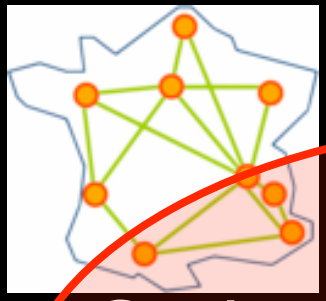
Scheduling parallel program with Work stealing

Vincent Danjean, Vincent.Danjean@imag.fr

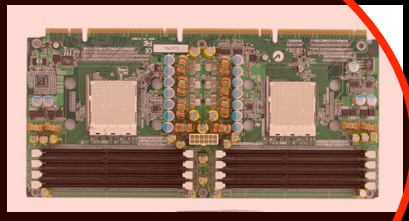
MOAIS project, INRIA Grenoble Rhône-Alpes
Seminar at UFRGS, Porto Alegre, Brazil

Moais Positioning



Grid

Cluster

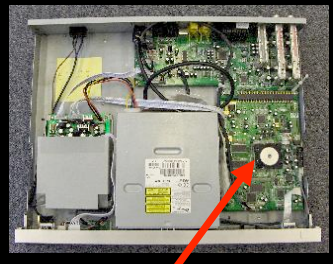


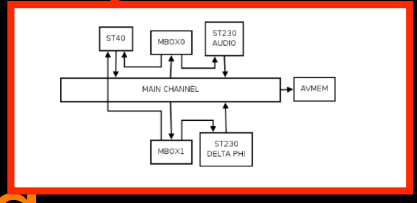
Multicore

KAAPI

GPU

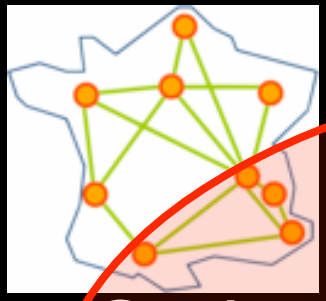
MPSoC





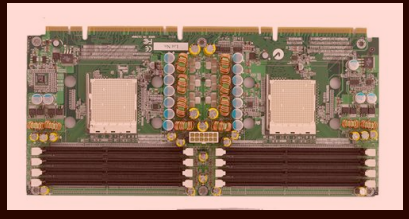
To mutually adapt application and scheduling

Moais Positioning



Grid

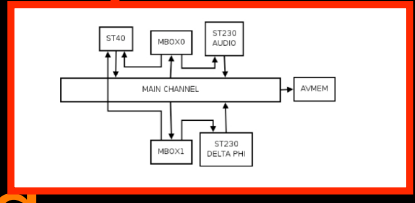
Cluster



Multicore

GPU

MPSoC



KAAPI

To mutually adapt application and scheduling

MOAIS group

- Funded by INRIA, CNRS, UJF, INPG
- Part of LIG (Laboratoire d'Informatique de Grenoble)
  - http://moais.imag.fr
- Jean-Louis Roch: project Leader
- 10 researchers
- 18 PhD students

Research actions

- Scheduling [Denis Trystram]
  - multi-objectif criteria, malleable and moldable model
- Parallel Algorithms [Jean-Louis Roch]
  - adaptive algorithms
- Virtual Reality [Bruno Raffin]
  - interactive simulation
- Runtime for HPC [Thierry Gautier]
  - grid and cluster, multi-processor

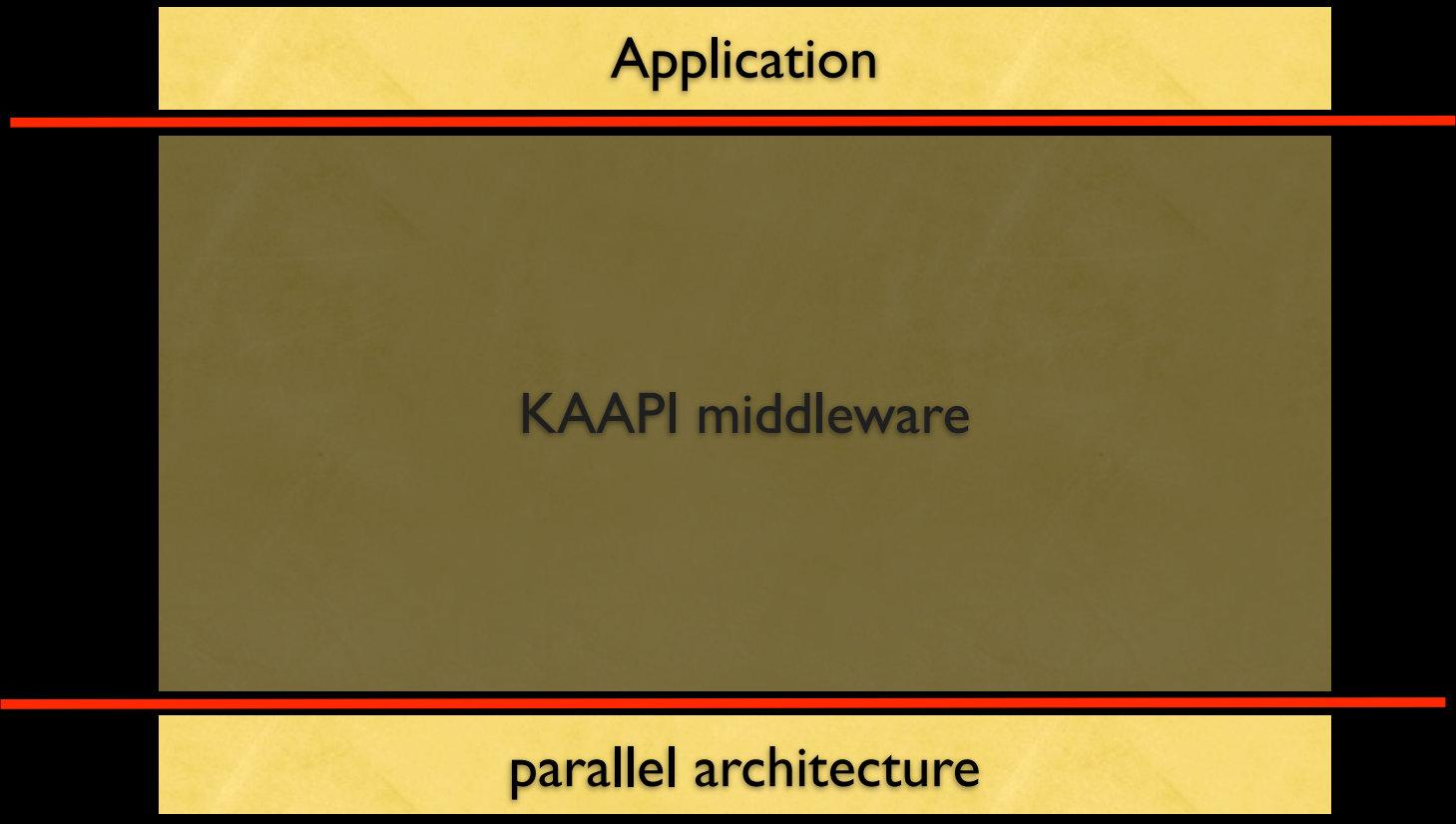
Outline

- Athapascan / Kaapi a software stack
- Foundation of work stealing
- Controls of the overheads
- Experiments with STL algorithms

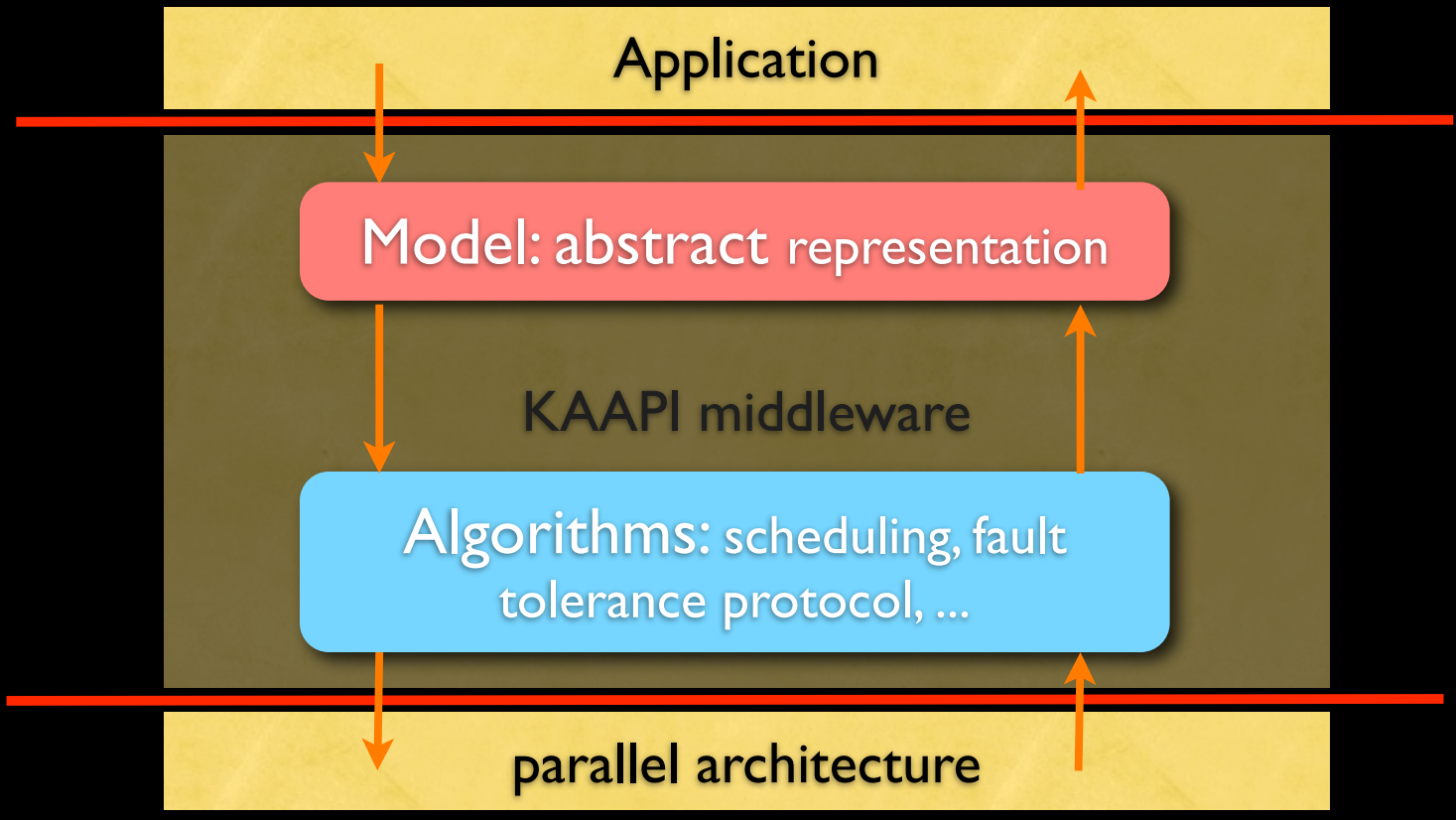
Goal

- Write once, run anywhere... with guaranteed performance
- Problem: heterogeneity
  - variations of the environment (#cores, speed, failure...)
  - irregular computation

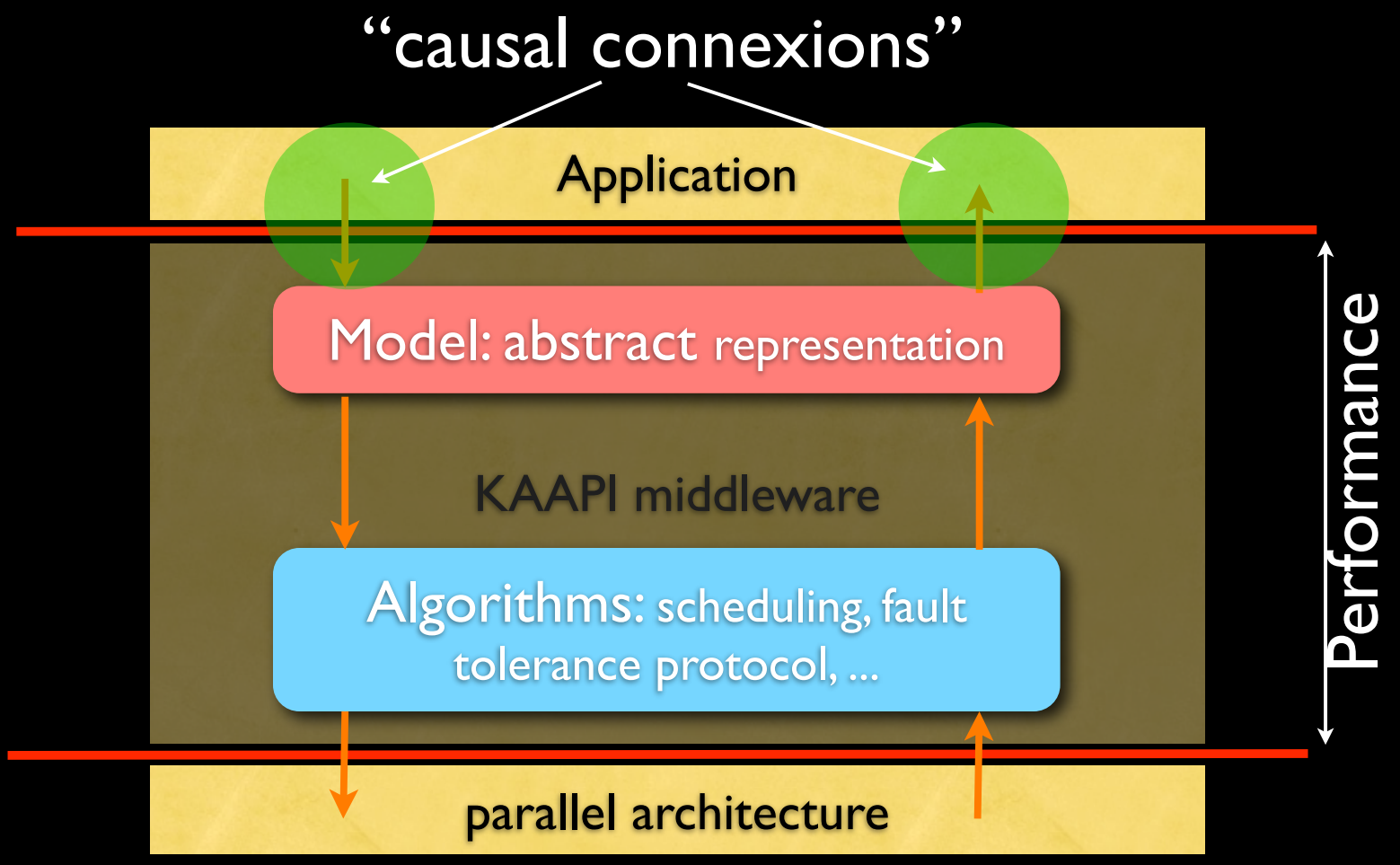
KAAPI Overview



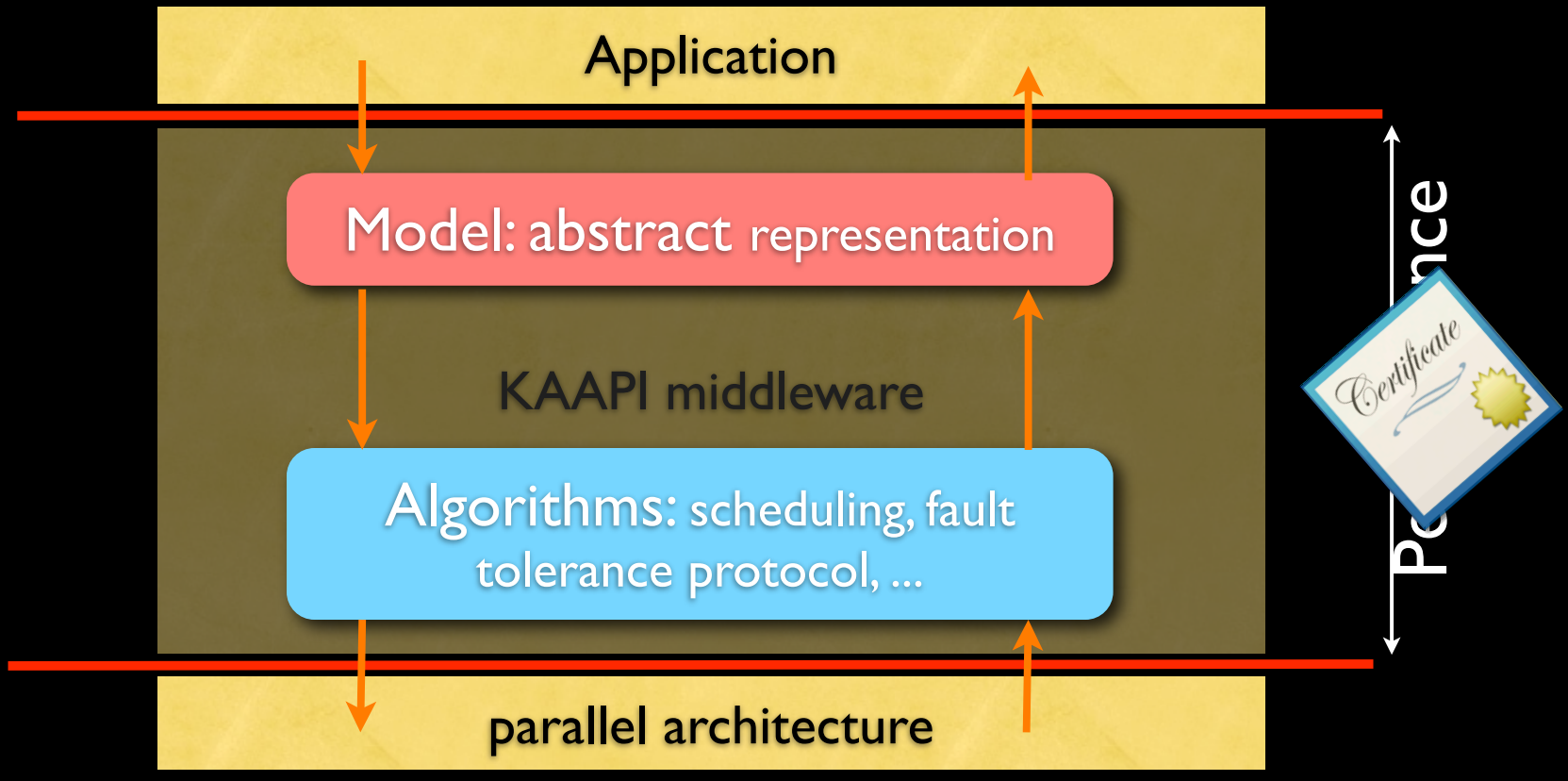
KAAPI Overview



KAAPI Overview



KAAPI Overview



API: Athapascan

- Global address space
  - Creation of objects in a global address space with ‘shared’ keyword
- Task = function call
  - Creation with ‘Fork’ keyword (≠ unix fork) ~ Cilk spawn
  - Tasks only communicate through shared objects
  - Task declares access mode (read, write, concurrent write, exclusive) to shared objects
- Automatic scheduling
  - Work stealing or graph partitioning

➡ ‘Sequential’ semantics

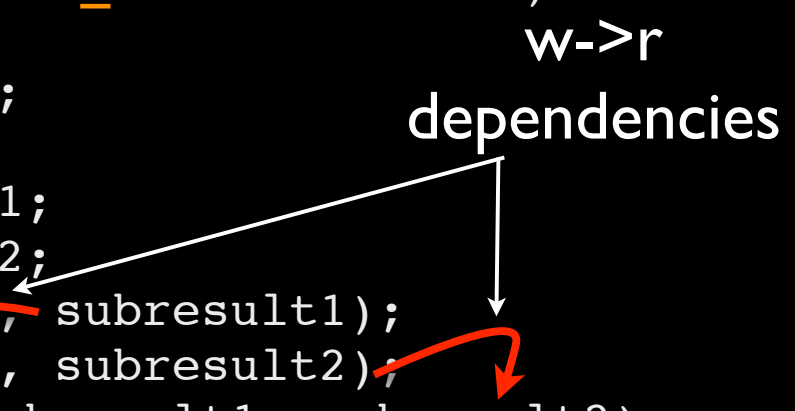
- C++ library, not a language extension
  - C language extension + compiler was prototyped

Fibonacci example

```
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() ); }
}
```

Fibonacci example

```
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() ); }
}
```



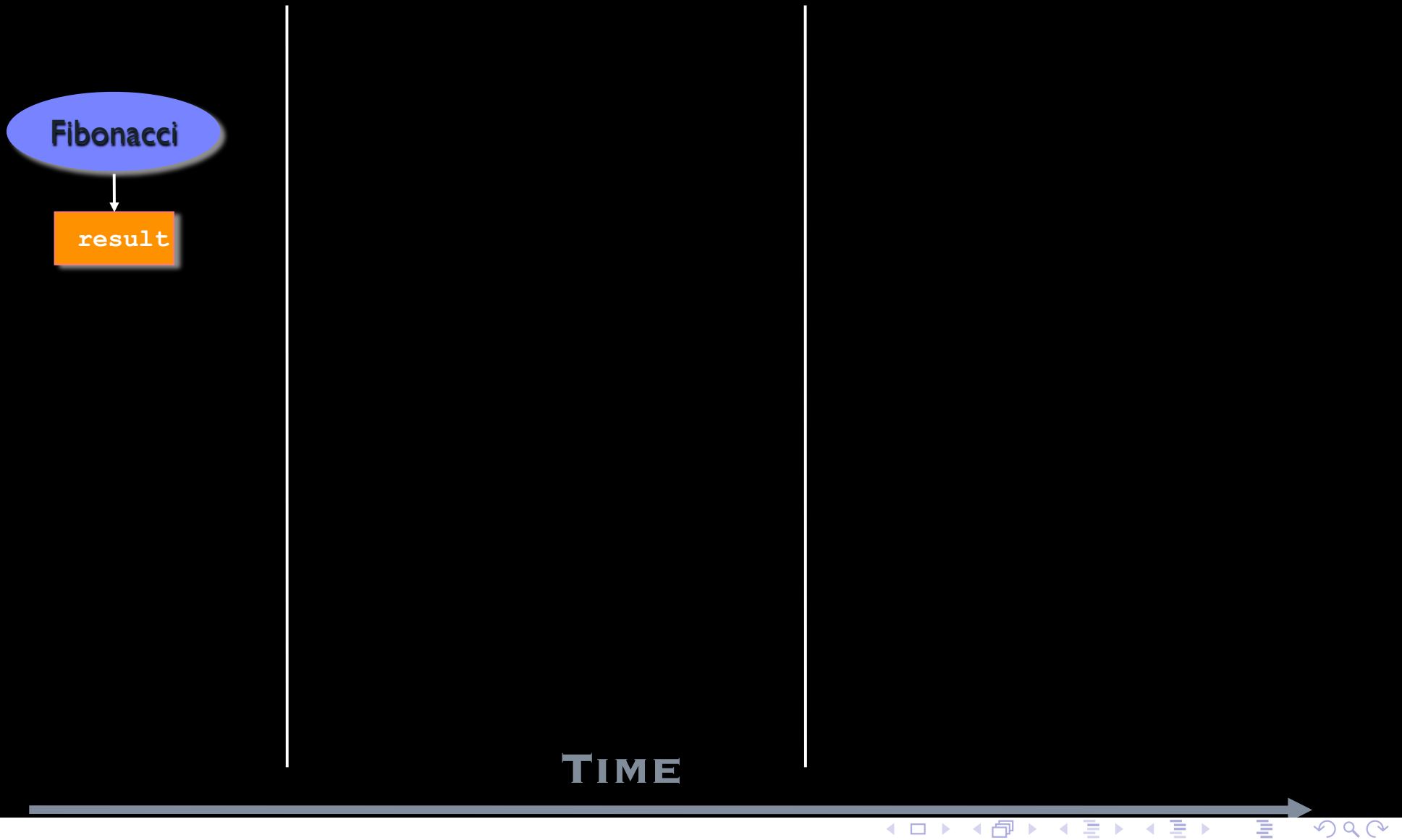
Semantics & C++ Elision

```
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() ); }
}
```

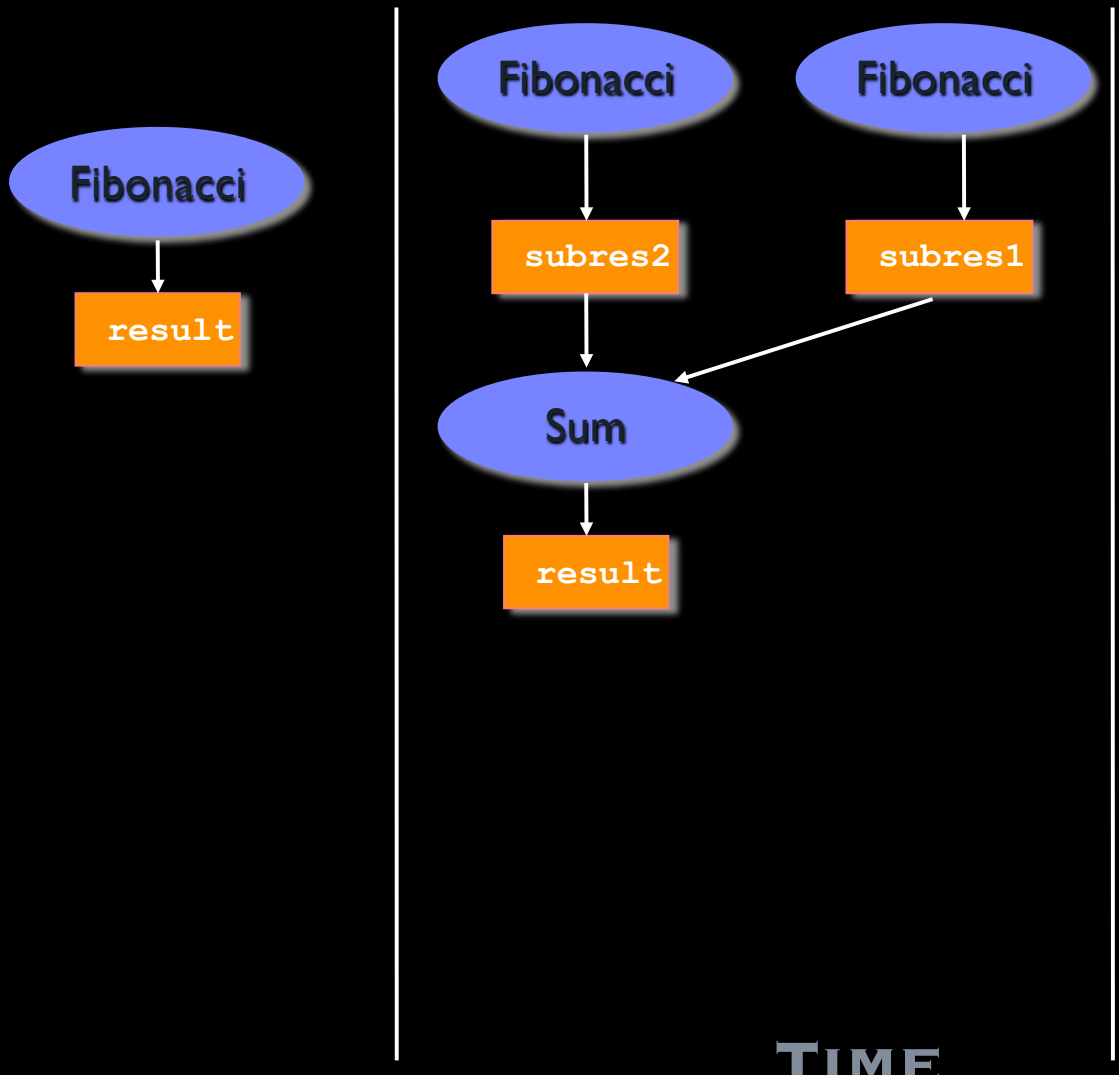
Semantics & C++ Elision

```
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          ; }
}
```

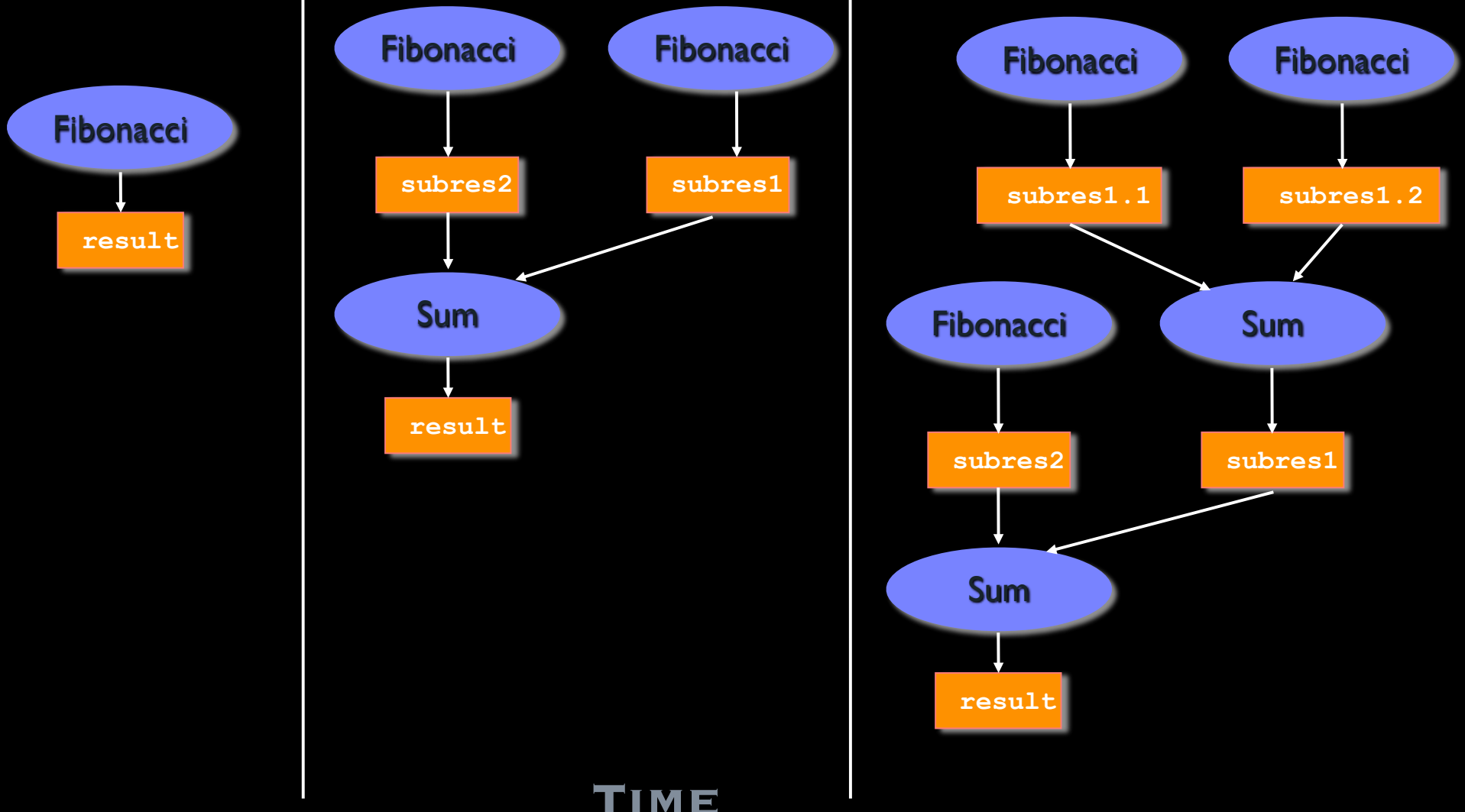
Online construction



Online construction



Online construction



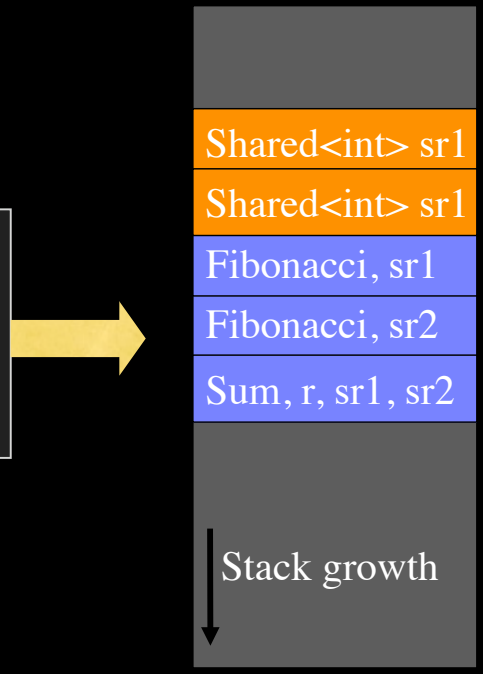
Data Flow Graph Interests

- One abstract representation for
  - Scheduling
    - data flow graph ⇒ precedences graph ⇒ dependences graph
      - classical numerical kernel ⇒ partitioning the dependences graph
  - Explicit data transfer
    - data to be transfered is explicit in the data flow graph
      - automatic overlapping communication by computation
  - Original Fault Tolerant Protocols
    - TIC [IET05, Europar05, TDSC09]
      - coupling scheduling by work stealing with abstract representation
    - CCK [TSI07, MCO08]
      - coordinated checkpointing with partial restart after failure
  - Sabotage Tolerance [PDP09]
    - dynamically adapt the execution to sabotage
- Drawback: complexity to manage it

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

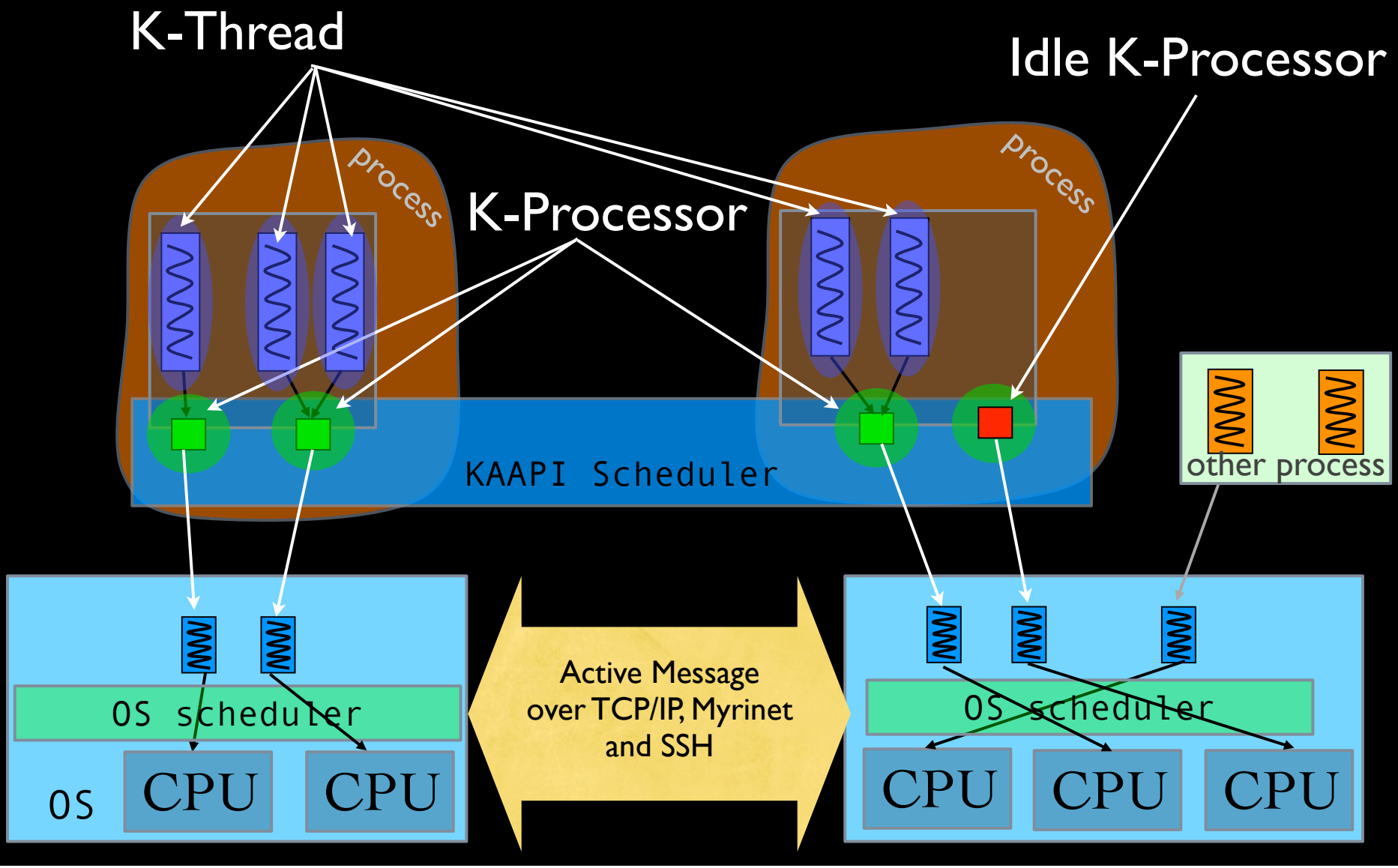
```
a1::Shared<int> subresult1;
a1::Shared<int> subresult2;
a1::Fork<Fibonacci>()(n-1, subresult1);
a1::Fork<Fibonacci>()(n-2, subresult2);
a1::Fork<Sum>()(result, subresult1, subresult2);
```



Execution model

- A (dynamic) set of UNIX processes
  - communication by active message
  - Each process is multithreaded
  - The root process forks the main task
- Threads inside processes are either
  - Performing work
    - excute tasks
  - idle and participate to scheduling
    - then it try to steal work (task) from other threads (work-stealing algorithm)

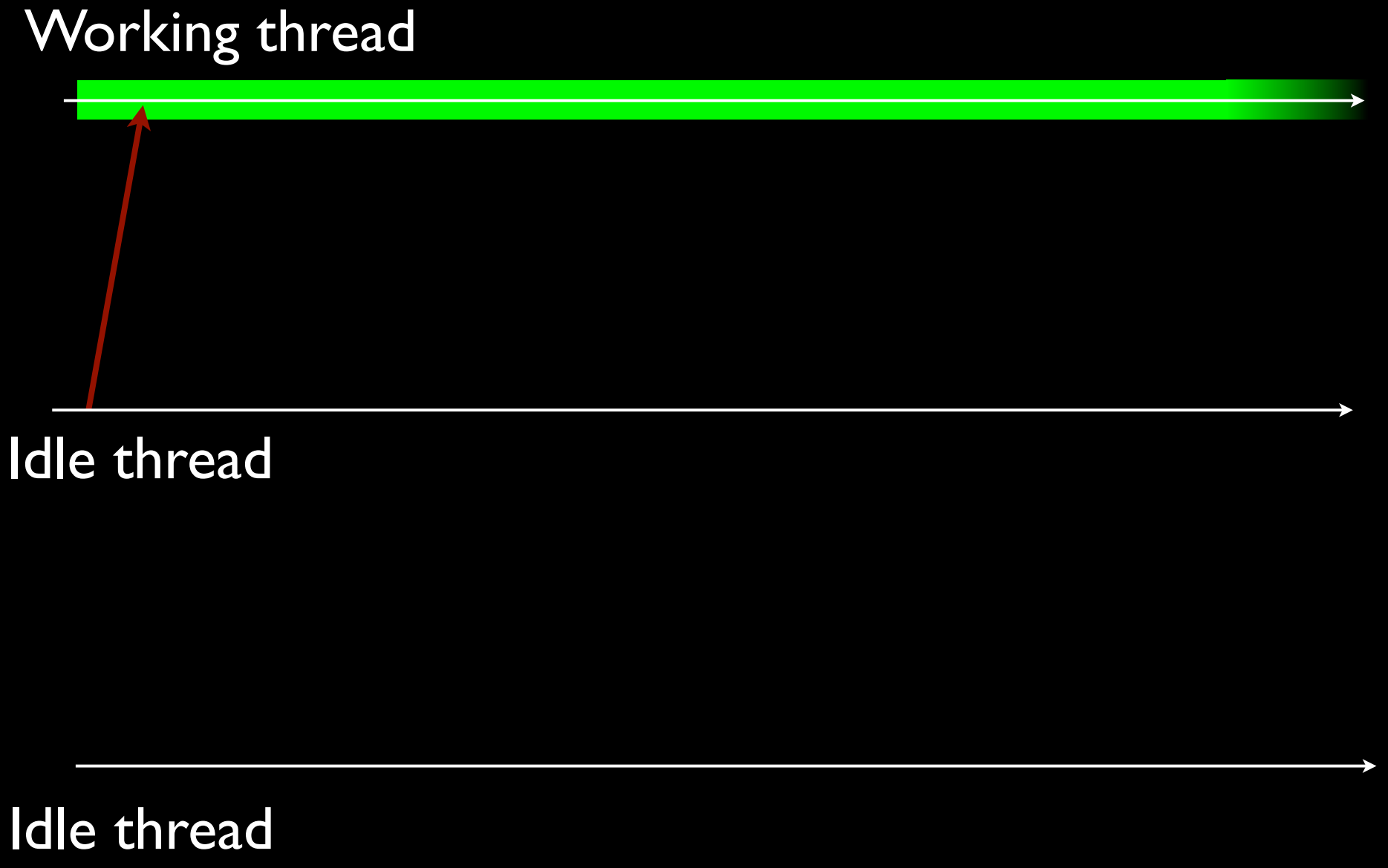
2 Level Scheduling



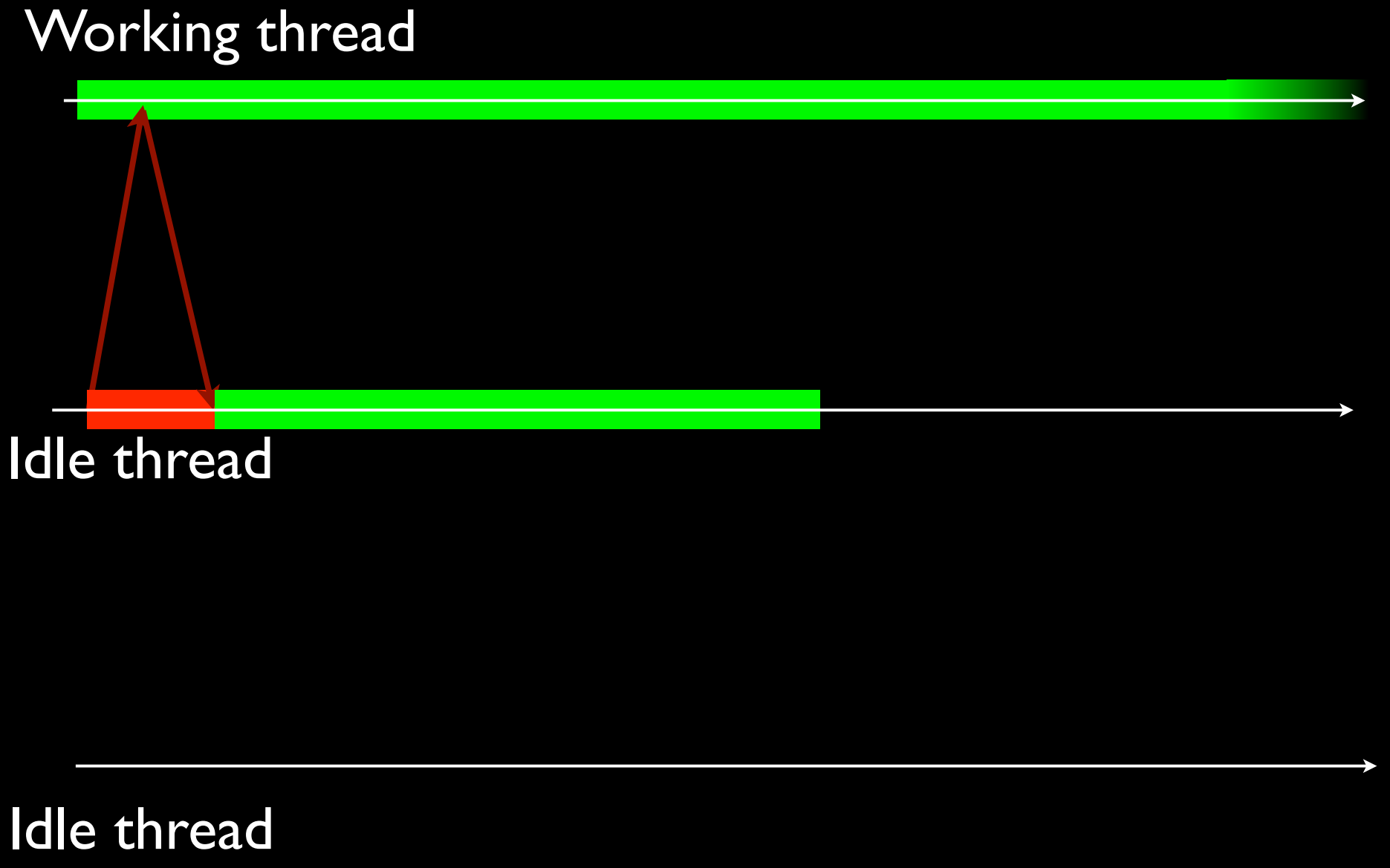
Outline

- Athapascan / Kaapi a software stack
- Foundation of work stealing
- Controls of the overheads
- Experiments with STL algorithms

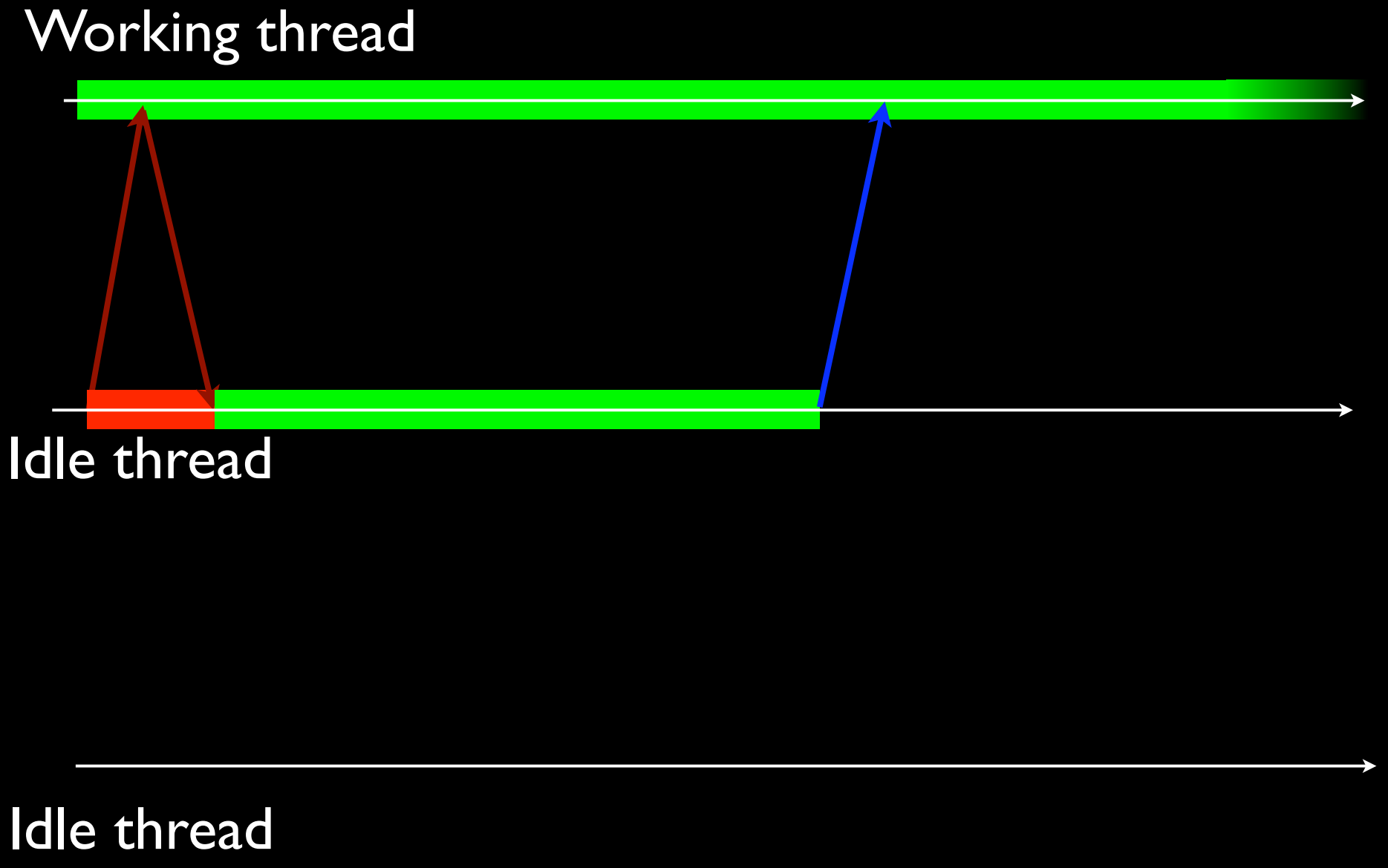
Principle of work stealing



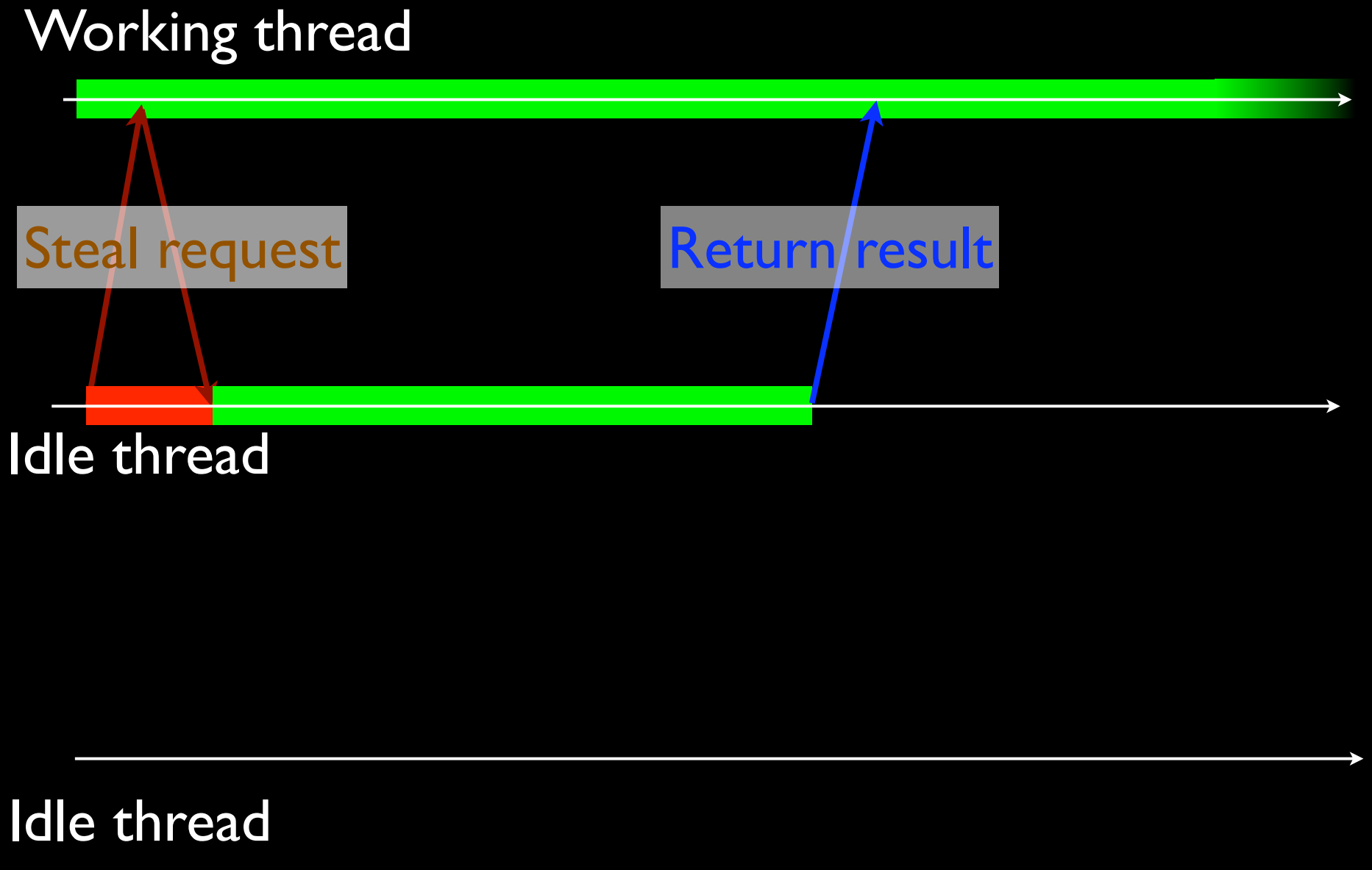
Principle of work stealing



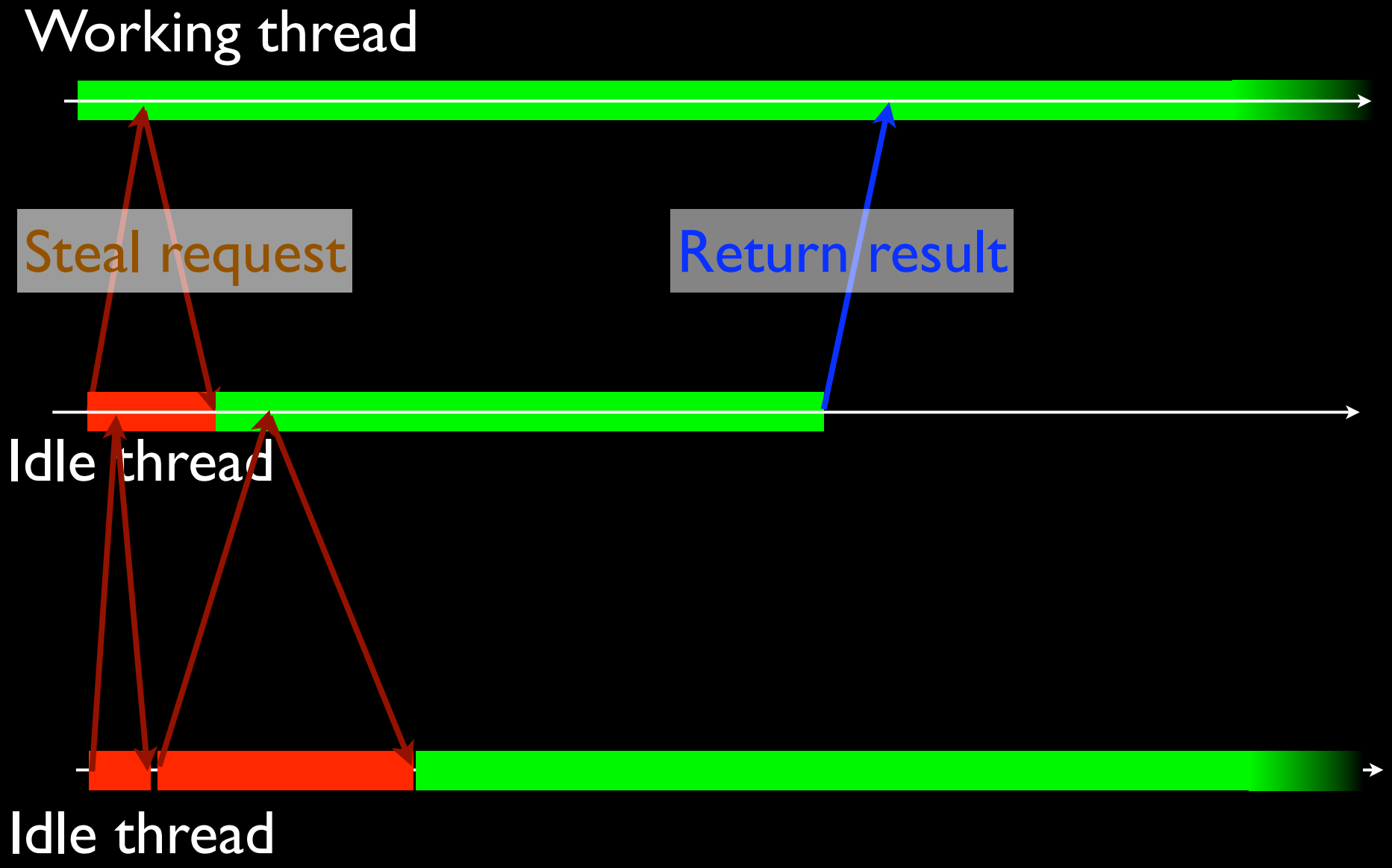
Principle of work stealing



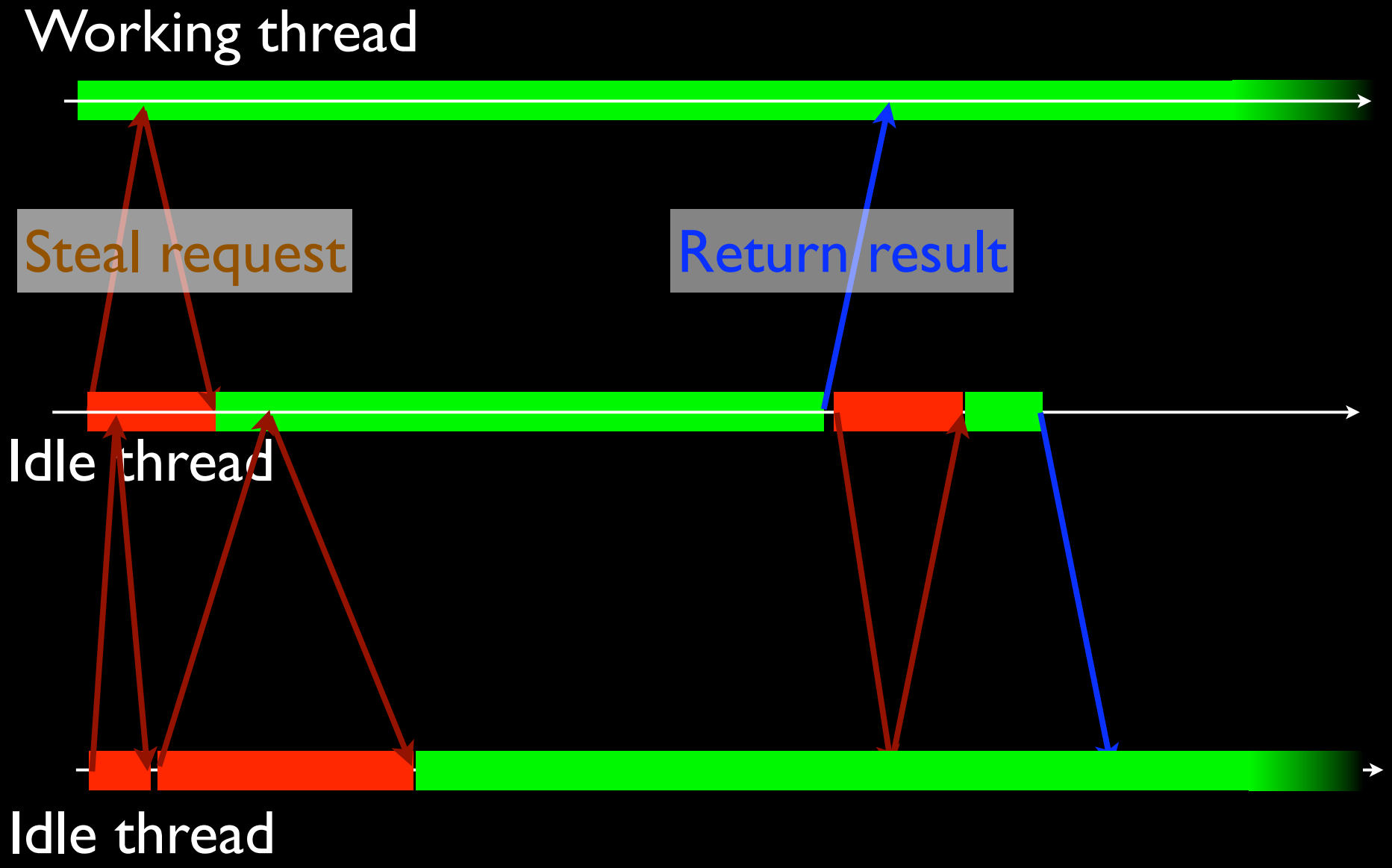
Principle of work stealing



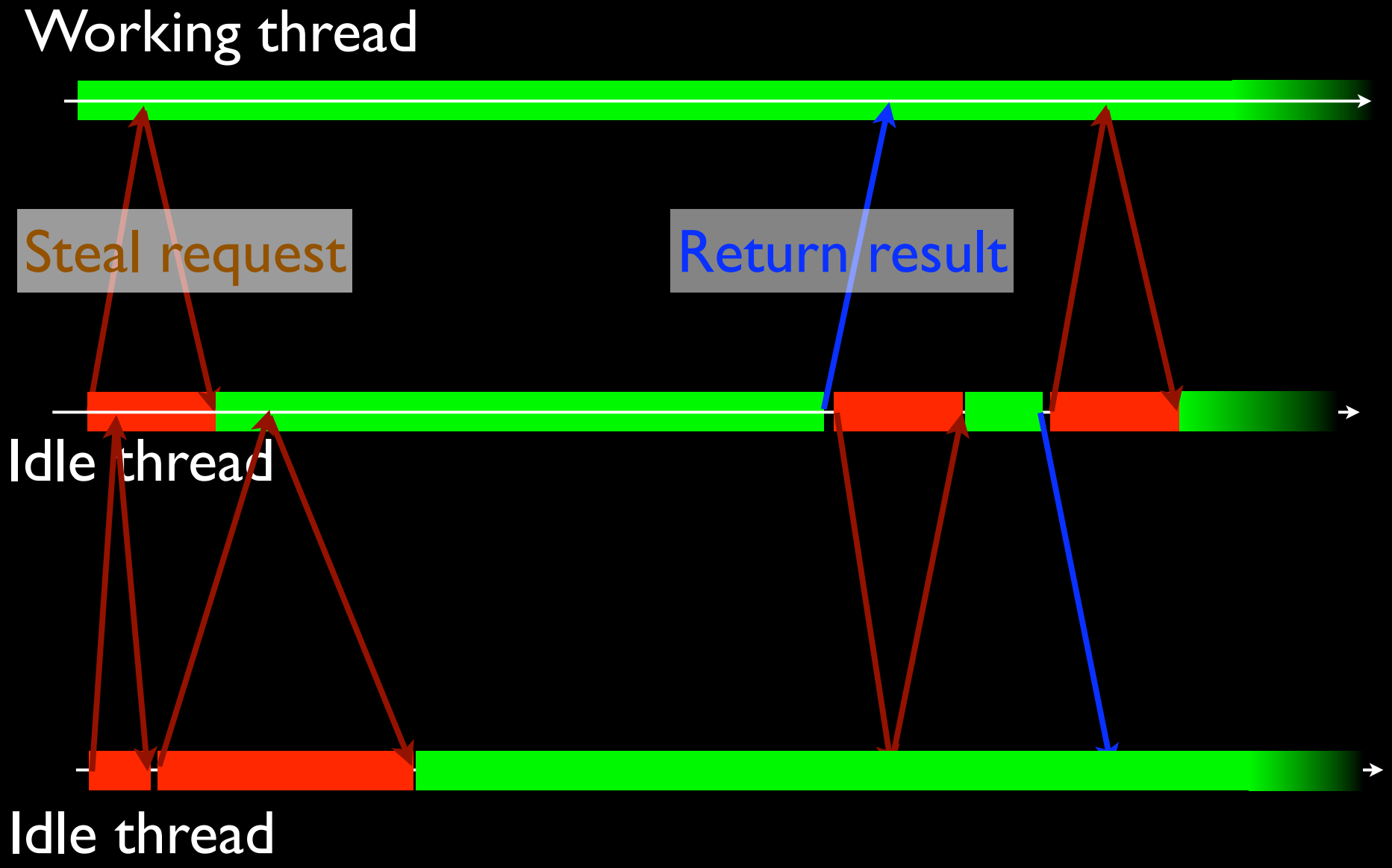
Principle of work stealing



Principle of work stealing



Principle of work stealing



Work stealing queue

- Task: basic unit of computation
- Each thread has its own work queue
  - push(task) -> () : push a task
  - pop() -> task : pop a task
  - push / pop: LIFO order

  } used by the owner thread

- A idle thread can steal task from a victim thread's workqueue
  - steal() -> task : steal a task from the queue
  - push / steal: FIFO order

  } used by the thief thread

- Random selection of victim threads

Assumption/Definitions

- Fork/Join recursive parallel program
  - (formally: strict multithreaded computation)
- Notation:
  - T_{seq} : "sequential work"
  - T_1 : execution time on 1 core of the parallel program, called "work"
  - T_∞ : time of execution on ∞ cores, called the "critical path" or the depth
  - T_p : time of execution on P cores
  - P : the number of processors

Remarks

- $P_\infty \stackrel{def}{=} T_1 / T_\infty$
  - available parallelism (maximum speedup)
  - e.g. Fibonacci(30) = 83040
    - $T1 = 5.285*10^{-2}s$, $T_\infty = 3.08*10^{-7}s$
    - $P_\infty \approx 171591$

- T_1 / T_{seq}
  - measure the work overhead
    - the creation of tasks
    - the work queue management (push/pop)

Theoretical bound

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

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

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

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

- [Blumofe, Leiseron, Focs 94, PPoPP 95], [Arora, Blumofe, Plaxton, SPAA 98], ...

With data flow graph ?

- Previous bounds: “fork/join” program
  - all created tasks are ready
- Not true in data flow program
  - complexity to compute data flow constraints

- Fortunately, the same bound holds
  - [Galilée, Doreille, Cavalheiro, Roch, PACT 98]
  - [Gautier, Roch, Wagner, ICCS2007]
    - in general case, the sequential execution is a valid execution
      - do not compute data flow constraints except on rare events

Remarks

- Other interesting results
  - Space efficient
  - "Same" bounds for heterogenous machines (dynamic speeds) with slightly modified work stealing
    - [Bender, Rabin, SPAA00]

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

- If $P << P_\infty$

➡ quasi linear speed up : $T_p \sim c_1 T_{seq} / P$

- Interest of fine grain parallel algorithm ($T_\infty << T_1$)

➡ $P_\infty >> 1$!! allows wide range of quasi linear speedup

Practical bound

- Cilk 95
  - c_1 ~ from 1 to 25, depends on the application
  - $c_\infty \sim 1$
  - $T_p \approx c_1 T_{seq}/p + c_\infty T_\infty$
- Athapascan/Kaapi
  - 1998: $T_1/T_{seq} \sim 1 - 10000$: bad representation !
  - 2004: $T_1/T_{seq} \sim 1 - 1000$: optimization
  - 2006: $T_1/T_{seq} \sim 1 - 100$: stack representation
  - 2008: $T_1/T_{seq} \sim 1 - 20$: + data flow constraints computed during steal operation

Related work

- Cilk [94, 95, 98]
  - one of the first language: 3 keywords cilk_spawn, cilk_sync, cilk_for
  - sequential semantic !!!!
  - theoretical guaranteed performance
  - shared memory
- Tascell [09]
  - indolent closure creation (on demand)
  - distributed
  - using same idea as a Cilk extension [96] that has never been tested
- X10 [04, 08]
  - experimental language for HPC
  - extend class of parallel program with work stealing strategy
- Satin [01], Capsule [06]

Related work

- Cilk [94, 95, 98]
  - one of the first langu
  - sequential semantic
  - theoretical guarante
  - shared memory
- Tascell [09]
  - indolent closure cre
  - distributed
  - using same idea as a Cilk
- X10 [04, 08]
  - experimental langua
  - extend class of para
- Satin [01], Capsule [06]

- Athapascan/Kaapi [99, 03, 05, 07]
  - macro data flow: 2 keywords Fork, Shared
  - sequential semantic !!!!
  - theoretical guaranteed performance
  - shared & distributed memory
  - fault tolerant protocols
  - fine grain parallelism
    - adaptive algorithm [TSI05 Fr, Europar06, Europar08]
    - cooperative work stealing [09 Fr]

Outline

- Athapascan / Kaapi a software stack
- Foundation of work stealing
- Controls of the overheads
- Experiments with STL algorithms

How to reduce T_{seq}/T_1?

- Why ? => WORK overhead
  - extra instructions from the sequential program
  - especially for short computation (->STL algorithms)
- Three technics
  1. adapt the grain size: stop parallelism after a threshold
     - but: increase 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
     - difficulty due to concurrent operations

How to reduce T_{seq}/T_1?

- Why ? => WORK overhead
  - extra instructions from the sequential program
  - especially for short computation (->STL algorithms)
- Three technics
  1. ~~adapt the grain size: stop parallelism after a threshold~~
     - ~~but: increase 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
     - difficulty due to concurrent operations

Cost of task

Cost of task

- Cost = Creation + Extra arithmetic work

Cost of task

- Cost = Creation + Extra arithmetic work
- Example: prefix computation
  - input: $\{a_i\}$, i=0..n, output: $p_i=\prod_{k=\{0..i\}} a_k$, $\forall$i=0..n

Cost of task

- Cost = Creation + Extra arithmetic work
- Example: prefix computation
  - input: $\{a_i\}$, i=0..n, output: $p_i=\prod_{k=\{0..i\}} a_k$, $\forall$i=0..n
  - sequential work:

$$T_{seq}=n$$

Cost of task

- Cost = Creation + Extra arithmetic work
- Example: prefix computation
  - input: $\{a_i\}$, i=0..n, output: $p_i=\prod_{k=\{0..i\}} a_k$, $\forall i=0..n$
  - sequential work:

$$T_{seq}=n$$

  - Parallel program => Ladner-Fisher (divide & conquer)

$$T_\infty=2\log_2 n,\ T_1=2n$$

Cost of task

- Cost = Creation + Extra arithmetic work
- Example: prefix computation
  - input: $\{a_i\}$, i=0..n, output: $p_i=\prod_{k=\{0..i\}} a_k$, $\forall$i=0..n
  - sequential work:

$$T_{seq}=n$$

  - Parallel program => Ladner–Fisher (divide & conquer)

$$T_\infty=2\log_2 n,\ T_1=2n$$

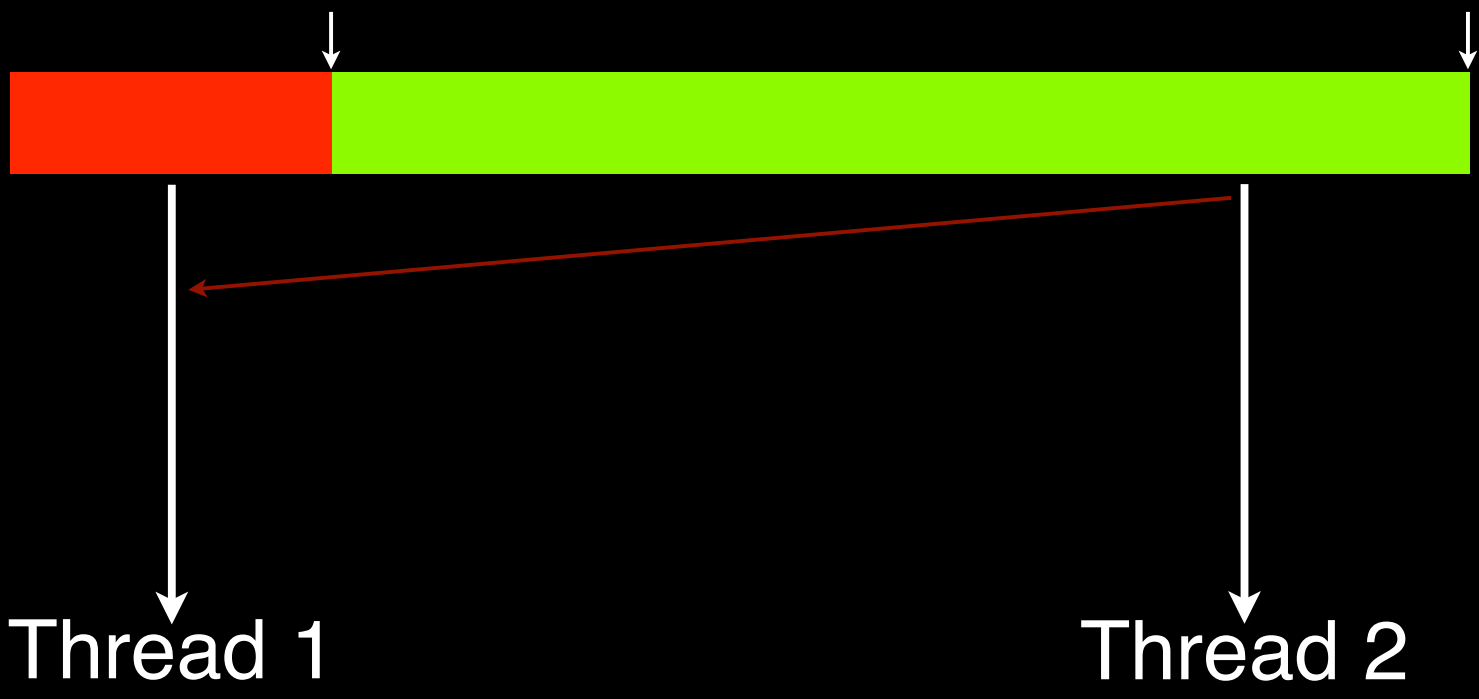
  - Fish's lower bound: any parallel algorithm with critical path $\log_2 n$ requires at least $4n$ operations

Adaptive algorithm

- [Roch, Traoré 07], [Roch, Traoré, Gautier 08]
  - Principe: create tasks when processors are idle !
- Task = apply $F(a_i)$ for all elements a_i of an array

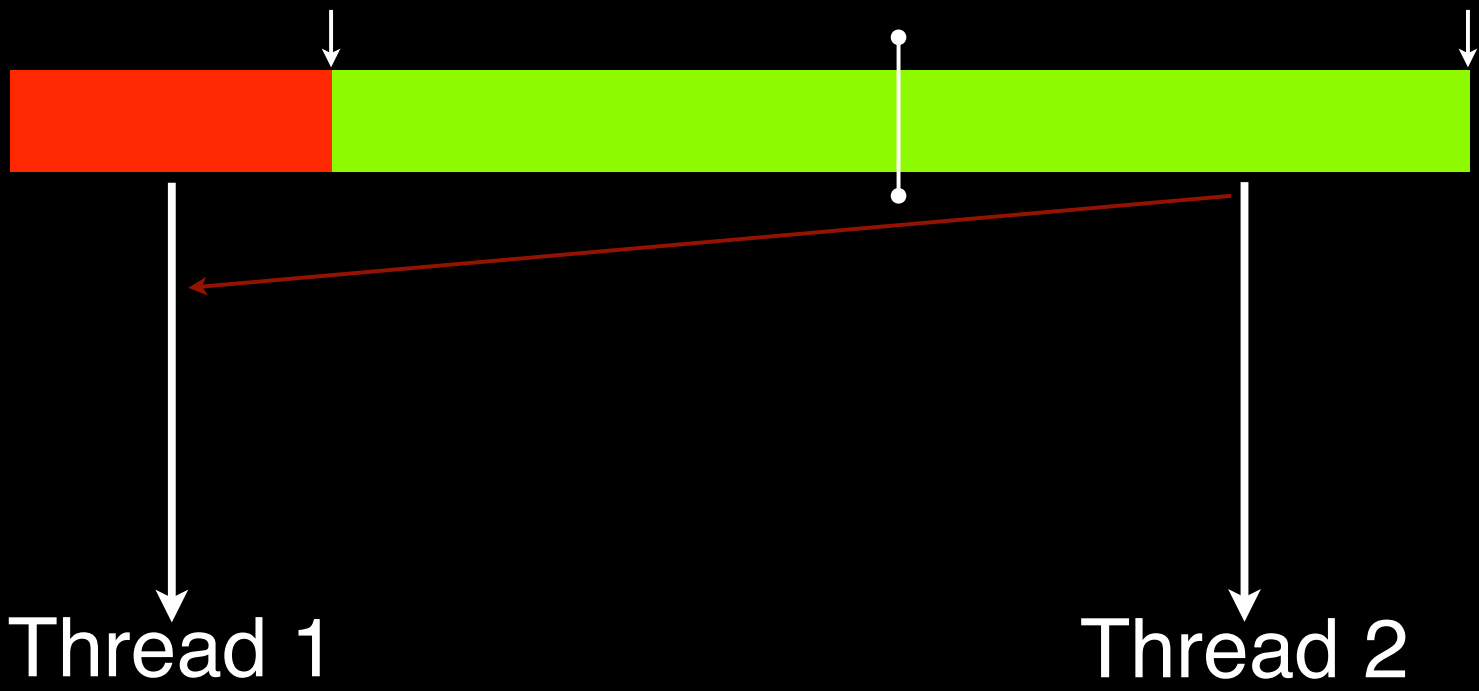
Adaptive algorithm

- [Roch, Traoré 07], [Roch, Traoré, Gautier 08]
  - Principe: create tasks when processors are idle !
- Task = apply $F(a_i)$ for all elements a_i of an array



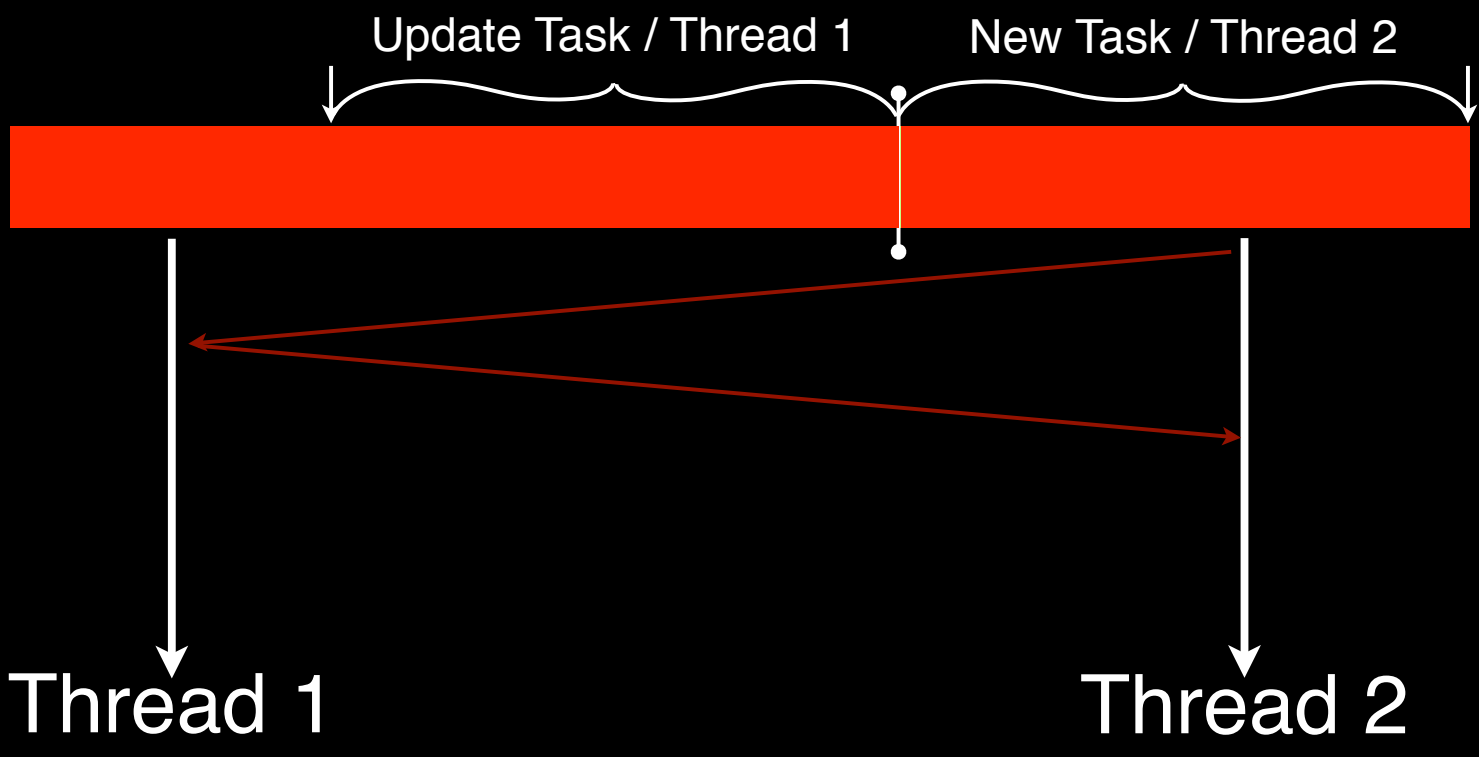
Adaptive algorithm

- [Roch, Traoré 07], [Roch, Traoré, Gautier 08]
  - Principe: create tasks when processors are idle !
- Task = apply $F(a_i)$ for all elements a_i of an array



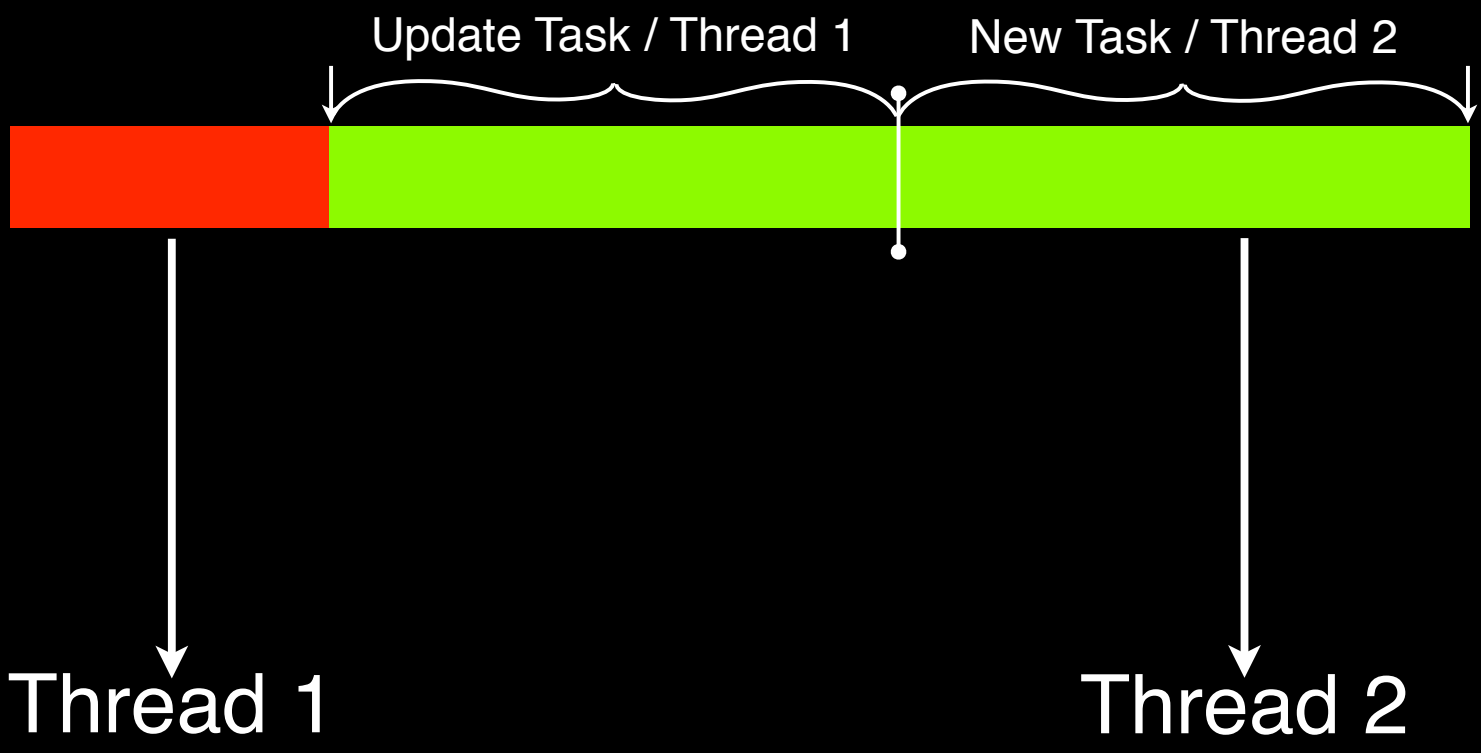
Adaptive algorithm

- [Roch, Traoré 07], [Roch, Traoré, Gautier 08]
  - Principe: create tasks when processors are idle !
- Task = apply $F(a_i)$ for all elements a_i of an array



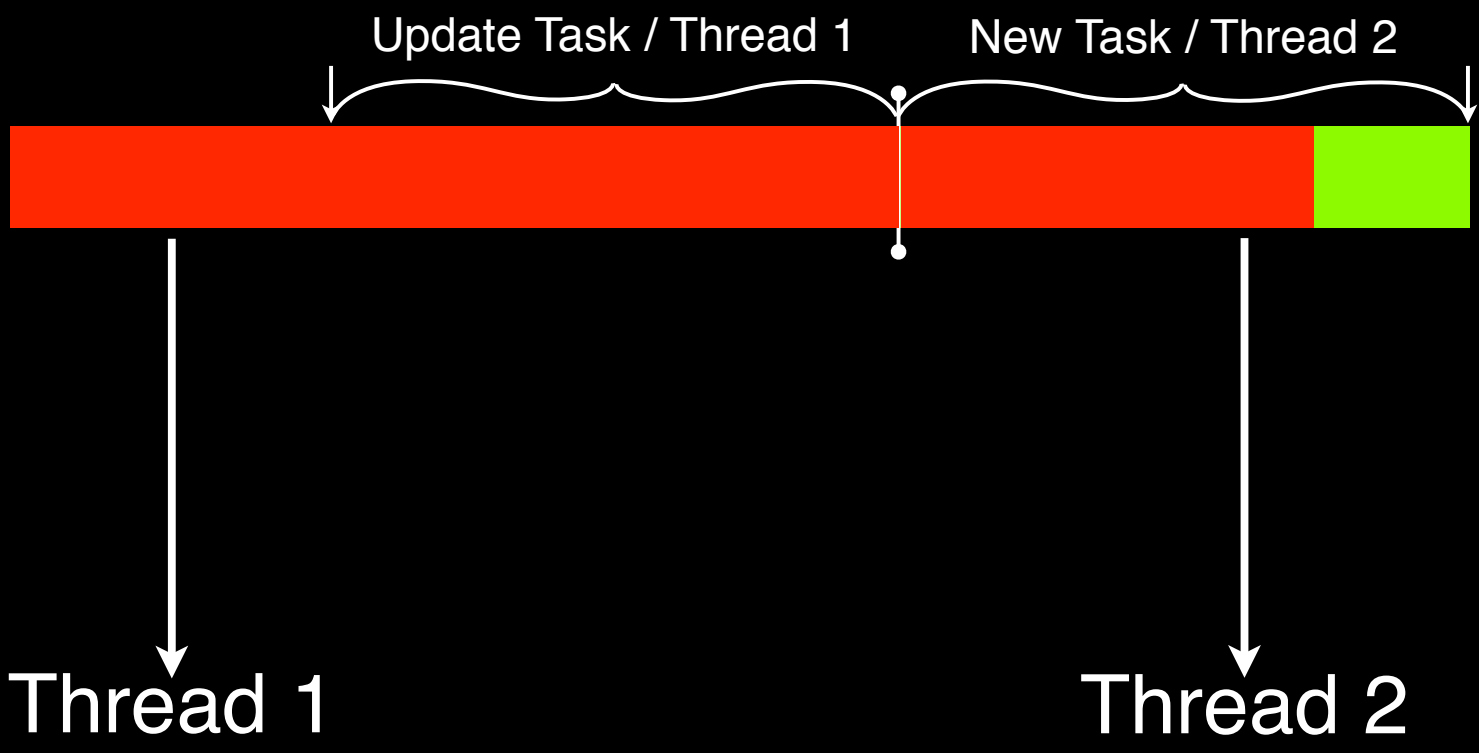
Adaptive algorithm

- Case of heterogeneous speed
  - Thread 2 is slower or as more work to do



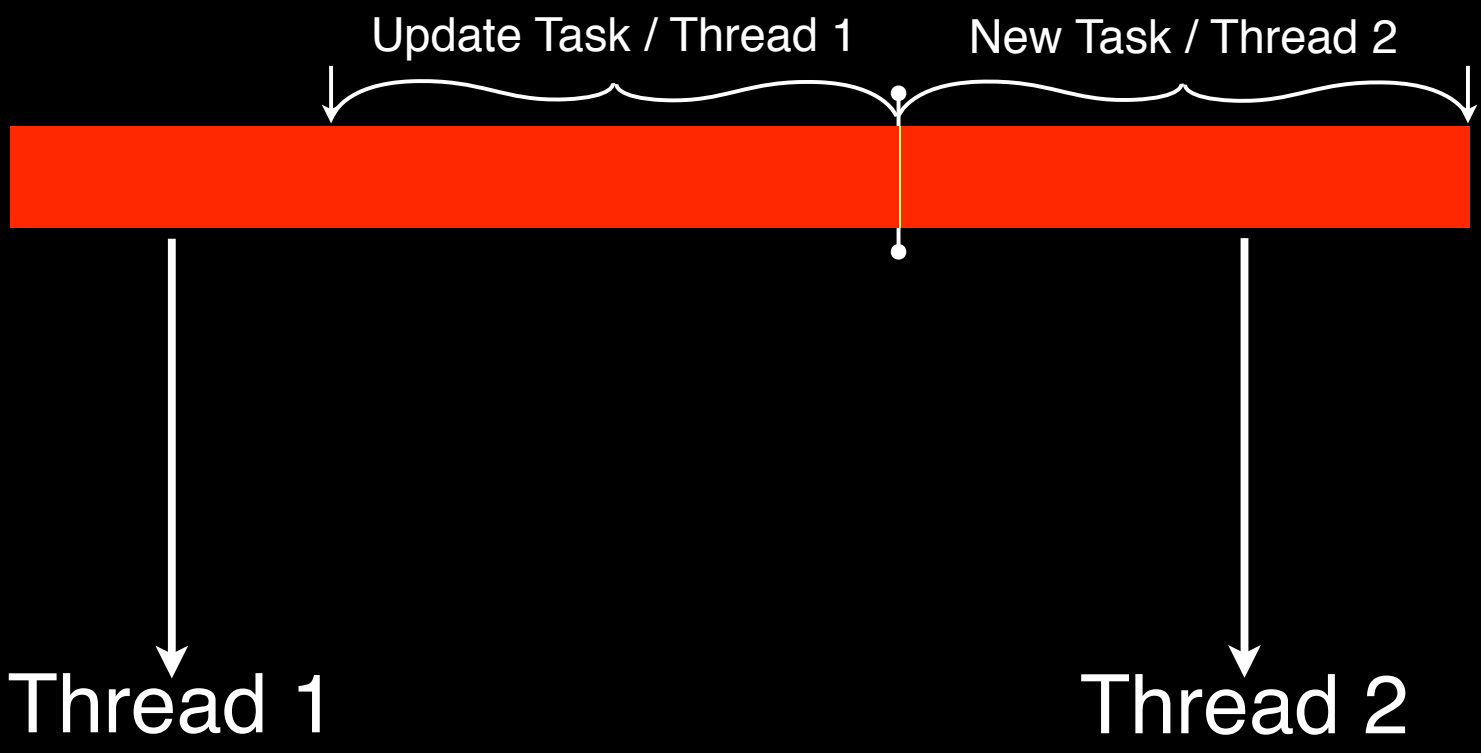
Adaptive algorithm

- Case of heterogeneous speed
  - Thread 2 is slower or as more work to do



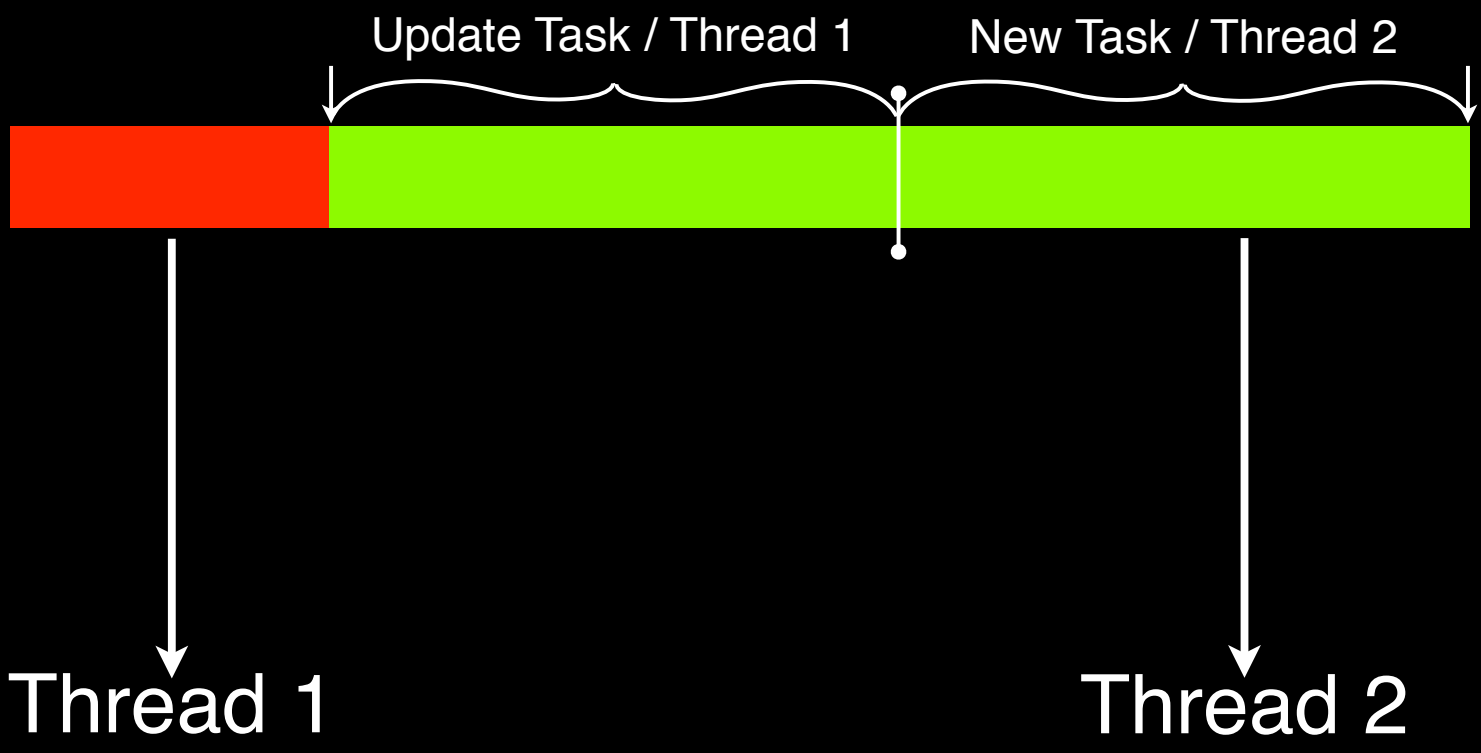
Adaptive algorithm

- Case of heterogeneous speed
  - Thread 2 is slower or as more work to do



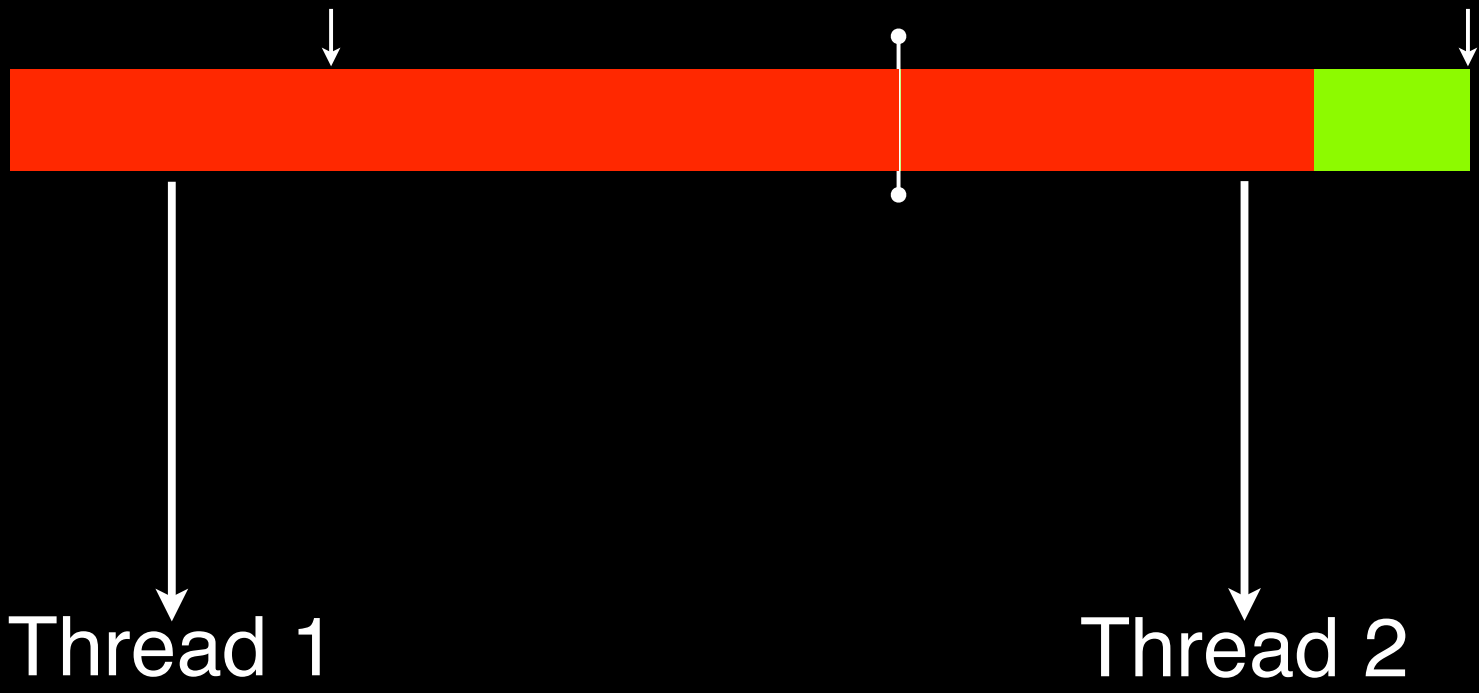
Adaptive algorithm

- Case of heterogeneous speed
  - Thread 2 is slower or as more work to do



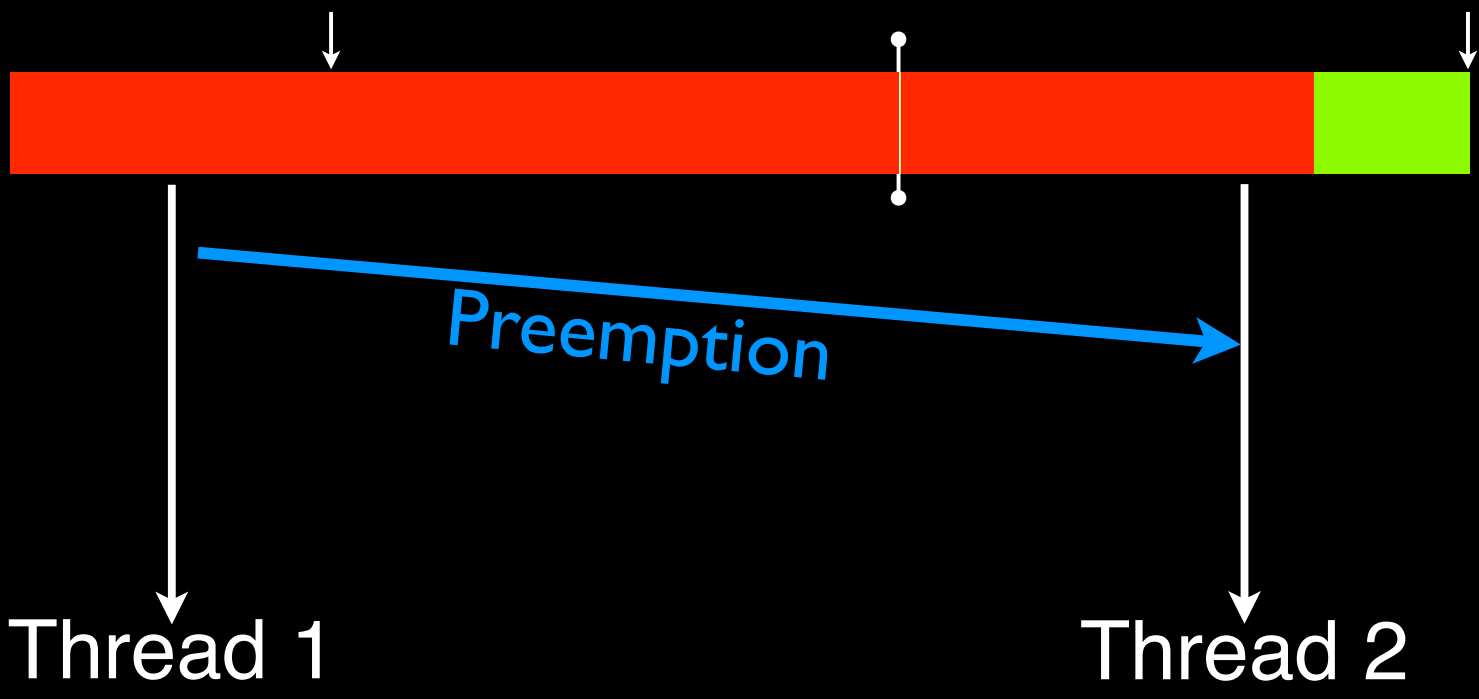
Adaptive algorithm

- Case of heterogeneous speed
  - Thread 2 is slower or as more work to do



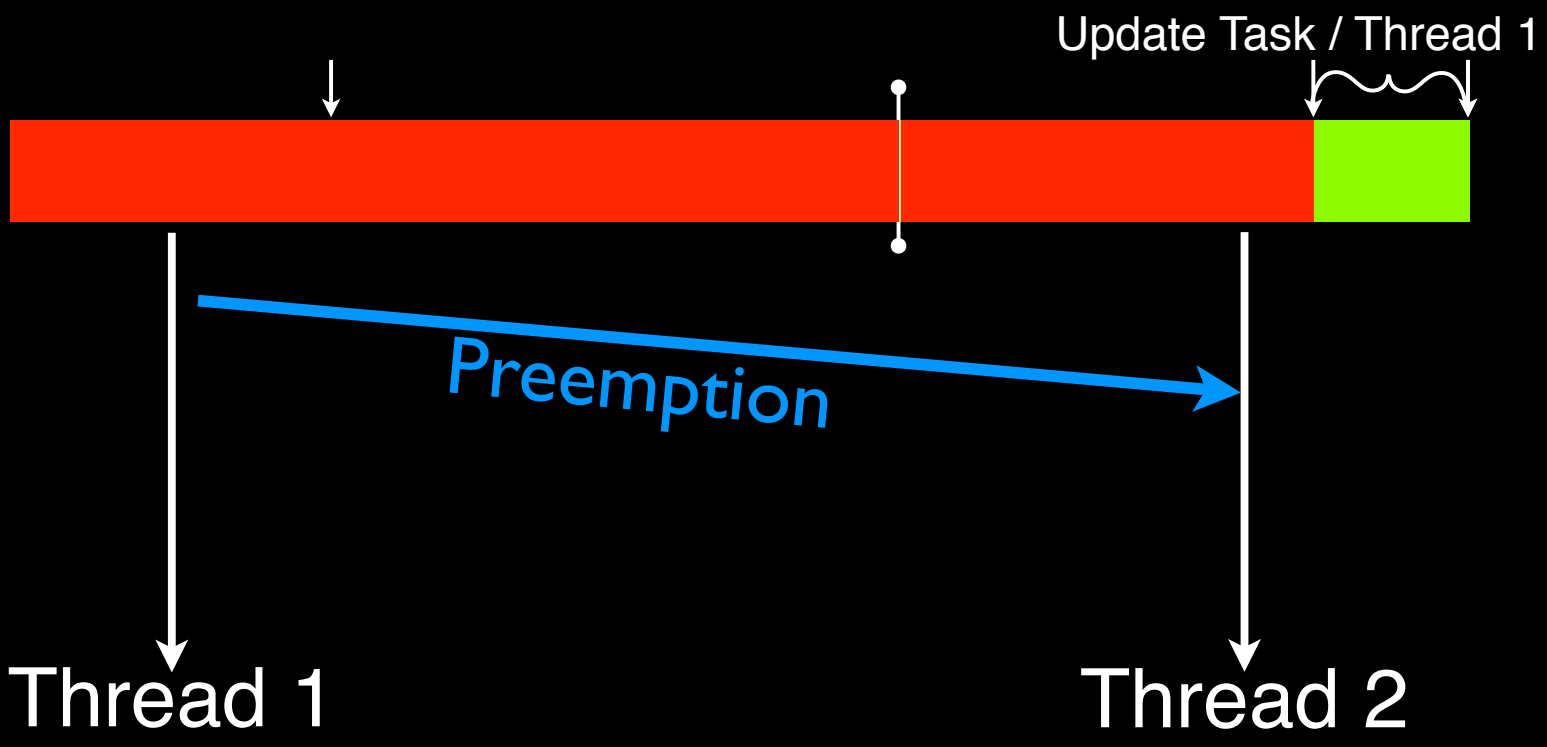
Adaptive algorithm

- Case of heterogeneous speed
  - Thread 2 is slower or as more work to do



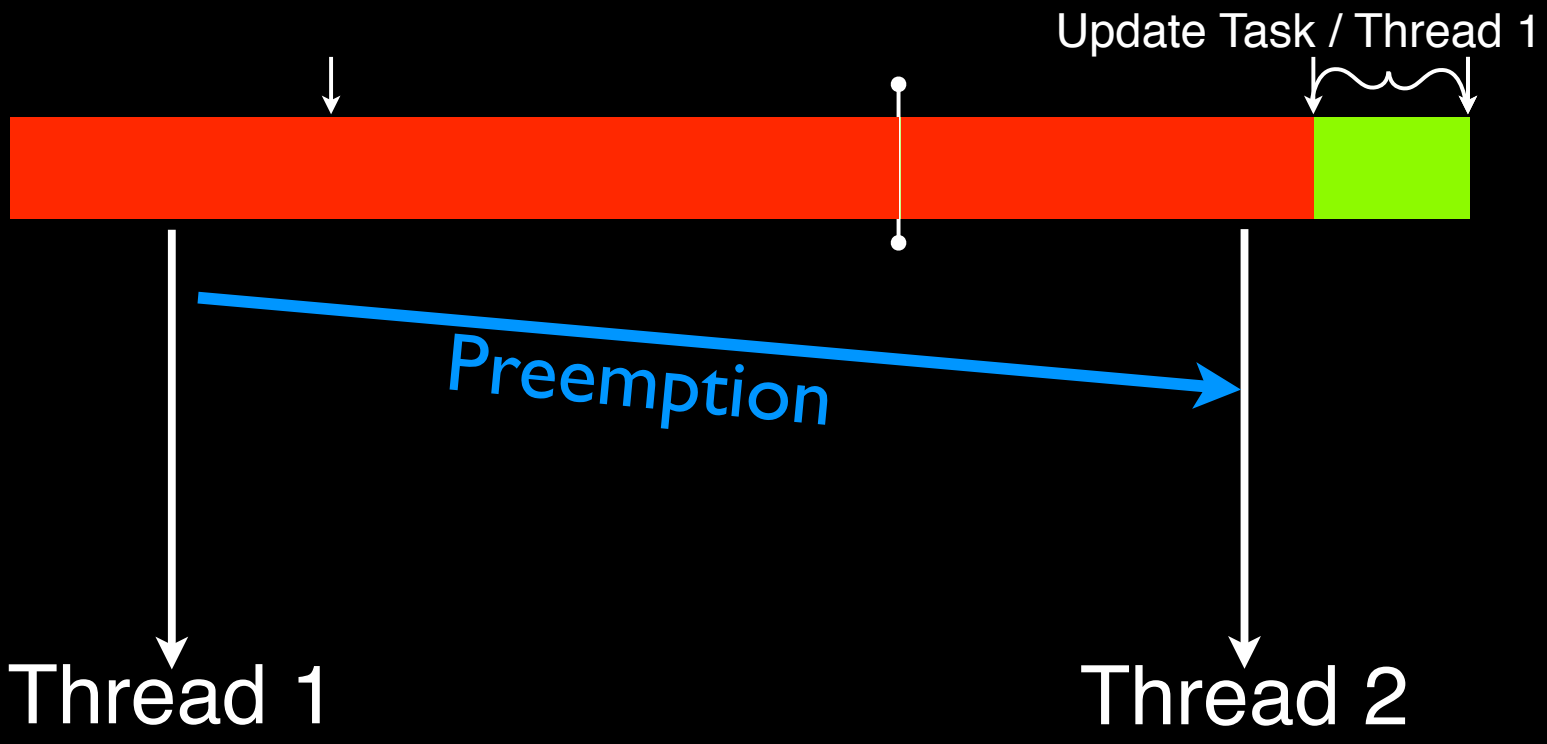
Adaptive algorithm

- Case of heterogeneous speed
  - Thread 2 is slower or as more work to do



Adaptive algorithm

- Case of heterogeneous speed
  - Thread 2 is slower or as more work to do
  - We always use preemption: main thread (sequential algorithm) preempts thieves



Algorithmic point of view

- 2 Algorithms
  - Sequential: efficiency
  - Parallel: to split work & merge partial results
- Scheduler
  - interleaved execution of the two algorithms depending on the idle CPUs
  - robust to heterogeneous processors

Workqueue optimization

- 3 operations
  - push / pop / steal
- 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]:
    - without limitation (other than hardware)

➡ C O S T L Y 'cas' operation [PPoPP09]

Role of the 'CAS'

- Consensus between N-thieves and the victim
  - In theory, not possible (if wait free) with less powerful synchronization primitive
  - e.g.: atomic register read/write, test&set
- How to avoid 'CAS' ?

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

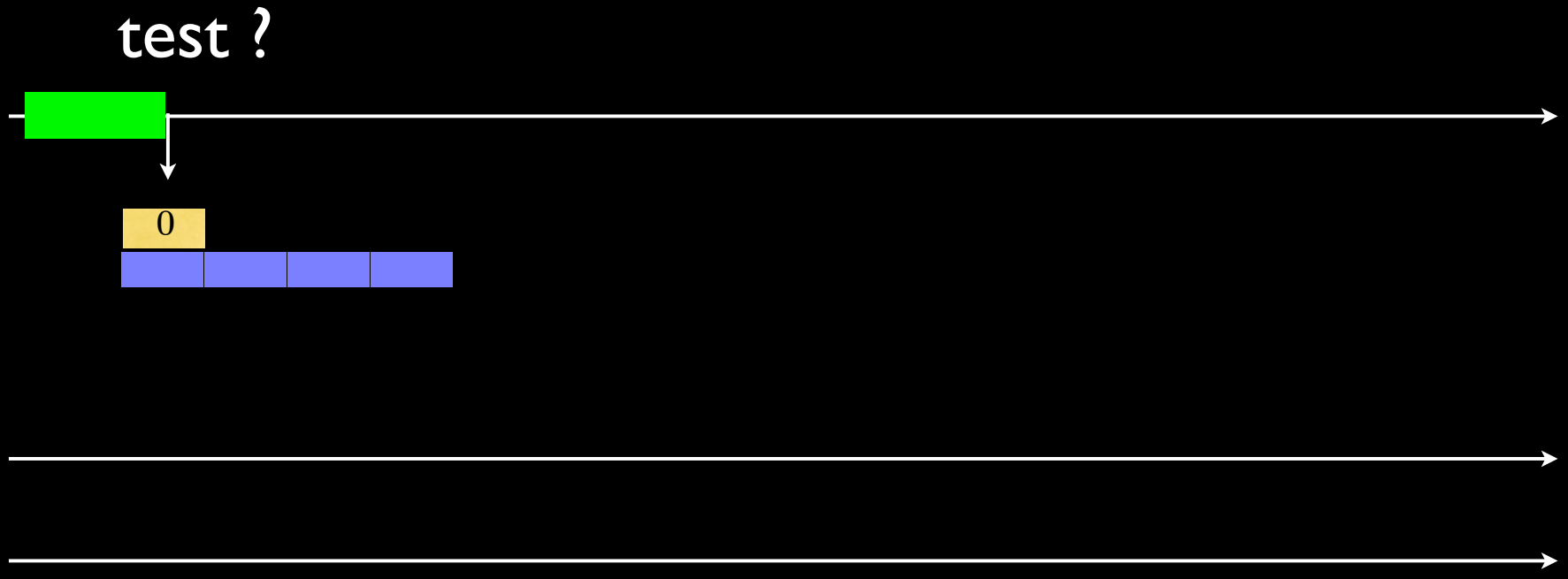
- Drawback
  - a task is returned (and executed) at least once
  - ... instead of exactly once

Cooperative approach

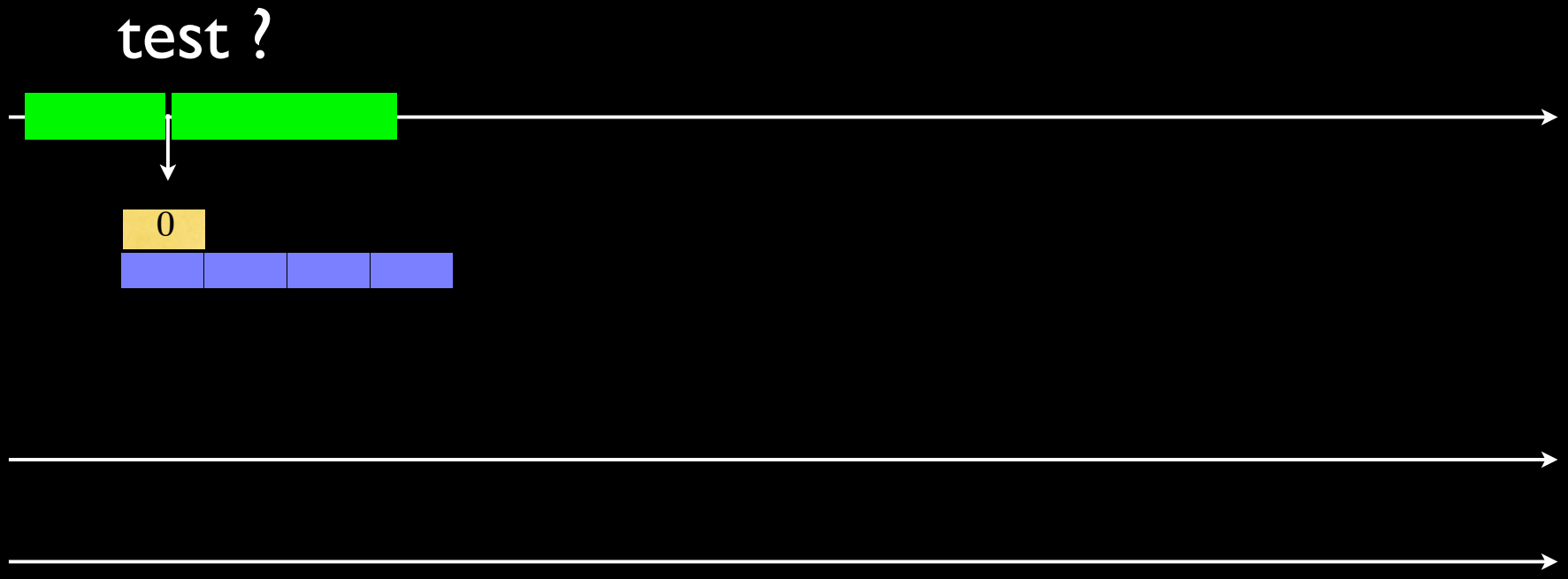
- [X. Besseron, C. Laferrière]
- Keep same semantics as usual
  - a task is extracted exactly once
- so...avoid concurrency between victim & thieves
  - the victim interrupts its work to process steal requests
  - some similarity with TasCell [09], Capsule [06]
- Drawback
  - the victim should poll requests
  - thieves are waiting

Cooperative WS

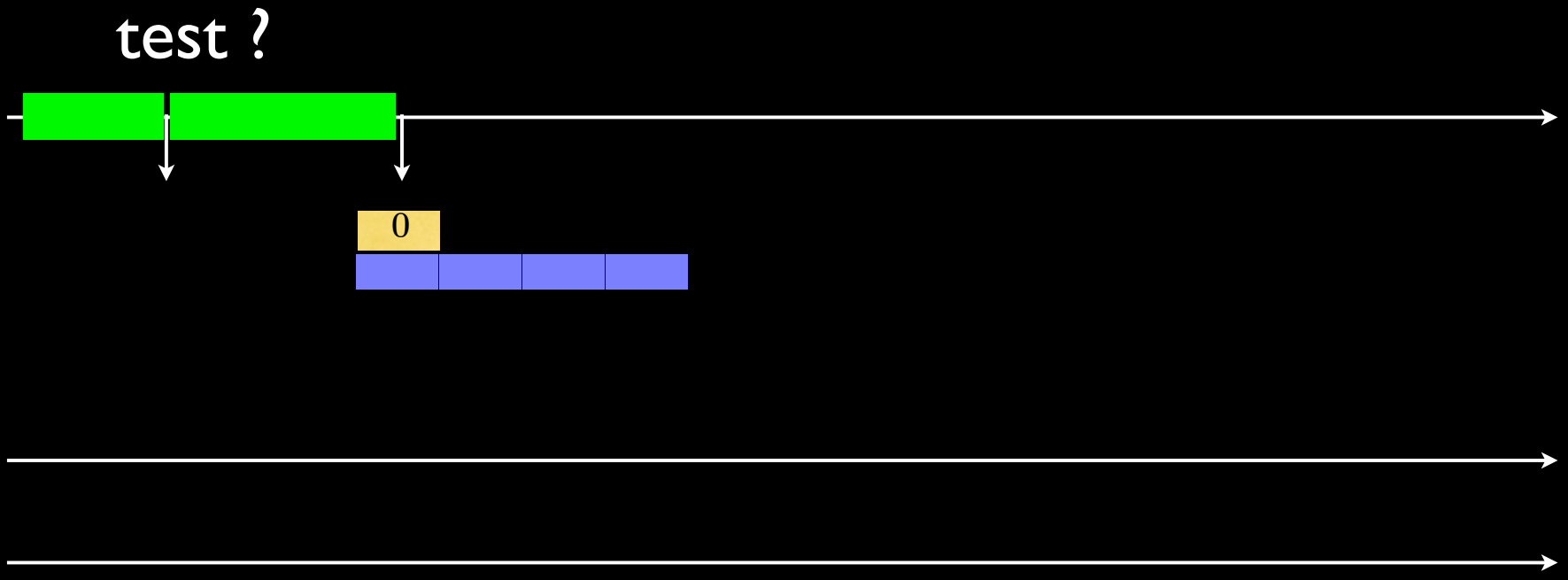
Cooperative WS



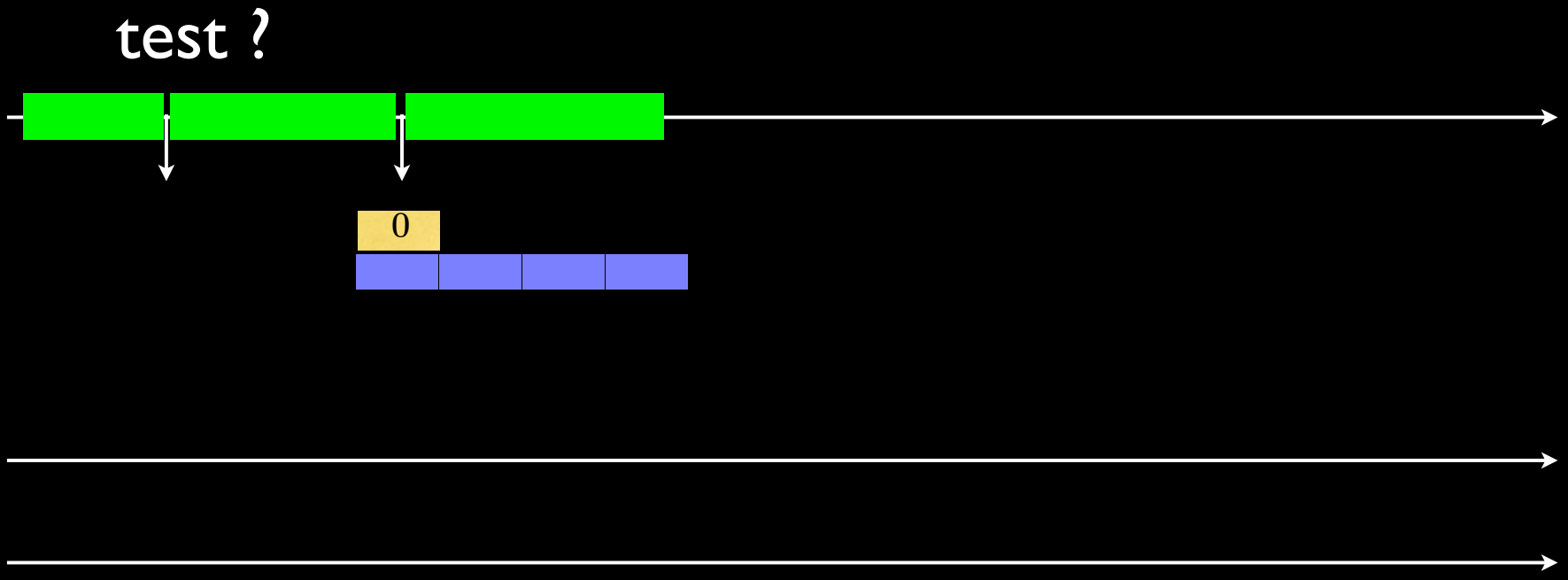
Cooperative WS



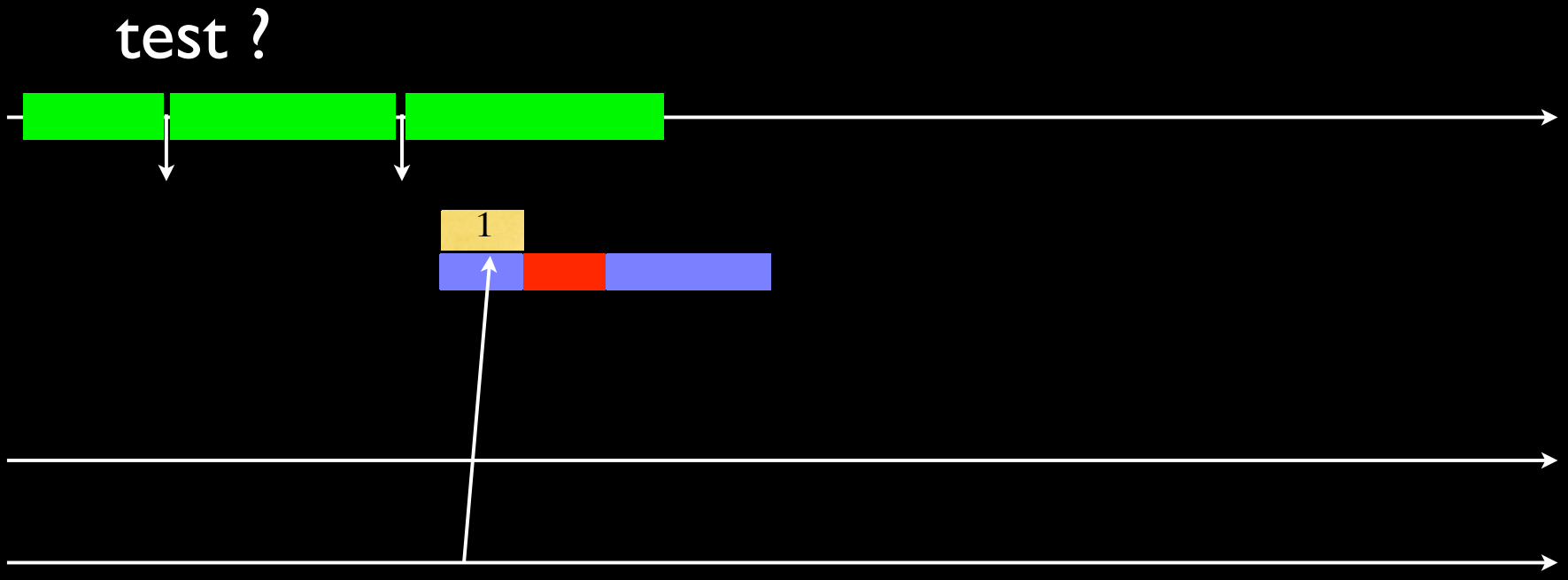
Cooperative WS



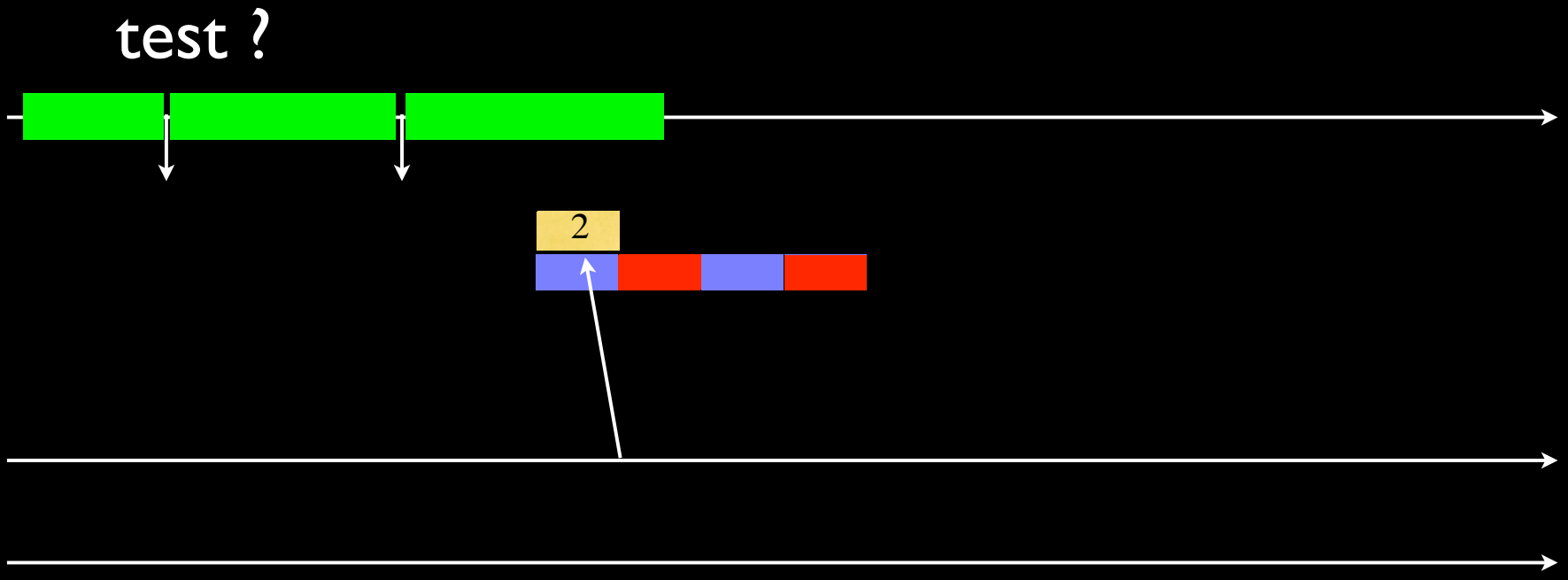
Cooperative WS



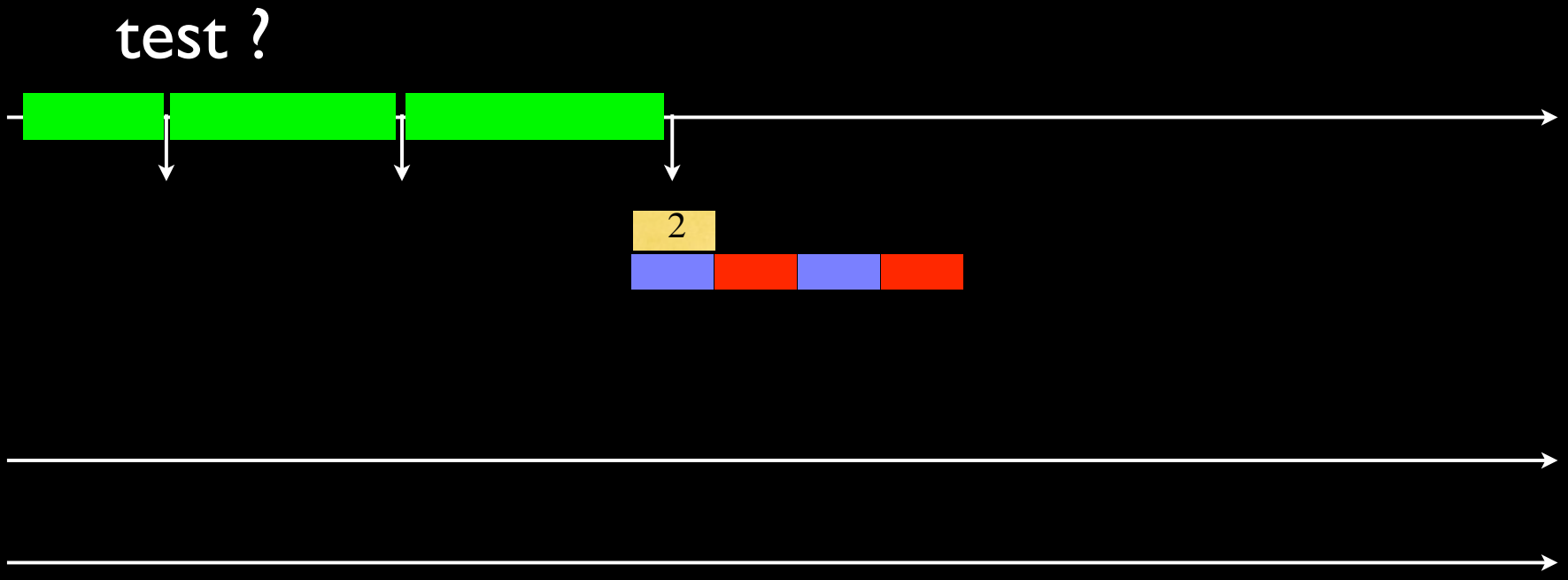
Cooperative WS



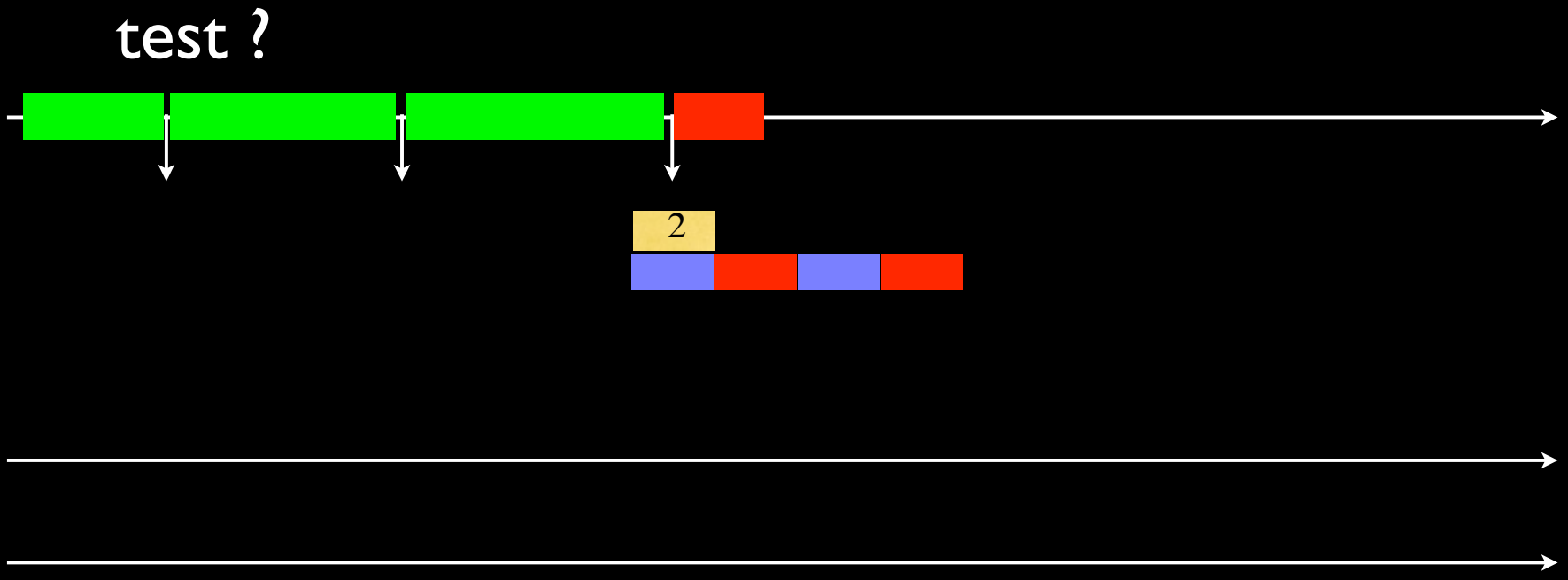
Cooperative WS



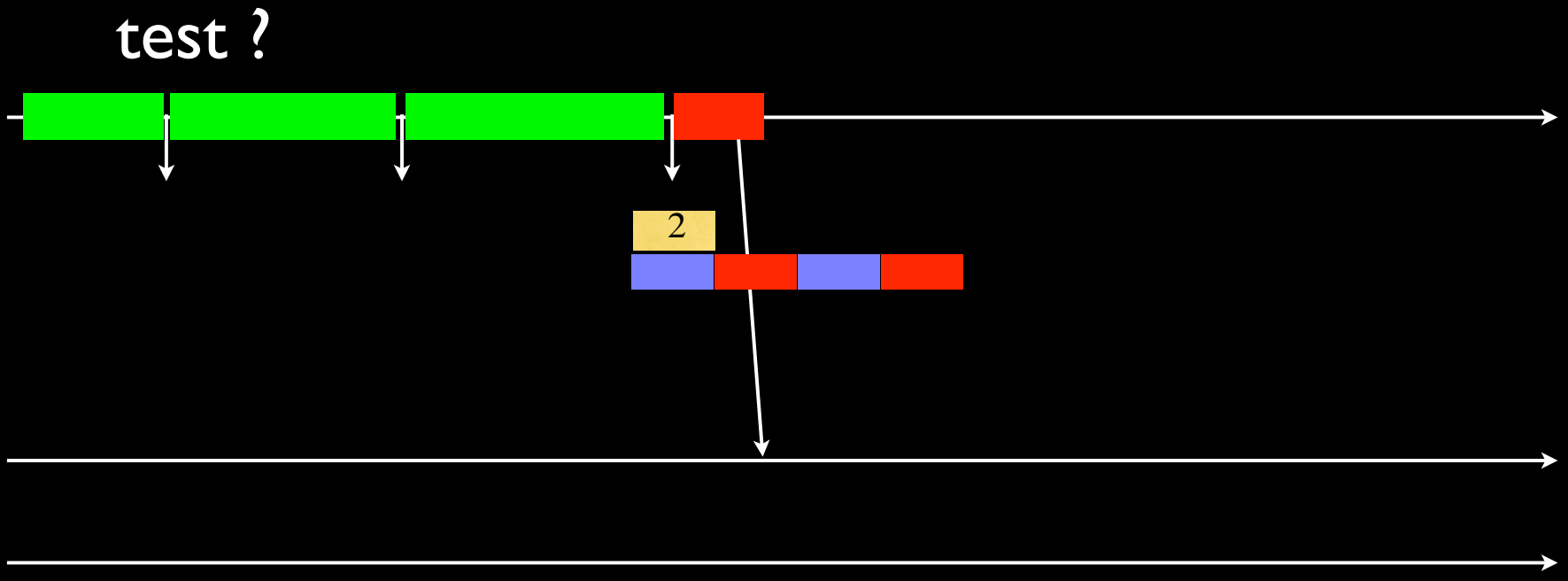
Cooperative WS



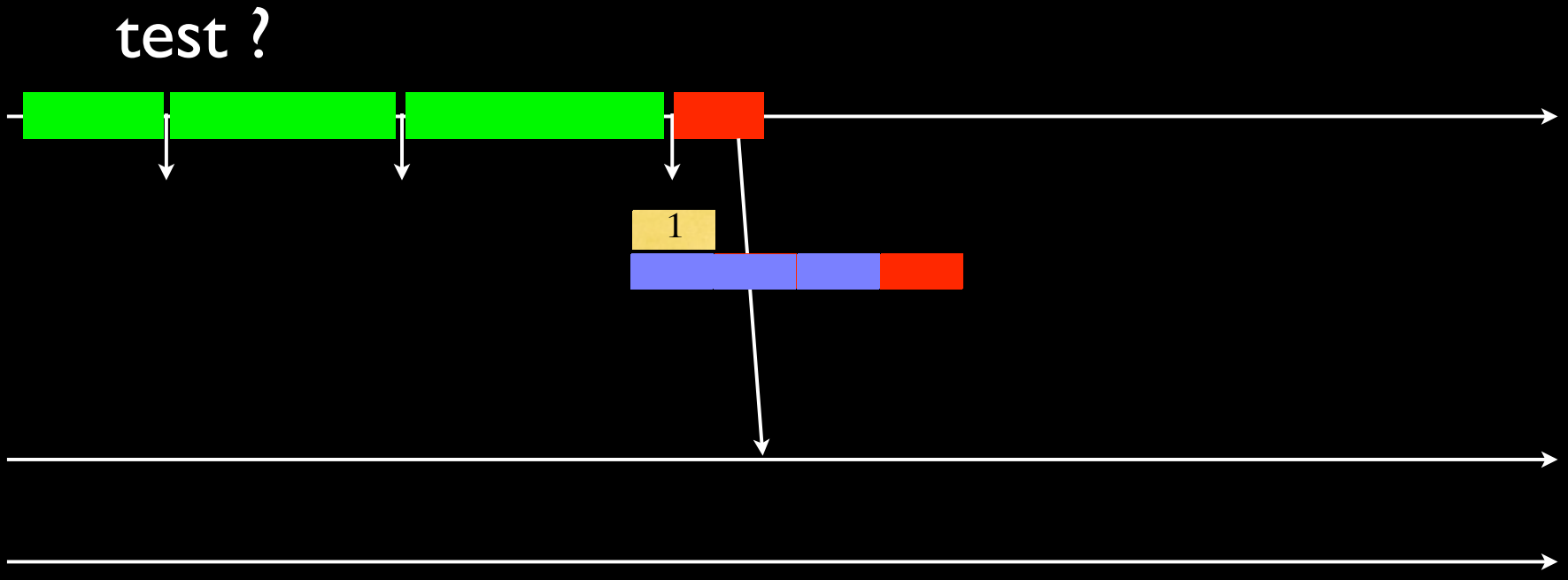
Cooperative WS



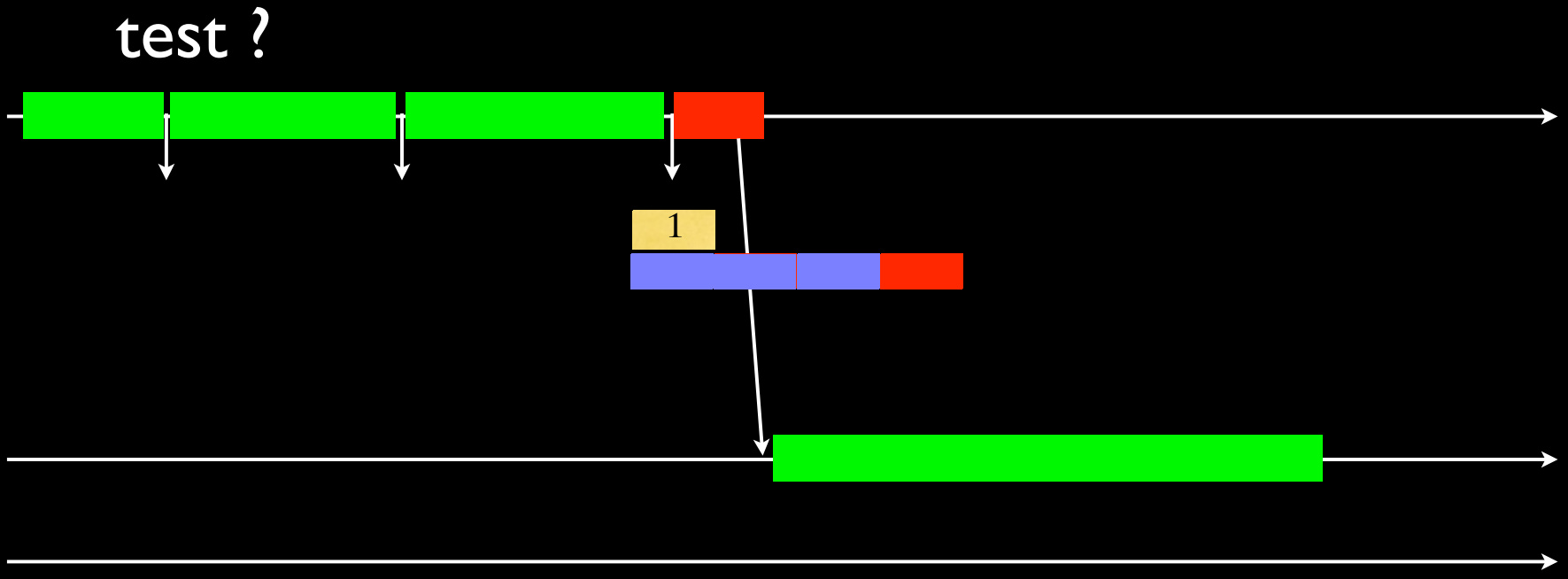
Cooperative WS



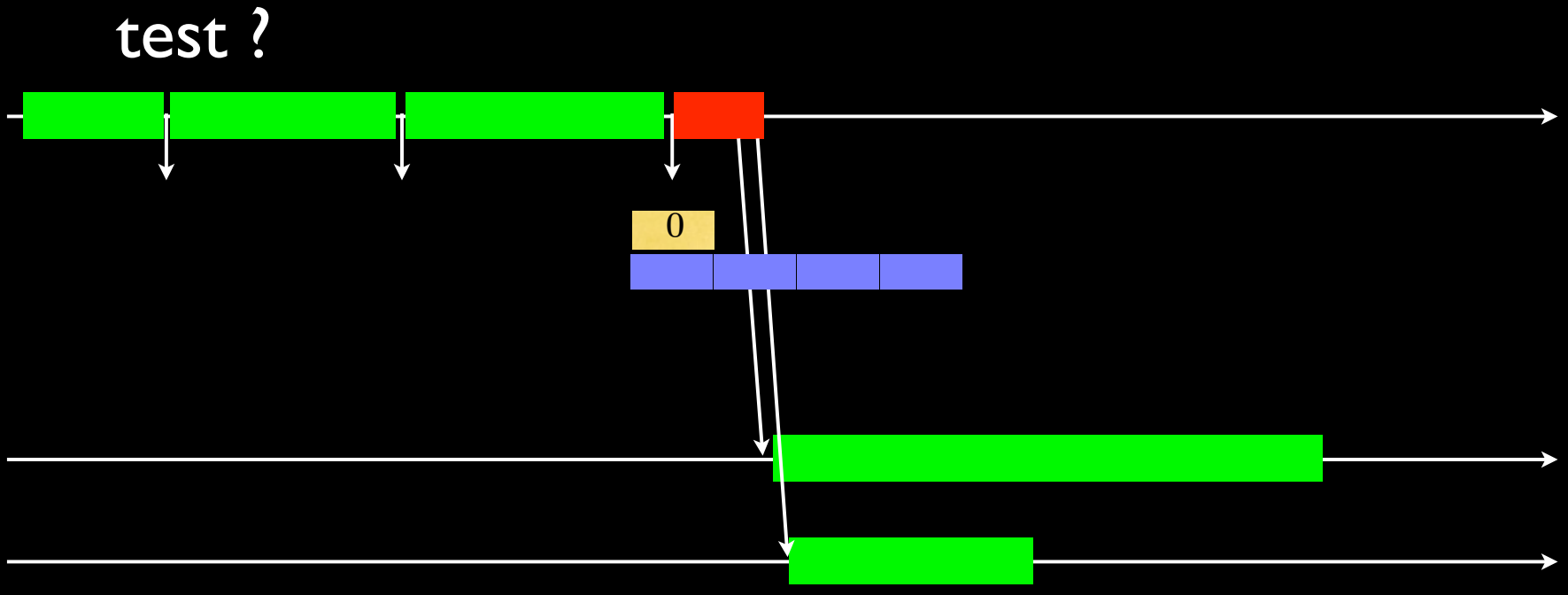
Cooperative WS



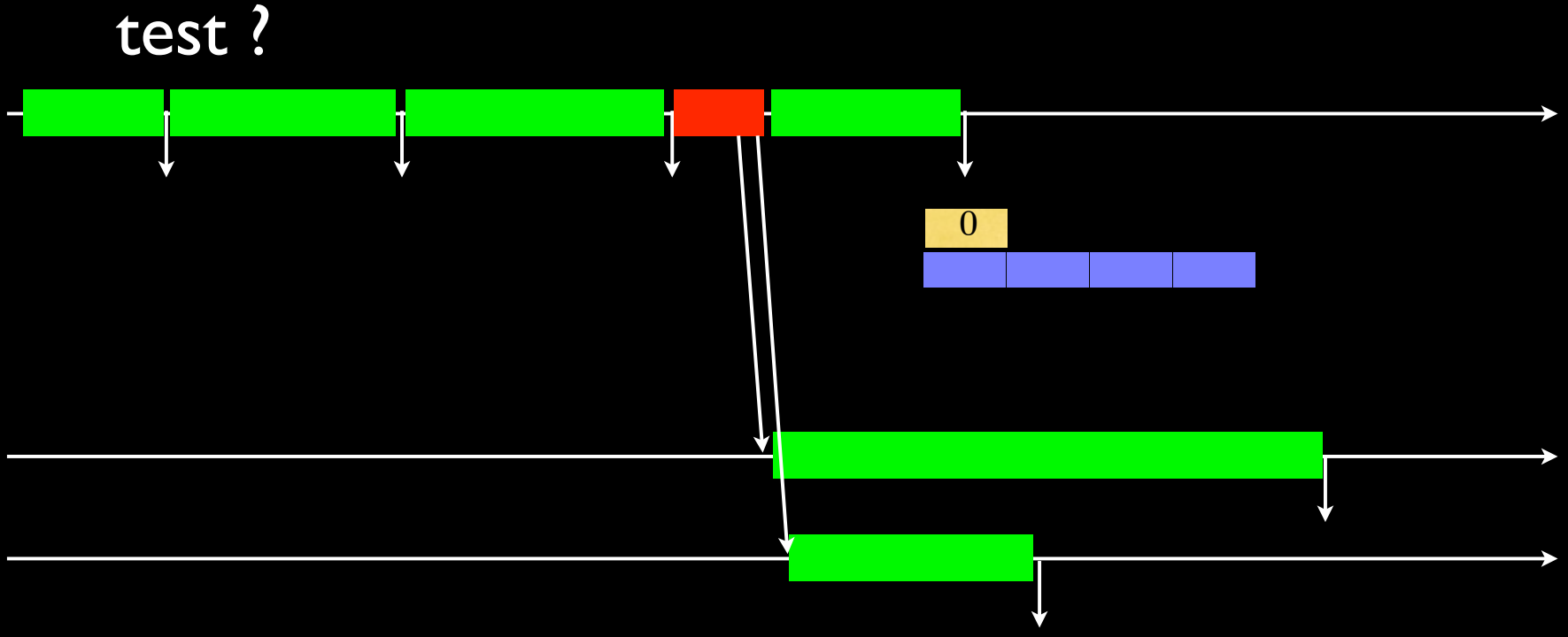
Cooperative WS



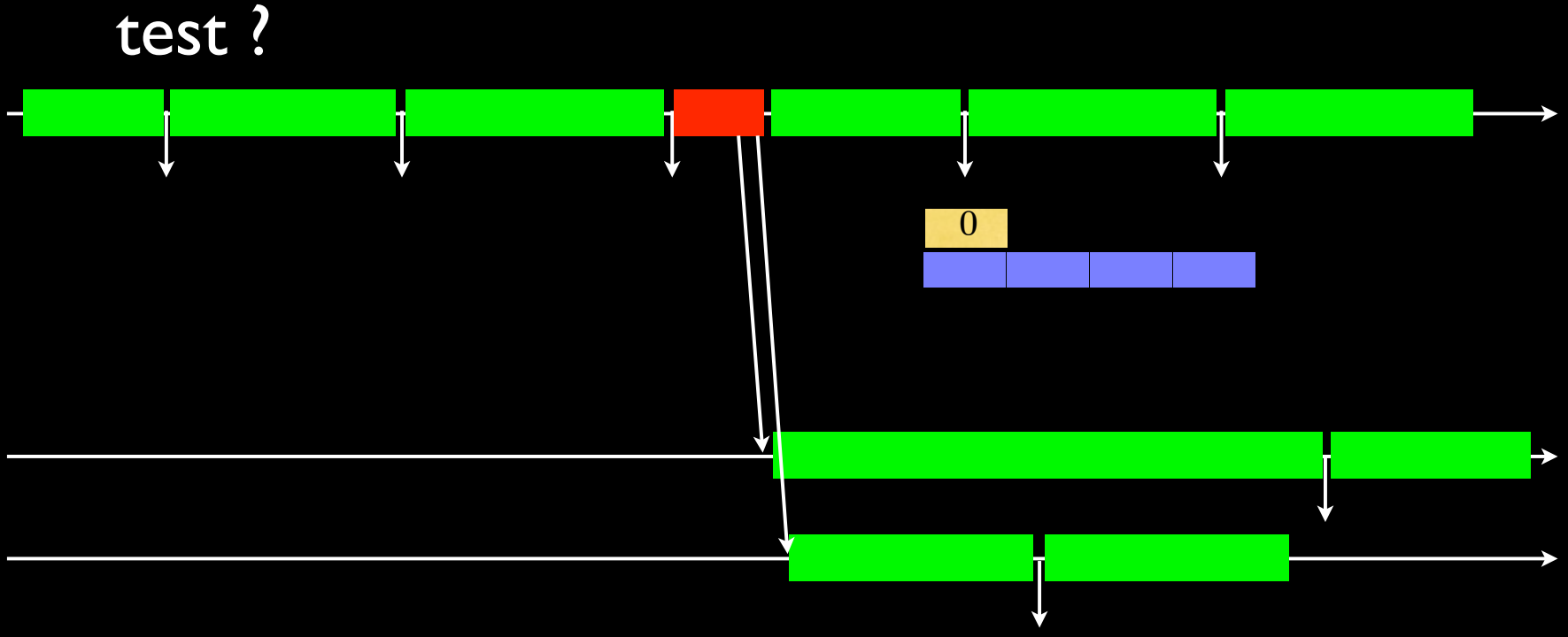
Cooperative WS



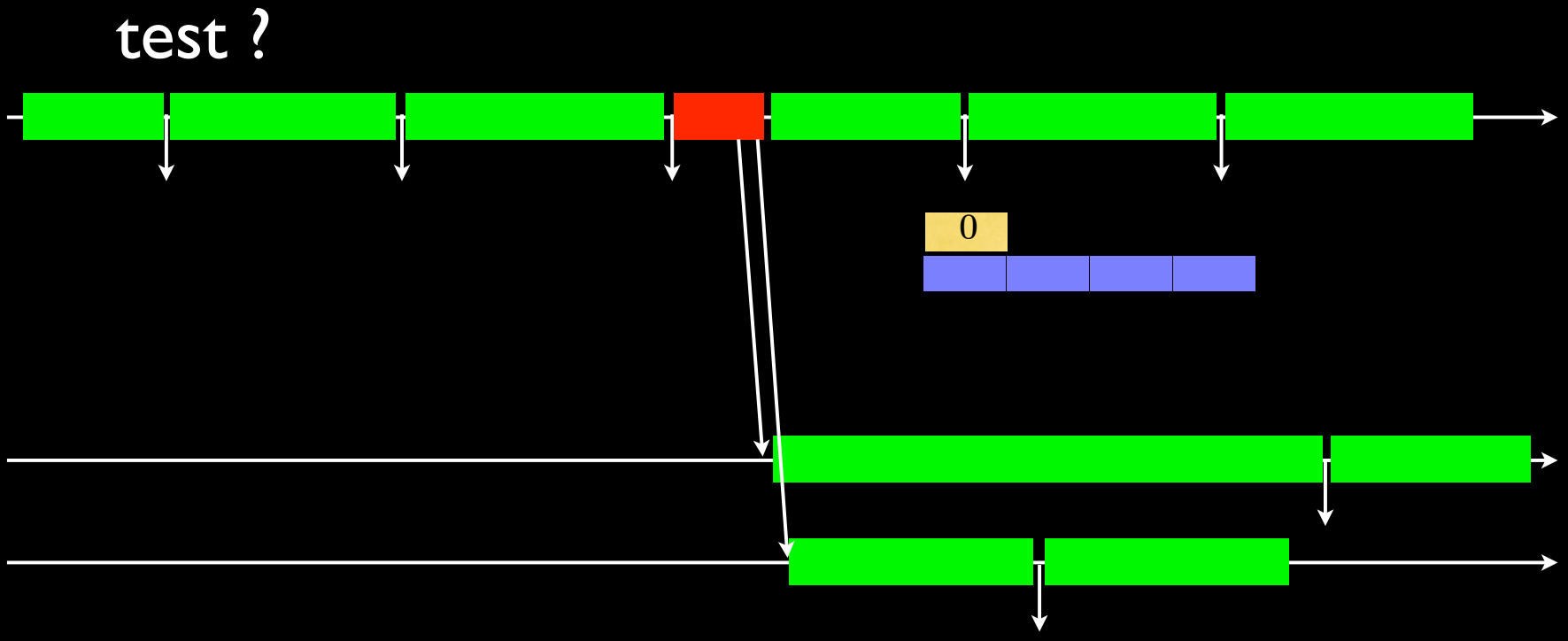
Cooperative WS



Cooperative WS



Cooperative WS



- Gain
  - work overhead: 1 read on memory
  - reply to K thieves in place of only 1
    - better workload balance

Outline

- Athapascan / Kaapi a software stack
- Foundation of work stealing
- Controls of the overheads
- Experiments with STL algorithms

KaSTL: Parallel STL

- [PhD Daouda Traoré, 2008]
- Technics applied to 95% of STL algorithms
  - STL: Standard C++ Template Library
- Comparison with other library
  - Cilk++, Intel Thread Building Block (TBB)
  - MCSTL (GNU STL parallelized with OpenMP)
- Multiprocessor : 8 AMD CPUs with 2 cores

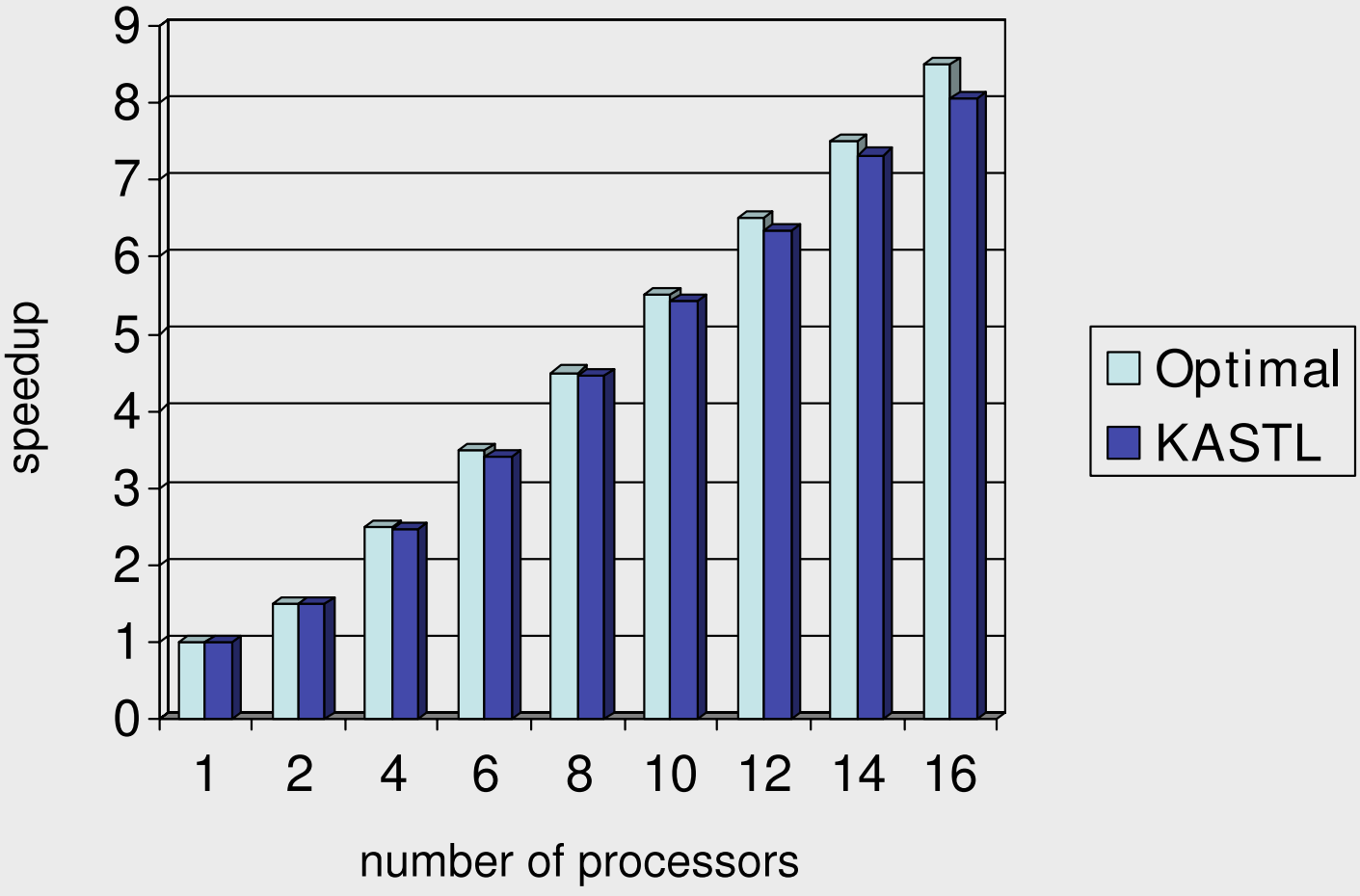
Experiments

- Two set of experiments
  - 1/ with adaptive algorithms
    - PhD of [Daouda Traoré]
  - 2/ with cooperative work stealing
    - [Daouda Traoré, Xavier Besseron, Christophe Laferrière]

- Methodology
  - average over 30 runs
  - 1 run = average of 100 basic experiments, do not take into account the first experiment

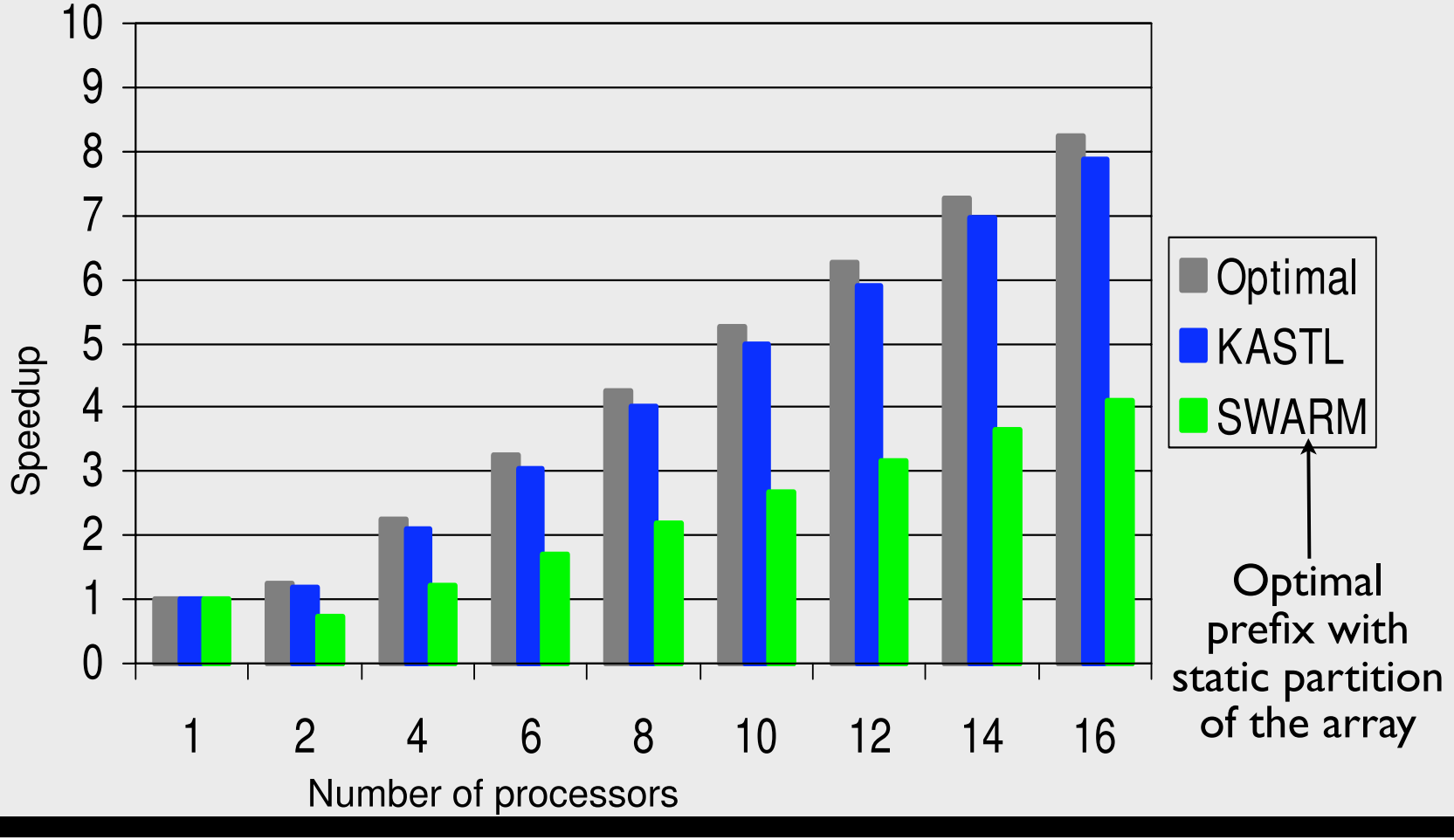
Prefix

- Homogeneous processors
  - * coarse grain, 30000 elements



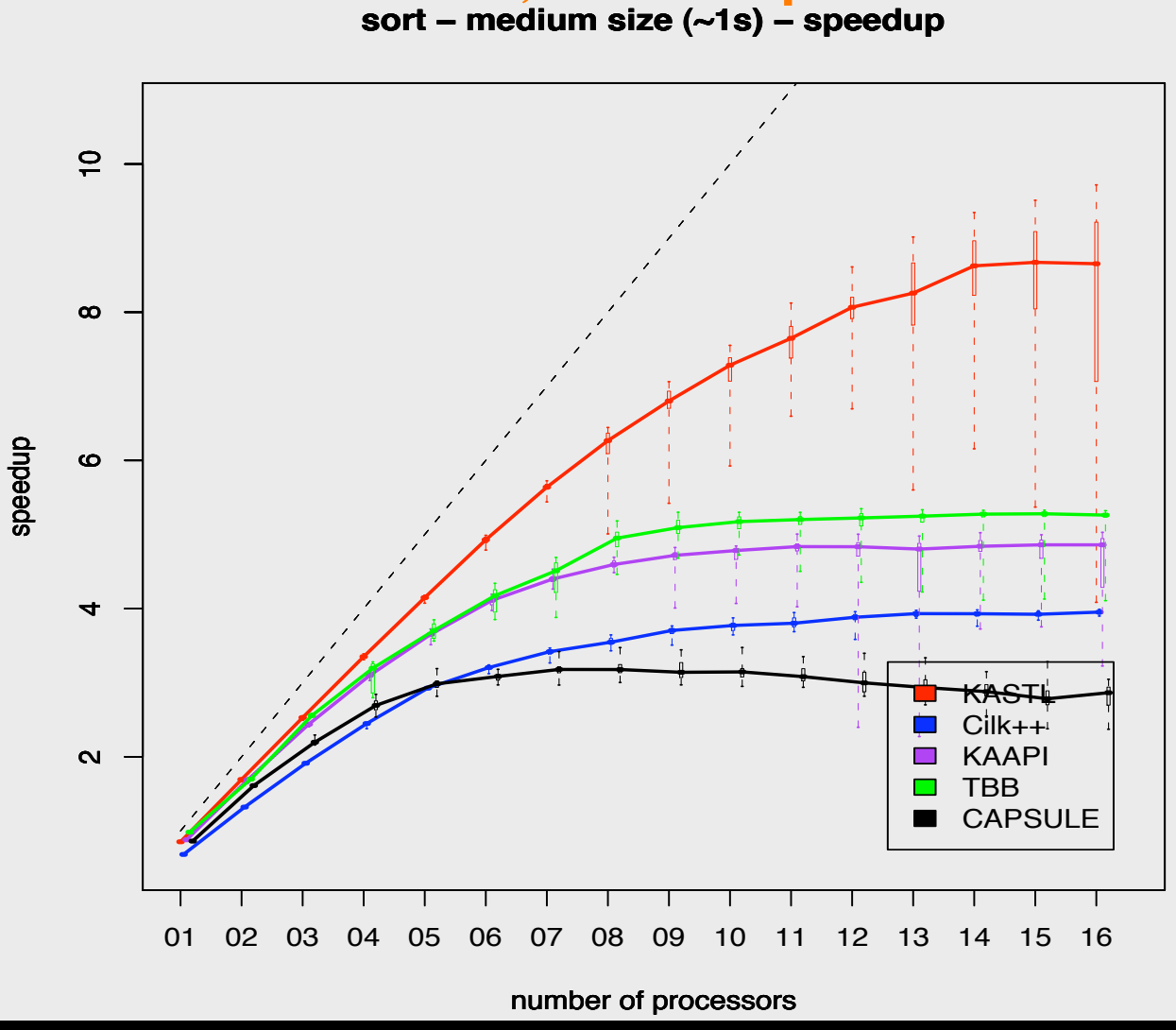
Prefix

- Heterogenous
  - (p−1) processors have the same speed S
  - 1 processor has speed S/2



Sort

- ~ 100M elements, 1s sequential time



Partial conclusion

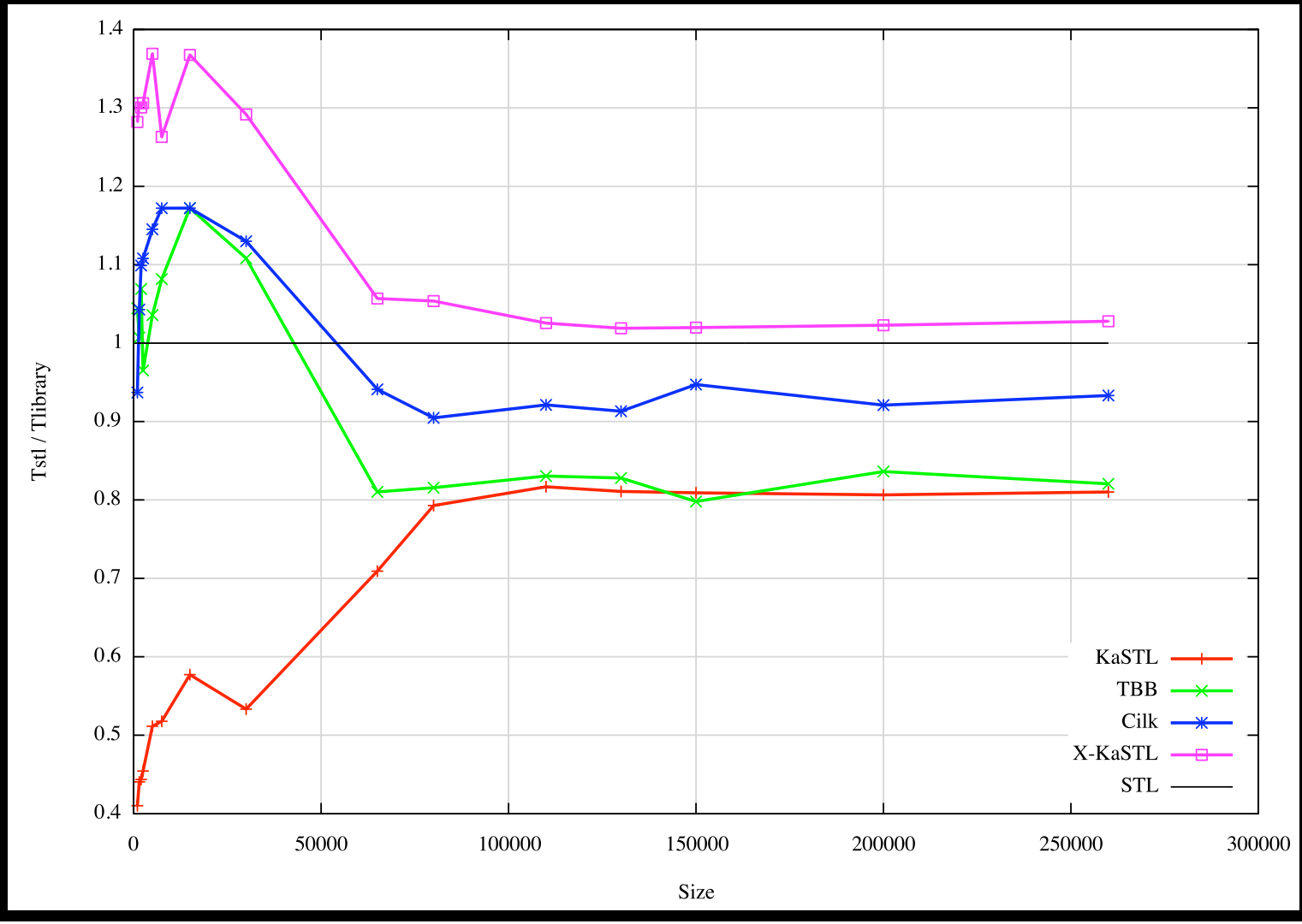
- Interest to adapt the task to activity (or idleness) of processors
- But coarse computation
  - sequential time > 0.1s

➡ + Cooperative work stealing

  - named X-KaSTL or CKaapi in diagrams

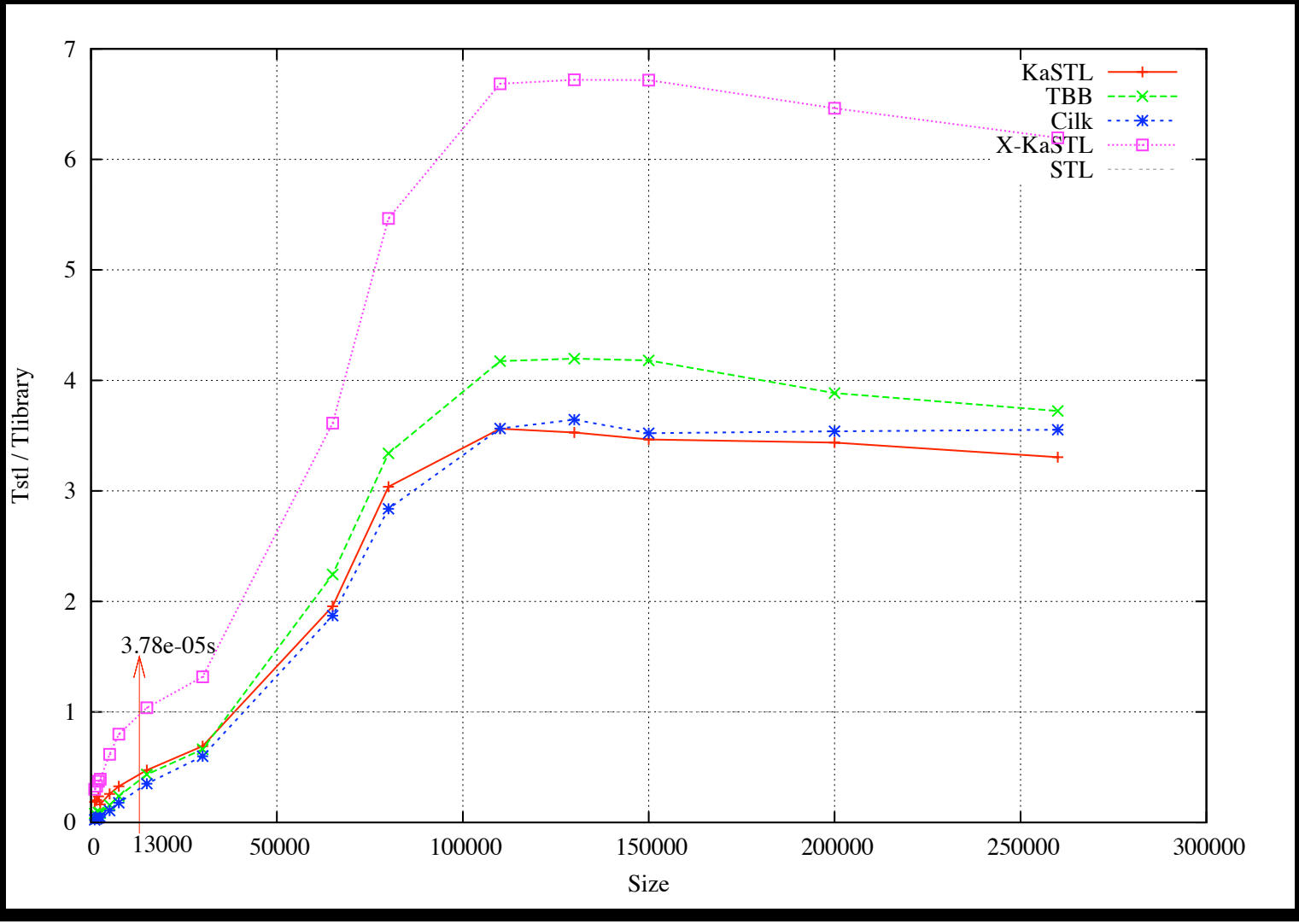
std::transform

- 1 processor



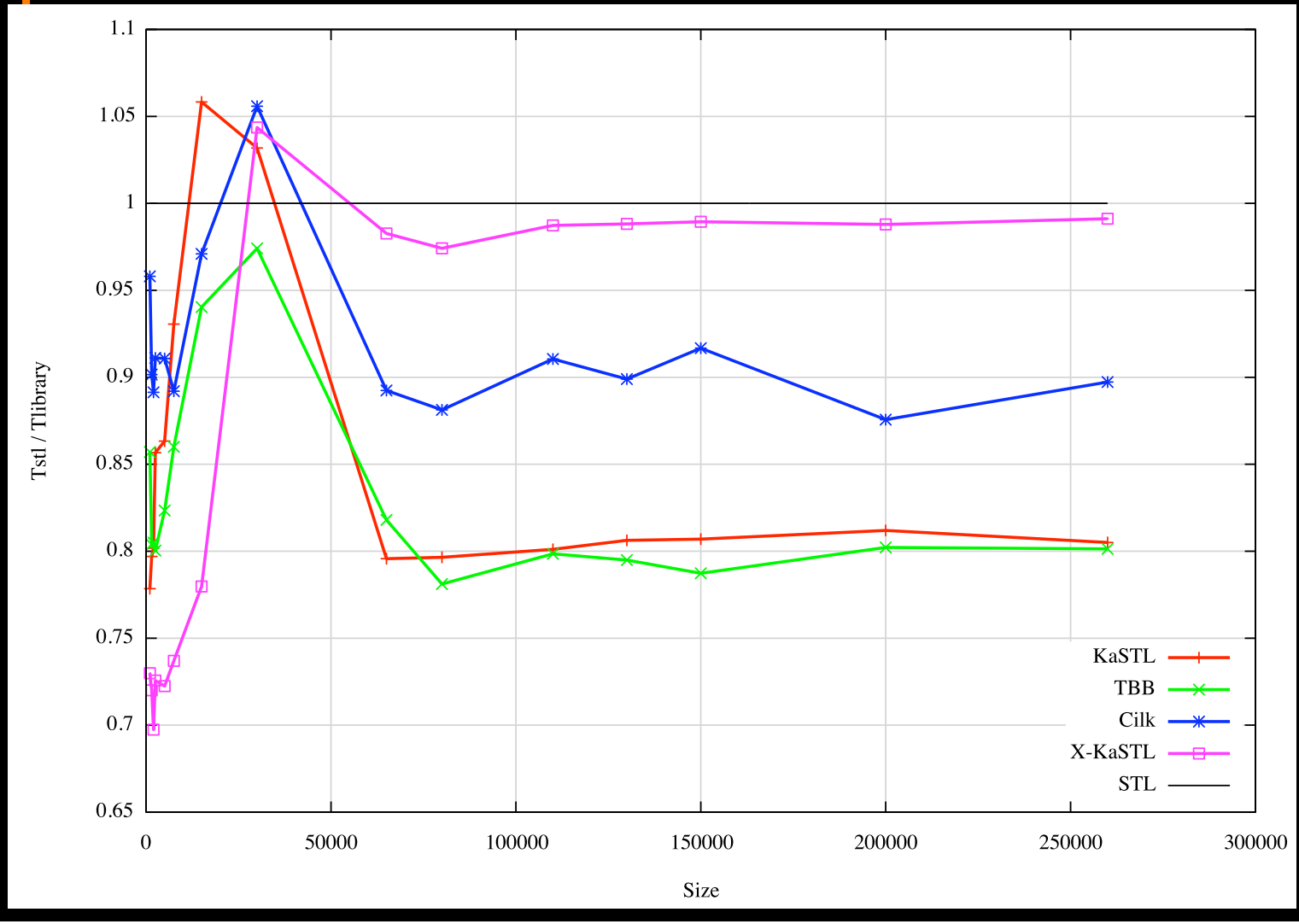
std::transform

- 8 processors NUMA machine



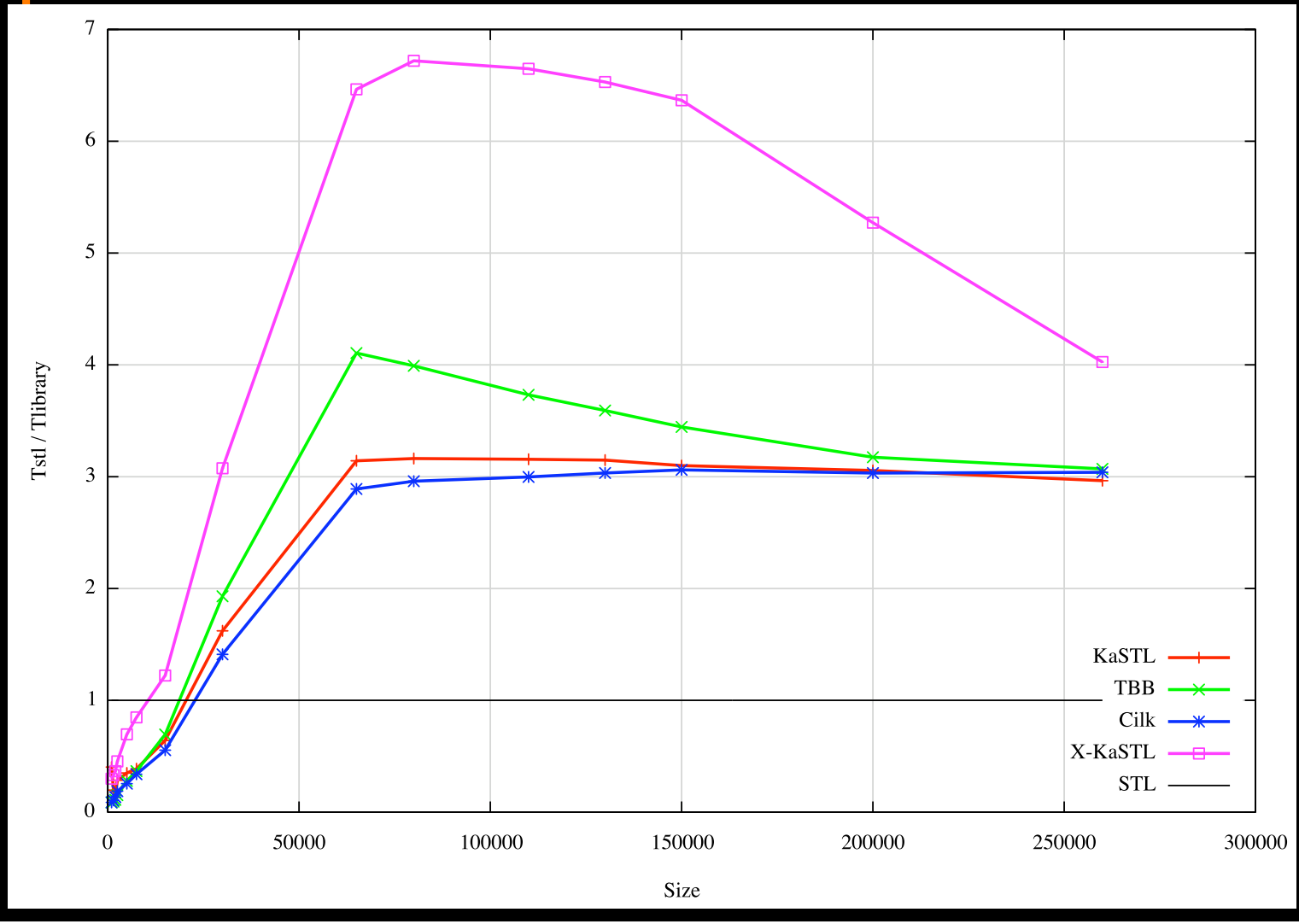
std::merge

- 1 processor



std::merge

- 8 processors



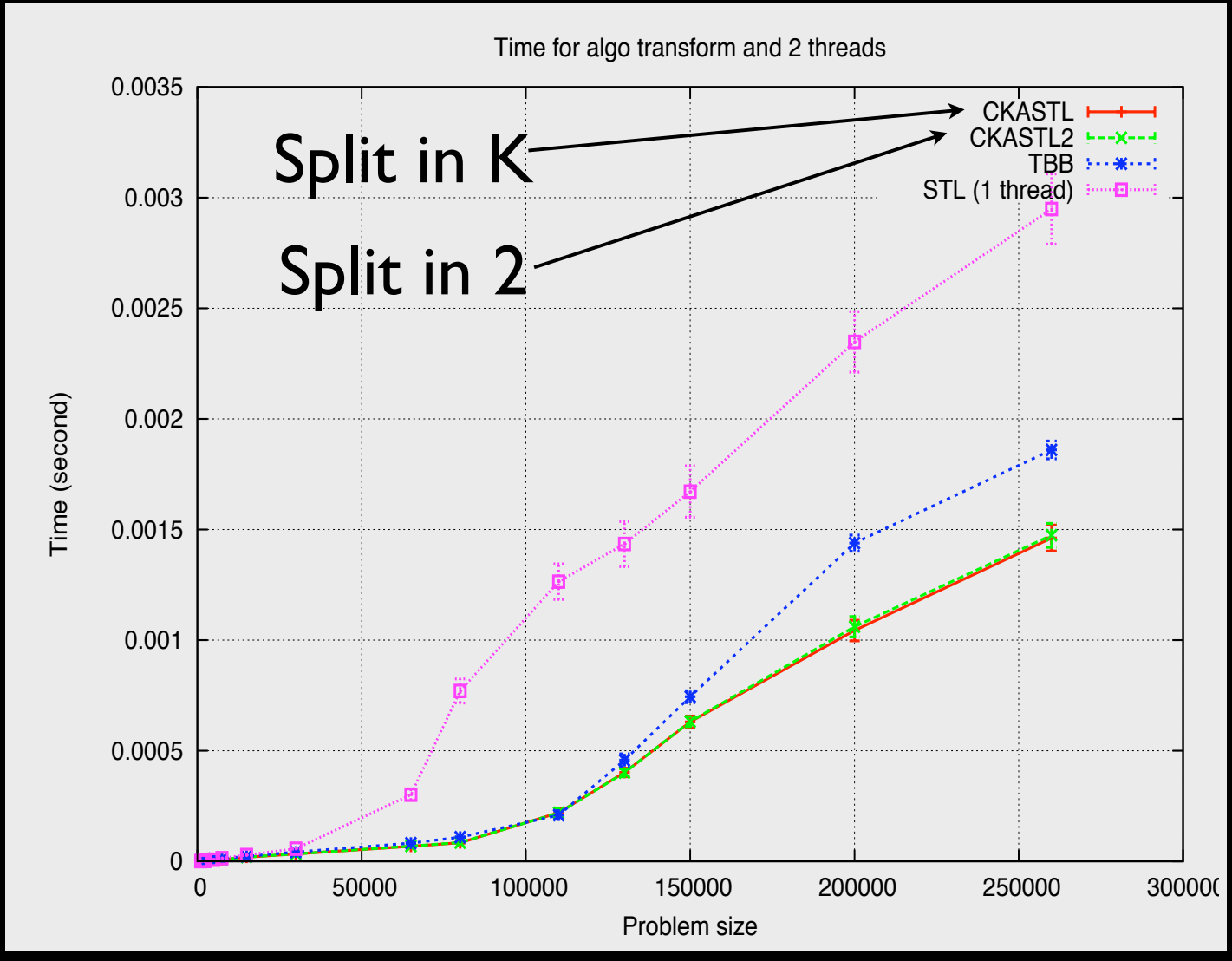
Partial conclusion

- Effective speedup at finer computation
  - sequential time ~ 10^{-3}s = 1ms

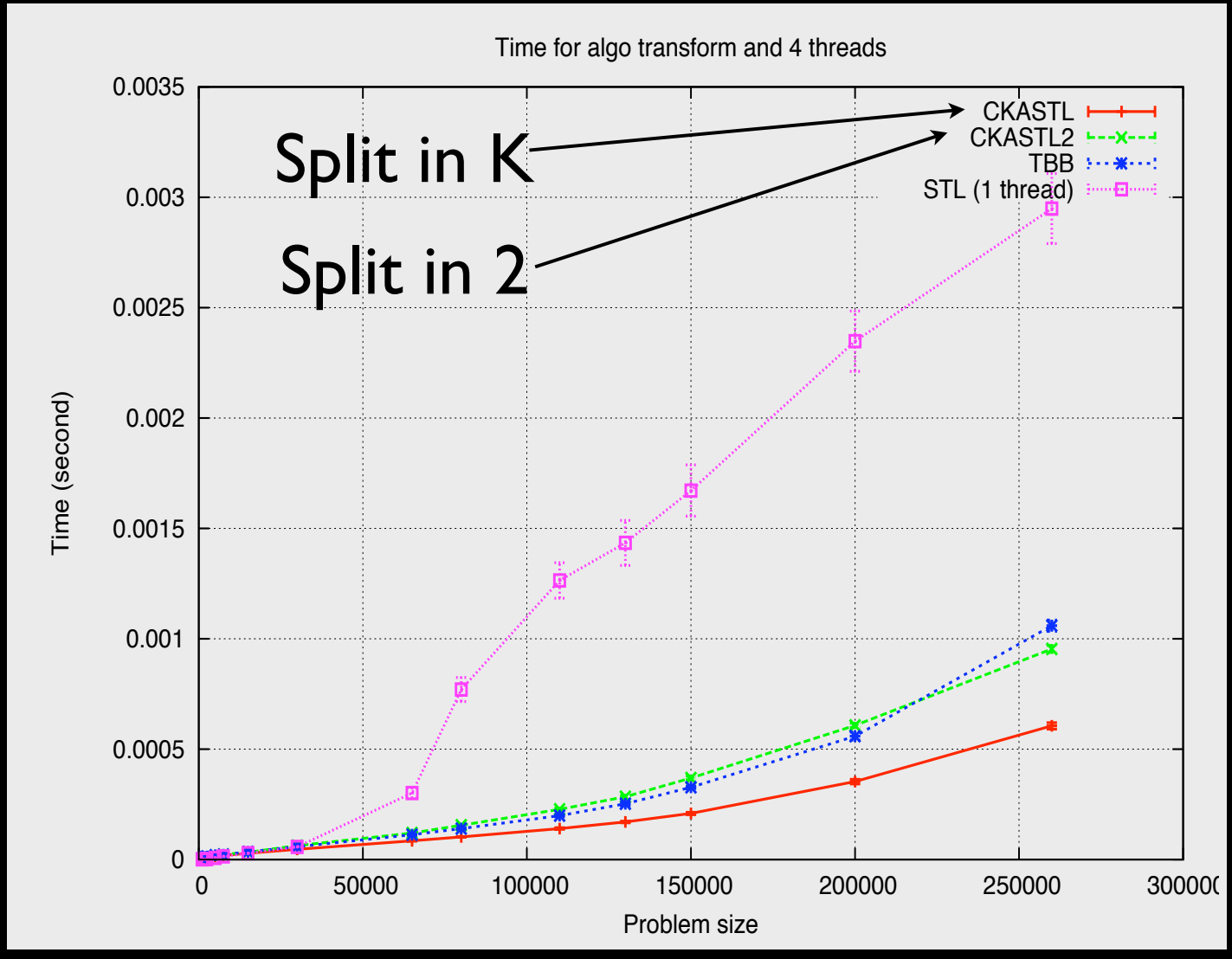
➡ impact of the better balance of work load

  - comparison split in K versus split in 2

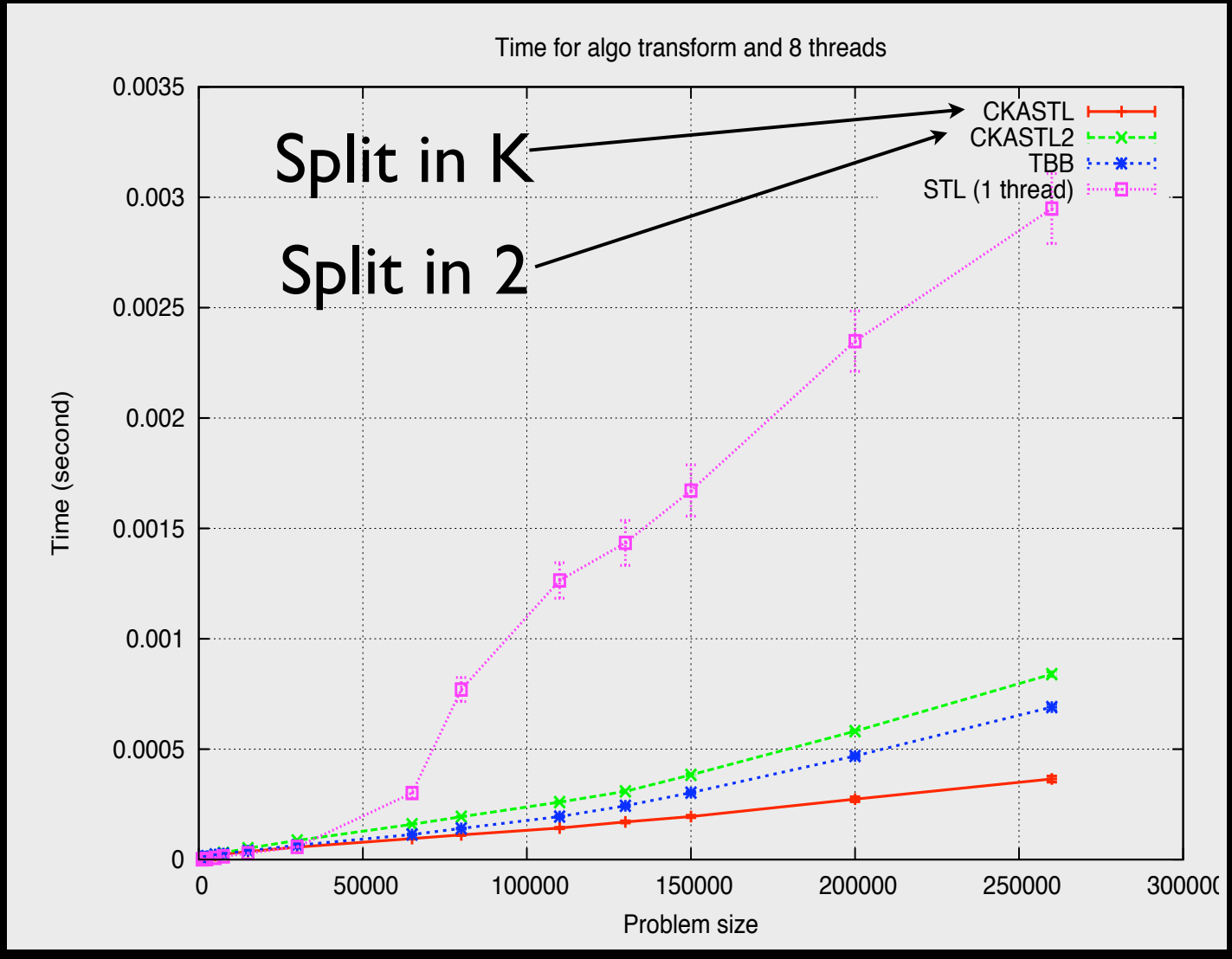
“Transform” 2 threads



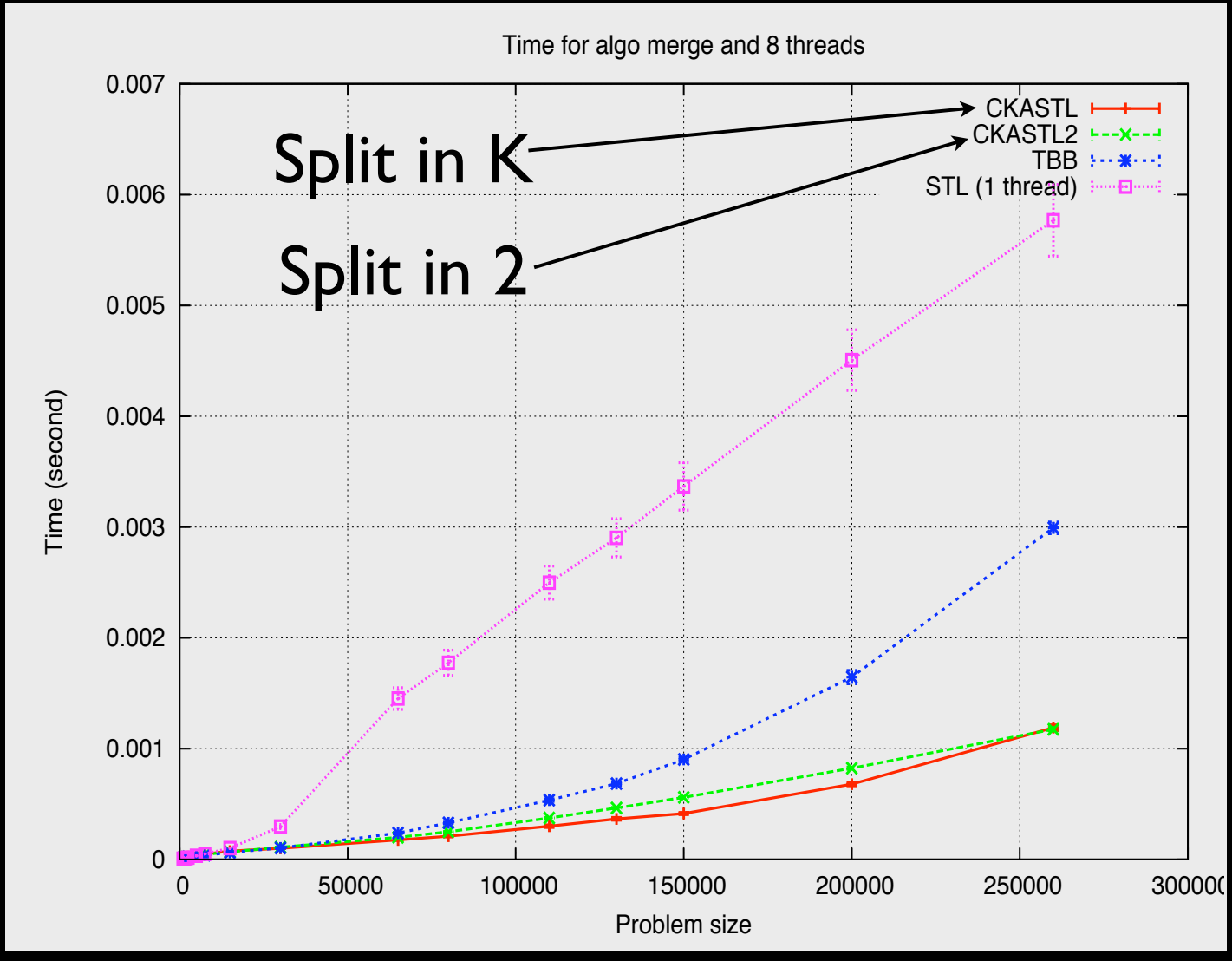
“Transform” 4 threads



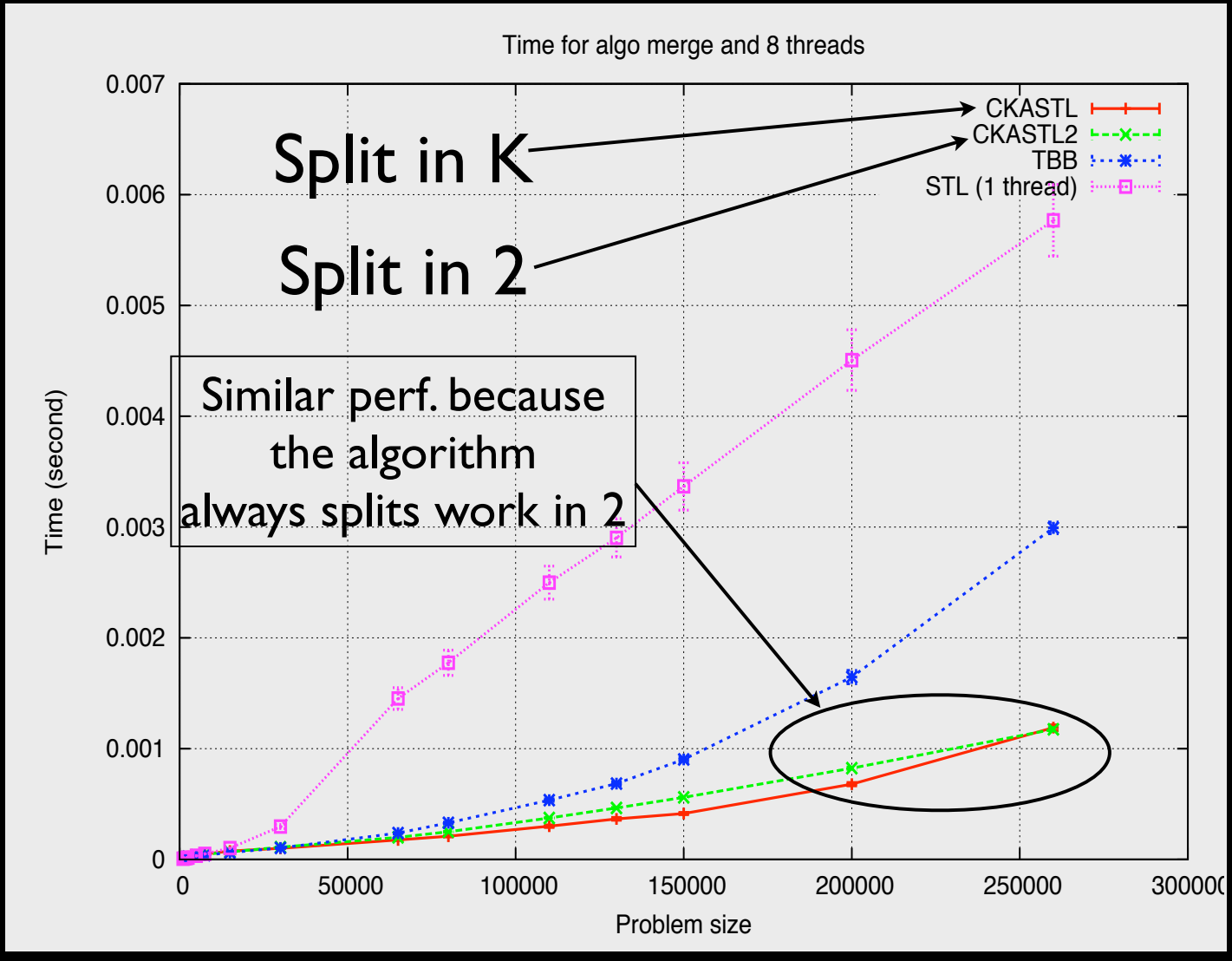
“Transform” 8 threads



“Merge”, 8 threads



“Merge”, 8 threads



Conclusion

- Athapascan/Kaapi
  - a data flow model with lazy task generation
    - reduce the work overhead
  - sequential semantics
  - original approach
  - effective parallelization of fine computation
    - with cooperative workstealing
- http://kaapi.gforge.inria.fr/

- Drawback
  - non standard research software
    - difficulty to port already parallelized applications

Perspectives

- Software being to be more robust
  - INRIA action to support development (2 years)
  - Ported on top of most Unix (Linux, MacOSX, [SunOS], iPhoneOS]
  - [2010] will be ported on IBM/BlueGene
  - [2010] will be ported on MPSoC with ST Microelectronics
- Viability of the cooperative approach
  - not yet theoretical fundation
  - coupling technics with concurrent work stealing

Perspectives

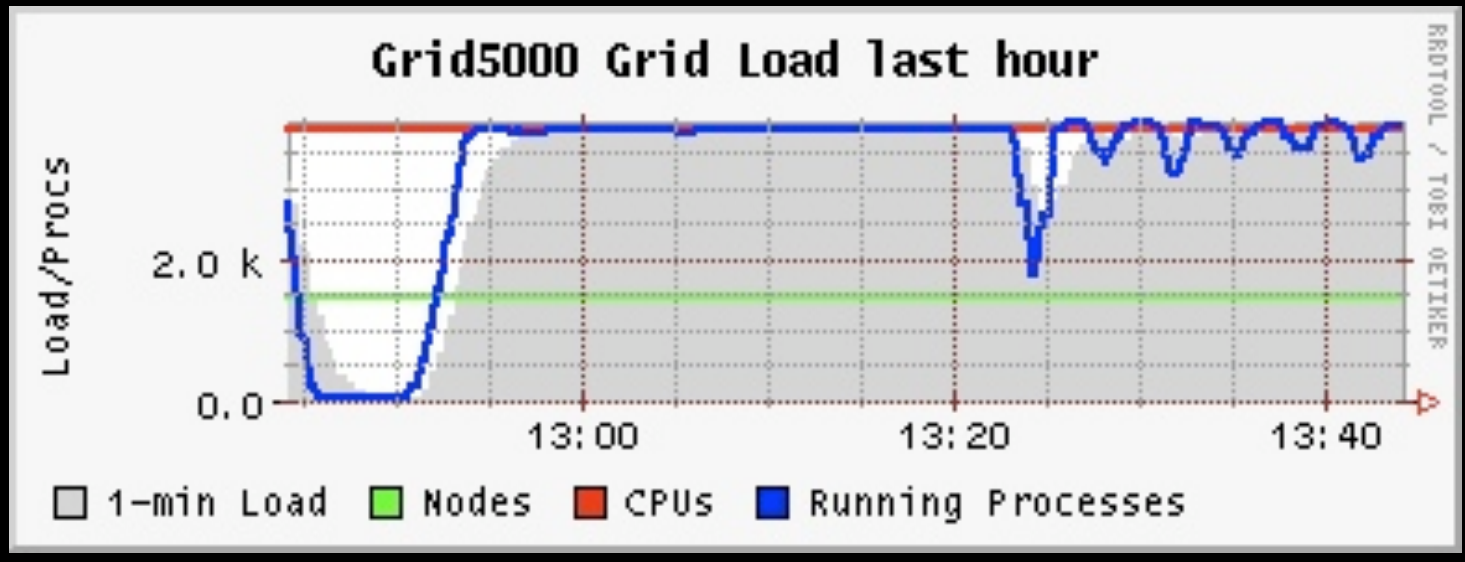
- Mixing CPUs & GPUs
  - preliminary work
  - deeper integration of the GPU as a processing resource
    - next Fermi GPU + driver ?
- Better coupling between OS & Middleware
  - importance to know (in advance) the available number of resources
- Taking into account NUMA architecture
  - ongoing work at MOAIS [JN Quintin, PhD]

Other applications

- [Parallelization of Bayesian computation]
- Parallel Computer Algebra
  - Linbox: http://www.linalg.org
- Combinatorial Opt. [PRiSM (Paris), B. Lecun]
- 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)

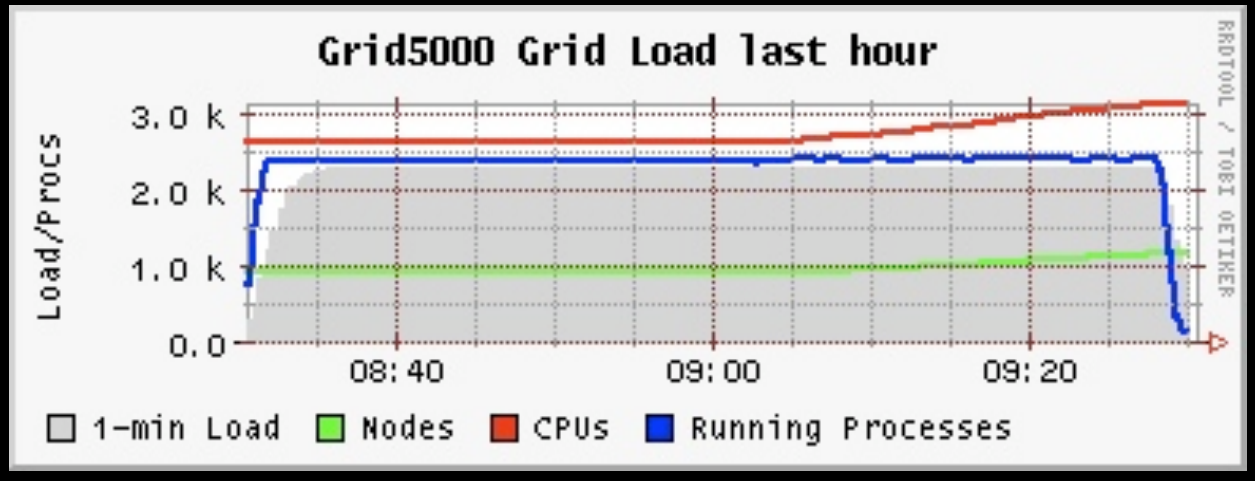

KAAPI

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

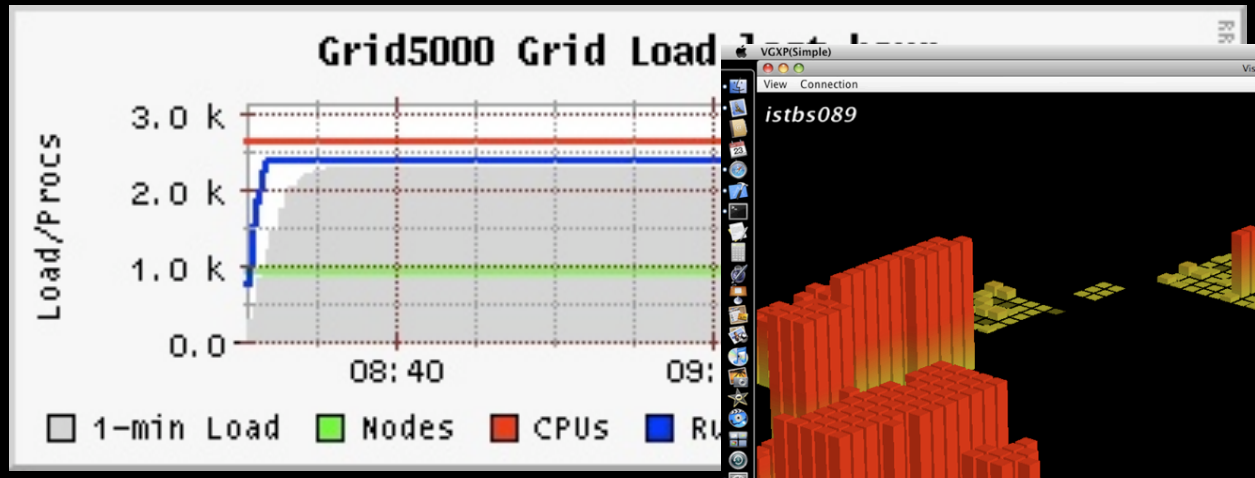
Monte Carlo / Option Pricing



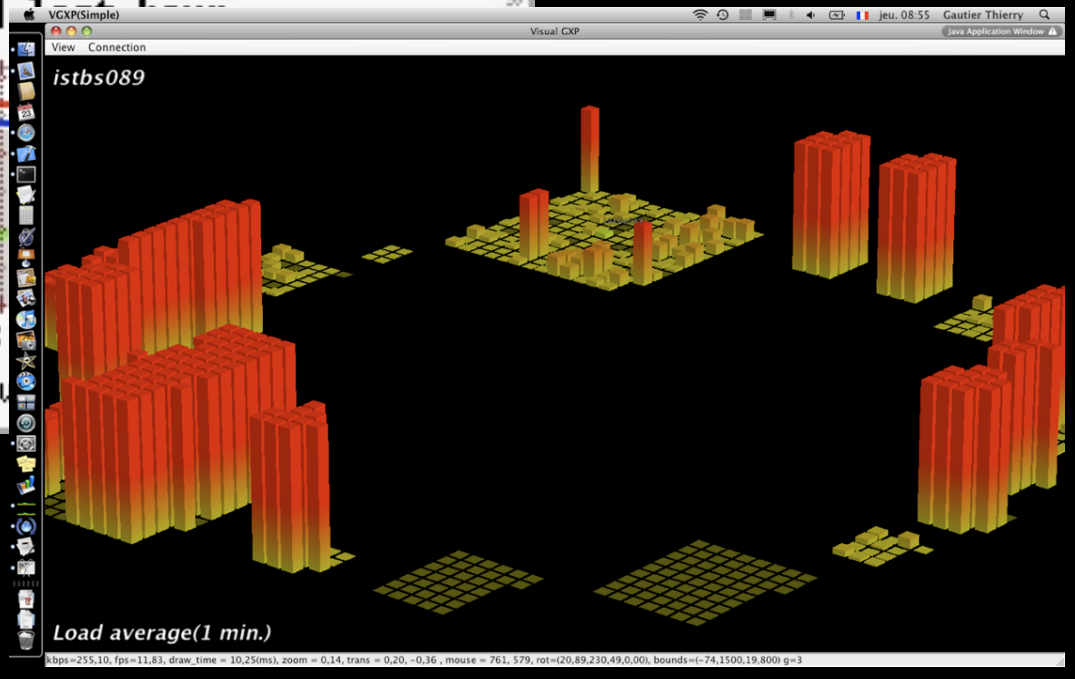
Grid5000: France

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

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:

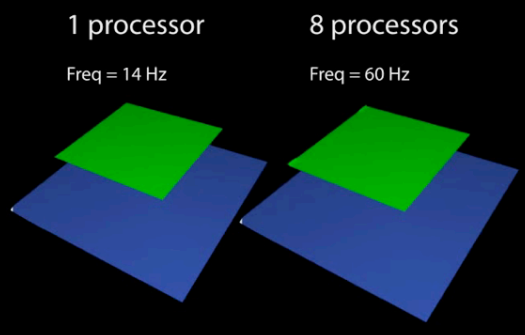
  Surgery simulation

Physics Simulation

- SOFA: real-time physics engine
- Strongly supported INRIA initiative
- Open Source:

  http://www.sofa-framework.org
- Target application:

  Surgery simulation

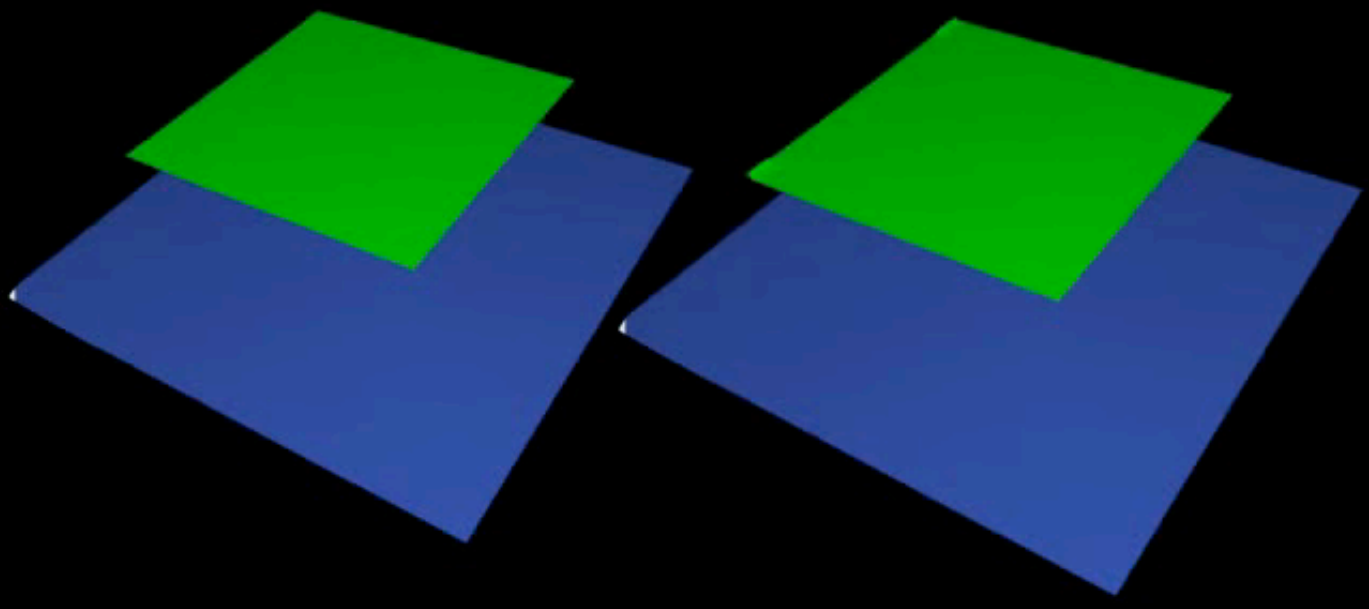


Physics Simulation

- SO
- Str
- Op

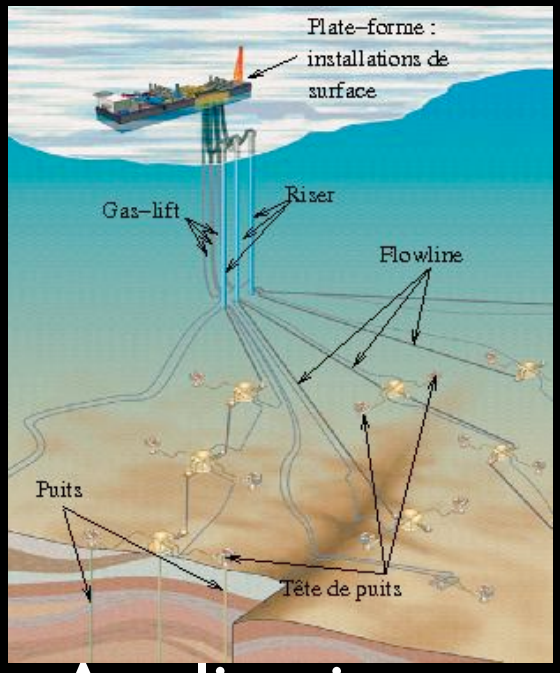
  htt
- Tar

  Sul

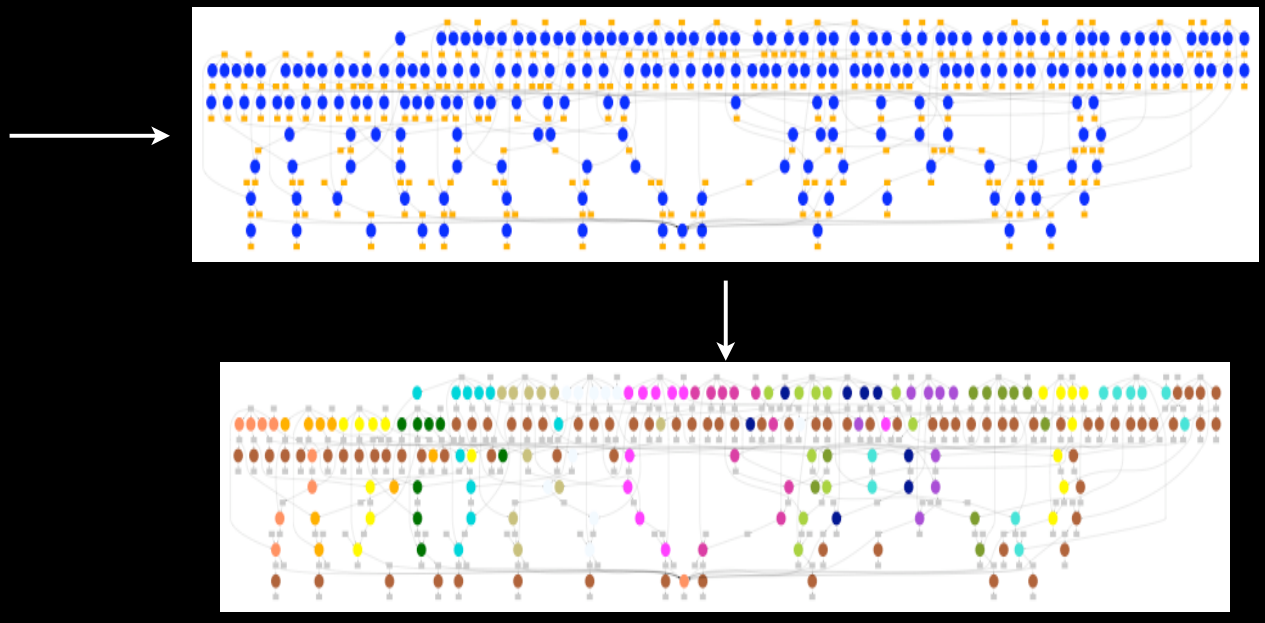


Iterative Application

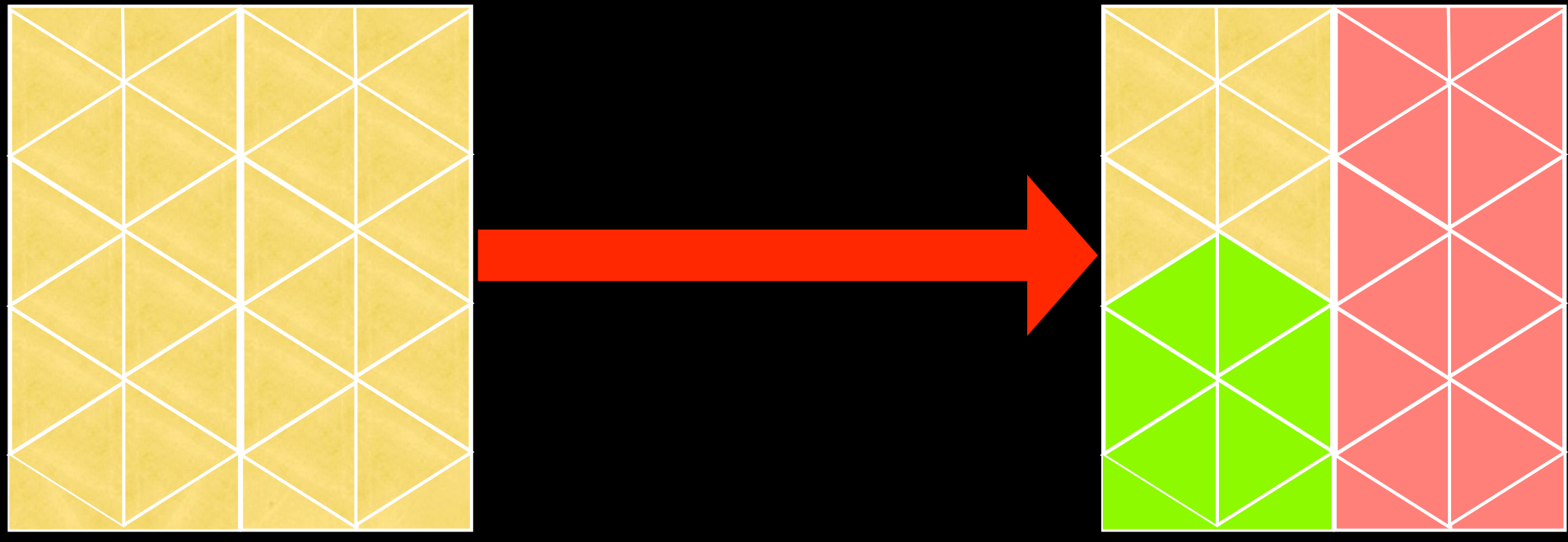
- Scheduling by graph partitioning
  - Metis / Scotch

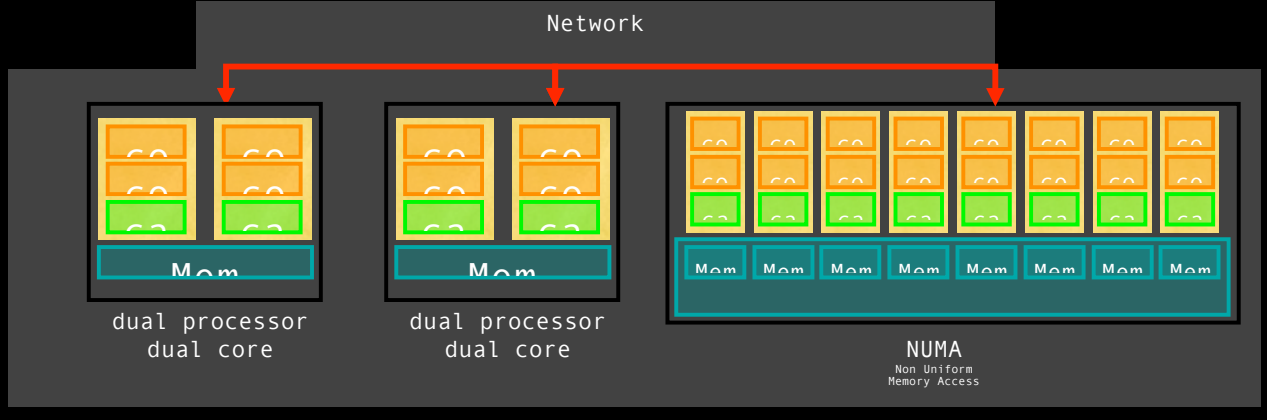


Application

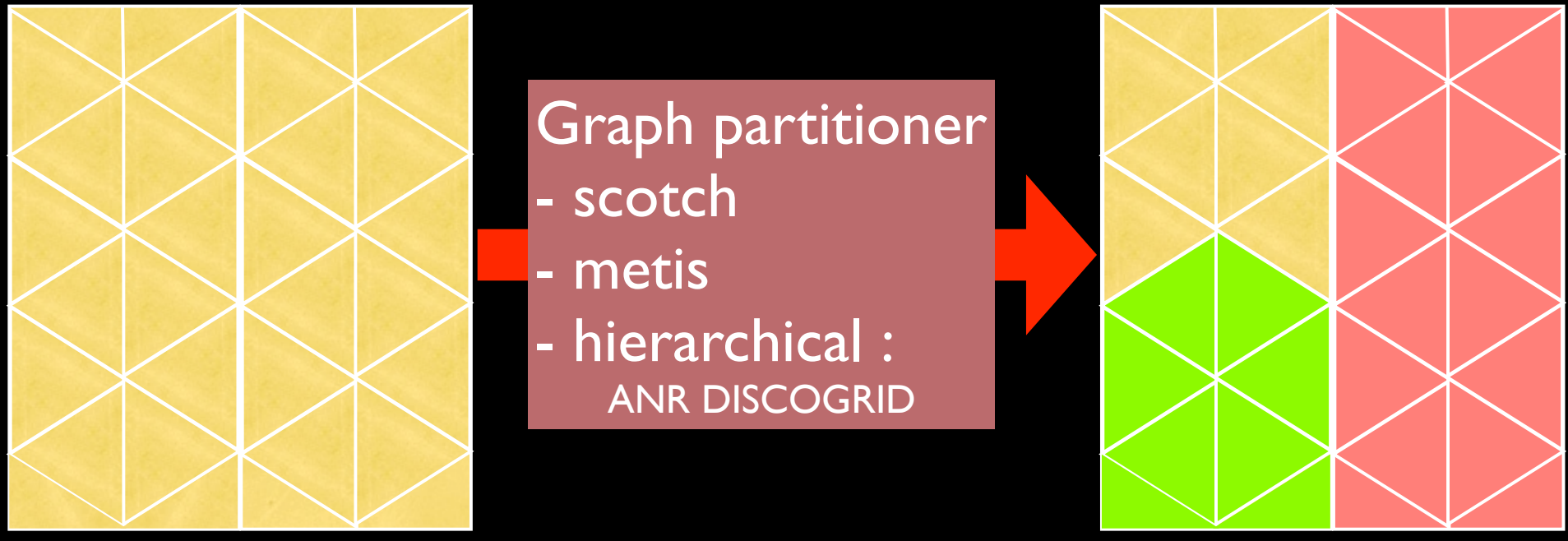


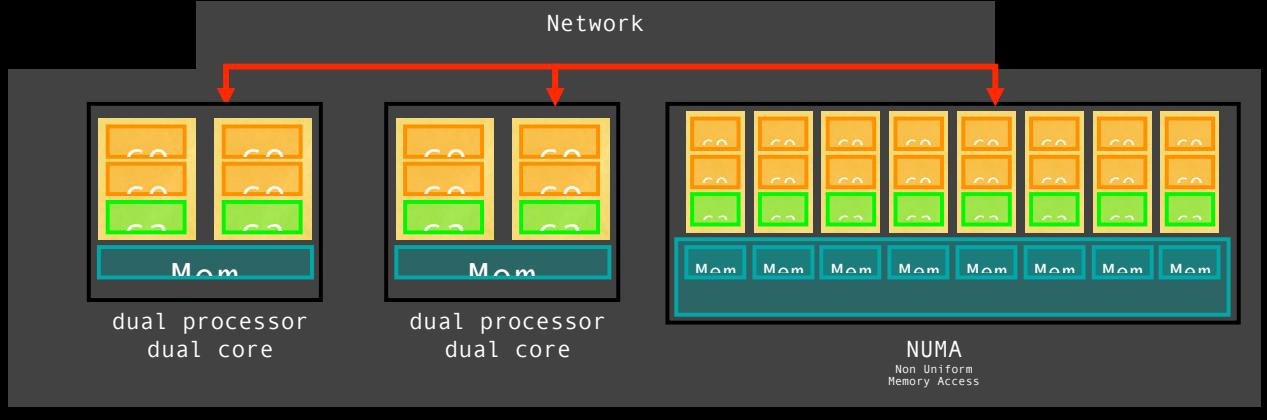
Domain decomposition





Domain decomposition





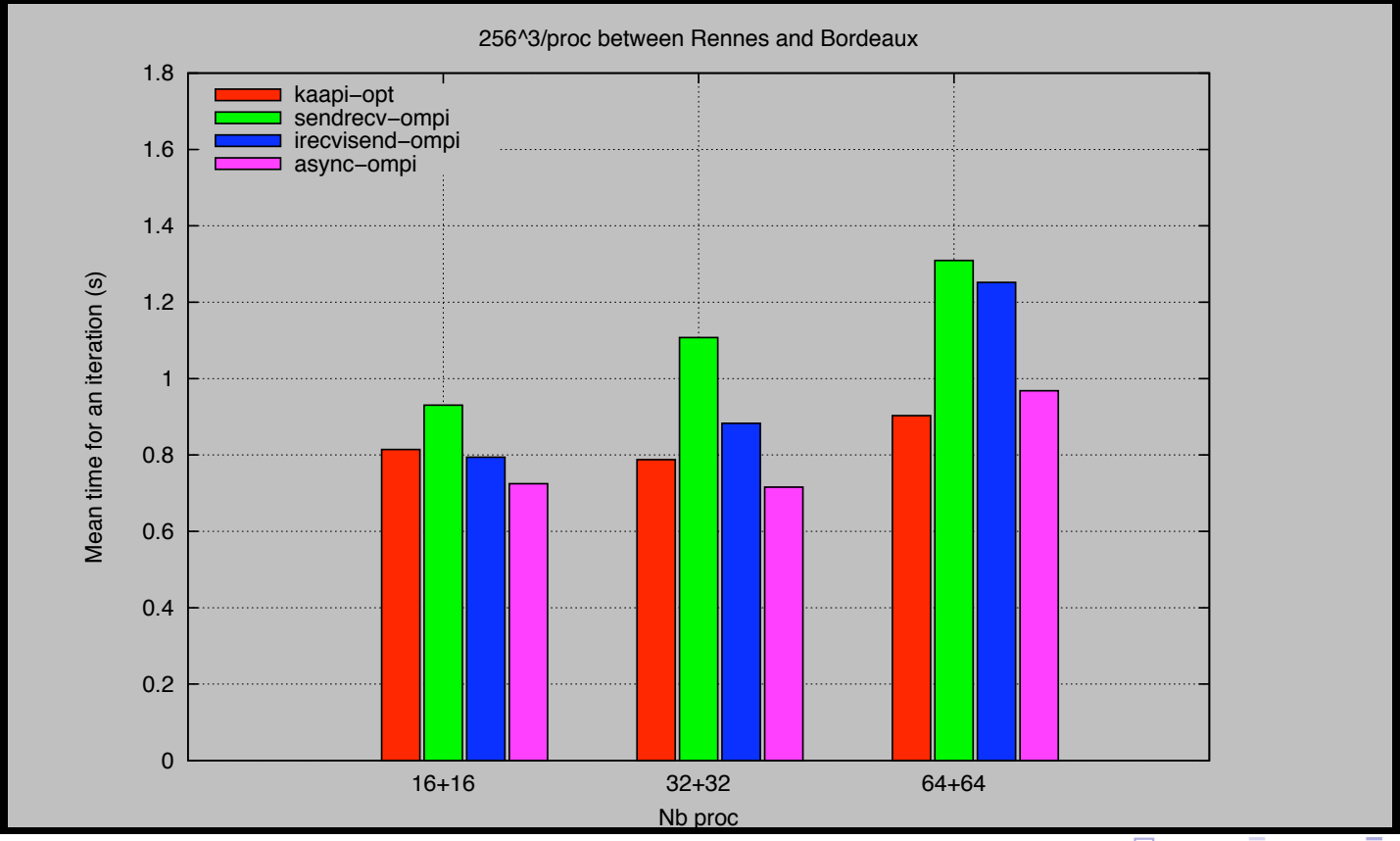
Experiments

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

D=256^3	# processors	Cluster (s)	Grid (s)	Overhead
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

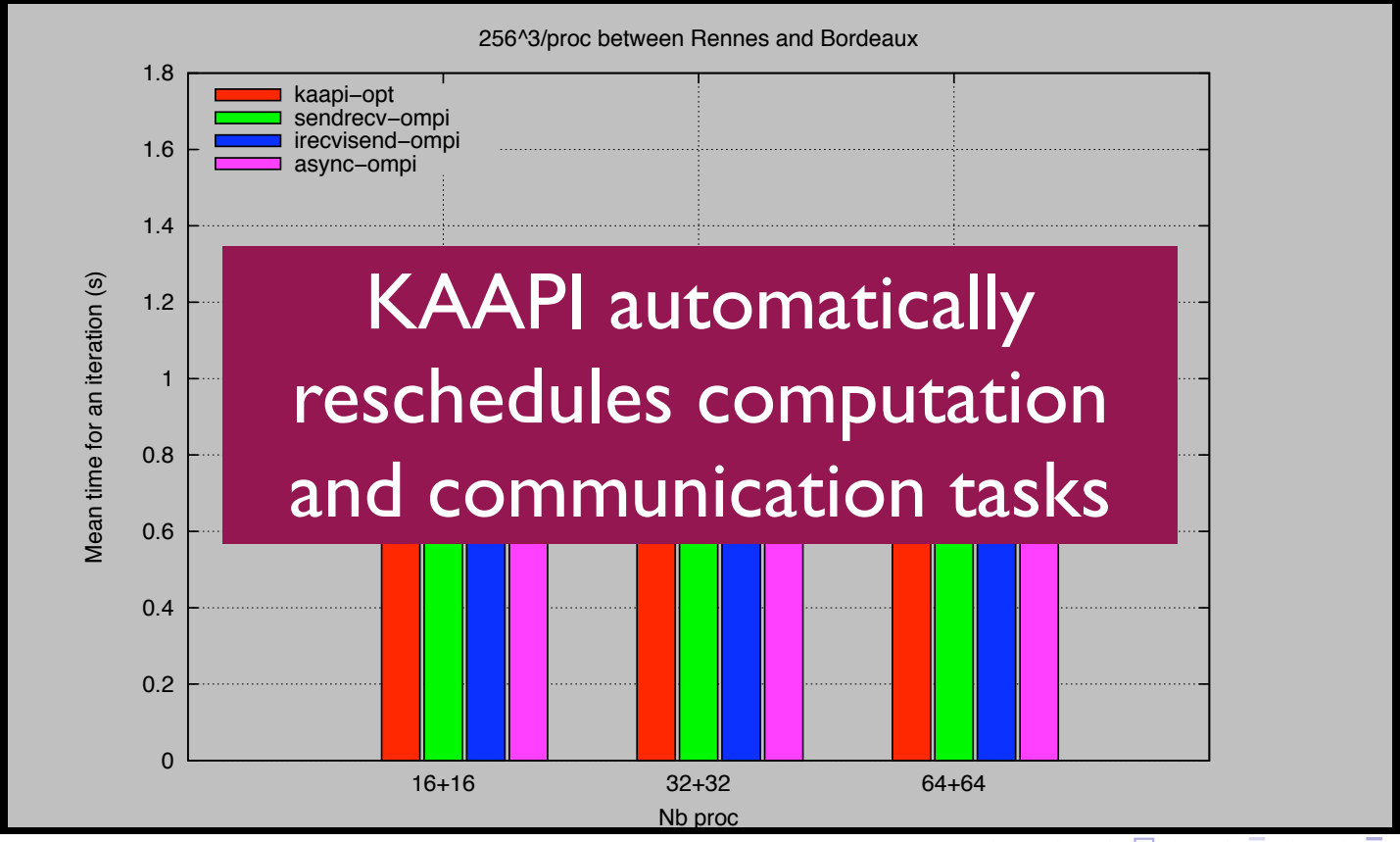
Optimizing MPI code

- **Overlapping communication by computation**
  - At the cost of important code restructuring



Optimizing MPI code

- **Overlapping communication by computation**
  - At the cost of important code restructuring



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