Decision tree building on multi-core using FastFlow

Author's version of
Concurrency Computat.: Pract. Exper. 2014; 26:800–820
Published online 27 June 2013 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3063

Decision tree building on multi-core using FastFlow

Marco Aldinucci[1] and Salvatore Ruggieri[2*] and Massimo Torquati[2]

[1] *Computer Science Department, University of Torino, Corso Svizzera 185, 10149 Torino, Italy*
[2] *Computer Science Department, University of Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy*

SUMMARY

The whole computer hardware industry embraced the multi-core. The extreme optimisation of sequential algorithms is then 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 in-depth study of the parallelisation of an implementation of the C4.5 algorithm for multi-core architectures. We characterise elapsed time lower bounds for the forms of parallelisations adopted, and achieve close to optimal performance. Our implementation is based on the FastFlow parallel programming environment and it requires minimal changes to the original sequential code.

Received . . .

KEY WORDS: parallel classification; decision trees; C4.5; multi-core

1. INTRODUCTION

After decades of enhancements of single core chips by increasing 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, which are used to equip the whole range of platforms, from laptop to high-performance distributed platforms. The multiplication of cores, however, does not always translate into greater performance. On one hand, sequential code will get no performance benefits from them. On the other hand, the exploitation of multi-core as distributed machines (e.g. via message-passing libraries) does not fully squeeze their potentiality, and often, it requires the full redesign of algorithms. Developers, including data and knowledge engineers, are then facing the challenge of achieving a trade-off between performance and human productivity (cost and time to solution) in developing applications on multi-core. Parallel software engineering have engaged this challenge mainly by way of design tools, in the form of high-level sequential language extensions and coding patterns [1, 2].

In this paper, we present an approach for growing and pruning decision trees on multi-core machines. To the best of our knowledge, this is the first work on the parallelisation of decision trees for modern shared-cache multi-core platforms. Even though there is an abundant literature for distributed systems, the direct application of such approaches to multi-cores suffers from the technical and methodological problems highlighted earlier. We consider the C4.5 algorithm by

*Correspondence to: S. Ruggieri, Computer Science Department, University of Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy. Email: ruggieri@di.unipi.it.

Quinlan [3], a cornerstone in data mining and machine learning (see e.g., [4]). C4.5 constructs a decision tree top-down in the *growing phase*, then prunes branches by a bottom-up transversal in the *pruning phase*. Tree growing starts from a set of cases with predictive attributes and a class value, and at each node it follows a Divide&Conquer pattern by selecting an attribute for splitting cases into subsets, which are assigned to child nodes, and then recursing on each of those subsets. Tree pruning consists of a doubly-recursive bottom-up visit of the decision tree, pruning sub-trees according to an error estimation function. More in detail, we start from the Yet another Decision Tree builder (YaDT) implementation of C4.5 [5], which is a from-scratch and efficient C++ version of the Quinlan's algorithm. YaDT is a quite paradigmatic example of sequential complex code, adopting extreme sequential optimisations, 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 CPU cores on the user's machine. Even when the execution time of a single tree growing and pruning is reasonably low, there are applications that motivate performance improvement. For instance, ensembling methods, such as windowing [3] and boosting [6], build a (long) sequence of classification models, where each classifier possibly depends on the predictive performances of the previous ones (hence, with no immediate parallelisation of the ensembling). Another case where an almost real-time efficiency is expected is explorative data analysis, where users can interactively navigate and re-build decision (sub-)trees [7].

Our approach for parallelising YaDT is based on the FastFlow programming framework [8], a recent proposal for parallel programming over multi-core 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 a *methodological approach* that will lead us to parallelise YaDT with minimal changes to the original sequential code, yet achieving up to $10\times$ boost in performance on a single twelve-core CPU. Nevertheless, because the pattern-based programming methodology of FastFlow is common to a large portion of the mainstream frameworks for multi-core [2, 9], the approach proposed can be verified with other frameworks.

In this paper, we provide an in-depth experimental analysis of the tree growing parallelisation of C4.5, with 64-bit compilation of all software and diversified hardware architectures. The main novel research contribution of this paper consists of characterising *lower bounds on the elapsed time of the forms of parallelisation* adopted. We also show how the performances achieved by our specific implementation are close to such bounds. Surprisingly enough, existing works on parallelisation of decision tree induction algorithms have conducted no such bound analysis of their performances. This is a major advancement over related approaches. The approach proposed has been replicated to tackle the parallelisation of the tree pruning phase, a common post-processing step that alleviates for the over fitting problem. The error-based decision tree pruning strategy of C4.5 is considerably faster than tree growing on sequential algorithms, but it becomes the next bottleneck once the growing phase has been parallelised and sped up. To the best of our knowledge, this is the first work to deal with parallel pruning of decision trees.

Preliminary results on our parallelisation of the decision tree growing phase appeared in [10]. The experimental framework has been revised in this paper by moving to a 64-bit compilation of all software (vs 32-bit) and to up-to-date hardware architectures. The other major contributions of this paper, including the characterisation and experimentation of time lower bounds and the parallelisation of the tree pruning phase, are completely original contributions.

This paper is organised as follows. In Section 2, the FastFlow programming environment is introduced. We recall in Section 3 the C4.5 decision tree growing algorithm, including the main optimisations that lead to YaDT. The parallelisation of YaDT is presented in detail in Section 4, followed by an experimental evaluation in Section 5. Section 6 discusses how the approach extends to the post-processing phase of decision tree pruning and compares memory occupation of the sequential and parallel versions of YaDT. We present related works in Section 7, and summarise the contributions of the paper in the conclusions.

DOI: 10.1002/cpe

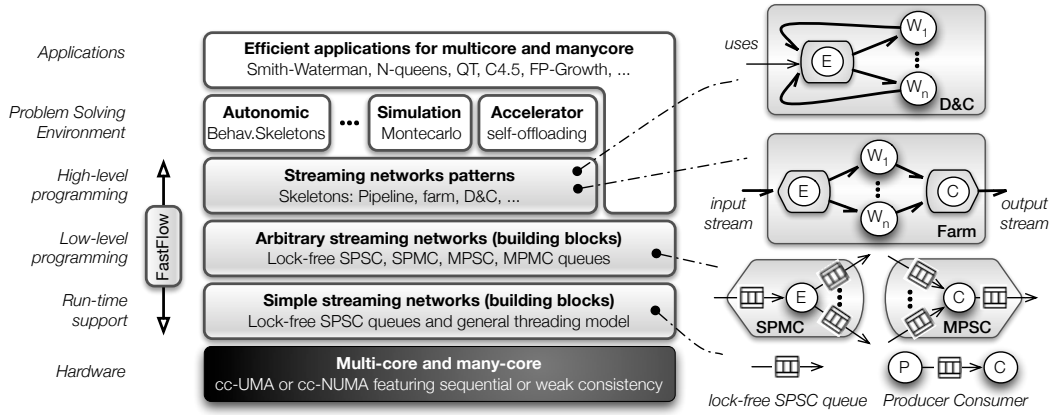

Figure 1. FastFlow layered architecture with pattern examples.

2. THE FASTFLOW PARALLEL PROGRAMMING ENVIRONMENT

FastFlow is a parallel programming framework aiming at simplifying the development of applications for multi-core platforms, being these applications either brand new or ports of legacy codes [8]. FastFlow fosters pattern-based programming and can support fine-grained parallel computations. It directly provides programmers with a reduced set of modular patterns as vehicles of common streaming paradigms. Because patterns are implemented as C++ templates, new user-defined variants of patterns can be defined by way of standard object oriented techniques.

Basically, parallel patterns provide 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. Data parallel computations can be exploited by way of the *map* and *reduce* patterns, which also work on streams and can be nested with stream patterns [11]. The possibility to efficiently handle both stream parallel and data parallel computations using the same programming model represents an 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 multi-core systems by building the skeletons themselves on top of a library of lock-free producer/consumer queues [12]. As shown in Figure 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 high-level programming patterns. The higher layer provides parallel programming patterns, which are compiled onto streaming networks that are implemented using only lock-free queues and *mediator* threads. As an example, a *farm* pattern is implemented as shown in Figure 1, mid right schema. A mediator (E or *Emitter*) is used to dispatch tasks appearing onto the input stream towards a pool of *worker* threads. These, in turn, deliver results to a mediator (C or *Collector*) that delivers the results onto the output stream.

FastFlow can be categorised as a high-level parallel programming framework. This category includes several mainstream frameworks such as Hadoop [13], Intel ArBB & TBB [2], and OpenMP [1]. Hadoop targets distributed platforms and it implements the MapReduce pattern only. Intel ArBB and OpenMP target shared-memory multi-core platforms but they provide mainly data parallel patterns. Intel TBB provides programmers with data parallel patterns (i.e., TBB algorithms), a limited support for stream parallelism (pipelining) but it does not support full control of parallelism

DOI: 10.1002/cpe

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

Figure 2. Node splitting procedure.

in recursive algorithms as it can be done in FastFlow. For instance, the TBB Range structure, which could be exploited to model recursive partitioning as in decision tree growing, supports only binary splits into sub-Ranges, thus forcing a parallelisation approach of multi-way splits more complex than needed. As shown in this paper, FastFlow enables the parallelisation of decision tree building with few, localised changes in the sequential code. FastFlow is available as open source software under LGPLv3 [14]. More details as well as performance comparisons against other parallel programming frameworks such as Cilk, OpenMP, and Intel TBB can be found in [14, 15].

3. THE C4.5 SEQUENTIAL ALGORITHM FOR DECISION TREE GROWING

Classifiers are induced by supervised learning from a relation $\mathcal{T}$ called the *training set*. An attribute C of the relation is the *class*, while the remaining ones $A_1, \ldots, A_m$ are the *predictive attributes*. Tuples in $\mathcal{T}$ are called *cases*. Predictive attributes can be discrete or continuous and unknown or unspecified values may appear in cases. The class is discrete and it does not admit unknown values. 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, 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 outcomes $A = d_1, \ldots, A = d_h$, where $d_1, \ldots d_h$ are the known values in the domain of A. A test on a continuous attribute A has 2 outcomes, $A \leq t$ and $A > t$, where t is a *threshold* value determined at the node. The C4.5 decision tree induction algorithm [3] is a constant reference in the development and analysis of novel classification models (see e.g., [4]). 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*. In this section, we restrict to consider the growing phase, whilst the pruning phase is discussed in Section 6.1. During tree growing, 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, associated with the whole training set $\mathcal{T}$ and all weights set to 1. At each node a Divide&Conquer algorithm selects an attribute for splitting.

Consider the method `node::split` in Figure 2. Let T be the set of cases associated at a node. For every class value c, the weighted frequency of cases in T whose class is c is computed (§2.2 — throughout the paper, we use the **§M.n** notation to reference line **n** from the pseudo-code in Figure **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 as a leaf (§2.3-6). Otherwise, the *information gain* of each attribute at the node is calculated (§2.7-8). Because 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 §2.7). For a discrete attribute A, the information gain of

DOI: 10.1002/cpe

```
    void tree :: build () {
3.2   queue<node *> q;
      root = new node( allCases );
3.4   q.push(root);
      while( !q.empty() ) {
3.6     node *n = q.front();
        q.pop();
3.8     n->split();
        for(int i=0; i<n->nChilds(); ++i) {
3.10      node *child = n->getChild(i);
          child->get_cases();
3.12      q.push( child );
        }
3.14    n->release_cases();
      }
3.16 }
```

Figure 3. Tree growing procedure.

splitting T into subsets $T_1, \ldots, T_h$, one for each known value of A, is calculated. For a continuous attribute A, cases in T with known value for A are first ordered ascending w.r.t. such an attribute, say to values $v_1, \ldots, v_k$. For each $i \in [1, k-1]$, fixed the semi-sum value $v = (v_i + v_{i+1})/2$, the information gain $gain_v$ is computed by considering the splitting of T into cases T_1^v whose value for the attribute A is lower than or equal than v, and cases T_2^v whose value is greater than v is considered. The value v' for which $gain_{v'}$ is maximum is called the *local threshold* and the information gain for the attribute A is set to $gain_{v'}$. The attribute A with the highest information gain is selected for the test at the node (§2.9). If A is continuous, the *threshold* of the split is computed (§2.10-11) 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 (§2.12-16). For each split of the selected attribute, a new child node is added, whose cases are selected from T according to the split test. Cases with unknown value of the selected attribute are passed to each child, but their weights are scaled.

The original Quinlan's implementation of C4.5 was written in ANSI C. Several optimisations were proposed in the last two decades, concerning both the data structures holding the training set and the computation of the information gain of continuous attributes, which is the most computationally expensive procedure [16, 17, 18]. YaDT [5] is a from scratch C++ implementation of C4.5 (Release 8, the latest), implementing and enhancing the optimisations proposed in the cited papers. Its object-oriented design 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, whose pseudo-code is shown in Figure 3 as method `tree::build`. The call to the method `get_cases` (§3.11) builds an array of cases (case id's, actually) and weights for a child node starting from the same data structure at the parent node. The method `release_cases` (§3.14) releases the data structure at the parent node once all children are queued. Experimental results show that YaDT reaches up to $10\times$ improvement over C4.5 with only 1/3 of its memory occupation [5, 18].

4. PARALLELISATION STRATEGIES OF DECISION TREE GROWING ALGORITHM

Two major approaches have been considered in the literature for parallelising decision tree growing. Task parallelism consists of splitting the processing of different subtrees into independent tasks in a Divide&Conquer fashion. Data parallelism consists in (logically or physically) distributing the training set among the processors by partitioning attributes or cases. In the related works (Section 7), we will provide details of the various approaches. We propose here a parallelisation of YaDT, called YaDT-FastFlow (YaDT-FF), obtained by stream parallelism, which allows us for mixing task and data 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

DOI: 10.1002/cpe

```
    void tree :: build_ff () {
4.2   root = new node( allCases );
      E=new ff_emitter(root,PAR_DEGREE);
4.4   std :: vector<ff_worker*> w;
      for(int i=0;i<PAR_DEGREE;++i)
4.6    w.push_back( new ff_worker());
      ff_farm<ws_scheduler>
4.8       farm(PAR_DEGREE*QSIZE);
      farm.add_workers(w);
4.10  farm.add_emitter(E);
      farm.wrap_around();
4.12  farm.run_and_wait_end();
    }
```

Figure 4. YaDT-FF tree growing setup.

```
    void * ff_emitter ::svc(void * task) {
5.2   if (task == NULL) {
        task = new ff_task(root);
5.4     int r = root->getNoCases();
        setWeight(task, r);
5.6     return task;
      }
5.8   node *n = task->getNode();
      int nChilds = n->nChilds();
5.10  for(int i=0; i<nChilds; i++) {
        node *child = n->getChild(i);
5.12    ctask = new ff_task(child);
        child->get_cases();
5.14    int r = child->getNoCases();
        setWeight(ctask, r);
5.16    ff_send_out(ctask);
      }
5.18  n->release_cases();
      if (!nChilds && noMoreTasks())
5.20    return NULL;
      return FF_GO_ON;
5.22 }

5.24 void * ff_worker ::svc(void * task) {
      node *n = task->getNode();
5.26  n->split();
      return task;
5.28 }
```

Figure 5. Emitter and worker definition for the NP strategy.

the Divide&Conquer paradigm. The FastFlow Divide&Conquer schema is shown in the top-right corner of Figure 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 approach: first, we accelerate the `tree::build` method (see Figure 3) 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 Figure 2) by exploiting data parallelism 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 Figure 4, which creates an emitter object (§4.2-3) and an array of worker objects (§4.4-6). The size of the array, `PAR_DEGREE`, is the parallelism degree of the farm skeleton. The root node of the decision tree is passed to the constructor of the emitter object. The parallelisation is managed by the FastFlow layer through a `ff_farm` object, which creates feedback channels between the emitter and the workers (§4.7-11). 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 First In First Ou (FIFO) queue. The emitter class `ff_emitter` and the worker class `ff_worker` define the behaviour of the farm skeleton through the methods `svc` (shorthand for *service*), which are called by the FastFlow run-time to process input tasks. Parallelisation strategies are defined by coding only these two methods. Let us describe them in detail.

4.1. Nodes parallelisation strategy (Figure 5)

At start-up the `ff_emitter::svc` method is called by the FastFlow run-time with a NULL parameter (§5.2). 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

DOI: 10.1002/cpe

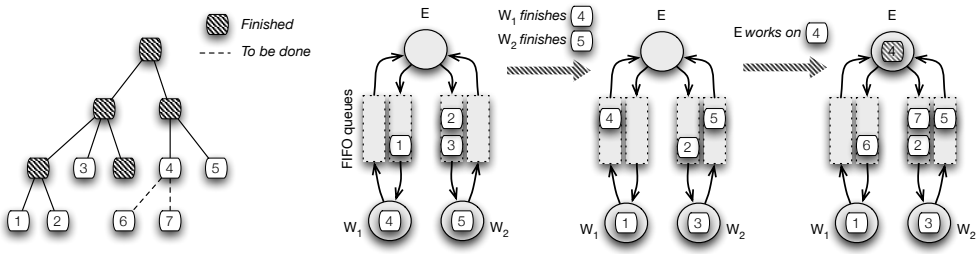

Figure 6. Example emitter and worker states in the NP strategy.

of cases at the root (§5.3-5). 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, by way of the `ff_send_out` method, produces in output the sub-tasks corresponding to the children of the node (§5.8-17). 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 (§5.18). If there are no child nodes and no more tasks in worker queues (§5.19-20), the emitter returns NULL as to signal that the computation is finished. Otherwise, the `FF_GO_ON` tag in the return statement (§5.21) tells the run-time that further tasks must be waited for from the input channel. The `ff_worker::svc` method for the generic farm worker (§5.24-28) calls the node splitting algorithm `node::split`, and then it immediately returns the computed task back to the emitter. The overall coding is simple — 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 in [18, Figure 1], showing that the YaDT implementation of `node::split` exhibits a low-variance elapsed time per case for the vast majority of nodes.

Figure 6 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 queue, and it starts processing node 1 from its input queue. Similarly, worker W_2 delivers node 5, and it starts processing node 3. Assume that the emitter reads first node 4 from its input queue [†]. Because node 4 has two child nodes, namely node 6 and 7, the emitter generates tasks for them, and delivers the tasks in the workers' queues. The scheduling policy determines which worker each task is assigned to. Notice that, because queues are FIFO, the scheduling policy does not affect the order of tasks within a worker's queue.

4.2. Nodes and attributes parallelisation strategy (Figure 7)

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 (§2.7-8). In such a case, the stopping criterion at §2.3 must be evaluated prior to the parallelisation, and the creation of the child nodes occurs after all information gains are computed. This leads to partitioning the code of `node::split` into three methods, as shown in Figure 8.

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 `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

[†] In general, the choice is non-deterministic.

DOI: 10.1002/cpe

```
    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->child_cnt = c;
        return FF_GO_ON;
7.14  }
      node *n = task->getNode();
7.16  if (task->isBuildAtt()) {
        if (--n->child_cnt>0)
7.18      return FF_GO_ON;
        n->splitPost();
7.20  }
      int nChilds = n->Childs();
7.22  for(int i=0; i< nChilds; i++) {
        node *child = n->getChild(i);
7.24    child->get_cases();
        int r = child->getNoCases();
7.26    int c = child->getNoAtts();
        if (! buildAttTest(r,c)) {
7.28      ctask =new ff_task(child,BUILD_NODE);
          setWeight(ctask, r);
7.30      ff_send_out(ctask);
        } else {
7.32      if (child->splitPre()) continue;
          for(int j=0; j<c; ++j) {
7.34        ctask =new ff_task(child,BUILD_ATT);
            ctask->att = j;
7.36        setWeight(ctask, r);
            ff_send_out(ctask);
7.38      }
          child->child_cnt = c;
7.40    }
        n->release_cases();
7.42    if (!nChilds && noMoreTasks())
          return NULL;
7.44  return FF_GO_ON; }

7.46  void * ff_worker::svc(void * task) {
        node *n = task->getNode();
7.48    if (task->isBuildAtt())
          n->splitAtt(task->att);
7.50    else
          n->split();
7.52    return task;
      }
```

Figure 7. Emitter and worker definition for the NAP strategy.

```
    bool node::splitPre() {
8.2   computeFrequencies();
      if (oneClass() || fewCases()) {
8.4     set_as_leaf(); return true;
      } return false;
8.6 } void node::splitAtt(int i) {
      gain[i] = gainCalculation(i);
8.8 } void node::splitPost() {
      int best = argmax(gain);
8.10  if (attr[best].isContinuous())
        findThreshold(best);
8.12  ns = attr[best].nSplits();
      for(int i=0;i<ns;++i)
8.14    childs.push_back(
          new node(selectCases(best,i)));
8.16 }
```

Figure 8. Partitioning of `node::split`.

emitter checks whether it concerns the processing of an attribute (§7.16). If this is the case (§7.17-20), 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 (§7.21), 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 (§7.24), and then the test `buildAttTest` at §7.28 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 (§7.28-30), we proceed as in the NP strategy; in the latter case (§7.32-37), we proceed as for the root node[‡]. Once child nodes are generated, the parent node can free cases at the node (§7.41). Finally, if there are no child nodes

[‡]Notice that tasks for nodes (resp., attributes) processing are labelled with `BUILD_NODE` (resp., `BUILD_ATT`).

DOI: 10.1002/cpe

and no more tasks in worker queues (§7.42-43), the emitter returns NULL as to signal that the computation is finished. Otherwise, it returns `FF_GO_ON`, thus waiting for other tasks. Concerning workers, based on the task label, the `ff_worker::svc` method (§7.47-53) 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 (§7.29), as for the NP strategy. Attribute processing tasks have a finer grain, which suggests that they must have assigned a lower weight. However, 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 (§7.9,§7.36).

The second issue is concerned with the test `buildAttTest`, which discriminates between nodes parallelisation and attributes parallelisation. Because 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 conditions. 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 (in YaDT, sorting c attributes by quicksort, which 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. Because 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 a higher number of (finer-grained) concurrent tasks.

As shown in the next section, the third condition exhibits the best performance.

5. PERFORMANCE EVALUATION OF PARALLELISATION STRATEGIES

In this section we show the performances obtained by the NP and the NAP strategies of YaDT-FF. The datasets used in experiments and their characteristics are reported in Table I, 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 [19], apart from datasets *SyD10M9A-xxx*, which are synthetically generated using function 5 of the QUEST data generator [20]. 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 other datasets are standard references in the literature, and they are among the top largest datasets at the UCI Machine Learning repository. Moreover, they are representative of datasets with different number of cases, number of total attributes, number/proportion of continuous attributes, and size of the decision tree. All experimental results are taken by performing 5 runs, excluding the highest and the lowest values, and computing the average of the remaining ones. This is done to smooth out the (actually, very limited) variability introduced by the operating system services running on the experimental machines.

DOI: 10.1002/cpe

Table I. Training sets used in experiments, including number of cases ($|\mathcal{T}|$), number of class values (NC), number of discrete and continuous attributes, and size and depth of the decision trees built from them.

			No. of attributes			Decision tree		
$\mathcal{T}$ name	$\|\mathcal{T}\|$	NC	Discrete	Continous	Total	Size	Depth	Unbalancing
p53Mutants	31 420	2	0	5408	5408	167	20	0.85
Census-Income	299 285	2	33	7	40	122 306	31	0.43
Forest Cover	581 012	7	44	10	54	41 775	62	0.42
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	2810	29	0.65
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

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

	Nehalem		Magny-Cours	
$\mathcal{T}$ name	32-bit	64-bit	32-bit	64-bit
p53Mutants	95.88	95.83	151.54	136.10
Census-Income	3.55	3.16	4.63	4.62
Forest Cover	16.40	13.53	19.10	19.27
U.S. Census	14.24	12.67	17.16	17.25
KDD Cup 99	17.16	15.39	22.67	22.62
SyD10M9A	106.23	114.15	133.41	134.36

5.1. 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 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 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.

5.2. 32-bit vs. 64-bit YaDT compilation

Preliminary results of the parallelisation of YaDT are reported in our conference paper [10]. 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 II we show the sequential execution times obtained from both 32-bit and 64-bit versions of the YaDT tree growing algorithm 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, which shows a performance penalty of about 7%. The higher performances of the 64-bit compilation can be justified by a better utilisation of the underling 64-bit architecture, for example, 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 because of larger pointer representation and larger padding in data structures) hence more cache-misses. Because almost all newer server and workstations are 64-bit architectures and compilers running on 64-bit OSes produce 64-bit executable by default, in the rest of the paper we consider only 64-bit executions.

DOI: 10.1002/cpe

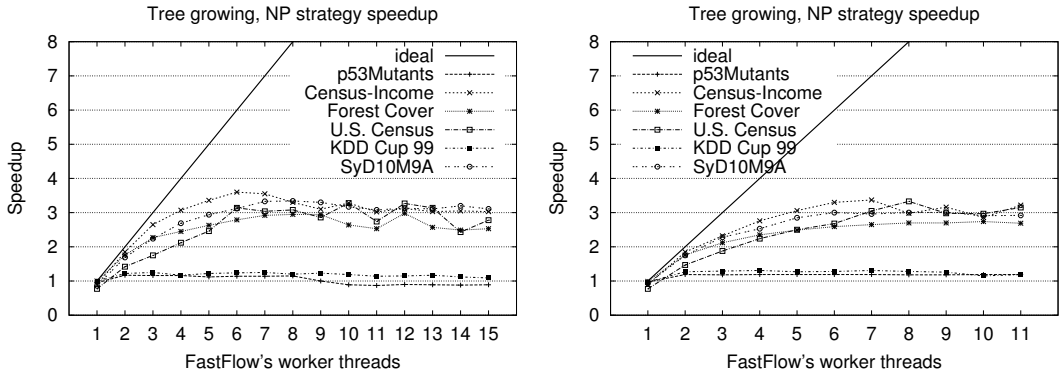

Figure 9. NP strategy speedup. Nehalem box (left), Magny-Cours box (right).

5.3. Performances of the NP strategy

Let us start considering the parallelisation of nodes processing. The speedup[§] obtained by varying the number of farm's worker threads is shown in Figure 9. The maximum speedup is similar on both architectures, and quite variable from a dataset to another; it ranges from 1.16 for *p53Mutants* 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 task parallelisation of decision tree construction over distributed memory architectures (see e.g., [21], and related work in Section 7). 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 strategy based on nodes parallelisation. In symbols, we write

$$lb(n) = t(n) + max_{m \in child(n)} lb(m)$$

where $t(n)$ 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, because 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 $lb(root)$ for the root node of the tree, after it has been completely built. Figure 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. Moreover, Figure 10 also highlights that the scalability of any parallelisation on the grain of nodes is *inherently limited*.

From the theoretical side, we observe that, because $t(n)$ 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 I, the *unbalancing* column reports the ratio between the size of the largest path and the

[§]The speedup metric is defined as $speedup(n) = T_{seq}/T_{par}(n)$ where T_{seq} is the sequential execution time and $T_{par}(n)$ is the parallel execution time with n worker threads.

DOI: 10.1002/cpe

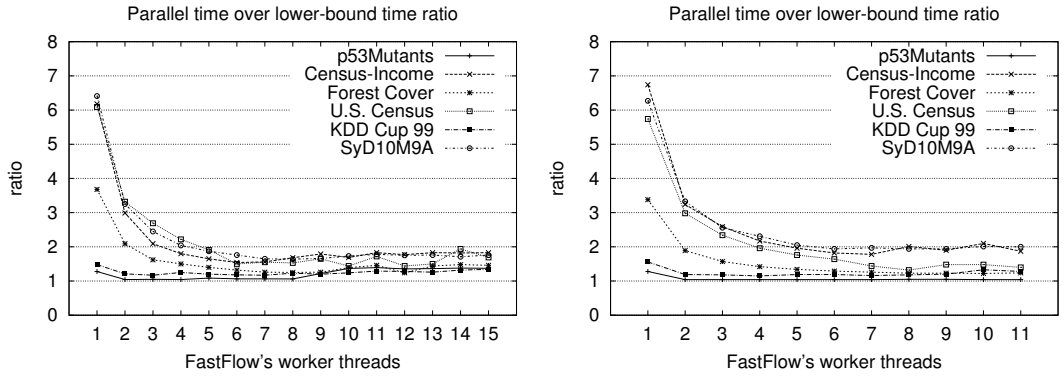

Figure 10. Ratio of elapsed time over lower-bound time for the NP strategy. Nehalem box (left), Magny-Cours box (right).

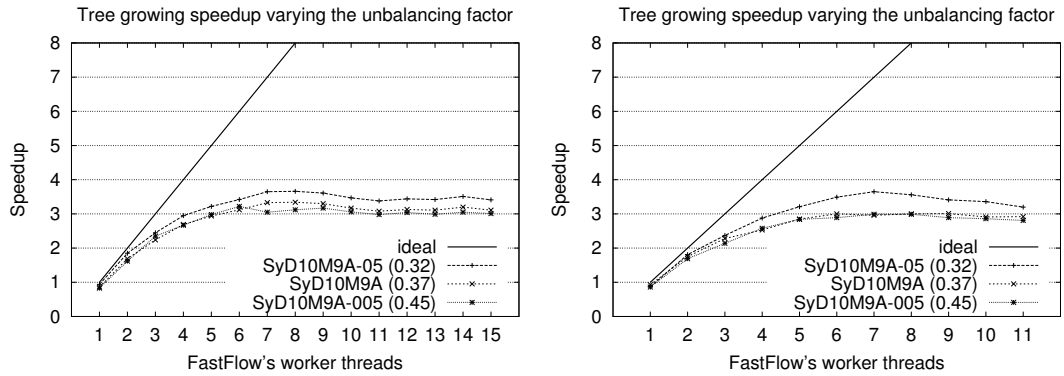

Figure 11. Speedup of the NP strategy for synthetic datasets with different unbalancing factors. Nehalem box (left), Magny-Cours box (right).

Table III. Effectiveness of *buildAttTest(c,r)* for different test conditions (Nehalem box, 7 worker threads). $|\mathcal{T}|$ = no. of cases in the training set, c = no. of attributes at the node, r = no. of cases at the node, and $\alpha = 1000$.

	Total Execution Time (sec.)		
$\mathcal{T}$ name	$\|\mathcal{T}\| < cr^2$	$\alpha < r$	$\|\mathcal{T}\| < cr \log r$
p53Mutants	**14.56**	15.67	14.71
Census-Income	0.76	0.77	**0.68**
Forest Cover	**1.83**	1.93	1.85
U.S. Census	**2.1**	2.14	2.19
KDD Cup 99	**3.46**	3.45	3.60
SyD10M9A	**17.07**	20.11	19.39

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. Figure 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.

DOI: 10.1002/cpe

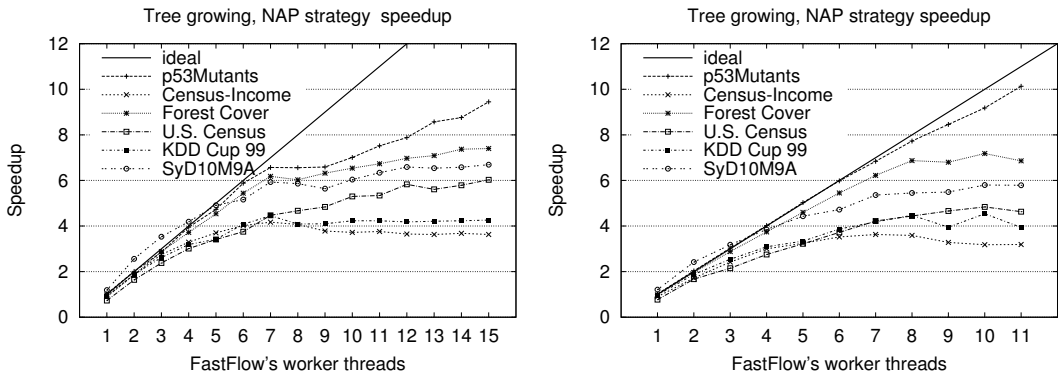

Figure 12. NAP strategy speedup. Nehalem box (left), Magny-Cours box (right).

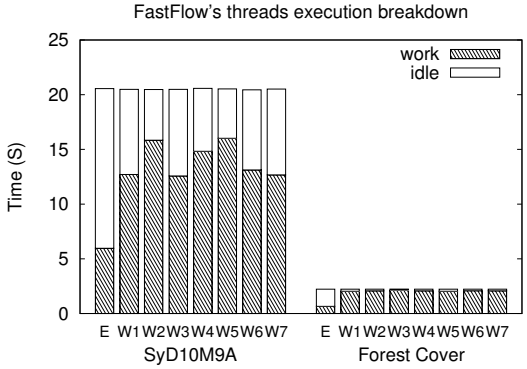

Figure 13. NAP strategy execution breakdown on emitter (E) and workers (W1–W7) (Nehalem box).

5.4. Performances of the NAP strategy

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 discussed in Section 4, the emitter relies on a test condition in order to decide whether to adopt attributes parallelisation. Table III shows that the test $|\mathcal{T}| < cr^2$ provides the best performances among the three test conditions presented in Section 4. This is justified by the fact that the test produces a 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 fix the test $|\mathcal{T}| < cr^2$.

The speedup of YaDT-FF with the NAP strategy is shown in Figure 12. It ranges from 4.1 to 9.45 on the Nehalem architecture and from 3.63 to 10.13 on the Magny-Cours box. Recall that the Nehalem box has eight physical cores (plus SMT hardware support), hence a speedup greater or equal than 7 can be considered optimal or very close to optimal. The speedup gain over the NP strategy is remarkable. Only for the *Census-Income* dataset, the smallest dataset, the speedup gain is just +15% over NP on the Nehalem, and just +8% over NP on the Magny-Cours machine. The attributes parallelism added by the NAP strategy leads the speedup of the *p53Mutants* dataset, the one with the largest set of attributes and the smallest tree size, from the lowest to the highest rank position when compared with the NP strategy. Notice that the *SyD10M9A* dataset apparently benefits from a super-linear speedup. Actually, this occurs because the speedup is defined and 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 Figure 13. As a matter of a fact, the FastFlow

DOI: 10.1002/cpe

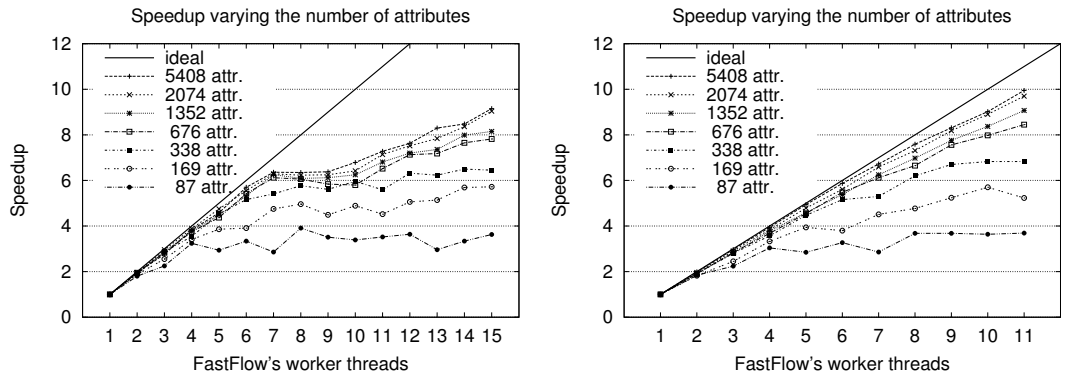

Figure 14. NAP strategy speedup for the *p53Mutants* dataset with a subset of its attributes. Nehalem box (left), Magny-Cours box (right).

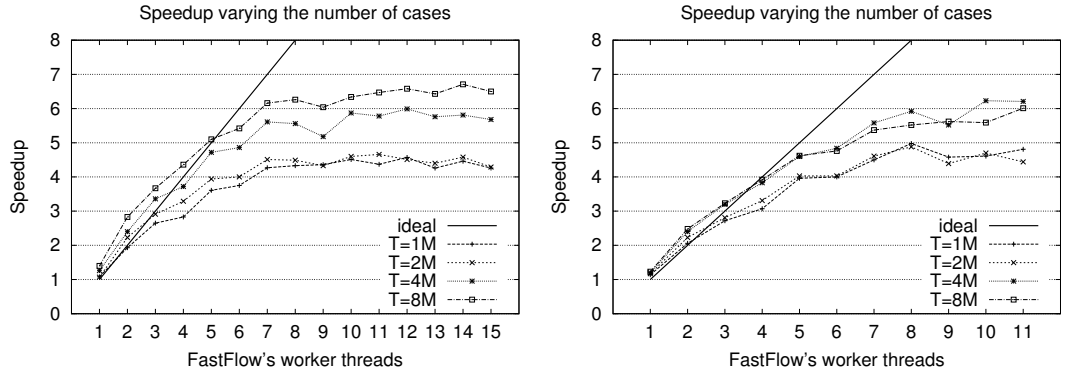

Figure 15. NAP strategy speedup on a random sample of T cases from dataset *SyD10M9A*. Nehalem box (left), Magny-Cours box (right).

farm-with-feedback pattern implicitly exploits pipeline parallelism between the emitter and the generic worker (see also [22]). Our definition of speedup is, however, justified by the fact that, on the emerging platforms with SMT support, the emitter thread can be mapped onto one of the processors running worker threads with very limited or no performance penalty (see later on the discussion on SMT).

YaDT-FF also exhibits a good scalability with respect to both the number of attributes (Figure 14) and the number of cases (Figure 15) in the training set. Figure 14 shows the speedup for the *p53Mutants* dataset with subsets of its predictive attributes. All such subsets include the 87 attributes selected in the decision tree built from all attributes, so that the decision tree remains the same for all of them. When enough parallelism is present, due to a large number of attributes compared to the number of worker threads, we obtain an almost optimal speedup. Figure 15 shows the speedup for subsets of cases of the *SyD10M9A* dataset. 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 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,

DOI: 10.1002/cpe

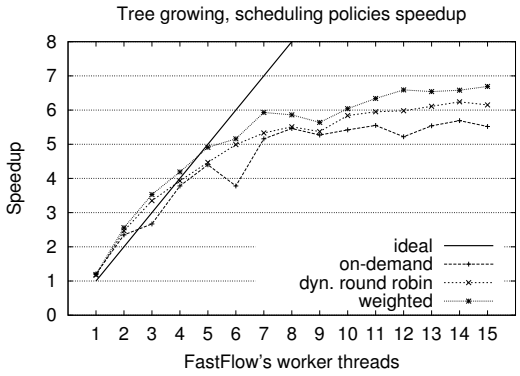

Figure 16. Speedup of different scheduling policies over *SyD10M9A* on the Nehalem box.

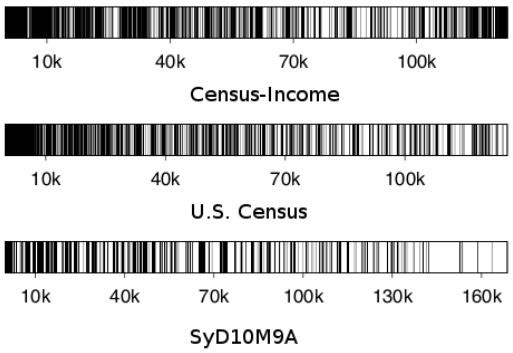

Figure 17. Nodes (white) vs attributes (black) parallelisation choices.

because 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.

5.5. 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 (§2.11) has to be performed only when a continuous attribute is selected for the test at a node. Figure 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 [15]). 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 dependence exists.

Figure 16 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 slots). 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

DOI: 10.1002/cpe

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 Figure 16 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 the WS policy is to re-arrange tasks that have already been scheduled to workers on the basis of the their weights (e.g., via priority-based Multiple Producer Multiple Consumer (MPMC) concurrent queue or by using work-stealing strategies). However, because 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 Figure 17 how often nodes parallelisation has been chosen by the emitter against the attributes parallelisation in the NAP strategy (we recall that the test condition $|\mathcal{T}| < cr^2$ was fixed). Black stripes lines in the figure denote attributes parallelisation choices whereas white stripes denote nodes parallelisation ones. As expected, the former case occurs more often when processing the top part of the decision tree (from left to the right, in the figure).

5.6. *Simultaneous multithreading*

The Nehalem hyperthreaded box may execute 2 threads simultaneously per each of its 8 physical cores. 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. Figure 12 shows the performance obtained by SMT when more than 7 worker threads are used. The speedup gains for the *U.S. Census*, *SyD10M9A*, *Forest Cover* and *p53Mutants* datasets range from 22% to 44%. For the other two datasets, the overprovisioning of threads does not significantly affect the overall performance (the gain is from 4% to 10%). These figures match the expected benefit for this kind of architectures [23], and they confirm the benefit of SMT support for YaDT-FF.

Simultaneous multithreading technology is also beneficial for non-blocking lock-free algorithms, such as ones used in the FastFlow run-time support. On SMT platforms busy-waiting instructions keep busy a processor context (decode unit and register file) but not execution units. This make it possible to run mediator threads, such emitters and collectors, on the same core of worker threads without impairing computation capability of the worker threads. The FastFlow run-time is very aggressive in using available cores: mediator threads are mapped on different cores till there are enough free cores for allocating worker threads, then they are mapped on a separate context of the same physical core hosting a worker thread. This approach both motivates the plotting of speedup against the number of worker threads and the super-linear speedup in the case when only few processors of the platform run worker threads. The assumption is anyway fair with respect to the maximum speedup achieved.

6. PARALLELISATION OF TREE PRUNING AND MEMORY OCCUPATION EVALUATION

6.1. *Parallelisation of decision tree pruning*

It is legitimate to ask ourselves whether the approach presented so far can be replicated to other problems consisting of a top-down visit/building of tree data structures. The answer is positive. We have designed an NP-like parallel version of the phase following decision tree growing that is known as *tree pruning*. Decision trees are commonly pruned to trade accuracy for simplicity, and to alleviate for the over fitting problem (see [24] for a survey of pruning strategies). The C4.5 system adopts a post-processing *error-based pruning* (EBP) strategy, which, like the tree growing algorithm, is a standard reference for novel proposals. The strategy consists of a non-trivial bottom-up visit of the decision tree, with an inner recursion visit of sub-trees – a form of doubly-recursive visit. See [25] for a computational complexity analysis of EBP.

Traditionally, efficiency of the pruning phase has not been a major concern, because pruning was by far computationally less expensive than tree growing. Research in sequential optimisation and

DOI: 10.1002/cpe

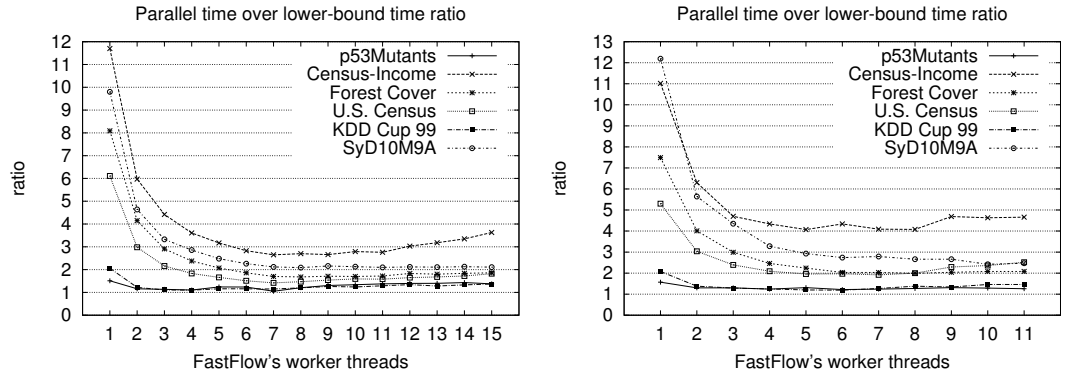

Figure 18. Ratio of elapsed time over lower-bound time for the tree pruning phase. Nehalem box (left), Magny-Cours box (right).

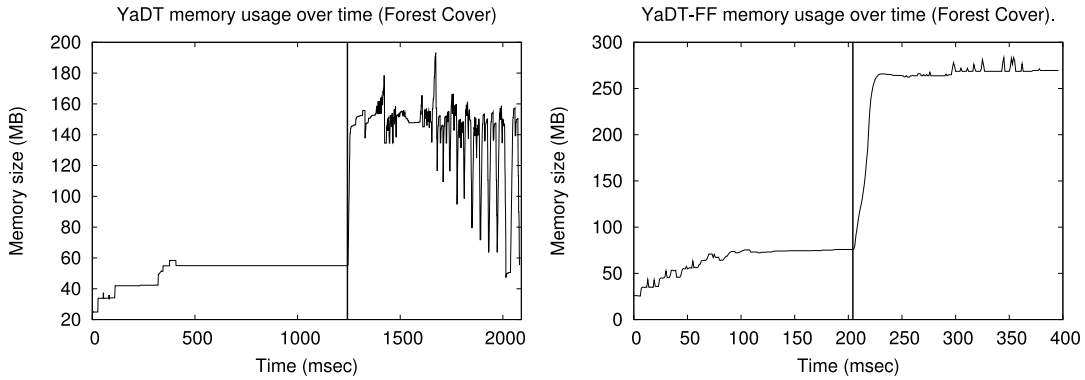

Figure 19. Memory usage over time on Nehalem box with 7 workers for YaDT (left) vs YaDT-FF (right) for dataset *Forest Cover*. The vertical lines mark the end of the growing phase and the start of the pruning phase.

parallelisation of the growing phase, however, has lead to the nowadays situation where the pruning time is in the order of magnitude of the growing time. For instance, the speedup figures of Figure 12 lead the elapsed times of tree growing in the order of magnitude of the tree pruning phase. To the best of our knowledge, however, there is no previous attempt at parallelising the pruning phase in the literature. Our NP-like implementation of the parallelisation is described in great detail in [26]. As for the NP strategy, a lower bound has been calculated by instrumenting the sequential code.

The plots in Figure 18 show 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 parallelisation approach proposed in this paper.

6.2. Memory occupation evaluation of decision tree parallelisation

Figure 19 reports memory occupation over time of YaDT and YaDT-FF for both the growing and pruning phases for the *Forest Cover* dataset. Memory footprint has been traced by reading the Linux's /proc/*pid*/statm memory statistic file every 10ms. With reference to the sequential execution of YaDT, we point out that the pruning phase uses considerable more memory compared to the growing phase. This is due to the double recursion visit algorithm of the pruning phase, which requires allocations of cases at a node in the outer visit and at all of its descendants in the inner

DOI: 10.1002/cpe

visit. Moreover, the depth-first implementation of the double recursion causes the high number of interleaved allocation and deallocation operations readily visible in the left-hand-side plot of Figure 19. The parallel execution of YaDT-FF requires some additional memory compared to YaDT. However, the proportions of memory occupation between the growing and the pruning phases are maintained. The memory allocations in the pruning phase are more stable than for YaDT, because tasks are weighted by the number of cases at a node (irrespective of whether they refer to the outer or the inner visit of a node). Let us now discuss why the overall memory occupation of YaDT-FF does not significantly exceeds the one of YaDT. First, we notice that the additional data structures needed by YaDT-FF include: FIFO queues, whose size is fixed (see Figure 4); `ff_task` objects, whose number is bounded by the size of FIFO queues; and additional member variables of the `node` class (e.g., `child_cnt`), whose memory occupation is proportional to the size of the decision tree being built. The remaining allocations consist of selecting cases at nodes starting from the cases at the parent node, for example, at (§5.13) for the NP strategy, and at (§7.24) for NAP strategy. The sequential code maintains the cases of the nodes in the frontier of the breadth first growth of the tree. The parallel code maintains the cases of a frontier in the top down (but not necessarily breadth first) tree growth. Because cases at a node are partitioned among its child nodes (apart from cases with unknown value of the tested attribute which are replicated among child nodes), the overall number of cases at the nodes in the two frontiers is the same, yet the frontier in the parallel version is typically larger. Summarising, the additional memory occupation of YaDT-FF is proportional to the memory occupation of YaDT and to the (fixed) size of the communication data structures.

7. RELATED WORK

There has been a recent blooming interest of the data mining and machine learning communities on algorithms for parallel platforms [27, 28, 29, 30, 31]. The recurring problem of designing advanced locking schemas in order to decrease the synchronisation overhead in shared memory machines have been considered in [29, 32]. YaDT-FF overcomes such a problem via the FastFlow lock-free synchronisation mechanisms: all synchronisations happen asynchronously in the emitter as the result of data-dependencies among tasks. In the following, we cluster related work on parallelisation of decision trees according to the *task*, *data*, or *hybrid* parallelisation paradigm adopted.

7.1. Task parallelism

Task parallelism consists of splitting the processing of different subtrees into independent tasks in a Divide&Conquer fashion. The NP strategy presented in Section 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* [21], because 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, which is a nontrivial issue because load balancing strategies are typically communication intensive and complex to be integrated in the code. As example, in the early work of [33], the whole training set is replicated in the memory of all the processors, in order to avoid communication of cases across processing nodes. In [34, 35], the Divide&Conquer paradigm is realised in a distributed environment enriched with a virtual shared memory support using a *loop-farm* skeleton, consisting of a farm where the output stream is sent back as input stream (see Figure 1, mid right schema). Compared to the NP strategy, an extra process is used (the collector C in Figure 1) for managing the termination condition. In order to find a suitable trade-off between load balancing and computation-to-communication ratio, workers expand not a single node but up to a given number of levels of a sub-tree. Finally, the NP strategy adopts a tailored scheduling policy, whilst their approach is based on a on-demand policy. In [36], a Pthread-based Divide&Conquer parallelisation on a shared-memory architecture is proposed. 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

DOI: 10.1002/cpe

is 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 on the elapsed time of any task parallelisation strategy, and shown that the NP performances are very close to such a bound.

7.2. Data parallelism

Data parallelism consists in distributing the training set among the processors by partitioning attributes or cases [37].

In vertical data partitioning, each processor computes the gain calculations for a subset of the attributes, for example, in distributed implementation, for the attributes assigned at the processor node [38]. Gain calculations are then exchanged between nodes to determine the best split. This solution suffers both from load balancing problems, because 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 test condition, 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 Figure 2), and, for distributed implementations, from high communication costs. Horizontal data distribution is exploited in the SPRINT classifier [39], which stores the training set according to the SLIQ layout [40]. The authors show a superior performance with respect to vertical data distributions in the direct parallelisation of the SLIQ sequential classifier. ScalParC [41] improves on SPRINT by adopting a distributed hash table that mitigates the communication cost problems. Concerning shared-memory machines, [22] proposes 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. Also, a hybrid version was considered. All versions are implemented on top of a master-worker infrastructure, which is similar to one used in YaDT-FF, thus also exploiting pipeline parallelism between the master and workers. However, synchronisations between master and workers occur 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 in [42], together with a deep investigation of the impact of data locality and cache misses. The RainForest sequential algorithm [17] has been parallelised in [43]. The approach builds a decision tree by levels, with a horizontal data partitioning for computing aggregate values needed in the gain calculation. Synchronisation occurs in order to sum up the partial aggregates and to choose the best split.

7.3. Hybrid task and data parallelism

Hybrid approaches have been explored as a means to control the communication overhead. In the *hybrid parallel formulation* [21] and in pCLOUDS [44], 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 lower levels. In [34], tasks are categorised 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, is inspired by the two latter works and distinguish from them because it does not need the redesign of the sequential algorithm but rather an *easy-yet-efficient* porting of the existing code; it targets multi-core rather than distributed memory machines; it adopts an effective test condition 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

DOI: 10.1002/cpe

mentioned works, but they are temporally inter-weaved and executed on the same stream parallel infrastructure.

8. CONCLUSIONS

The shift of hardware vendors towards multi-core computing requires re-thinking the design of applications to squeeze the real machine power. In this paper, we presented an in-depth study of a decision tree growing algorithm by porting YaDT, an implementation of C4.5, to multi-core using the FastFlow parallel programming framework. Our implementation required *minimal changes of the original sequential code*, yet using non-trivial hybrid task-data parallelisation strategies. This is a first relevant contribution of our approach, since human productivity, total cost and time to solution are important metrics in software development, as equally important as MIPS, FLOPS and speedup. Another major novel contribution is the *characterisation of elapsed time lower bounds* for the forms of parallelisation adopted, showing that our approach achieves close to optimal performance. Despite a large body of related work on the parallelisation of decision tree growing on distributed systems, such a lower bound analysis has not been previously considered. Finally, our proposed approach has been replicated in tackling the *parallelisation of the decision tree pruning phase*, which is another totally novel contribution of this paper.

ACKNOWLEDGEMENTS

This work has been partially supported by EC-STREP ParaPhrase project n. 288570. We thank the Competence Center Gateway for HPC of the IT Center, University of Pisa, for the use of the Magny-Cours box.

REFERENCES

1. Park I, Voss MJ, Kim SW, Eigenmann R. Parallel programming environment for OpenMP. *Scientific Programming* 2001; **9**:143–161.
2. Intel Corp. *Threading Building Blocks* 2013. URL http://www.threadingbuildingblocks.org/.
3. Quinlan JR. *C4.5: Programs for Machine Learning*. Morgan Kaufmann: San Mateo, CA, 1993.
4. Wu X, Kumar V, Quinlan JR, Ghosh J, Yang Q, Motoda H, McLachlan GJ, Ng A, Liu B, Yu PS, Zhou Z-H, Steinbach M, Hand DJ, Steinberg D. Top 10 algorithms in data mining. *Knowledge and Information Systems* 2008; **14**(1):1–37.
5. Ruggieri S. YaDT: Yet another Decision tree Builder. *Proc. of Intl. Conf. on Tools with Artificial Intelligence (ICTAI 2004)*, IEEE, 2004; 260–265.
6. Schapire RE, Freund Y. *Boosting: Foundations and Algorithms*. The MIT Press Cambridge, Massachusetts, London, England, 2012.
7. Pham NK, Do TN, Poulet F, Morin A. Treeview, exploration interactive des arbres de décision. *Revue d'Intelligence Artificielle* 2008; **22**(3-4):473–487.
8. Aldinucci M, Danelutto M, Kilpatrick P, Torquati M. Fastflow: high-level and efficient streaming on multi-core. *Programming Multi-core and Many-core Computing Systems*, Pllana S, Xhafa F (eds.). chap. 13, Parallel and Distributed Computing, Wiley, 2013.
9. Asanovic K, Bodik R, Demmel J, Keaveny T, Keutzer K, Kubiatowicz J, Morgan N, Patterson D, Sen K, Wawrzynek J, Wessel D, Yelick K. A view of the parallel computing landscape. *CACM* 2009; **52**(10):56–67.
10. Aldinucci M, Ruggieri S, Torquati M. Porting decision tree algorithms to multicore using FastFlow. *Proc. of European Conf. on Machine Learning and Knowledge Discovery in Databases (ECML/PKDD 2010)*, *LNCS*, vol. 6321, Springer-Verlag Berlin Heidelberg, 2010; 7–23.
11. Aldinucci M, Campa S, Kilpatrick P, Torquati M. Structured data access annotations for massively parallel computations. *Euro-Par 2012 Workshops, Proc. of the ParaPhrase Workshop on Parallel Processing*, *LNCS*, vol. 7640, Springer-Verlag Berlin Heidelberg, 2013; 381–390.
12. Aldinucci M, Danelutto M, Kilpatrick P, Meneghin M, Torquati M. An efficient unbounded lock-free queue for multi-core systems. *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing*, *LNCS*, vol. 7484, Springer: Rhodes Island, Greece, 2012; 662–673.
13. Hadoop. *Web Page*. Apache Software Foundation 2013. URL http://hadoop.apache.org/.
14. *FastFlow website* 2013. URL http://mc-fastflow.sourceforge.net.

DOI: 10.1002/cpe

15. Aldinucci M, Meneghin M, Torquati M. Efficient Smith-Waterman on multi-core with Fastflow. *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, Danelutto M, Gross T, Bourgeois J (eds.), IEEE: Pisa, Italy, 2010; 195–199.
16. Fayyad UM, Irani KB. On the handling of continuous-valued attributes in decision tree generation. *Machine Learning* 1992; **8**:87–102.
17. Gehrke JE, Ramakrishnan R, Ganti V. RainForest — A framework for fast decision tree construction of large datasets. *Data Mining and Knowledge Discovery* 2000; **4**(2/4):127–162.
18. Ruggieri S. Efficient C4.5. *IEEE Transactions on Knowledge and Data Engineering* 2002; **14**:438–444.
19. Frank A, Asuncion A. UCI machine learning repository 2013. URL http://archive.ics.uci.edu/ml.
20. IBM Almaden. Quest synthetic data generation code 2013. URL http://sourceforge.net/projects/ibmquestdatagen.
21. Srivastava A, Han EH, Kumar V, Singh V. Parallel formulations of decision-tree classification algorithms. *Data Minining & Knowledge Discovery* 1999; **3**(3):237–261.
22. Zaki M, Ho CT, Agrawal R. Parallel classification for data mining on shared-memory multiprocessors. *Proc. of Intl. Conf. on Data Engineering (ICDE 1999)*, IEEE, 1999; 198–205.
23. Sodan AC, Machina J, Deshmeh A, Macnaughton K, Esbaugh B. Parallelism via multithreaded and multicore CPUs. *IEEE Computer* 2010; **43**(3):24–32.
24. Esposito F, Malerba D, Semeraro G. A comparative analysis of methods for pruning decision trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 1997; **19**(5):476–491.
25. Ruggieri S. Subtree replacement in decision tree simplification. *Proc. of SIAM Intl. Conf. on Data Mining (SDM 2012)*, SIAM, 2012; 379–390.
26. Aldinucci M, Ruggieri S, Torquati M. Porting decision tree building and pruning algorithms to multicore using Fastflow. *Technical Report TR-11-06*, Dipartimento di Informatica, Università di Pisa 2011.
27. Choudhary AN, Kumar P, B BO, Misra S, Memik G. Accelerating data mining workloads: Current approaches and future challenges in system architecture design. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 2011; **1**:41–54.
28. Chu CT, Kim SK, Lin YA, Yu Y, Bradski GR, Ng AY, Olukotun K. Map-reduce for machine learning on multicore. *Proc. of the Annual Conference on Neural Information Processing Systems (NIPS 2006)*, MIT Press, 2006; 281–288.
29. Jin R, Yang G, Agrawal G. Shared memory parallelization of data mining algorithms: Techniques, programming interface, and performance. *IEEE Transactions on Knowledge and Data Engineering* 2005; **17**:71–89.
30. Stahl F, Bramer M. Scaling up classification rule induction through parallel processing. *The Knowledge Engineering Review* 2 2013; **FirstView**:1–28.
31. Ericson K, Pallickara S. On the performance of high dimensional data clustering and classification algorithms. *Future Generation Comp. Syst.* 2013; **29**(4):1024–1034.
32. Ravi VT, Agrawal G. Performance issues in parallelizing data-intensive applications on a multi-core cluster. *Proc. of Intl. Symposium on Cluster Computing and the Grid (CCGRID 2009)*, IEEE Computer Society, 2009; 308–315.
33. Darlington J, Guo Y, Sutiwaraphun J, To HW. Parallel induction algorithms for data mining. *Proc. of 2nd Intl. Symposium on Advances in Intelligent Data Analysis (IDA)*, *LNCS*, vol. 1280, Springer-Verlag Berlin Heidelberg, 1997; 437–445.
34. Coppola M, Vanneschi M. High-performance data mining with skeleton-based structured parallel programming. *Parallel Computing* 2002; **28**(5):793–813.
35. Becuzzi P, Coppola M, Ruggieri S, Vanneschi M. Parallelisation of C4.5 as a particular divide and conquer computation. *IPDPS Workshops*, *LNCS*, vol. 1800, Springer-Verlag Berlin Heidelberg, 2000; 382–389.
36. Narlikar GJ. A parallel, multithreaded decision tree builder. *Technical Report CMU-CS-98-184*, Computer Science Dept., Carnegie Mellon University Dec 1998.
37. Amado N, Gama J, Silva F. Parallel implementation of decision tree learning algorithms. *Progress in Artificial Intelligence*, *LNCS*, vol. 2258. Springer-Verlag London, UK, 2001; 34–52.
38. Freitas AA, Lavington SH. *Mining Very Large Databases with Parallel Processing*. 1st edn., Kluwer Academic Publishers: Norwell, MA, USA, 1997.
39. Shafer JC, Agrawal R, Mehta M. SPRINT: A scalable parallel classifier for data mining. *Proc. of Intl. Conf. on Very Large Databases (VLDB 1996)*, 1996; 544–555.
40. Mehta M, Agrawal R, Rissanen J. SLIQ: A fast scalable classifier for data mining. *Proc. of the Intl. Conference on Extending Database Technology (EDBT 1996)*, *LNCS*, vol. 1057, Springer-Verlag London, UK, 1996.
41. Joshi M, Karypis G, Kumar V. ScalParC: A new scalable and efficient parallel classification algorithm for mining large datasets. *Proc. of Intl. Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP 1998)*, IEEE, 1998; 573–579.
42. Bradford JP, Fortes JAB. Characterization and parallelization of decision-tree induction. *Journal of Parallel and Distributed Computing* 2001; **61**(3):322–349.
43. Jin R, Agrawal G. Communication and memory efficient parallel decision tree construction. *Proc. of SIAM Intl. Conf. on Data Mining (SDM 2003)*, SIAM, 2003; 119–129.
44. Sreenivas MK, Alsabti K, Ranka S. Parallel out-of-core divide-and-conquer techniques with application to classification trees. *Proc. of Intl. Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (IPPS/SPDP 1999)*, IEEE, 1999; 555–562.