Università di Pisa

Dipartimento di Informatica

Technical Report: TR-11-06

Porting Decision Tree Building and Pruning Algorithms to Multicore using FastFlow

Marco Aldinucci Salvatore Ruggieri Massimo Torquati

March 17th, 2011

ADDRESS: Largo B. Pontecorvo 3, 56127 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

Porting Decision Tree Building and Pruning Algorithms to Multicore using FastFlow

Marco Aldinucci[*] Salvatore Ruggieri[†] Massimo Torquati[‡]

March 17th, 2011

Abstract

The whole computer hardware industry embraced multicores. For these machines, the extreme optimisation of sequential algorithms is no longer sufficient to squeeze the real machine power, which can be only exploited via thread-level parallelism. Decision tree algorithms exhibit natural concurrency that makes them suitable to be parallelised. This paper presents an approach for *easy-yet-efficient* porting of an implementation of the C4.5 algorithm on multicores. The approach is based on the FastFlow parallel programming environment. The strength of our porting consists in minimal changes to the original sequential code. In addition to the tree building algorithm, we consider also the so far unaddressed problem of parallelising the error-based pruning with grafting algorithm of C4.5. We devise lower bounds for the forms of parallelisations adopted, and achieve performances close to such bounds.

Keywords Parallel classification, multicores, C4.5, error-based pruning, structured parallel programming, streaming parallelism.

1 Introduction

Computing hardware has evolved to sustain an insatiable demand for high-end performances along two basic ways. On the one hand, the increase of clock frequency and the exploitation of instruction-level parallelism boosted the computing power of the single processor. On the other hand, many processors have been arranged in multi-processors, multi-computers, and networks of geographically distributed machines. This latter solution exhibits a superior peak performance, but it incurs in significant software development costs. In the last two decades, the parallel computing research community has aimed at designing languages and tools to support the seamless porting of applications and the

[*]Computer Science Department, University of Torino, Italy. Email: adinuc@di.unito.it
[†]Computer Science Department, University of Pisa, Italy. Email: ruggieri@di.unipi.it
[‡]Computer Science Department, University of Pisa, Italy. Email: torquati@di.unipi.it

tuning of performances (e.g. Blumofe et al (1996); Pacheco (1996); Thies et al (2002)). These languages, apart from few exceptions that also focus on code portability, require a redesign of the application logic in an explicitly parallel language or model.

Up to now, clock speed and algorithmic improvements have exhibited a better performance/cost trade-off than application redesign, being the possibility to preserve the existing code its most important component. Data mining is not an exception in this regard. By surveying the papers in the main conferences and journals (Kargupta, 2011), there has been a diminishing number of proposals for parallel implementations of data mining algorithms in the last few years. After all, only a small fraction of data analysis projects can afford the cost of buying and maintaining a parallel machine and a data mining software capable of exploiting it. In most cases, data reduction techniques (such as sampling, aggregation, feature selection) can mitigate the problem while waiting the advancement in computational power of low-cost workstations.

Nowadays, however, this vision should be reinterpreted. After years of continual improvement of single core chips trying to increase instruction-level parallelism, hardware manufacturers realised that the effort required for further improvements is no longer worth the benefits eventually achieved. Microprocessor vendors have shifted their attention to thread-level parallelism by designing chips with multiple internal cores, known as Multicore or Chip Multiprocessors (Sodan et al, 2010). However, this process does not always translate into greater CPU performance: multicore are small-scale but full-fledged parallel machines and they retain many of their usage problems. In particular, sequential code will get no performance benefits from them. A workstation equipped with a quad-core CPU but running sequential code is wasting 3/4 of its computational power. Developers, including data miners, are then facing the challenge of achieving a trade-off between performance and human productivity (total cost and time to solution) in developing and porting applications to multicore. Parallel software engineering have engaged this challenge trying and design tools, in the form of high-level sequential language extensions and coding patterns, aiming at simplifying the porting of sequential codes while guaranteeing the efficient exploitation of concurrency (e.g Park et al, 2001; Vanneschi, 2002; Intel Corp., 2011).

This paper focuses on achieving this trade-off on a case study by adopting a methodology for the *easy-yet-efficient* porting of an implementation of the C4.5 decision tree induction algorithm (Quinlan, 1993) onto multicore machines. We consider the YaDT (Yet another Decision Tree builder) implementation of C4.5 (Ruggieri, 2004), which is a from-scratch and efficient C++ version of the well-known Quinlan's entropy-based algorithm. YaDT is the result of several data structure re-design and algorithmic improvements over Efficient C4.5 (Ruggieri, 2002), which is in turn is a patch to the original C4.5 implementation improving its performance mainly for the calculation of the entropy of continuous attributes. In this respect, we believe that YaDT is a quite paradigmatic example of sequential, already existing, complex code of scientific and commercial interest. In addition, YaDT is an example of extreme algorithmic sequential optimisation, for which the effort in designing further improvements would result in a minimal impact on the overall performances. Nevertheless, the potential for improvements is vast, and it resides in the idle core CPUs on the user's machine.

Our approach for parallelising YaDT is based on the FastFlow programming framework (Aldinucci and Torquati, 2011), a recent proposal for parallel programming over multicore platforms that provides a variety of facilities for writing efficient lock-free parallel patterns, including pipeline parallelism, task parallelism and Divide&Conquer computations. Besides technical features, FastFlow offers an important methodological approach that will lead us to parallelise YaDT with minimal changes to the original sequential code, yet achieving up to 7× boost in performance on a Intel dual-quad core. Nevertheless, MIPS, FLOPS and speedup have not to be the only metrics in software development. Human productivity, total cost and time to solution are equally, if not more, important (see e.g., Reed (2009)).

This paper includes considerable advancements over its conference version (see Aldinucci et al (2010b)). In addition to providing an extensive coverage of the FastFlow environment, and a in-depth description and experimental analysis of the tree building algorithm and of its parallelisation, we tackle the problem of parallelising the error-based tree pruning phase of C4.5. Such a phase is considerably faster than tree building on sequential algorithms, but it becomes the next bottleneck once the building phase has been parallelised and sped up. To the best of our knowledge, this is the first work to deal with parallel pruning. Moreover, we devise lower bounds for the forms of parallelisations adopted in both the building and pruning phases, and show how the performances achieved are close to such bounds. On the side of the experimental setup, we move from a 32-bit to a 64-bit compilation of all software, both sequential and parallel, as 64-bit machines are becoming the norm nowadays. Also, up-to-date hardware architectures are adopted in experiments.

The rest of the paper is organised as follows. In Sect. 2, the FastFlow programming environment is introduced. We recall in Sect. 3 the C4.5 decision tree building algorithm, including the main optimisations that lead to YaDT. The parallelisation of YaDT tree building is presented in detail in Sect. 4, followed by an experimental evaluation in Sect. 5. Similarly, the tree pruning algorithm is presented in Sect. 6, parallelised in Sect. 7, and experimented in Sect. 8. Finally, we discuss related works in Sect. 9, and summarise the contribution of the paper in the conclusions.

2 The FastFlow Parallel Programming Environment

FastFlow is a parallel programming framework aiming at simplifying the development of applications for multicore platforms, being these applications either brand new or ports of existing legacy codes. FastFlow promotes pattern-based programming and it has been specifically designed to efficiently support fine-grained parallel computations.

The key vision underneath FastFlow is that effortless development and efficiency can be both achieved by raising the level of abstraction in application design, thus providing designers with a suitable set of parallel programming patterns that can be compiled onto networks of parallel activities on the target platforms. FastFlow restricts the kind and the number of paradigms implemented in the library in such a way that a small but significant pattern set is implemented. The limited size of the library helps in keeping the implementation relatively easy to tune and maintain but does not affect expressivity: since patterns are implemented as C++ templates they can be extended using standard object oriented techniques without having to learn any exotic programming model.

Basically, parallel patterns implement structured synchronisations among concurrent entities where shared memory pointers are passed in a consumer-producer fashion. In particular, the FastFlow run time support takes care of all the synchronisations needed and related to the communication among the different parallel entities resulting from the compilation of the high level FastFlow pattern(s) used in an application. Those patterns model most of the typical stream based parallelism exploitation forms, including: *farm*, *farm-with-feedback* (suitable for implementing Divide&Conquer computations), *pipeline*, and their arbitrary nesting and composition.

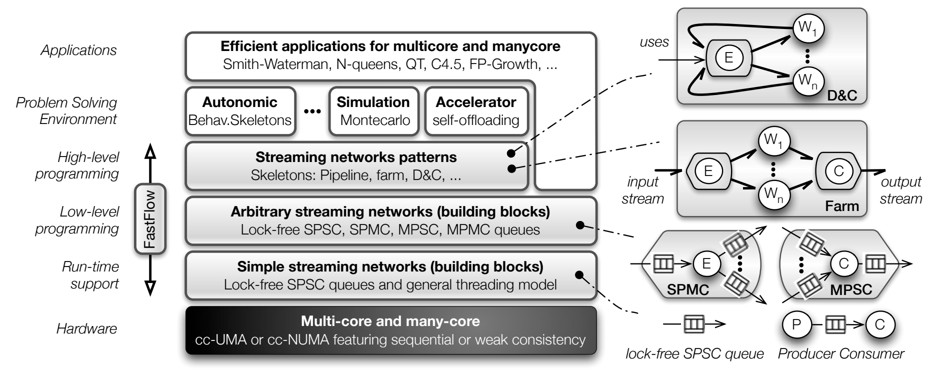
Thanks to the efficient synchronisation mechanisms provided by FastFlow, data parallel computations may also be implemented using the framework, as an example by transforming a data parallel computation (e.g. do-independent loop, map-reduce, etc.) into computations on a stream of items, each one corresponding to one or more pieces of data in which the data parallelism is exploited (i.e. sub-tasks). The possibility to efficiently handle both stream parallel and data parallel computations using the same general programming model represents a key advantage of FastFlow with respect to other frameworks that support either stream or data parallel computations.

The FastFlow patterns can be arbitrarily nested to model increasingly complex parallelism exploitation patterns. FastFlow implementation guarantees an efficient execution of the skeletons on currently available multicore systems by building the skeletons themselves on top of a library of lock free producer/consumer queues.

2.1 FastFlow Architecture

As shown in Fig. 1, FastFlow is conceptually designed as a stack of layers that progressively abstract the shared memory parallelism at the level of cores up to the definition of useful programming constructs and patterns. At the lowest tier of the FastFlow system we have the architectures that it targets: cache-coherent multiprocessors, and in particular commodity homogeneous multicore (e.g. Intel core, AMD K10, etc.).

The second tier provides mechanisms to define simple streaming networks



Applications
Efficient applications for multicore and manycore
Smith-Waterman, N-queens, QT, C4.5, FP-Growth, ...
Problem Solving Environment
Autonomic
Behav.Skeletons
Simulation
Montecarlo
Accelerator
self-offloading
High-level programming
Streaming networks patterns
Skeletons: Pipeline, farm, D&C, ...
Low-level programming
Arbitrary streaming networks (building blocks)
Lock-free SPSC, SPMC, MPSC, MPMC queues
Run-time support
Simple streaming networks (building blocks)
Lock-free SPSC queues and general threading model
Hardware
Multi-core and many-core
cc-UMA or cc-NUMA featuring sequential or weak consistency
FastFlow
uses
D&C
input stream
output stream
Farm
SPMC
MPSC
lock-free SPSC queue
Producer Consumer

Figure 1: FastFlow layered architecture with pattern examples.

whose *run-time support* is implemented through correct and efficient lock-free Single-Producer-Single-Consumer (SPSC) queues. This kind of queues do not requires any lock or memory barrier,[1] solid ground for a low-latency synchronisation mechanism for multicore. These synchronisations, which are asynchronous and non-blocking, do not induce any additional cache invalidation as it happens in mutual exclusion primitives (e.g. locks and interlocked operations), and thus do not add any extra overhead. SPSC FIFO queues is effectively used to build networks of communicating threads, which behave in a dataflow fashion. The formal underpinning of these networks dates back to Kahn Process Networks by Kahn (1974) and Dataflow Process Networks by Lee and Parks (1995).

The third tier generalises one-to-one to one-to-many (SPMC), many-to-one (MPSC), and many-to-many (MPMC) synchronisations and data flows, which are implemented using only SPSC queues and arbiter threads. This abstraction is designed in such a way that arbitrary networks of activities can be expressed while maintaining the high efficiency of synchronisations. The next layer up, i.e., *high-level programming*, provides a programming framework based on parallelism exploitation patterns. They are usually categorised in three main classes: Task, Data, and Stream Parallelism. FastFlow specifically focuses on Stream Parallelism. The skeleton disciplines concurrency exploitation within the generated parallel code: the programmer is not required to explicitly interweave the business code with concurrency related primitives.

The FastFlow patterns are implemented using the queues provided in FastFlow lower layers. As an example, a farm is implemented as shown in Fig. 1, mid right schema. An SPMC queue is used to dispatch tasks appearing onto the input stream towards a pool of *worker* threads. These, in turn, deliver results to a MPSC queue that eventually delivers the results onto the output stream.

As previously stated, the behaviour of patterns can be modified via object-oriented inheritance and polymorphism. As an example, in case ad-hoc scheduling and collection policies are needed with respect to the ones provided by the

[1]for Total Store Order processors, such as Intel core, AMD K10.

framework, the *emitter* (E) and the *collector* (C) node classes may be extended with application specific user policies (possible examples are FIFO ordering or priority scheduling/collection for the tasks, as we shall see in Sect. 4). The *farm* implementation can actually be seen as a three stage pipeline, the three stages being the emitter, the string of workers and the collector. The implementation of the high level parallel patterns has been designed to exploit the underlying mechanisms in such a way that high-level, efficient parallel constructs can be used by application programmers without being concerned by any of the related implementation details.

FastFlow is available as open source software under LGPLv3. Implementation details as well as performance comparisons against other programming tools such as POSIX, Cilk, OpenMP, and Intel TBB have been reported in Aldinucci and Torquati (2011) and Aldinucci et al (2010a).

2.2 Parallel Programming with Patterns: Background

Parallel programming patterns, also known as "algorithmic skeletons" (Cole, 1989), have been proposed as a vehicle to raise the level of abstraction of parallel programming aiming at simplifying the development of efficient parallel applications. Ideally, parallel patterns tackle with the difficult problems of parallel programming (i.e. concurrency exploitation, orchestration, mapping, tuning) moving them from the application design to development tools by capturing and abstracting the common paradigms of parallel programming and providing them with an efficient implementation, i.e., a toolkit of code generation techniques and a pre-optimised run-time support.

Traditionally, in parallel pattern-based programming the computation is organised according to application-independent high-level paradigms, which are usually categorised in three classes:

1. *Data Parallelism* is a method for parallelising a single task by processing independent data elements in parallel. Data parallelism also supports loop-level parallelism where successive iterations of a loop working on independent or read-only data are paralleled in different flows-of-control and concurrently executed.

2. *Task Parallelism* consists of running the same or different code on different executors (cores, machines, etc.). Task parallelism is usually explicit in the algorithm. Different flows-of-control (threads, processes, etc.) may communicate with one another as they work. Communication usually takes place to pass data from one thread/process to one or many others.

3. *Stream Parallelism* consists in the parallel processing of different items of a data stream, which can be either the input data or generated by the application internal programming mechanisms (e.g. via asynchronous function calls). It can be used when there exists a partial or total order in a computation; the *pipeline* is a paradigmatic stream parallel pattern.

Pragmatically, a given computational problem typically admits several algorithmic solutions exploiting patterns in different classes, or a different compositions of them. In addition, in many cases, patterns in different classes can be simulated one another. The extent of this generality is actually dependent by the set of patterns provided by a specific framework, which can also be designed to target one or more applicative scenarios (Asanovic et al, 2009; Dean and Ghemawat, 2008).

After (Cole, 1989) seminal work, early proposals of pattern-based programming frameworks have mainly focused on distributed memory platforms (e.g. clusters of workstations, grid); some of them, e.g. Google's MapReduce (Dean and Ghemawat, 2008), have evolved in mainstream of programming tools (Hadoop, 2011). Recent proposals, following the platform architecture trend, have shifted the focus to include multicores and shared address model; in addition to academic initiatives, such as FastFlow, it is worth mentioning consolidated industrial products such as the Intel Threading Building Block (TBB) library (Intel Corp., 2011), and to a limited extent the Microsoft Task Parallel Library (Leijen and Hall, 2007). These framework and their main features are surveyed in the related work section (Sect. 9).

Among this latter class of frameworks we selected FastFlow for the parallel implementation of YaDT for three main reasons:

1. As previously mentioned, the FastFlow framework puts forward stream parallelism as the primary class of patterns; they are then used to implement patterns from other classes by way of the layered architecture. In particular, the Divide&Conquer pattern, which is extensively used in YaDT, is naturally implemented by the *farm-with-feedback* FastFlow pattern. This pattern can be also realised with other frameworks directly using asynchronous function calls and recursion, but none of the most popular ones natively expose it as a pattern.

2. FastFlow is specifically designed to support the porting of existing software through the local refactoring of limited portions of code. This aim is shared with other programming framework, such as TBB and OpenMP that, however, mainly support the exploitation of loop-level data parallelism (e.g., *do_independent*) whereas the exploitation of other patterns of parallelism may require a substantial re-factoring of the code.

3. Thanks to the lock-free synchronisation mechanisms, FastFlow has shown a superior speedup on fine and very-fine grain tasks with respect to other frameworks, such as TBB and OpenMP (Aldinucci et al, 2010a)[2]. This flexibility is a key feature for application scalability in highly dynamic applications such as YaDT, where parallel tasks exhibit a large range of sizes, especially for data set leading to unbalanced decision trees.

[2]On a standard Intel core 2 FastFlow exhibits a reasonable speedup for tasks with execution time on the range of dozen of nanoseconds, whereas TBB requires a minimum granularity that is three orders of magnitude higher.

3 Decision Tree Building

Classification models, or simply *classifiers*, are induced by supervised learning from a relation $\mathcal{T}$ called the *training set*. An attribute C of the relation is called the *class*, while the remaining ones $A_1, \ldots, A_m$ are called the *predictive attributes*. Tuples in $\mathcal{T}$ are called *cases*. The overall objective of a classifier is to predict or to explain the value of the class on the basis of the values of predictive attributes by generalising the dependencies occurring in the cases from the training set. The *domain* of an attribute $dom(A_i)$ can be discrete, namely a finite set of values, or continuous, namely the set of real numbers. Also, the special value *unknown* is allowed in $dom(A_i)$ to denote unspecified or unknown values. The domain of the class $dom(C) = \{c_1, \ldots, c_{NC}\}$ is discrete and it does not include the unknown value.

A *decision tree* is a classification model in the form of a tree consisting of *decision nodes* and *leaves*. A leaf specifies a class value. A decision node specifies a *test* over one of the predictive attributes, which is called the attribute *selected* at the node. For each possible outcome of the test, a child node is present. A test on a discrete attribute A has h possible outcomes $A = d_1, \ldots, A = d_h$, where $d_1, \ldots d_h$ are the known values in $dom(A)$. A test on a continuous attribute has 2 possible outcomes, $A \leq t$ and $A > t$, where t is a *threshold* value determined at the node.

3.1 The C4.5 Tree-Induction Algorithm

The C4.5 decision tree induction algorithm by Quinlan (1993) is a constant reference in the development and analysis of novel proposals of classification models (see e.g., the comparison paper by Lim et al (2000)), and it is commonly considered among the top algorithms in data mining (see Wu et al (2008)). The core algorithm constructs the decision tree top-down in the *growing phase*, then prune branches by a bottom-up transversal in the *pruning phase*. Here, we concentrate on the growing phase, while the pruning phase will be presented in Sect. 6.

In the growing phase, each node is *associated* with a set of weighted cases, where weights are used to take into account unknown attribute values. At the beginning, only the root is present, with associated the whole training set $\mathcal{T}$ and all weights set to 1. At each node a Divide&Conquer algorithm is adopted to select an attribute for splitting. We refer the reader to the method `node::split` in Fig. 2 from the YaDT implementation of the algorithm.

Let T be the set of cases associated at a node. For every $c \in dom(C)$, the weighted frequency $freq(c, T)$ of cases in T whose class is c is computed (§2.2 — throughout the paper, we use the §M.N to reference line N from the pseudo-code in Fig. M). If all cases in T belong to the same class or the number of cases in T is less than a certain user specified threshold then the node is set to a leaf (§2.3-4). If T contains cases belonging to two or more classes, then the *information gain* of each attribute at the node is calculated (§2.6-7). Since the information gain of a discrete attribute selected in an ancestor node is necessarily 0, the number

```
     void node::split () {
2.2   computeFrequencies();
      if (onlyOneClass() || fewCases())
2.4     set_as_leaf () ;
      else {
2.6     for(int i=0;i<getNoAtts();++i)
          gain[i]= gainCalculation(i);
2.8     int best = argmax(gain);
        if (attr[best]. isContinuous())
2.10      findThreshold(best);
        ns=attr[best]. nSplits () ;
2.12    for(int i=0;i<ns;++i)
          childs . push_back(
2.14          new node(selectCases(best,i)));
      }
2.16 }
```

Figure 2: The original YaDT node splitting procedure.

```
     bool node::splitPre() {
3.2   computeFrequencies();
      if (onlyOneClass() || fewCases()) {
3.4     set_as_leaf () ;
        return true;
3.6   }
      return false;
3.8  }
     void node::splitAtt(i) {
3.10  gain[i]= gainCalculation(i);
     }
3.12 void node::splitPost() {
      int best = argmax(gain);
3.14  if (attr[best]. isContinuous())
        findThreshold(best);
3.16  ns=attr[best]. nSplits () ;
      for(int i=0;i<ns;++i)
3.18    childs . push_back(
            new node(selectCases(best,i)));
3.20 }
```

Figure 3: Partitioning of the `node::split` method into three steps.

of attributes to be considered at a node is variable (denoted by `getNoAtts` in §2.6).

For a discrete attribute A, the information gain of splitting T into subsets $T_1, \ldots, T_h$, one for each known value of A, is calculated[3]. For a continuous attribute A, cases in T with known value for A are first ordered ascending w.r.t. such an attribute. Let $v_1, \ldots, v_k$ be the ordered values of A for cases in T. Consider for $i \in [1, k-1]$ the value $v = (v_i + v_{i+1})/2$ and the splitting of T into cases T_1^v whose value for the attribute A is lower or equal than v, and cases T_2^v whose value is greater than v. For each value v, the information gain $gain_v$ is computed by considering the splitting above. The value v' for which $gain_{v'}$ is maximum is called the *local threshold* and the information gain for the attribute A is defined as $gain_{v'}$.

The attribute A with the highest information gain is selected for the test at the node (§2.8). When A is continuous, the *threshold* of the split is computed (§2.9-10) as the greatest value of A in the *whole* training set $\mathcal{T}$ that is below the local threshold. Finally, let us consider the generation of the child nodes (§2.12-14). When the selected attribute A is discrete, a child node for each known value from $dom(A)$ is created, and cases in T are partitioned over the child nodes on the basis of the value of the attribute A. When A is continuous two child nodes are created, and cases from T with known value of A are partitioned according to the boolean result of the test $A \leq t$, where t is the threshold of the split. Cases in T whose value for A is unknown are added to the set of cases of every child, but their weights are rebalanced.

[3]as follows: $gain(T, T_1, \ldots, T_h) = info(T) - \sum_{i=1}^{h} \frac{|T_i|}{|T|} \times info(T_i)$, where $info(S) = -\sum_{j=1}^{NC} \frac{freq(c_j,S)}{|S|} \times log_2(\frac{freq(c_j,S)}{|S|})$ is the entropy function.

3.2 From C4.5 to YaDT

The original Quinlan's implementation of C4.5 (Release 8, the latest) is written in ANSI C. It stores the training set in main-memory as an array of cases, and each case as an array of attribute values. The decision tree is grown depth-first. The computation of information gain takes $O(r)$ operations for discrete attributes, where $r = |T|$ is the number of cases at the node; and $O(r \, log \, r)$ operations for continuous attributes, where sorting is the predominant task. Finally, searching for the threshold of the selected continuous attribute (§2.10) requires $O(|\mathcal{T}|)$ operations, where $\mathcal{T}$ is the whole training set, since the data structure holding the training set allows only for a linear search. This prevents the implementation being truly a Divide&Conquer computation.

Efficient C4.5 (EC4.5, Ruggieri (2002)) is a patch software improving the efficiency of C4.5 in a number of ways. Continuous attribute values in a case are stored as indexes to the pre-sorted elements of the attribute domain. This allows for adopting a binary search of the threshold in the set of domain values at §2.10, with a computational cost of $O(log \, d)$ operations where $d = max_i |dom(A_i)|$. At each node, EC4.5 calculates the information gain of continuous attributes by choosing the best among three strategies according to an analytic comparison of their efficiency: the first strategy adopts *quicksort*; the second one adopts *counting sort*, which exploits the fact that in lower nodes of the tree ranges of continuous attributes tend to be narrow; the third strategy calculates the local threshold using a main-memory version of the RainForest algorithm without any sorting (Gehrke et al, 2000). C4.5 and EC4.5 are tied to a depth-first growth of the decision tree since they use a single array for storing cases, and they rearrange the cases of a child node to the initial segment of the array.

YaDT (Ruggieri, 2004) is a from scratch C++ implementation of C4.5. It inherits the optimisations of EC4.5, and adds further ones, such as searching the local threshold for continuous attributes by splitting at *boundary* values (Fayyad and Irani, 1992). Concerning data structures, the training set is now stored by columns, since most of the computations scan data by attribute values. Also, the cases at a node and their weights are stored as an array of pairs: the row-id's w.r.t. the training set, and the weight value. This and the object oriented design of YaDT allow for encapsulating the basic operations on nodes into a C++ class, with the advantage that the growing strategy of the decision tree can now be a parameter (depth first, breadth first, or any other top-down growth). By default, YaDT adopts a breadth first growth — which has a less demanding memory occupation. Its pseudo-code is shown in Fig. 4 as method `tree::build`. The call to the method `get_cases` (§4.11) builds the array of case row-id's and weights for a child node starting from the same data structure at the parent node. The method `release_cases` (§4.14) releases the data structure at the parent node once all children are queued. Experiments from Ruggieri (2002, 2004) show that YaDT reaches up to 10× improvement over C4.5 with only 1/3 of its memory occupation.

```
    void tree::build() {
4.2  queue<node *> q;
     root = new node( allCases );
4.4  q.push(root);
     while( !q.empty() ) {
4.6   node *n = q.front();
      q.pop();
4.8   n->split();
      for(int i=0;i<n->nChilds();++i) {
4.10    node *child = n->getChild(i);
        child->get_cases();
4.12    q.push( child );
      }
4.14  n->release_cases();
     }
4.16 }
```

Figure 4: YaDT tree growing procedure.

```
    void tree:: build_ff () {
5.2  root = new node( allCases );
     E=new ff_emitter(root,PAR_DEGREE);
5.4  std::vector<ff_worker*> w;
     for(int i=0;i<PAR_DEGREE;++i)
5.6   w.push_back( new ff_worker());
     ff_farm<ws_scheduler>
5.8     farm(PAR_DEGREE*QSIZE);
     farm.add_workers(w);
5.10 farm.add_emitter(E);
     farm.wrap_around();
5.12 farm.run_and_wait_end();
    }
```

Figure 5: YaDT-FF tree building setup.

4 Parallelising YaDT: Tree Building

We propose a parallelisation of YaDT, called YaDT-FF, obtained by stream parallelism. In this section, we concentrate on the growing phase. Each decision node is considered a task that generates a set of sub-tasks; these tasks are arranged in a stream that flows across a *farm-with-feedback* skeleton which implements the Divide&Conquer paradigm. The FastFlow Divide&Conquer schema is shown in the top-right corner of Fig. 1. Tasks in the stream are scheduled by an *emitter* thread towards a number of *worker* threads, which process them in parallel and independently, and return the resulting tasks back to the emitter. For the parallelisation of YaDT, we adopt a two-step strategy: first, we accelerate the `tree::build` method (see Fig. 4) by exploiting task parallelism among node processing, and we call this strategy *Nodes Parallelisation* (NP); then, we add the parallelisation of the `node::split` method (see Fig. 2) by exploiting parallelism also among attributes processing, and we call such a strategy *Nodes & Attributes Parallelisation* (NAP). The two strategies share the same basic setup method, `tree::build_ff` shown in Fig. 5, which creates an emitter object (§5.2-3) and an array of worker objects (§5.4-6). The size of the array, `PAR_DEGREE`, is the parallelism degree of the farm. The root node of the decision tree is passed to the constructor of the emitter object, so that the stream can be initiated from it. The overall farm parallelisation is managed by the FastFlow layer through a `ff_farm` object, which creates feedback channels between the emitter and the workers (§5.7-11). Parameters of `ff_farm` include: the size `QSIZE` of each worker input queue, and the scheduling policy (`ws_scheduler`), which is based on tasks weights. Basically, such a policy assigns a new task to the worker with the lowest total weight of tasks in its own input FIFO queue. The emitter class `ff_emitter` and the worker class `ff_worker` define the behaviour of the farm parallelisation through the class method `svc` (short name for *service*) that is called by the FastFlow run-time to process input tasks. Different parallelisation strategies can be defined by

```
     void * ff_emitter::svc(void * task) {
6.2    if (task == NULL) {
         task=new ff_task(root);
6.4      int r = root->getNoCases();
         setWeight(task, r);
6.6      return task;
       }
6.8    node *n = task->getNode();
       int nChilds = n->nChilds();
6.10   for(int i=0; i < nChilds; i++) {
         node *child = n->getChild(i);
6.12     ctask=new ff_task(child);
         child->get_cases();
6.14     int r = child->getNoCases();
         setWeight(ctask, r);
6.16     ff_send_out(ctask);
       }
6.18   n->release_cases();
       if (!nChilds && noMoreTasks())
6.20     return NULL;
       return FF_GO_ON;
6.22 }

6.24 void * ff_worker::svc(void * task) {
       node *n = task->getNode();
6.26   n->split();
       return task;
6.28 }
```

Figure 6: Emitter and Worker definition for the NP strategy.

changing only these two methods. The implementation of the NP and the NAP strategies are shown in Fig. 6 and Fig. 8 respectively. Let us describe them in detail.

NP strategy (Fig. 6). At start-up the `ff_emitter::svc` method is called by the FastFlow run-time with a NULL parameter (§6.2). In this case, a task for processing the root node is built (recall that the root node is passed to the constructor of the emitter, hence it is accessible), and its weight is set to the number of cases at the root (§6.3-5). By returning the task, the emitter queues it to some worker according to the weighted scheduling strategy. Upon receiving in input a task coming from a worker, the emitter produces in output the sub-tasks corresponding to the children of the node (§6.8-18). The `ff_send_out` method of the FastFlow runtime allows for queueing tasks without returning from the method. Notice that child nodes call the `get_cases` method to retrieve their cases from the parent node, and, finally, the parent node calls `release_cases` to free its cases. If there are no child node and no more tasks in worker queues (§6.19-20), the emitter returns NULL as to signal that the computation is finished. Otherwise, the `FF_GO_ON` tag in the return statement (§6.21) tells the run-time that further tasks must be waited for from the input channel. The `ff_worker::svc` method for a generic worker (§6.24-28) merely calls the node splitting algorithm `node::split`, and then it immediately returns the computed task back to the emitter. The overall coding is extremely simple and intuitive — almost a rewriting of the original `tree::build` method. Moreover, it is quite generalisable to any top-down tree-growing algorithm with greedy choice of the splitting at each node. The weighted scheduling policy is the most specific part; in particular, for the use of weights that are linear in the number of cases at the node. This is motivated by the experimental results of (Ruggieri, 2002, Fig. 1), showing that the YaDT implementation of `node::split` exhibits a low-variance elapsed time per case for the vast majority of nodes. Finally, Fig. 7 shows a snapshot of the NP strategy execution with two workers. A partially built tree is shown on the left hand side, with nodes 4 and 5 being processed by worker W_1 and W_2 respectively, and nodes 1, 2 and 3 already queued by the emitter. After processing node 4, worker W_1 delivers the task to the emitter MPSC queue, and it starts processing node 1 from its input queue. Similarly, worker W_2 delivers

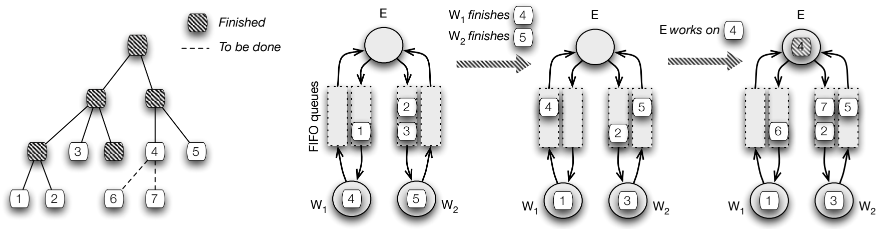


Figure 7: Example emitter and worker states in the NP strategy.

node 5, and it starts processing node 3. Assume that the emitter reads first node 4 from its MPSC input queue. Since node 4 has two child nodes, namely node 6 and 7, the emitter generates tasks for them, and delivers the tasks in the SPMC workers queues. The scheduling policy determines which worker each task is assigned to. Notice that, since queues are FIFO, the scheduling policy does not affect the order of tasks within a worker's queue.

NAP strategy (Fig. 8). The NAP strategy builds over NP. For a given decision node, the emitter follows a Divide&Conquer parallelisation over its children, as in the case of the NP strategy. In addition, for each child node, the emitter may decide to parallelise the calculation of the information gains in the `node::split` method (§2.6-7). In such a case, the stopping criterion at §2.3 must be evaluated prior to the parallelisation, and the creation of the child nodes must occur after all the information gains are computed. This leads to partitioning the code of `node::split` into three methods, as shown in Fig. 3.

For the root node, attribute parallelisation is always the case (§8.3-10). A task with label `BUILD_ATT` is constructed for each attribute, with the field `att` recording the attribute identifier (the index `i`). Tasks are weighted and queued. The information about how many tasks are still to be completed is maintained in the `child_cnt` field of the decision node — such a field is added to the original `node` class. Upon receiving in input a task coming from a worker, the emitter checks whether it concerns the processing of an attribute (§8.16). If this is the case (§8.17-20), the `child_cnt` counter is decremented until the last attribute task arrives, and then the `node::splitPost` method is called to evaluate the best split. At this point (§8.21), the emitter is provided with a processed node, either from a worker, or as the result of the `node::splitPost` call. For every child node, the cases are retrieved from the parent node (§8.24), and then the test `buildAttTest` at §8.28 controls whether to generate a single node processing task, or one attribute processing task for each attribute at the child node. In the former case (§8.28-30), we proceed as in the NP strategy; in the latter case (§8.32-37), we proceed as for the root node[4]. Once child nodes are generated, the parent node can free cases at the node (§8.41). Finally, if there are no child node and no more tasks in worker queues (§8.42-43), the emitter returns NULL as to signal that the computation is finished. Otherwise, it returns `FF_GO_ON`,

[4]Notice that tasks for node processing are labelled with `BUILD_NODE`, while tasks for attribute processing are labelled with `BUILD_ATT`

```
     void * ff_emitter::svc(void * task) {
8.2   if (task == NULL ) {
       if (root->splitPre()) return NULL;
8.4    int r = root->getNoCases();
       int c = root->getNoAtts();
8.6    for(int i=0;i<c;++i) {
        task=new ff_task(root,BUILD_ATT);
8.8     task->att = i;
        setWeight(task, r);
8.10    ff_send_out(task);
       }
8.12   root->child_cnt = c;
       return FF_GO_ON;
8.14  }
      node *n = task->getNode();
8.16  if (task->isBuildAtt()) {
       if (--n->child_cnt>0)
8.18    return FF_GO_ON;
       n->splitPost();
8.20  }
      int nChilds = n->Childs();
8.22  for(int i=0; i < nChilds; i++) {
       node *child = n->getChild(i);
8.24   child->get_cases();
       int r = child->getNoCases();
8.26   int c = child->getNoAtts();
       if (!buildAttTest(r,c)) {
8.28    ctask=new ff_task(child,BUILD_NODE);
        setWeight(ctask, r);
8.30    ff_send_out(ctask);
       } else {
8.32    if (child->splitPre()) continue;
        for(int j=0;j<c;++j) {
8.34     ctask=new ff_task(child,BUILD_ATT);
         ctask->att = j;
8.36     setWeight(ctask, r);
         ff_send_out(ctask);
8.38    }
        child->child_cnt = c;
8.40   }
       n->release_cases();
8.42  if (!nChilds && noMoreTasks())
       return NULL;
8.44  return FF_GO_ON; }

8.46 void * ff_worker::svc(void * task) {
      node *n = task->getNode();
8.48  if (task->isBuildAtt())
       n->splitAtt(task->att);
8.50  else
       n->split();
8.52  return task;
     }
```

Figure 8: Emitter and Worker definition for the NAP strategy.

thus waiting for other tasks. Concerning workers, based on the task label, the `ff_worker::svc` method (§8.47-53) merely calls the node splitting procedure or the information gain calculation for the involved attribute.

Let us now discuss in detail two relevant issues in the NAP strategy. Let r be the number of cases and c the number of attributes at a node.

The first issue is concerned with task weights. Node processing tasks are weighted with r (§8.29), as for the NP strategy. Attribute processing tasks have a finer grain, which suggests that they show be assigned a lower weight. However, we point out that, although attribute tasks are executed in parallel, there is a synchronisation point: all attribute tasks of a node must have been processed before the emitter could generate tasks for the child nodes. By giving a lower weight to attribute tasks, we run the risk that two or more of them are scheduled to the most unloaded worker, thus resulting in a sequential execution. For this reason, also attribute processing tasks are weighted with r (§8.9,§8.36).

The second issue is concerned with the test `buildAttTest`, which discriminates between nodes parallelisation and attributes parallelisation. Since the latter is finer grained, a test should select attributes parallelisation for larger nodes, and nodes parallelisation for smaller ones — where the size of a node is measured in terms of r and c. We have designed and experimented three test strategies. Attribute parallelisation is chosen respectively when:

- ($\alpha < r$) the number of cases at the node is above some hand-tuned threshold value α;
- ($|\mathcal{T}| < c\, r\, log\, r$) the average grain of node processing (sorting c attributes by quicksort algorithm, which is $r\, log\, r$ on average) is higher than a threshold that is dependent on the training set. Intuitively, the threshold should

Table 1: Training sets used in experiments, and size of unpruned decision trees.

			No. of attributes			Unpruned decision tree		
$\mathcal{T}$ name	$\|\mathcal{T}\|$	NC	discr.	contin.	total	size	depth	unbalancing
Census-Income	299,285	2	33	7	40	122,306	31	0.43
U.S. Census	2,458,285	5	67	0	67	125,621	44	0.49
KDD Cup 99	4,898,431	23	7	34	41	2,810	29	0.65
Forest Cover	581,012	7	44	10	54	41,775	62	0.42
SyD10M9A	10,000,000	2	3	6	9	169,108	22	0.37
SyD10M9A-05	10,000,000	2	3	6	9	184,325	24	0.32
SyD10M9A-005	10,000,000	2	3	6	9	128,077	23	0.45

be such that the test is satisfied at the root node, which is the coarser-grained task, and for nodes whose size is similar. Since the average grain of processing a single attribute at the root is $|\mathcal{T}|\, log\, |\mathcal{T}|$, we fix the threshold to a lower bound for such a value, namely to $|\mathcal{T}|$ — which turns out to be a lower bound for processing a single attribute at the root (attribute values must be scanned at least once);

- ($|\mathcal{T}| < c\, r^2$) the worst-case grain of node processing (quicksort is r^2 in the worst-case) is higher than a threshold that is dependent on the training set. As in the previous case, the threshold is set to the lower bound $|\mathcal{T}|$ for processing a single attribute at the root node. The higher value cr^2 in the right-hand-side, however, leads to selecting attributes processing more often than the previous case, with the result of *over-provisioning*, namely the creation of an higher number of (finer-grained) concurrent tasks.

As we will see in Sect. 5, the third test strategy shows the best performance.

5 Performance Evaluation I

In this section we show the performances obtained by YaDT-FF in the *growing phase*. The datasets used in experiments and their characteristics are reported in Table 1, including the number of discrete and continuous attributes, the size and depth of the decision tree, and its unbalancing factor (described later on). The datasets are publicly available from the UCI Machine Learning repository by Frank and Asuncion (2011), apart from datasets *SyD10M9A-xxx* which are synthetically generated using function 5 of the QUEST data generator by IBM Almaden (2003). We generated 3 distinct synthetic datasets by varying the distribution of the binary class as follows: 20%–80% for *SyD10M9A*; 50%–50% for *SyD10M9A-05*; and 5%–95% for *SyD10M9A-005*. *SyD10M9A* will be our reference synthetic dataset. All experimental results are taken by performing 5 runs, excluding the highest and the lowest values obtained and computing the average of the remaining ones.

Experimental framework. All experiments are executed on two different workstation architectures: *Nehalem)* a dual quad-core Intel Xeon E5520 Nehalem (16 HyperThreads) @2.26GHz with 8MB L3 cache and 24 GBytes of

Table 2: YaDT elapsed time (in seconds): 32-bit vs 64-bit compilation.

	Nehalem				Magny-Cours			
	32-bit		64-bit		32-bit		64-bit	
$\mathcal{T}$ name	build	prune	build	prune	build	prune	build	prune
Census-Income	3.55	0.97	3.16	0.79	4.63	1.09	4.62	1.09
U.S. Census	14.24	1.47	12.67	1.15	17.16	1.68	17.25	1.72
KDD Cup 99	17.16	6.5	15.39	5.85	22.67	10.81	22.62	10.86
Forest Cover	16.4	11.8	13.53	8.97	19.1	13.63	19.27	13.63
SyD10M9A	106.23	21.49	114.15	21.12	133.41	35.45	134.36	35.47

main memory with Linux x86_64; and *Magny-Cours)* a single 12 cores AMD Magny-Cours Opteron 6174 @2.2GHz with 12MB L3 cache and 128 GBytes of main memory, with Linux x86_64. They are quite standard representatives of current mid-to-high-end workstations.

The Nehalem-based machine exploits Simultaneous MultiThreading (SMT, a.k.a. HyperThreading) with 2 contexts per core and the novel Quickpath interconnect equipped with a distributed cache coherency protocol. The SMT technology makes a single physical processor appear as two logical processors for the operating system, but all execution resources are shared between the two contexts: caches of all levels, execution units, etc.

32-bit vs. 64-bit YaDT compilation. Preliminary results of the parallelisation of YaDT are reported in our conference paper by Aldinucci et al (2010b). There, we presented experiments on 32-bit compiled executables running on 64-bit Intel-based architectures. It is legitimate to ask ourselves whether the performances of YaDT are affected by a 64-bit compilation or on a different architecture. In Table 2 we show the sequential execution time obtained from both 32-bit and 64-bit versions of the YaDT building and pruning phases when running on 64-bit Intel and AMD architectures. As it can be observed, the 64-bit executable is moderately faster than the 32-bit one for almost all the datasets considered. One exception is the synthetic dataset (for the build phase), which shows a performance penalty of about 7%. The better performances of the 64-bit compilation can be justified by the fact that it provides better support of 64-bit integers (long data types in C/C++) and better utilisation of the underling 64-bit architecture. As a general rule, 64-bit code may benefit of extra registers not available for 32-bit code. On the other hand, the 64-bit compilation mode implies larger data types (mainly due to larger pointer representation and larger padding in data structures) hence more cache-misses. The tradeoff between this two aspects produces, in most cases, better performance results for the 64-bit code.

Since almost all newer server and workstations are 64-bit architectures and compilers running on 64-bit OSs produce 64-bit executable by default, in the rest of the paper we consider only 64-bit executions.

Parallel performance of the NP strategy. Let us start considering the parallelisation of nodes processing. The obtained speedup is shown in Fig. 9.

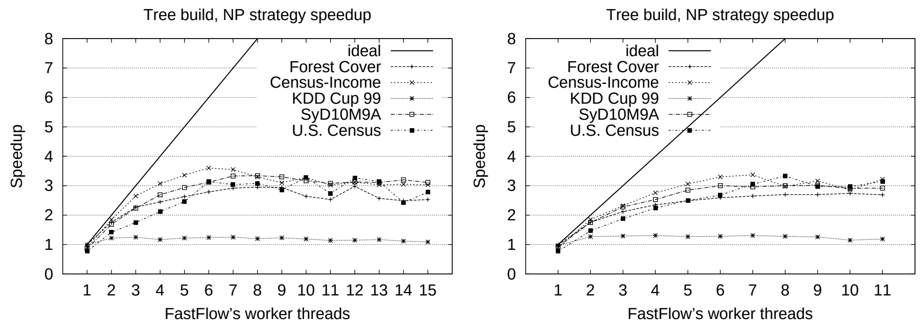


Figure 9: NP strategy speedup. Nehalem box (left), Magny-Cours box (right).

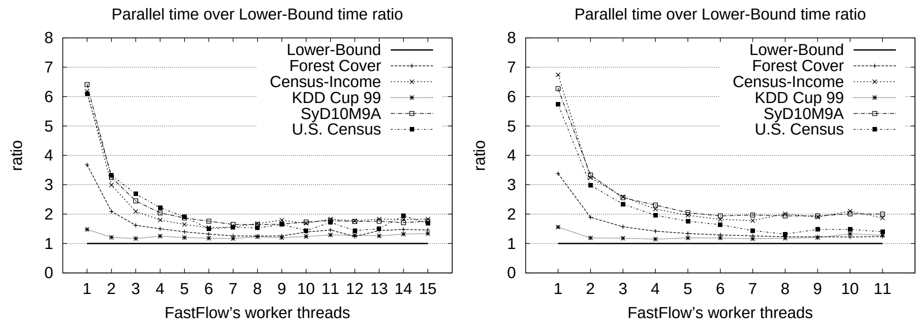
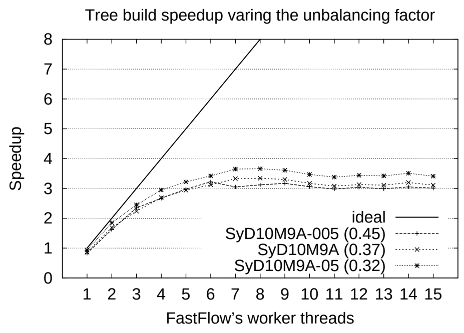


Figure 10: Elapsed over lower-bound time ratio of the tree build phase on the Nehalem box (left) and Magny-Cours box (right).

The maximum speedup is similar on both architectures, and quite variable from a dataset to another; it ranges from 1.31 for *KDD Cup 99* on the Magny-Cours box to 3.6 for the *Census-Income* dataset on the Nehalem box. As one would expect, exploiting inter-nodes parallelism alone is not enough to reach a close to optimal speedup, because a large fraction of the computing time is spent in the coarse-grained sequential computation of nodes, thus lacking enough parallelism. This phenomenon has been already observed in previous work on the (so called, task) parallelisation of decision tree construction over distributed memory architectures (see e.g., Srivastava et al (1999), and the related work Sect. 9). Let us provide here a meaningful justification by introducing a lower bound for any parallelisation strategy exploiting concurrency on the grain of nodes. Intuitively, the computation of a node can only start after its father has been processed, which, in turn, can only start after all of its ascendants have been processed. As a consequence, the elapsed time needed for the tree path with the highest computational cost is a lower bound for any nodes parallelisation strategy. In symbol, we write:

$$lb_b(n) = t_b(n) + max_{m \in child(n)} lb_b(m)$$



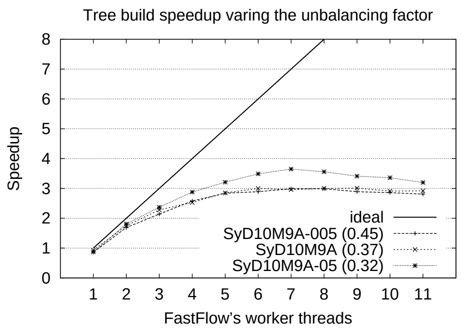


Figure 11: Speedup of the build phase for synthetic datasets with different unbalancing factors. Nehalem box (left), Magny-Cours box (right).

where $t_b(n)$ is the time for sequential processing (i.e., to execute `node::split`) of node n, and $child(n)$ is the set of child nodes of n. Observe that the lower bound is strict, since it assumes an oracle scheduler that gives priority to nodes along the path with the highest computational cost, a number of workers sufficient to compute in parallel all other nodes, and zero-time synchronisations.

From the experimental side, we have instrumented the sequential code to compute $lb_b(root)$ for the root node of the tree, after it has been completely built. Fig. 10 reports the ratio of the elapsed time of the NP strategy over the lower bound time. Notably, our implementation reaches a good efficiency, requiring at most twice the lower bound time. This confirms the effectiveness both of our design, in particular of the weighted scheduling policy as an online approximation of the oracle scheduler, and of the underlying FastFlow layer.

From the theoretical side, we observe that since $t_b(n)$ is proportional to the number of cases at node n, the path with the highest computational cost turns out to be the largest path, where the size of a path is measured as the sum of the number of cases at nodes in the path. In other words, the more a tree is unbalanced (cases are concentrated along a single path), the less a nodes parallelisation is efficient, independently from the number of worker threads and from the scheduling policy. In Table 1, the *unbalancing* column reports the ratio between the size of the largest path and the overall sum of the number of cases in all nodes of a tree. Notice that the three synthetic datasets *SyD10M9A-05*, *SyD10M9A* and *SyD10M9A-005* are similar as for number of cases and attributes, but they are purposely generated with increasing unbalance ratios. This was obtained by unbalancing the distribution of class values. Fig. 11 reports the speedups for the three synthetic datasets, confirming the theoretical analysis.

Summarising, although modest, the speedup of the NP strategy is notably close to the limit of its form of parallelisation, and, equally notably, it was achieved by a minimal effort to port the sequential code.

Parallel performance of the NAP strategy. The *NAP strategy* aims

	Total Execution Time (sec.)		
$\mathcal{T}$ name	$\lvert\mathcal{T}\rvert < cr^2$	$\alpha < r$	$\lvert\mathcal{T}\rvert < cr \log r$
Census-Income	0.76	0.77	**0.68**
U.S. Census	**2.1**	2.14	2.19
KDD Cup 99	**3.46**	3.45	3.60
Forest Cover	**1.83**	1.93	1.85
SyD10M9A	**17.07**	20.11	19.39

Effectiveness of *buildAttTest(c,r)* for different attributes cost models. $\lvert\mathcal{T}\rvert$ = no. of cases in the training set, c = no. of attributes at the node, r = no. of cases at the node, and $\alpha = 1000$.

Figure 12: Attributes parallelisation tests (Nehalem, 7 worker threads, best figures in boldface).

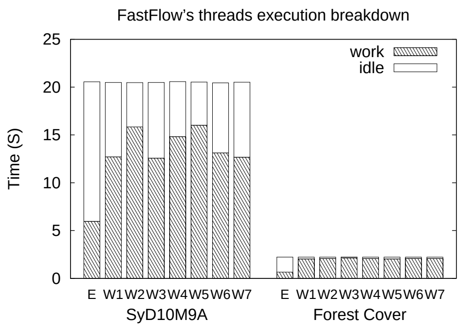
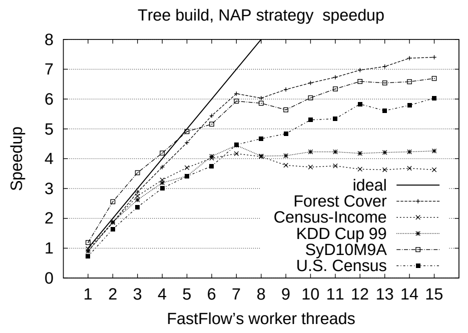


Figure 13: YaDT-FF build phase execution breakdown on emitter (E) and 7 worker threads (Nehalem).



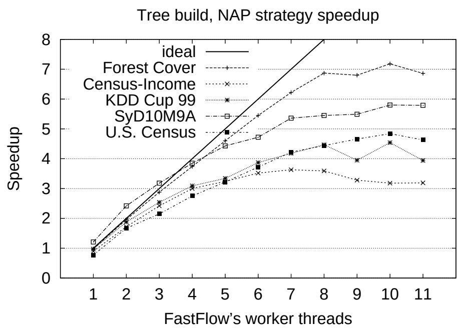
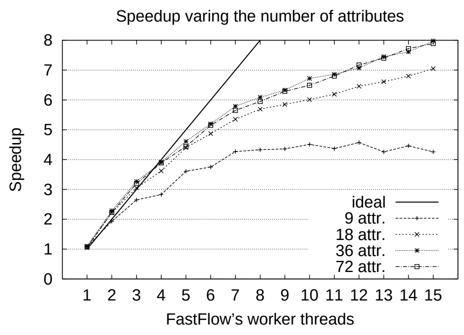


Figure 14: NAP strategy speedup. Nehalem box (left), Magny-Cours box (right).

at increasing the available parallelism by exploiting concurrency also in the computation of the information gain of attributes. This is particularly effective for nodes with many cases and/or attributes, because it reduces the sequential fraction of the execution. As discussed in Sect. 4, the emitter relies on a *cost model* in order to decide whether to adopt attributes parallelisation. We have tested the three cost models presented in Sect. 4. Fig. 12 shows that the test $|\mathcal{T}| < cr^2$ provides the best performance for almost all datasets. This is justified by the fact that the test produces an higher number of finer-grained tasks when compared to the test $|\mathcal{T}| < cr\, log\, r$, and it is dataset-tailored when compared to the test $\alpha < r$. In all of the remaining experiments, we use that model.

The speedup of YaDT-FF with the NAP strategy is shown in Fig. 14. It ranges from 4.1 to 7.4 on the Nehalem architecture and from 3.63 to 7.18 on the Magny-Cours box. The speedup gain over the NP strategy is remarkable. Only for the *Census-Income* dataset, the smallest dataset as for number of cases, the speedup gain is just +15% over NP on the Nehalem, and just +8% over NP on the Magny-Cours machine. Notice that the *SyD10M9A* dataset apparently ben-



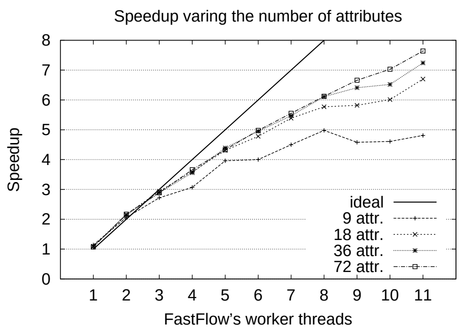
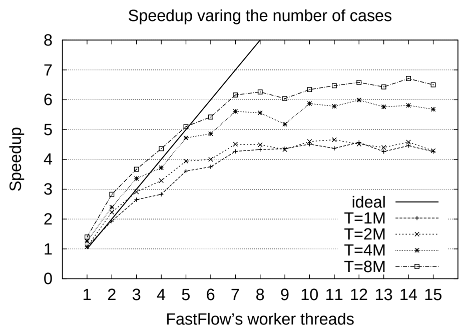


Figure 15: Speedup vs no. of attributes for $1M$ sample cases from *SyD10M9A*. Nehalem box (left), Magny-Cours box (right).



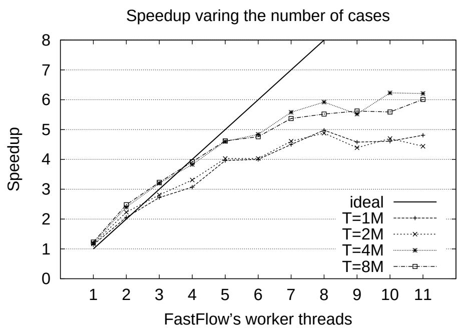


Figure 16: Speedup vs no. of sample cases (T) from *SyD10M9A*. Nehalem box (left), Magny-Cours box (right).

efits from a super-linear speedup. Actually, this happens because the speedup is plotted against the number of farm workers. Hence, the fraction of work done by the emitter thread is not considered, yet not negligible as shown in Fig. 13. As a matter of a fact, the FastFlow farm-with-feedback pattern implicitly exploits pipeline parallelism between the emitter and the generic worker (see also Zaki et al (1999)).

YaDT-FF also exhibits a good scalability with respect to both the number of attributes (Fig. 15) and to the number of cases (Fig. 16) in the training set. The plots refer to subsets of the *SyD10M9A* dataset possibly joined with randomly distributed additional attributes. In the former case, a speedup boost is obtained as soon as the number of attributes doubles the available hardware parallelism (18 attributes for 8 physical cores). In the latter case, the achieved speedup seamlessly increases with the number of cases in the training set.

Finally, we point out that a lower bound for the elapsed time of any parallelisation strategy exploiting attributes parallelism can be devised by the same reasonings as in the analysis of the NP strategy. Intuitively, the lower bound is

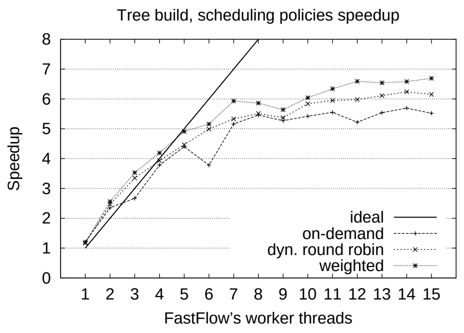


Figure 17: Speedup of different scheduling policies over *SyD10M9A*.

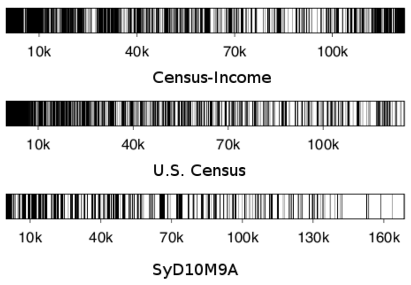


Figure 18: Nodes (white) vs attributes (black) parallelisation choices.

obtained by the elapsed time of the tree path with the highest computational cost, where the cost of processing a node is now the maximal sequential time for processing a single attribute at the node. Such a lower bound, however, results to be too strict, due to the fact that computing the information gain of an attribute for a small number of cases has a very fine-grain (for that reason, we switch to nodes parallelisation for smaller nodes), which is the order of the synchronisation overhead of any parallel approach.

Task scheduling. The parallelisation of decision tree construction algorithms may suffer from load balancing issues due to the difficulties in predicting the time needed for processing a node or a sub-tree. For example, the binary search of the threshold (§2.10) has to be performed only when a continuous attribute is selected for the test at a node. Fig. 13 shows that load balancing is not a critical issue for YaDT-FF with the NAP strategy. We motivate the good performance obtained by two main reasons: 1) the NAP strategy produces a significant over-provisioning of tasks with respect to the number of cores; these tasks continuously flow (in a cycle) from the emitter to the workers and they are subject to quite efficient online scheduling within the emitter; 2) FastFlow communications are asynchronous and exhibit very low overhead also for fine-grained tasks (see Aldinucci et al (2010a)). This makes it possible to sustain all the workers with tasks to be processed for the entire computation. The low overhead of the communications helps to reduce the dependence of the achieved speedup from the effectiveness of the scheduling policy. Nevertheless, such a dependence exists.

Fig. 17 shows results for three different scheduling policies: 1) Dynamic Round-Robin (DRR); 2) On-Demand (OD); 3) Weighted Scheduling (WS). The DRR policy schedules a task to a worker in a round-robin fashion, skipping workers with full input queue (with queue size set to 4096). The OD policy is a fully online scheduling, i.e., a DDR policy where each worker has an input queue of size 1. The WS policy is a user-defined scheduling that can be set up by assigning weights to tasks through calls to the `setWeight` method. We recall that YaDT-FF adopts a WS policy, with the weight of a task set to the number

r of cases at the node. It is immediate to observe from Fig. 17 that all the scheduling policies are fairly efficient. WS exhibits superior performance because it is tailored over the YaDT-FF algorithm; it actually behaves as a quite efficient online scheduling. A possible variant of WS with FIFO queues for workers' input would consist in re-arranging tasks that have already been scheduled to workers (e.g., via priority MPMC queues or work stealing strategies) on the basis of the their weights. However, since re-arranging cannot be implemented using solely the lock-free and fence-free mechanisms of FastFlow, it is likely that the additional overhead may easily overcome possible gains.

Finally, we show in Fig. 18 how often nodes parallelisation has been chosen by the emitter against the attributes parallelisation in the NAP strategy (we recall that the test $|\mathcal{T}| < cr^2$ was fixed). Black stripes lines in the figure denote attributes parallelisation choices whereas white stripes denote nodes parallelisation ones. As expected, the former occurs more often when processing the top part of the decision tree (from left to the right, in the figure).

Simultaneous MultiThreading. Let us briefly evaluate the benefits achieved using the Nehalem HyperThreaded box which may execute 2 threads simultaneously. SMT is essentially a memory latency hiding technique that is effective when different threads in a core exhibit a shared working set that induces high cache hit rate. However, even in non-ideal conditions, SMT is able to moderately increase instructions per clock-cycle count, hence the overall performance, by partially hiding costly main memory accesses with threads execution. Aldinucci et al (2010b) report a performance improvement attributable to SMT in the range 12% to 30%. From the plots in Fig. 14 is it possible to observe that the maximum speedup on both machines are very close, and that the HyperThreaded machine obtains a performance improvement for the *U.S. Census*, *SyD10M9A* and *Forest Cover* datasets that ranges from 22% to 29%, thus confirming the benefit of SMT.

6 Decision Tree Pruning

Decision trees are commonly pruned to improve accuracy and to alleviate for the over fitting problem. Surveys and empirical comparisons of pruning strategies are reported in Breslow and Aha (1997); Esposito et al (1997); Quinlan (1999); Hall et al (2003). The C4.5 system adopts a post-processing *error-based pruning* (EBP) strategy, which, like the tree building algorithm, is a standard reference for novel proposals.Traditionally, efficiency of the pruning phase has not been a major concern, since pruning was by far computationally less expensive than tree building. Research in sequential optimisation and parallelisation of the building phase, however, has lead to the nowadays situation where the pruning time is in the order of magnitude of the building time[5]. A line of research has concentrated on the integration of the two phases, as in the PUBLIC system by Rastogi and Shim (2000), which during tree growing evaluates a sufficient condition (a lower

[5]The 7× speedup of the NAP strategy (Fig. 14) leads to an elapsed time for the building phase that is comparable to the time of the pruning phase (Table 2).

```
      void tree::prune() {
19.2    root->prune();
      }

      double node::prune() {
19.6   get_cases();
       double n_err = error_est();
19.8   int nChilds = nChilds();
       if( nChilds == 0 ) {
19.10    release_cases();
         set_err(n_err);
19.12    return n_err;
       }
19.14  double tree_err = 0;
       for(int i=0; i < nChilds; i++) {
19.16    node *child = getChild(i);
         tree_err += child->prune();
19.18  }
       node *large = getLargestChild();
19.20  double l_err = large->graft(n, true);
       release_cases();
19.22  if(n_err <= l_err
              && n_err <= tree_err) {
19.24    set_leaf();
         set_err(n_err);
19.26    return n_err;
       }
19.28 if( l_err <= tree_err) {
         replace_with_son(large);
19.30    return prune();
       }
19.32  set_err( tree_err );
       return tree_err;
19.34 }

19.36 double node::graft(node *father,
                         bool subst) {
19.38  node *n = clone();
       n->set_father(father);
19.40  if(subst) n->get_father_cases()
       else n->get_cases();
19.42  int nChilds = n->nChilds();
       double tree_err = 0;
19.44  if( nChilds == 0 )
         tree_err = error_est();
19.46  else
         for(int i=0; i < nChilds; i++) {
19.48      node *child = n->getChild(i);
           tree_err += child->graft(n, false);
19.50    }
       n->release_cases();
19.52  delete n;
       return tree_err;
19.54 }
```

Figure 19: Error-based pruning with grafting.

bound on the MDL pruning strategy) that allows for concluding that a subtree would be subsequently pruned. This line is orthogonal with the objective of this paper, which consists of devising a parallelisation of the pruning strategy. To the best of our knowledge, there is no previous attempt at this in the literature.

The YaDT implementation of the C4.5 pruning strategy is reported in Fig. 19. It consists of a bottom-up transversal of the decision tree implemented by the method `node::prune`. At each decision node, cases at the node are first retrieved from the father node by a call to `get_cases` (§19.6). Then a pessimistic estimates is calculated: (1) of the error[6] in case the node is turned into a leaf (§19.7); and (2) of the sum of errors of child nodes in case the node is left as a decision node (§19.14-18). For a leaf node, (1) is set as the error estimate (§19.9-12). For a decision node, if (1) is lower or equal than (2) then the node is turned into a leaf (§19.22-27). In addition, it is estimated also: (3) the error of *grafting* a child sub-tree in place of the node (§19.19-20). More in detail, given the child node with the maximum number of cases associated, which we call the *largest* child, (3) is calculated by "moving downwards" the cases associated to the parent node towards the largest child. If (3) is lower than (1) and (2) then the subtree rooted at the node is replaced by the subtree rooted at the largest child (§19.28-29) and the pruning procedure is repeated on such subtree (§19.30) — since the cases of the ex-parent node now distribute differently in the subtree. Cases at the node are released before returning from the visit of the node (§19.10,§19.21). Fig. 20 shows the effects of grafting on a sample tree: the

[6]Without entering the details of EBP, it is sufficient here to notice that the method `node::error_est` takes constant time, since it makes calculations on summarisation and statistics already computed in the `node::get_cases` method.

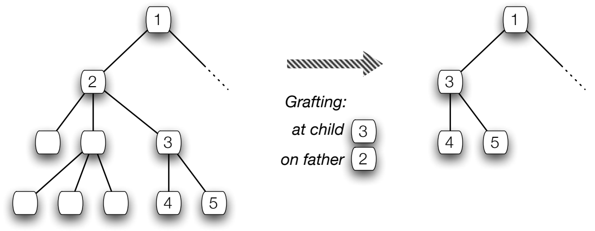


Figure 20: Example of grafting: father node 2 is replaced by its largest child 3.

child node 3 replaces its father node 2, and the subtrees rooted at the siblings of 3 are discarded.

As observed by Ruggieri (2004), grafting is time and memory consuming. YaDT offers an option whether to do or not to do grafting, with the result that performances are an order of magnitude better without it both in terms of time and memory, while classifier accuracy is slightly worse. Let us compare the two options on a ideal complete binary tree of depth d. Without grafting, EBP visits each node once, for a total of $V(d) = 2^d - 1$ nodes visited. With grafting, the number of node visited $V_g(d)$ is described by the recurrence relation:

$$V_g(1) = 1 \qquad V_g(d+1) = 2V_g(d) + 2^d$$

since at each node, its left and right children are visited as well as the whole subtree rooted at the largest child. The explicit solution of the relation is $V_g(d) = d2^{d-1}$, which is an order of magnitude higher than $V(d)$ in terms of the tree depth d.

Beyond this theoretical analysis, we observe that (1+2) requires a simple bottom-up transversal, where each node computes its error estimate and passes it upwards to the father node, whereas (1+2+3) requires for each node to compute, in addition, the error estimate of the sub-tree rooted at the largest child. Since the estimate is done with reference to the cases at the parent node, this requires to clone the data structures of the (largest!) subtree as it is being recursively visited in the `node::graft` procedure (§19.36-54). As a consequence, the additional visits of grafting are at higher running time and memory occupation costs.

7 Parallelising YaDT: Tree Pruning

In this section, we extend YaDT-FF by parallelising the pruning phase with a strategy that resembles the NP strategy of Sect. 4. Specifically, each decision node visited during pruning with grafting is considered a task that generates a set of sub-tasks; these tasks are arranged in a stream that flows across a *farm-with-feedback* skeleton. The setup method `tree::prune_ff` is shown in Fig. 21. Basically, it is the same code of the setup method in Fig. 5, apart from the

```
void tree::prune_ff() {
 E=new ff_prune_emitter(root,
        PAR_DEGREE);
 std::vector<ff_prune_worker*> w;
 for(int i=0;i<PAR_DEGREE;++i)
  w.push_back( new ff_prune_worker());
 ff_farm<ws_scheduler>
    farm(PAR_DEGREE*QSIZE);
 farm.add_workers(w);
 farm.add_emitter(E);
 farm.wrap_around();
 farm.run_and_wait_end();
}
```

Figure 21: Parallelisation of EBP with grafting: setup.

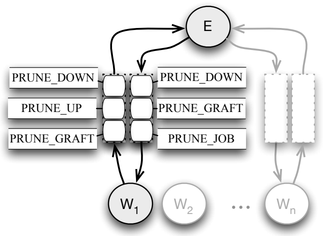


Figure 22: Parallelisation of EBP with grafting: task types.

creation of the root note (§5.2), since now the tree already exists. Also, notice that the WS policy (`ws_scheduler`) is adopted here as well.

Let us provide some intuitions on the parallelisation of the `tree::prune` method from Fig. 19. The visit of a node can be in one of the following states: (1) get cases at the node and compute EBP estimate (§19.6-13); (2) if the node is not a leaf, recursively visit its child nodes and sum their EBP estimates (§19.14-18); (3) recursively visit the subtree rooted at the largest child (§19.19-20); (4) release cases at the node, and test whether to transform the node into a leaf or to graft the largest child (§19.21-34). These states are handled by the emitter for the purpose of generating new tasks in the stream, and by workers for the purpose of performing computation. Tasks in the stream must then refer to the state of the visit at the node under consideration. This is somewhat similar to the use of `BUILD_NODE` and `BUILD_ATT` tags in the NAP strategy. Fig. 22 shows the tags of tasks flowing from the emitter towards the workers and vice-versa. For the emitter: `PRUNE_DOWN` means that the recursive visit at state (2) must be initiated for the child nodes; `PRUNE_UP` means that a child node has finished its recursive visit, and the parent node must decide whether to wait for other children, or to start (3) for the largest child; and `PRUNE_GRAFT` means that the recursive visit at state (3) must be initiated for the child nodes. For the workers: `PRUNE_DOWN` asks for performing the computation at state (1); `PRUNE_GRAFT` asks for performing the computation at state (3), namely (§19.38-45); and `PRUNE_JOB` asks for performing the computation at state (4).

Both the emitter and the workers source codes are then structured as a switch-case to perform actions on the basis of the task tag. The emitter code is reported in Fig. 23. At start-up the `ff_prune_emitter::svc` method is called by the FastFlow runtime with a NULL parameter (§23.2). A task for `PRUNE_DOWN` processing of the root node is built and its weight is set (§23.3-7). In general, the weight of a task will be fixed to the number of cases at the parent node. This is motivated by the fact that computation at a node basically consists of getting cases from the parent node, since, as noted in footnote 6, the calculation of error estimates takes constant time. Upon receiving in input a task coming from a worker, the emitter enters the switch-case on the basis of the task type.

```
       void * ff_prune_emitter::svc(void * task) {
23.2     if (task == NULL) {
          task=new ff_task(root,PRUNE_DOWN);
23.4      root->father = NULL;
          int r = root->getNoCases();
23.6      setWeight(task, r);
          return task;
23.8     }
         node *n = task->getNode();
23.10    int nChilds = n->Childs();
         switch( task->getType() ) {
23.12     case PRUNE_DOWN:
            n->prune_err = 0;
23.14       n->child_cnt = nChilds;
            int r = n->getNoCases();
23.16       for(int i=0; i < nChilds; i++) {
             node *child = n->getChild(i);
23.18        child->father = n;
             ctask=new ff_task(child,
23.20                           PRUNE_DOWN);
             setWeight(ctask, r);
23.22        ff_send_out(ctask);
            }
23.24       break;
          case PRUNE_UP:
23.26       node *fthr = n->father;
            if(fthr == NULL)
23.28        return NULL;
            fthr->prune_err += node->err;
23.30       if(--fthr->child_cnt > 0)
             return FF_GO_ON;
23.32       node *child = fthr->get_largest_child();
            task=new ff_task(child,
23.34                        PRUNE_GRAFT);
            fthr->cloned = fthr;
23.36       task->subst = true;
            task->subroot = fthr;
23.38       fthr->graft_err = 0;
            fthr->graft_cnt = 1;
23.40       fthr->child_cnt = 2;
            int r = fthr->getNoCases();
23.42       setWeight(task, r);
            ff_send_out(task);
23.44       break;
          case PRUNE_GRAFT:
23.46       node *subroot = task->subroot;
            node *fthr = task->father;
23.48       --subroot->graft_cnt;
            if(--fthr->child_cnt == 0) {
23.50        fthr->cloned->release_cases();
             delete fthr->cloned;
23.52       }
            if(nChilds == 0 || task->done) {
23.54        subroot->graft_err += n->graft_err;
             if(subroot->graft_cnt == 0) {
23.56         task=new ff_task(subroot,
                               PRUNE_JOB);
23.58         int r = subroot->getNoCases();
              setWeight(task, r);
23.60         return task;
             }
23.62       }
            subroot->graft_cnt += nChilds;
23.64       node *fthr = n->cloned;
            int r = fthr->getNoCases();
23.66       fthr->child_cnt = nChilds;
            for(int i=0; i < nChilds; i++) {
23.68        node *child = n->getChild(i);
             ctask=new ff_task(child,
23.70                          PRUNE_GRAFT);
             ctask->subst = false;
23.72        ctask->subroot = subroot;
             setWeight(ctask, r);
23.74        ff_send_out(ctask);
            }
23.76       break;
         }
23.78    return FF_GO_ON;
       }
```

Figure 23: Parallelisation of EBP with grafting: emitter.

In a `PRUNE_DOWN` task the emitter simply creates a task for each child (§23.12-24). To track the number of child tasks still to be processed, the counter `node::child_cnt`, introduced in the parallelisation of the building phase, is used. Additional members are added to the `node` structure and set: `prune_err` is used to accumulate the errors of child nodes; `father` is a pointer from a child back to its father node, needed in the `PRUNE_UP` state to reference fields in the father node.

In a `PRUNE_UP` task, we have the following sub-cases. If the node is the root (§23.26-28) we are finished. Otherwise, we accumulate the estimated error of the node in the `prune_err` field of the father node (§23.29). If the node is not the last processed child, the emitter simply returns, awaiting for other child tasks (§23.30-31). Otherwise, let `n` be the last processed child, and let us call the *top node* its father node. A task is build for initiating the recursive visit of the subtree rooted at the largest child of the top node (§23.32-43). This visit accumulates the estimates of nodes in the subtree into a field of the top node. The following members are added to the `node` structure for the purpose and set:

```
void * ff_prune_worker::svc(void * task) {
  node *n = task->getNode();
  int nChilds = n->Childs();
  switch( task->getType() ) {
   case PRUNE_JOB:
    double n_err = error_est();
    double tree_err = n->prune_err;
    double l_err = n->graft_err;
    if(n_err <= l_err
    && n_err <= tree_err) {
      n->release_cases();
      set_leaf();
      set_err(n_err);
      task->type = PRUNE_UP;
      return task;
    }
    if( l_err > tree_err) {
      n->release_cases();
      set_err ( tree_err );
      task->type = PRUNE_UP;
      return task;
    }
    node *large = n->getLargestChild();
    n->replace_with_son(large);
    nChilds = n->Childs();
    task->type = PRUNE_DOWN;
    // no break here
   case PRUNE_DOWN:
    node *fthr = n->father;
    if(fthr && fthr->getNoCases()
                        <MIN_PRUNE) {
      n->err = n->prune();
      n->set_father_recursive(fthr);
      task->type = PRUNE_UP;
      return task;
    }
    n->get_cases();
    if(nChilds == 0) {
      double n_err = n->error_est();
      n->release_cases();
      n->set_err(n_err);
      task->type = PRUNE_UP;
    }
    break;
   case PRUNE_GRAFT:
    node *fthr = n->father->cloned;
    task->done = false;
    if(fthr->getNoCases()
                  <MIN_GRAFT) {
      n->graft_err = n->graft(father,
                              task->subst);
      task->done = true;
      return task;
    }
    node *n2 = n->clone();
    n2->set_father(fthr);
    n->cloned = n2;
    if(task->subst)
    n2->get_father_cases();
    else n2->get_cases();
    if(nChilds == 0) {
      n->graft_err = n2->error_est();
      n2->release_cases();
      delete n2;
    }
    break;
  }
  return task;
}
```

Figure 24: Parallelisation of EBP with grafting: worker.

`graft_cnt` is a counter of the number of tasks still to be processed during the visit of the subtree; `graft_err` is the accumulator of the estimates. Also, the task structure has a member `subroot` to allow a node in the subtree to locate the top node in order to accumulate its error estimate. The task structure has a member `subst` to simulate the role of the `subst` parameter in the `node::graft` method from Fig. 19. Finally, since the visit builds a clone of the subtree, the member `cloned` is added to `node` to host a clone of the node. The top node provides the initial cases, and its field `cloned` is not needed to be a clone. The `node::child_cnt` member is used to set the number of child tasks still to be processed for a node. This is needed later on to release cases when all child nodes have retrieved their ones. Actually, since the subtree is cloned, the whole deletion of the cloned father is performed (§23.49-52). This must not occur for the top node, which is not cloned. We prevent this by setting `child_cnt` to 2 (§23.40).

In a `PRUNE_GRAFT` task, the top node that originated the subtree visit is referenced in the `subroot` variable (§23.46). Starting from it, the number of tasks still to be processed in the subtree can be updated (§23.48,§23.62). If the node is a leaf, or all its subtree has been processed by the worker, then in the if block (§23.55-61) the grafting error of the top node is updated and, if there are

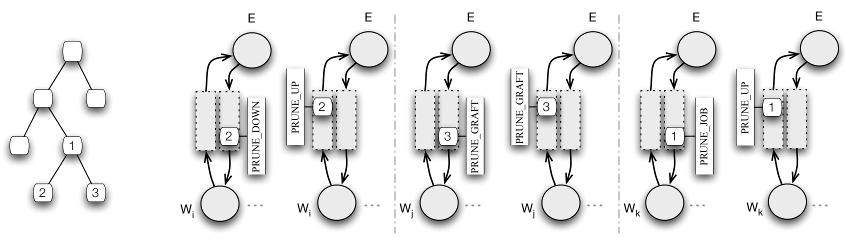


Figure 25: Example of parallelisation of the pruning phase.

no more task around, the grafting is finished and a task PRUNE_JOB is initiated for the top node. If the node is not a leaf, then the emitter creates a task for each child (§23.66-75). Notice that in this case, the subst member of the child task is set to **false** (§23.71).

The worker code is reported in Fig. 24. As for the emitter code, workers enter a switch-case on the basis of the input task type.

In a PRUNE_DOWN task, the worker has to calculate the estimated error. This can be done for the entire subtree, or for the single node only, on the basis of the number of cases at the father node. The parameter MIN_PRUNE sets the threshold for switching to sequential pruning (§24.30-36). Otherwise, the error estimate is calculated (§24.37-41) by cut & paste of the sequential code (§19.6-11). Notice that for leaf nodes (§24.42) and in case of sequential pruning (§24.34), the worker changes the task type to PRUNE_UP to denote that the visit of the subtree is finished.

Similarly, in a PRUNE_GRAFT task there is a parameter MIN_GRAFT for switching to sequential grafting (§24.48-54). In such a case, the done member of the task is set to **true** to let the emitter aware that the entire subtree has been processed. Otherwise, we proceed to clone the current node and to store a reference in the cloned member (§24.55-57). The rest of the code (§24.58-65) is cut & paste from the sequential method graft from Fig. 19.

In a PRUNE_JOB task, the following error estimates are available (§24.6-8): estimate of the node as a leaf; estimate of the whole subtree; and, estimate of grafting the largest child. We can now cut & paste sequential code for testing whether to turn the subtree into a leaf or whether to do grafting (§24.9-24). With respect to (§19.19-32), however, the second test is complemented. In fact, if grafting is performed, the recursive call (§19.30) has to be implemented by performing the worker actions of the PRUNE_DOWN case. This is the actual result of complementing the second test and not inserting a **break** statement (§24.27).

Fig. 25 shows example snapshots of a parallel execution. Starting from a PRUNE_DOWN task for node 1, the emitter yields PRUNE_DOWN tasks for its child nodes 2 and 3. Since both are leaves, workers eventually process them and return to the emitter a PRUNE_UP task (in the figure, this is shown for node 2). After receiving the task from the last child, the emitter queues a PRUNE_GRAFT task for the largest child, node 3 in the example. A worker processes the task

Table 3: Execution time (in seconds) varying MIN_PRUNE and MIN_GRAFT (Nehalem, 7 worker threads).

	MIN_PRUNE-MIN_GRAFT					
	100-100	1000-1000	1000-4000	4000-4000	16000-1000	16000-16000
Census-Income	0.22	**0.18**	0.23	0.32	0.23	0.23
U.S. Census	0.43	**0.32**	0.40	0.44	0.43	0.37
KDD Cup 99	3.64	**3.46**	3.62	3.50	3.58	3.55
Forest Cover	2.77	**2.14**	2.46	2.62	2.63	2.16
SyD10M9A	7.26	**5.95**	6.99	7.33	6.70	5.98

and returns it back to the emitter. Since node 3 is a leaf, the emitter generates a PRUNE_JOB task for the top node 1. The worker processing the task decides not to prune (making it a leaf, or grafting the largest child), and it returns a PRUNE_UP task towards the emitter.

8 Performance Evaluation II

In this section we discuss the performances obtained by YaDT-FF on the *pruning phase*. The experimental framework is the same described in Sect. 5.

Parallel performance. First, let us consider the setting of the thresholds MIN_PRUNE and MIN_GRAFT that control the switching from parallel to sequential execution. Differently from the parallelisation of the building phase, where switching to sequential execution was not contemplated, the task granularity can now reach very small values. In fact, the computation of a worker could simply consist of a call to get_cases, whose grain may be in the order of the overhead of the FastFlow runtime for task generation and scheduling. For this reason, we adopted a simple and effective strategy (previously used in Coppola and Vanneschi (2002) for the parallelisation of the build phase), which consists of setting machine-tailored thresholds for switching to sequential execution. Table 3 reports the execution times obtained using different threshold values for MIN_PRUNE and MIN_GRAFT. As it can be noticed, performances are not very sensitive for the architecture considered (Nehalem, 7 worker threads). Nevertheless, we have observed that the gap enlarges with the increasing of the parallelism . On the two experimented architectures (Nehalem and Magny-Cours), lower values show better performances, mainly because they lead to the generation of more tasks to be computed in parallel. However, there is a bottom line after which performances degrade due to excessive overhead. For all the following tests, we fixed both MIN_PRUNE and MIN_GRAFT to 1000.

Analogously to the performance analysis of the NP strategy, let us introduce now a lower bound for the elapsed time of the parallelisation of EBP pruning with grafting. Consider the sequential algorithm in Fig. 19. A lower bound for the parallelisation of the node::graft method is imposed by the path with the highest computational cost, since a node cannot return its error estimate until

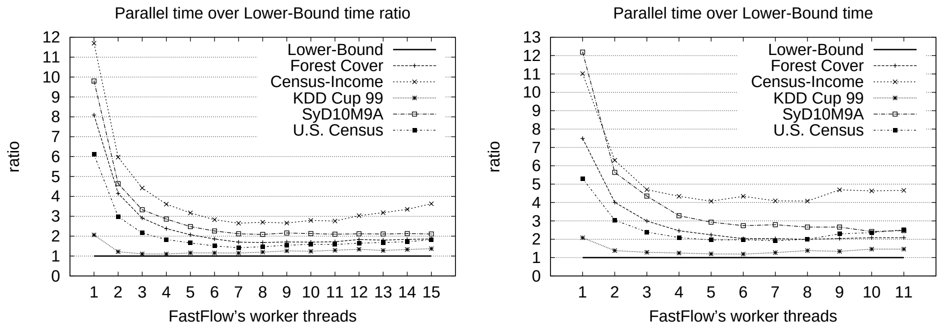


Figure 26: Elapsed over lower-bound time ratio of the tree pruning phase on the Nehalem box (left) and Magny-Cours box (right).

the last of its child nodes has returned. In symbol, we write:

$$lb_g(n) = t_g(n) + max_{m \in child(n)} lb_g(m)$$

where $t_g(n)$ is the time for sequential computations at node n (§19.38-44), and $child(n)$ is the set of child nodes of n. Similarly, a lower bound for the parallelisation of the `node::prune` method is imposed by the path with the highest computational cost, since a node cannot return its error estimate until the largest of its child nodes has returned and[7] the error estimate of grafting the largest child has been calculated. In symbol, we write:

$$lb_p(n) = t_p(n) + lb_g(largest(n)) + lb_p(largest(n))$$

where $t_p(n)$ is the time for sequential computations at node n (§19.6-14), and $largest(n)$ is the largest child of n. From the experimental side, we have instrumented the sequential code to compute $lb_p(root)$ for the root node of the tree to be pruned. From the theoretical side, since $t_g(n)$ and $t_p(n)$ are proportional to the number of cases at node n, the path with the highest computational cost turns out to be the largest path, as in the NP strategy.

The plots in Fig. 26 show, for the two reference architectures, the ratio between the elapsed time and the experimentally computed lower bound time for all the datasets. Apart from *Census-Income*, the elapsed time obtained by the parallelisation of the pruning phase is below or close to twice the lower-bound time. As observed for the NP strategy, such a lower bound is a strict limit, reachable only by an oracle scheduler. This confirms the effectiveness of the proposed parallelisation strategy.

The speedup obtained for the three synthetic datasets is shown in Fig. 27. We recall that they are similar apart from the unbalancing factor of the decision tree induced from them (see Table 1). As expected, the lower the unbalancing factor, the better the speedup.

[7]`node::graft` cannot be called before the largest child has returned, since during the visit of the largest child its subtree could be changed by grafting descendants.

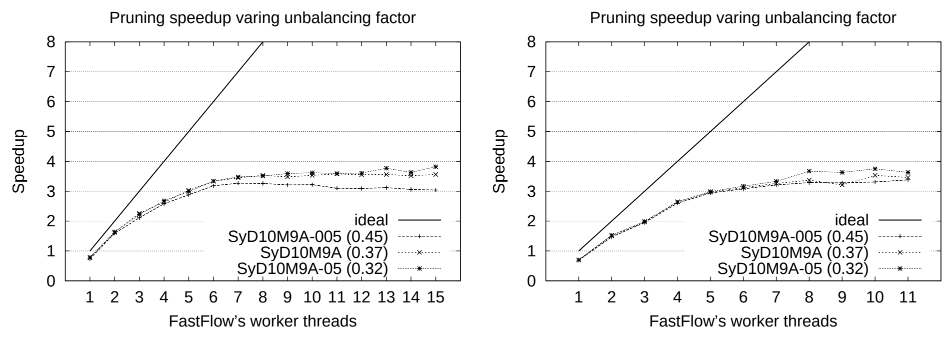


Figure 27: Speedup of the pruning phase for synthetic datasets with different unbalancing factors. Nehalem box (left), Magny-Cours box (right).

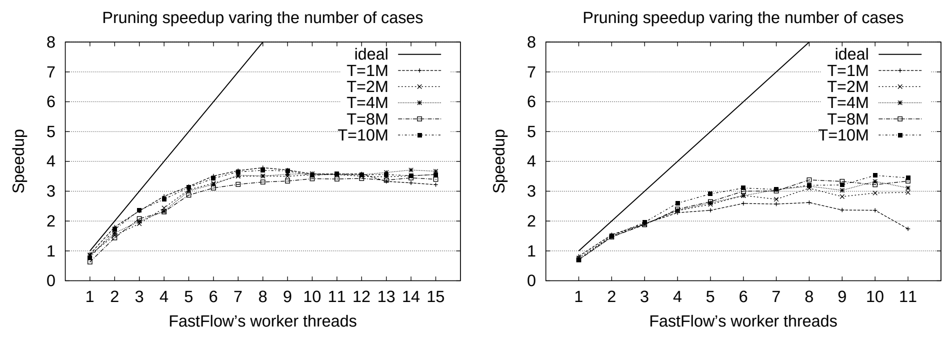


Figure 28: Pruning phase speedup vs no. of sample cases (T) from *SyD10M9A*. Nehalem box (left), Magny-Cours box (right).

Let us consider now scalability with respect to the number of cases. In order to plan a controlled experiment, we cannot resort to building and then pruning decision trees on a variable number of cases, as done in Fig. 16, since the decision trees in input to the pruning phase would be of different sizes and balancing factors. Rather, we first fix the decision tree to be pruned, by growing a tree from the whole *SyD10M9A* dataset, and then we prune it by starting from a subset of cases at the root node. This is a controlled experiment, where one input (the decision tree to be pruned) is fixed, and the other (the number of cases) is varied. Fig. 28 shows that the speedup slightly increases with the number of cases. Due to the lower-bound on parallel execution $lb(root)$, however, speedups are upper-bounded, and the maximum speedup is reached with about 8 workers.

Finally, we point out that the performances of the pruning phase are not affected by the number of attributes, since there is no attribute-dependent computation (see footnote 6). Therefore, it is not worth studying scalability w.r.t. the number of attributes, as done in the building phase (see Fig. 15).

Task scheduling. We tested the same scheduling policies as in the build

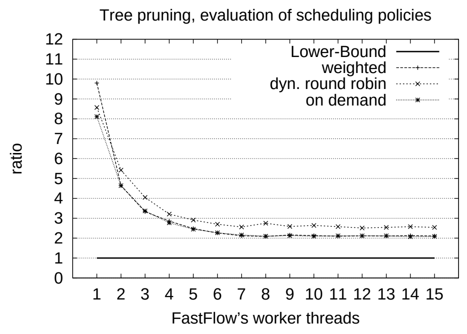


Figure 29: Elapsed over lower-bound time ratio of different scheduling policies over *SyD10M9A* (Nehalem).

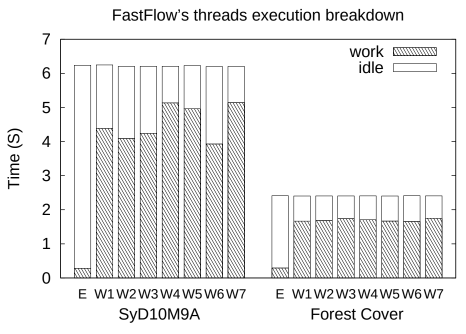


Figure 30: YaDT-FF prune phase execution breakdown on emitter (E) and 7 worker threads (Nehalem).

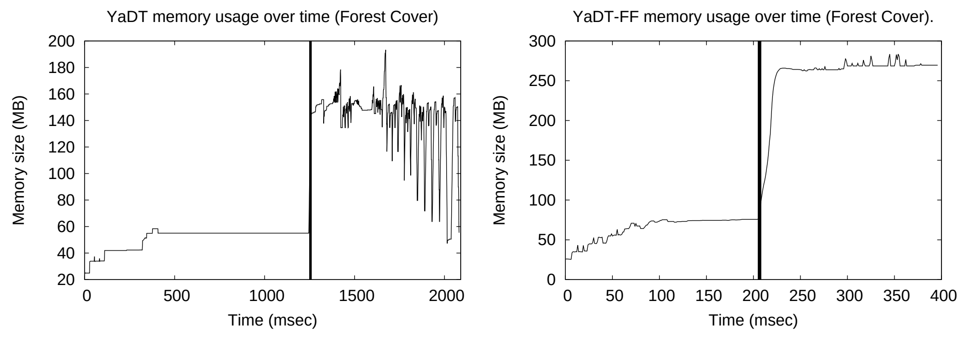


Figure 31: Memory usage over time on Nehalem box (using 7 workers) for YaDT (left) vs YaDT-FF (right) for the *Forest Cover* dataset. The vertical line denotes the end of the building phase and the beginning of the pruning phase.

phase: 1) Dynamic Round-Robin; 2) On-Demand; 3) Weighted Scheduling. Fig. 29 shows the ratio of the elapsed time over the lower-bound time for the *SyD10M9A* dataset. All policies are quite efficient. Here again, the weighted policy exhibits the best performance.

Although the intrinsically unbalanced nature of the computation in the pruning phase puts an upper-bound on the maximum attainable speedup, load balancing is not, instead, a critical issue. Fig. 30 shows the thread execution breakdown for *SyD10M9A* and *Forest Cover*.

Memory occupation. Fig. 31 reports memory occupation over time for YaDT (left) and YaDT-FF (right) for both the building and pruning phases over the *Forest Cover* dataset. Memory footprint has been traced reading Linux's `/proc/`*pid*`/statm` memory statistic file every 10ms. In the case of sequential execution, it is interesting to point out that the pruning phase requires considerable additional memory with respect to the building phase, and that there is an high number of interleaved allocation and deallocation operations due to the depth-first visit of the tree. The same proportion in the memory occupation applies also to the parallel execution, showing that the proposed parallelisation strategy is able to obtain close to optimal execution time without using a significant amount of additional memory. This result can be explained by the parallelisation of the `node::grafting` method, the one that clones subtrees. Consider the `PRUNE_GRAFT` case in the emitter (Fig. 23). The counter `child_cnt` is used to deallocate a cloned node once that all of its child nodes have retrieved their cases (§23.49-51). Since cases at the father node are partitioned among child nodes, the overall number of cases at the nodes in the task queues is at most twice the total number of cases (this does not consider cases with unknown value of the tested attribute, which are replicated among child nodes). Therefore, memory occupation is upper-bounded and independent from the degree of parallelism. The same reason also justifies the more stable memory usage of the parallel version.

9 Related Work

We first review the literature on parallelisation of the decision tree *building* phase, commenting on how existing proposals relate to our approach. As already observed, to the best of our knowledge, there is no related work on the parallelisation of the *pruning* phase. Next, we review existing (pattern-based) parallel programming frameworks, and relate them to the FastFlow framework.

9.1 Parallelising Decision Tree Building

Parallel computing research aimed at addressing three main classes of data mining issues: 1) solve inherently distributed problems, e.g., mining datasets that are bound to specific sites due to privacy issues; 2) manage larger datasets by exploiting the aggregate memories of several machines; 3) decrease the processing time of algorithms. We observe that (1) hardly promotes the reuse of existing sequential code and it often requires the redesign of sequential algorithms. Concerning (2), the ever-increasing availability of main memory space is making this class of approaches loosing interest. Finally, (3) has to face with the current state of the art of hardware architecture, namely multicores, and, at the same time, with the challenging problem of code reuse. Our approach clearly belongs to this class. According to the parallelisation paradigms presented in Sect. 2.2, related work can be categorised as follows.

Task Parallelism. It consists of splitting the processing of different subtrees into independent tasks in a Divide&Conquer fashion. The NP strategy presented in Sect. 4 fully adheres to this approach, despite task parallelism is realised via stream parallelism. In distributed implementations, the approach is also referred to as *partitioned tree construction* (Srivastava et al, 1999), since tree construction consists of dynamically distributing the decision nodes among the processors for further expansion. The approach suffers from load balancing problems due to the possible different sizes of the trees constructed by each processor. In distributed implementations, this is a not trivial issue since load balancing strategies are typically communication intensive and complex to be integrated in the code. As example, in the early work by Darlington et al (1997) the whole training set is replicated in the memory of all the processors, in order to avoid communication costs due to moving cases across processing nodes. In Coppola and Vanneschi (2002), a *distributed shared tree* data container is introduced to reduce the coding complexity, while the Divide&Conquer paradigm is realised with a *farm-with-feedback* skeleton, as in the present work. The distributed implementation requires a non trivial porting and tuning effort in order to find a suitable trade-off between load balancing and computation-to-communication ratio. Narlikar (1998) proposed a Pthread-based Divide&Conquer parallelisation on a shared-memory architecture. The approach differs from YaDT-FF in using a parallel quicksort, in relying on the dynamic creation of a large number of concurrent threads (which might seriously impair the run-time efficiency — the implementation is actually tested on a single synthetic dataset), and in the hand-made code porting and tuning process. As a general advancement over related work, we have characterised a lower bound for any task parallelisation strategy, and shown that the NP strategy reaches performances very close to the lower bound.

Data Parallelism. It consists in distributing the training set among the processors by partitioning attributes or cases. The two options are referred to as *vertical* or *horizontal* data partitioning (Amado et al, 2001).

In vertical data partitioning, each processor computes the gain calculations for a subset of the attributes, e.g., in distributed implementation, for the attributes assigned at the processor node (Freitas and Lavington, 1997). Gain calculations are then exchanged between nodes to determine the best split. This solution suffers both from load balancing problems, since the cost of gain calculation is not uniform across (discrete vs continuous) attributes, and, for distributed implementations, from high communication costs. The NAP strategy in our approach adopts a similar method for gain calculations. As already observed, however, load balancing is tackled by switching to the NP strategy on the basis of a cost model, communication costs are negligible in a shared memory environment, and synchronisation costs are minimum due to the design of the FastFlow framework.

In horizontal data partitioning, cases are evenly distributed among the processing nodes. Each processor computes the aggregate values (of its cases) needed for information gain calculation, and it exchanges them with the other nodes to determine the best split. This solution suffers from a heavy re-coding of the node splitting procedure (see Fig. 2), and, for distributed implementations, from high communication costs. Horizontal data distribution is exploited in the SPRINT classifier (Shafer et al, 1996), which stores the training set according to the SLIQ layout (Mehta et al, 1996). The authors show a superior performance with respect to vertical data distributions in the direct parallelisation of the SLIQ sequential classifier. ScalParC (Joshi et al, 1998) improves on SPRINT by adopting a distributed hash table that mitigates the communication cost problems. Concerning shared-memory machines, Zaki et al (1999) propose a porting of SPRINT to several data parallel versions, from BASIC to the Moving-Windows-K (MWK) algorithm, exhibiting progressively weaker coupling: a global barrier for BASIC and a conditional variable per node for MWK. They also propose a hybrid version. All versions are implemented on top of a master-worker infrastructure, that is similar to one used in YaDT-FF, thus also exploiting pipeline parallelism between the master and workers. However, synchronisations among master and workers happen via Pthread mutexes and conditional variables, and therefore exhibit low scalability for fine grained tasks. The work also highlights the limits of mutual exclusion as synchronisation mechanism in this class of algorithms. A variant of ScalParC for shared memory ccNUMA systems is proposed by Bradford and Fortes (2001), together with a deep investigation of the impact of data locality and cache misses. The RainForest sequential algorithm (Gehrke et al, 2000), which is adopted as one of the strategies of YaDT for handling continuous attributes, has been parallelised by Jin and Agrawal (2003). The approach builds a decision tree by levels, with an horizontal data partitioning for computing aggregate values needed in the gain calculation. Synchronisation occurs in order to sum up the partial aggregates and decide the best split. Finally, an adaption of horizontal data partitioning to streaming datasets has been recently proposed in Ben-Haim and Tom-Tov (2010).

Hybrid Task and Data Parallelism. The hybrid task and data parallelism has been explored as a means to control the communication overhead. In the *hybrid parallel formulation* (Srivastava et al, 1999) and in pCLOUDS (Sreenivas et al, 1999), a data parallel approach is used for the top levels of the tree, i.e., when the grain of decision node computation is large, and a task parallel approach for the low levels. Coppola and Vanneschi (2002) classify tasks as large, intermediate or small. Large tasks process a single decision node. Intermediate tasks process a sub-tree up to a maximum number of decision nodes. Small tasks sequentially process the whole sub-tree of a decision node. YaDT-FF, and in particular the NAP strategy, takes inspiration from the two latter works and distinguish from them since: it does not need the redesign of the sequential algorithm but rather an *easy-yet-efficient* porting of the existing code; it targets multicore rather than distributed memory machines; it adopts an effective cost model for deciding whether to parallelise on nodes (task parallelism) or on attributes (data parallelism); the two parallel approaches are not used in successive phases as in mentioned works but they are temporally inter-waved and executed on the same stream parallel infrastructure.

Finally, we mention that the interest of the data mining community on algorithms for multicore platforms is blooming, see e.g., Choudhary et al (2011); Chu et al (2006); Jin et al (2005); Buehrer (2008). The general problem of designing advanced locking schemas for decreasing the synchronisation overhead in shared memory machines has been considered by Jin et al (2005) and by Ravi and Agrawal (2009). YaDT-FF overcomes the problem via the FastFlow lock-free synchronisation mechanisms: all synchronisations in the decision tree building and pruning phases happen asynchronously in the emitter as the result of data-dependencies among tasks.

9.2 (Pattern-based) Parallel Programming Frameworks

Early proposals of pattern-based parallel programming frameworks have been mainly focused on distributed memory platforms, such as clusters of workstations and grids. We mention ASSIST (Vanneschi, 2002), Münster skeleton library (Kuchen, 2002), and Google MapReduce (Dean and Ghemawat, 2008), which brings to mainstream of out-of-core data processing the map-reduce paradigm. All these skeleton frameworks provide several parallel patterns (algorithmic skeletons) covering mostly task and data parallelism. These patterns can usually be nested to model more complex parallelism exploitation patterns according to the constraints imposed by the specific programming framework. More recent pattern-based frameworks, following the platform architecture trend, have shifted the focus on multicores and shared address model; in addition to FastFlow, it is worth to mention the Intel Threading Building Block (TBB) library (Intel Corp., 2011), and to a limited extent the Microsoft Task Parallel Library (Leijen and Hall, 2007). All of them are definitely higher-level compared with the Pthread library that has been used in the shared memory implementations of classification algorithms previously mentioned. The main features of these frameworks, as well as many other experimental ones, are surveyed in González-Vélez and Leyton (2010).

In addition to pattern-based frameworks, other high-level programming frameworks also aim at simplifying the design of efficient applications on multi-cores and thus are related with FastFlow and to the present work. StreamIt (Thies et al, 2002) is an explicitly parallel programming language based on the Synchronous Data Flow model that enables the assembly of program modules (called filters) in a *pipeline* fashion, possibly with a *FeedbackLoop*, or according to a *SplitJoin* data-parallel schema. Streaming applications are also targeted by TBB through the *pipeline* construct, which also provides programmers with thread-safe containers and some parallel patterns (called "algorithms"); TBB does not support any kind of non-linear streaming network, which therefore has to be embedded in a pipeline with significant programming and performance drawbacks. Intel's Concurrent Collections (CnC), which declaratively models concurrent activities as data streams and control dependencies, has been recently proposed as candidate substrate for parallel patterns (Newton et al, 2010). *OpenMP* (Park et al, 2001) is a popular thread-based framework for multi-core architectures mostly targeting data parallel programming even if it is currently being extended to incorporate stream processing. OpenMP supports, by way of language pragmas, the low-effort parallelisation of the sequential programs; however, these pragmas are mainly designed to exploit loop-level data parallelism (e.g. *do_independent*). CnC and OpenMP do not natively support neither *farm* nor *Divide&Conquer* patterns, even tough they can be simulated with lower-level features.

		1E+1W	1E+2W	1E+3W	
$\mathcal{T}$ name	Seq.Time (S)	Time (S)			Max Boost
Census-Income	3.95	4.01	2.1	1.45	2.72×
U.S. Census	13.82	18.67	8.40	5.83	2.37×
KDD Cup 99	21.24	23.20	11.83	9.20	2.31×
Forest Cover	22.50	25.26	12.50	8.37	2.69×
SyD10M9A	135.27	123.20	57.69	41.66	3.24×

Table 4: YaDT vs YaDT-FF on a Nehalem quad-core. Build time plus EBP pruning with grafting time is considered (E=Emitter, W=Worker).

10 Conclusions

Nowadays, and for foreseeable future, the performance improvement of a single core will no longer satisfy the ever increasing computing power demand. For this reason, the computer hardware industry shifted to multicore, and thus the extreme optimisation of sequential algorithms is no more sufficient to squeeze the real machine power. Software designers are then required to develop and to port applications on multicore. In this paper, we have presented the case study of decision tree building and pruning algorithms, porting YaDT to multicores using the FastFlow parallel programming framework. The strength of our approach consists in the minimal change of the original code with, at the same time, non-trivial parallelisation strategies. We have characterised lower bounds for the forms of parallelisations adopted, and obtained performances close to such bounds. This confirms the effectiveness both of our design and of the underlying FastFlow layer. Parallelisation of the pruning phase is a totally novel contribution of this paper. Eventually, we want to stress the results in the case of a low cost quad-core architecture that may be currently present in the desktop PC of any data analyst. Table 4 shows that the parallelisation of YaDT, including both building and pruning, boosts up in the $2.3 - 3.2\times$ range, with no additional cost for a dedicated parallel hardware.

Acknowledgements

We thank the Competence Center Gateway for HPC of the IT Center, University of Pisa, for the use of the Magny-Cours box.

References

Aldinucci M, Torquati M (2011) FastFlow website. `http://mc-fastflow.sourceforge.net/`

Aldinucci M, Meneghin M, Torquati M (2010a) Efficient Smith-Waterman on multi-core with FastFlow. In: Danelutto M, Gross T, Bourgeois J (eds) Proc. of Intl. Euromicro Conf. on Parallel Distributed and Network-based Processing (PDP 2010), IEEE, pp 195–199

Aldinucci M, Ruggieri S, Torquati M (2010b) Porting decision tree algorithms to multicore using FastFlow. In: Proc. of European Conf. on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD 2010), Springer, LNCS, vol 6321, pp 7–23

Amado N, Gama J, Silva F (2001) Parallel implementation of decision tree learning algorithms. In: Brazdil P, Jorge A (eds) Progress in Artificial Intelligence, LNCS, vol 2258, Springer, pp 34–52

Asanovic K, Bodik R, Demmel J, Keaveny T, Keutzer K, Kubiatowicz J, Morgan N, Patterson D, Sen K, Wawrzynek J, Wessel D, Yelick K (2009) A view of the parallel computing landscape. CACM 52(10):56–67

Ben-Haim Y, Tom-Tov E (2010) A streaming parallel decision tree algorithm. Journal of Machine Learning Research 11:849–872

Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y (1996) Cilk: An efficient multithreaded runtime system. Journal of Parallel and Distributed Computing 37(1):55–69

Bradford JP, Fortes JAB (2001) Characterization and parallelization of decision-tree induction. Journal of Parallel and Distributed Computing 61(3):322–349

Breslow LA, Aha DW (1997) Simplifying decision trees: A survey. The Knowledge Engineering Review 12:1–40

Buehrer GT (2008) Scalable mining on emerging architectures. Phd thesis, Columbus, OH, USA

Choudhary AN, Kumar P, B BO, Misra S, Memik G (2011) Accelerating data mining workloads: Current approaches and future challenges in system architecture design. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 1:41–54

Chu CT, Kim SK, Lin YA, Yu Y, Bradski GR, Ng AY, Olukotun K (2006) Map-reduce for machine learning on multicore. In: Proc. of the Annual Conference on Neural Information Processing Systems (NIPS 2006), MIT Press, pp 281–288

Cole M (1989) Algorithmic Skeletons: Structured Management of Parallel Computations. Research Monographs in Parallel and Distributed Computing, Pitman

Coppola M, Vanneschi M (2002) High-performance data mining with skeleton-based structured parallel programming. Parallel Computing 28(5):793–813

Darlington J, Guo Y, Sutiwaraphun J, To HW (1997) Parallel induction algorithms for data mining. In: Proc. of 2nd Intl. Symposium on Advances in Intelligent Data Analysis, Reasoning about Data (IDA), Springer, LNCS, vol 1280, pp 437–445

Dean J, Ghemawat S (2008) MapReduce: Simplified data processing on large clusters. CACM 51(1):107–113

Esposito F, Malerba D, Semeraro G (1997) A comparative analysis of methods for pruning decision trees. IEEE Transactions on Pattern Analysis and Machine Intelligence 19(5):476–491

Fayyad UM, Irani KB (1992) On the handling of continuous-valued attributes in decision tree generation. Machine Learning 8:87–102

Frank A, Asuncion A (2011) UCI machine learning repository. URL `http://archive.ics.uci.edu/ml`

Freitas AA, Lavington SH (1997) Mining Very Large Databases with Parallel Processing, 1st edn. Kluwer Academic Publishers, Norwell, MA, USA

Gehrke JE, Ramakrishnan R, Ganti V (2000) RainForest — A framework for fast decision tree construction of large datasets. Data Mining and Knowledge Discovery 4(2/4):127–162

González-Vélez H, Leyton M (2010) A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. Software: Practice and Experience 40(12):1135–1160

Hadoop (2011) Web Page. Apache Software Foundation, URL `http://hadoop.apache.org/`

Hall LO, Bowyer KW, Banfield RE, Eschrich S, Collins R (2003) Is error-based pruning redeemable? International Journal on Artificial Intelligence Tools 12(3):249–264

IBM Almaden (2003) Quest synthetic data generation code. URL `http://www.almaden.ibm.com/software/quest`

Intel Corp (2011) Threading Building Blocks. URL `http://www.threadingbuildingblocks.org/`

Jin R, Agrawal G (2003) Communication and memory efficient parallel decision tree construction. In: Proc. of SIAM Intl. Conf. on Data Mining (SDM 2003), SIAM, pp 119–129

Jin R, Yang G, Agrawal G (2005) Shared memory parallelization of data mining algorithms: Techniques, programming interface, and performance. IEEE Transactions on Knowledge and Data Engineering 17:71–89

Joshi M, Karypis G, Kumar V (1998) ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. In: Proc. of Intl. Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP 1998), IEEE, pp 573–579

Kahn G (1974) The semantics of a simple language for parallel programming. In: Rosenfeld JL (ed) Information processing, North Holland, Amsterdam, Stockholm, Sweden, pp 471–475

Kargupta H (2011) Distributed data mining bibliography. URL `http://www.csee.umbc.edu/~hillol/DDMBIB`

Kuchen H (2002) A skeleton library. In: Monien B, Feldman R (eds) Proc. of Intl. Euro-Par Conference (Euro-Par 2002), Springer, Paderborn, Germany, LNCS, vol 2400, pp 620–629

Lee E, Parks T (1995) Dataflow process networks. Proc of the IEEE 83(5):773–801

Leijen D, Hall J (2007) Optimize managed code for multi-core machines. MSDN Magazine URL `http://msdn.microsoft.com/en-us/magazine/cc163340.aspx`

Lim T, Loh W, Shih Y (2000) A comparison of prediction accuracy, complexity, and training time of thirthy-tree old and new classification algorithms. Machine Learning Journal 40:203–228

Mehta M, Agrawal R, Rissanen J (1996) SLIQ: A fast scalable classifier for data mining. In: Proc. of the Intl. Conference on Extending Database Technology (EDBT 1996), Springer, LNCS, vol 1057

Narlikar GJ (1998) A parallel, multithreaded decision tree builder. Tech. Rep. CMU-CS-98-184, Computer Science Dept., Carnegie Mellon University

Newton R, Schlimbach F, Hampton M, Knobe K (2010) Capturing and composing parallel patterns with Intel CnC. In: Proc. of USENIX Workshop on Hot Topics in Parallelism (HotPar 2010), Berkley, CA, USA

Pacheco PS (1996) Parallel programming with MPI. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

Park I, Voss MJ, Kim SW, Eigenmann R (2001) Parallel programming environment for OpenMP. Scientific Programming 9:143–161

Quinlan JR (1993) C4.5: Programs for Machine Learning. Morgan Kaufmann, San Mateo, CA

Quinlan JR (1999) Simplifying decision trees. International Journal of Human-Computer Studies 51(2):497–510

Rastogi R, Shim K (2000) PUBLIC: A decision tree classifier that integrates building and pruning. Data Mining & Knowledge Discovery 4(4):315–344

Ravi VT, Agrawal G (2009) Performance issues in parallelizing data-intensive applications on a multi-core cluster. In: Proc. of IEEE/ACM Intl. Symposium on Cluster Computing and the Grid (CCGRID 2009), IEEE Computer Society, pp 308–315

Reed D (2009) High-Performance Computing: Where'd The Abstractions Go? BLOG@CACM

Ruggieri S (2002) Efficient C4.5. IEEE Transactions on Knowledge and Data Engineering 14:438–444

Ruggieri S (2004) YaDT: Yet another Decision tree Builder. In: Proc. of Intl. Conf. on Tools with Artificial Intelligence (ICTAI 2004), IEEE, pp 260–265

Shafer JC, Agrawal R, Mehta M (1996) SPRINT: A scalable parallel classifier for data mining. In: Proc. of Intl. Conf. on Very Large Databases (VLDB 1996), pp 544–555

Sodan AC, Machina J, Deshmeh A, Macnaughton K, Esbaugh B (2010) Parallelism via multithreaded and multicore CPUs. IEEE Computer 43(3):24–32

Sreenivas MK, Alsabti K, Ranka S (1999) Parallel out-of-core divide-and-conquer techniques with application to classification trees. In: Proc. of Intl. Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing, IEEE, pp 555–562

Srivastava A, Han EH, Kumar V, Singh V (1999) Parallel formulations of decision-tree classification algorithms. Data Minining & Knowledge Discovery 3(3):237–261

Thies W, Karczmarek M, Amarasinghe SP (2002) StreamIt: A language for streaming applications. In: Proc. of Intl. Conf. on Compiler Construction (CC 2002), London, UK, pp 179–196

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

Wu X, Kumar V, Quinlan JR, Ghosh J, Yang Q, Motoda J, McLachlan GJ, Ng AFM, Liu B, Yu PS, Zhou ZH, Steinbach M, Hand DJ, Steinberg D (2008) Top 10 algorithms in data mining. Knowledge and Information Systems 14(1):1–37

Zaki M, Ho CT, Agrawal R (1999) Parallel classification for data mining on shared-memory multiprocessors. In: Proc. of Intl. Conf. on Data Engineering (ICDE 1999), IEEE, pp 198–205