Adaptive parallel computing with Kaapi

[1]Clément Pernet, clement.pernet@imag.fr
[2]Thierry Gautier, thierry.gautier@inrialpes.fr

[1,2]MOAIS project, INRIA Grenoble Rhône-Alpes

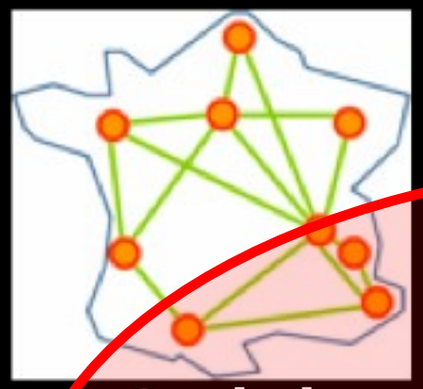
[2]Visiting Position at ArTeCS Group, Complutense, Madrid, Spain

Moais Project

http://moais.imag.fr

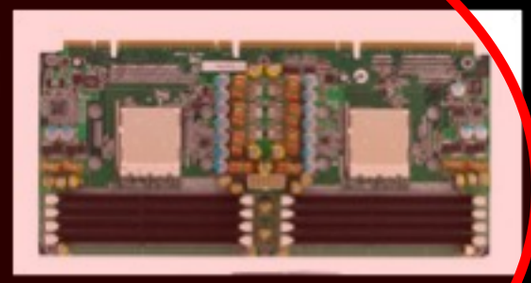
- Leader
  - Jean-Louis Roch
- 10 Members
  - Vincent Danjean, Pierre-François Dutot, Thierry Gautier, Guillaume Huard, Grégory Mounié, Clément Pernet, Bruno Raffin, Denis Trystram, Frédéric Wagner
- About 20 PhD students

Kaapi Positioning



Grid

Cluster

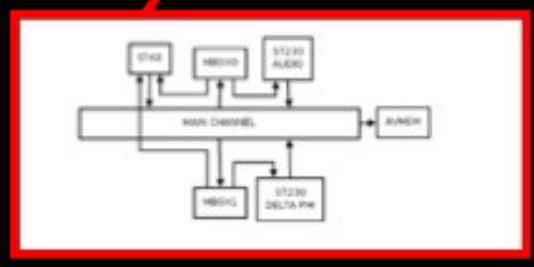


Multicore

KAAPI

GPU

MPSoC



- To mutually adapt application and scheduling
- Extension to target GPU & MPSoC

Goal

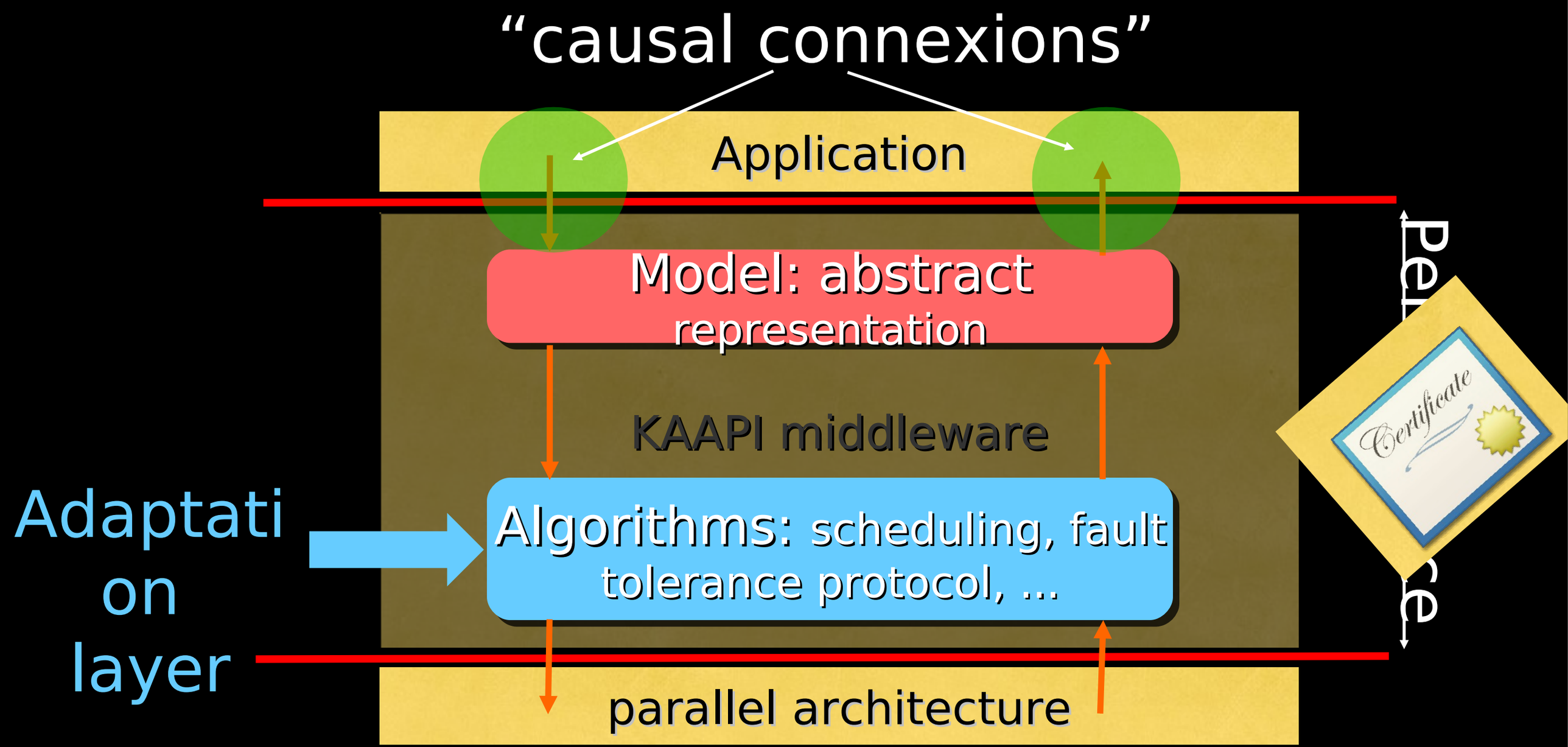
- Write once, run anywhere... with guaranteed performance

- Problem: heterogeneity
  - variations of the environment (#cores, speed, failure...)
  - irregular computation

Outline

- Athapascan / Kaapi
  - Abstract representation & Task model
- Scheduling
  - Graph Partitioning & Work stealing
- Fault tolerance
  - CCK: Coordinated Checkpointing / Graph partitioning
- Applications in computer algebra
- Conclusions

KAAPI Overview



API: Athapascan

- Global address space
  - Creation of objects with ‘shared’ keyword
- Task = function call
  - Creation with ‘Fork’ keyword ~ Cilk spawn
  - Tasks only communicate through shared objects
  - Task declares access mode (read, write, concurrent write, exclusive) to shared objects

Properties

- Dynamic macro data flow graph
  - Dependencies between tasks are known
- Automatic scheduling
  - Work stealing or graph partitioning
- ‘Sequential’ semantics
  - À la Cilk/TBB but with data flow dependencies
- C++ library, not a language extension
  - C language extension + compiler was prototyped

C++ Elision

```
struct Fibonacci {
  void operator()( int n, a1::Shared_w<int> result )
  {
     if (n < 2) result.write( n );
     else {
        a1::Shared<int> subresult1;
        a1::Shared<int> subresult2;
        a1::Fork<Fibonacci>()(n-1, subresult1);
        a1::Fork<Fibonacci>()(n-2, subresult2);
        a1::Fork<Sum>()(result, subresult1, subresult2);
     }
  }
};

struct Sum {
  void operator()( a1::Shared_w<int> result,
                   a1::Shared_r<int> sr1,
                   a1::Shared_r<int> sr2 )
  { result.write( sr1.read() + sr2.read() ); }
}
```

C++ Elision

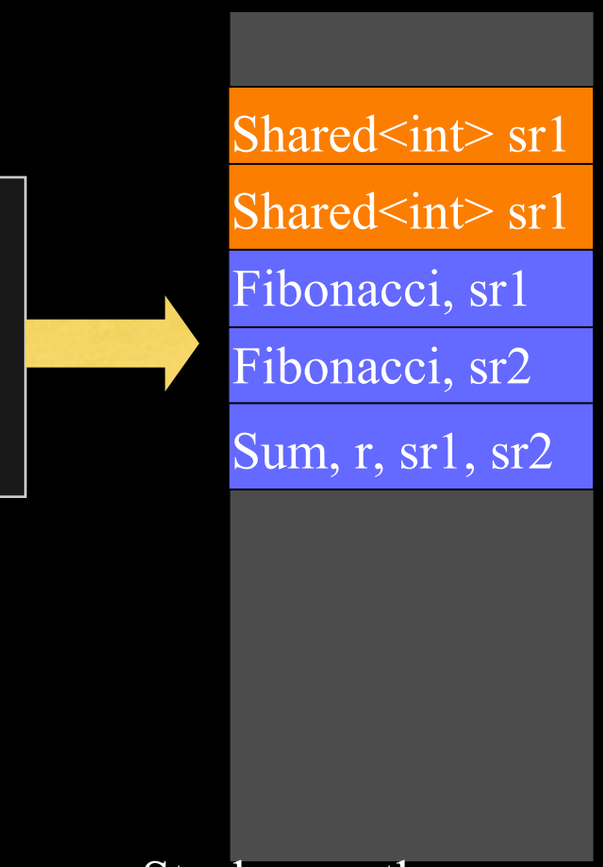
```
struct Fibonacci {
  void operator()( int n,                    int& result )
   {
     if (n < 2) result =       n  ;
     else {
                  int  subresult1;
                  int  subresult2;
                Fibonacci ()(n-1, subresult1);
                Fibonacci ()(n-2, subresult2);
                Sum ()(result, subresult1, subresult2);
     }
   }
};

struct Sum {
  void operator()(                    int& result,
                           int  sr1,
                           int  sr2 )
   { result =       sr1          + sr2           ; }
}
```

Stack management

- Stack based allocation
  - Tasks and accesses to shared data are pushed in a stack
    - close to the management of the C function call stack
  - O(1) allocation time
  - O(#parameters) initialization time

```
a1::Shared<int> subresult1;
a1::Shared<int> subresult2;
a1::Fork<Fibonacci>()(n-1, subresult1);
a1::Fork<Fibonacci>()(n-2, subresult2);
a1::Fork<Sum>()(result, subresult1, subresult2);
```



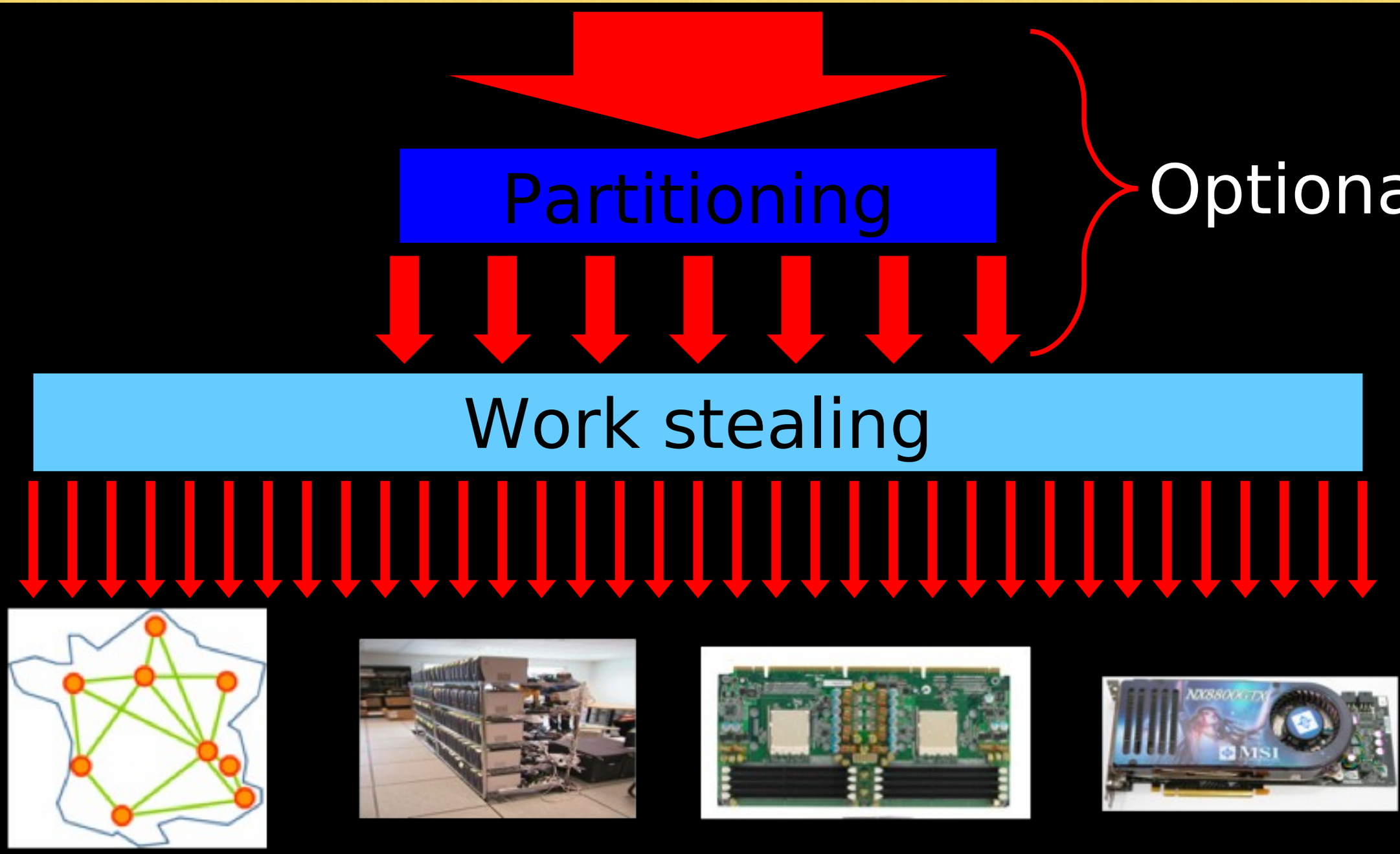
- Cost
  - About 10 times an empty function call
  - [Cilk++ : about 25 times a function call]
- Further optimization: compilation / binary rewriting

Outline

- ✓ Athapascan / Kaapi
  - ✓ Abstract representation & Task model
- Scheduling
  - Graph Partitioning & Work stealing
- Fault tolerance
  - CCK: Coordinated Checkpointing / Graph partitioning
- Applications in computer algebra
- Conclusions

Two level scheduling

At execution time : Data Flow Graph /

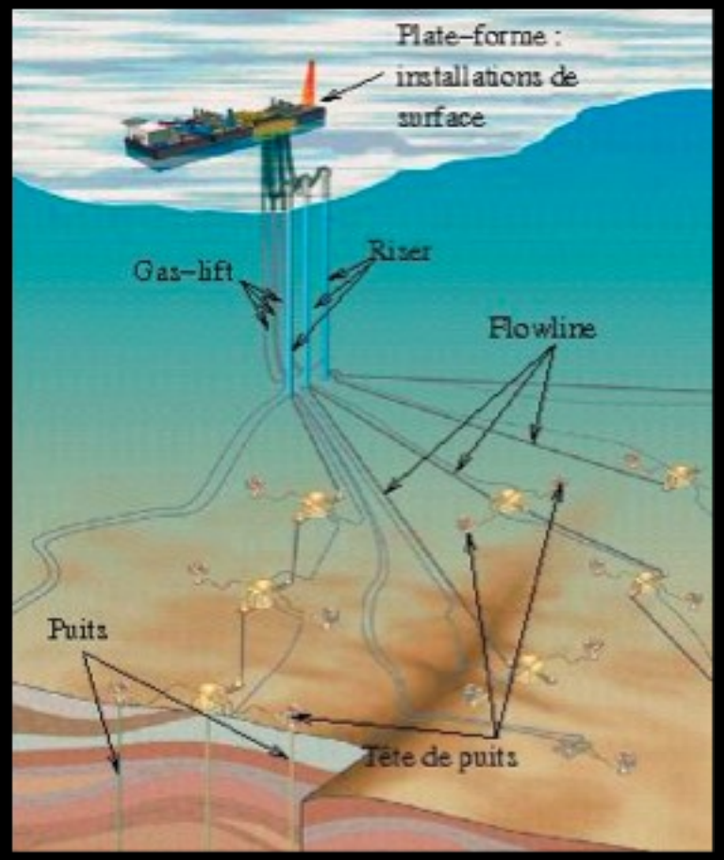


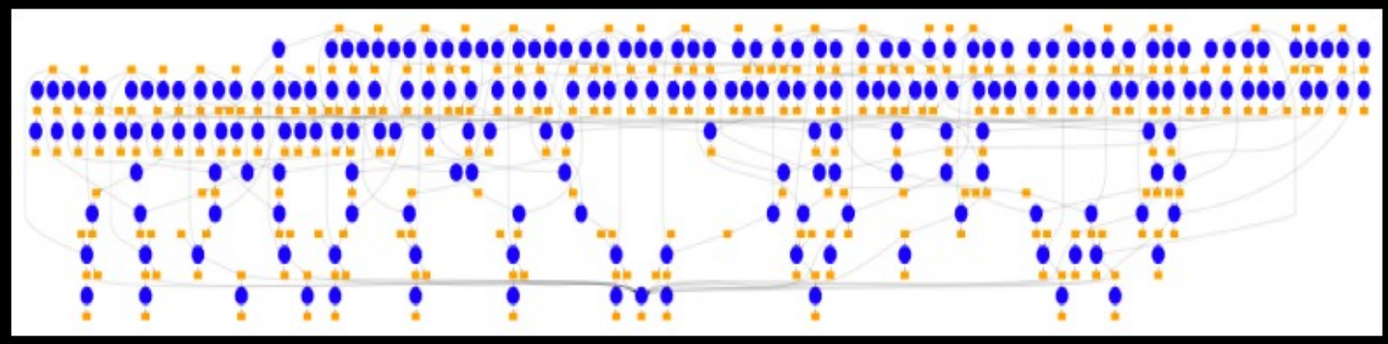
Graph partitioning

- Input: data flow graph
- Output: k partitions of the tasks
  - 1 partition = data flow graph
  - Communication = couple of tasks
    - One to broadcast the data / in the partition where data is produced
    - One(s) to receive the data / in the partition(s) where data is consumed
- Algorithms
  - Based on METIS/Scotch graph partitioner
  - DSC / ETF: oriented graph
  - Recursive Geometric Partitioner (required spatial attributes on data)
  - Local rescheduling to improve overlapping

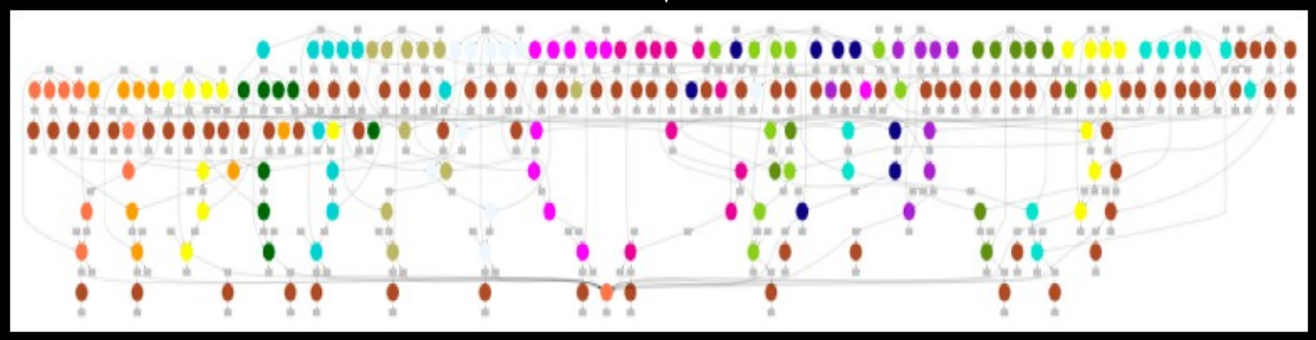
Iterative Application

- Scheduling by graph partitioning
  - Metis / Scotch

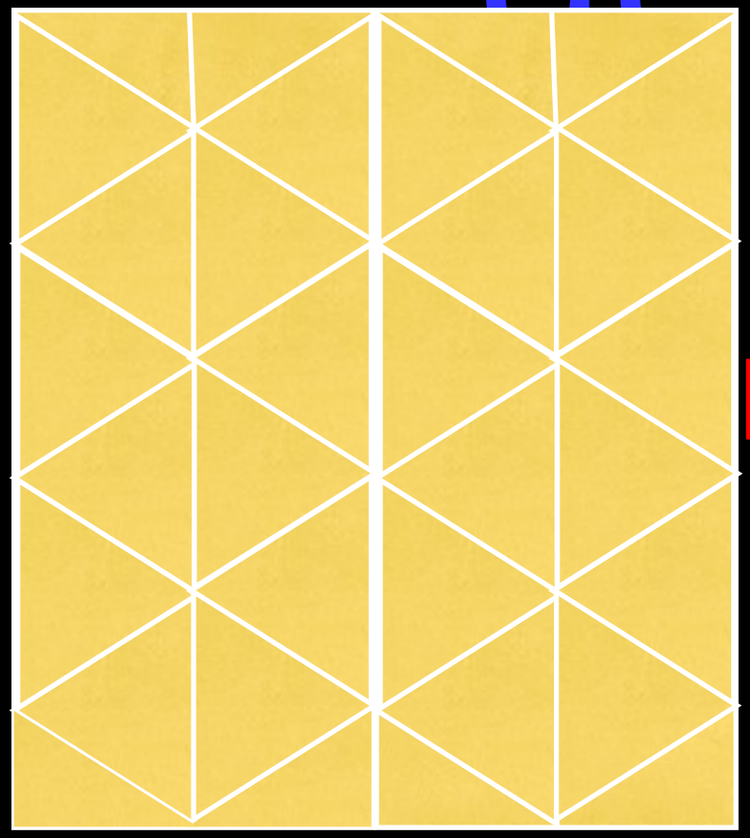




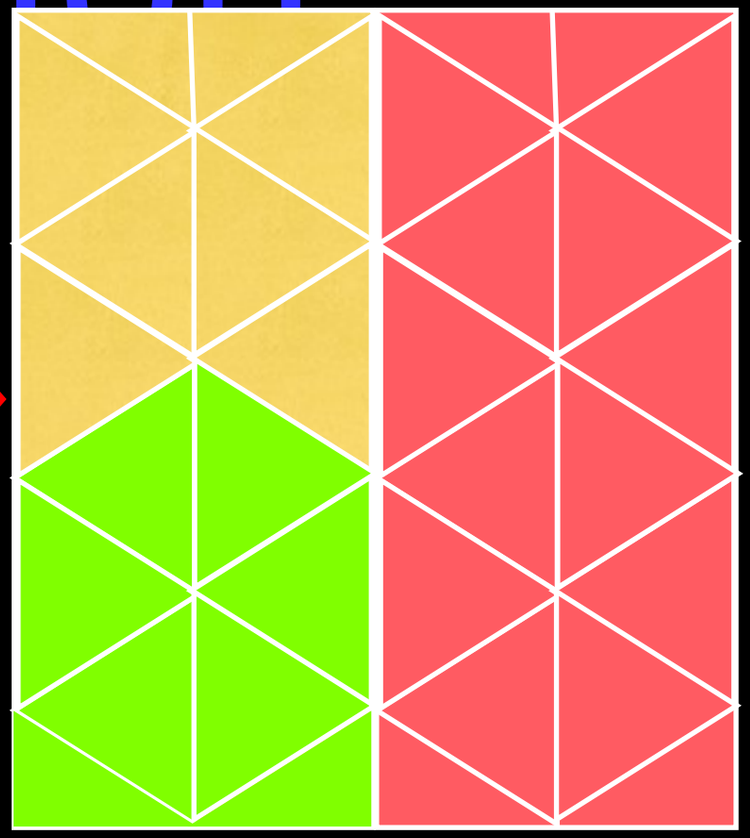
Partitioning

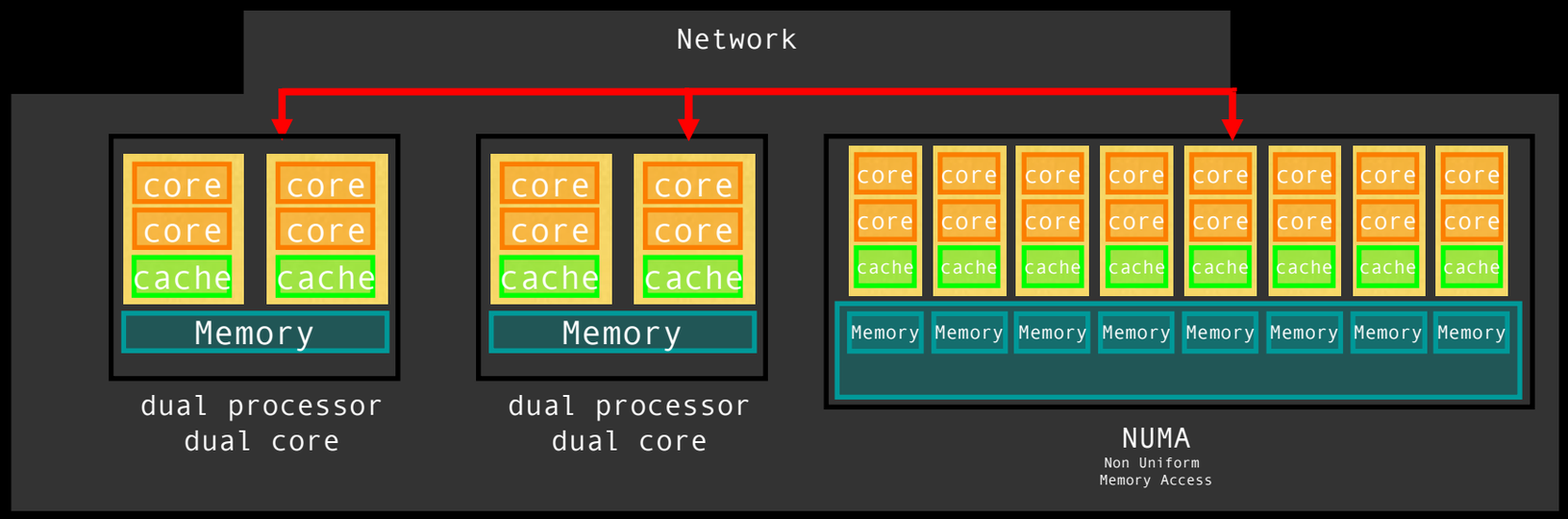


Domain decomposition



Graph partitioner
- scotch
- metis
- hierarchical :
  ANR DISCOGRID





Experiments

- Finite Difference Kernel
  - Regular Grid / Constant size sub domain D per processor
  - Kaapi / C++ code versus Fortran MPI code
  - Cluster : N processors on a cluster / Grid : N/4 processors per cluster, 4 clusters

➔ Automatic overlapping latency by computation

D=256^3	# processors	Cluster (s)	Grid (s)	TCluster/ TGrid
KAAPI	1	0.49	0.49	-
	64	0.55	0.84	0,53
	128	0.65	0.91	0,4
MPI	1	0.44	0.44	-
	64	0.66	2.02	2,06
	128	0.68	1.57	1,31

Graph Partitioning

- Initial data / work distribution
  - Important to avoid bottleneck and performance

- Cost to compute partitions
  - Cost of the basic partitioning algorithm
    - It depends on the scheduling algorithm, e.g. ETF is costly
  - O(#tasks) to convert graph, compute a local scheduling, …
  - This cost is only paid for the first iteration of iterative application

Work stealing

- Idle processors try to steal work from selected victim

- Well suited for recursive divide & conquer approach
- Also used with parallel STL primitives

Theoretical bound

- Using work stealing scheduler with random selection of victim, the expected time on P processors is:

$$T_p = O(T_{seq} / P + T_\infty)$$

  - [Galilée, Doreille, Cavalheiro, Roch, PACT 98],[Gautier, Roch, Wagner, ICCS2007]
  - Similar bounds with an other context: [Blumofe, Leiseron, Focs 94, PPoPP 95], [Arora, Blumofe, Plaxton, SPAA 98], ...

- The expected number X_p of steal requests per thread is:

$$X_p = O(T_\infty)$$

Work first principle

- Due to [Cilk Team]
  - "Minimize scheduling overhead borne by work at the expense of increasing the critical path."
- Application to compute of data flow constraints
  - During steal request: find the first task ready and steal it
    - Here we compute data flow dependencies !
  - Standard execution: processor execute tasks following a (valid) order
    - Stolen task stop execution if it could introduce future dependencies
    - The processor becomes idle and steal work

Outline

- ✓ Athapascan / Kaapi a software stack
  - ✓ Abstract representation & Task model
- ✓ Scheduling
  - ✓ Graph Partitioning & Work stealing
  - Controls of the overheads
- Fault tolerance

  CCK: Coordinated Checkpointing / Graph partitioning
- Applications in computer algebra
- Conclusions

How to reduce T_1/T_{seq}?

- Why ? WORK overhead reduce efficiency
  - extra instructions from the sequential program
  - especially for short computation

- Three principal technics

1. adapt the grain size: stop parallelism after a threshold
   - but: may increase dramatically T_∞, reduce the average parallelism and increase the number of steal requests
   - difficulty to adjust it automatically
2. reduce the cost to create task
   - ...ideally do not create task !
3. optimize the cost of workqueue operations to push/pop tasks
   - difficulty due to concurrent operations

Cost of task

- Cost = Creation + Extra arithmetic work
  - Example: prefix computation
  - Fish's lower bound: any parallel algorithm with critical path $\log_2 n$ requires at least 4n operations

- Adaptive Algorithm
  - [Roch, Traoré 07], [Roch, Traoré, Gautier 08]
  - Principle: create tasks when processors are idle !
  - Task should provide a way to extract / merge work
    - Adaptive Task in the API

Workqueue optimization

- 3 operations
  - push / pop + steal
- Main algorithms
  - Cilk: T.H.E. protocol
    - serialization of thieves to a same victim
    - thief/victim atomic read/write + lock in rare case
  - ABP [SPAA00]:
    - lock free (Compare&Swap), but prone to overflow
  - Chase & Lev [SPAA05]: extend ABP
    - without limitation (other than hardware)

C O S T L Y ‘cas’ operation [PPoPP09]

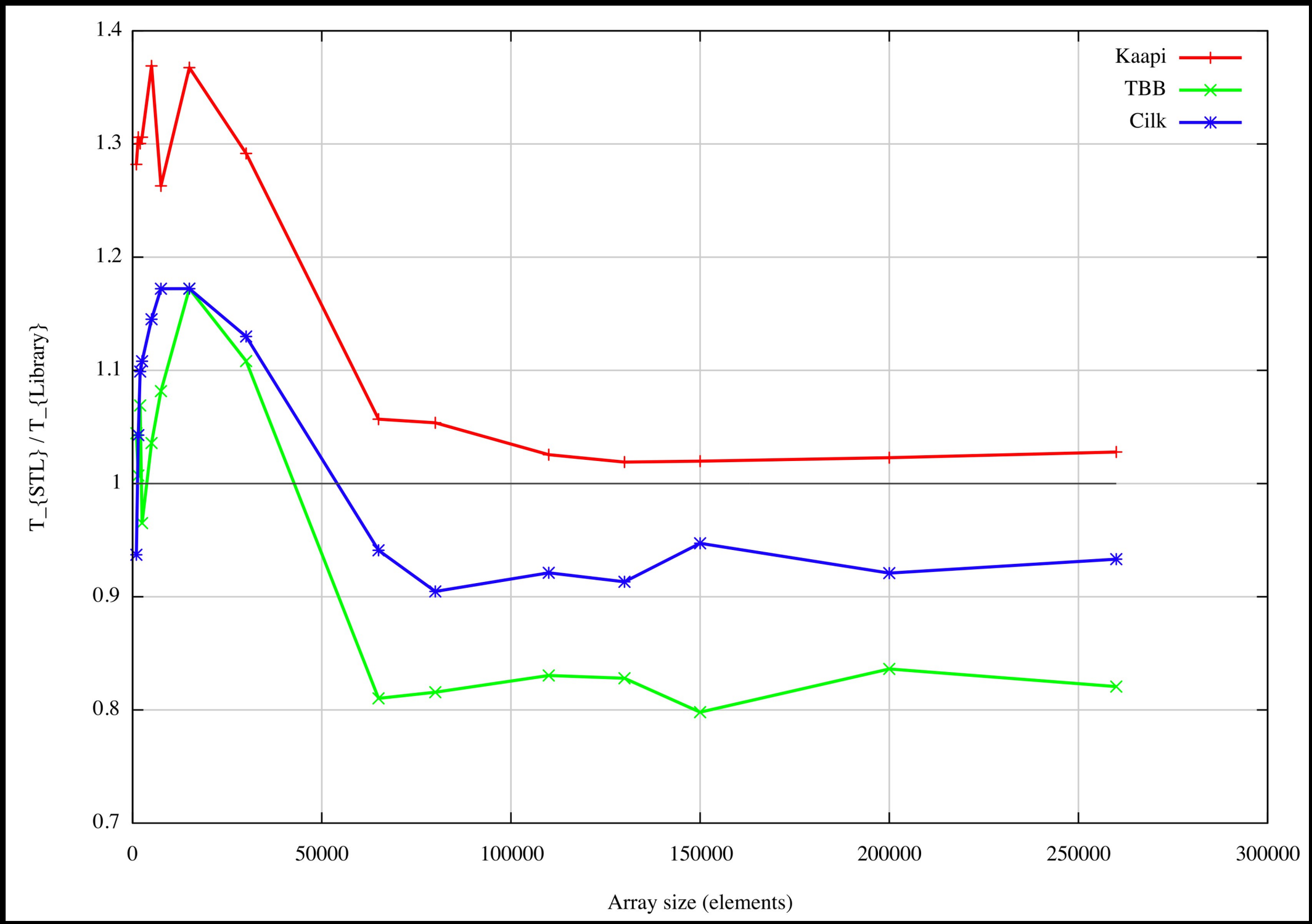
Cooperative work-stealing

- [X. Besseron, C. Laferrière]
- Keep same semantics as usual
  - a task is extracted exactly once
- avoid concurrency between victim & thieves
  - the victim interrupts its work to process steal requests
- Drawback
  - the victim should poll requests / thieves are waiting
- Advantage: initial load distribution
  - several steal requests may be processed together

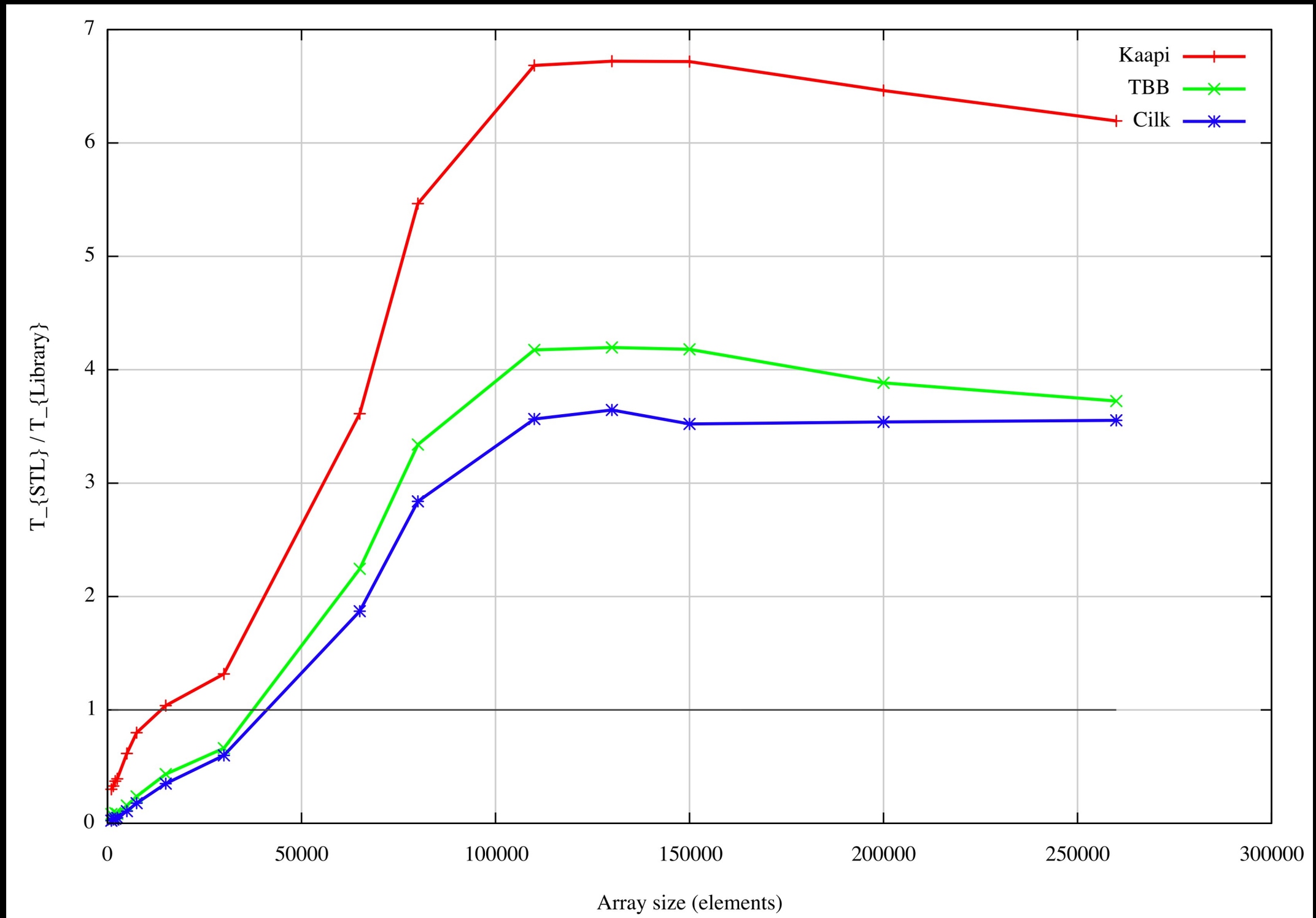
Experiments on multicore

- STL algorithms
  - adaptive algorithms: PhD of [D. Traoré]
  - with cooperative work stealing: [Xavier Besseron, C. Laferrière]
- Comparison with Cilk++ / TBB
- Methodology
  - average over 300 runs
  - do not take into account the first measure
- 16 cores, 8 sockets multicore machine (opteron, 2.2Ghz).

T_1/T_{seq} on std::transform

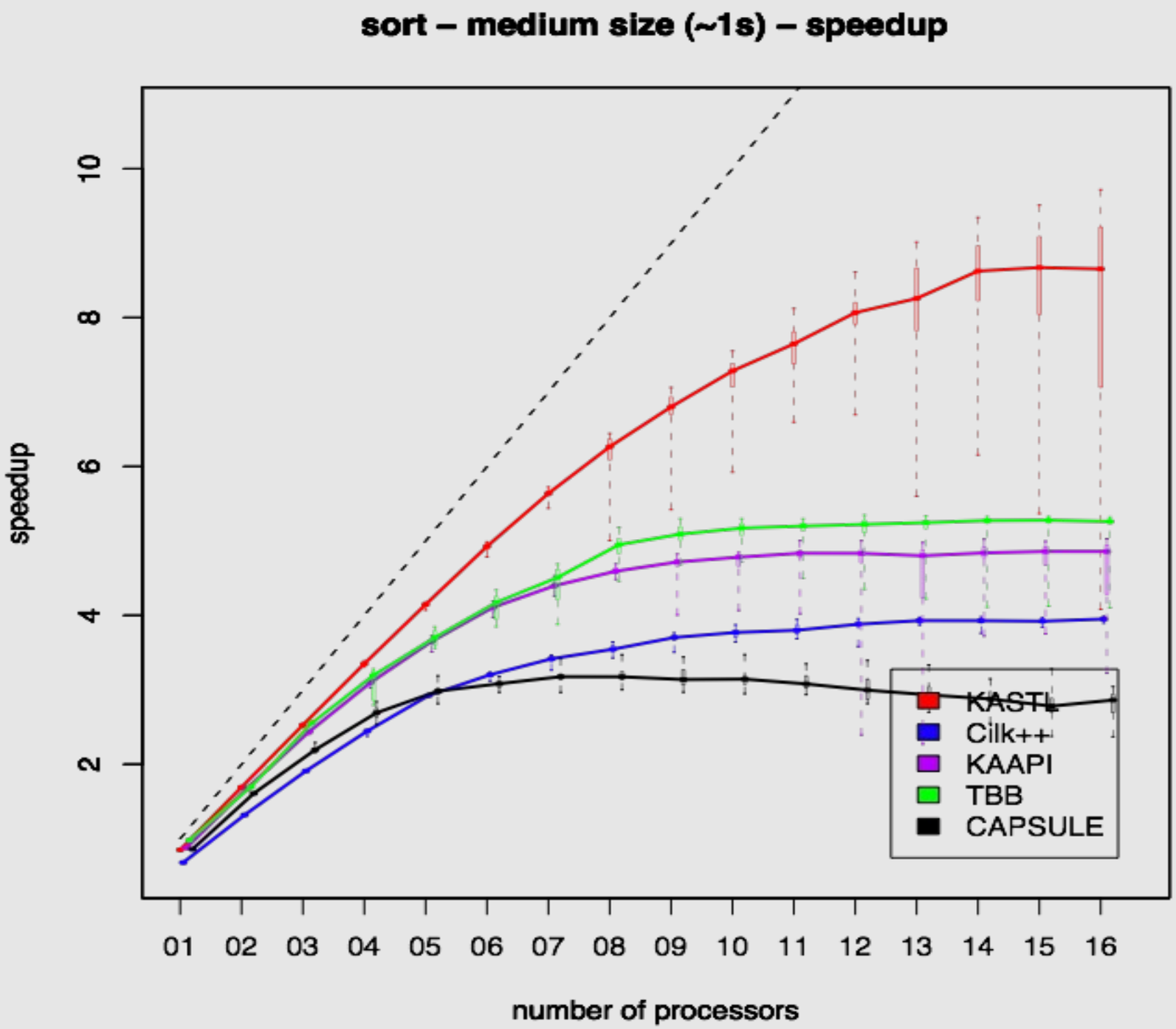


T_{stl} / T_{8} std::transform

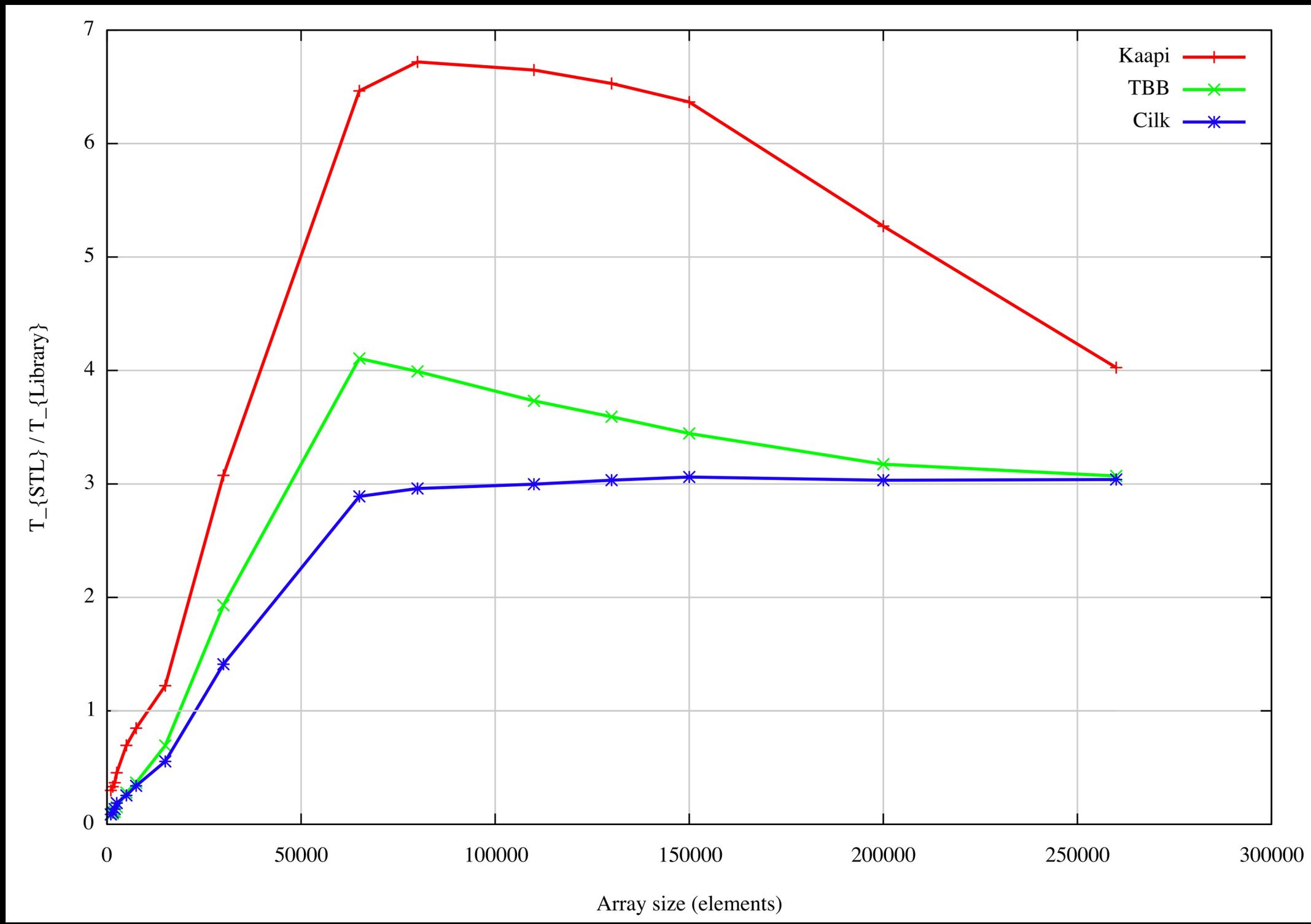


Speedup / Sort

- ~ 100M elements, 1s sequential time



T_{stl} / T_{8} std::merge



Outline

- ✓ Athapascan / Kaapi a software stack
  - ✓ Abstract representation & Task model
- ✓ Scheduling
  - ✓ Graph Partitioning & Work stealing
  - ✓ Controls of the overheads
- Fault tolerance
  - CCK: Coordinated Checkpointing / Graph partitioning
- Application in computer algebra
- Conclusions

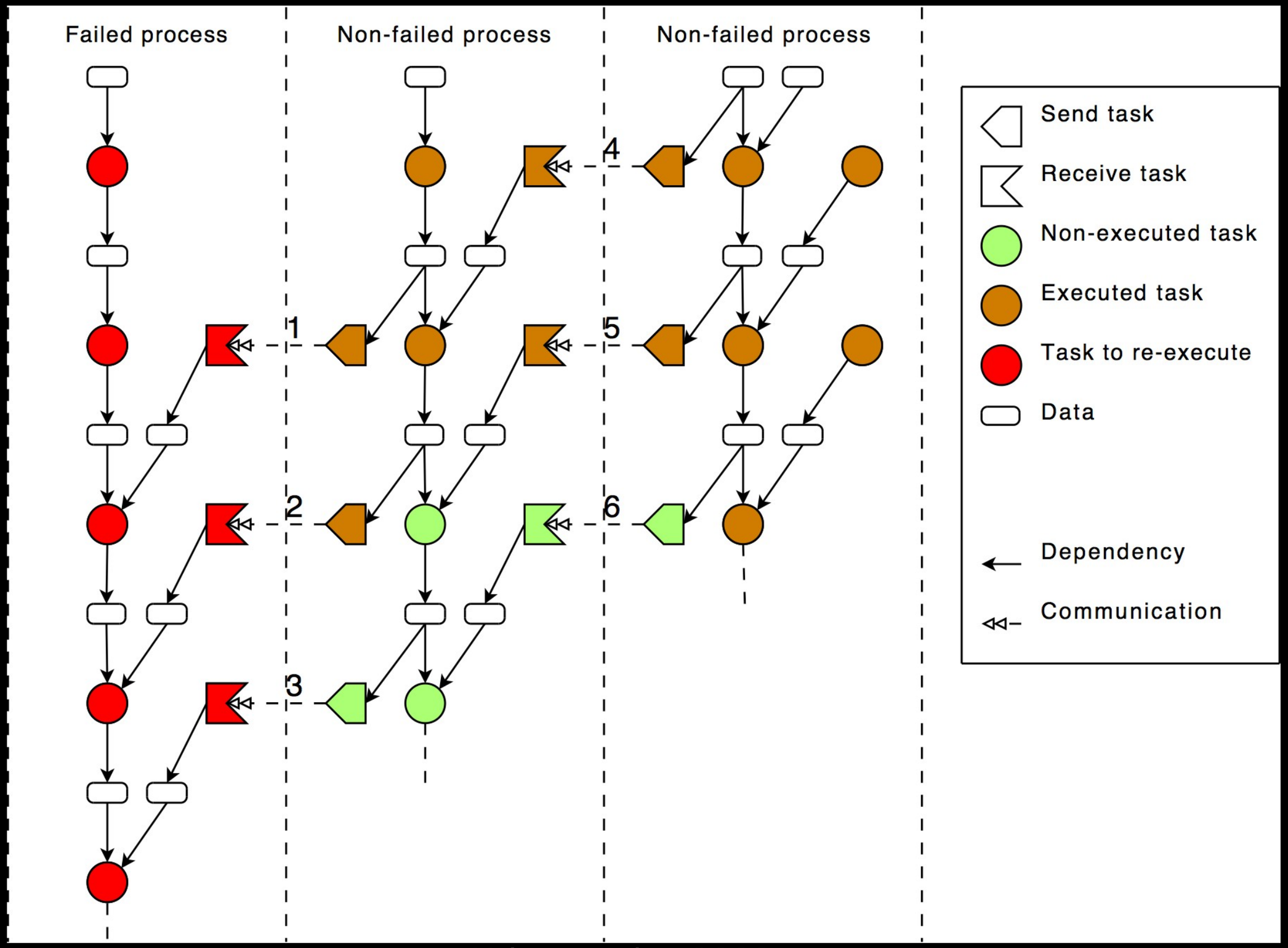
Fault Tolerance

- State of application = state of the data flow graph
- Two specialized protocols
  - TIC: Theft Induced Checkpointing
    - Periodic checkpoint + forced checkpoint on steal
  - CCK: Coordinated Checkpoint

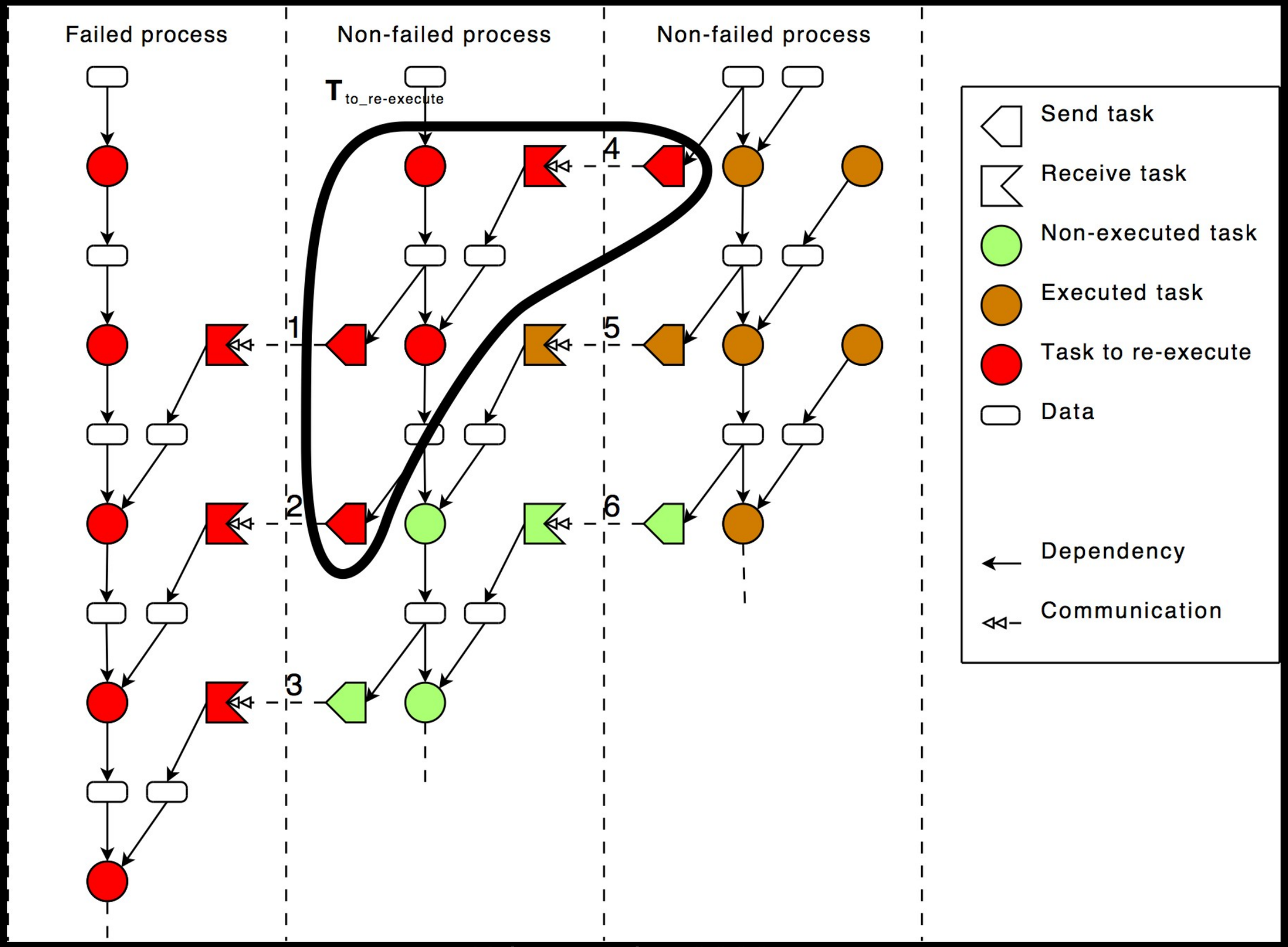
Coordinated Checkpoint

- PhD [Xavier Besseron]
  - Checkpoint as a special case of dynamic reconfiguration of parallel application
- Classical protocol restart
  - Global restart:
    - Failed processes replaced by new ones
    - All processes restart from their last checkpoint
    - Restart time is, in worst case, the checkpoint period
- CCK protocol restart
  - Partial restart:
    - Detect lost communications for the failed processes
    - Find the set of strictly required computations to make the global state coherent
    - Schedule statically this task set

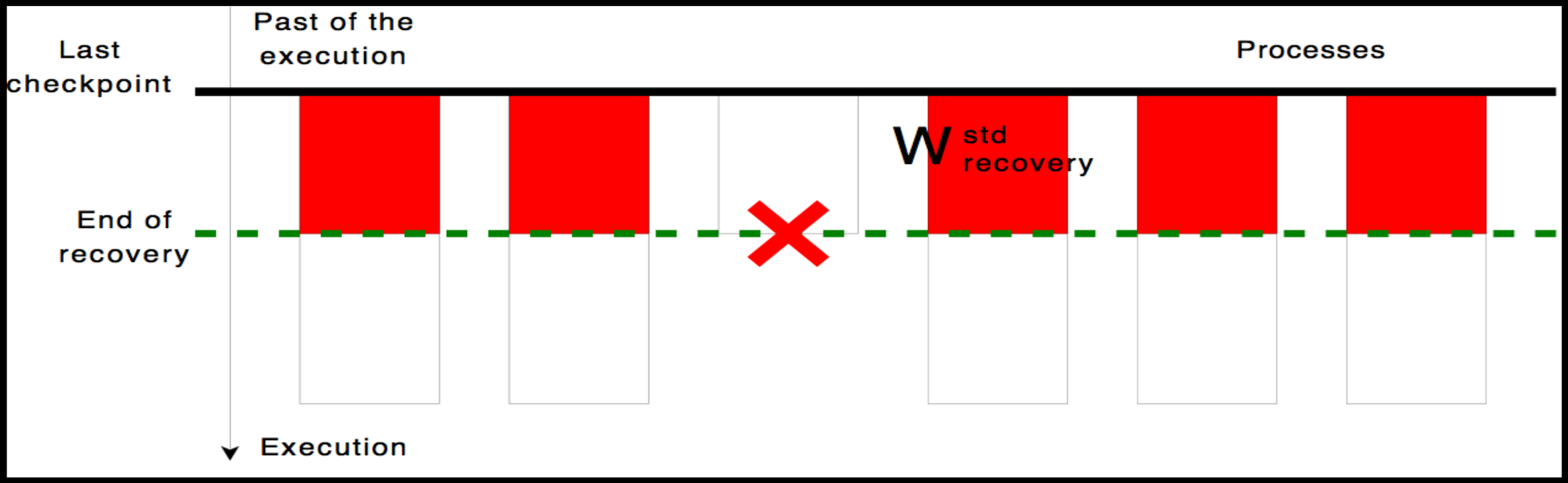
After a failure

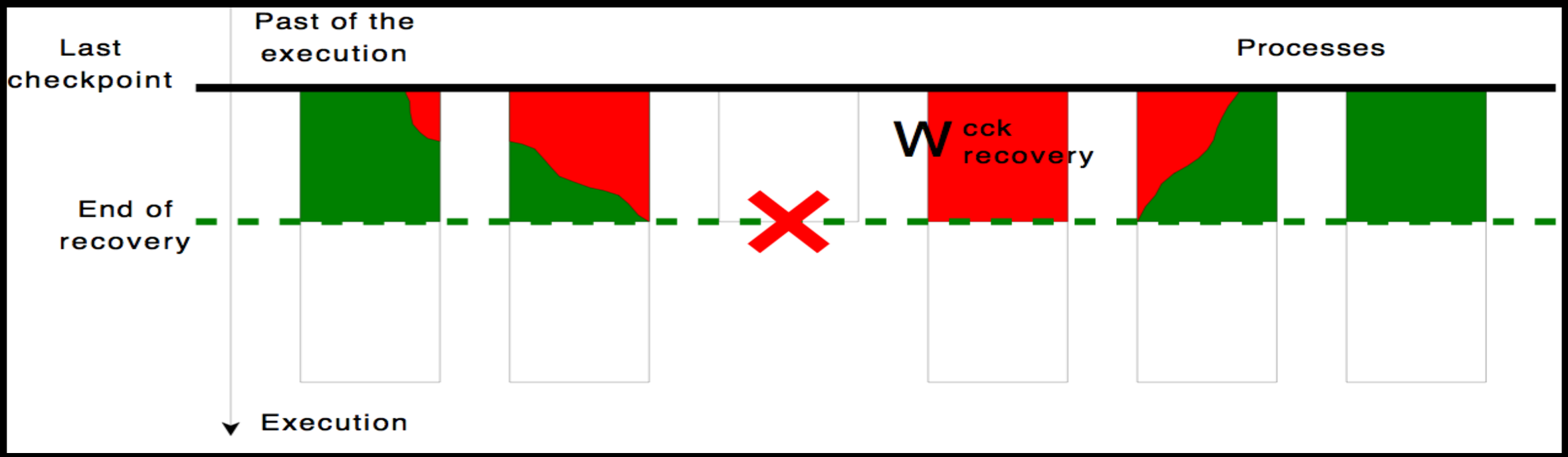


Tasks to reexecute



Restart Std / CCK





Outline

- ✓ Athapascan / Kaapi a software stack
  - ✓ Abstract representation & Task model
- ✓ Scheduling
  - ✓ Graph Partitioning & Work stealing
  - ✓ Controls of the overheads
- ✓ Fault tolerance
  - ✓ CCK: Coordinated Checkpointing / Graph partitioning
- Applications in computer algebra
- Conclusions

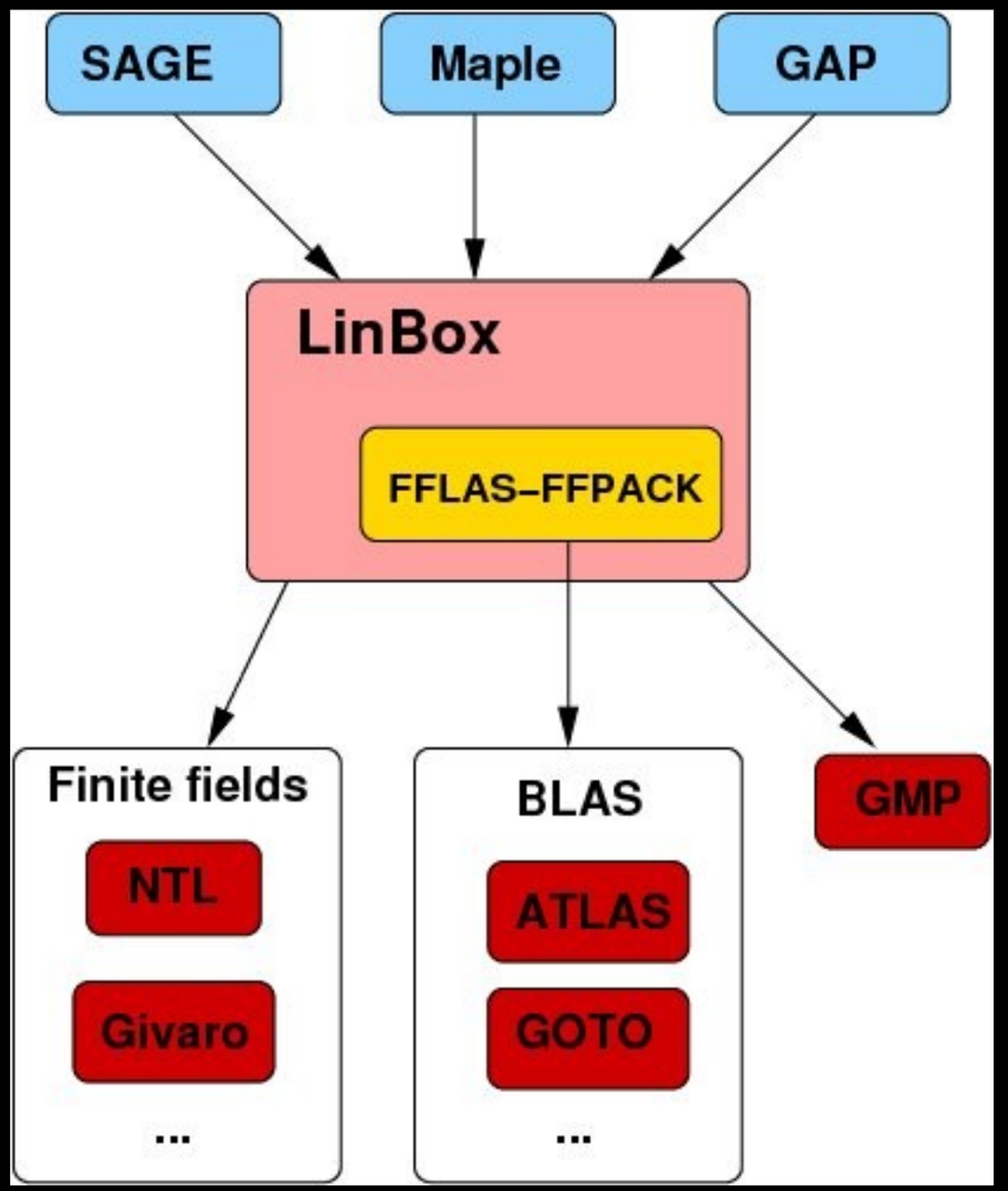
Application in computer algebra

- Formula manipulation : symbolic comp.
- Experimental maths: conjecture testing
  - Number theory,Graph theory
- Certified numerical computations
- Computational biology:
  - DNA sequencing, molecular conformation
- Cryptanalysis
  - Factorization, Discrete log, Groebner basis ....

Boil down to exact linear algebra over Z, Q, and GF(q)

LinBox:exact linear algebra

- Dense, sparse, blackbox matrices
- Over Z, Q, GF(q)
- Genericity:
  - Wrt. Domain, algorithm, matrix implementation, ...
  - PnP modules (field implementations, optional libraries,...)



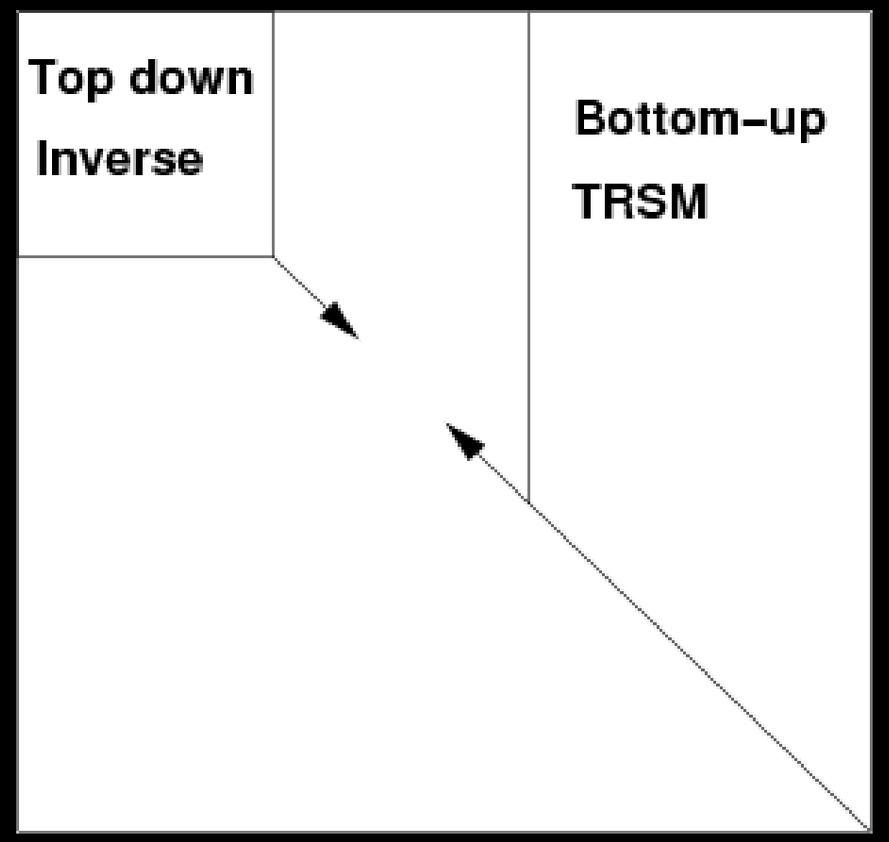
Specificities

- Variable size arithmetic
  - Need for a dynamic work load scheduler
- But easy parallelism: Multimodular approach
  - Allow fault tolerance on bysantine failures
- The cost of genericity: isolate building blocks
  - Krylov iteration factory (Wiedemman, Lanczos, …)
  - Multimodular factory (early term. Fault tolerance,...)
  - Sparse multifrontal solver
  - Dense linalg subroutines over GF(q) (cf PBLAS)

Kaapi & LinBox

- *Data* rather than *Task* parallelism
- STL semantics (par_for) + workstealing
- Adaptive algorithms: eg TRSM

Solve UX=B



Conclusions

- Athapascan/Kaapi
  - A high level model to abstract architecture
  - Performance mostly depends on the “adaptation layer”
    - Work stealing is scalable on large number of processors (~ 4000 cores on both the G5K National Academic Grid + Japanese Intrigger Grid)
    - Effective parallelization of fine computation with cooperative work stealing
  - More experiments should be done
    - Scheduling / execution on cluster or grid: Comparison with Charm++ ?
    - Fault tolerance: fault free execution, time to restart… Charm++ ?
  - http://kaapi.gforge.inria.fr/

Perspectives

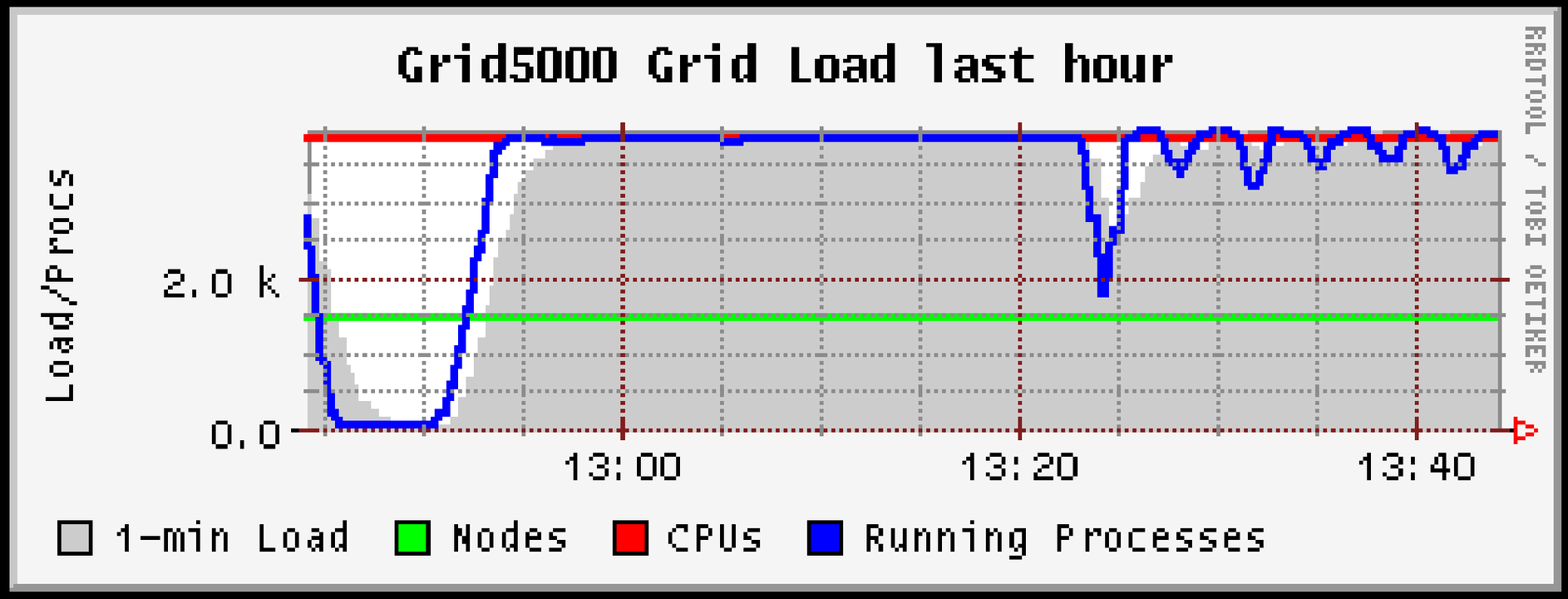
- Blue Gene port [2010]
  - Communication layer: switch to DCMF or CkDirect [Charm++]?
- Mixing CPUs & GPUs
  - preliminary work
  - deeper integration of the GPU as a processing resource
    - Wait the next Fermi GPU + driver ?
- Taking into account hierarchical architecture
  - ongoing work at MOAIS on hierarchical work stealing [Jean-Noël Quintin, PhD]


KAAPI

Other applications

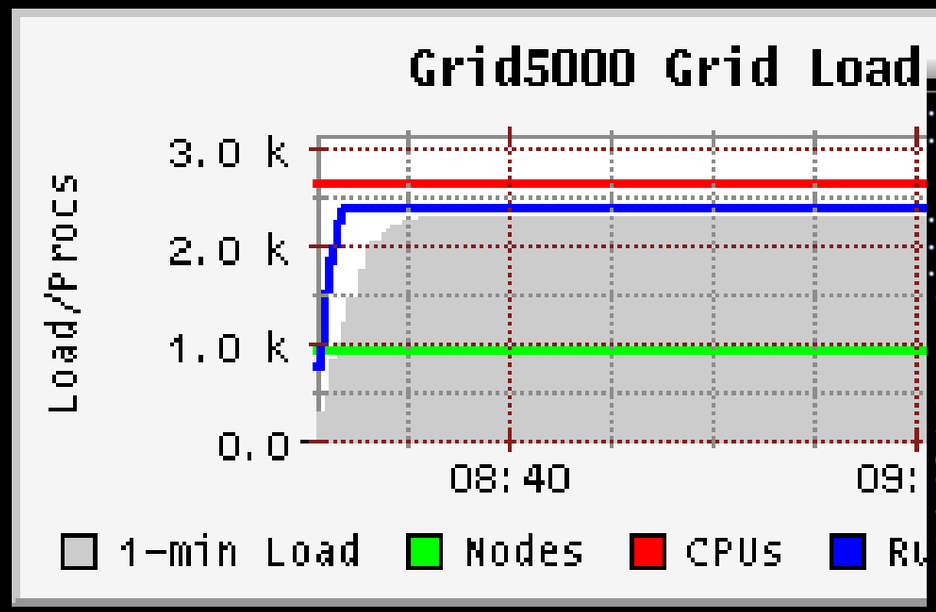
- Parallel Computer Algebra
  - Linbox: http://www.linalg.org
- Combinatorial Opt. [PRiSM (Paris), B. Lecun]
  - QAP / Q3AP problems
- Academic applications
  - [III, IV, V Grid@Work contest]
    - NQueens
    - Option Pricing application based on Monte Carlo Simulation
  - Numerical kernel for CEM, CFD Grid application
    - Finite difference / Finite element
  - Reaction / diffusion with Chemical species
    - Finite difference
- SOFA (http://www-sofa-framework.org), See B. Raffin talk

NQueens [2006,2007]

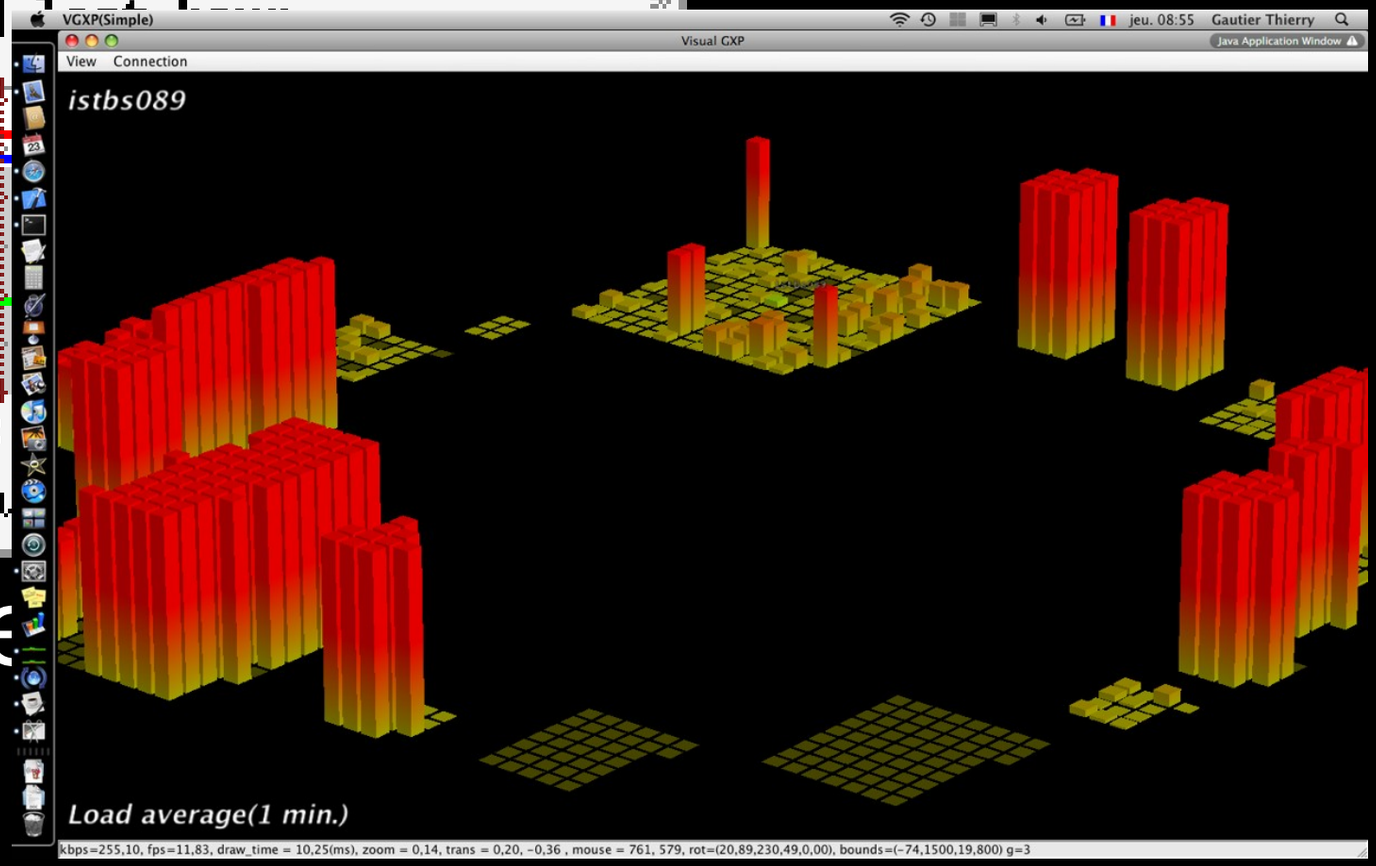


- Grid5000 (French academic national grid)
  - 2006: N=23 in 74min on 1422 cores
  - 2007: N=23 in 35mn 7s on 3654 cores
- Taktuk: fast deployment tool

Monte Carlo / Option Pricing



Grid5000: France



Intrigger: Japan

- 3609 cores: ~2700 Grid5000 ~900 Intrigger
- SSH connection between Japan-France

Physics Simulation

- SOFA: real-time physics engine
- Strongly supported INRIA initiative
- Open Source:
- http://www.sofa-framework.org
- Target application:
- Surgery simulation

Interactive Physical Simulation

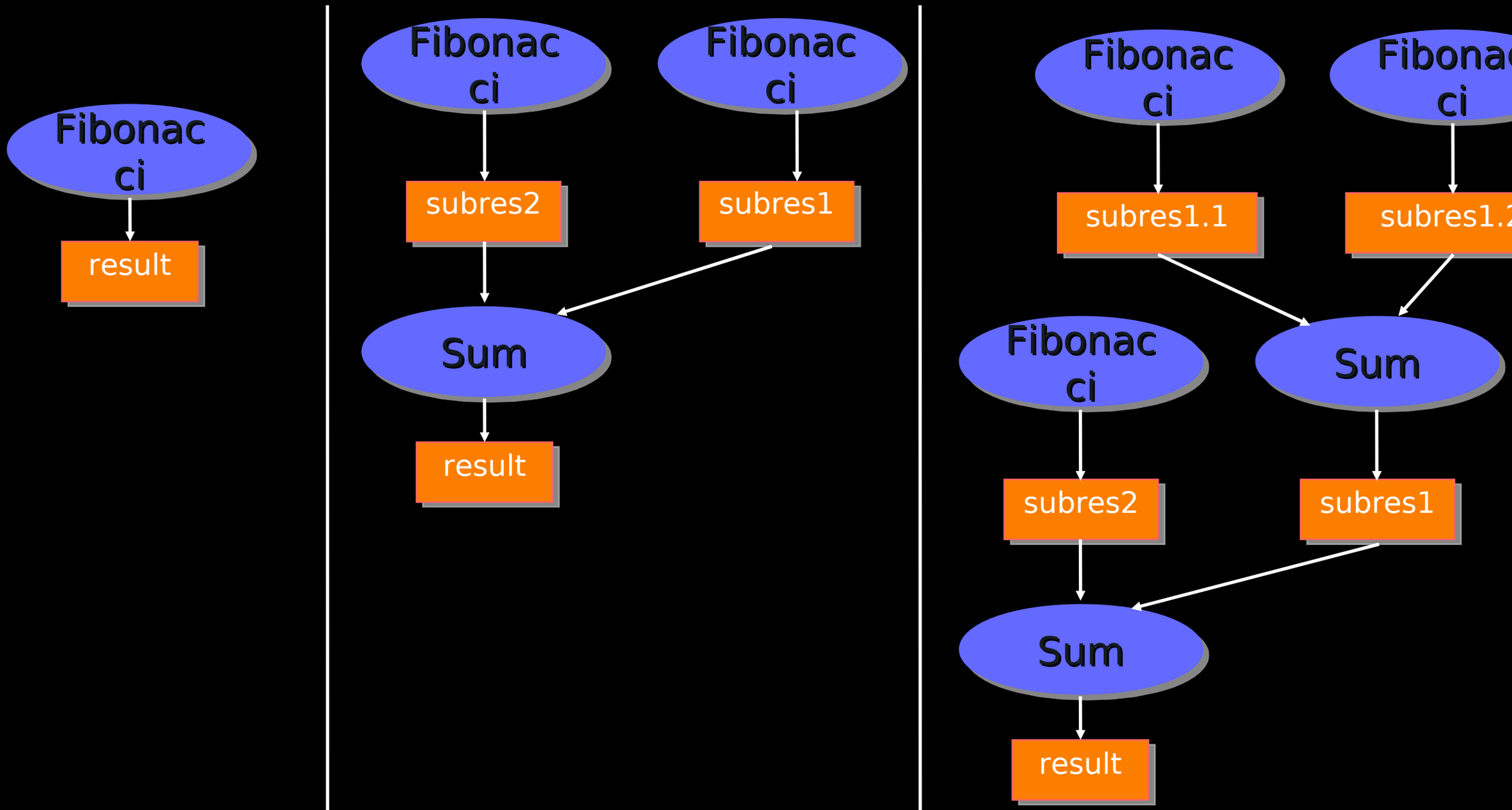
on Multicore Architectures

Communication

- Active message like communication protocol
- Multi-network (TCP, Myrinet, ssh tunnel with TakTuk)
- High capacity to overlap communication by computations
- Original message aggregation protocol

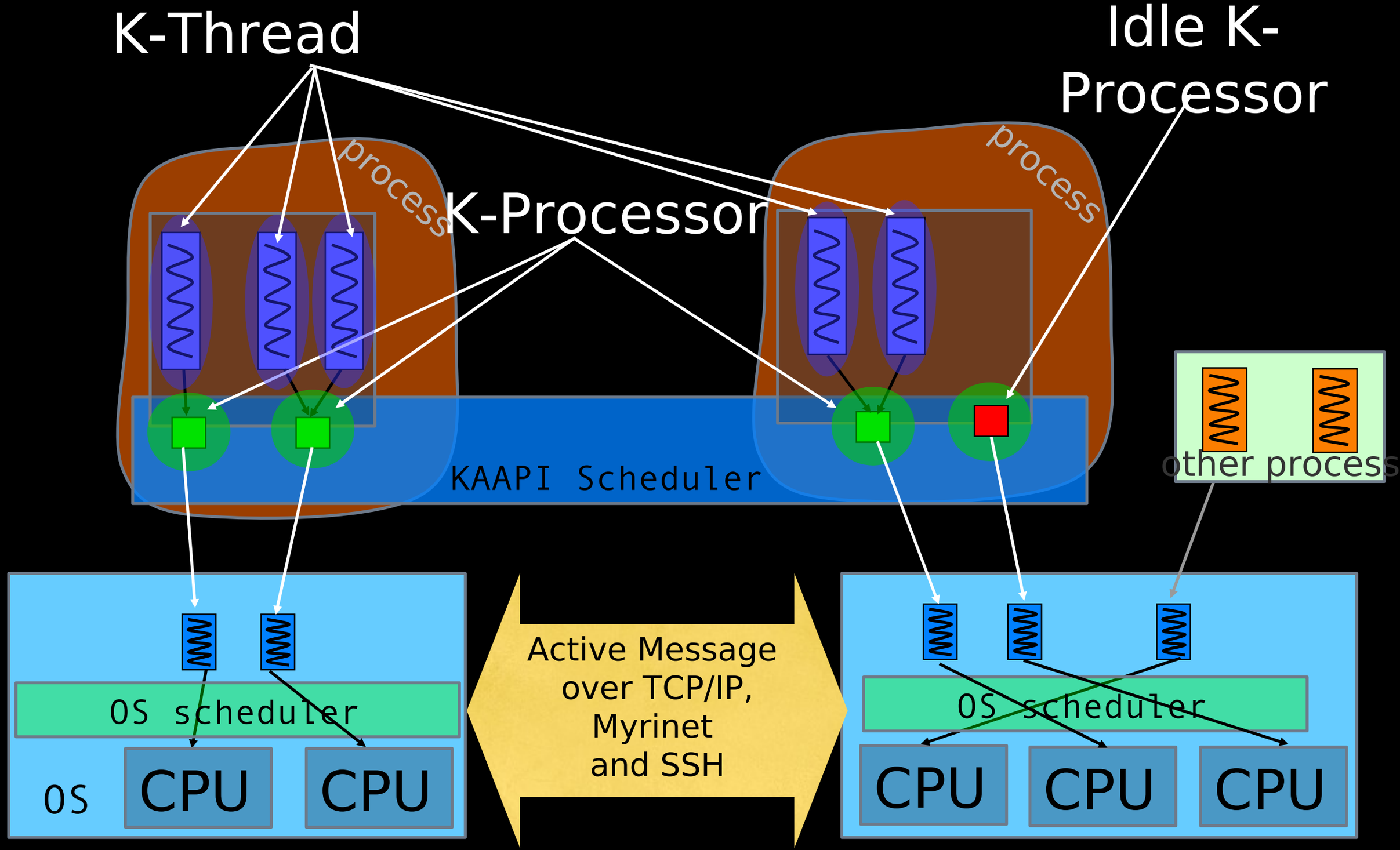

KAAPI

Online construction



Time

2 Level Scheduling



Relaxed semantics

- “Idempotent work stealing” [PPoPP09]
  - Maged M. Michael, Martin T. Vechev,
  - Vijay A. Saraswat (work also on X10 language)
  - avoid CAS in pop operation

More Performance

- Drawback
  - a task is returned (and executed) at least once
  - ... instead of ***exactly once***