Porting Decision Tree Algorithms to Multicore using FastFlow

Marco Aldinucci
aldinuc@di.unito.it
Computer Science Department
University of Torino Italy

Salvatore Ruggieri,
ruggieri@di.unipi.it
Computer Science Department
University of Pisa Italy

Massimo Torquati
torquati@di.unipi.it
Outline

- Motivations
- Objectives
- Backgrounds
  - The YaDT algorithm
  - The FastFlow programming framework
- YaDT porting on multicore using FastFlow
  - Experimental Results
- Conclusions
Motivations

• Architectural issues:
  • Multicore CPUs are mainstream (we have some laptop with 4 or 6 cores CPU)
  • Clock speed and Instruction Level Parallelism (ILP) are not enough anymore to improve sequential algorithm
  • CPU designers shifted from ILP to Thread Level Parallelism
  • On our multicore workstations, we (shall) have some (many?) “spare” (idle) cores that can be used to improve applications performance.
  • Of course, this applies to data mining software too.
Motivations

Let’s consider the **sequential time** on different CPUs for a same data mining algorithm (YaDT, an implementation of C4.5):

### x86_64 Intel Xeon

<table>
<thead>
<tr>
<th>year</th>
<th>CPU</th>
<th>Caches</th>
<th>Cores/Threads</th>
<th>Time (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006</td>
<td><a href="mailto:E5150@2.6GHz">E5150@2.6GHz</a></td>
<td>4MB L2</td>
<td>2/2</td>
<td>161</td>
</tr>
<tr>
<td>2007</td>
<td><a href="mailto:E5420@2.5GHz">E5420@2.5GHz</a></td>
<td>2x6MB L2</td>
<td>4/4</td>
<td>161</td>
</tr>
<tr>
<td>2009</td>
<td><a href="mailto:E5520@2.2GHz">E5520@2.2GHz</a></td>
<td>4x256KB L2 8M L3</td>
<td>4/8</td>
<td>114</td>
</tr>
<tr>
<td>2010</td>
<td><a href="mailto:E5620@2.4GHz">E5620@2.4GHz</a></td>
<td>4x256KB L2 12MB L3</td>
<td>4/8</td>
<td>112</td>
</tr>
</tbody>
</table>

### AMD64 Opteron

<table>
<thead>
<tr>
<th>year</th>
<th>CPU</th>
<th>Caches</th>
<th>Cores/Threads</th>
<th>Time (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010</td>
<td><a href="mailto:6136@2.4GHz">6136@2.4GHz</a></td>
<td>8x512KB L2 2x6MB L3</td>
<td>8/8</td>
<td>142</td>
</tr>
<tr>
<td>2010</td>
<td><a href="mailto:6174@2.2GHz">6174@2.2GHz</a></td>
<td>8x512KB L2 2x6MB L3</td>
<td>12/12</td>
<td>148</td>
</tr>
</tbody>
</table>

**Optimized sequential code does not get any performance benefit from multicore evolution**

Motivations

- Application issues:
  - There exist many extremely optimised sequential data mining algorithms

<table>
<thead>
<tr>
<th>Dataset name</th>
<th>EC4.5 / C4.5</th>
<th>YaDT / EC4.5</th>
<th>YaDT / C4.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forest Cover</td>
<td>0.31</td>
<td>0.28</td>
<td>0.09</td>
</tr>
<tr>
<td>KDD Cup 99</td>
<td>0.42</td>
<td>0.23</td>
<td>0.10</td>
</tr>
</tbody>
</table>

- YaDT is 10 times faster than C4.5 with only 1/3 of its memory occupation
- It seems not realistic to design any further (significant)
Objectives

- The only viable solution to improve performance is to parallelise or accelerate the sequential code on multicore.
- We present our approach for the *easy-yet-efficient* porting of a highly optimized implementation of C4.5 (YaDT - ICTAI 2004) to multicore.
- We face the problem of achieving a good *trade-off* between performance and human productivity during the porting.
- Preserving as much as possible the original code is one of the main objectives.
- The approach is based on the FastFlow framework.
- We also want to validate the methodological approach offered by the FastFlow framework.
**Backgrounds**

- **FastFlow**: a skeleton-based parallel programming framework
  - General architecture
  - Programming model
  - Sequential code acceleration via FastFlow self-offloading

- **YaDT (Yet another Decision Tree builder)**
  - Design principles
  - Core algorithm pseudo-code
• High-level programming based on two main concepts:
  • **Skeletons** (a.k.a well-known computational patterns)
  • **Stream parallelism** programming paradigm
High-level programming based on two main concepts:
- **Skeletons** (a.k.a well-known computational patterns)
- Stream parallelism programming paradigm
FastFlow

- High-level programming based on two main concepts:
  - **Skeletons** (a.k.a well-known computational patterns)
  - **Stream parallelism** programming paradigm

FastFlow

• High-level programming based on two main concepts:
  • **Skeletons** (a.k.a well-known computational patterns)
  • **Stream parallelism** programming paradigm

FastFlow: design objectives

- High-level programming
  - Provides basic skeletons: pipeline, farm, D&C
  - Allows to build complex streaming networks
  - C++ STL-like implementation

- Efficient support for multicore
  - Lock-free cache friendly point-point communications
  - Lock-free fast and reliable memory allocator

- Streaming
  - Native stream-parallelism support provided as data-flow graph
  - Effective approach for breaking the “memory wall”
FastFlow: streaming-networks

pipeline

farms (master-worker)

farms with feedback (D&C)

Arbitrary data-flow graph (using low-level SPSC queues)

composition example

FastFlow: streaming-networks

- Pipeline
- Farms (master-worker)
- Farms with feedback (D&C)
- Arbitrary data-flow graph (using low-level SPSC queues)

Composition example
FastFlow: software accelerator

- FastFlow can be used as a software accelerator
- Targets the parallelisation of legacy (C/C++) code
- Requires only local intervention in the code
- Uses spare or idle cores
- The user transforms loops and recursive computation in stream of tasks which are offloaded to the accelerator
YaDT algorithm

- YaDT is a from scratch C++ implementation of C4.5 algorithm
  - ICTAI 2004
  - Available from [http://www.di.unipi.it/~ruggieri/](http://www.di.unipi.it/~ruggieri/)
- Adds several optimizations
  - On data structures (e.g., dataset maintained by column)
  - On algorithms (e.g., specialised sorting, Fayad & Irani opt.)
  - Object oriented design
  - Breath-first tree growing strategy
- Leading to 10x improvement over C4.5 R8
YaDT algorithm

- The tree growing procedure is a simple D&C computation

```cpp
void tree::build() {
    queue<node *> q;
    node * root = new node(allCases);
    q.push(root);
    while( !q.empty() ) {
        node * n = q.front();
        q.pop();
        n->split();
        for(int i=0; i<n->nChilds(); ++i)
            q.push( n->getChild() );
    }
}
```

YaDT algorithm

- The tree growing procedure is a simple D&C computation

 void tree::build() {
    queue<node *> q;
    node * root = new node(allCases);
    q.push(root);
    while( !q.empty() ) {
        node * n = q.front();
        q.pop();
        n->split();
        for(int i=0;i<n->nChilds();++i)
            q.push( n->getChild() );
    }
}
YaDT algorithm

- node splitting procedure

```cpp
void node::split() {
    computeFrequencies();
    if (onlyOneClass() || fewCases())
        set_as_leaf();
    else {
        for(int i=0;i<getNoAtts();++i)
            gain[i]=gainCalculation(i);
        best=argmax(gain);
        if (attr[best].isContinuous())
            findThreashold(best);
        int ns=attr[best].nSplits();
        for(int i=0;i<ns;++i)
            childs.push_back(new node(selectCases(best,i));
    }
}
```

YaDT algorithm

node splitting procedure

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

- Computes the weighted frequency of classes too few cases, it’s a leaf node.
- For each attribute, computes the information gain.
- Selects the attribute with the highest information gain.
- If the attribute is continuous, the threshold of the split is computed over the whole training set.
- Node’s children queueing.
YaDT porting on multicore

- We propose a parallelisation of the YaDT induction algorithm via stream-parallelism using FastFlow
  - Each decision node is considered a task, each node could generates a set of sub-tasks
  - Tasks are arranged in a stream that flows across the FastFlow D&C paradigm (i.e. *farm with feedback channels* skeleton)
- YaDT-FF is the name of the YaDT porting on multicore
- For the parallelisation, we followed a two-phase approach:
  - **Node-Parallelisation (NP strategy)**
    - First we parallelised only the `tree::build` method
  - **Node and Attribute Parallelisation (NAP strategy)**
    - On top of the NP strategy we added the parallelisation of the `node::split` method

YaDT-FF: overall schema

YaDT  
main thread

tree::build

YaDT-FF  
main thread

tree::build_ff

YaDT-FF: overall schema

```
struct ff_task {
    decision_tree *node;
    task_kind wt;
    unsigned numattrs;
    unsigned weight;
    unsigned worker_idx;
}
```
YaDT-FF: overall schema

```
struct ff_task {
    decision_tree *node;
    task_kind wt;
    unsigned numattrs;
    unsigned weight;
    unsigned worker_idx;
}
```
YaDT-FF: overall schema

```
struct ff_task {
    decision_tree *node;
    task_kind wt;
    unsigned numattrs;
    unsigned weight;
    unsigned worker_idx;
};
```
YaDT-FF: overall schema

YaDT

main thread

tree::build

YaDT-FF

main thread

tree::build_ff

struct ff_task {
  decision_tree *node;
  task_kind wt;
  unsigned numattrs;
  unsigned weight;
  unsigned worker_idx;
}

YaDT-FF: overall schema

YaDT

main thread

tree::build

YaDT-FF

main thread

tree::build_ff

... and so on in a D&C fashion

struct ff_task {
  decision_tree *node;
  task_kind wt;
  unsigned numattrs;
  unsigned weight;
  unsigned worker_idx;
}
YaDT-FF: D&C setup

```cpp
void tree::build_ff() {
    node * root = new node(allCases);
    ff_emitter * E = new ff_emitter(root, PAR_DEGREE);
    std::vector<ff_worker*> workers;
    for(int i=0; i<PAR_DEGREE; ++i)
        workers.push_back( new ff_worker() );
    ff_farm<ws_scheduling> farm(PAR_DEGREE*QSIZE);
    farm.add_workers(workers);
    farm.add_emitter(E);
    farm.wrap_around();
    farm.run_and_wait_end();
}
```
YaDT-FF: D&C setup

```cpp
void tree::build_ff() {
    node * root = new node(allCases);
    ff_emitter * E = new ff_emitter(root, PAR_DEGREE);
    std::vector<ff_worker*> workers;
    for(int i=0; i<PAR_DEGREE; ++i)
        workers.push_back(new ff_worker());
    ff_farm<ws_scheduling> farm(PAR_DEGREE*QSIZE);
    farm.add_workers(workers);
    farm.add_emitter(E);
    farm.wrap_around();
    farm.run_and_wait_end();
}
```

- The setup is the same for both parallelisation strategies, what changes is the class `ff_emitter` and `ff_worker` (i.e. the `svc` method)
- Different task-scheduling policies can be implemented changing only the `ws_scheduling` class
YaDT-FF: NP strategy (Emitter/Worker)

```c
void * ff_emitter::svc(void *task) {
    if (!task) {
        task=new ff_task(root,BUILD_NODE);
        setWeight(task, root->getNoCases());
        return task;
    }
    node *n= task->getNode();
    if (noMoreTasks() && !n->nChilds()) return NULL;
    for(int i=0;i<n->nChilds();++i) {
        node *child=n->getChild(i);
        newtask=new ff_task(child,BUILD_NODE);
        setWeight(newtask, child->getNoCases());
        ff_send_out(newtask);
    }
    return FF_GO_ON;
}

void * ff_worker::svc(void *task) {
    (task->getNode())->split();
    return task;
}
```

YaDT-FF: NP strategy (Emitter/Worker)

```c
void * ff_emitter::svc(void *task) {
    if (!task) {
        task=new ff_task(root,BUILD_NODE);
        setWeight(task, root->getNoCases());
        return task;
    }
    node *n= task->getNode();
    if (noMoreTasks() && !n->nChilds()) return NULL;
    for(int i=0;i<n->nChilds();++i) {
        node *child=n->getChild(i);
        newtask=new ff_task(child,BUILD_NODE);
        setWeight(newtask, child->getNoCases());
        ff_send_out(newtask);
    }
    return FF_GO_ON;
}

void * ff_worker::svc(void *task) {
    (task->getNode())->split();
    return task;
}
```

at the beginning the task is NULL
termination condition
FF run-time calls, this schedules the task toward one of the Workers
Tells the FF run-time to check the input queue for another task
YaDT-FF: NAP strategy

- In order to improve the performance, we parallelise the for-loop in the `node::split` procedure (NAP builds over the NP strategy).
- We divide the original method in 3 distinct methods in order to perform the information gain computation phase in parallel.

```cpp
void node::split() {
    computeFrequencies();
    if (onlyOneClass() || fewCases())
        set_as_leaf();
    else {
        for(int i=0;i<getNoAtts();++i)
            gain[i]=gainCalculation(i);
        best=argmax(gain);
        if (attr[best].isContinuous())
            findThreshhold(best);
        int ns=attr[best].nSplits();
        for(int i=0;i<ns;++i)
            childs.push_back(new node(selectCases(best,i));
    }
```

YaDT-FF: NAP strategy

- In order to improve the performance, we parallelise the for-loop in the `node::split` procedure (NAP builds over the NP strategy).
- We divide the original method in 3 distinct methods in order to perform the information gain computation phase in parallel.

```cpp
void node::split() {
    computeFrequencies();
    if (onlyOneClass() || fewCases())
        set_as_leaf();
    else {
        for(int i=0;i<getNoAtts();++i)
            gain[i]=gainCalculation(i);
        best=argmax(gain);
        if (attr[best].isContinuous())
            findThreashold(best);
        int ns=attr[best].nSplits();
        for(int i=0;i<ns;++i)
            childs.push_back(new node(selectCases(best,i));
    }
}
```

(splitPre(), splitAtt(int idx), splitPost())
YaDT-FF: NAP strategy (Emitter 1/2)

```c
void * ff_emitter::svc(void *task) {
    if (!task) {
        if (root->splitPre()) return NULL;
        r = root->getNoCases(); c = root->getNoAtts();
        for (int i = 0; i < c; ++i) {
            task = new ff_task(root, BUILD_ATT);
            task->att = i; setWeight(task, r);
            ff_send_out(task);
        }
        root->attTask = c;
        return FF_GO_ON;
    }
    node *n = task->getNode();
    if (task->isBuildAtt()) {
        if (--n->attTask > 0) return FF_GO_ON;
        n->splitPost();
    }
    if (noMoreTasks() && !n->nChilds()) return NULL;
}
```

(continue...)
YaDT-FF: NAP strategy (Emitter 1/2)

```c
void * ff_emitter::svc(void *task) {
    if (!task) {
        if (root->splitPre()) return NULL;
        r = root->getNoCases(); c = root->getNoAtts();
        for (int i = 0; i < c; ++i) {
            task = new ff_task(root, BUILD_ATT);
            task->att = i; setWeight(task, r);
            ff_send_out(task);
        }
        root->attTask = c;
        return FF_GO_ON;
    }
    node *n = task->getNode();
    if (task->isBuildAtt()) {
        if (--n->attTask > 0) return FF_GO_ON;
        n->splitPost();
    }
    if (noMoreTasks() && !n->nChilds()) return NULL;
}
```

At the beginning the task is NULL for the root node, attribute parallelisation is always the case for each attribute, creates an attribute task FF run-time calls, this schedules the task toward one of the Workers remembers how many tasks we have to wait for tells the FF run-time to check the input queue for another task tells if the task is an attribute task we have to wait that all attribute tasks come back all tasks have arrived termination condition

(continue...)

void * **ff_emitter::svc**(void *task) {
    ...
    for(int i=0; i<n->nChilds(); ++i) {
        node *child=n->getChild(i);
        r = child->getNoCases(); c = child->getNoAtts();
        if (!buildAttTest(r, c)) {
            newtask = new ff_task(child, BUILD_NODE);
            setWeight(newtask, r); **ff_send_out**(newtask);
        } else {
            if (child->splitPre()) continue;
            for(int j=0; i<c; ++j) {
                newtask = new ff_task(child, BUILD_ATT);
                newtask->att = j; setWeight(newtask, r);
                **ff_send_out**(newtask);
            }
            child->attTask = c;
        }
    }
    return **FF_GO_ON**;
}
YaDT-FF: NAP strategy  

for each childs rooted in the node

decides if to apply nodes or attributes parallelisation using a cost model

creates a node task

for each attribute, creates an attribute task

sets the weight of the task for the weighted scheduling policy

void * ff_emitter::svc(void *task) {
    ...
    for(int i=0;i<n->nChilds();++i) {
        node *child=n->getChild(i);
        r= child->getNoCases(); c=child->getNoAtts();
        if (!buildAttTest(r,c)) {
            newtask=new ff_task(child,BUILD_NODE);
            setWeight(newtask, r); ff_send_out(newtask);
        } else {
            if (child->splitPre()) continue;
            for(int j=0;i<c;++j) {
                newtask=new ff_task(child,BUILD_ATT);
                newtask->att=j; setWeight(newtask, r);
                ff_send_out(newtask);
            }
            child->attTask=c;
        }
    }
    return FF_GO_ON;
}
YaDT-FF: NAP strategy (Worker)

- The worker implementation is straightforward

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

YaDT-FF: NAP strategy (Worker)

- The worker implementation is straightforward

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

- Computes information gain on one attribute
- Computes information gain on all attributes (as in the NP strategy)
YaDT-FF: node vs attr. choice

- The test function `buildAttTest` decides whether to perform nodes or attributes parallelisation on the base of a cost model.
- We considered 3 cost models, i.e. we use attributes parallelisation when:
  1. $\alpha < r$  
     The number of cases is above some hand-tuned threshold
  2. $|T| < cr \log r$  
     Node’s average processing grain is higher then a threshold
  3. $|T| < cr^2$  
     Node’s worst-case processing grain is higher then a threshold

- Nodes (white stripes) vs. attributes (black stripes) parallelisation choices for the cost model 3 (the best in the experiments):

```
<table>
<thead>
<tr>
<th>10k</th>
<th>40k</th>
<th>70k</th>
<th>100k</th>
</tr>
</thead>
<tbody>
<tr>
<td>Census PUMS</td>
<td>SyD10M9A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Training sets used in experiments

- Several **standard training sets** tested

<table>
<thead>
<tr>
<th>name</th>
<th>N. of cases</th>
<th>NC</th>
<th>No. of attributes</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>discr.</td>
<td>contin.</td>
</tr>
<tr>
<td>Census PUMS</td>
<td>299,285</td>
<td>2</td>
<td>33</td>
<td>7</td>
</tr>
<tr>
<td>U.S. Census</td>
<td>2,458,285</td>
<td>5</td>
<td>67</td>
<td>0</td>
</tr>
<tr>
<td>KDD Cup 99</td>
<td>4,898,431</td>
<td>23</td>
<td>7</td>
<td>34</td>
</tr>
<tr>
<td>Forest Cover</td>
<td>581,012</td>
<td>7</td>
<td>44</td>
<td>10</td>
</tr>
<tr>
<td>SyD10M9A</td>
<td>10,000,000</td>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>
YaDT-FF: NP strategy speedup

- **Moderate speedup**, there’s not enough parallelism to go round
- The performance/cost ratio is very high because of minimal changes to the sequential code
- The approach is generally applicable to top down tree-growing algs.

The NAP strategy obtains a **good speedup** ranging from 4 (Census PUMS) to 7.5 (Forest Cover) with an efficiency of 93%.

The HyperThreaded box provides 12-30 % performance improvement.
YaDT-FF: NAP strategy speedup

Dual quad-core (16 threads) Xeon E5520@2.2

- Speedup vs. no. of attributes for 1M samples cases from SyD10M9A
- Speedup vs. no. of samples cases from SyD10M9A (right)
YaDT-FF: load-balancing

- The decision of which worker’s queue to allocate a task is always a critical problem in parallelism.
- Simple task scheduling policy leads to load-imbalance among workers due to the difficulties to predict the actual task workload.
- The NAP strategy does not suffer too much of load imbalance due to the high over-provisioning of tasks (nodes and attributes parallelism)
- The designed \textit{weighed scheduling} (WS) policy assigns a new task to the worker with the lowest total weight of tasks in its input queue
- Tasks are weighted with the number of cases at a node
- WS exhibits the best performance
YaDT-FF: execution breakdown

- NAP’s execution breakdown using 7 worker threads
Conclusions

- With **minimal code changes** to the YaDT sequential algorithm performances are significantly improved (up to 7x speedup).
- Our approach uses **node and attributes parallelism** and a weighted problem-aware load balancing technique.
- **FastFlow** offers an effective methodology for the acceleration of recursive sequential algorithms.

- **FastFlow** is open-source (LGPLv3) available at:  
Thank you.