Efficient Multi-Core Programming and In-Situ Result Analysis

Bruno Raffin
MOAIS Team, Grenoble, France

Overview

Moais: INRIA research team located at Grenoble

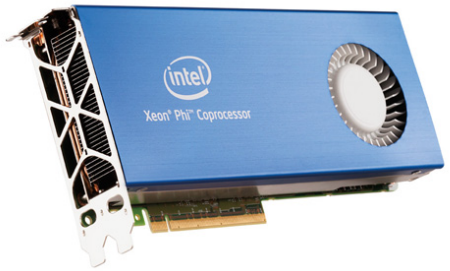
- Focused on parallel programming, algorithms and scheduling

- Main software tools:
  - KAAPI: runtime for efficient multi-core programming
  - FlowVR: middleware for in-situ data processing and visualization

- Long term collaborations with Brazilian teams (started back in the 90's):
  - UFRGS
  - USP

- **Today's talk:**
  - **In-situ result analysis**
  - **Task based multi-core programming**

Introduction



- Multi/many core processors are reshaping the HPC architectures

- Today a compute node can hold:
  - Several multi-core processors (4 sockets, 8 cores each)
  - Multiple accelerators (2 GPUs or 2 Intel Xeon Phi)

- Tianhe-2 (#1@Top500 2013): 16,000 computer nodes with
  - 2 Intel Ivy Bridge processors
  - 3 Xeon Phi chips
  - Total of 3,120,000 cores

→Massive parallelism at node level

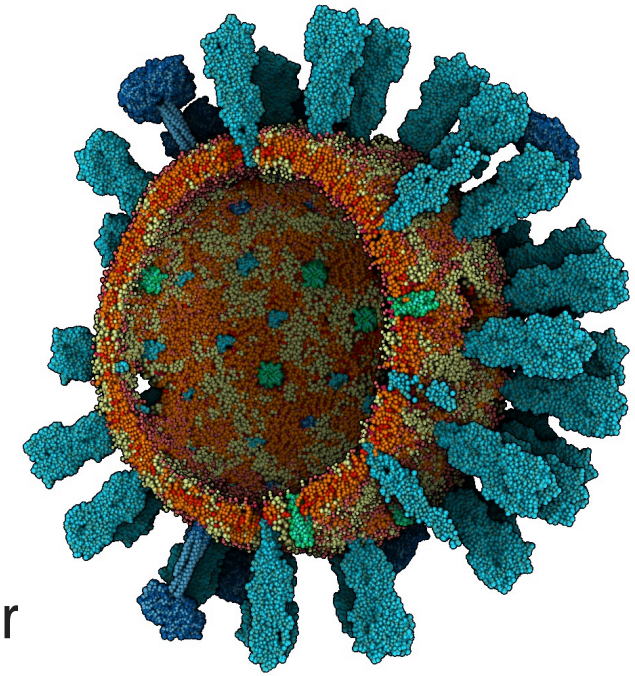
→ Increasing gap between networking and computing capabilities

Challenges:

- **How to efficiently take benefit of all these cores**
- **How to overcome the I/O bottleneck**

In-Situ Processing: Motivation

More processing power → Produce more data (to save)



Difficult to cope with this data deluge:

- I/O system too slow
- Storage capabilities limited
- Post-processing usually performed:
  - at computer center on a small cluster
  - at scientist office on a small machine

→Slow, need to reread the data from disks, transfer them, etc.

In-Situ Processing

Process the data (as much as possible):

- when they are produced
- where they are produced

Main idea: embed part of the post-processing tasks in the simulation

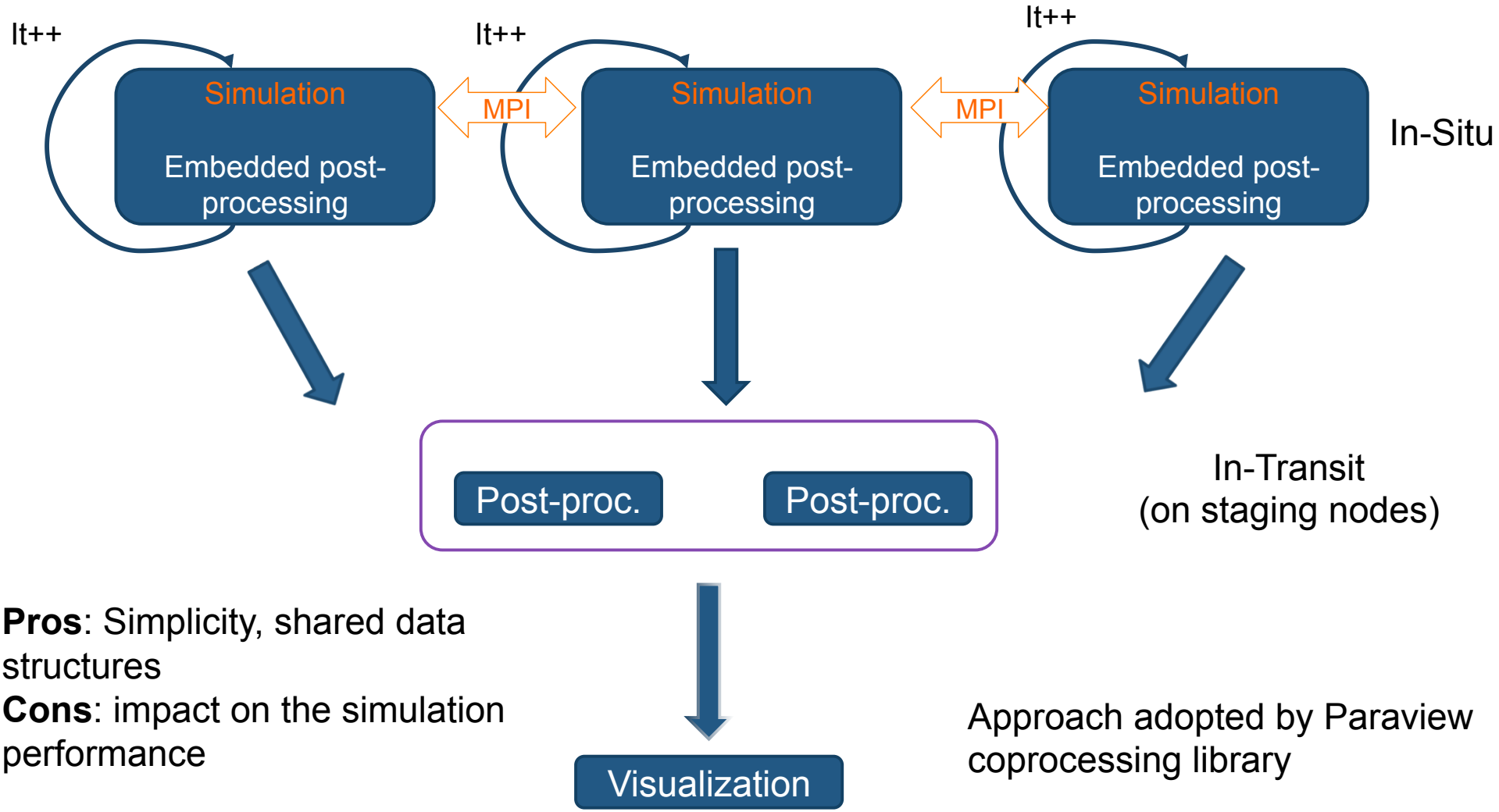
Benefits:

- Reduce the amount of data to move
- Use the supercomputer booked for the simulation
- Enable live result analysis (stop simulation if diverge, finer steering also possible)

Success if:

- Not intrusive on the simulation code
- Limited impact on simulation performance (< 10%)
- Reasonably easy to set-up

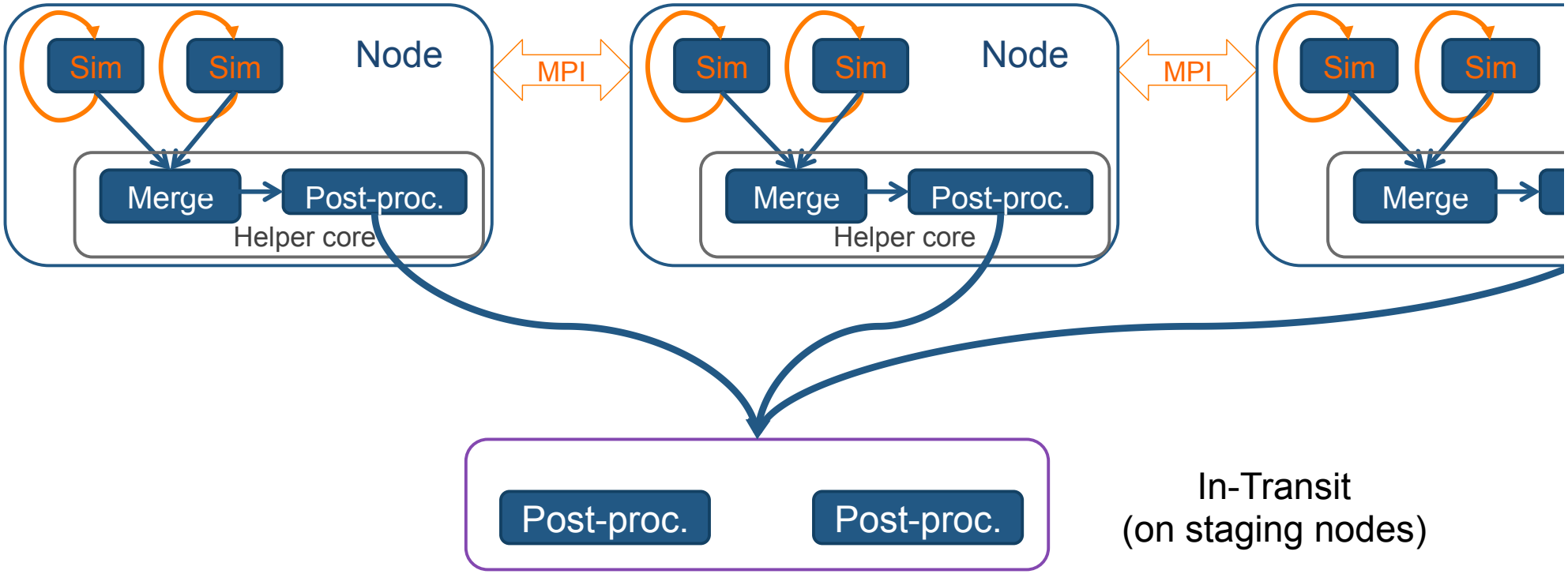
Synchronous In-Situ



Pros: Simplicity, shared data structures
Cons: impact on the simulation performance

Approach adopted by Paraview coprocessing library

Asynchronous In-Situ



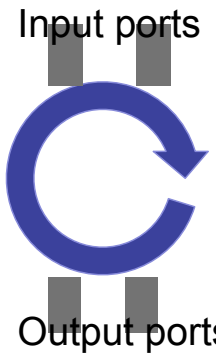
Pros: less intrusive, **more efficient**
Cons: more complex software tool, data copy.

Take benefit of the simulation inefficiency (usually not able to use all the cores of a node)

Approach adopted by FlexIO, Damaris, FlowVR

FlowVR: Middleware for In-situ Processing

1. Develop components:



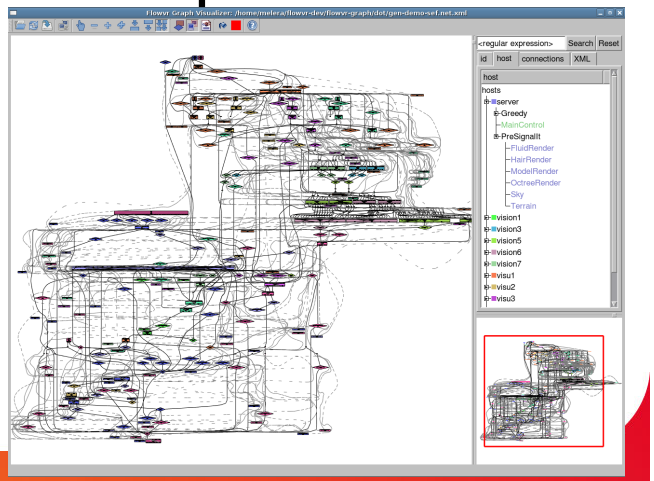

```
While ( wait(inputs) )
        get()
        compute()
        put()
end
```

Simple API (limit code intrusion)

2. Assemble components: Python script
3. Instantiate parameters and execute script

FlowVR

http://flowvr.sf.net

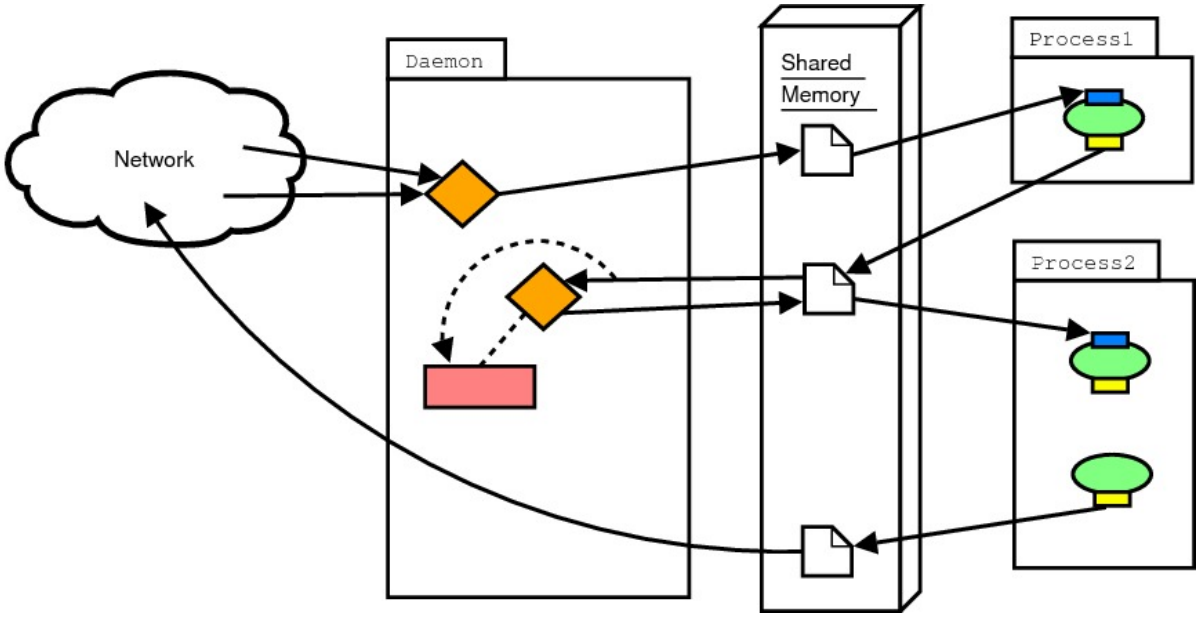


FlowVR: Middleware for In-situ Processing

4. Launch execution

FlowVR runtime takes care of data exchanges:

- Intra-node: pointer exchange through shared memory (limit copies)
- Inter-node: socket or MPI messages.



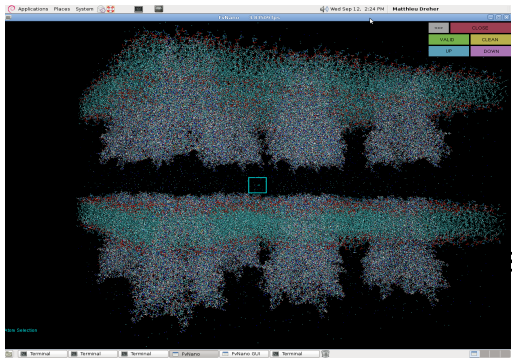
Use Case: Molecular Dynamics

Parallel simulator: Gromacs (MPI)

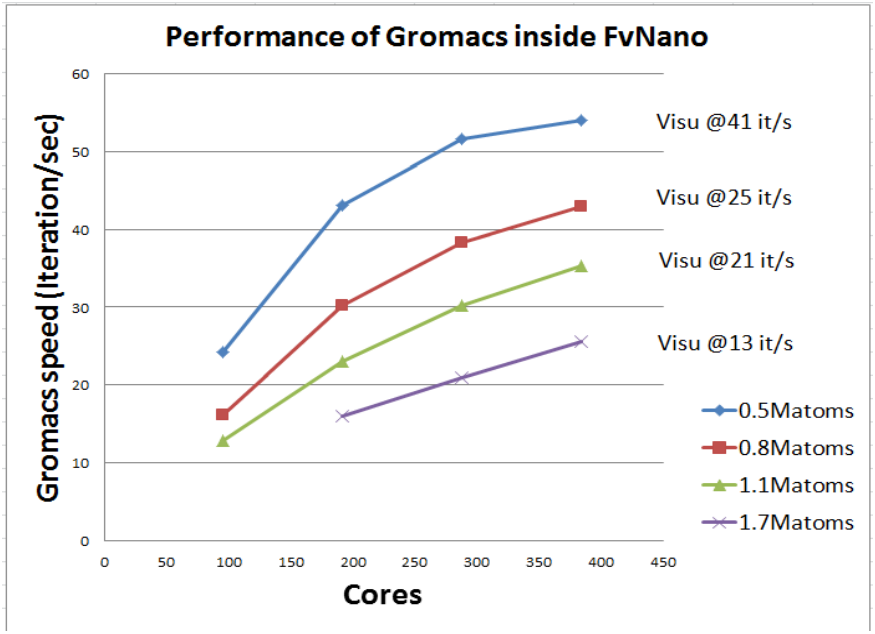
Asynchronous In-situ with FlowVR:

1. Extract atom positions from Gromacs processes
2. Water molecules filtered out asynchronously at each node
3. Remaining atoms forwarded to visualization node

8 cores per node: 7 gromacs processes, 1 for FlowVR
Impact simulation performance by 5%



1.7Matoms model



Published at ICCS 2013

Programming Model: Task Based

```
Int fib (int n)
{
      int x, y;
      if (n<2) return n;
      x = fib (n-1);
       y = fib (n-2);
       return x+y;
}
```

Parallelizing Fibonacci with Cilk

```
cilk Int fib (int n)
{
      int x, y;
      if (n<2) return n;
      x = spawn fib (n-1);
      y = spawn fib (n-2);
      sync;
      return x+y;
}
```

Task declaration

Task creation

Wait the completion of all previously (sequential order) spawned tasks

Tasks model:

- Without dependencies: cilk, TBB, OpenMP
- With: XKAAPI, OmpSS, OpenMP >= 4

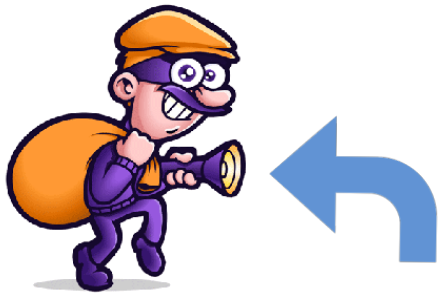
Runtime: Work Stealing

Work stealing: dynamically balance task executions amongst available cores.

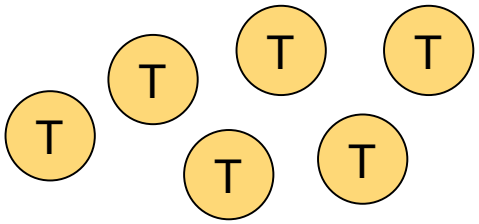
Each core:

- Queue locally generated tasks
- If local tasks available
  - Execute them
- Oterwise
  - Steal tasks from other cores

Work stealing based: Cilk, TBB, XKAAPI

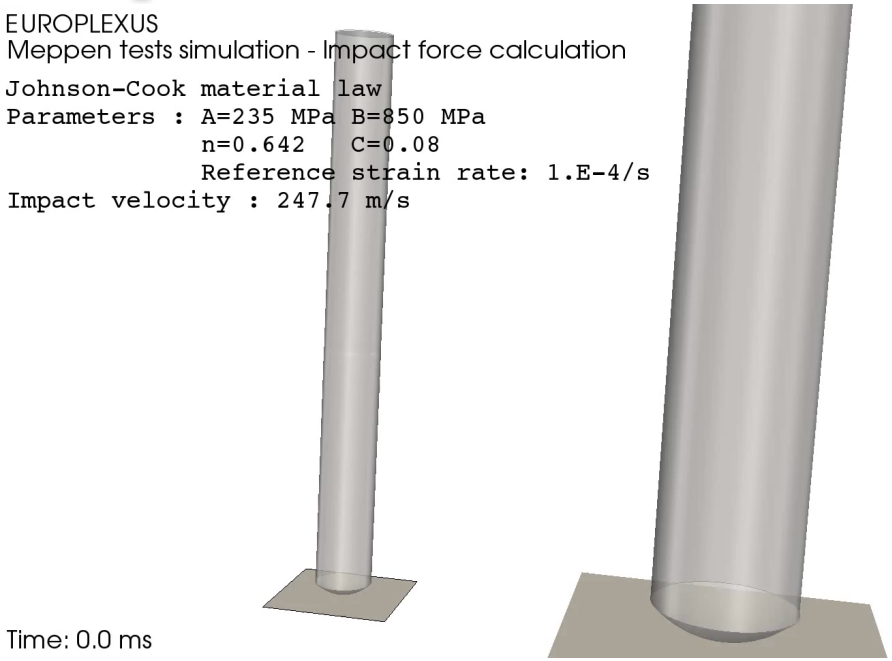


Idle cores steal tasks



Workers generate tasks

Europlexus [CEA - IRC - EDF - ONERA]



Fast transcient dynamics simulator (solid/solid, fluid/solid impact simulations)

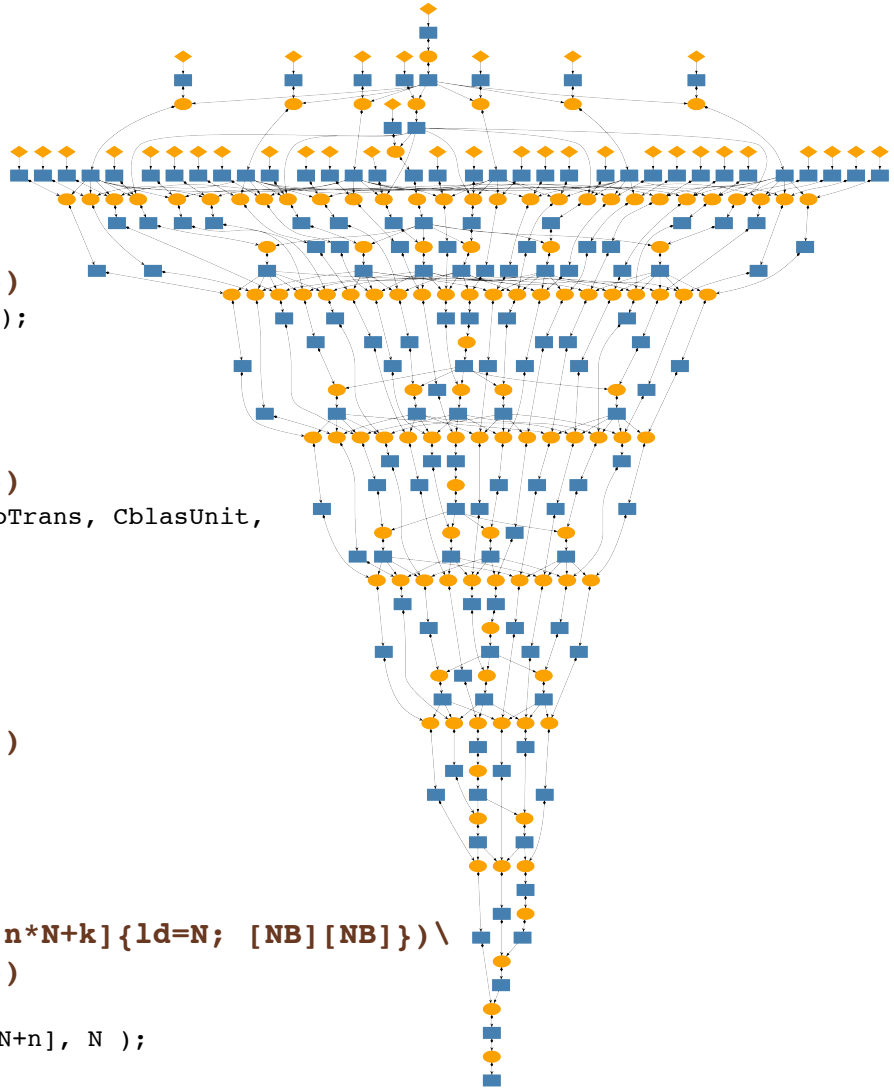
Our contribution: rely on work stealing (XKAAPI) to speed-up intra-node computations

Initial work: parallelization of 2 loops (80% of compute time) and Cholesky factorization

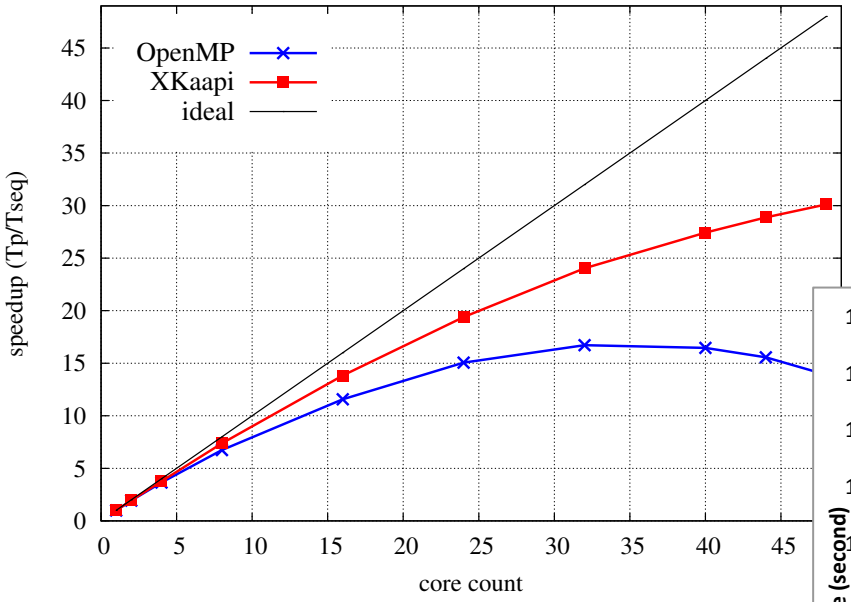
Grand Prix SFEN 2013.

Tile Cholesky factorization

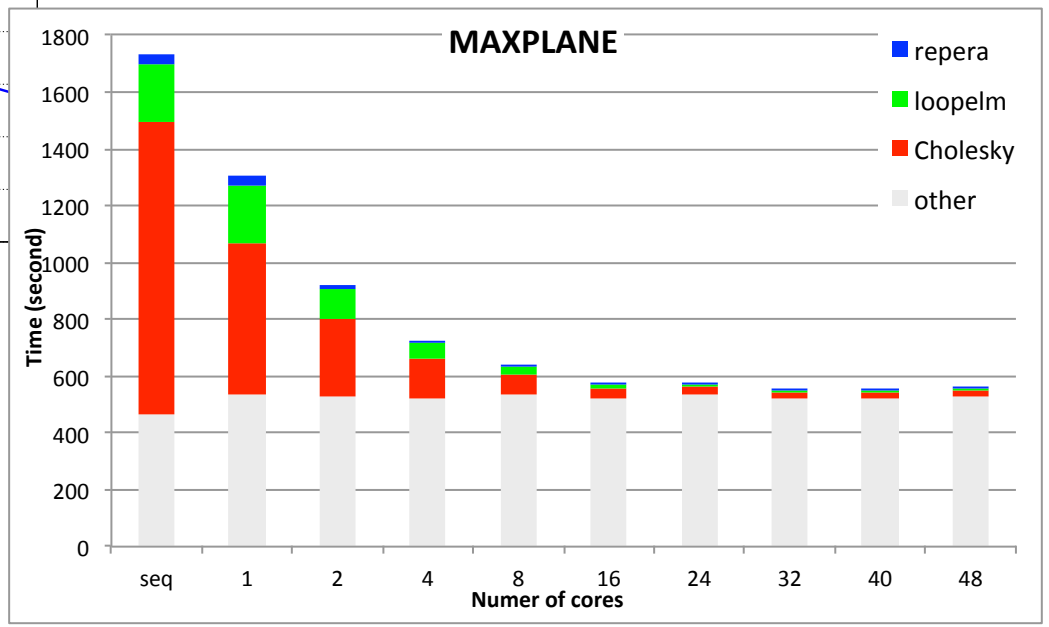
```
#include <cblas.h>
#include <clapack.h>

void Cholesky( double* A, int N, size_t NB )
{
  for (size_t k=0; k < N; k += NB)
  {
#pragma kaapi task readwrite(&A[k*N+k]{ld=N; [NB][NB]})
    clapack_dpotrf( CblasRowMajor, CblasLower, NB, &A[k*N+k], N );

    for (size_t m=k+ NB; m < N; m += NB)
    {
#pragma kaapi task read(&A[k*N+k]{ld=N; [NB][NB]}) \
                   readwrite(&A[m*N+k]{ld=N; [NB][NB]})
      cblas_dtrsm ( CblasRowMajor, CblasLeft, CblasLower, CblasNoTrans, CblasUnit,
        NB, NB, 1., &A[k*N+k], N, &A[m*N+k], N );
    }

    for (size_t m=k+ NB; m < N; m += NB)
    {
#pragma kaapi task read(&A[m*N+k]{ld=N; [NB][NB]}) \
                   readwrite(&A[m*N+m]{ld=N; [NB][NB]})
     cblas_dsyrk ( CblasRowMajor, CblasLower, CblasNoTrans,
        NB, NB, -1.0, &A[m*N+k], N, 1.0, &A[m*N+m], N );

      for (size_t n=k+NB; n < m; n += NB)
      {
#pragma kaapi task read(&A[m*N+k]{ld=N; [NB][NB]}, &A[n*N+k]{ld=N; [NB][NB]})\
                   readwrite(&A[m*N+n]{ld=N; [NB][NB]})
        cblas_dgemm ( CblasRowMajor, CblasNoTrans, CblasTrans,
          NB, NB, NB, -1.0, &A[m*N+k], N, &A[n*N+k], N, 1.0, &A[m*N+n], N );
      }
    }
  }
}
```



Europlexus: Performance Results



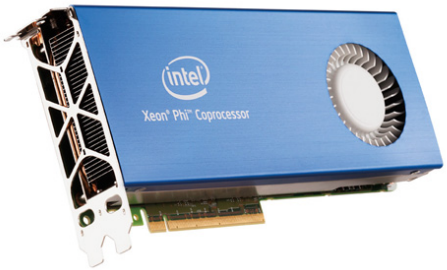
Cholesky factorization.
Matrice size: 59462, Block size: 88



Global speed-up with KAAPI

Future work: OpenMP 4 interface + Kaapi runtime

Early Results with Intel Xeon Phi

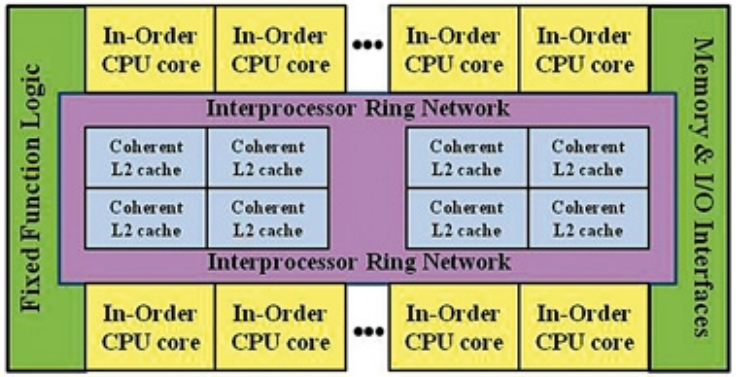


Intel Xeon Phi co-processor:

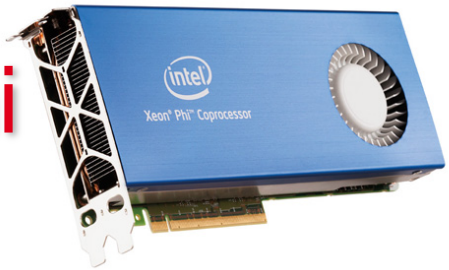
- 60 X86 cores with wide vector processing engine (4 Hyperthreads per core)
- One global memory
- Cache coherent architecture
- Connect on the PCI bus

Supported programming environments:

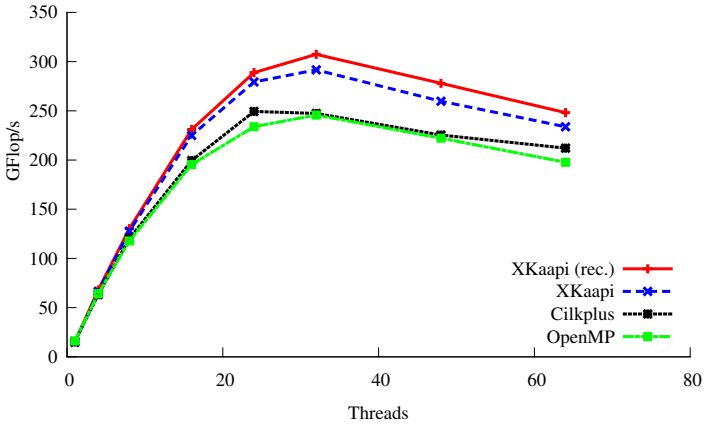
- MPI, OpenMP, TBB, Cilk
- XKAAPI ported in a couple of days



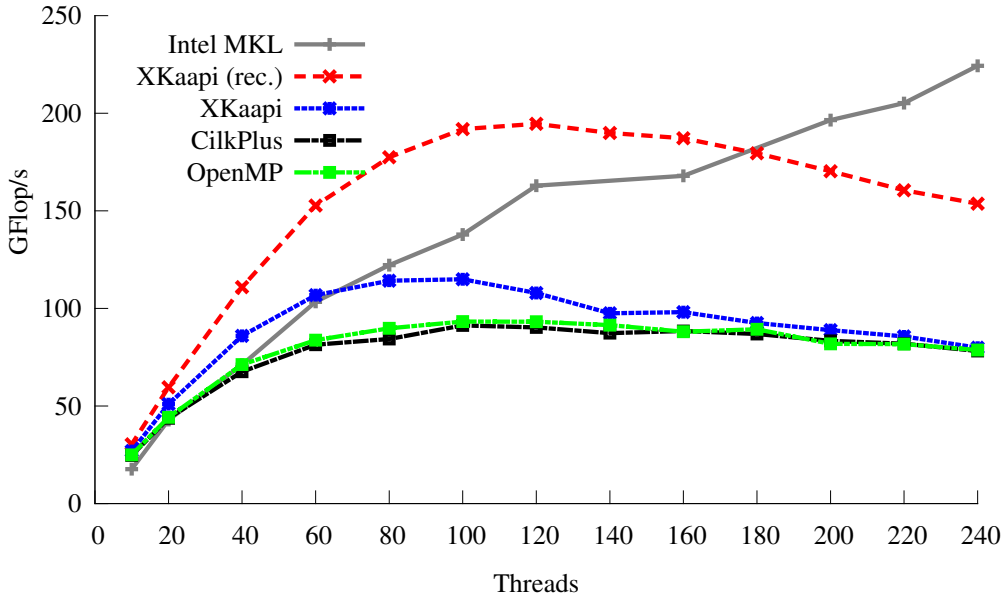
Early Results with Intel Xeon Phi



Main benefit compared to GPU: programming ease



4x8 SandyBridge cores



Intel Mic

Cholesky Factorization
Matrix size 8192, block size: 256

Conclusion

In-Situ Processing:

- Reduce the network traffic and disk usage.
- Post-processing becomes co-processing
- Enable live analysis

FlowVR: a tool for asynchronous in-situ processing

Multi-core progamming with KAAPI

- Task based (OpenMP 4 interface)
- Work stealing runtime
- Multicores, multi-CPU & multi-GPU architectures, intra GPU, intra Xeon Phi

XKAAPI: low overhead, high performance rutime engine.