# Hybrid Bulk Synchronous Parallelism Library for Clustered SMP Architectures

Khaled Hamidouche, Joel Falcou, Daniel Etiemble

hamidou, joel.falcou, de@lri.fr

LRI, Université Paris Sud 11

91405 Orsay, France

## Outline

# Introduction

- Today's machines are hierarchical

  Cluster, SMP, Multi-cores

- Hard to efficiently program

  low level programming model MPI, OpenMP

- Performance depends on

  Application: data size, comm/comp pattern

  Architecture: CPU, bandwidth, …

## High level parallel programming tools

- High level parallel programming models
- High performance
- Easy to manipulate

## BSP Model (Leslie G Valiant:1990 )

- Three components:
  - Machine Model
  - Programming Model
  - Cost model

# BSP Model (Leslie G Valiant:1990 )

## 1- Machine Model

- Describes a parallel machine
  - Set of Processors
  - Point to point communication
  - Synchronization
- Experimental Parameters

  P: Number of processors

  r: CPU speed (FLOPS)

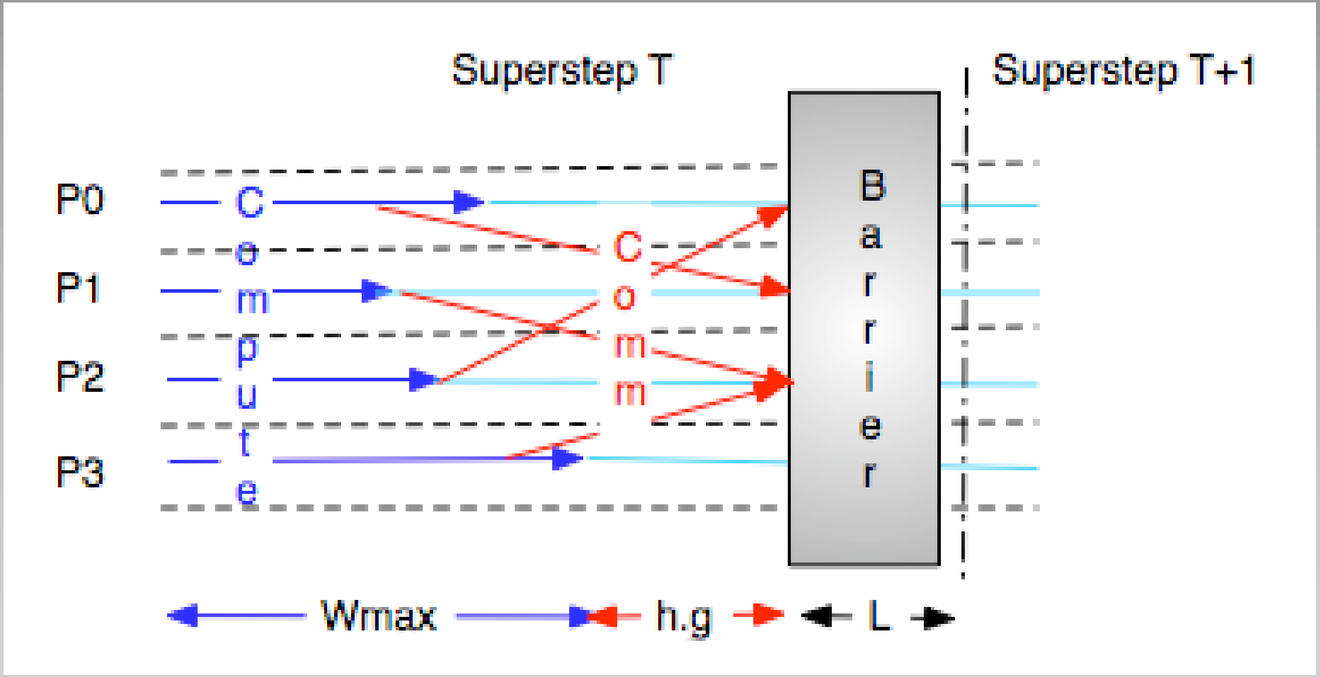
  g: Communication speed (sec/byte)

  L: Synchronization time (sec)

# BSP Model (Leslie G Valiant:1990 )

## 2- Programming Model

- Describes the structure as a sequence of steps



# BSP Model (Leslie G Valiant:1990 )

## 3- Cost Model

- Estimates the time

$T = \sum \delta_i$

$\delta = W\_max + \max h.g + L$

## BSP++

- Object-oriented implementation of the BSML Library [gava:09] in C++
- Notion of Parallel vector
- Functional programming support

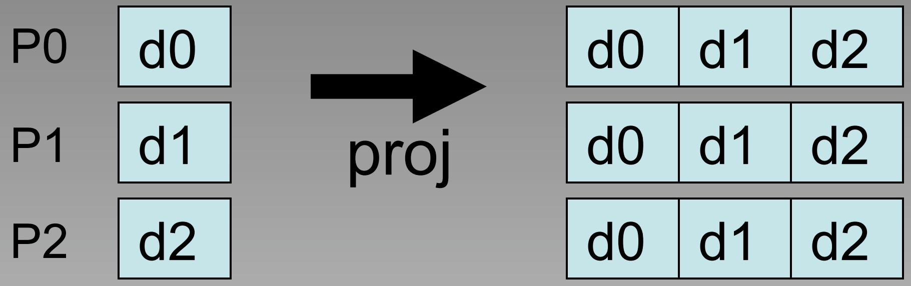
  Boost.Phoenix and C++ lambda-function

# BSP++ API

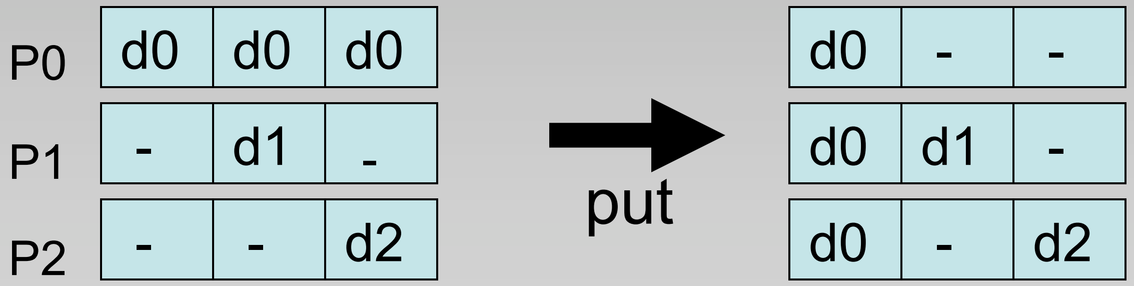
- par<T> : Concept of parallel vector, many constructors
- sync (): Explicit synchronization, MPI or OpenMP barrier
- proj : result_of::proj<T> proj (par<T> &)
- put: result_of:: put<function<T(int)> > put (par<function<T(int)> >& )

# BSP++ API

- proj : MPI_allgather and asynchronous
  OpenMP copy
  + sync ().

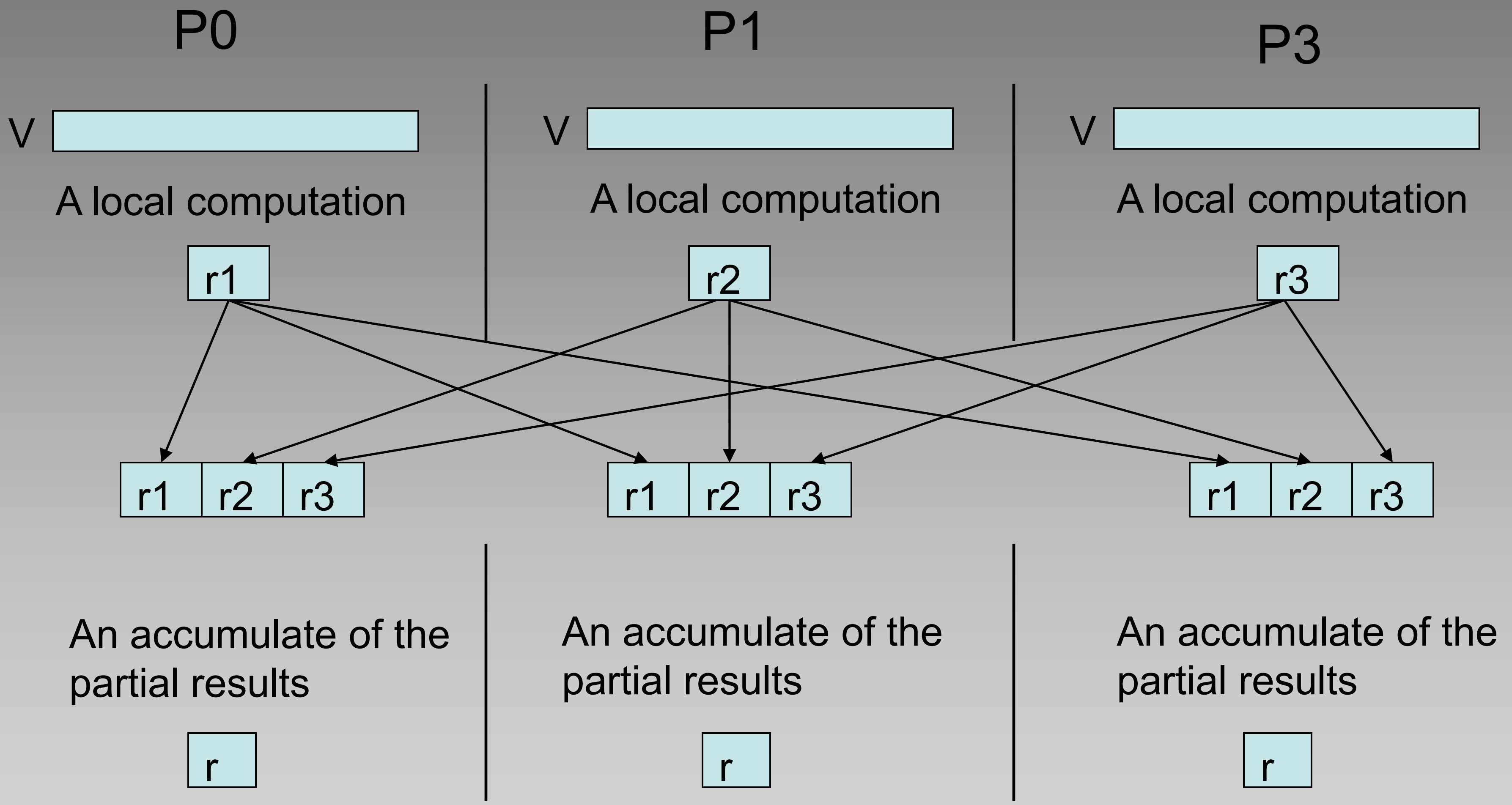


- put: Matrix P: Pij = value of Proc i to send to Proc j
  MPI_alltoall and asynchronous OpenMP copy.
  + sync ().



# Example

Inner product program


P0
P1
P3
V
V
V
A local computation
A local computation
A local computation
r1
r2
r3
r1 r2 r3
r1 r2 r3
r1 r2 r3
An accumulate of the partial results
An accumulate of the partial results
An accumulate of the partial results
r
r
r

# Example

## BSP++ Inner product

```
# include<bsppp/bsppp.hpp>
     int main (int argc, char** argv)
     {
       BSP_SECTION(argc, argv)
       {
           par<vector<double> >   v;
           par< double >          r;

           // step 1 :  perform local inner-product
           *r=std::inner_product( v->begin(),  v->end(), v->begin(), 0.);

           // the global exchange
           result_of::proj<double> exch = proj (r);

           // step 2 :    accumulate the partial results
           *r= std::accumulate (exch.begin(), exch.end() );

           sync ();
         }
      }
```

# Hybrid programming support

Objection of BSP is the cost of L

(dominant for large parallel machines)

Table. Variation of L (in ms) and g (in second per M b) on A 4x4 cores machine (AMD machine)

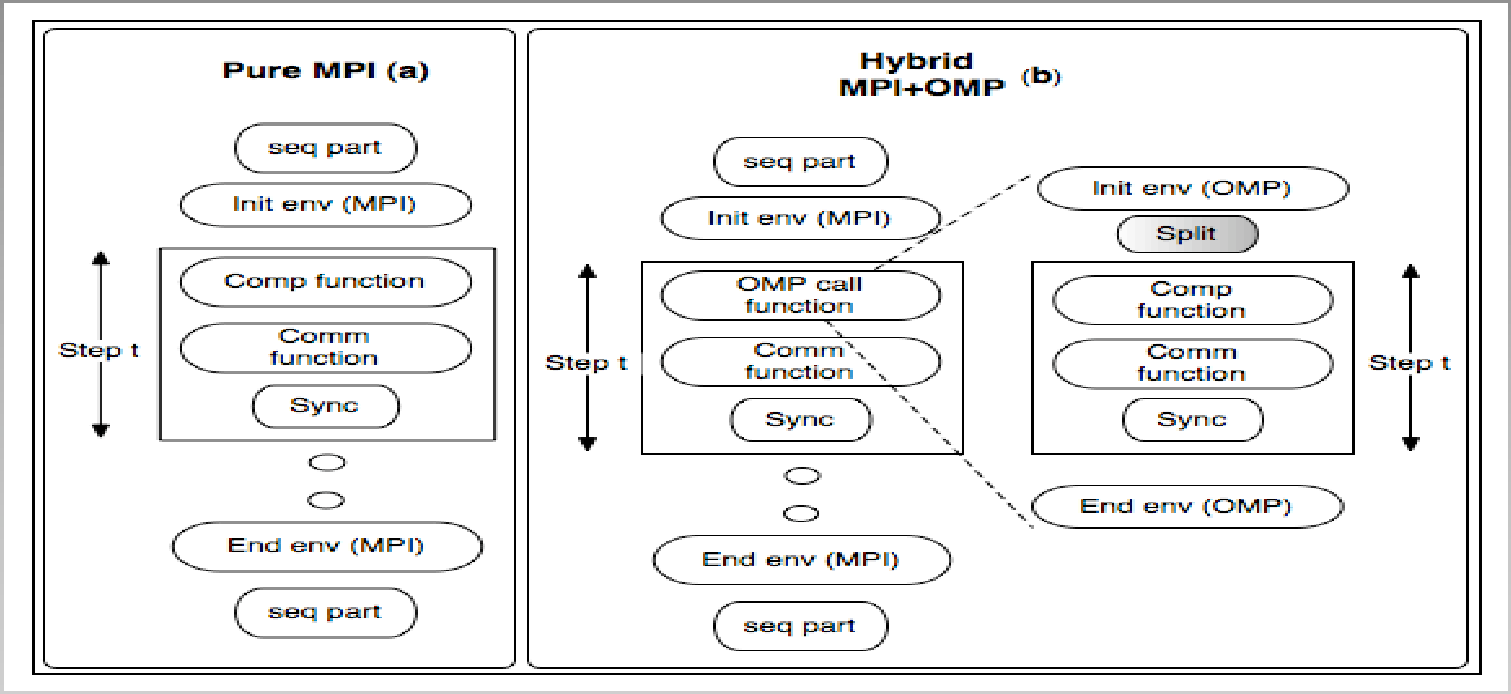
| | MPI | | | OpenMP | | |
|---|---|---|---|---|---|---|
| P | 4 | 8 | 16 | 4 | 8 | 16 |
| g | 0.087 | 0.22 | 1.69 | 0.025 | 0.069 | 0.68 |
| L | 4.46 | 20.8 | 108.0 | 2.94 | 8.13 | 13.1 |

- Impact of OpenMP: synchronization is up to 8 times faster

- Turn the hybrid BSP machine into two BSP machines with different values of L and g

# Hybrid BSP with BSP++

- Same code for both MPI and OpenMP
- Add a split function



δ = Wmax + h_mpi .g_mpi + h_omp .g_omp + L_mpi + L_omp

# Hybrid BSP++ example

```
double omp_inner_prod (vector<double> const& in, int argc, char ** argv )
{
   double value;
   BSP_SECTION(argc, argv)
   {
      par<vector<double> > v= split (in);
      par<double>                                     r;
      *r = std::inner_product(v->begin(), v->end(), v->begin(), 0.);
      result_of::proj exch = proj(r);
      value = std::accumulate (exch.begin(), exch.end());
   }
   return value;
}
```

```
BSP_SECTION(argc, argv)
{    par<vector<double> >       data;
     par<double>                         result;
     *result= omp_inner_prod (*data, argc,argv);
     result_of::proj<double> exch= proj(result);
     *result= std::accumulate (exch.begin(), exch.end() );
}
```

# Experimental results

- Platforms :

1- AMD machine:

* 2 GHz Quad processor quad cores (16 cores)

* 16 Gb of RAM (shared memory)

* gcc4.3, OpenMP 2.0 and OpenMPI 1.3

2- CLUSTER machine:

* Grid5000 platform; Bordeaux site

* 4 nodes, Bi-processor Bi-cores (2,6 GHz)

* gcc4.3, MPICH2.1.0.6 library

# Experimental results

- Protocols :

1- BSP++ vs BSPlib:

* AMD machine

* EDUPACK benchmarks (Inprod, FFT, LU)

2- BSP++: MPI vs OpenMP:

* AMD machine

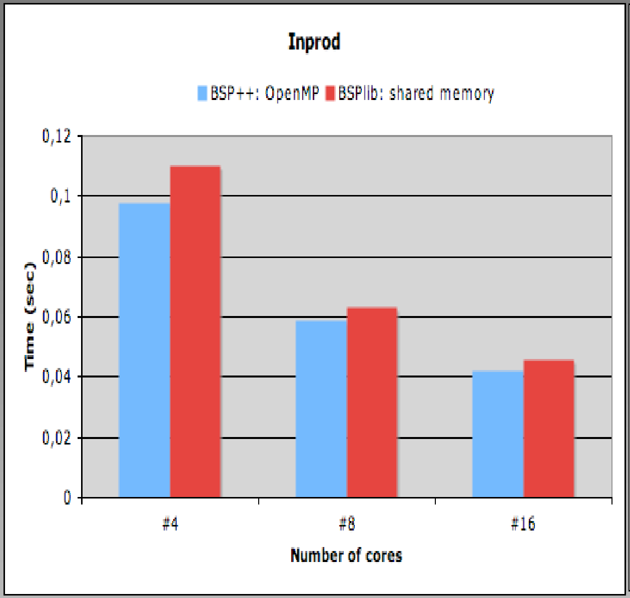
* Inprod, Matrix-vector Multiplication GMV, Matrix-matrix Multiplication GMM and Text Count function of the google MAP reduce Algorithm Benchmarks

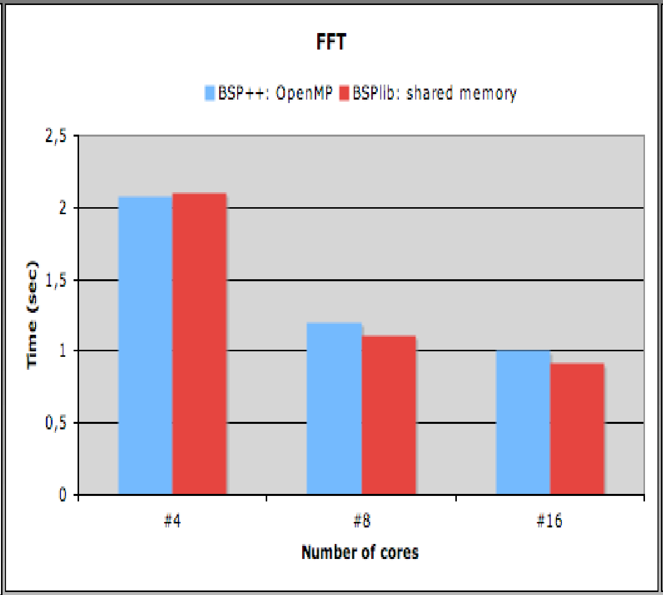
3- BSP++: MPI vs Hybrid:

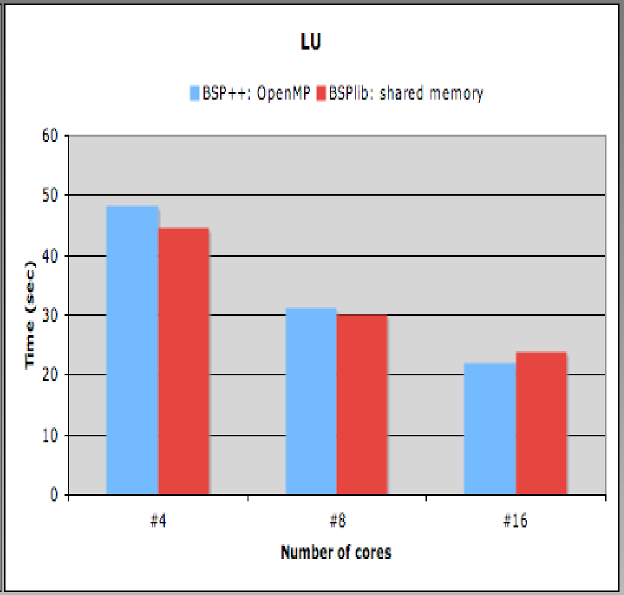
* Cluster machine

* Same benchmarks

# 1- BSP++ vs BSPlib



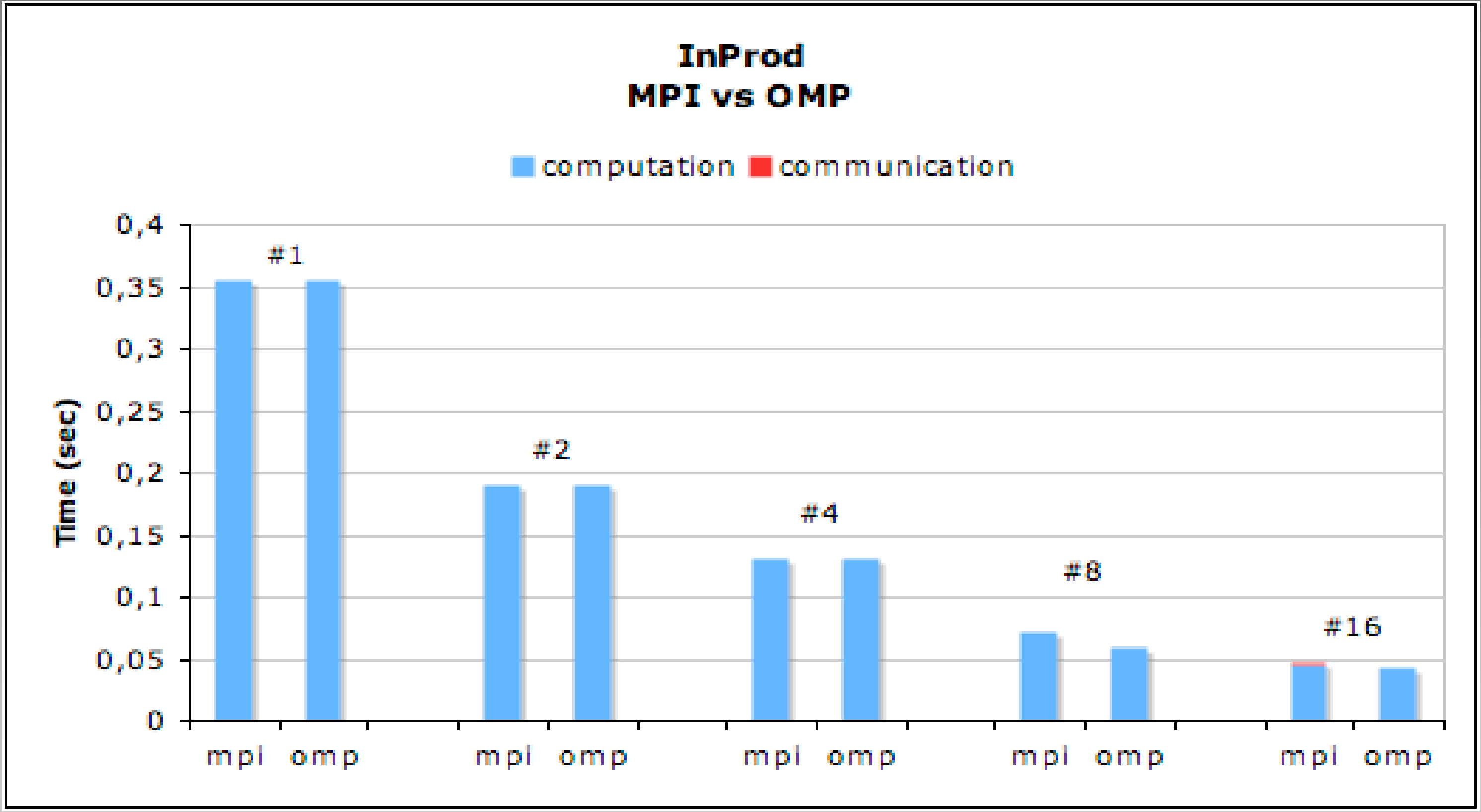




Overall execution time for BSP++ on OPENMP and the BSPlib EDUPACK benchmarks on the AMD machine

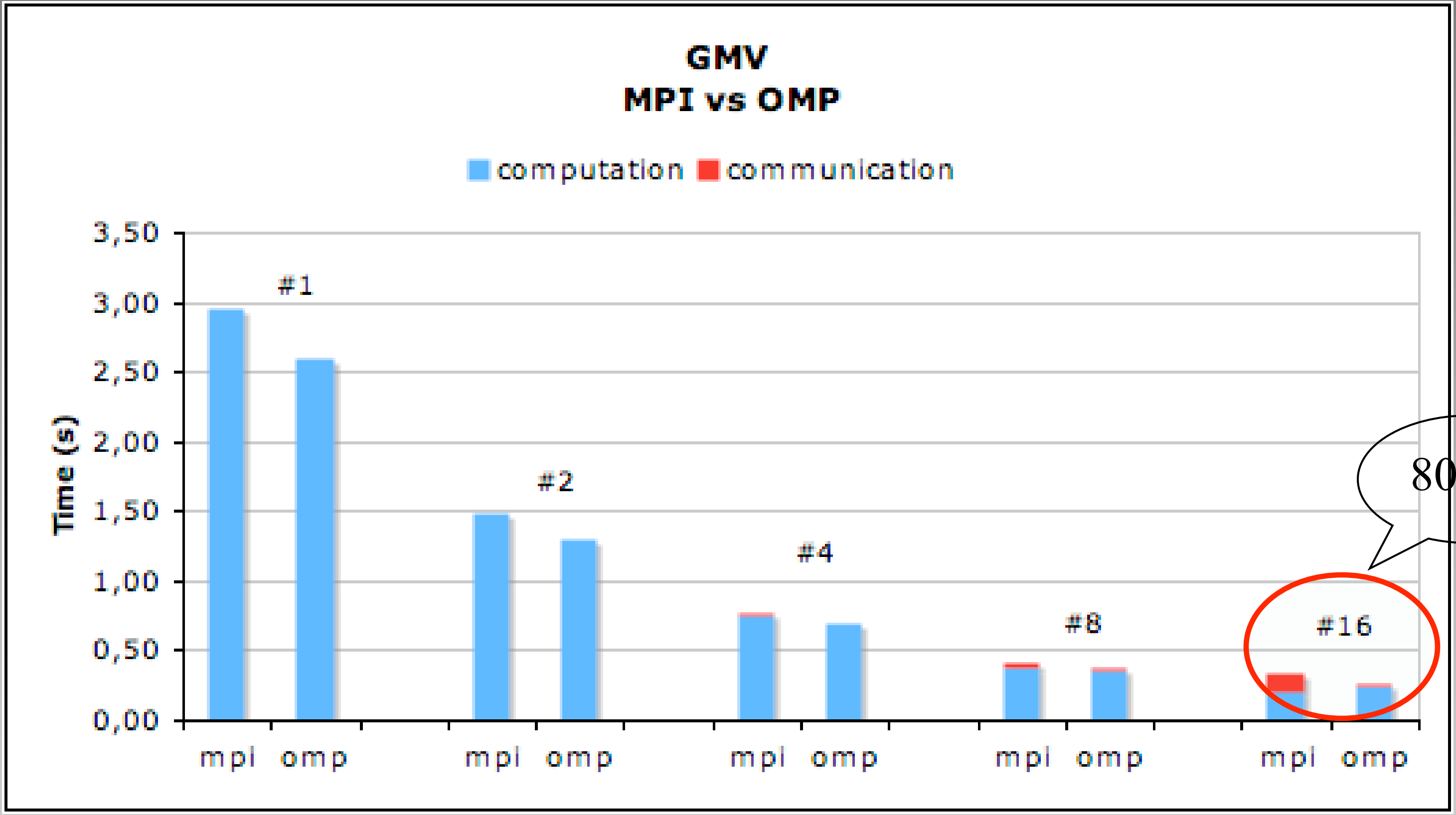
- Same performances
- No overhead of the generic template implementation

# 2- BSP++: MPI vs OpenMP



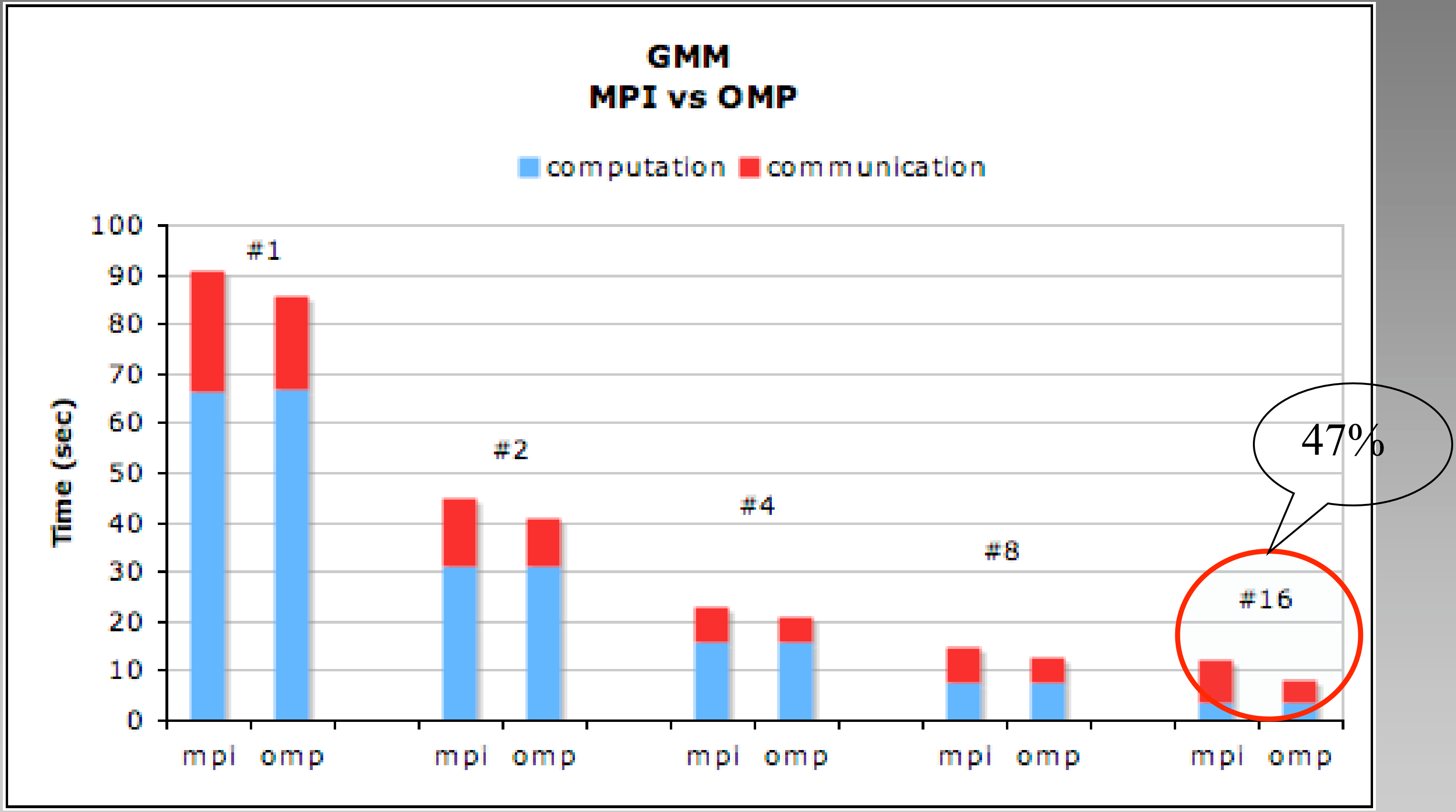
Execution time of the InProd benchmark on the AMD machine for 64 10^6 elements

# 2- BSP++: MPI vs OpenMP



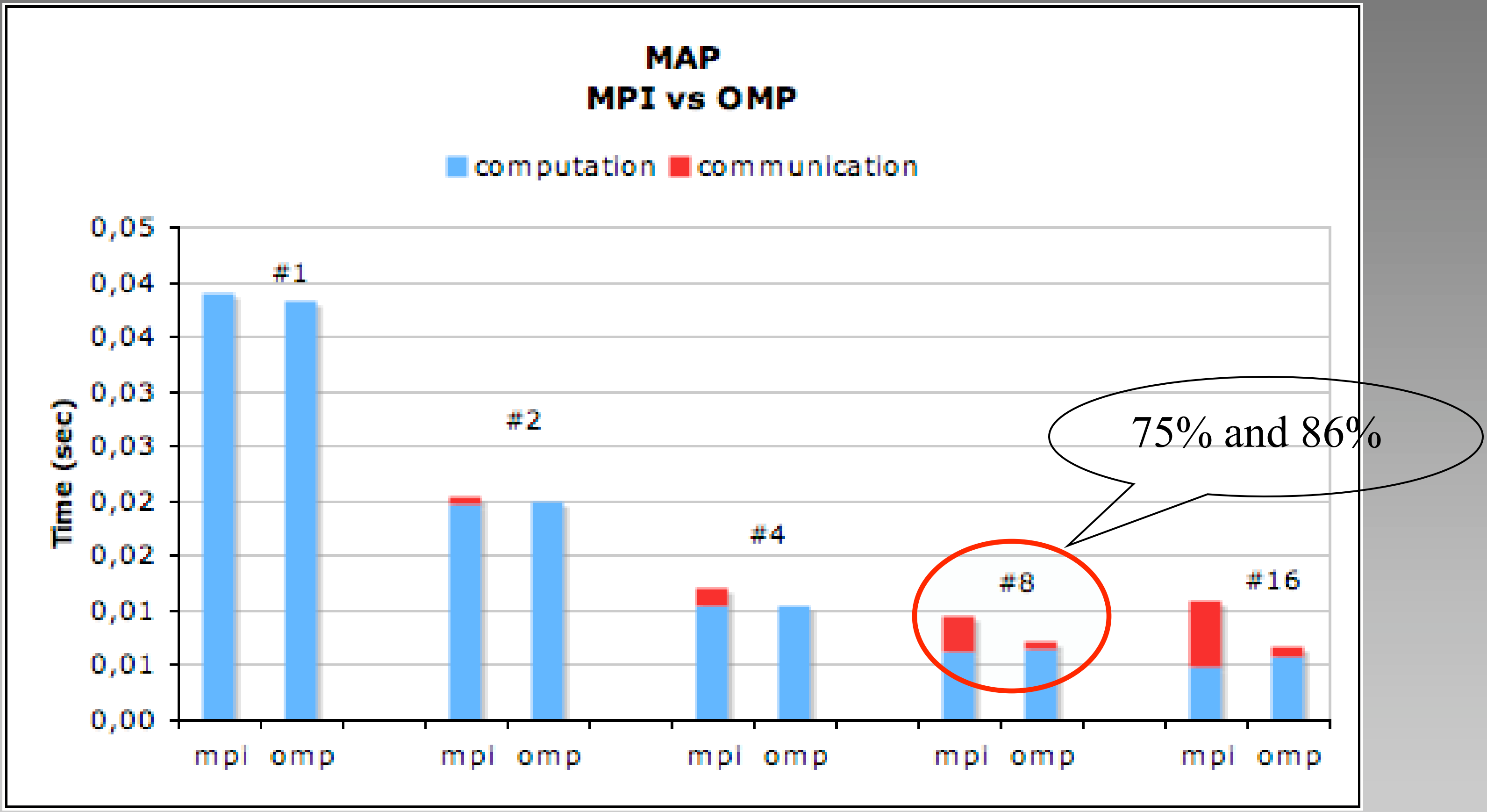
Execution time of the GMV benchmark on the AMD machine with a 8192 x 8192 matrix

# 2- BSP++: MPI vs OpenMP



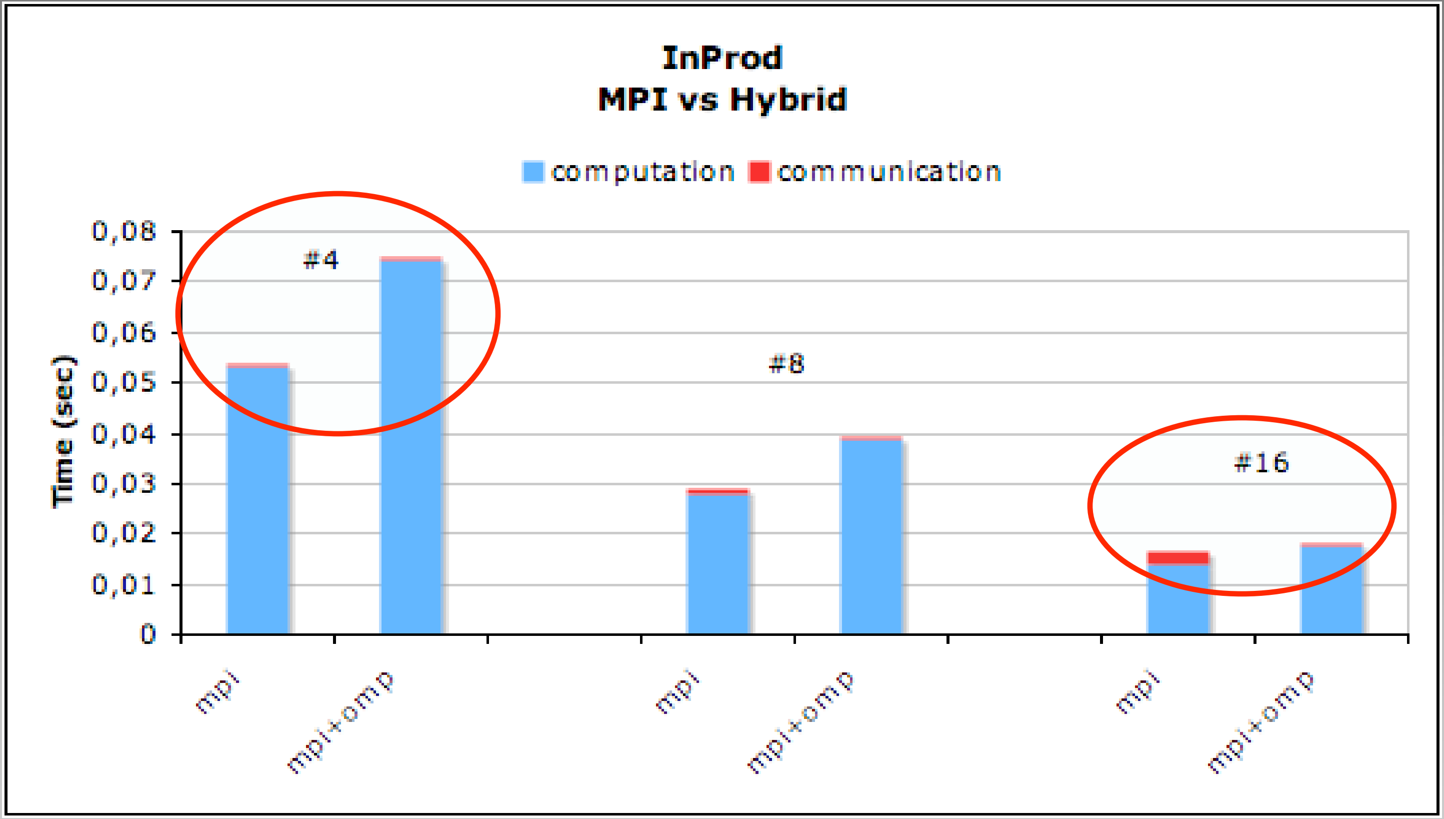
Execution time of the GMM benchmark on the AMD machine with 2048 x 2048 matrices

## 2- BSP++: MPI vs OpenMP



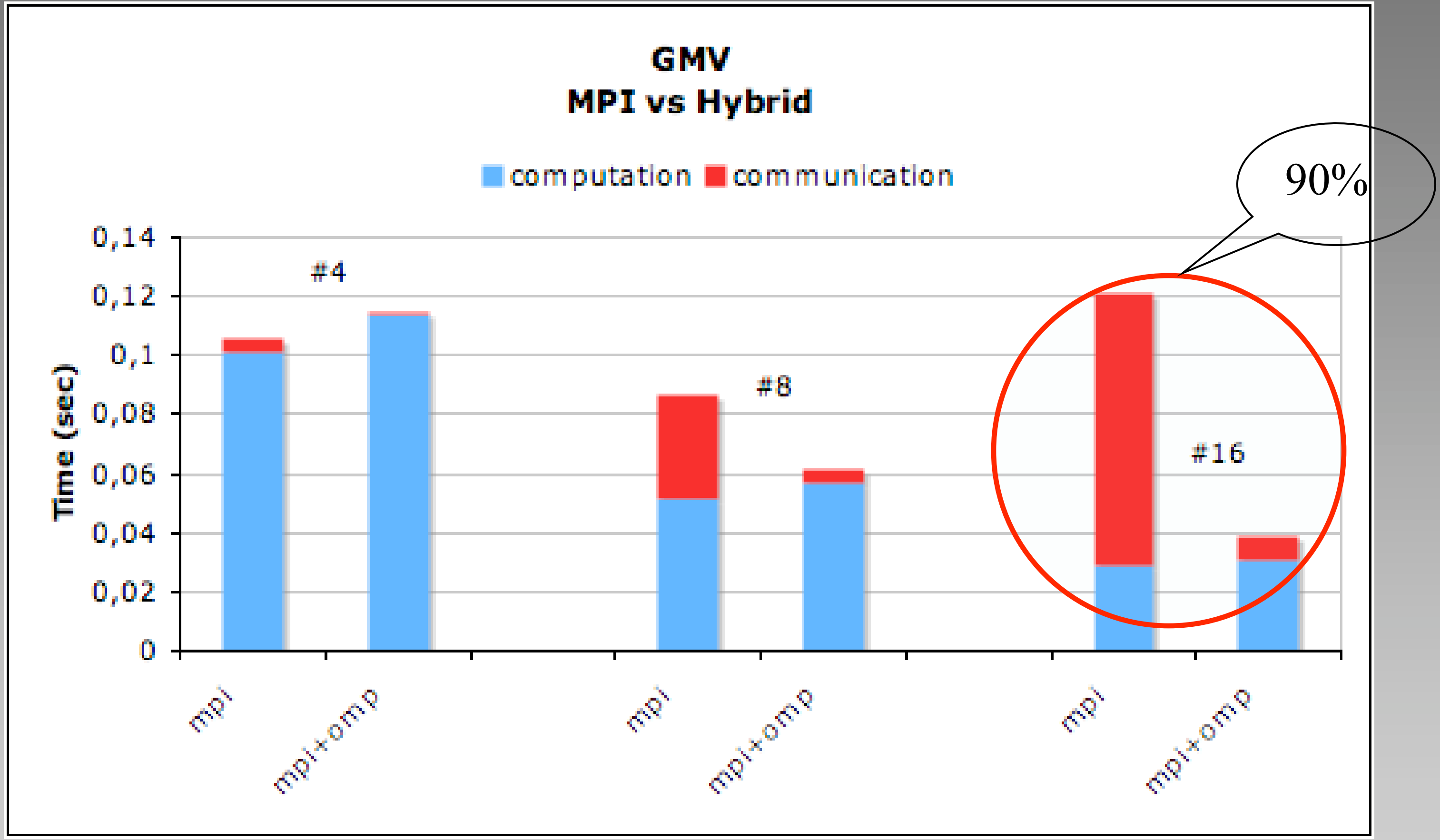
Execution time of the MAP benchmark on the AMD machine for 150000 words list

# 3- BSP++: MPI vs Hybrid



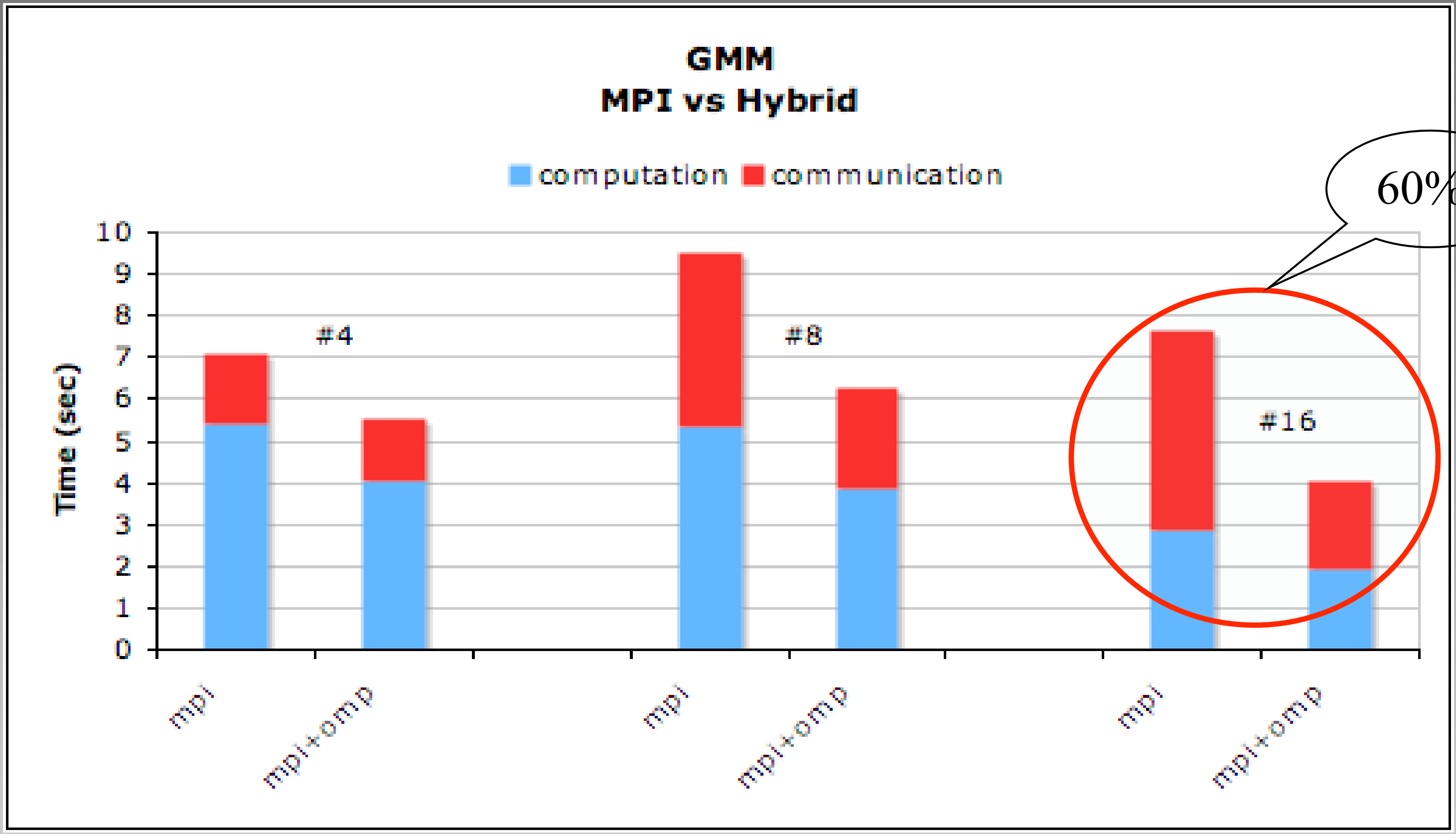
Execution time of the InProd benchmark on the Cluster machine for 64 10^6 elements

# 3- BSP++: MPI vs Hybrid



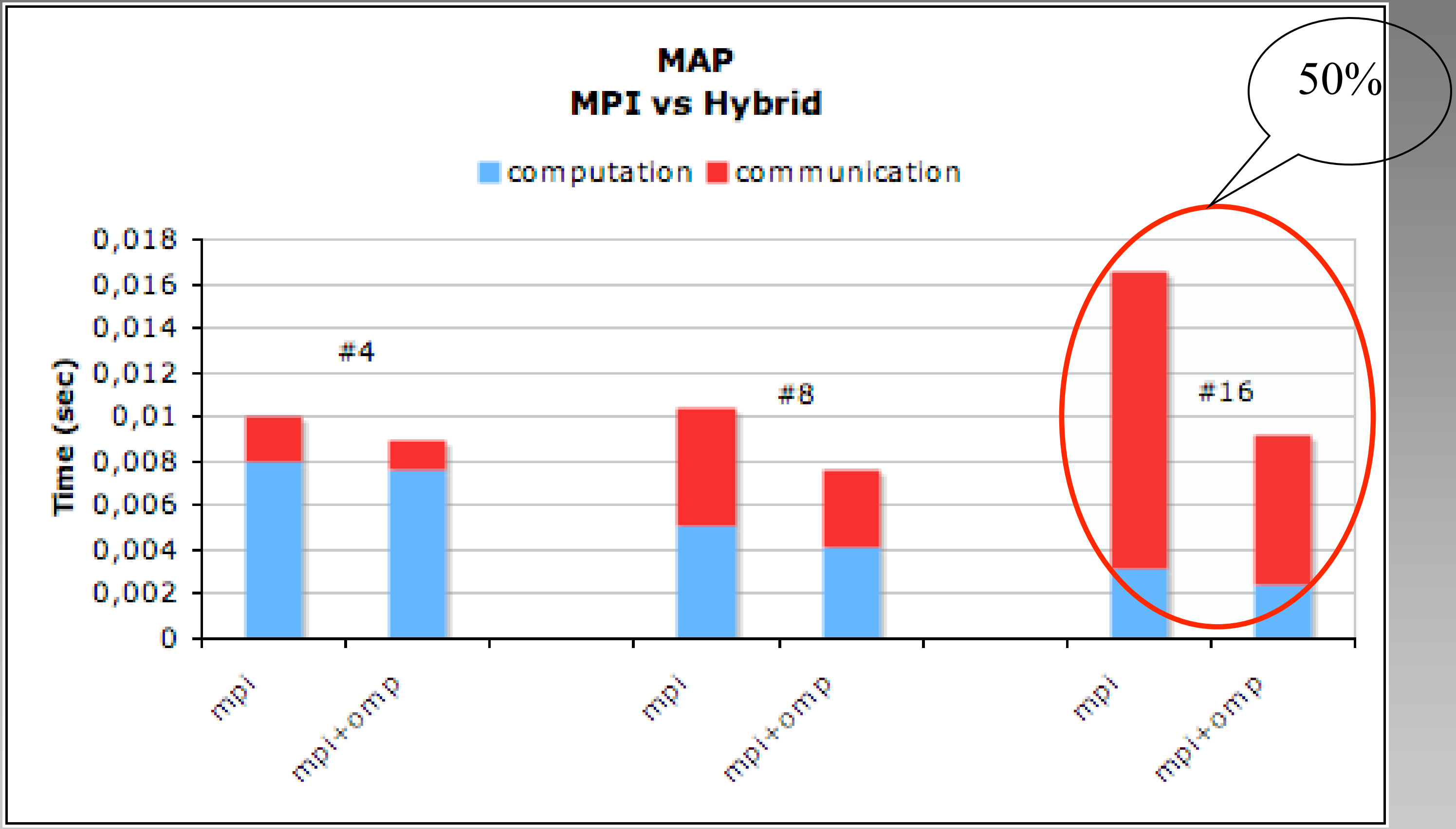
Execution time of the GMV benchmark on the Cluster machine with a 8192 x 8192 matrix

# 3- BSP++: MPI vs Hybrid



Execution time of the GMM benchmark on the Cluster machine with 2048 x 2048 matrices

# 3- BSP++: MPI vs Hybrid



Execution time of the MAP benchmark on the Cluster machine for 150000 words list

# Conclusion

- MPI and OpenMP as a native targets
  - Both versions scale
  - No overhead of the C++ implementation
- Simplify the design of Hybrid MPI+OpenMP codes
  - Using the same code
- Support a large number of practical development idioms

# Framework

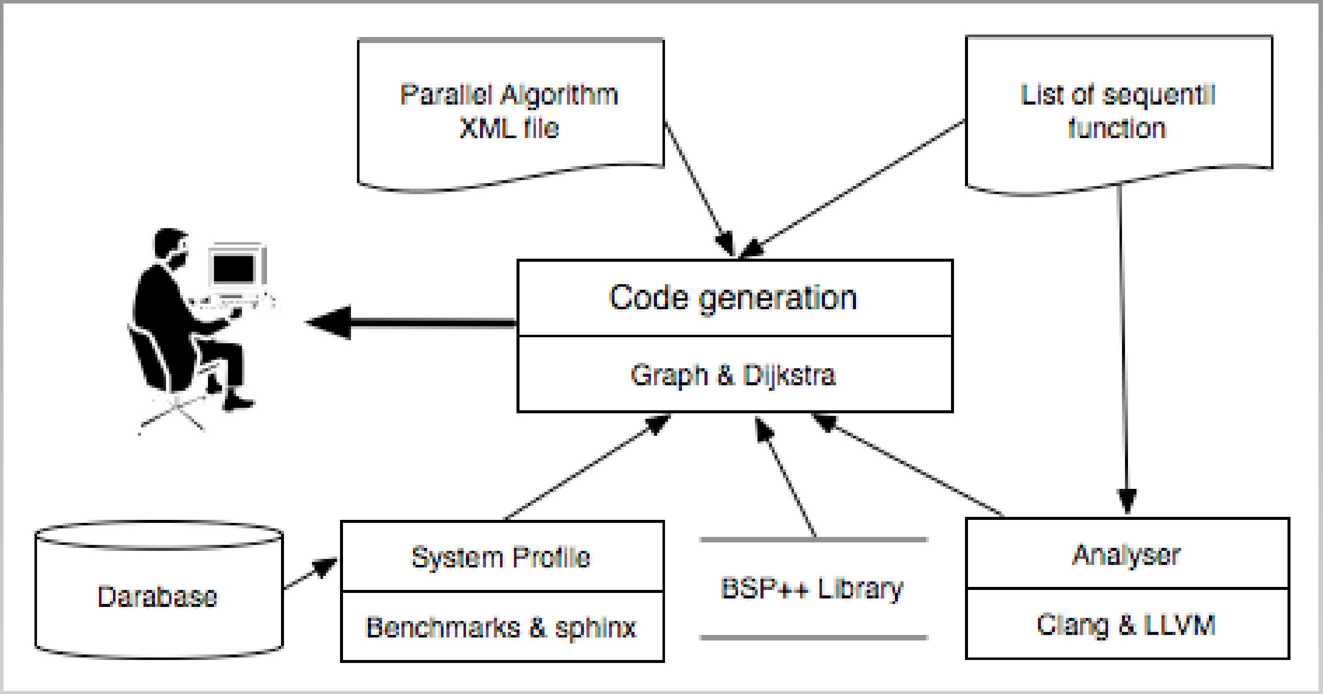
❖ Use the BSP cost model to estimate the execution time of each step

❖ Select the best configuration (number of MPI process and number of OpenMP) for each step

❖ Generate the corresponding code using the BSP++ library

# Framework architecture

❖ Three modules : Analyzer, Searcher, Generator



# Framework modules

❖ Analyzer: estimates the execution time by predicting the values of Tcomp and Tcomm

❑ Computing time : count the number of cycles for the sequential function

Clang: generates the byte-code for the user function

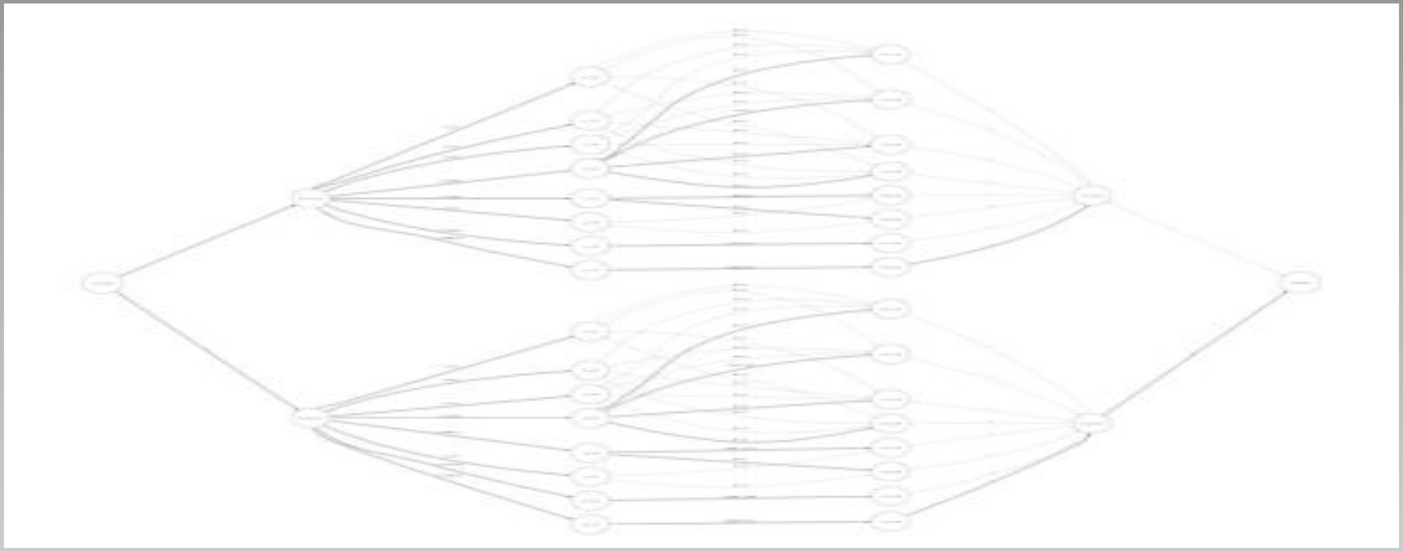
LLVM: New pass in the compiler to count the number of cycles in the byte-code

❑ Communication time : estimate the value of g and L for MPI and OpenMP

Using runtime benchmarks: probe-benchmark and Sphinx tool

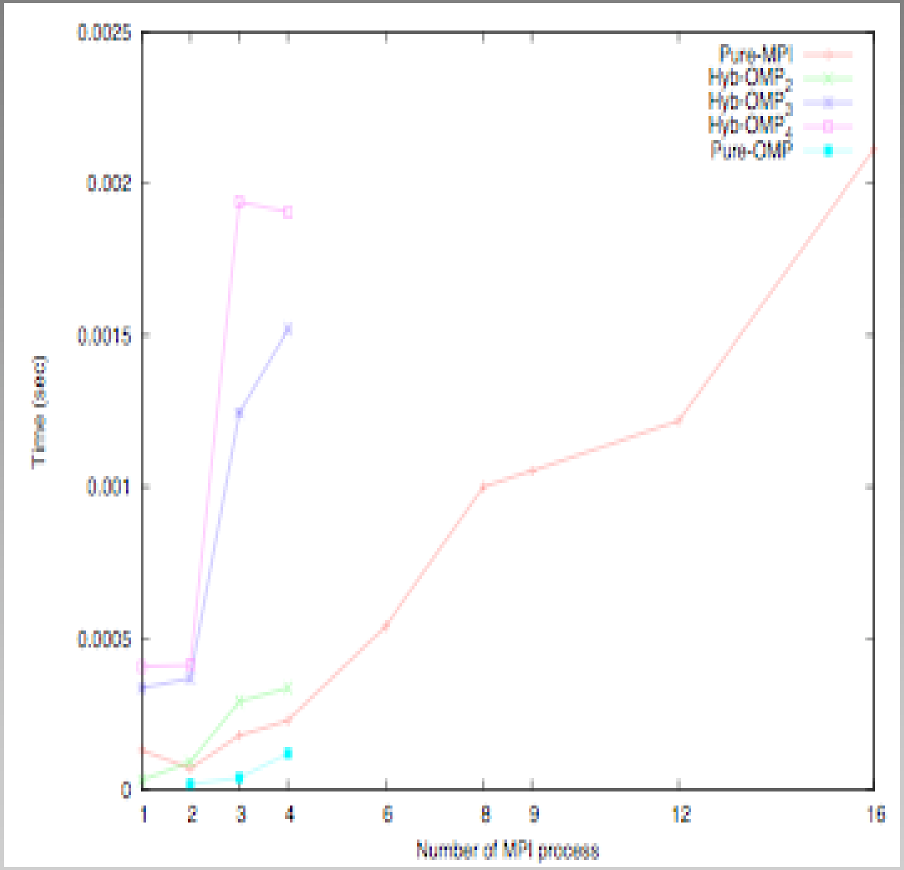
# Framework modules

- Searcher : find the best configurations
  - Build a graph for all valid configuration
  - Use the Dijkstra Short path to find the fastest execution

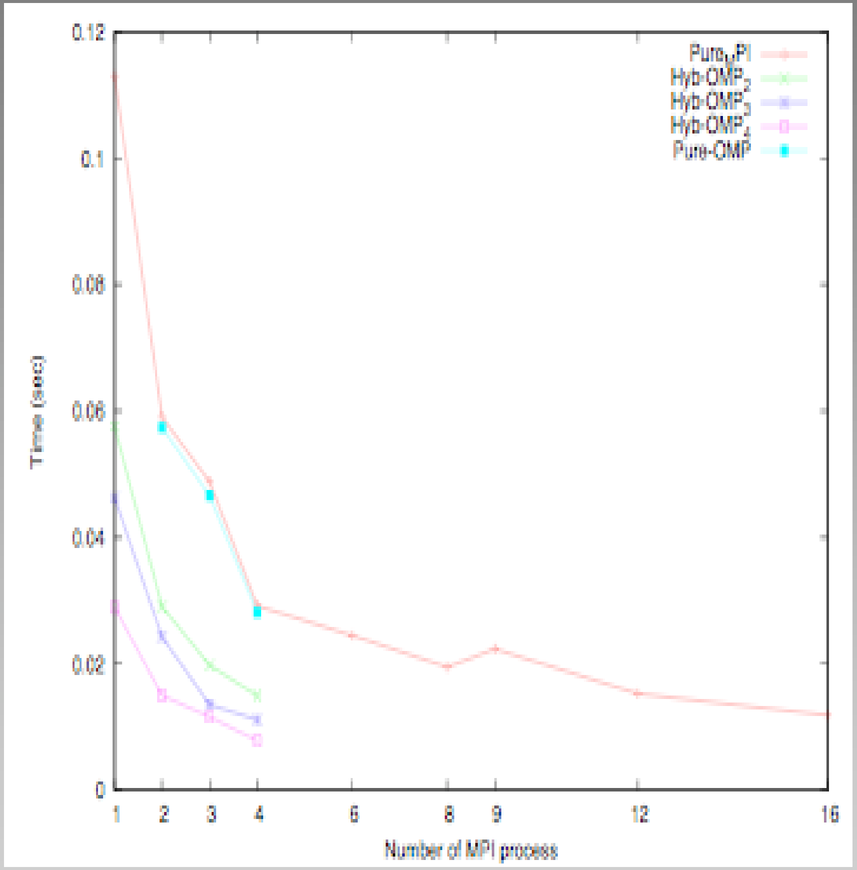


- Generator : generates the corresponding code for each configuration in the shortest path by using the BSP++ library

# Experimental results

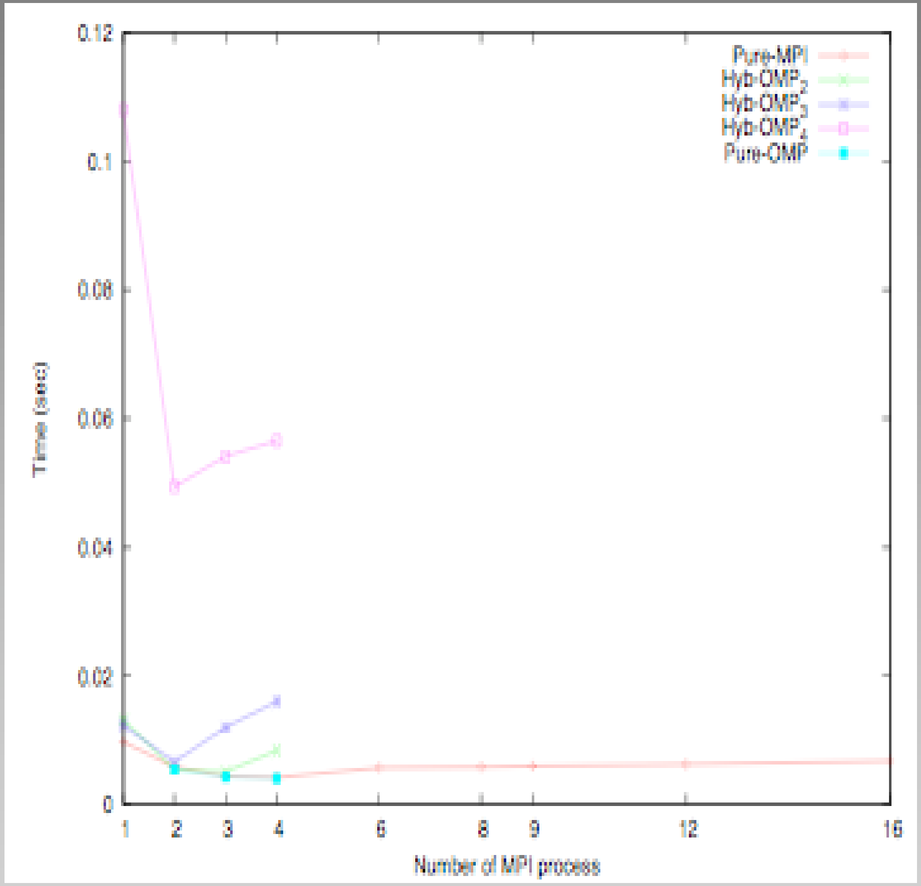


Inner product small size 16K

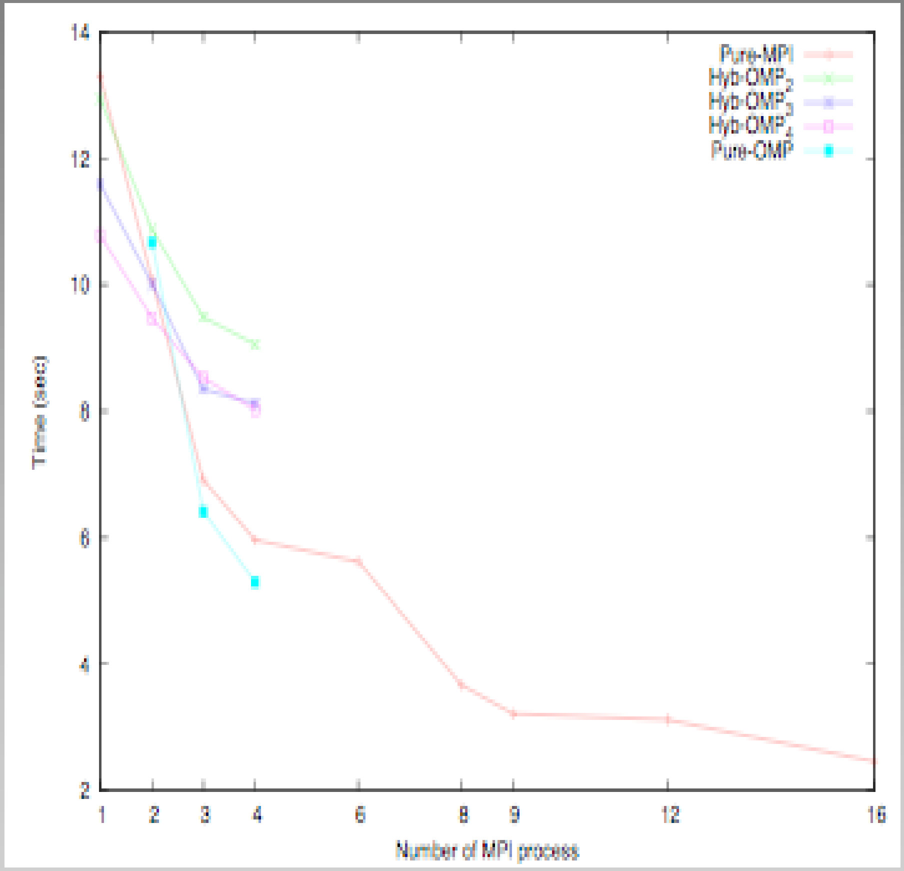


Inner product big size 64M

# Experimental results



PSRS small size (81920 elements)



PSRS big size (8192 x 10^4 elements)

## Future works

- ❖ Implementation of BSP++ on Cell and GPU :

  Hybrid MPI+OpenMP+GPU

- ❖ BSP based containers and algorithms:

  Write a subset of C++ standard library as BSP algorithm

# Hybrid Bulk Synchronous Parallelism Library for Clustered SMP Architectures

Khaled Hamidouche, Joel Falcou, Daniel Etiemble
hamidou, joel.falcou, de@lri.fr
LRI, Université Paris Sud 11
91405 Orsay, France