# KAAPI :

# Adaptive Runtime System for Parallel Computing

Thierry Gautier, thierry.gautier@inrialpes.fr
Bruno Raffin, bruno.raffin@inrialpes.fr

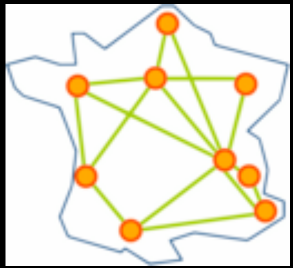
MOAIS project, INRIA Grenoble Rhône-Alpes

# 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**

# Moais Positioning



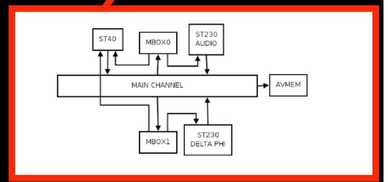
Grid

Cluster

Multicore

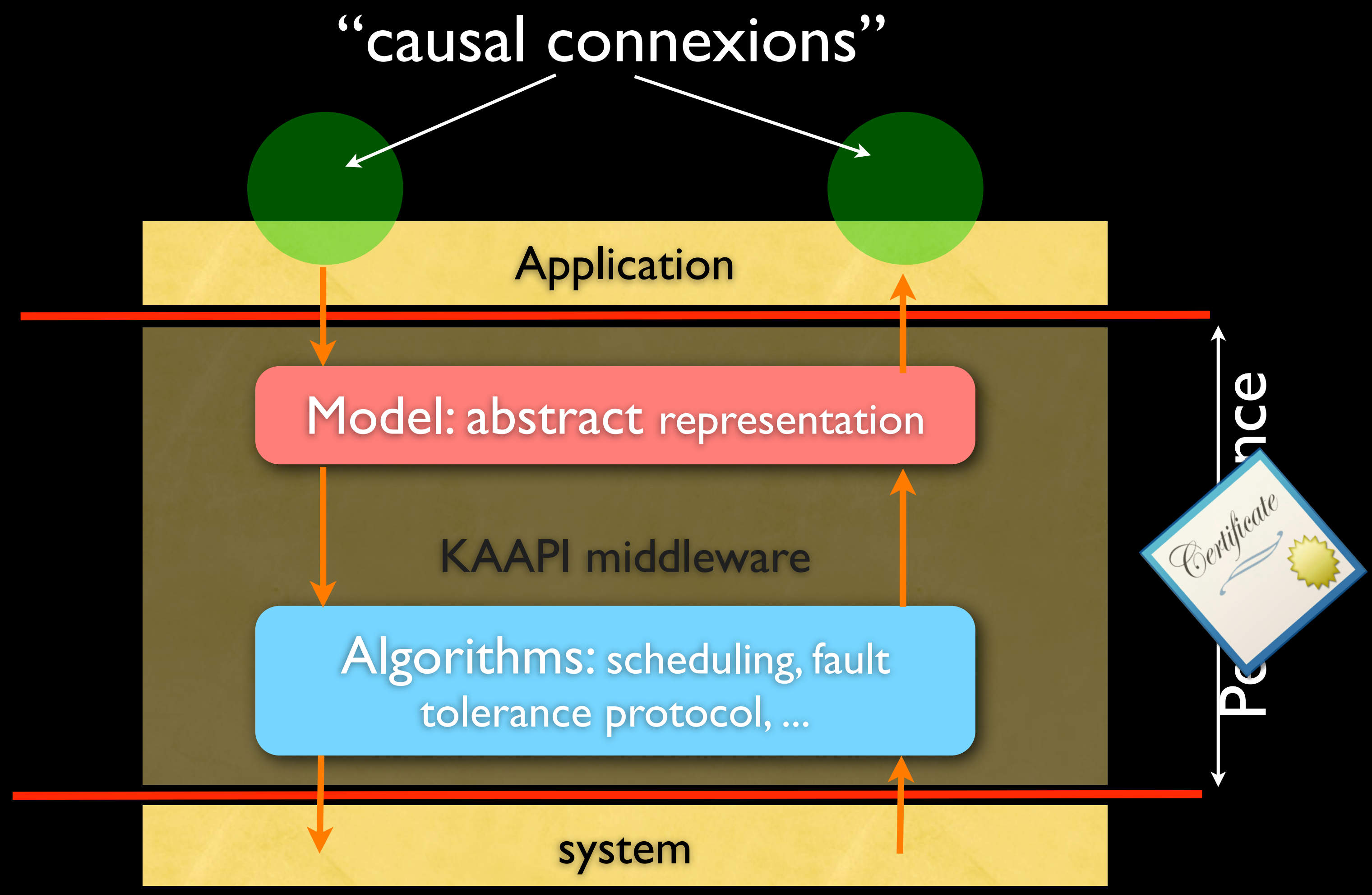

GPU

MPSoC




**To mutually adapt application and scheduling**

# KAAPI Overview



# API

## Global address space

- Creation of objects in a global address space with ‘shared’ type

## Task

- Creation with ‘Fork’ keyword (~ Cilk spawn)
- Tasks only communicate through shared objects

## Automatic scheduling

- work stealing or graph partitioning

## ‘Sequential’ semantics

similar to TBB/Cilk but with data flow dependencies

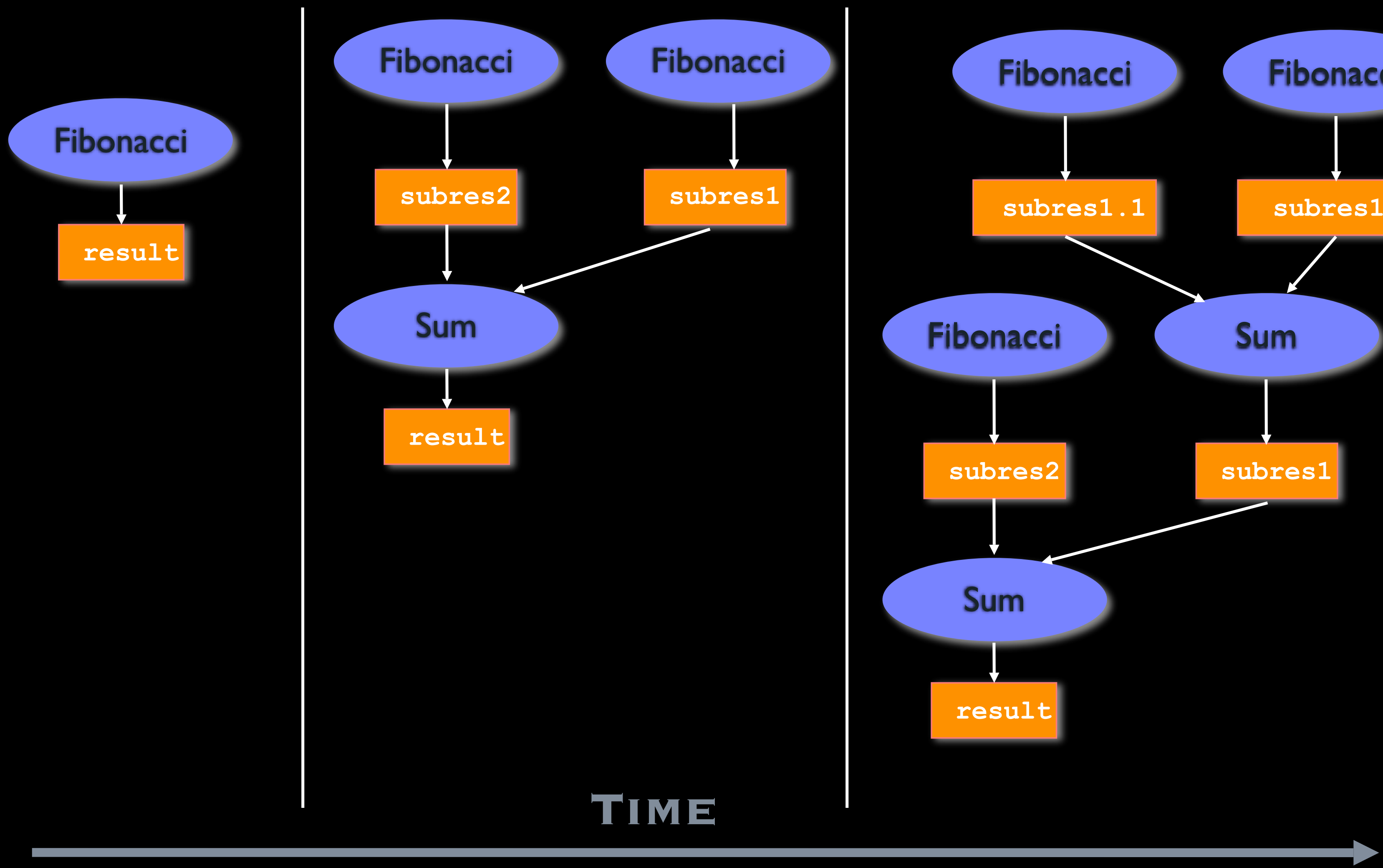
# C++ Elision

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

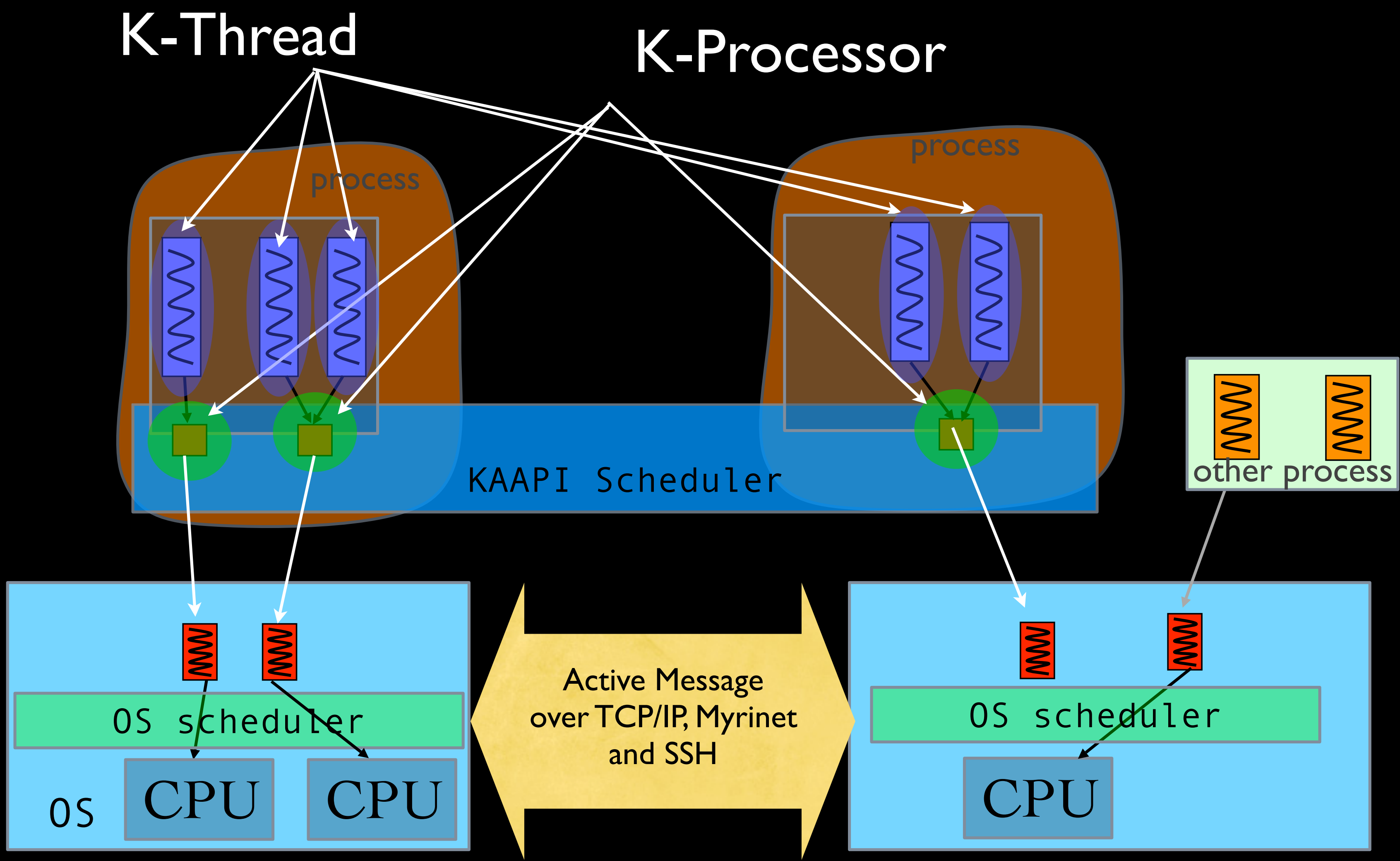
# C++ Elision

```
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          ; }
}
```

# Abstract Representation



# 2 Level Scheduling



# Performance Guarantee

- ## Notations
  - $T_s$ : Sequential work, time of sequential execution
  - $T_1$ : Time of the parallel algorithm on 1 core
  - D: Critical Path
  - P: Number of cores
- ## Properties
  - with high probability, number of steals is

    $$O(P \times D)$$

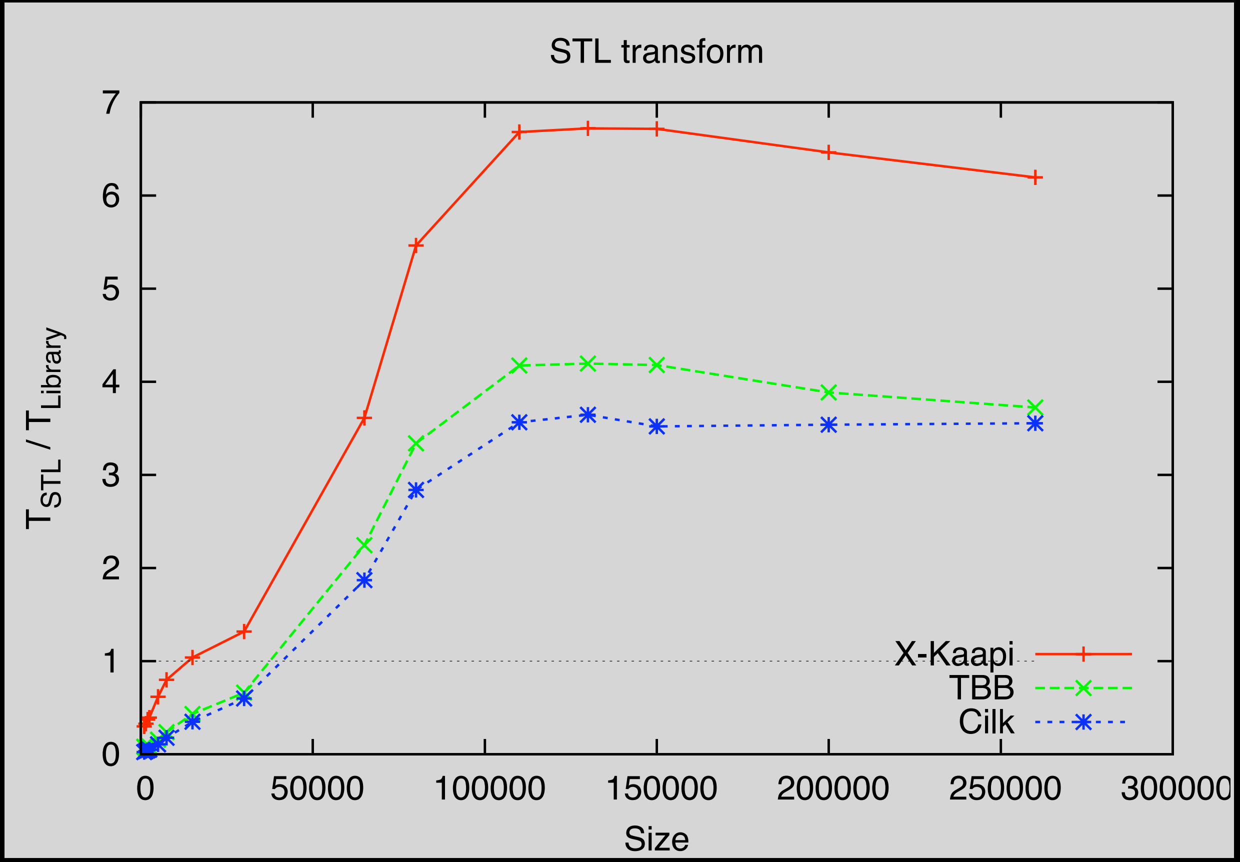
  - with high probability, execution time is

    $$T_p \leq T_1 / P + O(D)$$

~ Also similar bound of Cilk' extension with Rabin et al.

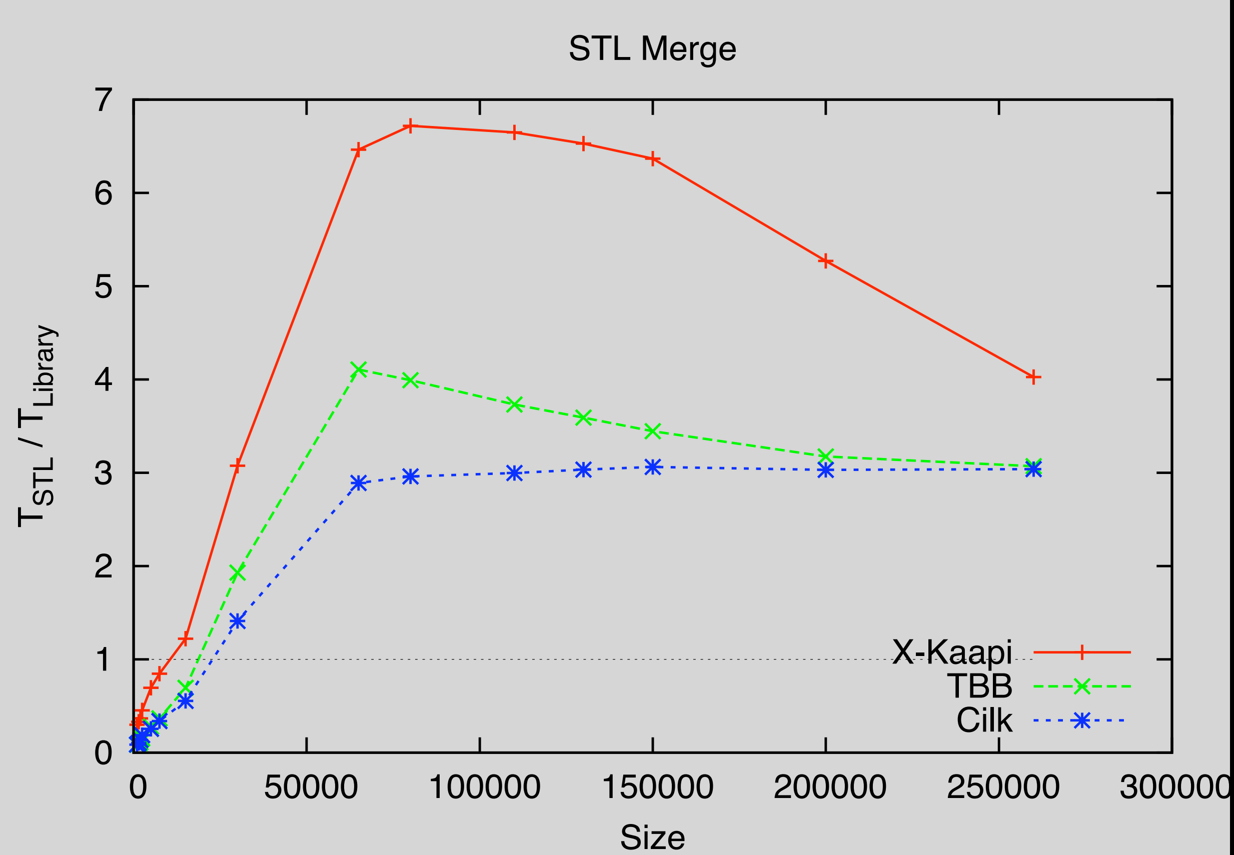
# Comparison with Cilk/TBB

- 8 processors NUMA machine
  - STL Transform, Ratio $T_{stl}$ / $T_{library}$ on 8 cores



# Comparison with Cilk/TBB

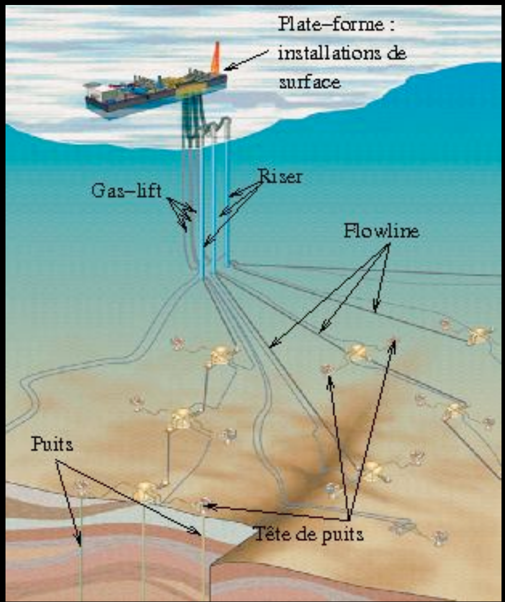
- 8 processors NUMA machine
  - STL Merge, Ratio $T_{stl}$ / $T_{library}$ on 8 cores



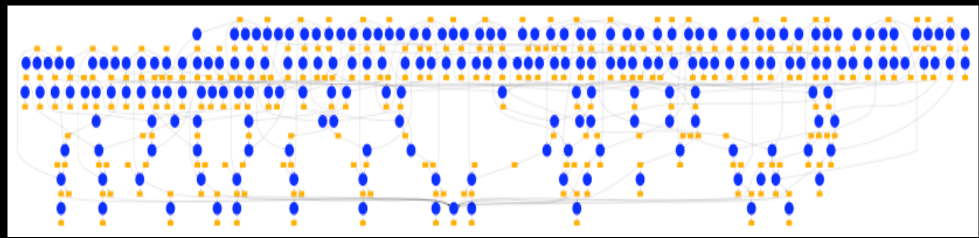
# Grid Experiments

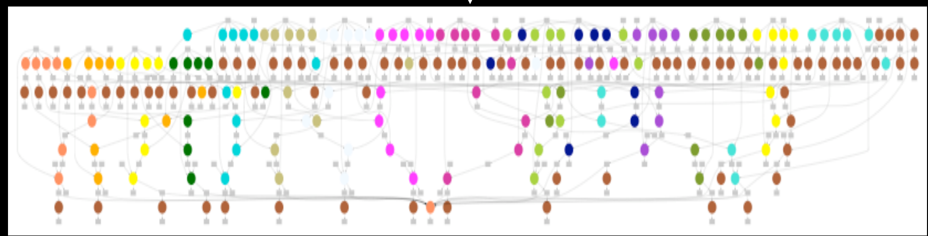
# Iterative Application

- **Scheduling by graph partitioning**
  - **Metis / Scotch**




Application





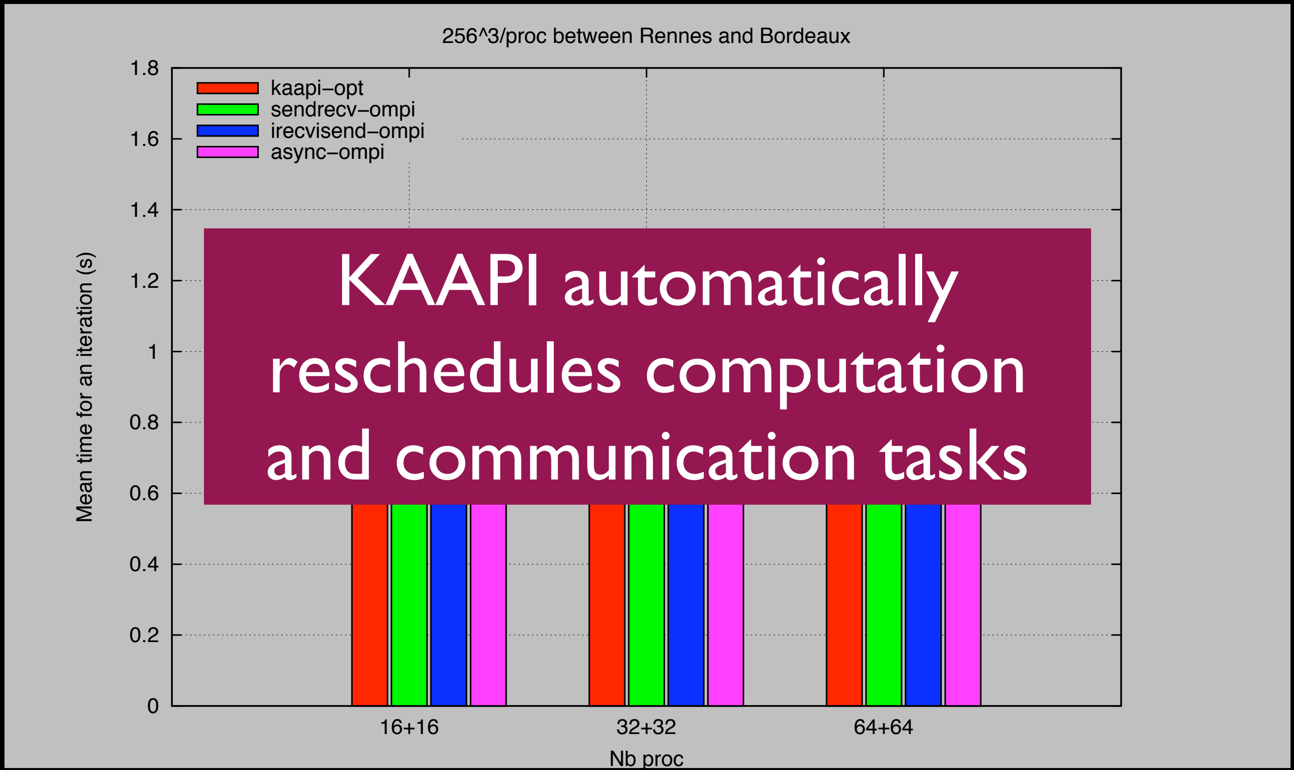
# Experiments

- **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**

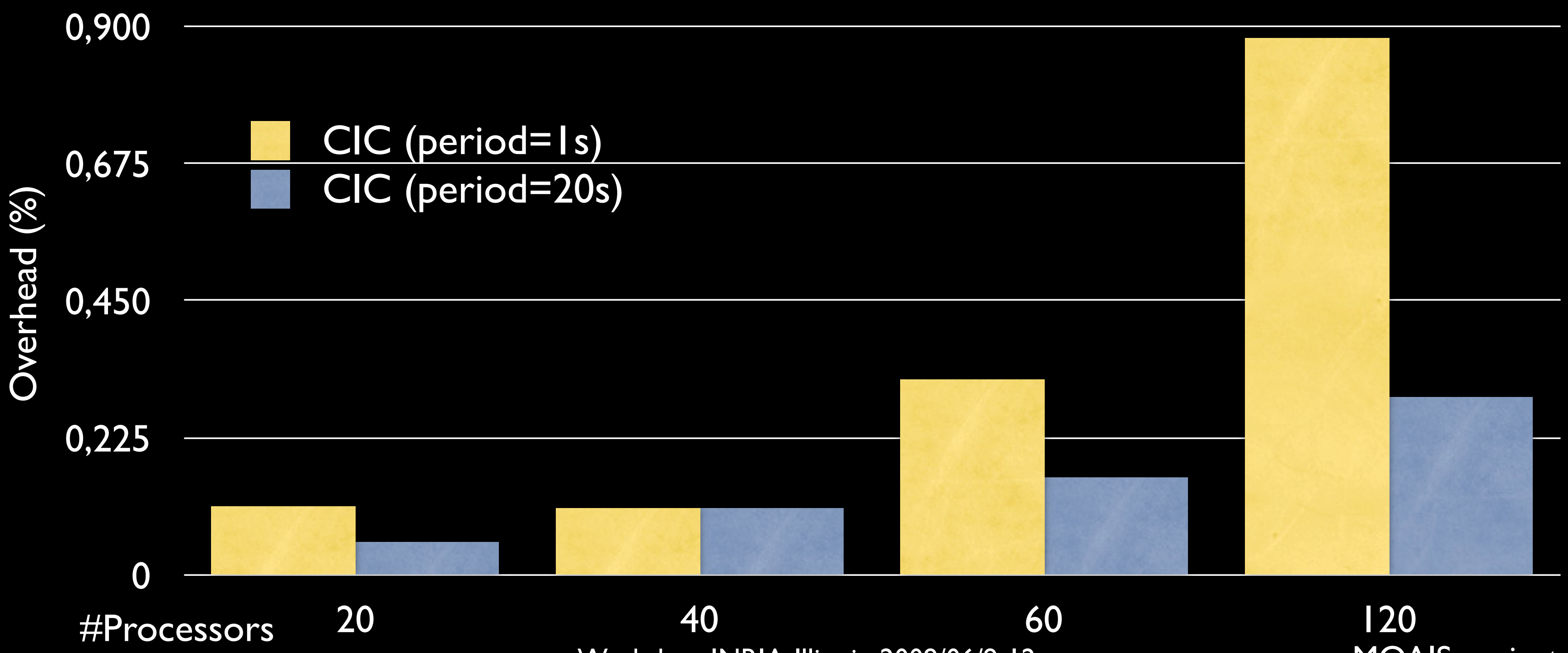


# Fault Tolerance

- State of application = state of the data flow graph
- Two specialized protocols
  - TIC: Theft Induced Checkpointin
    - Periodic checkpoint + forced checkpoint on steal
  - CCK:  for iterative applications
    - Coherent checkpoints
    - only recovery of failed process + $\varepsilon_{application}$

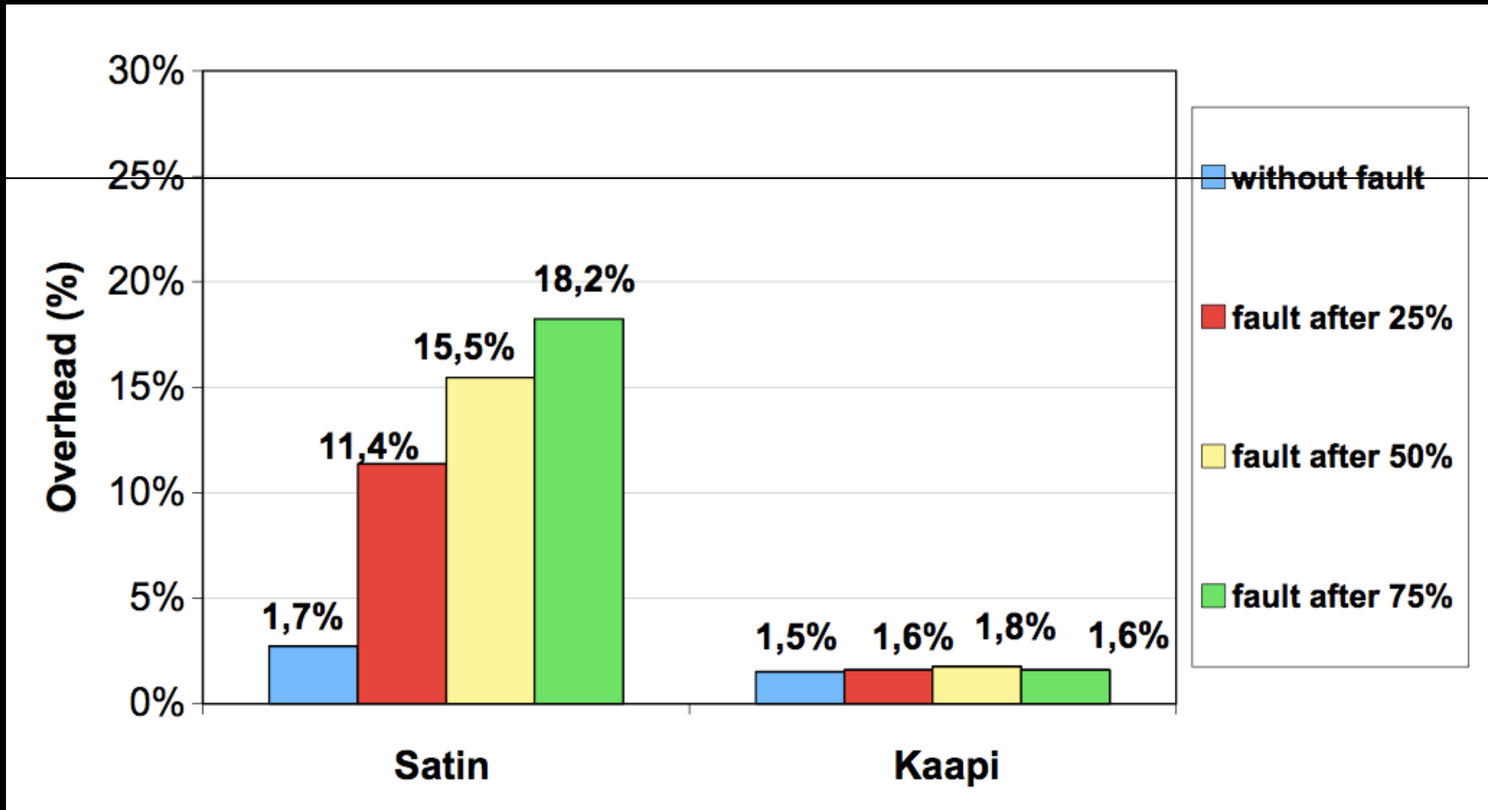
# Protocol Scalability

- **Implemented using distributed checkpoint services**
  - two checkpointing periods
  - max overhead observed: 0.9%
  - TIC: overhead increases as the number of processors increases



# Comparison with Satin

- 32 processors, synthetic recursive app.

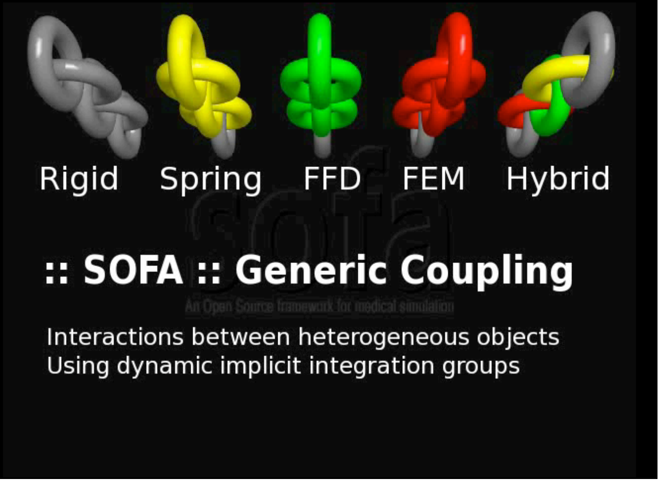


# Physics Simulation

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

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

  Surgery simulation



# Multi CPU/GPU SOFA

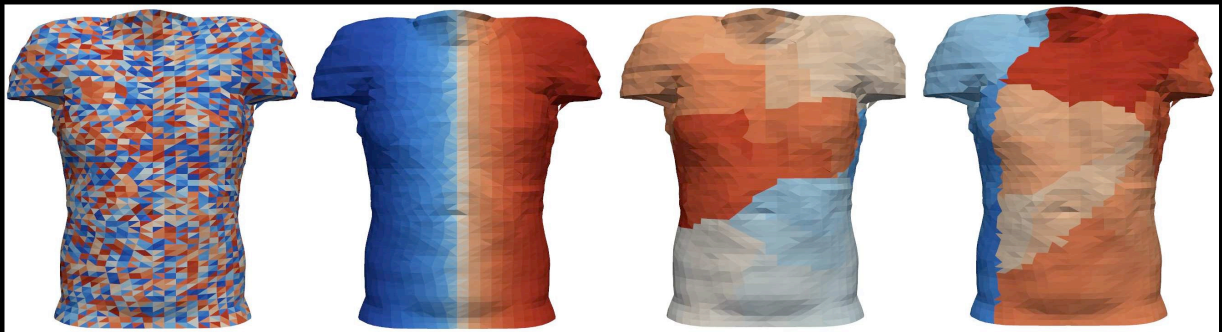
- SOFA: 2 levels of parallelization
  - KAAPI: graph partitioning and work stealing
  - Nvidia Cuda
- On-going: work stealing between CPUs and GPUs

# SOFA

Interactive Physical Simulation

on Multicore Architectures

# Oblivious Algorithms



- **Cache oblivious algorithms**
  - **Irregular meshes: 2-20x on CPU, 1.2-2.7x on GPU**
- **On-going work: cache oblivious + adapted work stealing strategy**

# Conclusions

- KAAPI: flexible framework for parallel programming and fine scheduling control:
  - work stealing : recursive computation or local scheduling
  - graph partitioning : iterative application
- Data dependency graph:
  - used for scheduling or fault tolerance protocols
- On going work on hybrid architectures and large scale machines (BlueGene)

# Questions?

- http://kaapi.gforge.inria.fr
- http://www-sofa-framework.org
- http://moais.imag.fr