SMP-based) parallel DBMS. When aiming at high performance DM and on-line transaction processing, RAM-based approaches are also used (e.g. the GemStone OLAP solution based on a distributed cache approach [9]).

For the sake of performance, we assume that the in-memory data representation does not change when moving form main memory to other memory levels, in order to efficiently perform direct access to tables loaded from secondary memory.

3. Design Proposal

We propose a distributed SW architecture where sequential, parallel/concurrent clients can access the PDR. The PDR is structured as a multiple-layer memory hierarchy sketched in Fig.2, where the levels are the memory local to each process (M0), a shared memory level (M1) and the secondary memory level (M2).

Global memory is distinguished into primary and secondary. With the technology currently used in computational clusters, and a hardware or software implementation of the primary shared memory level, it is safe to assume that primary global memory (M1) is smaller than secondary one (level M2, which is the aggregation of the available discs) and at least two orders of magnitude faster w.r.t. access latency. With the same assumption, access to local discs, when they are present, and to non-local disk storage exhibits the same latency. The PDR however does not exploit the three-level memory hierarchy as such. Levels M0 and M2 are used to process/hold data, while memory level M1 is reserved to hold meta-data, i.e. information about blocks of data.

We show in Fig.1 the layout of data within the repository. We call each database managed

in the PDR a dataset. All records in a dataset have the same structure, defined in term of provided types (integers, floats, dates, boolean, raw data, record keys) and of user-defined nominal types (defined as sets of labels). Thus the dataset is a bi-dimensional table with heterogeneous columns.

Blocks are the smallest amount of data transfer and of parallel work decomposition. Their size is fixed at dataset creation: a larger block size typically increases I/O bandwidth and DM algorithm efficiency, and decreases the available parallel work on a dataset. The external memory paradigm is applied to manage the data, using levels M0 and M2.

The upper level of the PDR provides the API and implements all local data management functions. The user code interfaces to the data by means of C++ classes, that manage data buffers in main memory, allow to operate on meta-data and delegate the I/O to the lower implementation levels.

Dataset global meta-data, including definitions of user-defined types, are managed by the PDR and are kept linked with each dataset, with low I/O overhead and memory occupation, and don't need to live in shared memory, as they are fixed at dataset creation[1].

Each block of data has also a linked data space (Fig.1), where synopsis data structures defined by the program exploiting the PDR are kept. The purpose of this additional space is to speed up application execution: programs can quickly store and retrieve synthetic information about a data block, to choose which blocks to process next in sequential/parallel computations, and sufficient statistics, useful to DM algorithms to avoid loading the data at all. We show a few examples in Sect.5.

Synopsis data structures will have a separate API from that of the data (allowing to define and operate on them), and a different implementation, on the assumptions that sufficient statistics are much smaller than the dataset, can change according to the algorithms, have a dynamic structure, and need to be shared among different parts of the algorithms much more often than the large blocks of the dataset. Thus we assume that the sufficient statistics should be stored on a fast memory that also supports synchronizations, like the (virtually) shared memory level M1.

I/O of data blocks is implemented by a lower level of parallel data servers with minimal centralized support to coordinate them. A block transfer engine can be implemented on each separate computing node, cooperating with the I/O servers.

The PDR design is architecture-independent, but to avoid data conversion across the memory levels we assume an homogeneous architecture, with the same kind of CPU and O.S. to execute all processes, and a common runtime (C++) to access in-memory data.

4. Current Implementation

We carried on the first implementation of the Parallel Data Repository as part of a parallel KDD engine within the SAIB project [3].

Mining Algorithms in the SAIB system are structured parallel applications written with the ASSIST parallel programming environment. They are developed following a common set of interfaces and used as interchangeable basic components within the KDD system. The PDR is used as an external object (an active object interfaced to all application processes and possibly

[1]In our case PVFS handles only a few, very large files, and we do not support arbitrary changes in the dataset structure. Hence we don't incur in performance bottlenecks from the PVFS meta-data server, and PDR's own global meta-data are read-only.

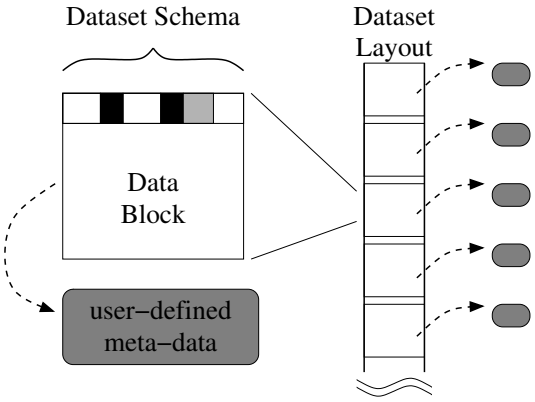

Figure 1. Layout of data within the PDR.

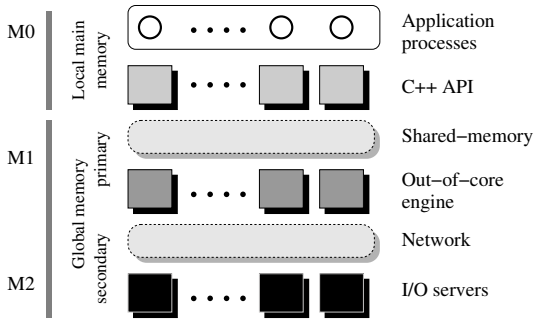

Figure 2. Implementation layers of the PDR.

being itself implemented as a parallel application). We exploit the parallel structure of the applications to ensure that parallel activities in the algorithm do actually operate on separate subsets of the data blocks.

The prototype PDR is structured as two layers, an *interface* level and an *implementation* one. The interface layer is implemented by a shared library, interfacing application processes to the data. This layer provides high-level functions to (1) manage datasets life cycle (2) operate on the logical/physical schemes of datasets and define dataset views, (3) load and unload blocks of data from multiple datasets to in-memory tables, (4) access to and management of in-memory tables, also performing integrity checks. The distinction between the logical schema and the physical one of a dataset, with the accompanying API to manipulate schemes, exists because programs can choose to operate only on part of the attributes of a dataset, and is also exploited to hide record field reordering performed by th PDR implementation to compact the data layout.

The interface layer performs also read/write operations of data blocks from shared/parallel devices. With respect to the abstract architecture of Fig.2, the implementation of the interface layer merges within the C++ API the essential functions of the OOC engine.

The PDR implementation layer performs out-of-core data block transfer form secondary memory, exploiting different I/O supports (POSIX I/O, parallel file system). We have employed a parallel file system (PVFS version 1 [7]) to implement the I/O layer shown in Fig.2. I/O servers actually map transparently to the *iod* PVFS daemons. This choice was convenient for the first implementation as it avoided the immediate need for developing a parallel OOC engine. We also support sequential file-systems, with NFS as a special case. It is thus possible to share a PDR dataset with sequential applications, loosing the parallel I/O advantages, but exploiting the same API. Combining multiple disc spaces into a single PDR space, effectively reimplementing the parallel file system functionalities, is something possible with respect to the abstract architecture we envisioned, but as a research direction it is yet unexplored.

The shared-memory level M1 has not yet been integrated in the PDR architecture. We emulate its functionalities exploiting in the algorithm the shared memory data-structures provided by the ASSIST run-time. Shared dynamic data structures can be defined and used within the program. The prototype has the full functionality of the abstract architecture, except for the ability to define synopsis data structures that are automatically persistent with the dataset, and the for need to explicitly manage these structures in the program code.

As a final remark, the proposed architecture can also be exploited on large clusters to pursue a RAM-based approach. In the general case, however, to hold medium size databases in (virtual) shared memory changes the parameters of the memory hierarchy exploited, and makes the proposed PDR architecture less useful.

5. Examples

Several algorithms from the data mining field can exploit the PDR interface, and most clustering algorithms fall in this category [6]. The BIRCH and CURE approaches are based on sufficient statistics and representatives, the STING grid-based approach and the density-based approach of OPTICS and DBSCAN rely on parallel and secondary memory techniques in order to optimize the running time. Many parallel and sequential optimizations for the classical k-means/medoid clustering algorithms are based on multiple levels of summary information associated with spatial data partitions.

Many low-level tasks (e.g. sorting, searching) can be expressed using the PDR primitives, as they allow to emulate those offered by OOC frameworks like FG [5].

Classification by tree induction is another notable source of examples: these algorithms are divide and conquer in nature, and recursively split the input dataset to build the classification tree. At each split, we need to compute a set of statistics over the data associated to the current tree node (histograms of the combinations of different attribute values within each data partition).

We have developed a prototype of the C4.5 algorithm that interfaces to the PDR. It reorders the dataset as needed to keep node-related partitions into separate sets of blocks. Blocks are then assigned to processing elements either to exploit data-parallel computation of the statistics on a single node of the classification tree, or to perform a task-parallel expansion of different nodes. Each block is represented in the algorithm by its linked meta-data, which include the array holding its histograms, and actual data transfer is performed only when needed to perform in-memory computation on that block.

Meta-data associated to data-block are used in an iterative clustering algorithm [3] to establish how many unclustered records belong to each block, in order to check before each linear scan computation if a data rearrangement operation can improve the overall execution time.

We have also developed algorithms for simpler data-management tasks like sorting, filtering and merging of datasets. As a general remark on them, implementation of algorithms based on the scan pattern points out the need of a block prefetch API. Fig. 4 reports test results of simple select/key-join operations on a dataset of size 2GB with a cluster of 8 Pentium-4 processing nodes linked by Gbit Ethernet.

6. Conclusions

We have introduced an architecture for a Parallel Data Repository to be coupled with high-level, structured parallel languages in the implementation of Data Mining algorithms. The PDR is based on the exploitation of the out-of-core programming paradigm and on a semantic tailored to mining algorithms, which are data-intensive and often employ a simple bi-dimensional view of the input data.

In the overall, we are working to verify that the PDR provides the right level of expressiveness to implement parallel DM algorithms with minimal coding effort, portability and and high performance. This is the same goal our research group more generally pursues w.r.t. parallel

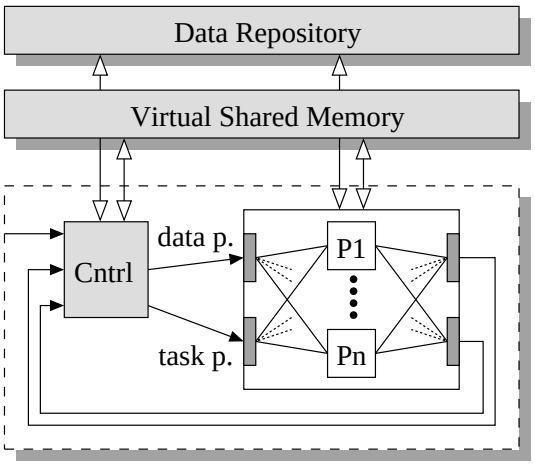

Figure 3. High-level structure of the C45 prototype.

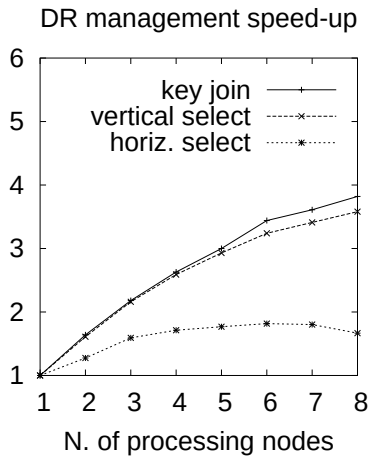

Figure 4. Performance benchmarks of the PDR implementation.

and Grid programming, and that frameworks like FG [5] aim at w.r.t. out-of-core, cluster-based computing.

The current status of development already shows some of the advantages of the architecture: we were able to develop a modular, parallel KDD engine based on the PDR, which runs on a cluster and interfaces to a parallel file system. Current block transfer engine is quite simple, though. While it already interfaces to parallel I/O resources, it does not yet offer the ability to do intensive data prefetch under algorithm control.

The full design of the PDR requires us to further develop the prototype, by providing an explicit link between each data block and the corresponding synopsis data structures, as well as an API to define and operate on them, in such a way that the associated data and their update code can be permanently attached to a dataset when needed.

Another development we are evaluating is to perform some basic data-reduction and data-parallel operation in the I/O server, to reduce network bandwidth requirements for I/O, and to ease the design of simplest parallel data management algorithms.

Acknowledgments

This work has been supported by the SAIB Project on High-performance infrastructures for financial applications, funded by MIUR and leaded by ATOS Origin, by the Italian MIUR FIRB Grid.it project, n. RBNE01KNFP, on High-performance Grid platforms and tools, and by the Italian MIUR Strategic Project L.449/97-2000 on High-performance distributed enabling platforms.

References

[1] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance Grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer, 2005. (to appear, draft available as TR-04-09, Dept. of Computer Science, University of Pisa, Italy, Feb. 2004).

[2] Lars Arge, Rakesh Barve, David Hutchinson, Octavian Procopinc, Laura Toma, Darren Erik Vengroff, and Rajiv Wickeremesinghe. *TPIE User Manual and Reference*, 0.9.01b edition, November 1999. Draft.

[3] Massimo Coppola, Paolo Pesciullesi, Roberto Ravazzolo, and Corrado Zoccolo. A Parallel Knowledge Discovery System for Customer Profiling. In Marco Danelutto, Domenico Laforenza, and Marco Vanneschi, editors, *Euro-Par'04 Parallel Processing*, number 3149 in Lecture Notes in Computer Science, pages 381–390, 2004. ISBN: 3-540-22924-8.

[4] Massimo Coppola and Marco Vanneschi. High-Performance Data Mining with Skeleton-based Structured Parallel Programming. *Parallel Computing, special issue on Parallel Data Intensive Computing*, 28(5):793–813, 2002. ISSN: 0167-8191.

[5] Thomas H. Cormen and Elena Riccio Davidson. Fg: A framework generator for hiding latency in parallel programs running on clusters. In *17th International Conference on Parallel and Distributed Computing Systems (PDCS 2004)*, pages 137–144, 2004.

[6] Jiawei Han and Micheline Kamber. *Data Mining, Concepts and Techniques*. Morgan Kaufmann, 2001.

[7] W. B. Ligon III and R. B. Ross. PVFS: Parallel virtual file system. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*, pages 391–430. MIT Press, 2001.

[8] OLE-DB technical documentation. `http://www.microsoft.com/data/oledb`.

[9] GemStone Systems. Gemfire Enterprise Technical WhitePaper. `http://www.gemstone.com/products/gemfire/GemFireTechnical_WP.pdf`, 2005.

[10] Rajeev Thakur, William Gropp, and Edwing Lusk. A Case for Using MPI's Derived Datatypes to Improve I/O Performance. In *Proc. of SC98: High Performance Networking and Computing*. IEEE, November 1998.

[11] Marco Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, 2002.

[12] Jeffrey Scott Vitter. External Memory Algorithms and Data Structures: Dealing with MASSIVE DATA. *ACM Computing Surveys*, 33(2):209–271, June 2001.