# Parallelisation of C4.5 as a Particular Divide and Conquer Computation

Primo Becuzzi, Massimo Coppola, Salvatore Ruggieri, and Marco Vanneschi

Dipartimento di Informatica, Universita' di Pisa

**Abstract** In this work we show the research track and the current results about the application of structured parallel programming tools to develop scalable data-mining applications. We discuss the exploitation of the divide and conquer nature of the well known C4.5 classification algorithm in spite of its in-core memory requirements. The opportunity of applying external memory techniques to manage the data is advocated. Current results of the experiments are reported.

The main research goal of our group in the past years has been the design and implementation of parallel programming environments to ease the engineering of High Performance applications. To provide a test bed for the current environment, as well as to investigate the theoretical problems involved in the parallelisation of largely used algorithms, we are developing parallel versions of Data-Mining (DM) applications.

Work is ongoing [2,3] to develop DM computational kernels that exhibit both code and performance portability over various parallel architectures, where performance means parallel speed-up and scalability to large databases.

Here we present the current results about the C4.5 algorithm, focusing on two main issues:

– the parallel scalability achievable using structured programming tools (a software engineering perspective)
– the evaluation of strategies to improve the support for tree-structured irregular computations, regarding C4.5 as an algorithm of the Divide and Conquer class.

Dealing with data which exceeds in size the local memory is the most common issue in High Performance Data Mining. We plan to enhance C4.5, which is an in-core classifier, to efficiently manage huge data sets.

We will introduce in the SkIE programming environment some external memory operations (like scan and sort), that exploit at first the sum of all the local memories. We will use those primitives to turn the C4.5 algorithm into a kind of out-of-core classifier. We will investigate if the same approach can be applied to the lower level of the memory hierarchy, disk-resident data.

The the paper is organized as follows. In section 1 we define the problem. Section 2 describes our programming environment. We compare our approach with previous ones in section 3, together with the explanation of the first results. Section 4 explains in detail the current results. The aim and the improvements expected from future work are the subject of the last section.

## 1 Problem statement

We are interested in the parallelisation of the C4.5 core, that is the building of the decision tree, as described in [5]. We'll leave out the evaluation and simplification phases, and we won't discuss the replication of cases with unspecified attributes, nor the windowing and trials techniques, even if these can be conveniently applied.

General divide-and-conquer (*D&C*) algorithms split (*divide*) large problems into smaller and smaller ones of the same form; the smallest ones are solved, then a recomposition (*conquer*) phase combines the solutions of larger and larger subproblems, up to the initial one. We give here a *D&C* description of the core of C4.5.

**Input:** a database $D$ of $n$ elements, each one a $k$-tuple of attributes $(a_1, \dots a_k)$. Each $a_i \in A_i$, where $A_i$ can be an interval (continuous attributes) or a finite set of values (categorical ones). One of the categorical attributes is distinguished as the class of each tuple.

**Output:** a decision tree $T$, i.e. a tree which predicts the class of tuples in terms of the values of the other attributes. Leaves of the tree are homogeneous subsets of the data. Each interior node (decision node) defines a split in the data determined by the test of the values of a single attribute, one branch is made for each outcome of the test.

**D1:** $\forall i \in \{1, k\}$ evaluate for attribute $i$ the information gain $g(D, i)$; let $j$ be the index that maximizes it.

**D2cat:** if $A_j$ is categorical, let $c = |A_j|$. Split $D$ into $c$ classes according to the value of $a_c$ of each tuple.

**D2cont:** if $A_j$ is continuous, let $c = 2$ and split $D$ in half such that $D_1 \leq D_2$. (*) Find *in all the data* the threshold value $t \in A_c$ that is the best approximation of the split point.

**D3:** build the root node for T and register the choices made so far.

**Rec:** each $D_1, \dots D_c$ that does satisfy the stopping criterion becomes a leaf. Process recursively all the others.

**Conquer:** add each returned leaf or tree $T_1, \dots T_c$ as the corresponding son of the root of $T$. Return $T$ as answer.

This is the *growth* phase of the tree, that is followed by a *prune* phase which is less hard to accomplish and we do not describe here.

Steps D1–D3 are the divide phase. The cost of step D1 is $O(n)$ operations for each categorical attribute, $O(n \log n)$ for continuous ones. Categorical attributes already used by an ancestor node are never checked again, since their information gain is zero. Both D2 steps, which are exclusive, require $O(n)$ operations, but finding the threshold (*) in step D2cont has an additional cost $O(N \log N)$ in the size $N$ of the whole initial database.

This behaviour, more deeply analyzed in [6], can impair load predictability and balancing of parallel implementations, and prevents them from being truly *D&C* computations. As already noted in [4], the threshold information is not needed until the pruning phase, so the threshold selection can be delayed and computed in an amortized way at the end of the classification phase.

We will assume now that the Conquer step in C4.5 has nearly no cost, and we will come back later to this point.

From an I/O operational point of view, all the steps require linear scans, or sorting and searching through the data in the current partition. Other parallel algorithms like [7] and [8] have been devised to deal with huge data partitions through special data structures and scheduling policies. We want to design a small set of general primitives that allow us to express the algorithm and can be implemented in a standard, user-friendly way in a high-level language. The research in the field of external memory algorithms [10] has produced theoretical analysis and scalable solutions for such a class of basic operations, that are suitable for application to the memory hierarchy of a parallel architecture.

We wanted to maintain the C4.5 sequential results as a reference point[1], and exploit the *D&C* aspect of the computation. Thus solutions that require to change the split criterion to binary, like in [7], or using a different splitting method [8], were regarded as not fully satisfying.

## 2 The Programming Environment

The SkIE environment [9] we are using and developing is a parallel coordination language based on the concept of *skeleton* . A skeleton is a basic form of parallelism that builds blocks of parallel code by composing simpler blocks (eventually sequential code) in an abstract, modular way. The aim of this approach is at reaching both source code and performance portability across different parallel architectures.

The skeleton run-time support deals with almost all the low level details of parallelism and concurrency (i.e. process mappings, communications, scheduling and load balancing). Application development is thus enhanced by software reuse, rapid prototyping, and a lesser need for performance debugging.

The SkIE user has to pick up a conceptual parallelisation that can be expressed using only the available skeletons. He is relieved of most of the low-level details of the parallelisation, but on the other hand he has little intervention on these aspects, in a trade-off between expressive power and efficiency.

The SkIE semantics is data-flow oriented, with an explicit vision of the streams of tasks. Among the used skeletons, the farm exploits task parallelism over a stream, providing automatic load distribution and balancing. The pipe is the functional composition, with pipeline parallelism exploited. A loop skeleton allows repeated processing of (part of) a stream by another parallel module. Other skeletons like map, reduce, deal with the basic forms of data-parallelism.

All communication set-up is transparently handled by the compiler at the interfaces between the modules, but this imposes some constraints on having fixed-size data structures as parameters. To overcome this limitation and allow

[1] Even if it is argued in the literature that a split criterion that requires sorting uses too much computation w.r. to the accuracy of the results.

more expressiveness, a virtual shared memory support is being integrated into the high-level interface. The abstraction is defined of dynamic, out-of-core shared data objects (SO), that are stored in the aggregate memory of the computer.

## 3 Related work and first experiments

With respect to the parallelisation, following [4] we can classify most of the previous approaches into *attribute parallel*, which assign each $A_i$ to a processor to execute step D1 in parallel, *data parallel* ones, which split the database among the processors and handle most operations collectively, and *task parallel* ones, which try to exploit the recursive definition to start separate computations at each branch.

The tree structure can be highly irregular, and this reflects in the computation. Since the classification workload cannot be foreseen for any given subtree, pure data parallel solutions in the literature exhibit no good parallel scalability, like the synchronous approach in [8]. As discussed in the same paper, even partitioned, more task-oriented solutions with static load-balancing cannot properly handle the irregular load, and a hybrid solutions is proposed.

We choose, instead, to explore task parallelism at first. A similar choice was made in [11], where data parallelism is combined with pipelining. Our experiments are a valuable research alternative, since we take advantage of the automatic load balancing and task pipelining in the SkIE skeletons.

Our prototypes exploits task parallelism, with each worker processor expanding a node of the C4.5 tree into a subtree, up to a certain depth $l$.

We refer to [1] for a detailed presentation of the first parallel version, which uses farm parallelism in a structure close to that in Fig.1.

Since the user has no control on the task distribution done by the farm skeleton, in our pure task approach there are only two parameters to tune: the depth $l$ of each expansion, and the selection order of waiting tasks.

Using a fixed, on-demand scheduling of tasks requires some property to hold for the given task stream. This first version of the program, which we call MP, was impaired by an excessive communication load and computation variance. The communications were all the same size, comparable to the database size.

The computation of thresholds in step D2cont (*), and the use of too high values for the $l$ parameter led to a highly variable worker load, which resulted in poor load balancing. Only minor improvements were obtained by using more complex or adaptive strategies to adjust the value of $l$ at run-time.

To improve upon the MP solution, we have started a new research path by introducing in the environment the abstraction of shared objects (SO), to be used to remove data replication and unwanted centralization points. This strategy has lead to (1) use the SO to improve communications, (2) switch to a full *D&C* computation, delaying threshold calculation. Next steps are (3) distribute the database among the workers, providing remote access methods, (4) turn the decision tree into a SO itself, thus removing the centralization point.

It is correctly pointed out in [8] that task parallelism alone for classification is not scalable because of large nodes. A further step will be to introduce data-parallel collective operations on the external data structures. Once distributed operation are provided, a first phase of the computation could proceed in a data parallel fashion (either doing attribute or data partitioning).

We argue that it is possible to achieve an efficient implementation of these operations. We will take advantage of the fact that, in the SkIE environment, this implementation is completely independent from the details of the DM algorithms.

## 4 Current results

Up to now, the path described has been followed to its 2nd step, with work ongoing to reach step 3. We call the prototype at step 1 MP+SM, and the current one DT (it is the same with the Delayed Threshold calculation ).

Fig. 1 shows the parallel structure we used for both. The data are still replicated. The task parallelism is applied in the Divide phase through a farm skeleton, each task contains a single node which has to be expanded into a subtree. As we have said, the depth $l$ of the computed subtrees is used to tune the amount of expansion. The expansion politics is explained later.

A single process owns the decision tree and does the Conquer step; it collects and joins subtrees. Nodes still needing to be expanded are sent back to the input of the parallel loop. The Conquer phase is executed in pipeline with the Divide phase inside the loop, to hide its latency.

All the test results are measured on a QSW CS-2 with 10 SPARC processors and a high performance communication network, using the data set Adult from the UCI repository.

With the farm dynamic load balancing, we don't need to evaluate in advance the computation needed by the subtree, which is repeatedly expanded no more than $l$ levels each time. It is enough that the variance of the node workload is bounded. Our work has been oriented at reducing this variance.

Each task requires an amount of communication proportional to the size of its partition, so using the shared objects to store the data keeps communication and computation costs closer. The gain is clear in Fig.2a from the comparison between MP and MP+SM execution times.

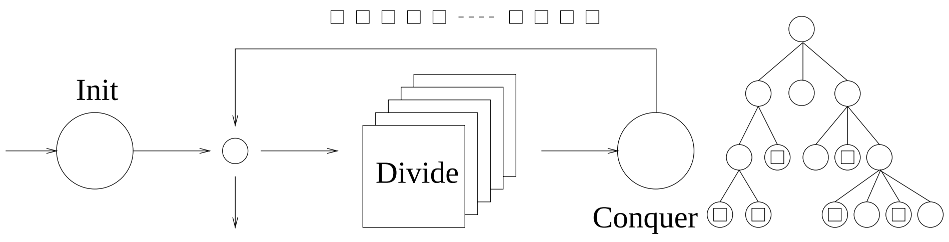

**Figure1.** Parallel structure of the prototype

The communication latency becomes almost independent from the size of the task, see in Fig.3b the communication cost for one worker. The communication overhead exceeds the computation only for very small tasks.

The MP+SM solution has another source of load imbalancing in the threshold calculation, which is $O(N \log N)$ operations. Delaying the calculation of thresholds until the end of the growth phase minimizes the load variance, allows a *D&C* formalisation and reduces the computational effort [4,6].

The comparison of MP+SM and TD in fig.3a underlines the positive effect of delaying the threshold calculation. The overall effect on the completion time is even greater, as reported in fig.2a.

Now the behaviour of the application is easier to analyze than with MP and MP+SM, with different parameters having more understandable effects.

Using a simple depth-first visit does not allow to exploit task parallelism in full. Moreover, since the average expansion cost increases with node size, giving priority to the bigger nodes results in a regular, decreasing computational load, which can be easily balanced.

The depth first selection in the first versions of MP made unsuccessful both the standard scheduling policies of the support and any attempt to use adaptive expansion at the workers.

If task computation time is too low w.r. to communication, parallelism is useless. Varying the amount of node expansion in the sequential code now succeeds in controlling the computation to communication ratio. In fig.3b it is easy to see the task size where communication (and idle) time and computation time on average balance. Here a second parameter controlling the expansion comes into play, the maximum amount of nodes for a subtree. In fig.2b we can see that raising this parameter from 512 to 2048 allows most of the small nodes to be computed immediatly instead of becoming new smaller tasks, this way improving the computation to communication ratio.

In fig.3b the communication time is higher for the same code when run with 6 workers instead of just one. The Conquer process is too busy to keep up with the output of the farm: it definitely becomes a bottleneck. This is partly due to the amount of communication, and mostly to the fact that the sequential code inside the module spends most of the time inside recursive visits the tree.

Theoretically the conquer operation should be fast, but the amount of work required to update the centralized data structure, as well as some inefficiency in its implementation, are no longer negligible. The effect shows up clearly with more and more worker nodes, and gets worse when there is more useful parallelism, as the arrival rate of new tasks increases.

## 5 Expected Results and Future work

The next change to the application will be needed to remove the Conquer bottleneck. We could change the tree visits into heap accesses, lowering the overhead, but we see as a long term solution to store the tree in the shared memory space.

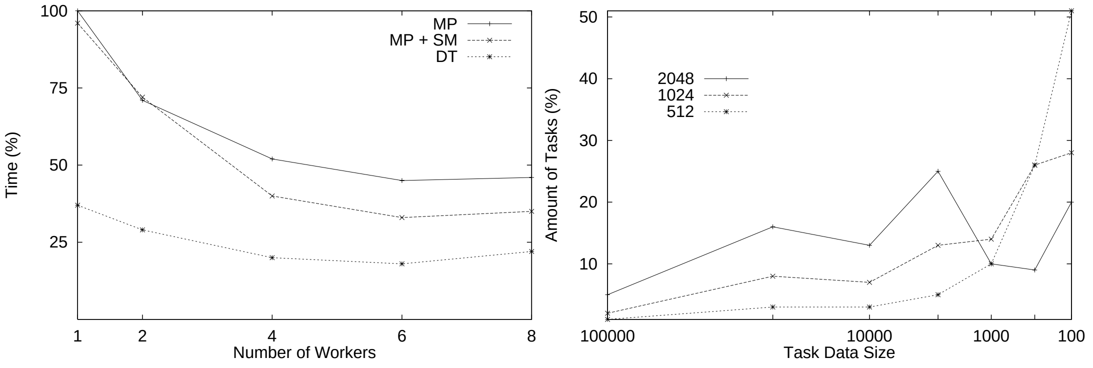

**Figure2.** (a) Results of the three different parallel implementations. (b) Distribution of task size with a varying subtree buffer size.

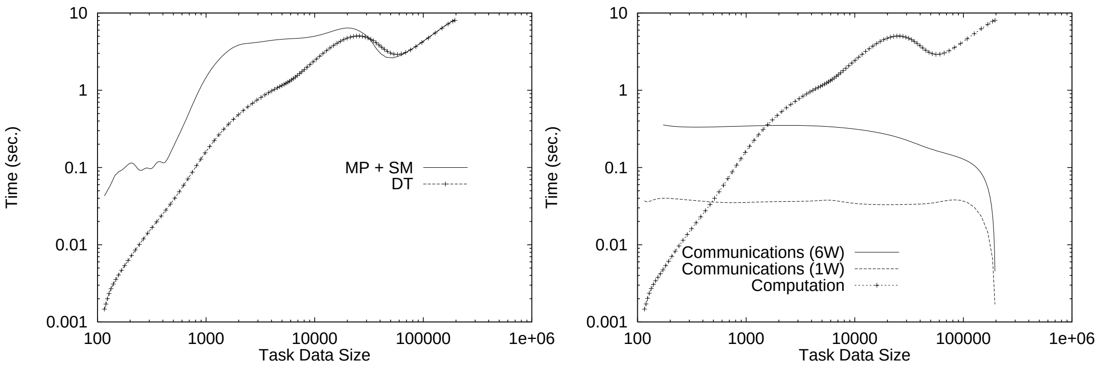

**Figure3.** (a) Computation time of task w.r.t. to size, with and without threshold calculation. (b) Computation vs communication and idle time per task, 1 and 6 workers.

Handling the merge operations in a decentralized way will lead to higher parallel scalability.

The utilization of the shared space is also needed to overcome the current assumption that the database fits in memory. A separate development path has already started about maintaining the database and the tree as linked structures in the shared memory space.

The current results, obtained through the integration of the shared objects abstraction into the SkIE skeleton programming model, strongly suggest that the approach is effective in solving the communication/computation problems for a class of irregular computations involving large data sets.

SkIE makes also easy to use the current parallel application as a building block for bigger ones. It is straightforward to set up a control loop that uses a set of C4.5 parallel blocks to scan multiple windows of a database at the same time.

We have not yet addressed the first few iterations in the task parallelisation: the expansion time for the first nodes accounts for 10% to 20% of the computation time, and things would get worse with bigger databases. The solution will be to compute the huge tasks exploiting data parallelism, and to switch to task parallelism as soon as no big tasks are still waiting. This can be done by using a new skeleton of the SkIE environment, which can switch from on demand scheduling to data-driven one. Secondary memory algorithms could be used to efficiently implement global operations like distributed sorting of the data.

Summing up we could move from a main-memory implementation of C4.5 to a skeleton structured implementation, where the main layer of the memory is the aggregate memory of the parallel architecture, whether a massively parallel architecture or a cluster of workstation with virtual shared memory support. The solutions devised should also be applied to the design of a generic skeleton composition that implements, at least, those D&C computations in which the conquer function is basically a merge operation.

## References

1. P. Becuzzi, M. Coppola, D. Laforenza, S. Ruggieri, D. Talia, and M. Vanneschi. Data analysis and data mining with parallel architectures: Techniques and experiments. Technical report, Consorzio Pisa Ricerche, project "Parallel Intelligent Systems for Tax Fraud Detection", December 1998.
2. P. Becuzzi, M. Coppola, and M. Vanneschi. Association rules in large databases, additional results. http://www.di.unipi.it/∼coppola/ep99talk.ps, Aug 1999.
3. P. Becuzzi, M. Coppola, and M. Vanneschi. Mining of Association Rules in Very Large Databases: a Structured Parallel Approach. In *Euro-Par'99 Parallel Processing*, volume 1685 of *LNCS*. Springer, 1999.
4. John Darlington, Yike Guo, Janjao Sutiwaraphun, and Hing Wing To. Parallel Induction Algorithms for Data Mining. In *Advances in intelligent data analysis: reasoning about data IDA'97*, volume 1280 of *LNCS*, 1997.
5. J.R. Quinlan. *C 4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, 1993.
6. S. Ruggieri. Efficient C4.5. Draft, http://www-kdd.di.unipi.it/software.
7. John Shafer, Rakesh Agrawal, and Manish Mehta. SPRINT: A Scalable Parallel Classifier for Data Mining. In *Proceedings of the 22nd VLDB Conference*, 1996.
8. A. Srivastava, E.H. Han, V. Kumar, and V. Singh. Parallel Formulations of Decision-Tree Classification Algorithms. *Data Mining and Knowledge Discovery*, 3(3), 1999.
9. M. Vanneschi. PQE2000: HPC Tools for Industrial Applications. *IEEE Concurrency: Parallel, Distributed & Mobile Computing*, 6(4):68–73, Oct-Dec 1998.
10. Jeffrey Scott Vitter. External Memory Algorithms and Data Structures: Dealing with MASSIVE DATA. Draft, http://www.cs.duke.edu/~jsv, January 2000.
11. Mohammed J. Zaki, Ching-Tien Ho, and Rakesh Agrawal. Scalable Parallel Classification for Data Mining on Shared-Memory Multiprocessors. In *Proc. of the IEEE Int'l Conference on Data Engineering*, March 1999.