Porting Decision Tree Algorithms to Multicore using FastFlow

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

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

Outline

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

Motivations 1/3

- 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 2/3

- 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

year	CPU	Caches	Cores/Threads	Time (S)
2006	E5150@2.6GHz	4MB L2	2/2	**161**
2007	E5420@2.5GHz	2x6MB L2	4/4	**161**
2009	E5520@2.2GHz	4x256KB L2 8M L3	4/8	**114**
2010	E5620@2.4GHz	4x256KB L2 12MB L3	4/8	**112**

AMD64 Opteron

year	CPU	Caches	Cores/Threads	Time (S)
2010	6136@2.4GHz	8x512KB L2 2x6MB L3	8/8	**142**
2010	6174@2.2GHz	8x512KB L2 2x6MB L3	12/12	**148**

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

Motivations 3/3

- Application issues:
  - There exist many extremely optimised sequential data mining algorithms
  - Let's consider YaDT (ICTAI 2004) w.r.t. C4.5 (Quinlan's book 1993) and EC4.5 (TKDE 2002):

Dataset name	Execution Time Ratio		
	EC4.5 / C4.5	**YaDT / EC4.5**	**YaDT / C4.5**
Forest Cover	0.31	0.28	0.09
KDD Cup 99	0.42	0.23	0.10

  - 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

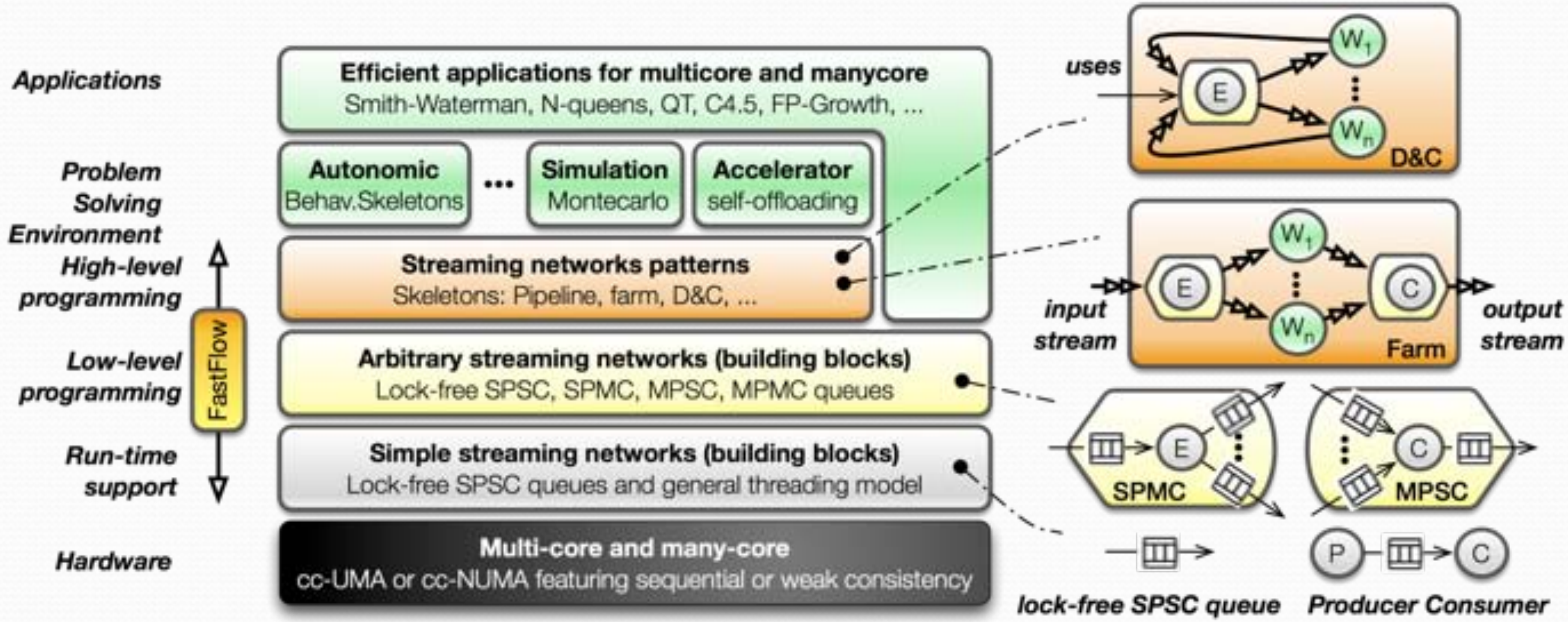
Backgrounds

- 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

FastFlow

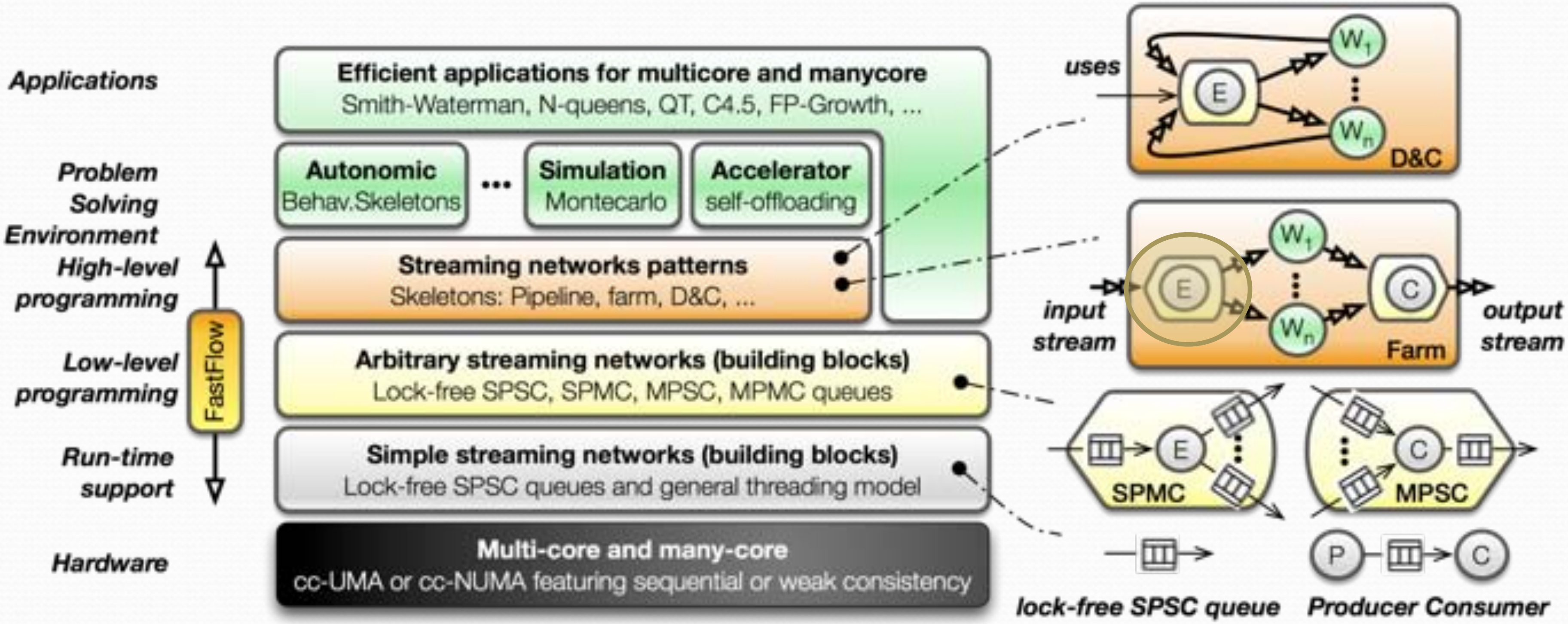
1/4



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

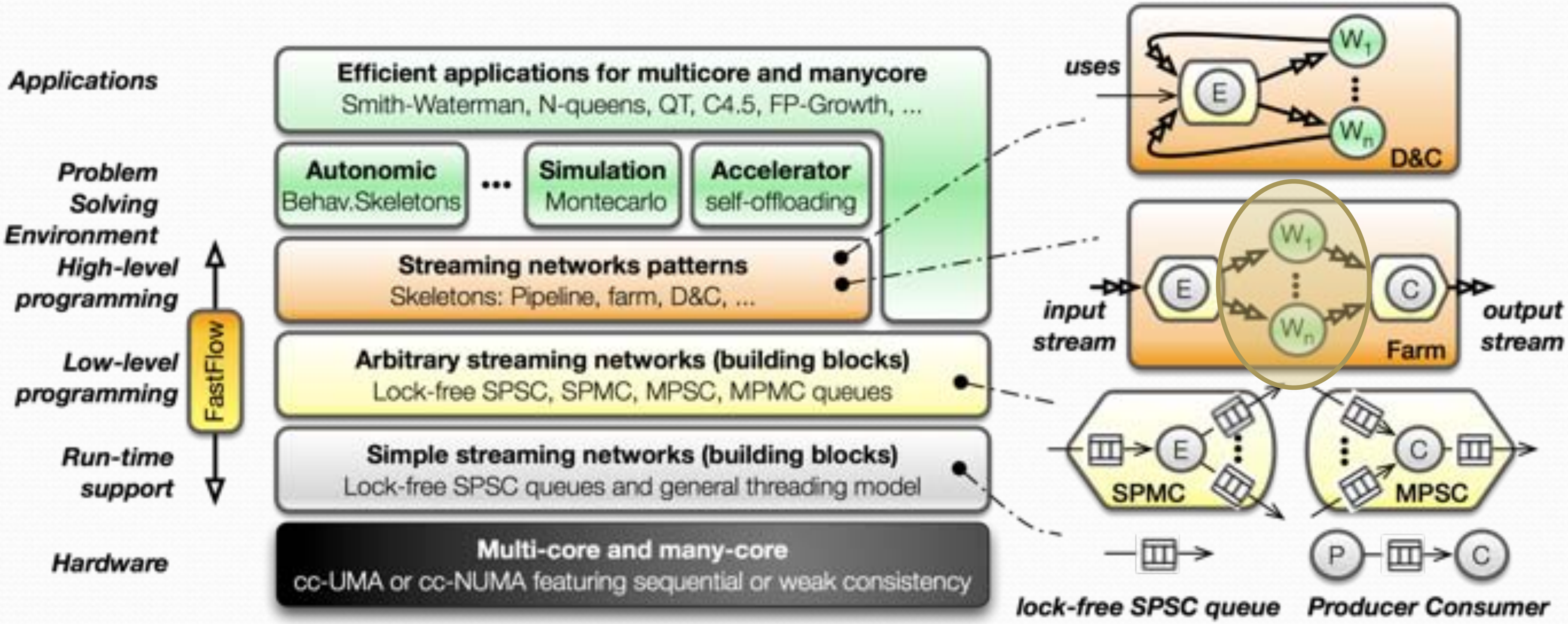
FastFlow

1/4



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

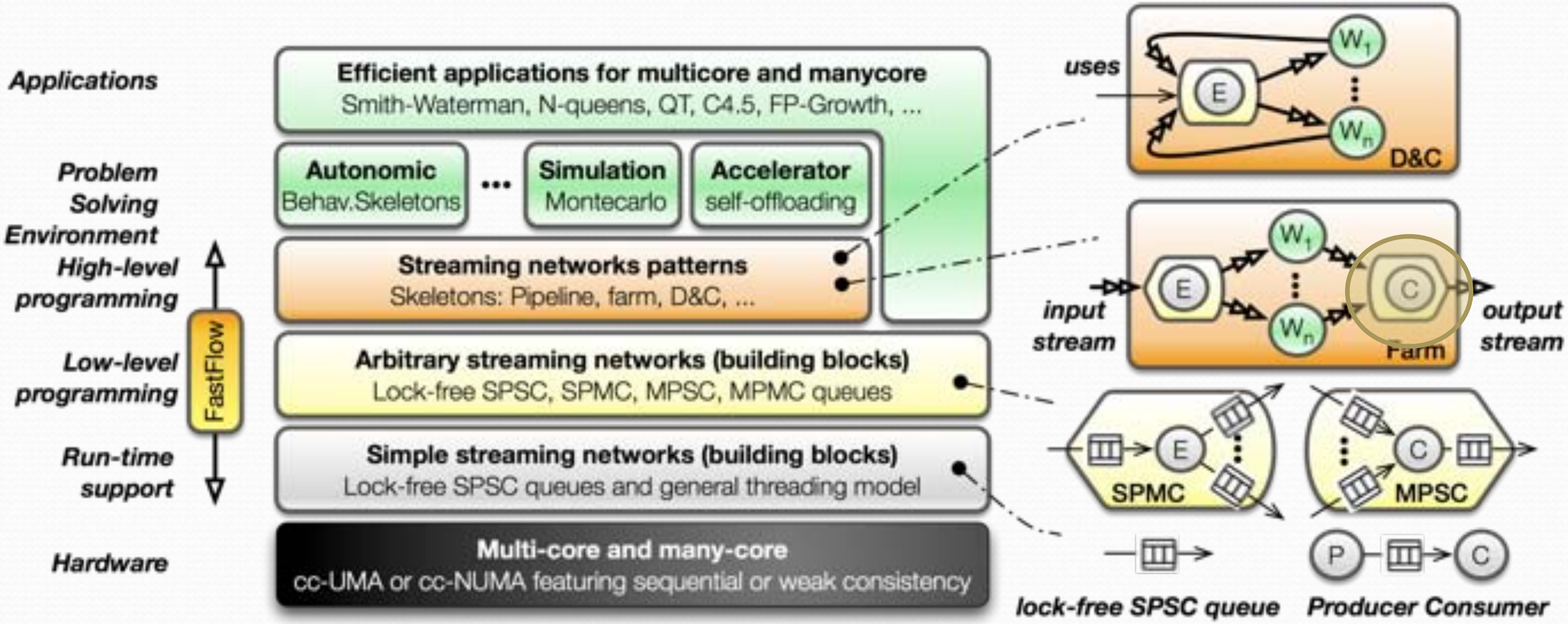
FastFlow



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

FastFlow

1/4

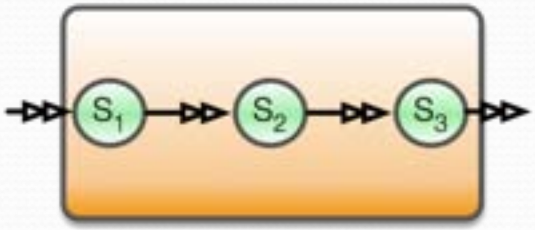


- High-level programming based on two main concepts:
  - Skeletons (a.k.a well-known computational patterns)
  - Stream parallelism programming paradigm

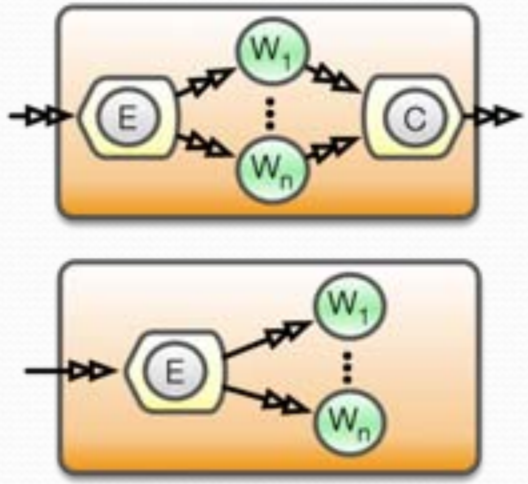
FastFlow: design objectives 2/4

- 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*"

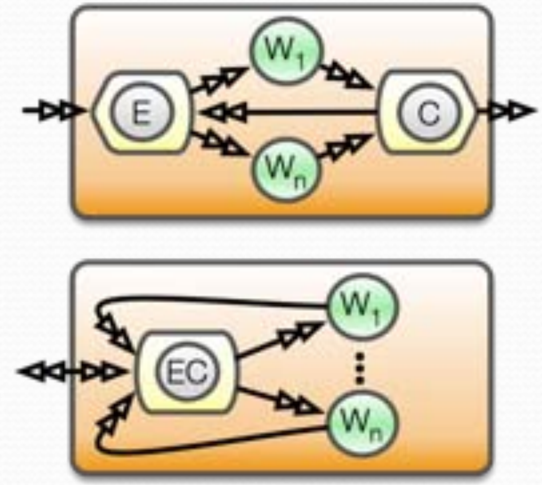
FastFlow: streaming-networks 3/4



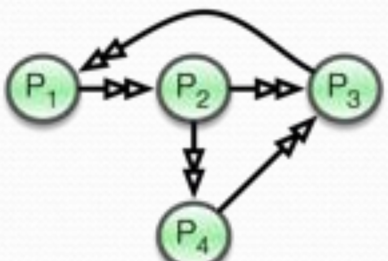
pipeline



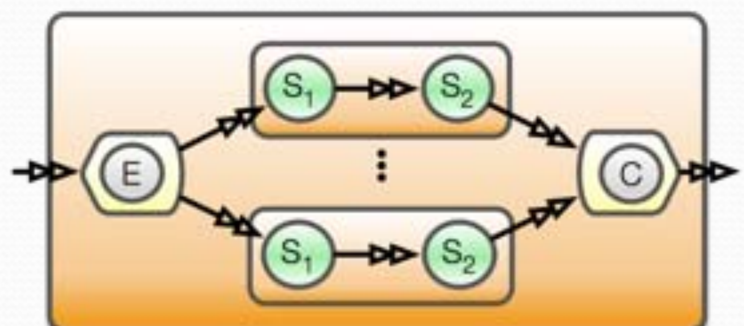
farms (master-worker)



farms with feedback (D&C)

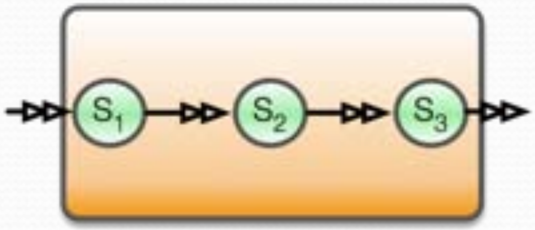


Arbitrary data-flow graph
(using low-level SPSC queues)

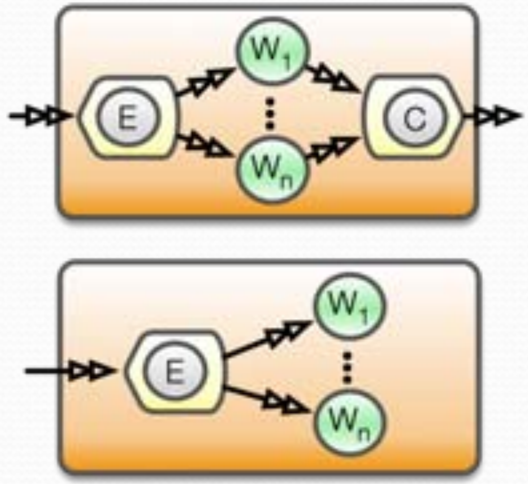


composition example

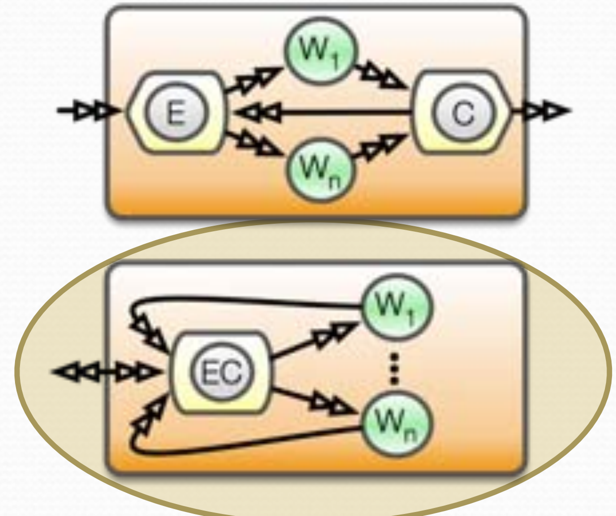
FastFlow: streaming-networks 3/4



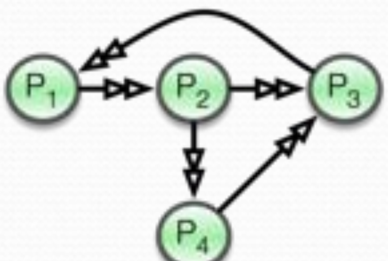
pipeline



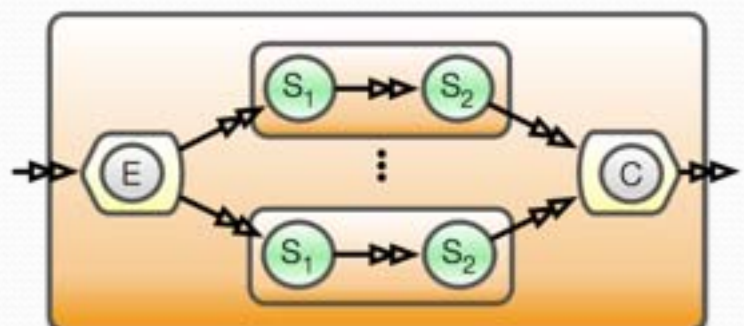
farms (master-worker)



farms with feedback (D&C)

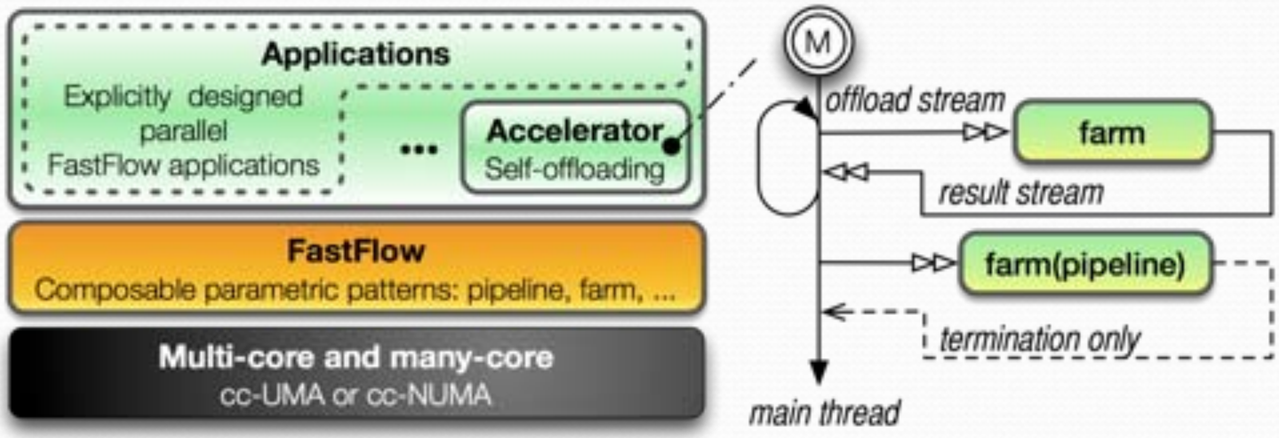


Arbitrary data-flow graph
(using low-level SPSC queues)



composition example

FastFlow: software accelerator 4/4



- 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 1/3

- YaDT is a from scratch C++ implementation of C4.5 algorithm
  - ICTAI 2004
  - Available from 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 2/3

- 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 2/3

- 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() );
  }
}
```

the recursion is implemented using a queue

selects an attribute for splitting

each child node feeds the recursion queue

YaDT algorithm 2/3

- node splitting procedure

```
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())
       findThreashhold(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

```
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())
      findThreashhold(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

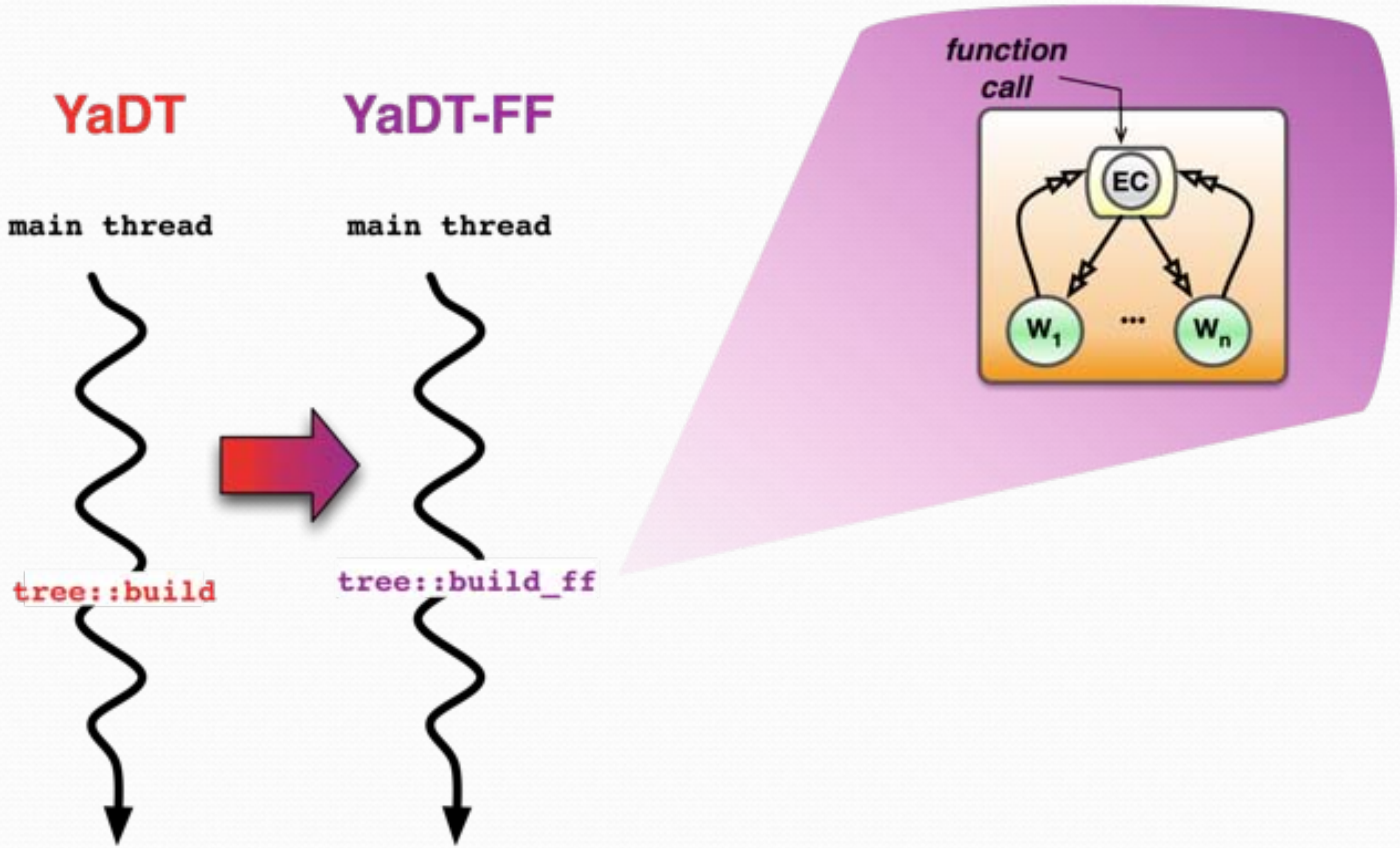
if the attribute is continuous the threshold of the split is computed over the whole training set

node's childs queueing

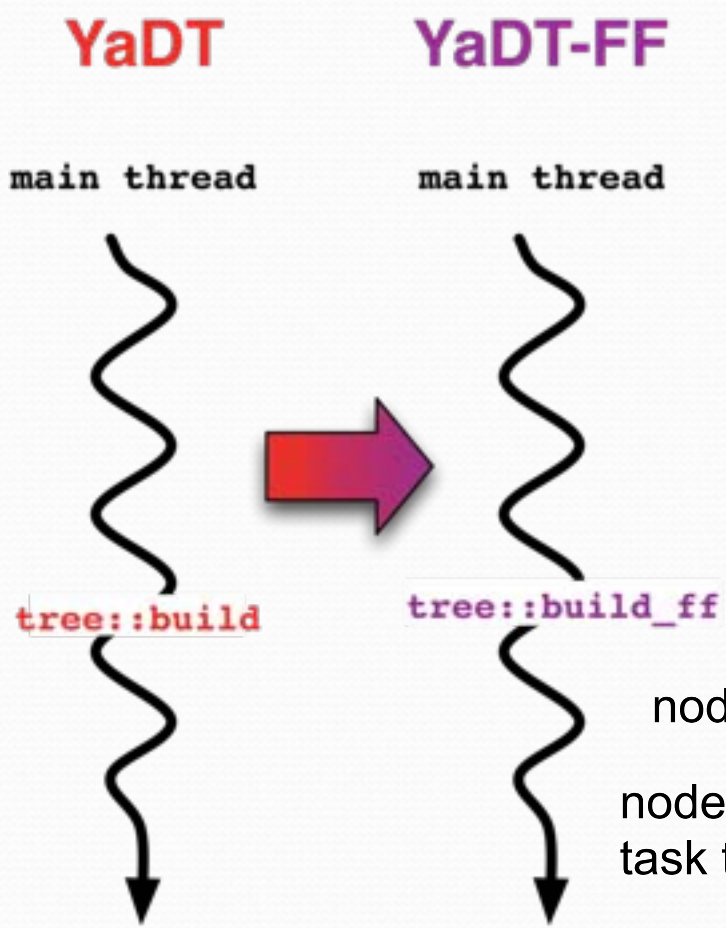
YaDT porting on multicore

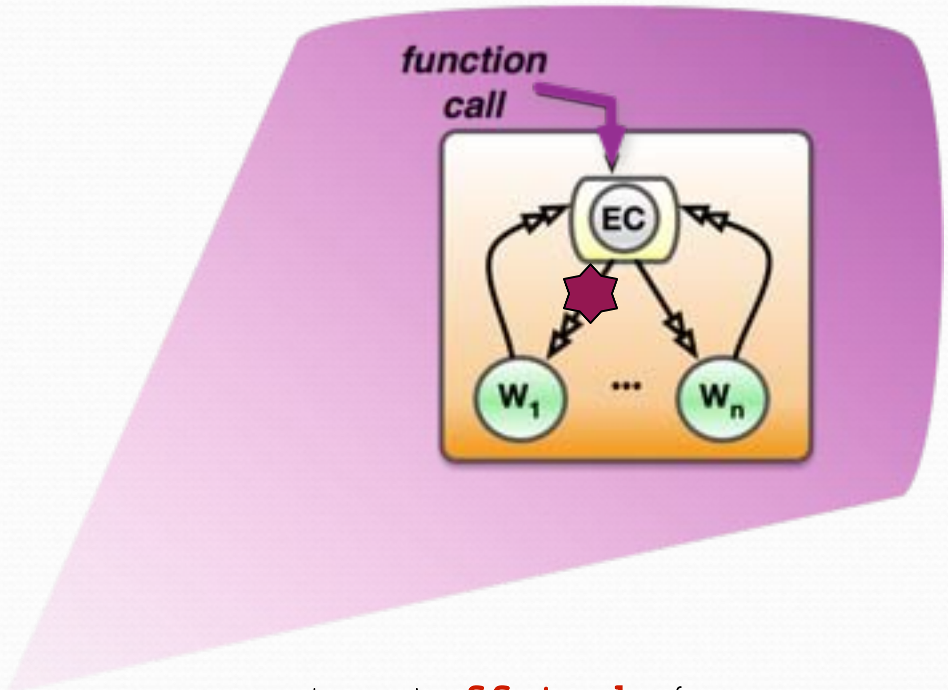
- We propose a parallelisation of the YaDT induction algorithm via stream-parallelism using FastFlow
  - Each decision node is considered a task, each node could generates a set of sub-tasks
  - Tasks are arranged in a stream that flows across the FastFlow D&C paradigm (i.e. *farm with feedback channels*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

YaDT-FF: overall schema



YaDT-FF: overall schema



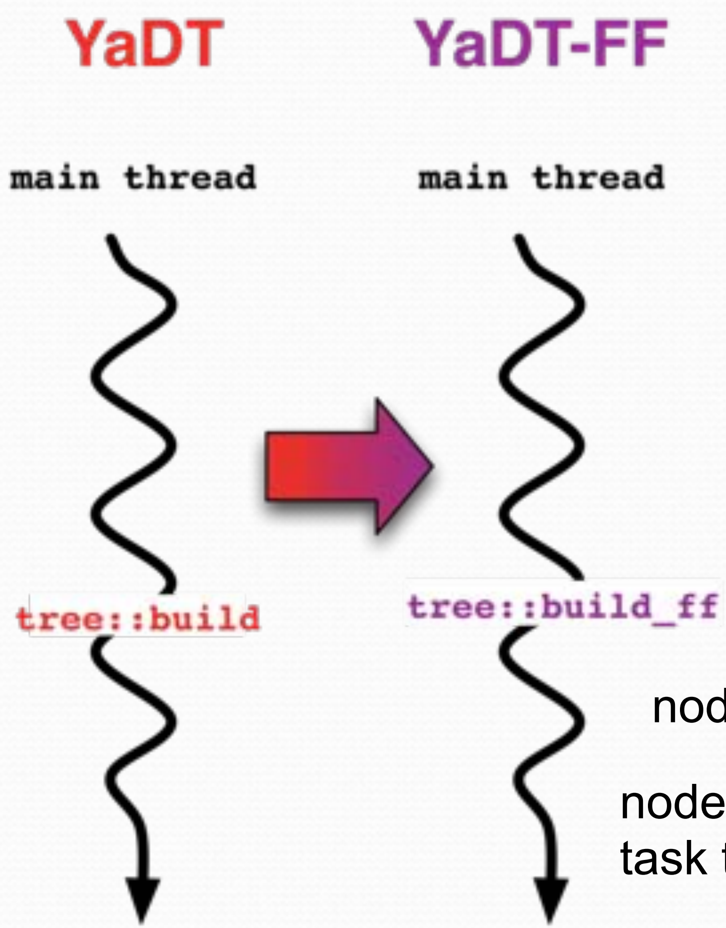


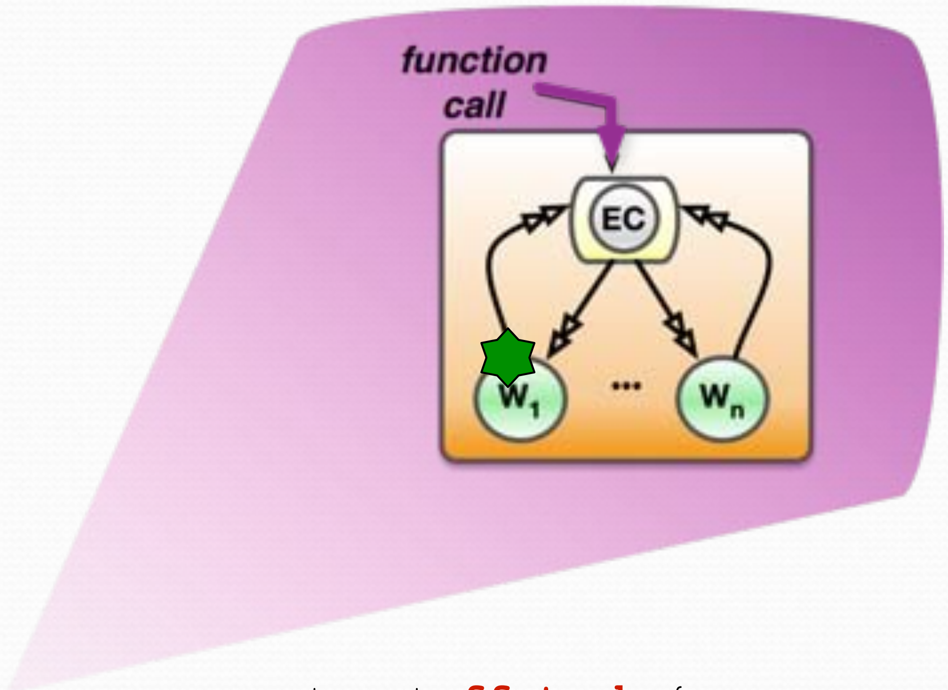
node pointer

node or attribute task tag

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

YaDT-FF: overall schema



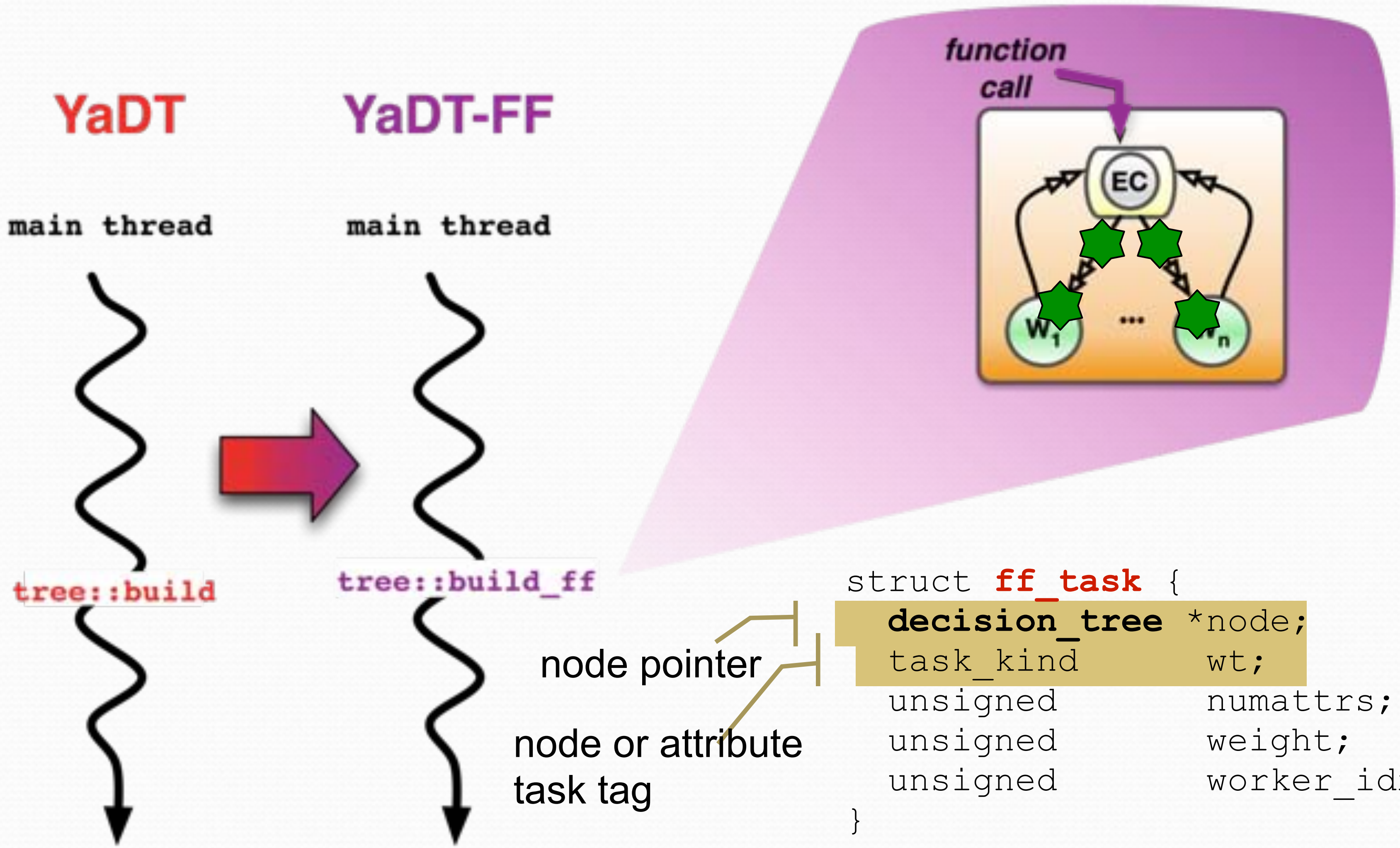



```
struct ff_task {
  decision_tree *node;
  task_kind     wt;
  unsigned      numattrs;
  unsigned      weight;
  unsigned      worker_idx;
}
```

node pointer

node or attribute task tag

YaDT-FF: overall schema

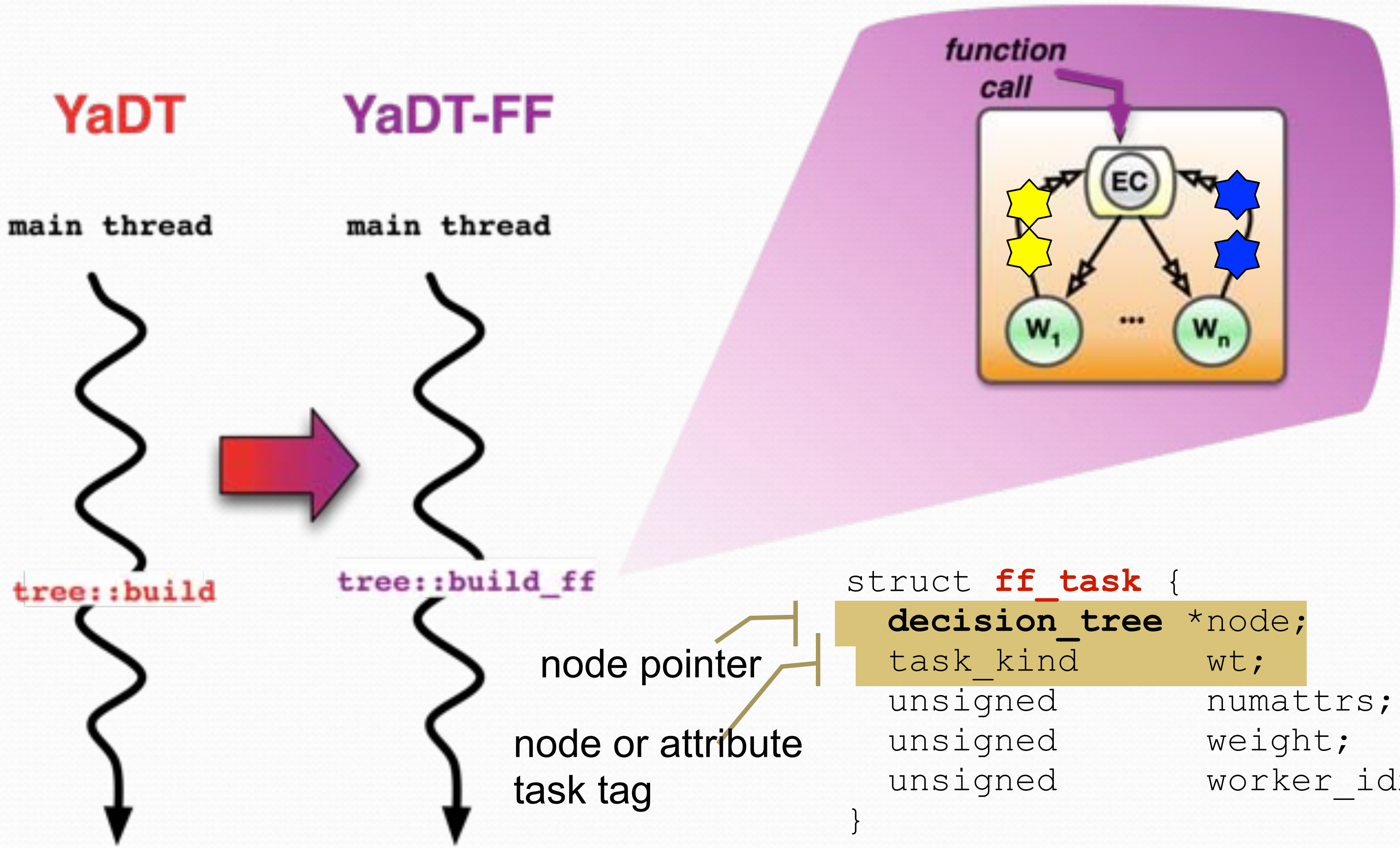



```
struct ff_task {
  decision_tree *node;
  task_kind     wt;
  unsigned      numattrs;
  unsigned      weight;
  unsigned      worker_idx;
}
```

node pointer

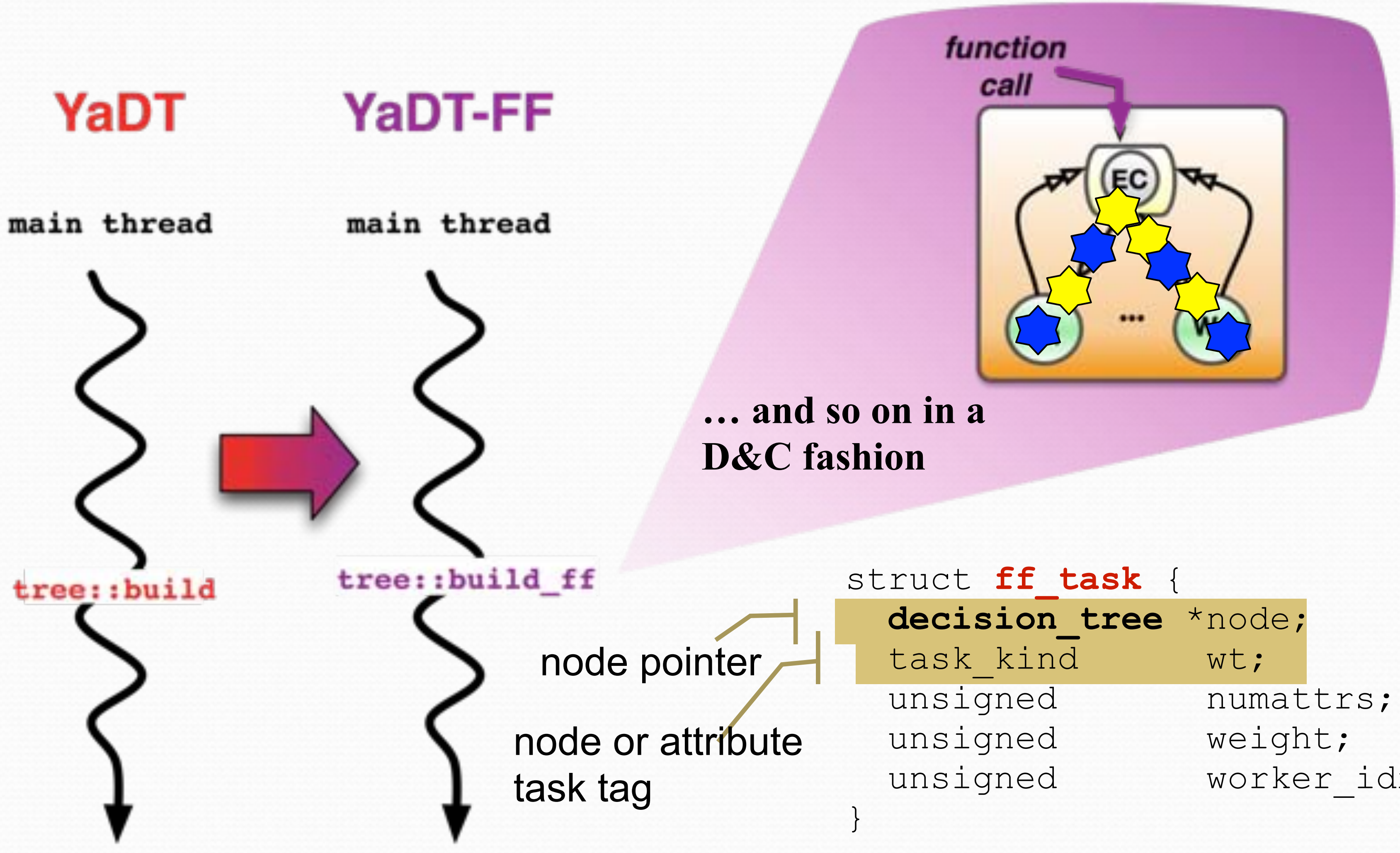
node or attribute task tag

YaDT-FF: overall schema




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

YaDT-FF: overall schema



… and so on in a D&C fashion

node pointer

node or attribute task tag

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

YaDT-FF: D&C setup

```
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

```
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();
}
```

- ff_emitter(root,PAR_DEGREE): creates the Emitter object
- workers.push_back(new ff_worker()): creates PAR_DEGREE worker objects
- farm.add_workers(workers): adds all workers
- farm.add_emitter(E): adds the Emitter
- farm.wrap_around(): creates feedback channels
- farm.run_and_wait_end(): excutes the farm, and waits for the 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)

```
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)

```
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;
}
```

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

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

```
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())
      findThreashhold(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

```
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())
      findThreashhold(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)

```
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)

```
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;
```

- `if (!task) {` — at the beginning the task is NULL
- `(root->splitPre())` — for the root node, attribute parallelisation is always the case
- `ff_task(root,BUILD_ATT)` — for each attribute, creates an attribute task
- `ff_send_out(task);` — FF run-time calls,this schedules the task toward one of the Workers
- `root->attTask=c;` — remembers how many tasks we have to wait for
- `return FF_GO_ON;` — tells the FF run-time to check the input queue for another task
- `task->isBuildAtt()` — tells if the task is an attribute task
- `if (--n->attTask > 0) return FF_GO_ON;` — we have to wait that all attribute tasks come back
- `n->splitPost();` — all tasks have arrived
- `noMoreTasks() && !n->nChilds()` — termination condition

(**continue…**)

YaDT-FF: NAP strategy (Emitter 2/2)

```
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 (Emitter 2/2)

```
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;
}
```

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

YaDT-FF: NAP strategy (Worker)

- The worker implementation is straightforward

```
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

```
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

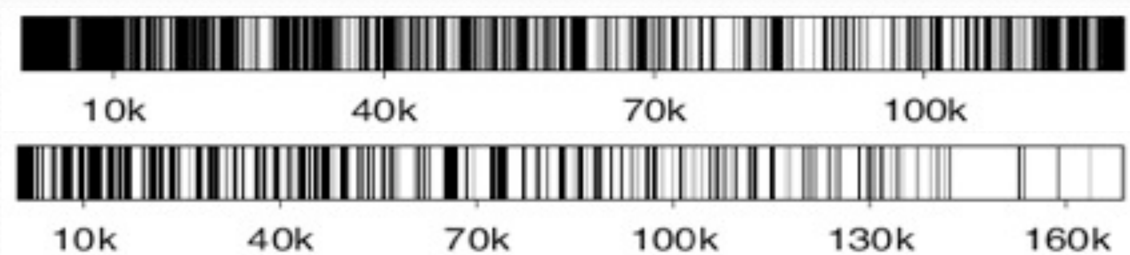
computes information gain on all attributes (as in the NP strategy)

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):



Census PUMS

SyD10M9A

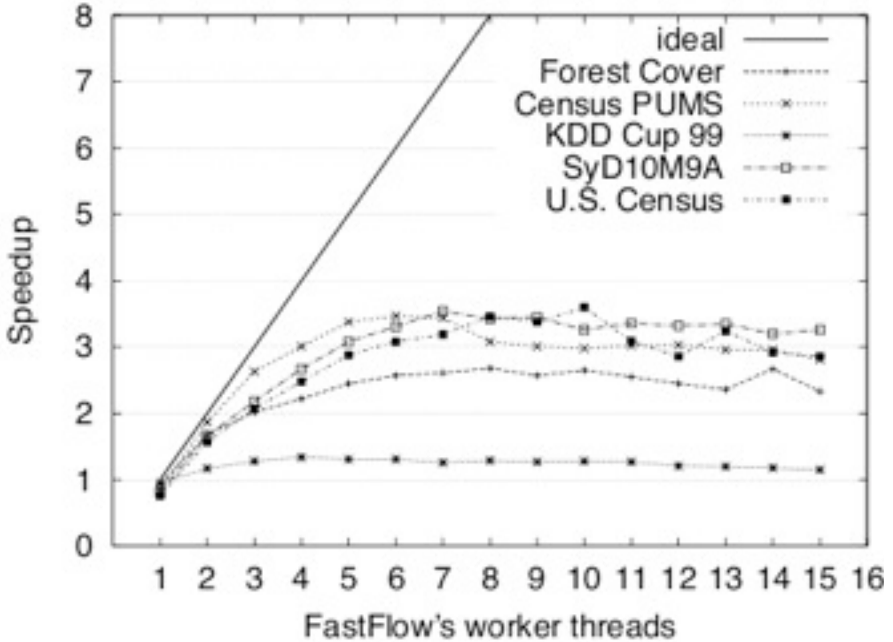
Training sets used in experiments

- Several standard training sets tested

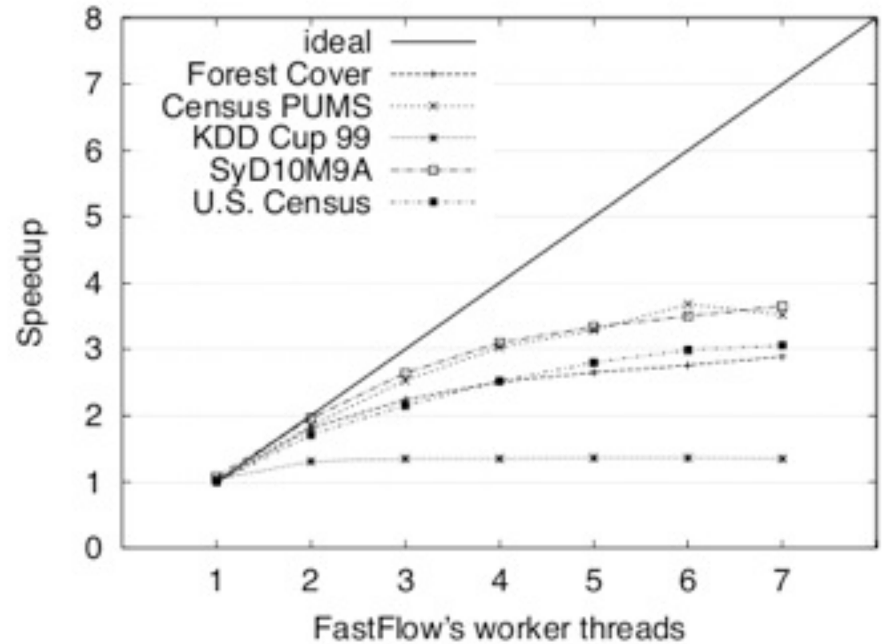
name	N. of cases	NC	No. of attributes			Tree	
			discr.	contin.	Total.	size	depth
Census PUMS	299,285	2	33	7	40	122,306	31
U.S. Census	2,458,285	5	67	0	57	125,621	44
KDD Cup 99	4,898,431	23	7	34	41	2,810	29
Forest Cover	581,012	7	44	10	54	41,775	62
SyD10M9A	10,000,000	2	3	6	9	169,108	22

YaDT-FF: NP strategy speedup

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



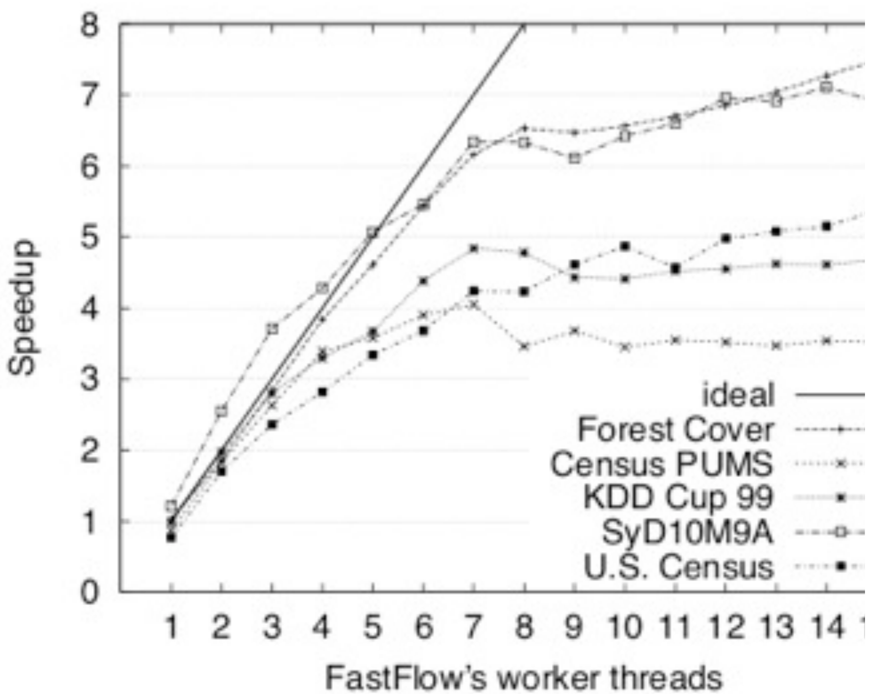
Dual quand-core (8 threads) Xeon E5420@2.5

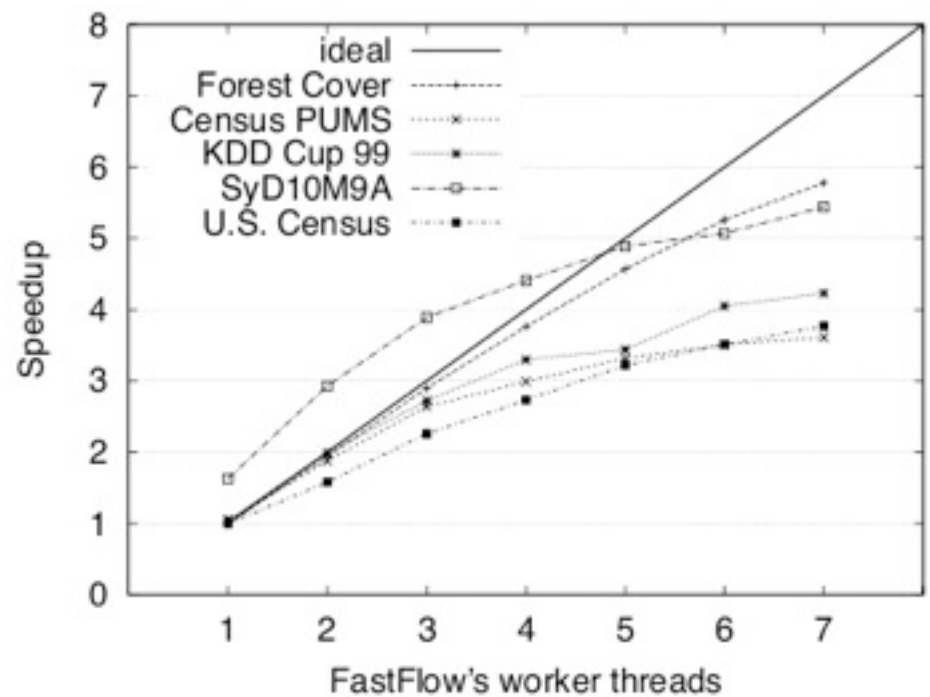


- **Moderate speedup**, there's not enough parallelism to go round
- The performance/cost ratio is very high because of minimal changes to the sequential code
- The approach is generally applicable to top down tree-growing algs.

YaDT-FF: NAP strategy speedup

Dual quand-core (16 threads) Xeon E5520@2.2 Dual quand-core (8 threads) Xeon E5420@2.5

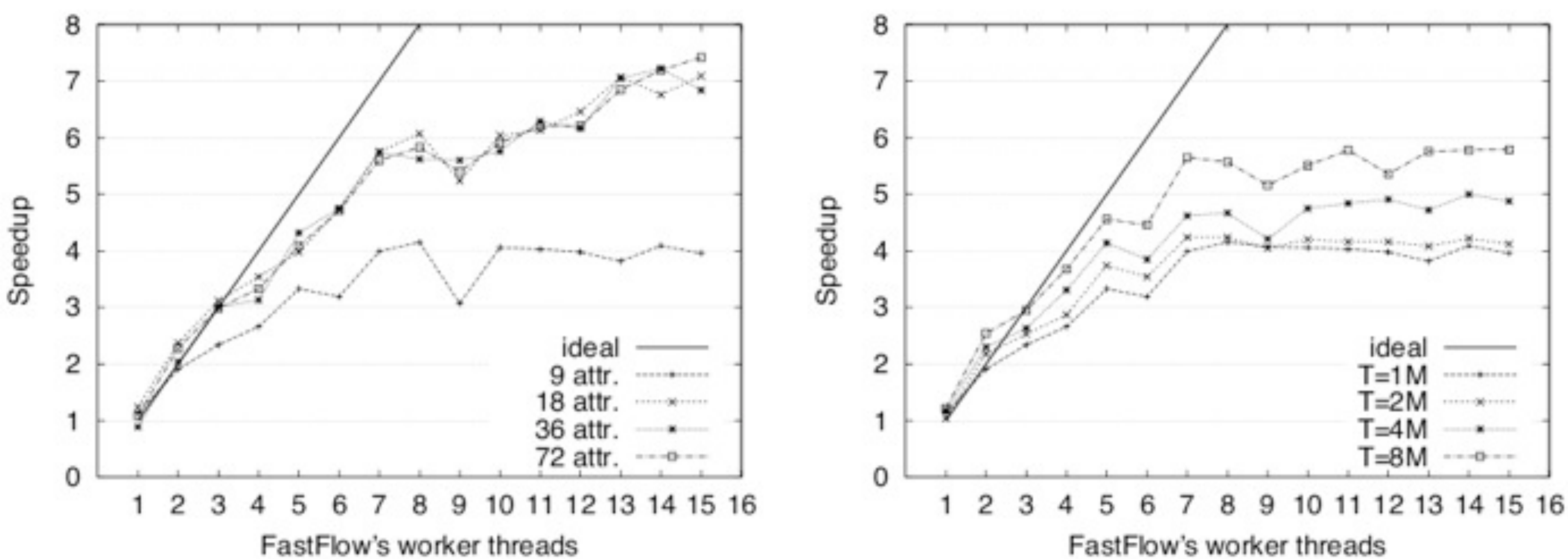




- The NAP strategy obtains a good speedup ranging from 4 (Census PUMS) to 7.5 (Forest Cover) with an efficiency of 93%
- The HyperThreaded box provides 12-30 % performance improvement

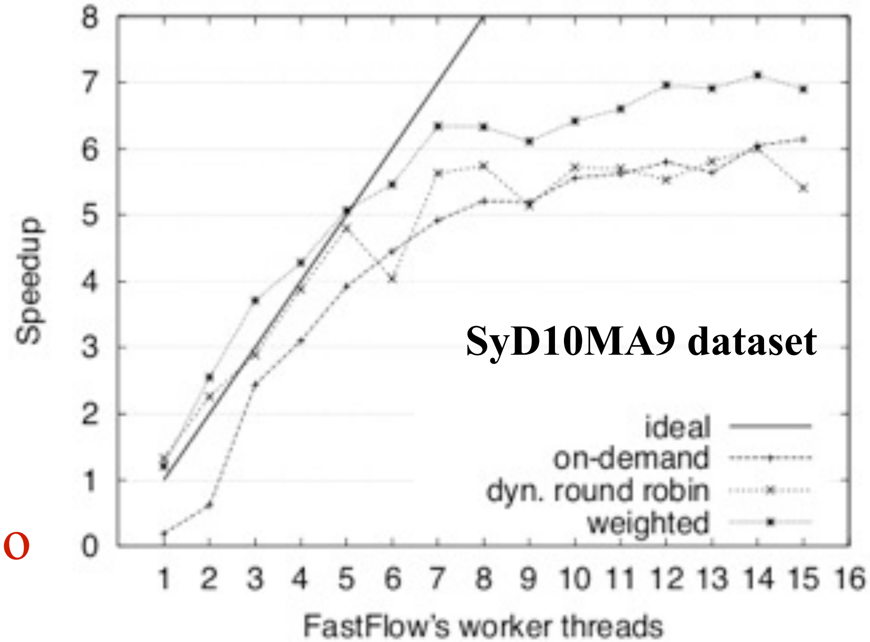
YaDT-FF: NAP strategy speedup

Dual quand-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)

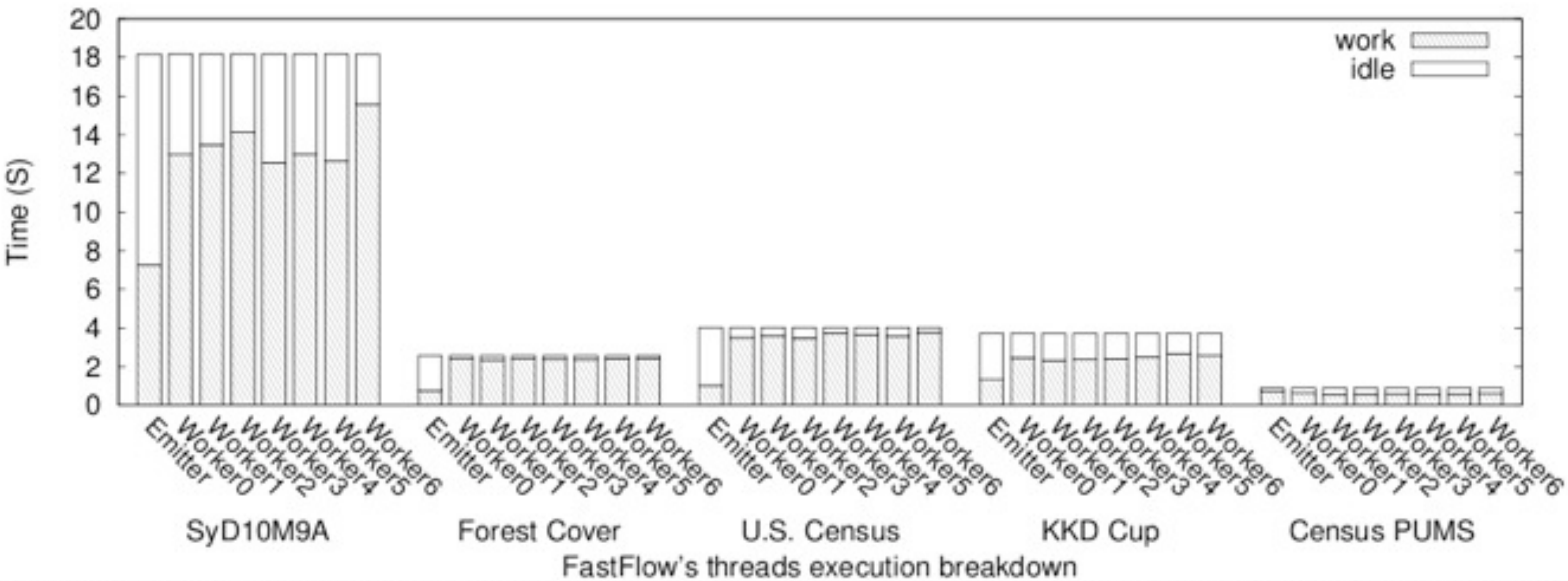
YaDT-FF: load-balancing



- 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 *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

"Porting Decision Tree Algorithms to Multicore using FastFlow" M.Aldinucci, S.Ruggieri, M.Torquati

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:

  http://mc-fastflow.sourceforge.net

Thank you.