Università di Pisa
Dipartimento di Informatica

Technical Report: TR-10-11

Porting Decision Tree Algorithms to Multicore using FastFlow

Marco Aldinucci Salvatore Ruggieri Massimo Torquati

May 18th, 2010

Address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

Porting Decision Tree Algorithms to Multicore using FastFlow

Marco Aldinucci* Salvatore Ruggieri Massimo Torquati

May 18th, 2010

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 parallel porting requires minimal changes to the original sequential code, and it is able to exploit up to 7× speedup on an Intel dual-quad core machine.

Keywords Parallel classification, C4.5, multicores, structured parallel programming, streaming.

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 aimed at designing languages and tools to support the seamless porting of applications and the tuning of performances [13, 3, 21, 20]. These languages, apart from few exceptions that also focus on code portability [13, 21], 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

*Computer Science Department, University of Torino, Italy. Email: adinuc@di.unito.it

not an exception in this regard. By surveying the papers in the main scientific conferences and journals, there is a diminishing number of proposals for parallel implementations of data mining algorithms in the last few years. After all, only a small percentage 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 memory and 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. 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 engaged this challenge trying to 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 [13, 3, 21, 2].

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 [15] onto multicore machines. We consider the YaDT (Yet another Decision Tree builder) [17] implementation of C4.5, 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 [16], 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, which makes it unpractical to design further optimisations. 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 [1], 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 (D&C) computations. Besides technical features, FastFlow offers an important methodological approach that will lead us to parallelise YaDT with minimal

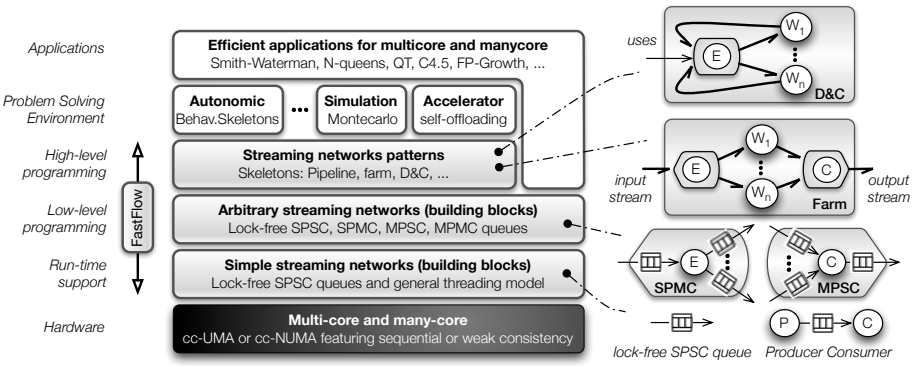

Figure 1: FastFlow layered architecture with pattern examples.

changes to the original sequential code, yet achieving up to 7× boost in performance on a Intel dual-quad core. 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.

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 construction algorithm, including the main optimisations that lead to YaDT. Then the parallelisation of YaDT is presented in detail in Sect. 4, followed by experimental evaluation and discussion in Sect. 5. Finally, we report related works in Sect. 6, and summarise the contribution of the paper in the conclusions.

2 The FastFlow Parallel Programming Environment

FastFlow is a parallel programming framework aiming to *simplify* the development of *efficient* applications for multicore platforms, being these applications either brand new or ports of existing legacy codes. The key vision underneath FastFlow is that effortless development and efficiency can be both achieved by raising the level of abstraction application design, thus providing designers with a suitable set of parallel programming patterns that can be compiled onto efficient networks of parallel activities on the target platforms. To fill the abstraction gap, 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 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] and thus they constitute a 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.

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 (a.k.a. *skeletons* [5]). They are usually categorised in three main classes: Task, Data, and Stream Parallelism. FastFlow specifically focuses on Stream Parallelism, and in particular provides: *farm*, *farm-with-feedback* (i.e. Divide&Conquer), *pipeline*, and their arbitrary nesting and composition. These high-level skeletons are actually factories for parametric patterns of concurrent activities, which can be instantiated with sequential code (within white circles in Fig. 1) or other skeletons, then cross-optimised and compiled together with lower FastFlow tiers. 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.

We refer to [1] for implementation details. FastFlow is open source available at `http://sourceforge.net/projects/mc-fastflow/` under LGPLv3 license.

3 Decision Trees: From C4.5 to YaDT

A decision tree is a classifier induced by supervised learning from a relation $\mathcal{T}$ called the *training set*. Tuples in $\mathcal{T}$ are called *cases*. An attribute C of the relation is called the *class*, while the remaining ones $A_1, \ldots, A_m$ are called the *predictive attributes*. 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 tree data structure 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.

[1] for Total Store Order processors, such as Intel core, AMD 10.

3.1 The C4.5 Tree-Induction Algorithm

The C4.5 decision tree induction algorithm [15] is a constant reference in the development and analysis of novel proposals of classification models [12]. The core[2] algorithm constructs the decision tree top-down. 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 $\mathcal{T}$. At each node a D&C 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 the 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 value 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 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, cases in T with known value for A are first ordered 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 set to be 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 attribute A. When A is continuous two child nodes are created, and cases from T with known value of A are partitioned accordingly to the boolean result of the test $A \leq t$, where t is the threshold of the split. Cases in T whose value for attribute A is unknown are added to the set of cases of every child, but their weights are rebalanced.

[2] In this paper, we concentrate on the *growth* phase of the algorithm. The subsequent *prune* phase is computationally less expensive.

[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.

```
    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.

3.2 From C4.5 to YaDT

The original Quinlan's implementation of C4.5 maintains the training set as an array of cases. Each case is 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 number of cases in the whole training set. This linear search prevents the implementation being truly a D&C computation.

Efficient C4.5 (EC4.5) [16] 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 accordingly 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 continuous attributes ranges tend to be narrow; the third strategy calculates the local threshold using a main-memory version of the RainForest [7] algorithm, without any sorting.

YaDT [17] is a from scratch C++ implementation of C4.5. It inherits the optimisations of EC4.5, and adds further ones, e.g., searching the local threshold for continuous attributes by considering splittings at *boundary* values (Fayyad and Irani method). Concerning data structures, the training set is now stored

```
    void tree::build() {
4.2   queue<node *> q;
      node *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    q.push( n->getChild(i) );
      }
4.12 }
```

Figure 4: YaDT tree growing procedure.

```
    void tree:: build_ff () {
5.2   node *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 D&C setup.

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

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

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

by columns, not by rows, since most of the computations scan data by attribute values. Most importantly, the object oriented design of YaDT allows 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 main memory occupation. Its pseudo-code is shown in Fig. 4 as method `tree::build`. Experiments from [16, 17] show that YaDT reaches up to 10× improvement over C4.5 with only 1/3 of its memory occupation.

4 Parallelising YaDT

We propose a parallelisation of YaDT, called YaDT-FF, obtained by stream parallelism. 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 D&C paradigm. The FastFlow D&C 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

```
void * ff_emitter :: svc(void * task) {
7.2  if (task == NULL ) {
      if (root->splitPre()) return NULL;
7.4   int r = root->getNoCases();
      int c = root->getNoAtts();
7.6   for(int i=0;i<c;++i) {
       task=new ff_task(root,BUILD_ATT);
7.8    task->att = i;
       setWeight(task, r);
7.10   ff_send_out(task);
      }
7.12  root->attTasks = c;
      return FF_GO_ON;
7.14 }
     node *n = task->getNode();
7.16 if (task->isBuildAtt()) {
      if (--n->attTasks>0)
7.18   return FF_GO_ON;
      n->splitPost();
7.20 }
     nChilds = n->Childs();
7.22 if (noMoreTasks() && !nChilds)
      return NULL;
7.24 for(int i=0; i < nChilds; i++) {
      node *child = n->getChild(i);
7.26  int r = child->getNoCases();
      int c = child->getNoAtts();
7.28  if (!buildAttTest(r,c)) {
       ctask=new ff_task(child,BUILD_NODE);
7.30   setWeight(ctask, r);
       ff_send_out(ctask);
7.32  } else {
       if (child->splitPre()) continue;
7.34   for(int j=0;j<c;++j) {
        ctask=new ff_task(child,BUILD_ATT);
7.36    ctask->att = j;
        setWeight(ctask, r);
7.38    ff_send_out(ctask);
       }
7.40   child->attTasks = c;
      }
7.42  return FF_GO_ON;
     }
7.44
     void * ff_worker :: svc(void * task) {
7.46  node *n = task->getNode();
      if (task->isBuildAtt())
7.48   n->splitAtt(task->att);
      else n->split();
7.50  return task;
     }
```

Figure 7: Emitter and Worker definition for the NAP strategy.

emitter. For the parallelisation of YaDT, we adopt a two-phases 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 changing only these two methods. The implementation of the NP and the NAP strategies are shown in Fig. 6 and Fig. 7 respectively.

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, and its weight is set to the number of cases at the root (§6.3-5). Upon receiving in input a task coming from a worker, the emitter checks the termination conditions (§6.10), and then produces in output the sub-tasks corresponding to the children of the node (§6.12-18). The `ff_send_out` method of the FastFlow runtime allows for queueing tasks without returning from the method. Finally, the `FF_GO_ON` tag in the return statement (§6.19) tells the run-time that the computation is not finished (this is stated by returning NULL), namely further tasks must be waited for from the input channel. The `ff_worker::svc` method for a generic worker (§6.22-25) 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[4]; 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 [16, Fig. 1], which show how the YaDT implementation of `node::split` exhibits a low-variance elapsed time per case for the vast majority of nodes.

NAP strategy (Fig. 7). The NAP strategy builds over NP. For a given decision node, the emitter follows a D&C 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 (§7.3-10). 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 `attTasks` 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 (§7.16). If this is the case (§7.17-20), the `attTasks` 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, the emitter is given a processed node (either from a worker, or as the result of the `node::splitPost` call). Unless the termination conditions occur (§7.22), the emitter proceeds with outputing tasks. The `buildAttTest` at §7.28 controls for each child node whether to generate a single node processing task, or one attribute processing task for each attribute at the child node. In the former case (§7.29-31), we proceed as in the NP strategy; in the latter case

[4]It was not among FastFlow strategies, and it has been added as a result of YaDT-FF.

(§7.33-38), we proceed as for the root node[5]. Based on the task label, the `ff_worker::svc` method for a generic worker (§7.46-50) merely calls the node splitting procedure or the information gain calculation for the involved attribute.

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

The first issue concerns task weights. Node processing tasks are weighted with r (§7.30), as in the NP strategy. Although attribute processing tasks have a finer grain, which suggests a lower weight, there exists a synchronisation point – all attribute tasks must be processed before the emitter can generate tasks for the child nodes. By giving a lower weight, we run the risk that all attribute tasks are assigned to the most unloaded worker, thus obtaining a sequential execution of the attribute tasks! For these reasons, attribute processing tasks are weighted with r as well (§7.9,§7.37).

The second issue concerns the test `buildAttTest`, which decides whether to perform nodes or attributes parallelisation. We have designed and experimented three cost models. Attribute parallelisation is chosen respectively when:

- $(\alpha < r)$ the number of cases is above some hand-tuned threshold value α;
- $(|\mathcal{T}| < c\, r\, log\, r)$ the average grain of node processing (quicksort is $r\, log\, r$ on average) is higher than a threshold that is dependent on the training set. Intuitively, the threshold should 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 an attribute at the root is $|\mathcal{T}|\, log\, |\mathcal{T}|$, we fix the threshold to a lower bound for such a value, namely to $|\mathcal{T}|$;
- $(|\mathcal{T}| < c\, r^2)$ the worst-case grain of node processing (quicksort is r^2) is higher than a threshold that is dependent on the training set. As in the previous case, the threshold is set to $|\mathcal{T}|$. The higher value cr^2, however, leads to selecting attributes processing more often than the previous case, with the result of task over-provisioning.

All tests are monotonic in the number r of cases at the node. Hence, if the nodes parallelisation is chosen for a node, then it will be chosen for all of its descendants. As we will see in Sec. 5, the third cost model shows the best performance.

5 Performance Evaluation

In this section we show the performances obtained by YaDT-FF. The datasets used in the tests with their characteristics are reported in Table 1. They are publicly available from the UCI KDD archive, apart from *SyD10M9A* which is synthetically generated using function 5 of the QUEST data generator. All

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

			No. of attributes			Tree	
$\mathcal{T}$ name	$\lvert\mathcal{T}\rvert$	NC	discr.	contin.	total	size	depth
Census PUMS	299,285	2	33	7	40	122,306	31
U.S. Census	2,458,285	5	67	0	67	125,621	44
KDD Cup 99	4,898,431	23	7	34	41	2,810	29
Forest Cover	581,012	7	44	10	54	41,775	62
SyD10M9A	10,000,000	2	3	6	9	169,108	22

Table 1: Training sets used in experiments, and size of the induced decision tree.

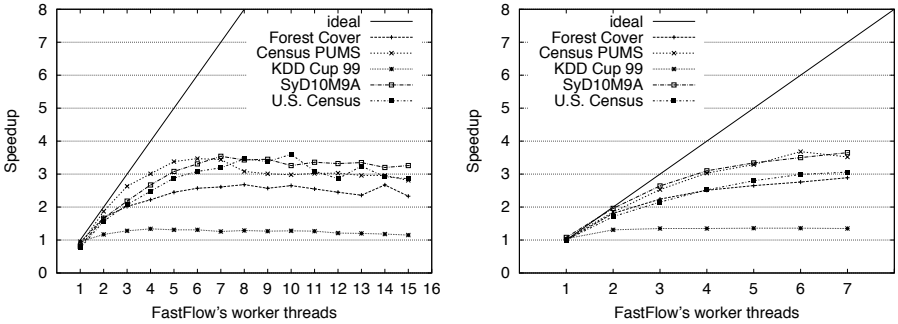

Figure 8: NP strategy speedup. Nehalem box (left), Harpertown box (right).

presented experimental results are taken performing 5 runs, excluding the higher and the lower value obtained and computing the average of the remaining ones.

Experimental framework. All experiments were executed on two different Intel workstation architectures: *Nehalem)* a dual quad-core Xeon E5520 Nehalem (16 HyperThreads) @2.26GHz with 8MB L3 cache and 24 GBytes of main memory with Linux x86_64. *Harpertown)* a dual quad-core Xeon E5420 Harpertown @2.5GHz 6MB L2 cache and 8 GBytes of main memory, with Linux x86_64. They are a quite standard representative of current and immediately preceding generation of (low-cost) server boxes. 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. 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.

Performance. Let us start considering the *NP strategy*, i.e., the parallelisation of nodes processing. The obtained speedup is shown in Fig. 8. The maximum speedup is similar on both architectures, and quite variable from a dataset to another; it ranges from 1.34 to 3.54 (with an efficiency of 45%). 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 nodes (those in the higher levels of the tree), thus lacking parallelism. This phenomenon has been already observed in previous work on the parallelisations of decision tree construction over distributed memory architectures [9]. These systems, however, suffer from load balancing problems, which we will handle later on, and high costs of communications, which in shared memory architectures do not occur. Summarising, although the NP strategy yields a modest speedup, it is worth noting that the effort required to port the sequential code was minimal.

The *NAP strategy* aims 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 presented in Sec. 4, the emitter relies on a *cost model* in order to decide whether to adopt attributes parallelisation. We have tested the three cost models discussed in Sec. 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 exhibits an higher task over-provisioning if compared to the test $|\mathcal{T}| < cr \log r$, and it is dataset-tailored if compared to $\alpha < r$. In all of the remaining experiments, we use that model.

The speedup of YaDT-FF with the NAP strategy is shown in Fig. 9. It ranges from 4 to 7.5 (with an efficiency of 93%). The speedup gain over the NP strategy is remarkable. Only for the *Census PUMS* dataset, the smallest dataset as for number of cases, the speedup gain is just +12% over NP. Notice that the *SyD10M9A* dataset apparently benefits 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. 14.

YaDT-FF also exhibits a good scalability with respect to both the number of attributes (Fig. 10) and to the number of cases (Fig. 11) in the training set. The plots refer to subsets of the *SyD10M9A* dataset possibly joined with randomly distributed additional attributes. In the former case, the maximum speedup (7×) is reached as soon as the number of attributes doubles the available hardware parallelism (18 attributes for 8 cores). In the latter case, the achieved speedup increases with the number of cases in the training set.

Load-balancing. 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. This is exacerbated in the parallelisation of the original C4.5 implementation, because of the linear search of the threshold (§2.10). Fig. 14 shows that load balancing is not a critical issue for YaDT-FF with the NAP strategy. We motivate this 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 online scheduling within the emitter; 2) FastFlow communications are asynchronous and exhibit very low overhead [1]. This makes it possible to sustain all the workers with tasks to be processed for

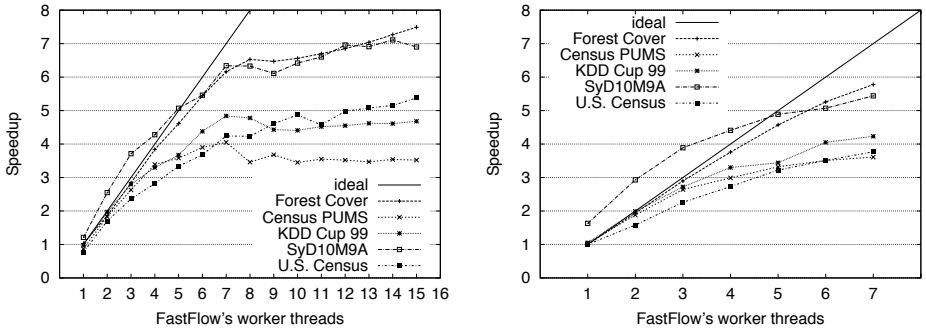

Figure 9: NAP strategy speedup. Nehalem box (left), Harpertown box (right).

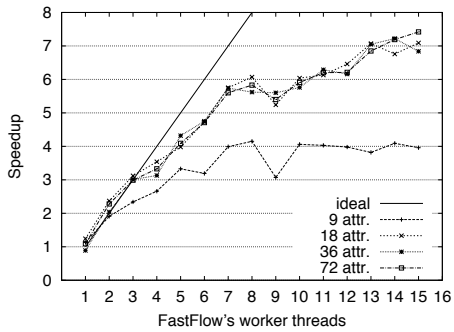

Figure 10: Speedup vs no. of attributes for $1M$ sample cases from *SyD10M9A*.

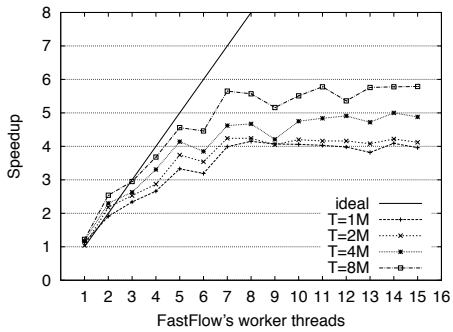

Figure 11: Speedup vs no. of sample cases (T) from *SyD10M9A*.

the entire run. This also reduces the dependence of the achieved speedup from the effectiveness of the scheduling policy. Nevertheless, such dependence exists.

Fig. 13 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 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. 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. 13 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. Finally, we show in Fig. 15 how often nodes parallelisation has been chosen by

	Total Execution Time (sec.)		
$\mathcal{T}$ name	$\lvert\mathcal{T}\rvert < cr^2$	$\alpha < r$	$\lvert\mathcal{T}\rvert < cr \log r$
Census PUMS	**0.85**	0.85	0.91
U.S. Census	**3.28**	3.51	3.35
KDD Cup 99	**3.76**	3.80	3.77
Forest Cover	**2.64**	2.66	2.73
SyD10M9A	16.90	**16.68**	18.16

Effectiveness of *buildAttTest(c,r)* implementations for different attributes parallelisation cost models. $|\mathcal{T}|$ is the number of cases, c is the number of node attributes, r is the number of node training cases, and $\alpha = 1000$. Bold figures highlight best results.

Figure 12: Attributes parallelisation tests (Nehalem, 7 worker threads).

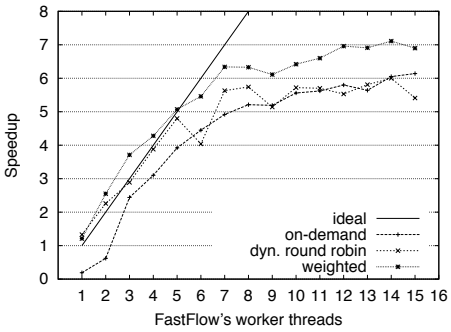

Figure 13: Speedup of different scheduling policies over *SyD10M9A*.

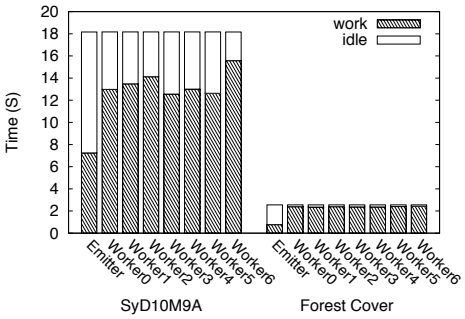

Figure 14: YaDT-FF execution breakdown (Nehalem, 7 worker threads).

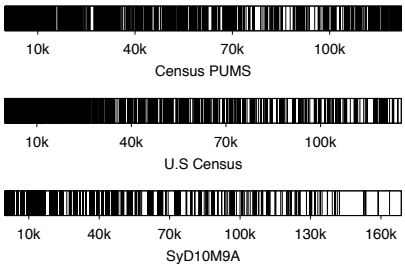

Figure 15: Nodes (white) vs attributes (black) parallelisation choices.

the emitter against the attributes parallelisation (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. We briefly evaluate the benefits achieved using the Nehalem HyperThreaded box. 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 memory operations with threads execution. The comparison between the two graphs in Fig. 9 shows that several datasets benefit of (about) 30% improvement due to SMT; some others, such as *Census PUMS* and *KDD Cup*, show only a modest benefit (about 12%). These figures match the expected benefit for this kind of

architectures. As future work, we believe that the effectiveness of SMT can be further improved by devising a *cache-aware* weighted scheduling policy. This will not affect the general structure of the code but only the calls the `setWeight` method (§6.16).

6 Related Work

Over the last decade, parallel computing aimed at addressing three main classes of data mining issues: 1) solve inherently distributed mining problems, e.g., mining of datasets that are bound to specific sites due to privacy issues; 2) manage larger datasets by exploiting the aggregate memories of different machines; 3) decrease the processing time of mining algorithms. In many cases, the latter two issues have been jointly addressed by trying to bring in-core datasets that are out-of-core on a single machine. This approach, which often requires the redesign of the algorithms or the introduction of new scalable data structures, is loosing interest with the ever-increasing availability of main memory space. Our work distinguishes from this approach, even if it clearly belongs to the third class.

Considering the parallelisation methodology, related works can be categorised as follow: 1) exploiting attributes parallelism by partitioning the training set by columns and then adopting data parallelism [18, 11]; 2) exploiting nodes parallelism by independently building different nodes or sub-trees adopting task parallelism [6]; 3) combining the two above in various fashions [19, 22].

Several works focus on distributed-memory machines, including SPRINT [18], ScalParC [11], pCLOUDS [19], and the approach of [6]. On these shared-nothing architectures the cost of communications is the critical aspect. The use of scalable data structures and of efficient load balancing techniques, trying to minimise costly data redistribution operations, are the most important factors to obtain good performance. As an example, the earliest SPRINT parallel algorithm adopts scalable SLIQ data structure for representing the dataset, but it suffers from communication bottlenecks addressed in the successor system ScalParC. pCLOUDS [19] combines both the data parallel and the task parallel approaches. It exploits data parallelism for large decision nodes, then it switches to a task parallel approach as soon as the nodes become small enough. The proposal of [6] categorises tasks in three different classes: large, intermediate and small ones. Large tasks process a decision node, as to increase the degree of available parallelism. Intermediate tasks process a sub-tree up to a given number of nodes and within computation time bounds. Small tasks sequentially process the whole sub-tree of a node.

YaDT-FF takes inspiration from the two latter works and distinguish from them for: 1) it does not need the redesign of the sequential algorithm but rather an *easy-yet-efficient* porting of the existing code; 2) it targets multicore rather than distributed memory machines; 3) it adopts an effective cost model for deciding whether to parallelise on nodes or on attributes. In conclusion, to the best of our knowledge, very few works specifically target data mining

$\mathcal{T}$ name	Seq.Time (S)	1E+1W	1E+2W	1E+3W	Max Boost
		Time (S)			
Census PUMS	4.46	4.3	2.37	1.69	2.64×
U.S. Census	17.67	17.97	11.17	7.8	2.26×
KDD Cup 99	18.11	17.26	9.12	6.67	2.71×
Forest Cover	16.99	16.97	8.74	5.86	2.90×
SyD10M9A	103.21	93.95	52.34	39.37	2.62×

Table 2: YaDT vs YaDT-FF on a Nehalem quad-core (E= Emitter, W=Worker).

systems on multicore architectures [10, 8, 4, 14], but none specifically decision tree algorithms.

7 Conclusions

Nowadays, and for foreseeable future, the performance improvement of a single core will no longer satisfy ever increasing computing power demand. For this, computer hardware industry shifted to multicore, and thus the extreme optimisation of sequential algorithms is not longer 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 algorithms, by porting YaDT using the FastFlow parallel programming framework. The strength of our approach consists in the minimal change of the original code with, a the same time, a non-trivial parallelisation strategy implementation (nodes and attributes parallelism plus weighted problem-aware load balancing) and with notable speedup. 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 2 shows that the parallelisation of YaDT boosts up to 2.9×, with no additional cost to buy a specific parallel hardware.

References

[1] M. Aldinucci, M. Meneghin, and M. Torquati. Efficient Smith-Waterman on multi-core with FastFlow. In *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, pages 195–199, Pisa, Italy, 2010. IEEE.

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

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[4] G. T. Buehrer. *Scalable mining on emerging architectures*. Phd thesis, Columbus, OH, USA, 2008.

[5] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

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

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

[8] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *Proc. of the Intl. Conference on Very Large Data Bases (VLDB)*, pages 577–588, 2005.

[9] E. Han, A. Srivastava, , and V. Kumar. Parallel formulation of inductive classification parallel algorithm. Technical report, Department Computer and Information Science, University of Minnesota, 1996.

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

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

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

[13] I. Park, M. J. Voss, S. W. Kim, and R. Eigenmann. Parallel programming environment for OpenMP. *Scientific Programming*, 9:143–161, 2001.

[14] J. Pisharath, J. Zambreno, B. Ozisikyilmaz, and A. Choudhary. Accelerating data mining workloads: Current approaches and future challenges in system architecture design. In *Proc. of Workshop on High Performance and Distributed Mining*, 2006.

[15] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[16] S. Ruggieri. Efficient C4.5. *IEEE Transactions on Knowledge and Data Engineering*, 14:438–444, 2002.

[17] S. Ruggieri. YaDT: Yet another Decision tree Builder. In *16th IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI 2004)*, pages 260–265. IEEE, 2004.

[18] J. C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. 22nd Int. Conf. Very Large Databases*, pages 544–555, 1996.

[19] M. K. Sreenivas, K. Alsabti, and S. Ranka. Parallel out-of-core divide-and-conquer techniques with application to classification trees. In *IPPS '99/SPDP '99: Proc. of the 13th Intl. Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 555–562, Washington, DC, USA, 1999. IEEE.

[20] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 11th Intl. Conference on Compiler Construction (CC)*, pages 179–196, London, UK, 2002. Springer-Verlag.

[21] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12), 2002.

[22] M. Zaki, C.-T. Ho, and R. Agrawal. Parallel classification for data mining on shared-memory multiprocessors. In *Proc. of Intl. Conference on Data Engineering (ICDE)*, pages 198–205. IEEE, 1999.