Automatic Hybrid MPI+OpenMP Code Generation

Khaled Hamidouche, Joel Falcou

05/17/2011

LRI, University Paris Sud XI

Disclaimer

There is actually **no** Boost.Proto in this presentation

The March of Hybrid Parallelism

What's up on the HPC planet ?

- Machines are becoming more and more hybrids
- HPC Top500 : 80% of clusters of multicores
- HPC Top10 : multicores + GPGPUs or Cell Processors
- Most modern desktop computer are small HPC nodes

The March of Hybrid Parallelism

What's up on the HPC planet ?

- Machines are becoming more and more hybrids
- HPC Top500 : 80% of clusters of multicores
- HPC Top10 : multicores + GPGPUs or Cell Processors
- Most modern desktop computer are small HPC nodes

So is the free lunch free again ?

- Difficulties scale changed
- Combining all these new toys become increasingly complex
- Does having more mean it obviously goes faster ?

What's left to do so ?

What happens in the literature ?

What's left to do so ?

What happens in the literature ?

- Performance improvement using MPI and OpenMP

What's left to do so ?

What happens in the literature ?

- Performance improvement using MPI and OpenMP
- Poor performance adding OpenMP to MPI programs

What's left to do so ?

What happens in the literature ?

- Performance improvement using MPI and OpenMP
- Poor performance adding OpenMP to MPI programs

Hybrid programming is a complex problem

- Architecture: network bandwith, number of cores, type of accelerators ...
- Application: Communication computation ratio, problem size ...
- Programming model: MPI, MPI+OpenMP, MPI+CUDA, OpenMP+CUDA, MPI+OpenMP+CUDA, oh my ...

Purpose of this talk

Our Objectives

- Find a way to simplify this mess
- Can we find a decent programming model for this ?

Our Work

- A library for hybrid programming
- A tool to help in configuration exploration
- All using Boost of course

Talk Layout

Programming Tools and Models

Message Passing Interface (MPI)

- Run multiple process across distributed nodes
- Process use **Message** to communicate
- Provides a set of ready-to-use communications primitives

OpenMP

- Standard language extension for shared memory system
- Parallelism is expressed as **parallel sections** using `#pragma`
- Provides functions for threads handling and synchronization

Higher Level Models

What do we need

- Architecture asbtraction
- Performances estimation
- Easy to use for the end user

Higher Level Models

What do we need

- Architecture asbtraction
- Performances estimation
- Easy to use for the end user

What's available ?

- Stream processing
- Parallel Skeletons
- Bulk Synchronous Parallelism

Higher Level Models

What do we need

- Architecture asbtraction
- Performances estimation
- Easy to use for the end user

What's available ?

- Stream processing
- Parallel Skeletons
- Bulk Synchronous Parallelism

Bulk Synchronous Parallelism

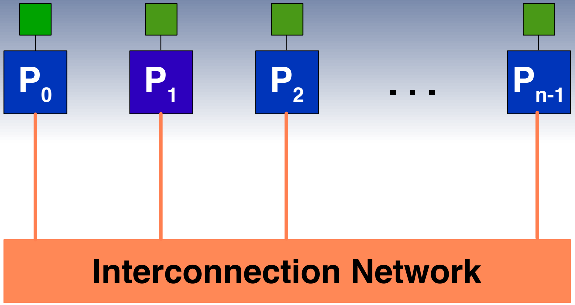
Origin

- Proposed by L. Valiant in 1990
- Present a constrained form of parallelism
- Bridge the gap between machine and programs

Principles

- A Machine Model
- A Cost Model
- A Programming Model

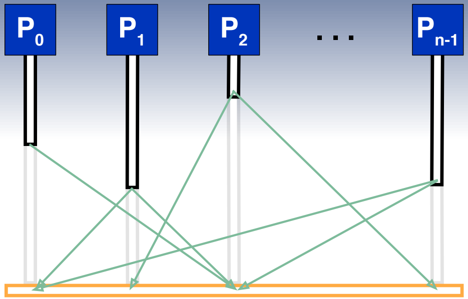
BSP Machine Model



Definition

- Multiple Computing units : local memory + processor
- One all-to-all interconnection network
- A global barrier mechanism

BSP Programming Model



Definition of a Super-Step

- An asynchronous computation step
- A all-to-all communication step
- A global barrier

BSP Cost Model

Definition

- W_i : computation time on processor i
- h : amount of bytes to transfer
- g : network throughput
- L : Time for performing a barrier

Cost of one super-step

$$\Omega = \max W_i + h \times g + L$$

Existing BSP Library

Oxford BSPLib [Hill:96]

- C based
- Rely on low-level shared memory runtime
- Provides 20+ primitives for communications over different medium

BSML [Gava:09]

- Functionnal implementation of BSP in Caml
- Notion of parallel 'vector'
- Two communications + one synchronization primitives
- Provides an extended syntax for BSP construct in ML

Why BSP ?

BSP Pros and Cons

- Straightforward `Seq of Par` programming model
- Hybrid programming support with a black-box approach
- Limited support for task parallelism
- Barrier costs impact programm structure

Our Plans

- Provide a BSP like library for parallel programming
- Provide a tool for BSP application description
- Use BSP cost Model to explore configuration space

Talk Layout

BSML primitives

Distributed Vector

A BSP distributed vector is a vector where each element lives on a different BSP node

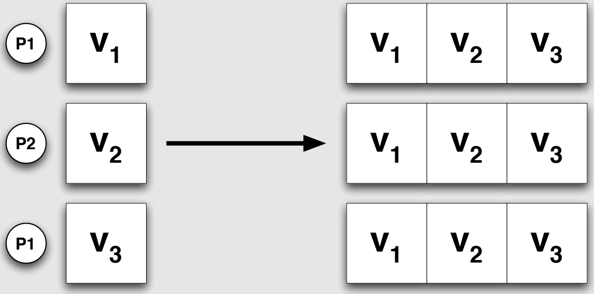
- `«v»` : build a vector from value or a function *v*
- `$v$` : access to the local vector element
- A parallel vector of type $'a$ has type $par'a$

BSML primitives

The `proj` function

- Replicates a parallel vector around all BSP nodes
- Prototype: $proj : par\ 'a- > par\ (int- >'\ a)$

Semantic of *proj v*

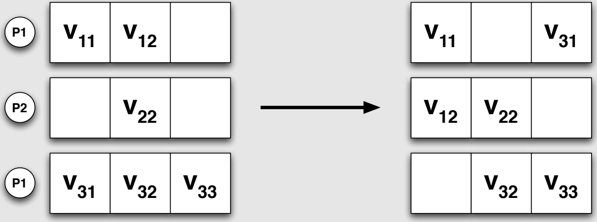


BSML primitives

The `put` function

- Generic all-to-all communications function
- Prototype: $put : par\ (int -> {}'a) -> par\ (int -> {}'a)$

Semantic of *put vf*



A sample BSML Code

BSML Inner Product

```
let inner_product v =
  let local = << Array.fold_left (+.) (Array.map2 (*.) $v$ $v$) >>
    in let gathered = proj local
      in Array.fold_left (+.) (Array.make gathered nprocs) ;;
```

A sample BSML Code

BSML Inner Product

```
let inner_product v =
  let local = << Array.fold_left (+.) (Array.map2 (*.) $v$ $v$) >>
    in let gathered = proj local
      in Array.fold_left (+.) (Array.make gathered nprocs) ;;
```

How does it works

- Build a distributed vector from $v[i]^2$ in parallel
- Exchange partial results with all nodes
- Perform a final reduction

From BSML to BSP++

Why looking at BSML

- Provides a compact and abstract interface
- BSML likes playing with lambda and so do we

The Plan

- Implement BSML interface and abstraction ic C++
- Try to work on the functionnal side to limit errors
- Try to play nice with C++ functionnal idioms

BSP++ 101

Main Program Structure

- Managed main handles parallel runtime
- Everything in a BSP programm is parallel

BSP++ 101

Main Program Structure

- Managed main handles parallel runtime
- Everything in a BSP programm is parallel

Example

```
#include <bsppp/bsppp.hpp>

int bsp_main( int argc, char const* argv[] )
{
  // Starting from here, everythign is parallel

}
```

BSP++ primitives

Parallel vector : `par<T>`

- `par<T>` is a BSP distributed `T`
- Constructible from values, functions and ranges

BSP++ primitives

Parallel vector : `par<T>`

- `par<T>` is a BSP distributed `T`
- Constructible from values, functions and ranges

`par<T>` Interface

```
// distributed default construction
par<T> p;

// distributed replication
T v;
par<T> p = v;

// distributed initialization from a Callable Object
T foo(std::size_t pid);
par<T> p = foo;
```

BSP++ primitives

Parallel vector : par<T>

- par<T> is a BSP distributed T
- Constructible from values, functions and ranges

par<T> Interface

```
// Access to local value
par<T> p;

T x = *p;

// Envelope behavior
par< vector<T> > p;
p->resize(n);
```

BSP++ primitives

The `proj` and `put` function

- BSML returns function **value**
- Let's return **Callable Object** embedding the result
- Make them Range for easier interoperability

BSP++ primitives

The `proj` and `put` function

- BSML returns function **value**
- Let's return **Callable Object** embedding the result
- Make them Range for easier interoperability

Examples

```
par< float > r = 1.f / _1;
result_of::proj<float> exch = proj (r);

// Value at machine 1
cout << exch(1) << endl;

// Iterate over value receive from all machines
std::for_each( exch.begin(), exch.end(), ref(cout) << _1 );
```

BSP++ primitives

The `proj` and `put` function

- BSML returns function **value**
- Let's return **Callable Object** embedding the result
- Make them Range for easier interoperability

Examples

```
par< float > r = 1.f / _1;

auto inv = put(  [&r](int dst) { if(dst % 2) return *r; else return -*r; }  );

// Value at machine 1
cout << (*inv)(1) << endl;
```

A sample BSP++ code

BSP++ Inner Product

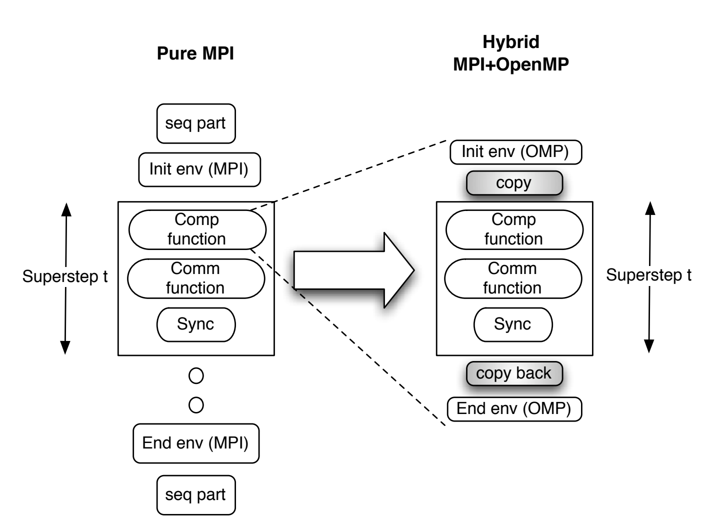
```
template<class Range>
typename iterator_value<typename Range::const_iterator>::type
inner_product( Range const& input )
{
  typedef typename
          iterator_value<typename Range::const_iterator>::type value_type;

  par<Range> v = slice( input );
  par< value_type > r;

  *r = std::inner_product( v->begin(), v->end(), v->begin(), value_type() );

  result_of::proj<value_type> exch = proj (r);
  *r = std::accumulate(exch.begin(), exch.end() );
}
```

Support for Hybrid programming



Support for Hybrid programming

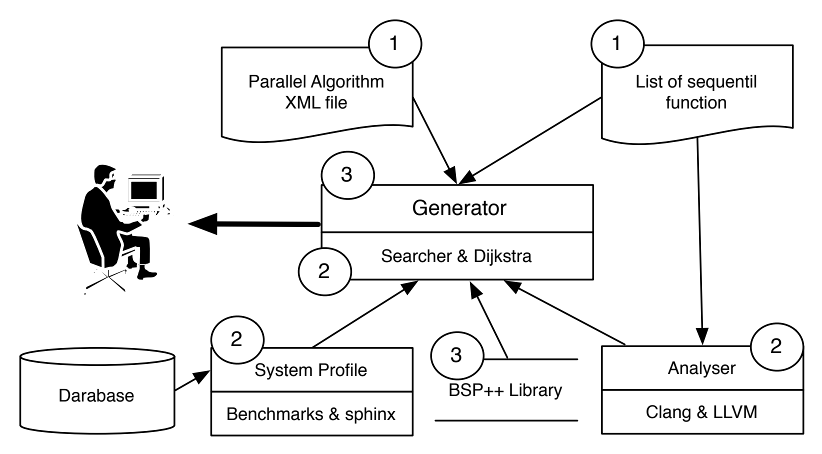
BSP++ Hybrid Inner Product

```
template<class Range>
typename iterator_value<typename Range::const_iterator>::type
inner_product_omp( Range const& input )
{
  typedef typename
          iterator_value<typename Range::const_iterator>::type value_type;
  BSP_HYB_START(argc, argv)
  {
    par<Range> v = slice( input );
    par< value_type > r;
    *r = std::inner_product( v->begin(), v->end(), v->begin(), value_type() );
     result_of::proj<value_type> exch = proj (r);
    *r = std::accumulate(exch.begin(), exch.end() );
  }
}

template<class Range>
typename iterator_value<typename Range::const_iterator>::type
inner_product( Range const& input )
{
  typedef typename
          iterator_value<typename Range::const_iterator>::type value_type;

  par<Range> v = slice( input );
  par< value_type > r;
  *r = inner_product_omp( v );
  result_of::proj<value_type> exch = proj (r);
  *r = std::accumulate(exch.begin(), exch.end() );
}
```

The BSPGen Framework



Analysis and Exploration

Analysis

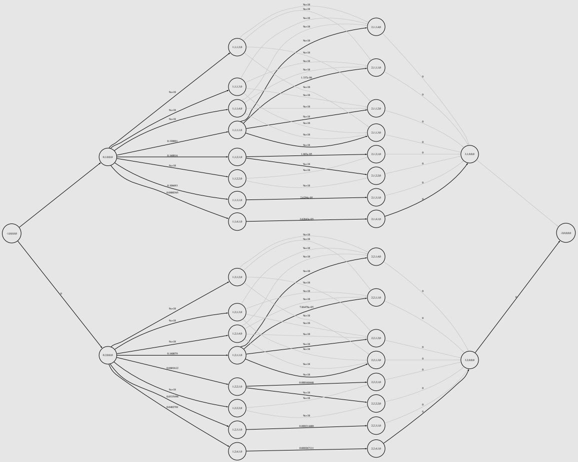
- Compile each sequential function using LLVM/Clang
- Parse results to find out an estimation of runtime costs
- Estimate communication from offline benchmarks

Configuration exploration

- Buy a directed graph of the sequence of super-steps
- Compute all combination of node/core configurations
- Weights edge with estimated runtime cost
- Run a simple Shortest Path algorithm

Analysis and Exploration

Configuration exploration



Talk Layout

Objectives

Coverage

- Simple kernels
- Three applications

Test machines

- a 4x4 cores NUMA machine using AMD processors
- 256 nodes from the French GRID'5000 infra-structure
- a 3 Cell processors cluster

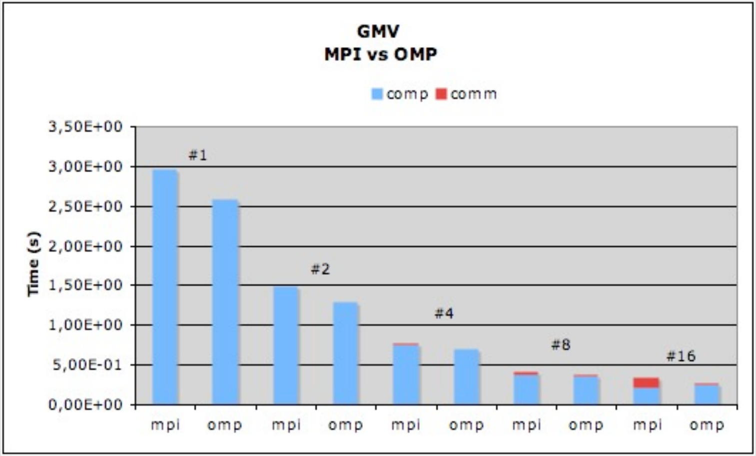
Simple Kernels

Chosen Kernels

- Matrix-Vector product (GEMV)
- MapReduce

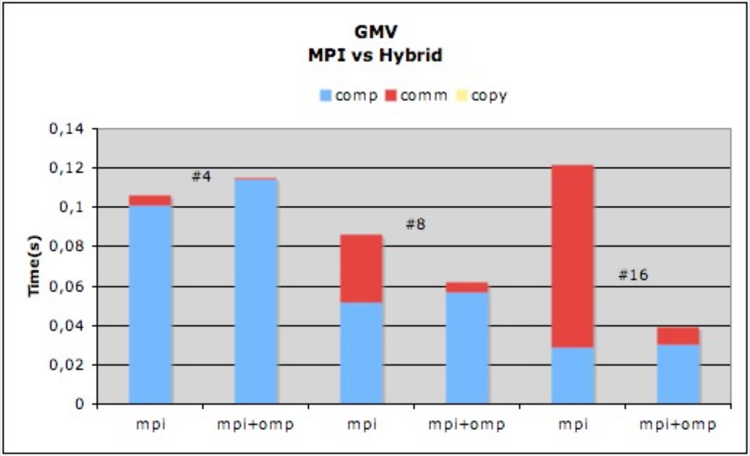
Simple Kernels

Results



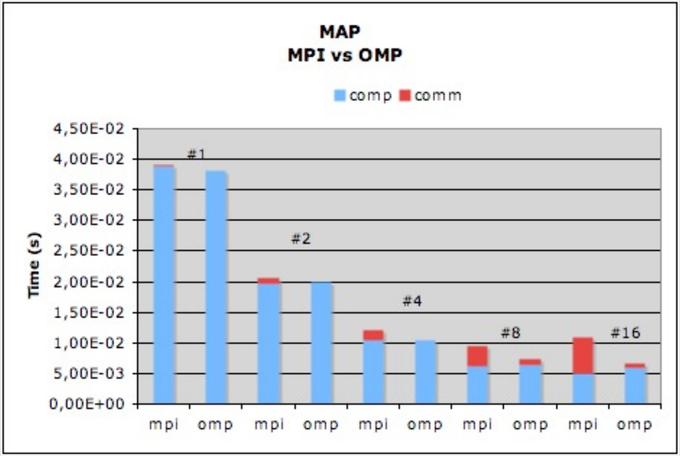
Simple Kernels

Results



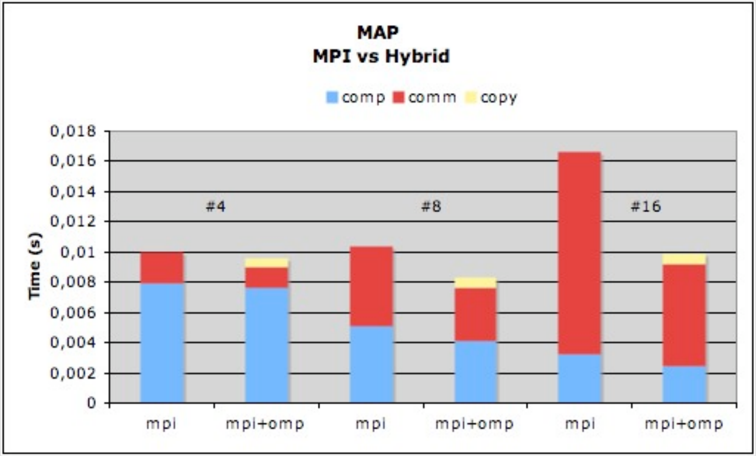
Simple Kernels

Results



Simple Kernels

Results



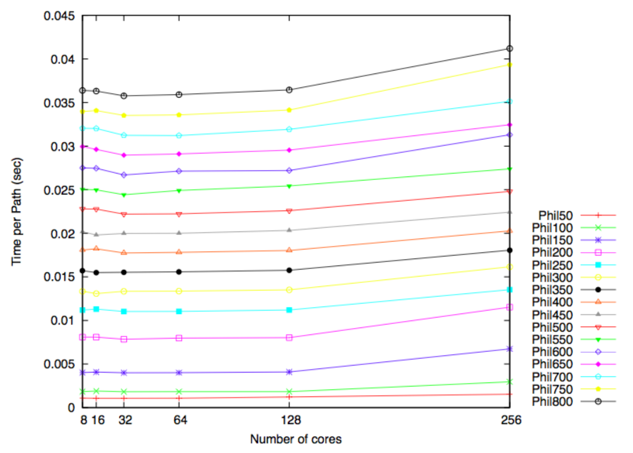
Model Checking

Parallel Approximate Model Checking [Peyronnet:08]

- Complex systems need verification
- Turn system into a set of condition driven states
- Try to solve time-logic predicates over the model
- Large problem, can be solved approximately (APMC)

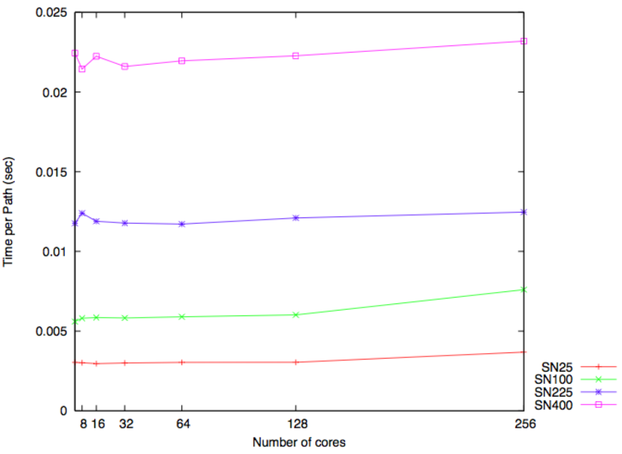
Model Checking

Results



Model Checking

Results



Model Checking

Results

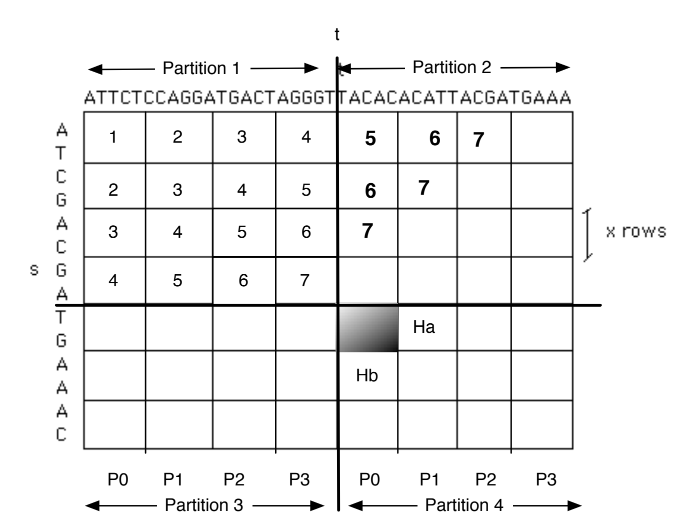
- Great scalability over more than 200 cores
- Parallel APMC allows for larger problem to be verified
- See [Hamidouche PDMC 2010] for more

Swith and Waterman DNA Comparisons

Principles

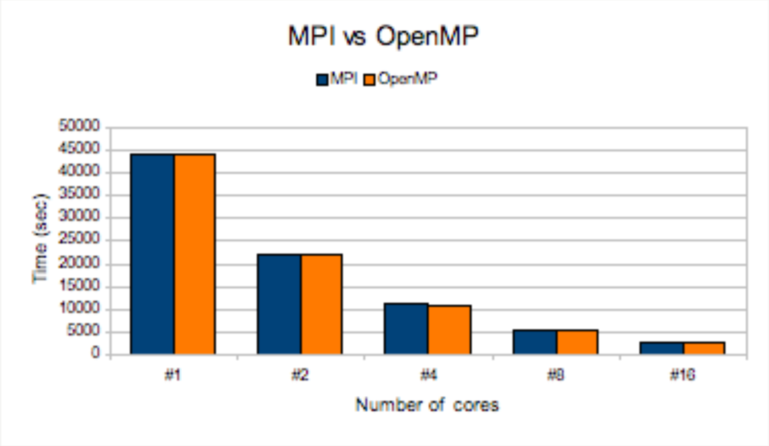
- Compute distance between two DNA sequences
- Heuristic method : BLAST fast but not accurate
- Direct method : S&W accurate but slow

Swith and Waterman DNA Comparisons



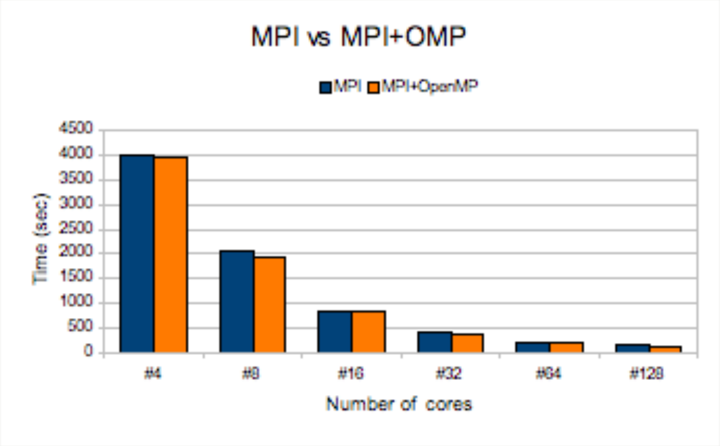
Swith and Waterman DNA Comparisons

Results - 1 MBases comparisons



Swith and Waterman DNA Comparisons

Results - 1 MBases comparisons



Talk Layout

Contributions

BSP++

- Implement BSP in an efficient, C++ way
- Supports black-box hybridation
- Show scalability and usability
- Play with it: https://github.com/jfalcou/bsppp

BSPGen

- Ease the configuration exploration of BSP programs
- Interoperability between Boost and clang
- To be extended

Future Works

New Architectures

- Cell Processor : done with Cell-MPI
- GPGPU: require multistage programming

More BSP with Phoenix 3

- Functionnal version of BSP
- Allow for automatic merging of super-step
- Solve the multistage problem
- Can we force people to write lambda everywhere ?

Thanks for your attention