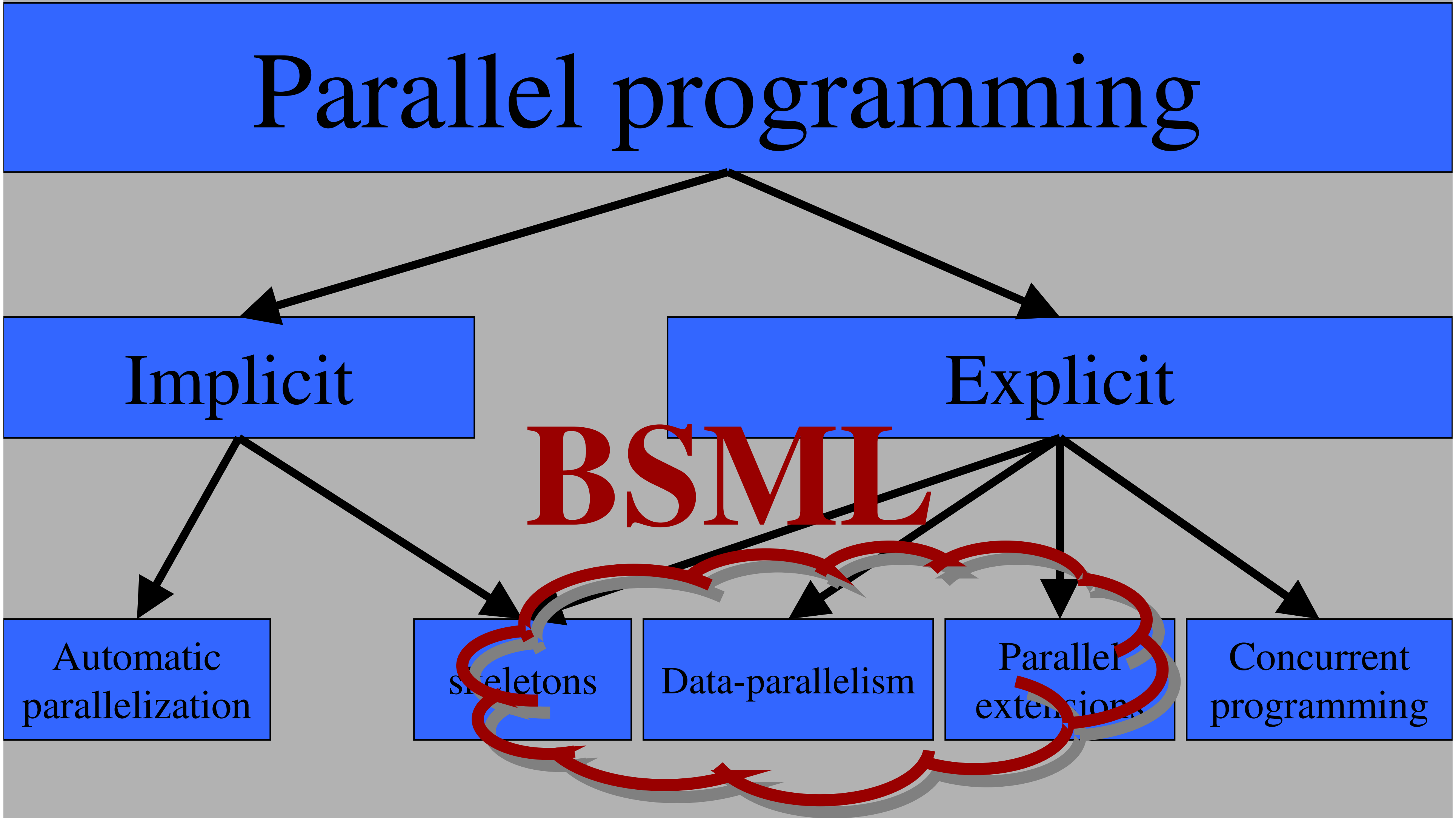
Frédéric Gava

Bulk-Synchronous Parallel ML

Examples of a high-level parallel language and a cost based methodology

UNIVERSITÉ PARIS-EST
lacl

Background



Parallel programming
Implicit
Explicit
BSML
Automatic parallelization
Data-parallelism
Parallel
Concurrent programming

Outline

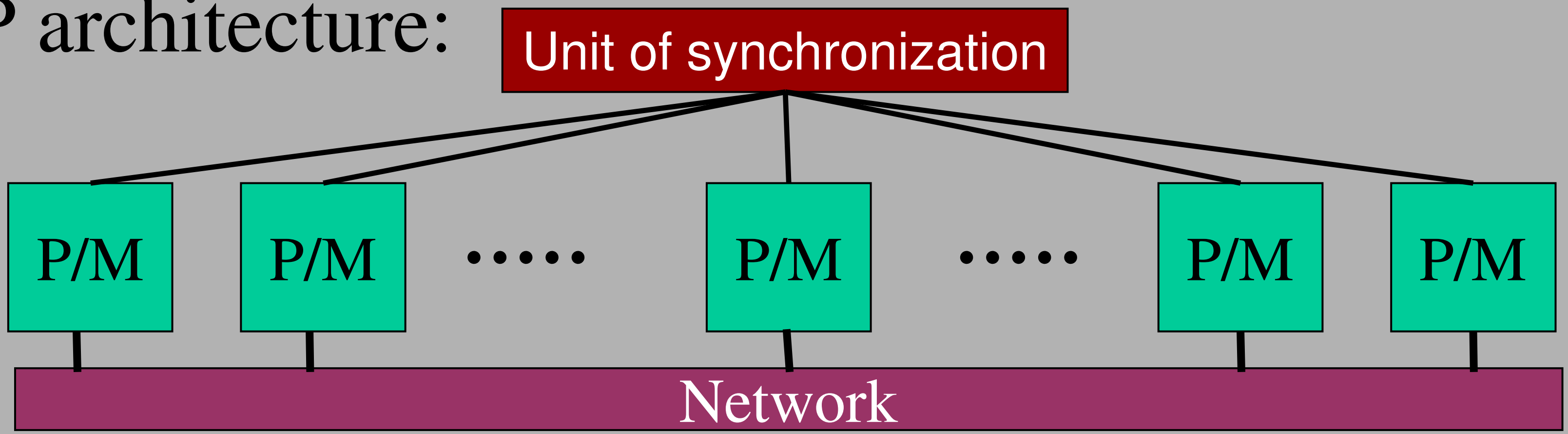
The BSML language

The BSML « spirite »

- *Bugs grow faster than Moore's law.* (G. Berry)
  - High-level language ⇨↘ lines of code ⇨↘ number of bugs
  - Certified library ⇨↘ number of bugs
- *Small is beautiful.* (R. H. Bisseling)
  - BSML only use 5 primitives…
- *Who would drive a non-deterministic car ?* (G. Berry)
  - Propriety of confluence of the semantic of BSML
- *French Proverb : « All the roads go to Roma » But the better way is to choose the shorter*
  - One can give BSP costs to BSML programs
  - Different of concurrent programming : cost and confluence

The BSP model

BSP architecture:



Characterized by:

- **p** **Number** of processors
- **r** Processors **speed**
- **L** **Global** synchronization
- **g** **Phase of communication** (1 word at most sent of received by each processor)

Model of execution

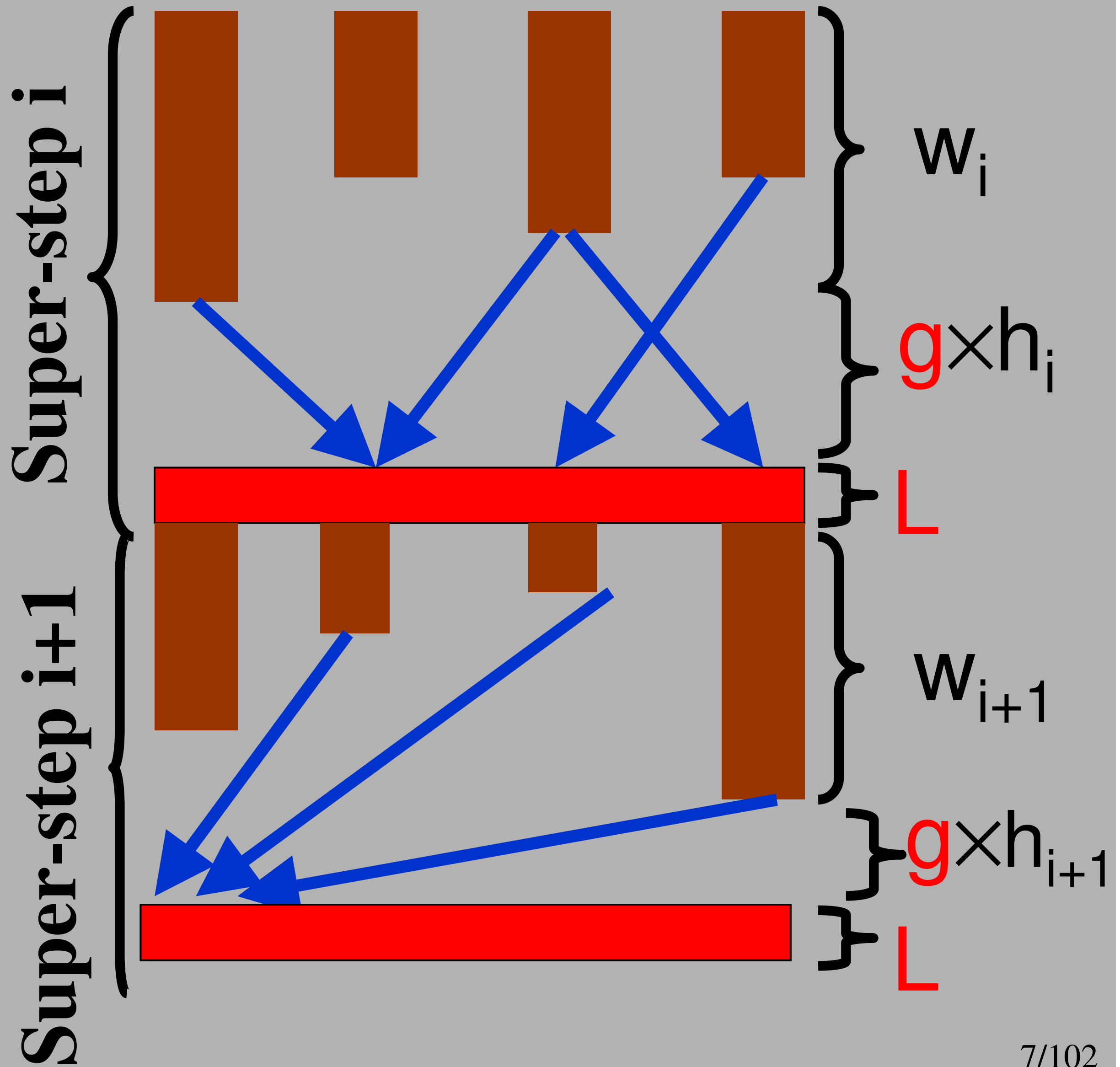
Beginning of the super-step i

Local computing on each processor

Global (collective) communications between processors

Global synchronization : exchanged data available for the next super-step

$$\text{Cost}(i) = (\max_{0 \le x < \mathbf{p}} w^x_i) + h_i \times \mathbf{g} + \mathbf{L}$$

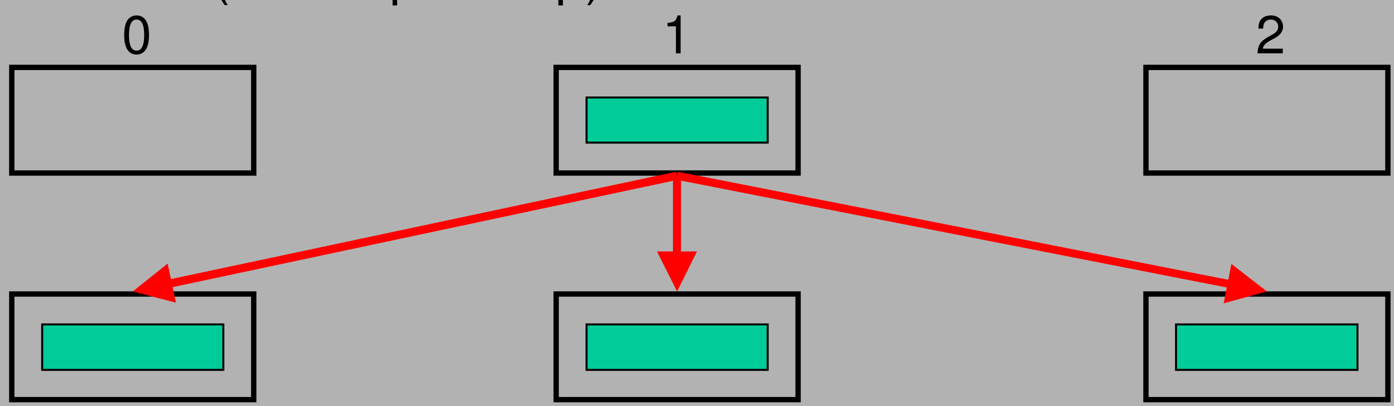


A libertarian model

- No master :
  - Homogeneous power of the nodes
  - Global (collective) decision procedure instead
- No god :
  - Confluence (no divine intervention)
  - Cost predictable
  - Scalable performances
- Practiced but confined

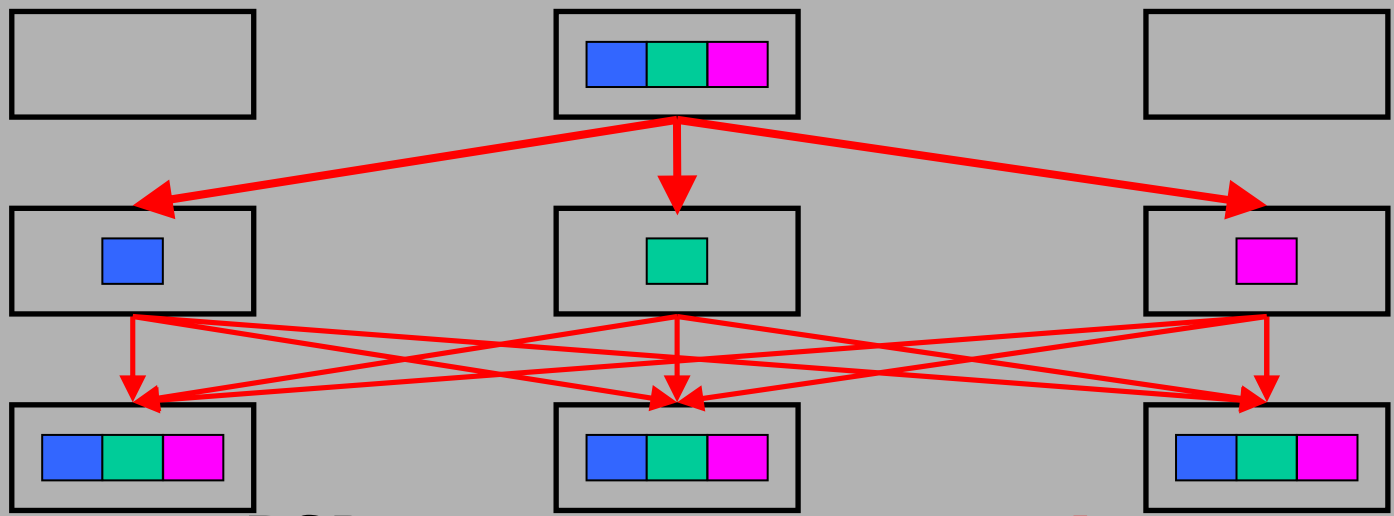
Example : broadcast

- Direct **broadcast** (one super-step):



BSP cost = $p \times n \times g + L$

- **Broadcast** with 2 super-steps:

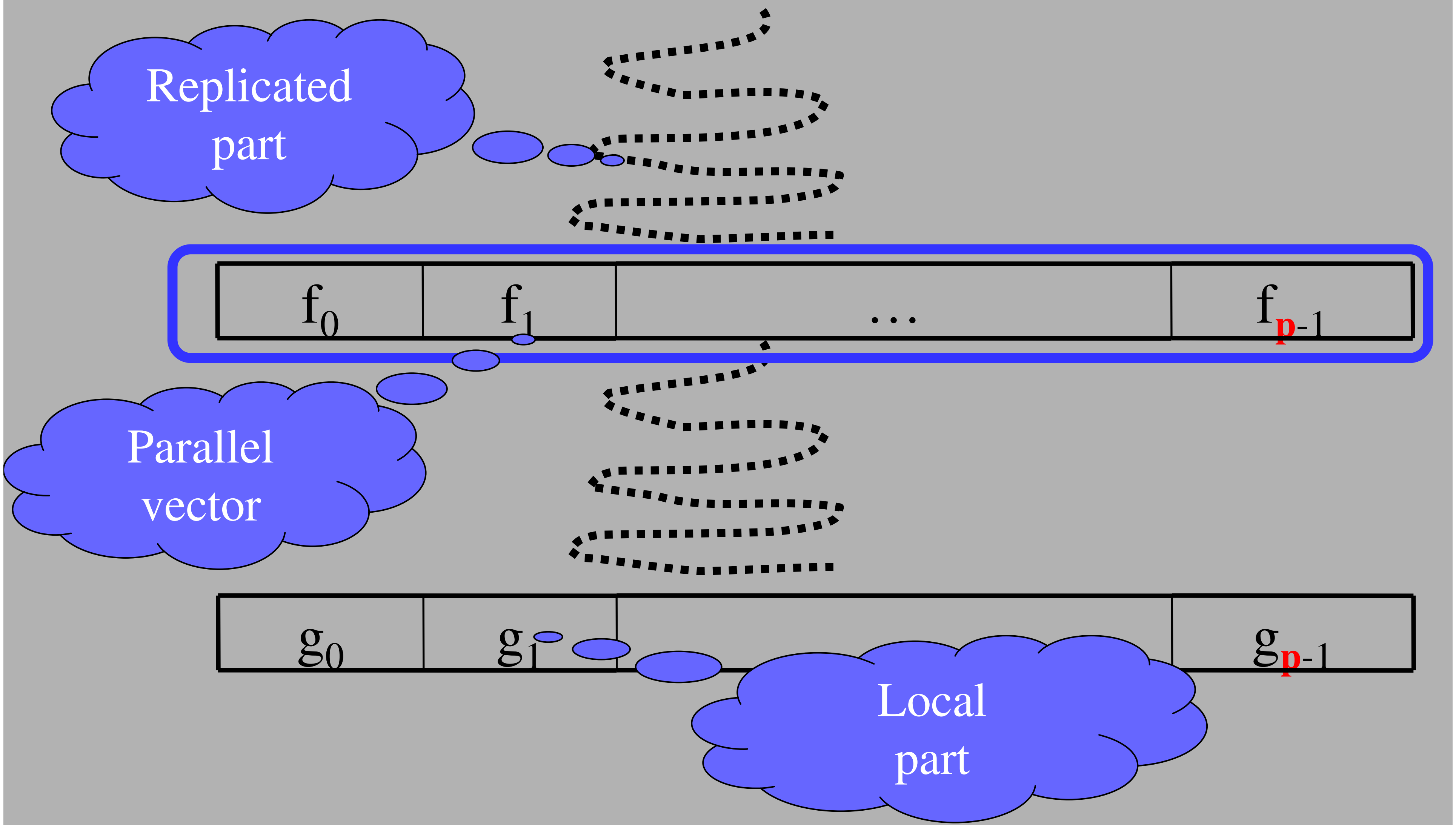


BSP cost = $2 \times n \times g + 2 \times L$

The BSML language

- **Structured parallelism** as an explicit parallel extension of the High level (functional) language **ML**
- **BSP cost** predictions
- Implemented as a parallel library for the **"Objective Caml" language**
- Using a parallel data structure called **parallel vector**
- Using **5 parallel primitives :**
  - Outside vector : classical O'Caml code with calls to the parallel primitives
  - Inside vector : classical O'Caml code

A BSML program



Parallel primitives of BSML

- Asynchronous primitives:
  - Creation of a vector (creation of local values)

    mkpar : (int $\rightarrow$ α) $\rightarrow$ α par
  - Parallel point-wize application

    apply : (α $\rightarrow$ β) par $\rightarrow$ α par $\rightarrow$ β par
- Synchronous and communications primitives:
  - Communications

    put : (int$\rightarrow\alpha$) par $\rightarrow$ (int$\rightarrow\alpha$) par
  - Projection of local values (to be replicated)

    proj : α par $\rightarrow$ (int$\rightarrow\alpha$)

Primitives asynchrones

- **mkpar** : (int $\rightarrow \alpha$) $\rightarrow \alpha$ par

(mkpar f)

(f 0)	(f 1)	…	f (p-1)

- **apply** : ($\alpha \rightarrow \beta$) par $\rightarrow \alpha$ par $\rightarrow \beta$ par

apply

f_0	f_1	…	f_{p-1}
v_0	v_1	…	v_{p-1}

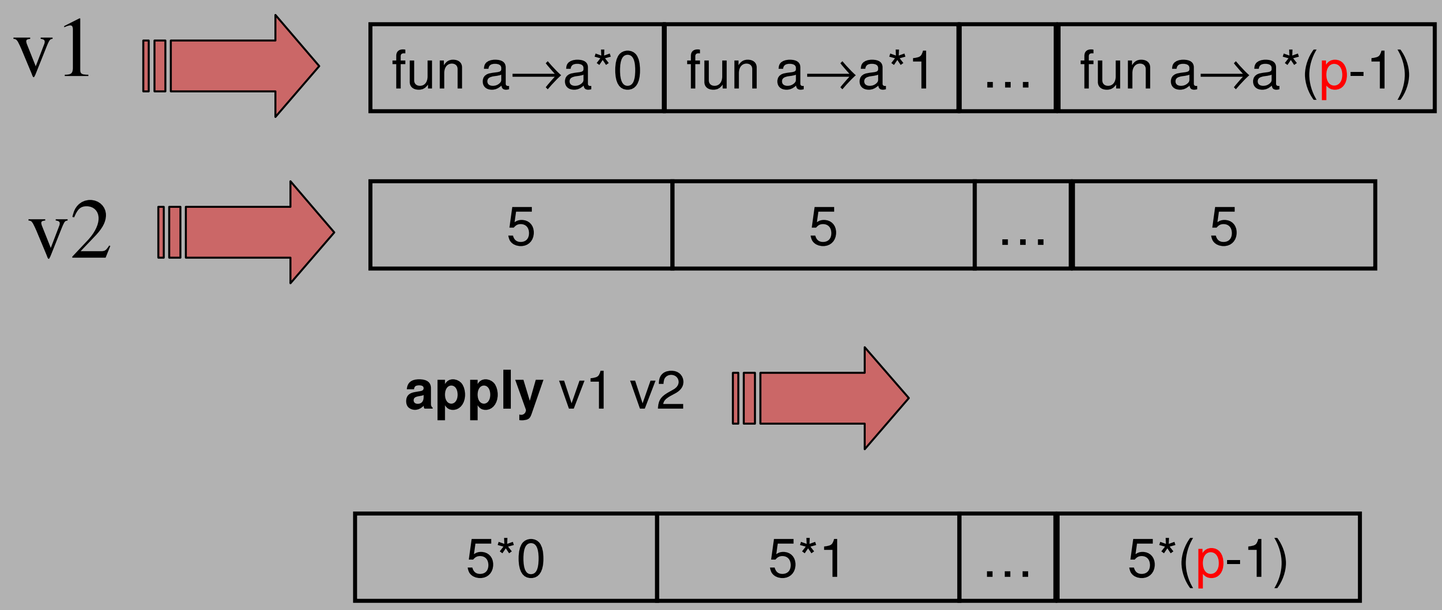
$f_0\ v_0$	$f_1\ v_1$	…	$f_{p-1}\ v_{p-1}$

Example

let v1 = **mkpar** (**fun** pid a→a*pid)

and v2 = **mkpar** (**fun** _→5)

in apply v1 v2



Usefull Functions

- Simple computations :

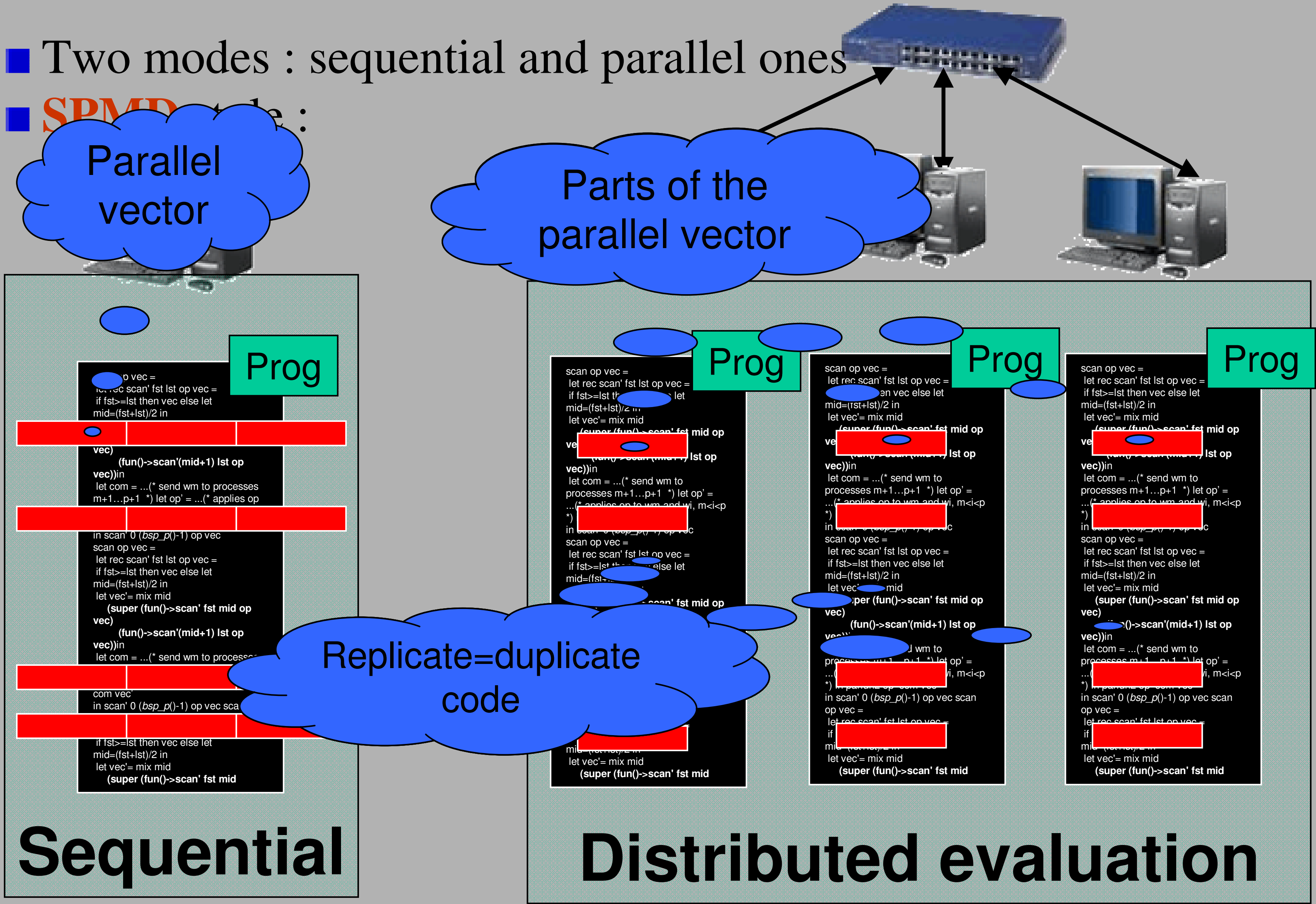
```
(* val replicate : α → α par *)
let replicate x = mkpar (fun _ → x)

(* val apply2 : (α → β → γ) par → α par → β par → γ par *)
let apply2 f v1 v2 = apply (apply f v1) v2

(* val parfun : (α → β) → α par → β par *)
let parfun f v = apply (replicate f) v
```

Execution of BSML programs

- Two modes : sequential and parallel ones
- **SPMD** [illegible]le :



Primitives synchrones

- **put** : (int→α) par→(int→α) par

put

0	1	2	3
None	Some $\mathbf{v}_2$	None	None
None	None	Some $\mathbf{v}_5$	None
None	Some $\mathbf{v}_3$	None	None
Some $\mathbf{v}_1$	Some $\mathbf{v}_4$	None	None

0	1	2	3
None	None	None	Some $\mathbf{v}_1$
Some $\mathbf{v}_2$	None	Some $\mathbf{v}_3$	Some $\mathbf{v}_4$
None	Some $\mathbf{v}_5$	None	None
None	None	None	None

- **proj** : α par→(int→α)

proj

v_0	v_1	…	v_{p-1}

f

such that (f i)=v_i

Projection

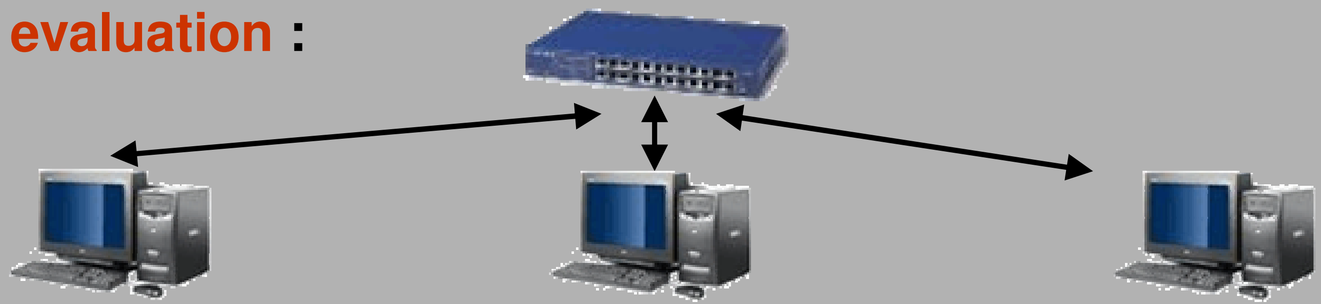
Sequential :

proj

v_0	v_1	…	v_{p-1}

f such that $(f\ i)=v_i$

Distributed evaluation :



proj

v_0	v_1	v_2

f f f

Usefull functions

- Patterns of communication :

```
(* val replicate_total_exchange: α par →α list *)
let replicate_total_exchange vec = List.map (proj vec) (list_procs())

(* val bcast_direct : int →α par →α par *)
let bcast_direct root vv =
  let mkmsg = applyat root (fun v dst →Some v) (fun _ dst →None) vv
   in parfun noSome (apply (put mkmsg) (replicate root))
```

Parallel composition

- Multi-programming
- **Several programs** on the same machine
- Primitive of **parallel composition**: Superposition
- **Divide-and-conquer** BSP algorithms

Parallel Superposition

- **super :** (unit $\rightarrow \alpha$) $\rightarrow$ (unit $\rightarrow \beta$) $\rightarrow \alpha \times \beta$

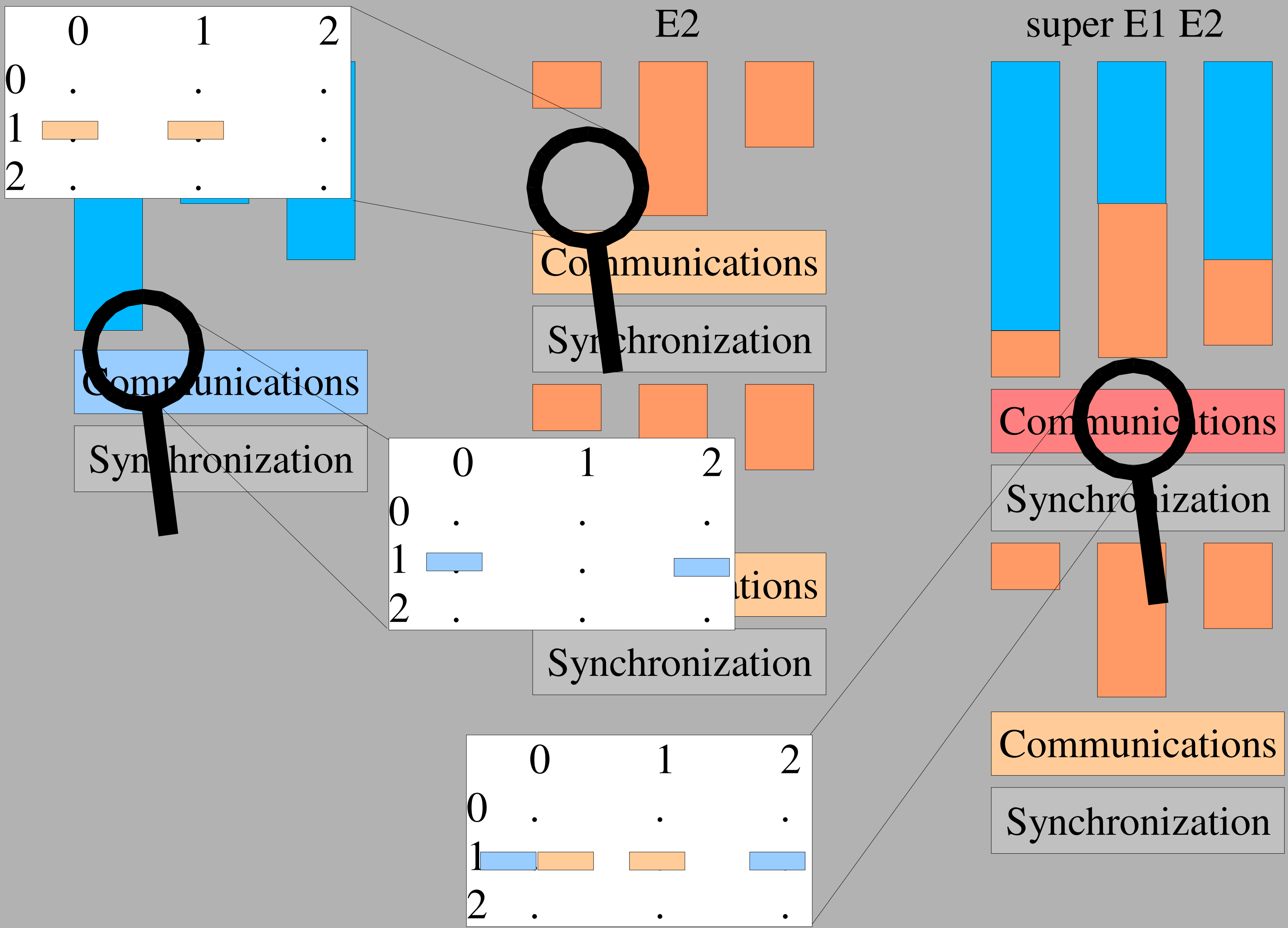
  super E_1 E_2 ➔ (E_1 (), E_2()))

- **Fusion** of communications/synchronisations using **super-threads**

- **Keep** the BSP model

- **Pure functional** semantics

Parallel Superposition



Example, prefixes calculus

scan : $\alpha \rightarrow (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha$ par $\rightarrow \alpha$ par

$$\texttt{scan e (+)} <v_0, \ldots, v_{p-1}>$$

$$= <e, v_0+v_1, \ldots, v_0+v_1+\ldots+ v_{p-1}>$$

Code in BSML :

```
let scan_direct op e vv =
 let mkmsg pid v dst=if dst<pid then None else Some v in
 let procs_lists= mkpar (fun pid →from_to 0 pid) in
 let rcv_msgs= put (apply (mkpar mkmsg) vv) in
 let values= parfun2 List.map (parfun (compose noSome) rcv_msgs) procs_lists in
  applyat 0 (fun _ →e) (List.fold_left op e) values
```

Parallel prefixes

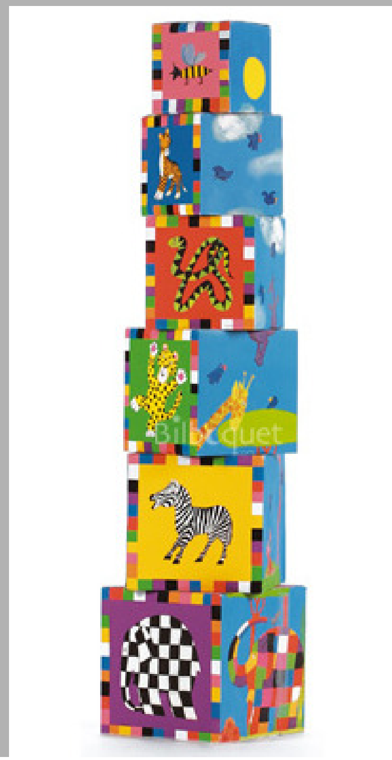
- If we suppose associative operator (+)
  - a+(b+c)=(a+b)+c or better

On a processor

  - $a+(b+(c+d))=\boxed{(a+b)}+\boxed{(c+d)}$

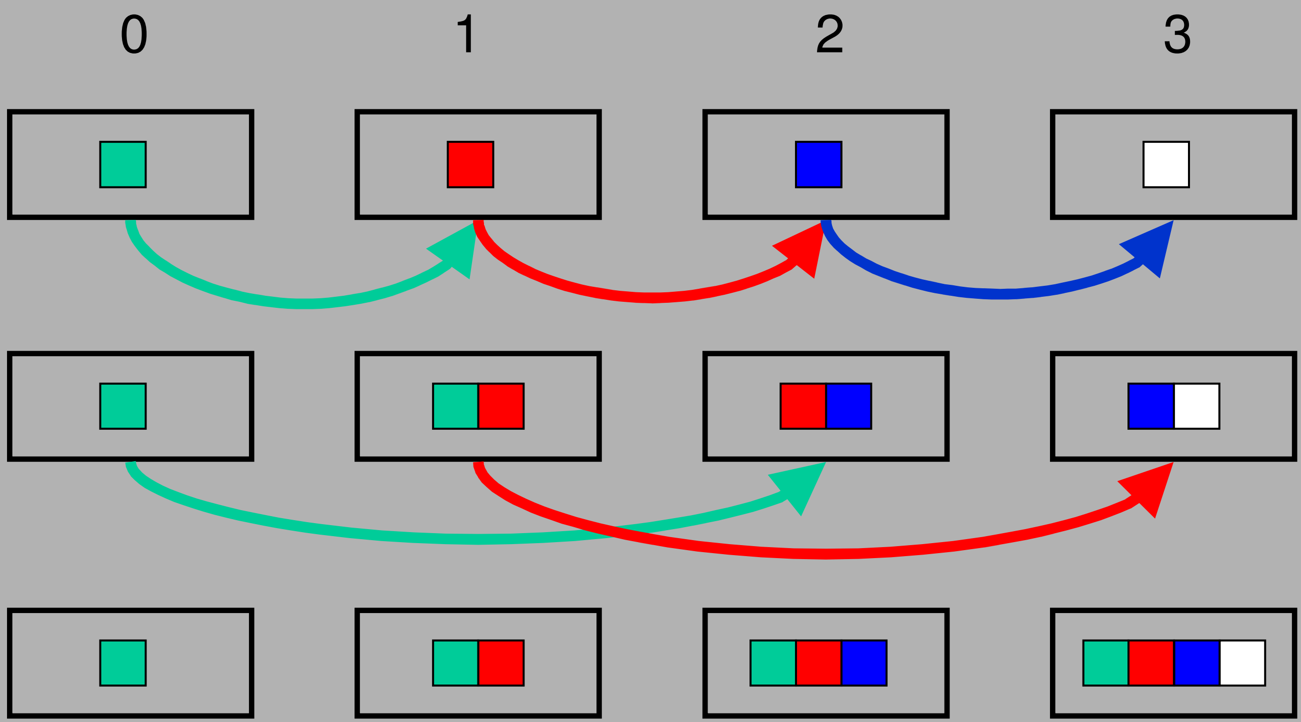
On another processor

- Example :



Parallel Prefixes

- Classical log(p) super-steps method :



Cost = log(p) × (Time(op)+Size(d)×**g**+**L**)

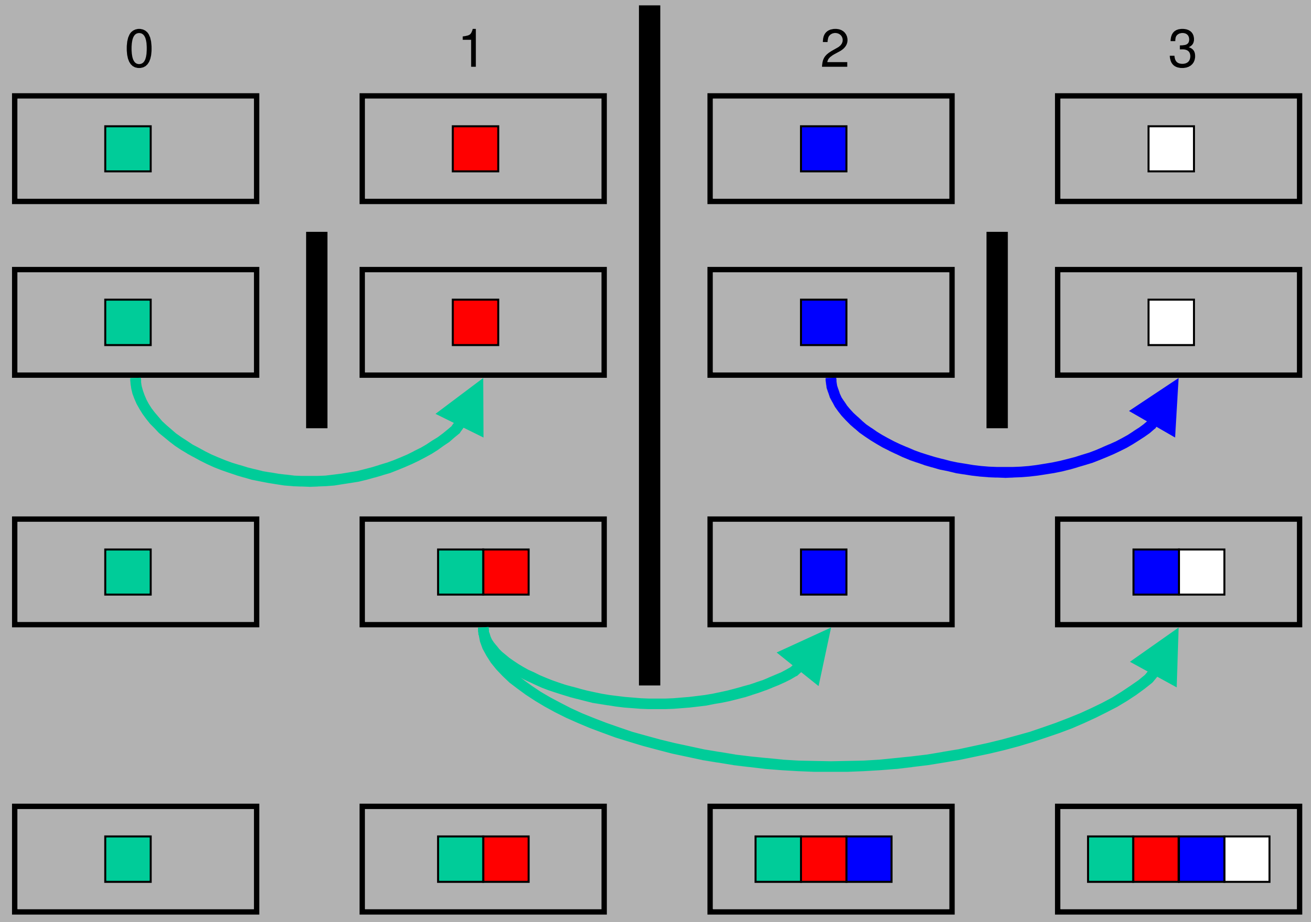
Parallel Prefixes

- BSML code of this method :

```
let scan_logp op e vec =
 let rec scan_aux n vec =
  if n >= (bsp_p()) then (applyat 0 (fun _ →e) (fun x →x) vec) else
   let msg = mkpar (fun pid v dst→
     if ((dst=pid+n)or(pid mod (2∗n)=0))&&(within_bounds (dst−n))
      then Some v else None)
   and senders = mkpar (fun pid→natmod (pid−n) (bsp_p()))
   and op' = fun x y→match y with Some y'→op y' x | None →x in
    let vec' = apply (put (apply msg vec)) senders in
    let vec''= parfun2 op' vec vec' in
     scan_aux (n∗2) vec'' in
 scan_aux 1 vec
```

Parallel Prefixes

- Divide-and-conquer method :



Parallel Prefixes

- BSML code of this method :

```
let scan_super op e vec =
 let rec scan' fst lst op vec =
  if fst>=lst then vec
   else
    let mid = (fst+lst)/2 in
    let vec'= super_mix mid (super (fun()→scan' fst mid op vec)
                                   (fun()→scan'(mid+1) lst op vec)) in
    let msg vec = apply (mkpar(fun i v→
      if i=mid
       then (fun dst→if inbounds (mid+1) lst dst then Some v else None)
      else (fun dst→ None))) vec
    and parop = parfun2(fun x y→match x with None→y|Some v→op v y) in
    parop (apply(put(msg vec'))(mkpar(fun i→mid))) vec' in
  applyat 0 (fun _ →e) (fun x→x) (scan' 0 (bsp_p()−1) op vec)
```

A cost based methology

- BSML is a safe high-order parallel language
- BSP model allows cost analysis of programs
- Methodology :

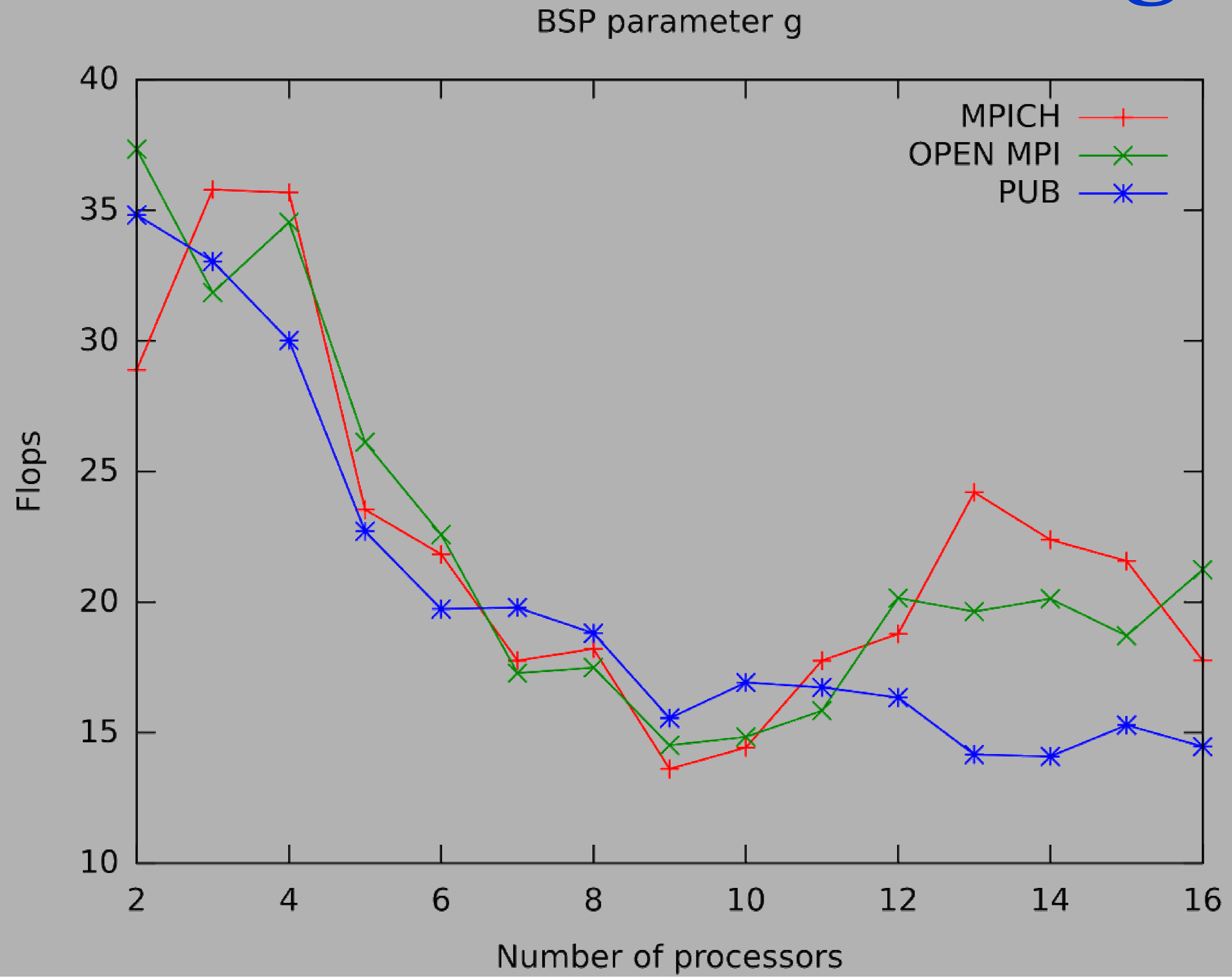
1) Program your sequential algorithm in ML
2) Program one or more parallel algorithms in ML
3) Choose the best follow your BSP parameters ; depending of
    1. Number of processor
    2. Architecture of your network, nodes, etc.
    3. Library of communication (MPI, TCP/IP in O'CAML, PUB, etc.)

- We need (easy to write using different h-relations) to bench our BSP parameters in BSML

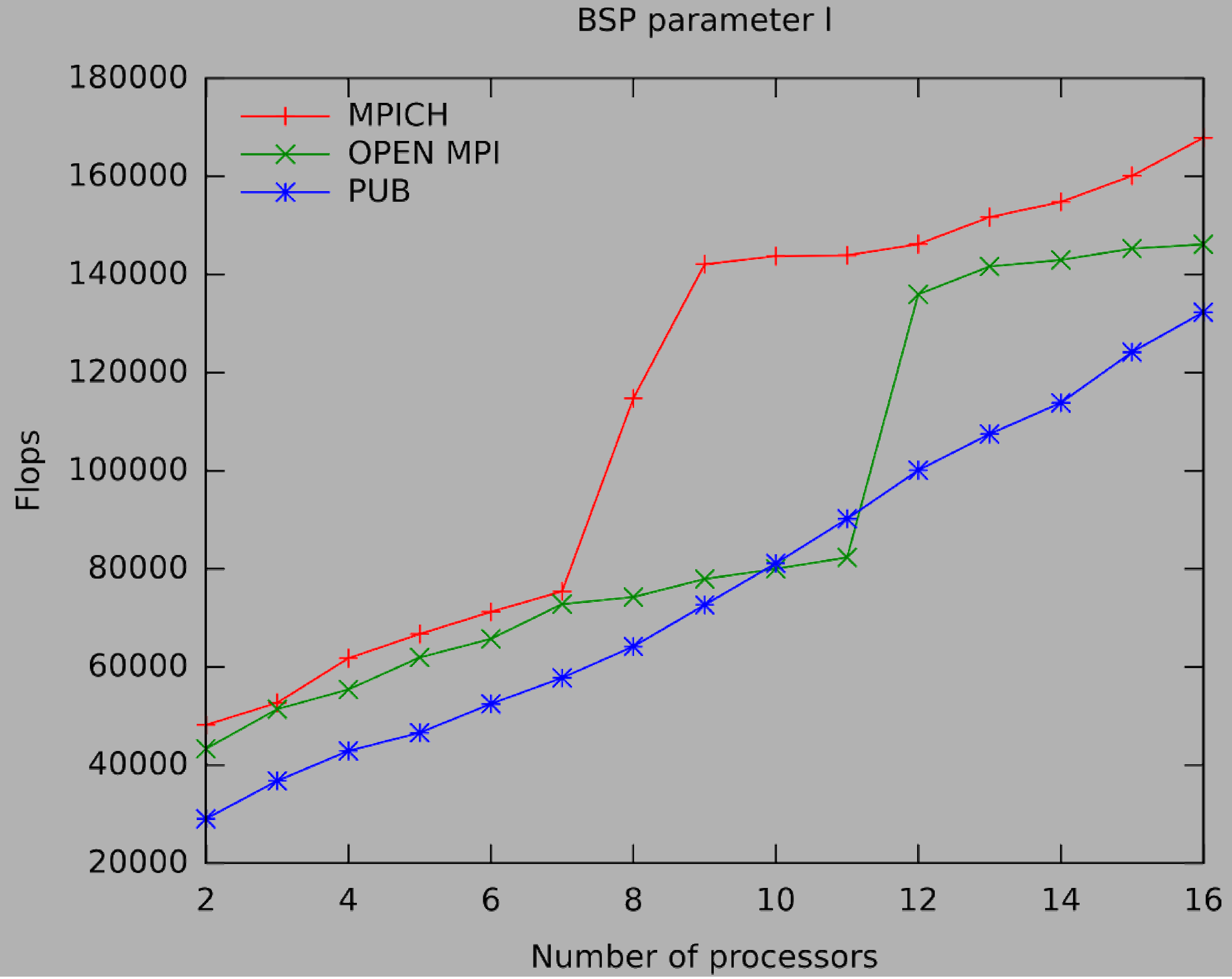
Our parallel machine

- Cluster of PCs
  - Pentium IV 2.8 Ghz
  - 512 Mb RAM
- A front-end Pentium IV 2.8 Ghz, 512 Mb RAM
- Gigabit Ethernet cards and switch,
- Ubuntu 7.04 as OS

Our BSP Parameters ‘g’



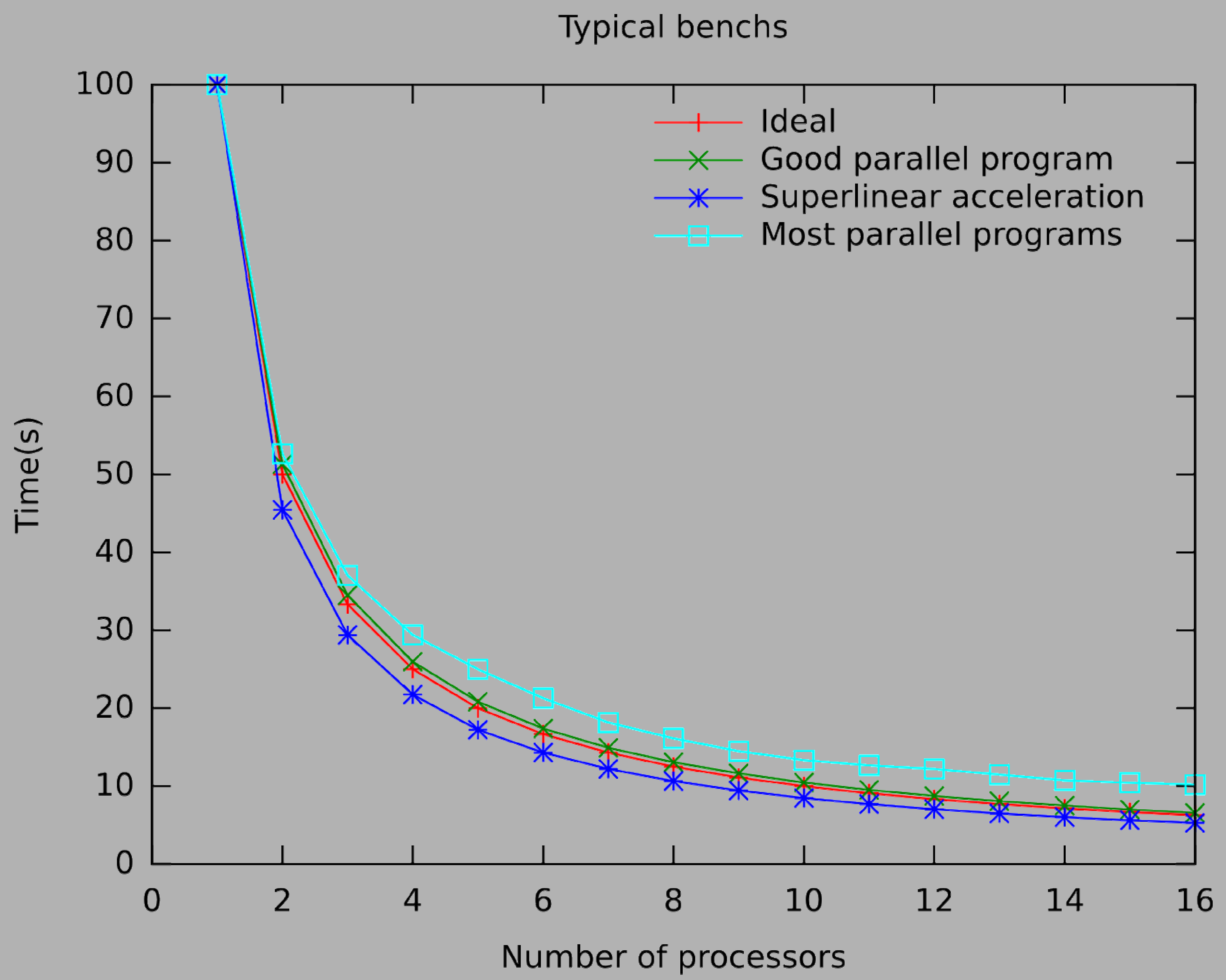
Our BSP Parameters ‘L’



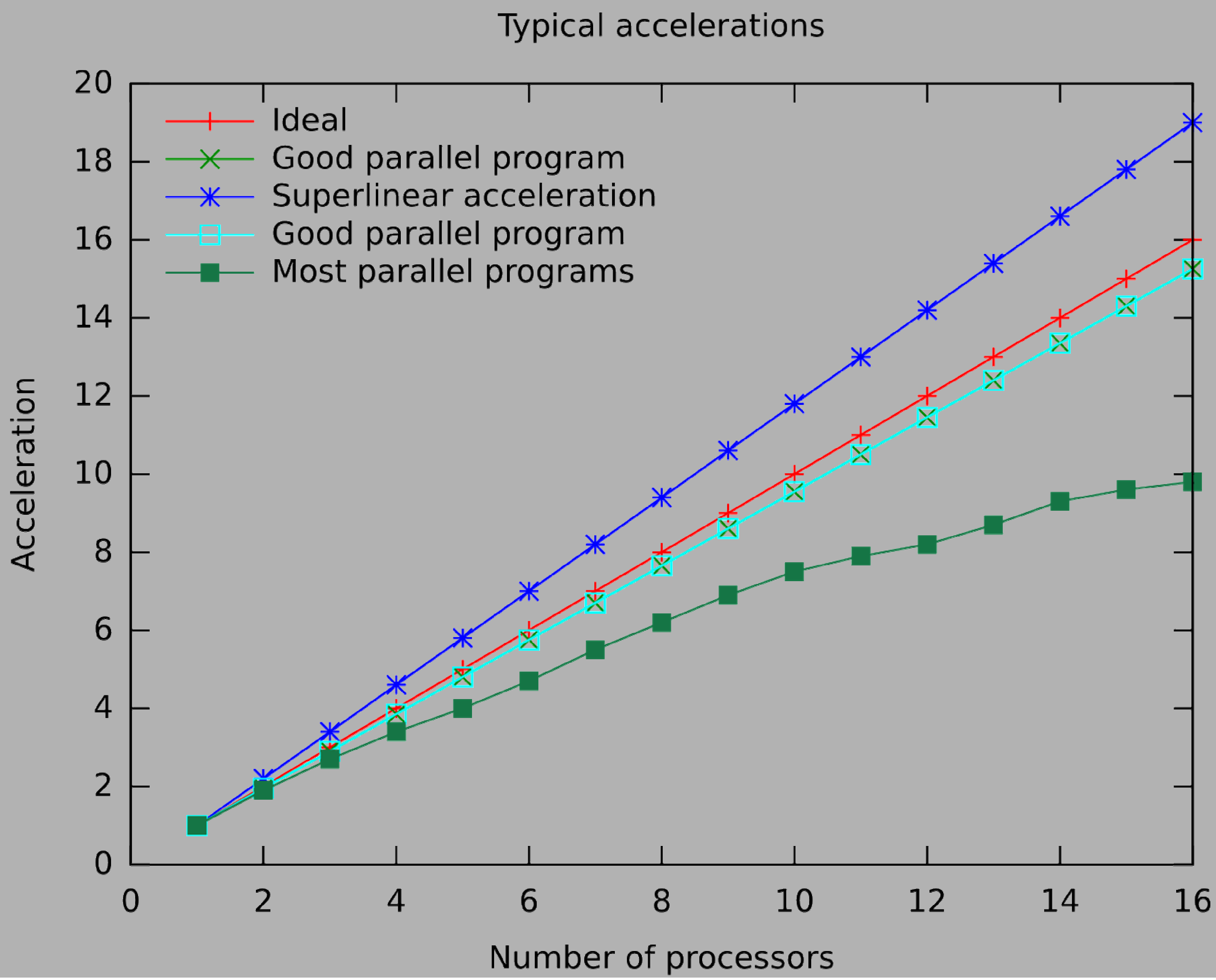
How to read bench

- There are many manners to publish benchs :
  - Tables
  - Graphics
- The goal is to say « *it is a good parallel method, see my benchs* » but it is often easy to arrange the presentation of the graphics to hide the problems
- Using graphics (from the simple to hide to the hardest) :
  1) Increase size of data and see for some number of processors
  2) Increase number of processors to a typical size of data
  3) Acceleration, i.e, Time(seq)/Time(par)
  4) Efficienty , i.e, Acceleration/Number of processors
  5) Increase number of processors and size of the data

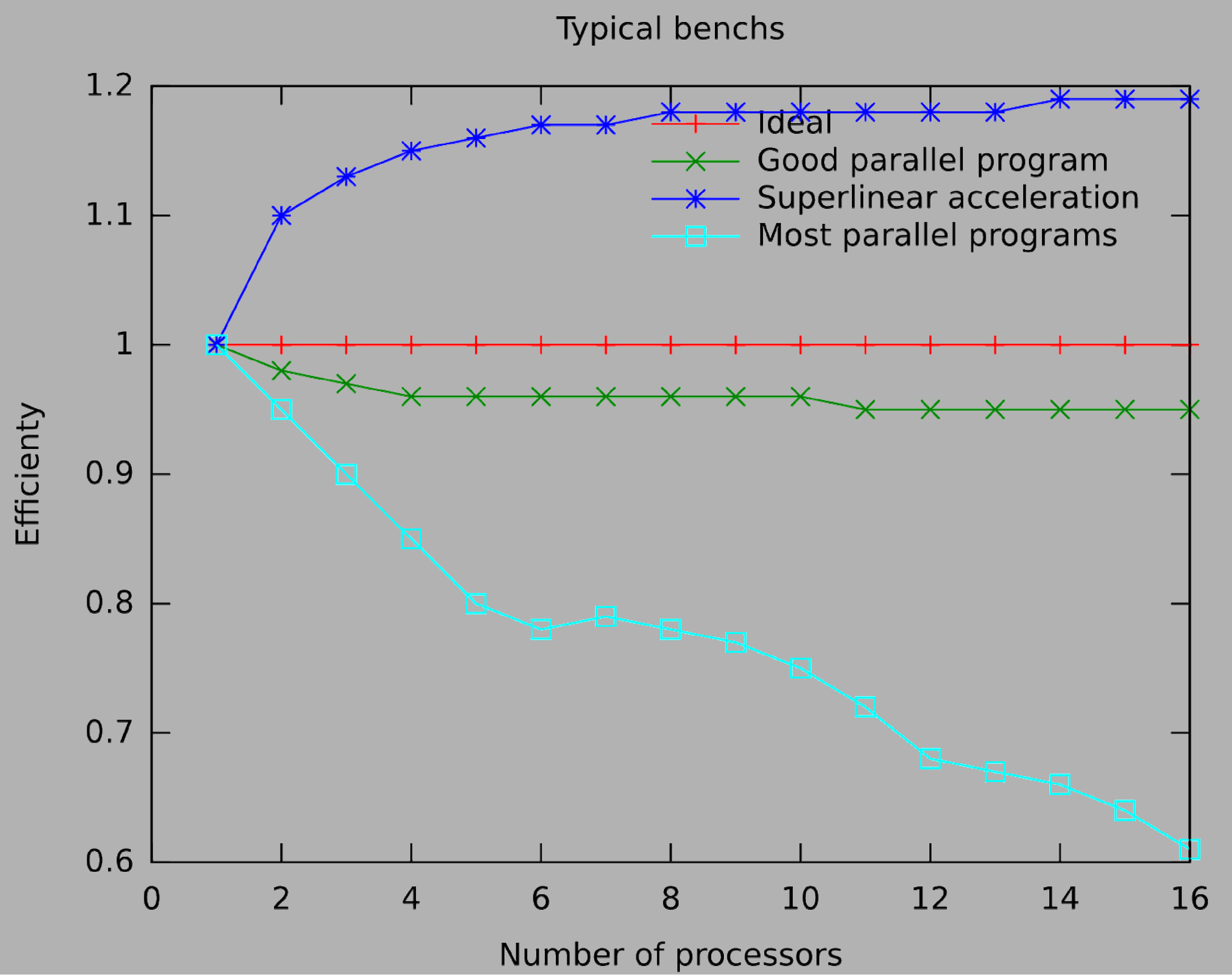
Increase number of processors



Acceleration

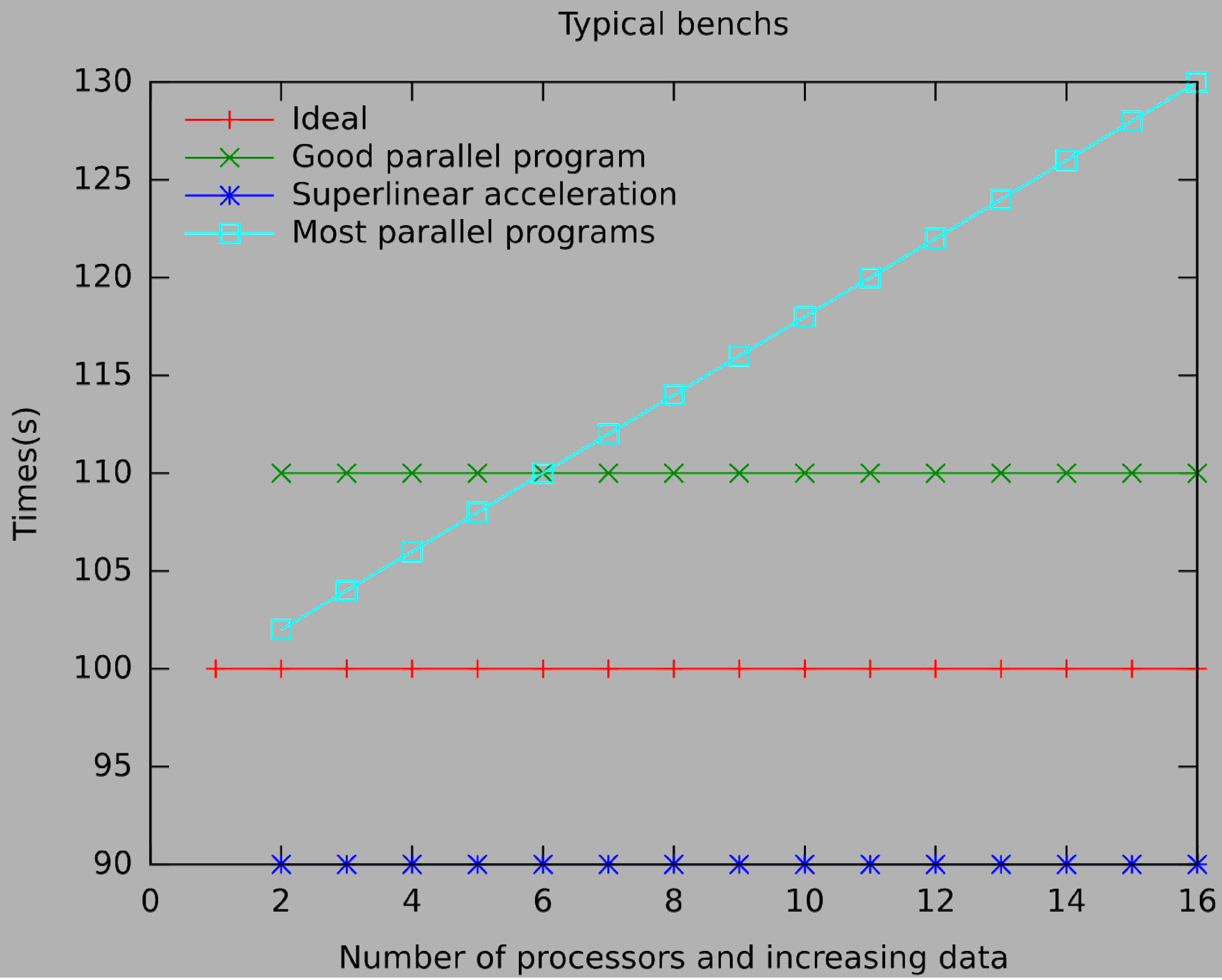


Efficienty



Typical benchs
Efficienty
Number of processors
Ideal
Good parallel program
Superlinear acceleration
Most parallel programs
1.2
1.1
1
0.9
0.8
0.7
0.6
0
2
4
6
8
10
12
14
16

Increase data and processors



More complicated examples

N-body problem

Presentation

- We have a set of body
  - coordinate in 2D or 3D
  - point masse
- The classic N-body problem is to calculate the gravitational energy of N point masses that is :

$$E = -\sum_{\substack{i=1 \\ i \neq j}}^{N} \sum_{j=1}^{N} \frac{m_i \times m_j}{r_i - r_j}$$

- Quadratique complexity…
- In practice, N is very big and sometime, it is impossible to keep the set in the main memory

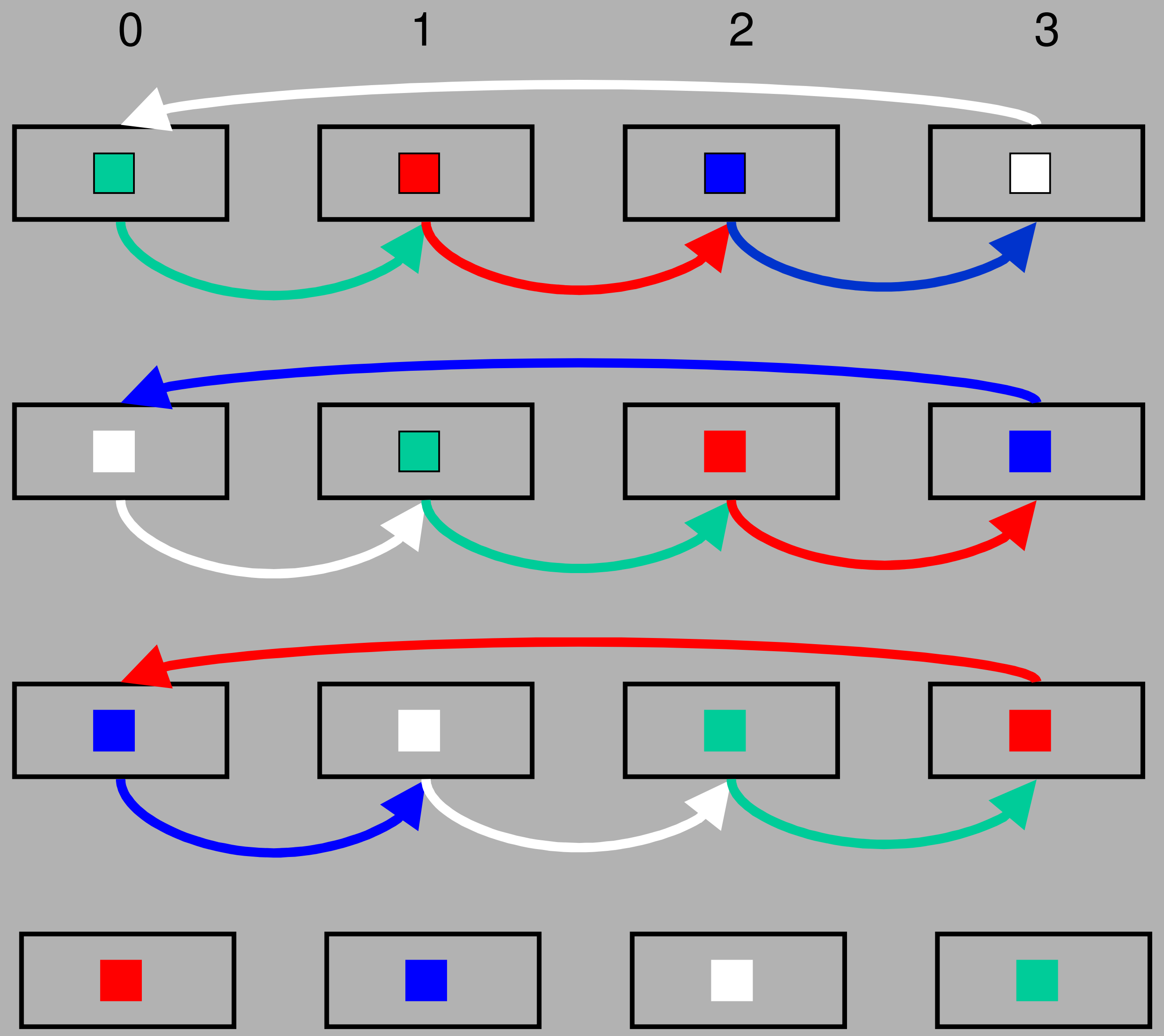
Parallel methods

- Each processor has a sub-part of the original set
- Parallel method one each processsor :

1) compute local interactions
2) compute interactions with other point masses
3) parallel prefixes of the local interactions

- For 2) simple parallel methods :
  - using a total exchange of the sub-sets
  - using a systolic loop

Systolic loop



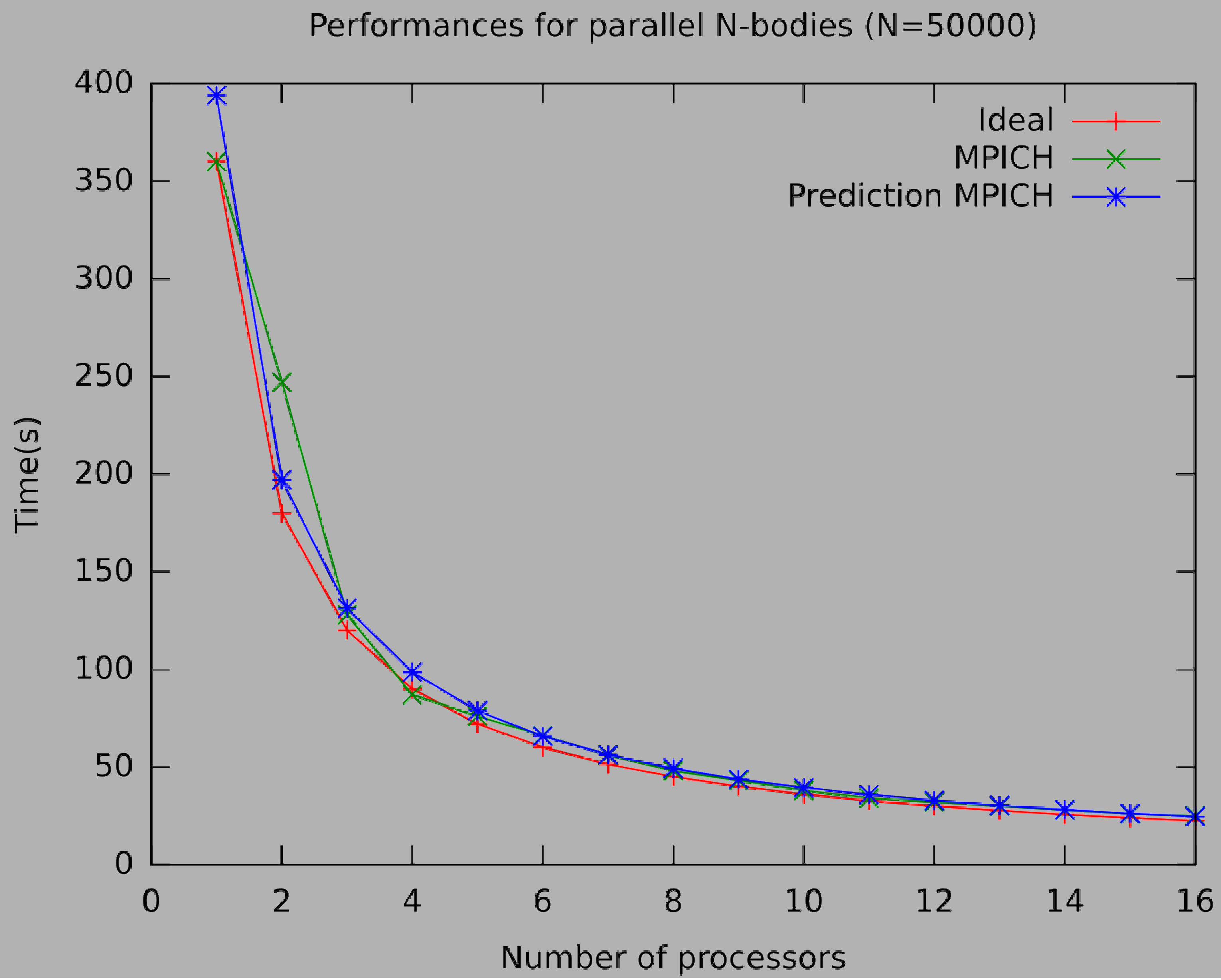
Systolic loop in BSML

```
(* val systolic:(α →α →β ) →(γ →β par →γ ) →α par →γ →γ *)
let systolic f op vec init =
 let rec calc n v res =
  if n=0 then res else
   let newv=Bsmlcomm.shift_right v in
    calc (n−1) newv (op res (parfun2 f vec newv))
  in calc (bsp_p()) vec init
```

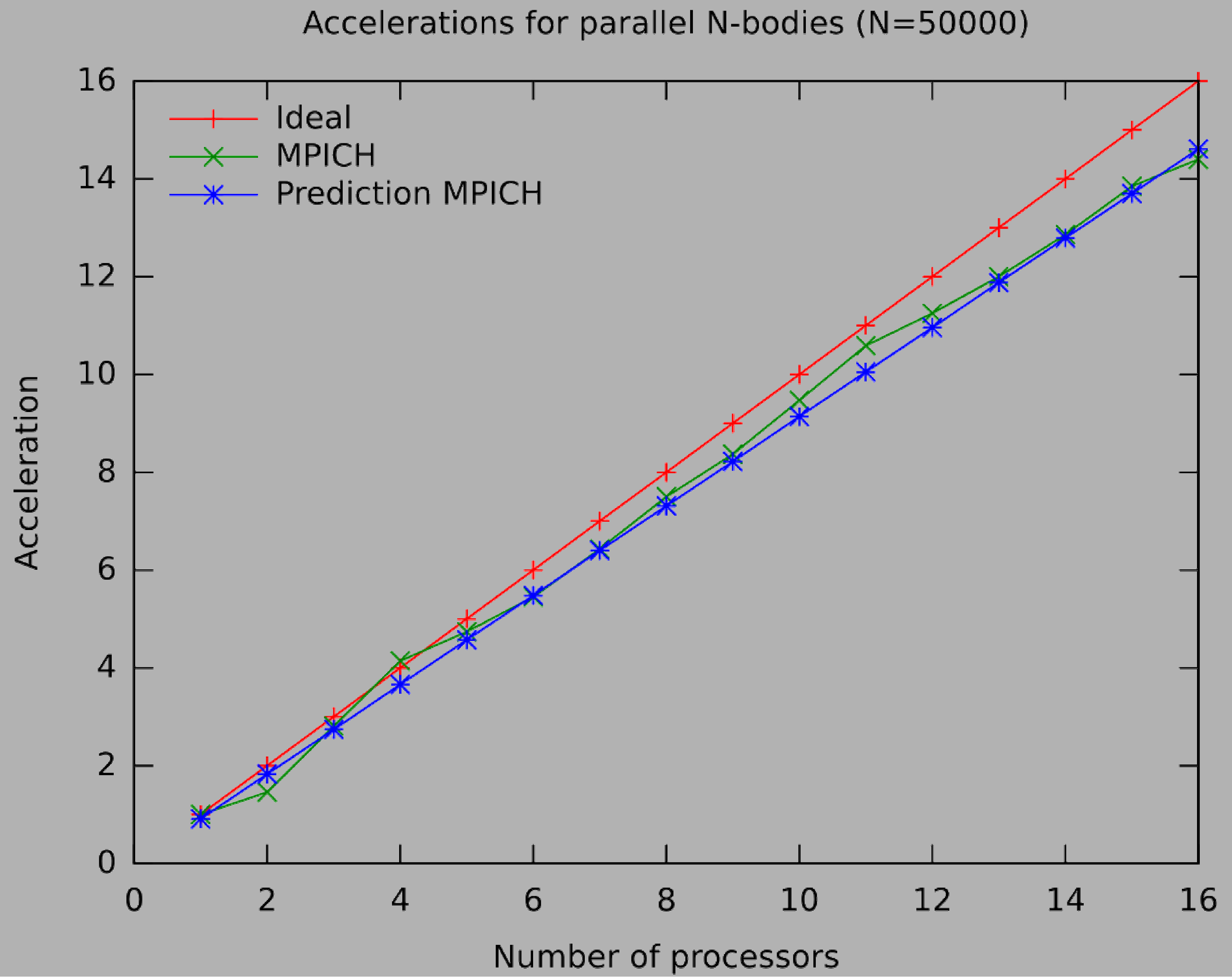
Cost of the systolic method :

$$N \times \mathbf{g} + \mathbf{p} \times \mathbf{l} + 2 \times N + \frac{N}{\mathbf{p}} \times N + \mathbf{l} + \mathbf{p} \times \mathbf{g} + \mathbf{l}$$

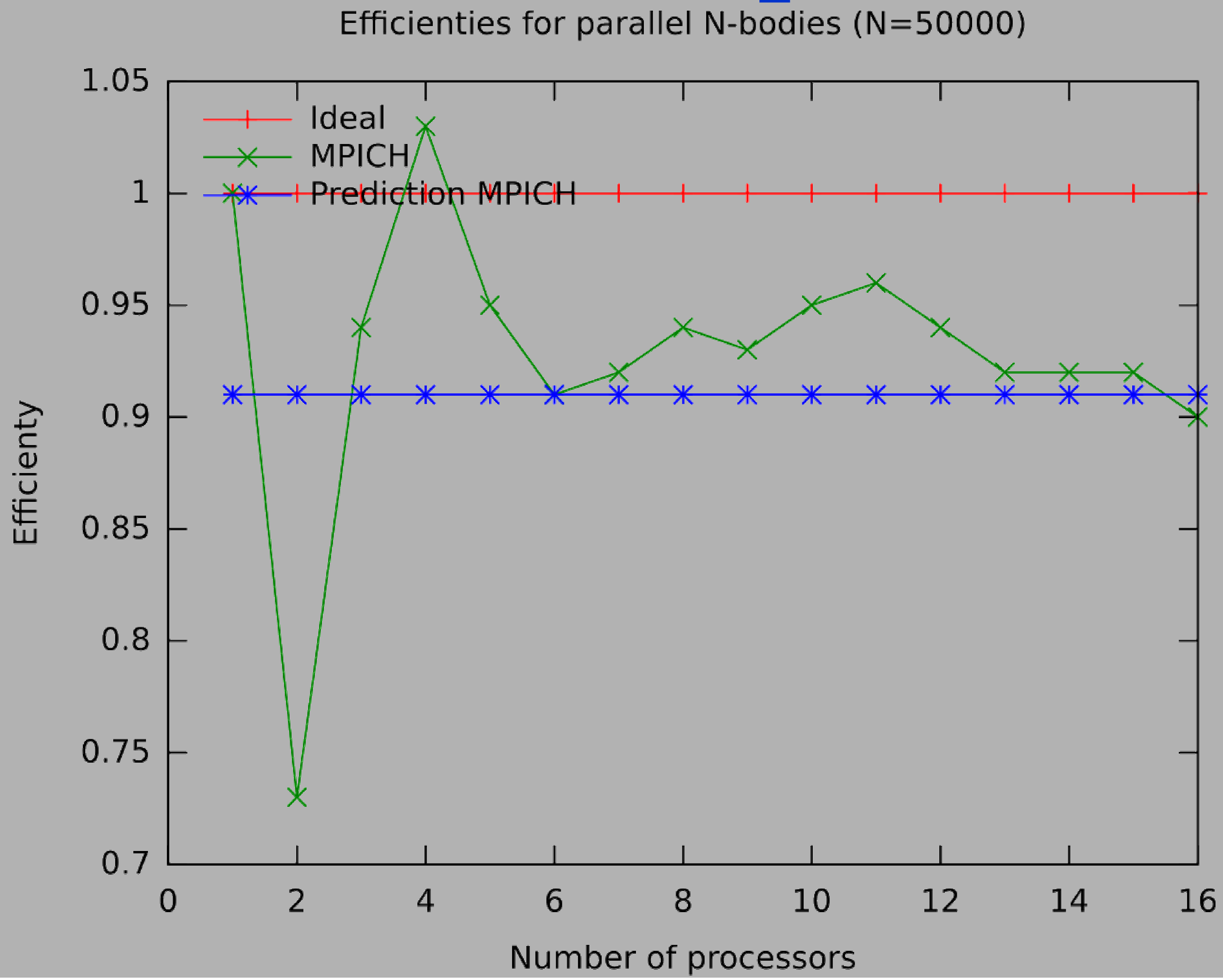
Bench and BSP predictions



Bench and BSP predictions



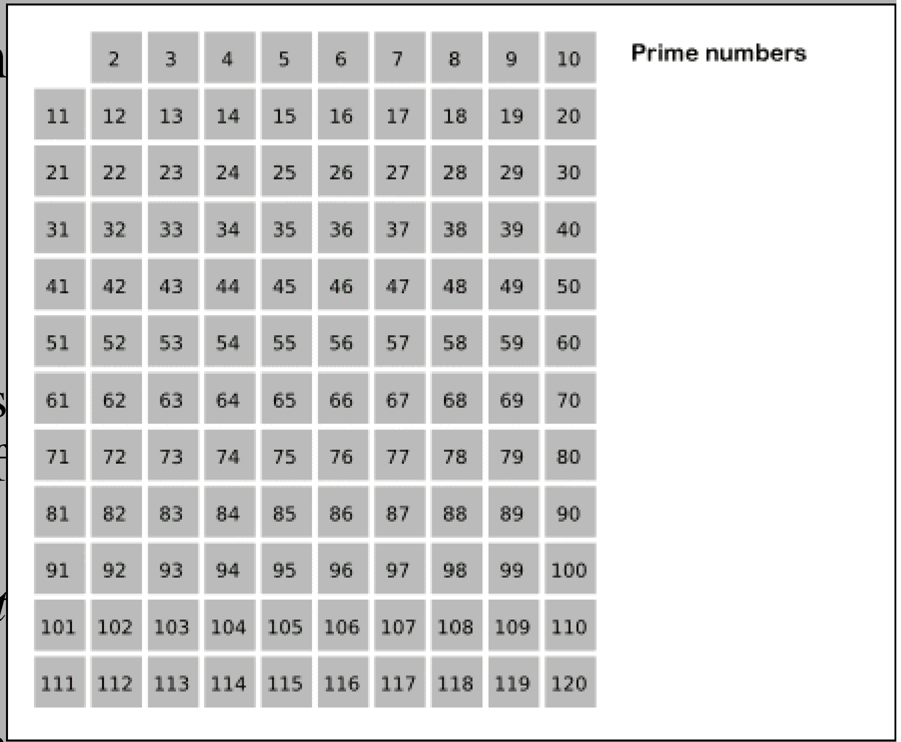
Bench s and BSP predictions



Sieve of Eratosthenes

Presentation

- Classic : find the prime number by enumeration
- Pure functional implementation using list
- Complexity : n×log(n)/log(log(n))
- We used :
  - *elim:int list→int→int* list which deletes from a list all the integers multiple of the given parameter
  - *final elim:int list→int list→int list* iterates elim
  - *seq_generate:int→int→int list* which returns the list of integers between 2 bounds
  - *select:int→int list→int list* which gives the first prime numbers of a list.



Parallel methods

- Simple Parallel methods :
  - using a kind of scan
  - using a direct sieve
  - using a recursive one
- Different partitions of data
  - per block (for scan) :

11,12,13,14,15	16,17,18,19,20	21,22,23,24,25

  - cyclic distribution :

11,14,17,20,23	12,15,18,21,24	13,16,19,22,25

Scan version

- Method using a scan :
  - Each processor computes a local sieve (the processor 0 contains thus the first prime numbers)
  - then our scan is applied and we eliminate on processor i the integers that are multiple of integers of processors i–1, i–2, *etc*.
- Cost : as a scan (logarithmic)

Direct version

- Method :
  - each processor computes a local sieve
  - then integers that are less to $\sqrt{n}$ are globally exchanged and a new sieve is applied to this list of integers (thus giving prime numbers)
  - each processor eliminates, in its own list, integers that are multiples of this first primes
- BSML Code :

```
let eratosthene_direct n =
 let listes = mkpar (fun pid→ local_generation n pid) in
 let etape1 = parfun (local_eratosthene n) listes in
 let selects = parfun (select n) etape1 in
 let echanges = replicate_total_exchange selects in
 let premiers = local_eratosthene n
                          (List.fold_left (List.merge compare) [] echanges) in
 let etape2 = parfun (final_elim premiers) etape1 in
  applyat 0 (fun l→2::3::5::7::(premiers@l)) (fun l→l) etape2
```

Inductive version

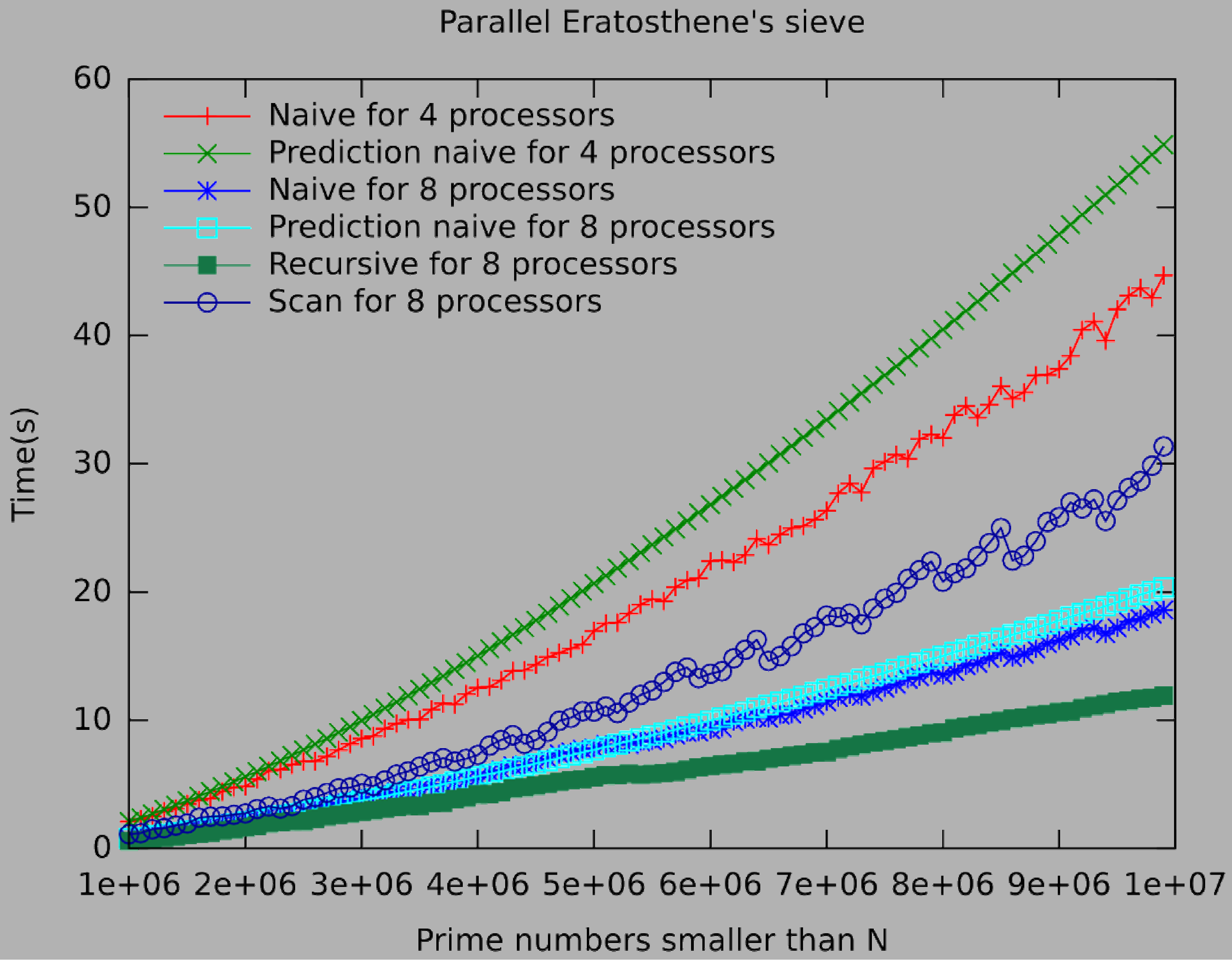
- Recursive method by induction over n :
  - We suppose that the inductive step gives the $\sqrt{n}$th first primes
  - we perform a total exchange on them to eliminates the non-primes.
  - End of this induction comes from the BSP cost: we end when n is small enough so that the sequential methods is faster than the parallel one
- Cost :

$$
\begin{aligned}
\mathrm{Cost}(n) &= \frac{\sqrt{m}\times m}{\log(m)} + \sqrt{n}\times \mathbf{g} + \mathbf{l} + \mathrm{Cost}(\sqrt{n}) \\
\mathrm{Cost}(n) &= \frac{\sqrt{n}\times n}{\log(n)} && \text{if BSP cost} > \text{complexity}
\end{aligned}
$$

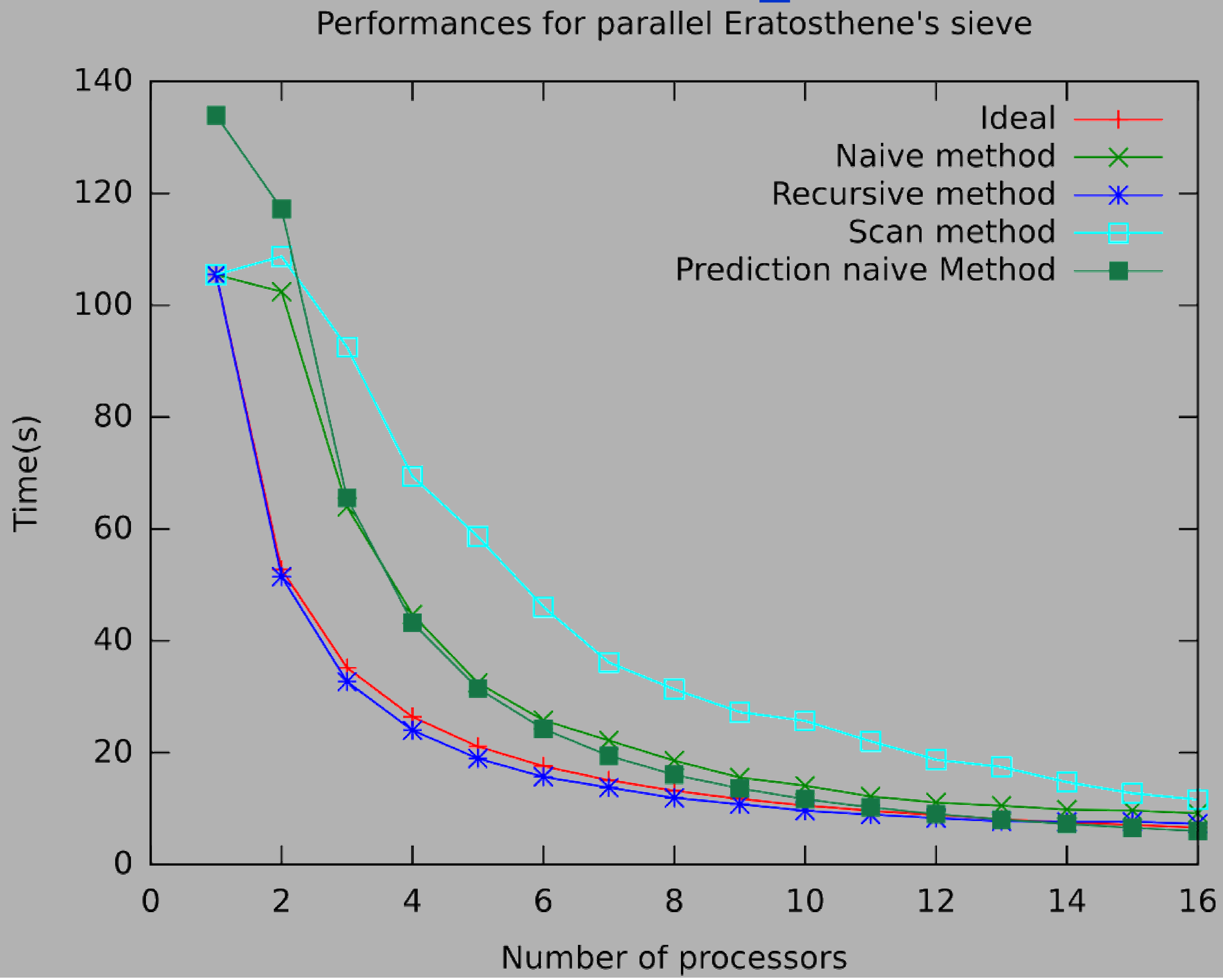
Induction version in BSML

```
let rec eratosthene n =
  if (fin_recursion n) then apply (mkpar distribution) (replicate (seq_eratosthene n))
  else
   let carre_n = int_of_float (sqrt (float_of_int n)) in
   let prems_distr = eratosthene carre_n in
   let listes = mkpar (fun pid →local_generation2 n carre_n pid) in
   let echanges = replicate_total_exchange prems_distr in
    let prems = (List.fold_left (List.merge compare) [] echanges) in
     parfun (final_elim prems) listes
let eratosthene_rec n =
 applyat 0 (fun l→2::3::5::7::l) (fun l→l) (eratosthene n)
```

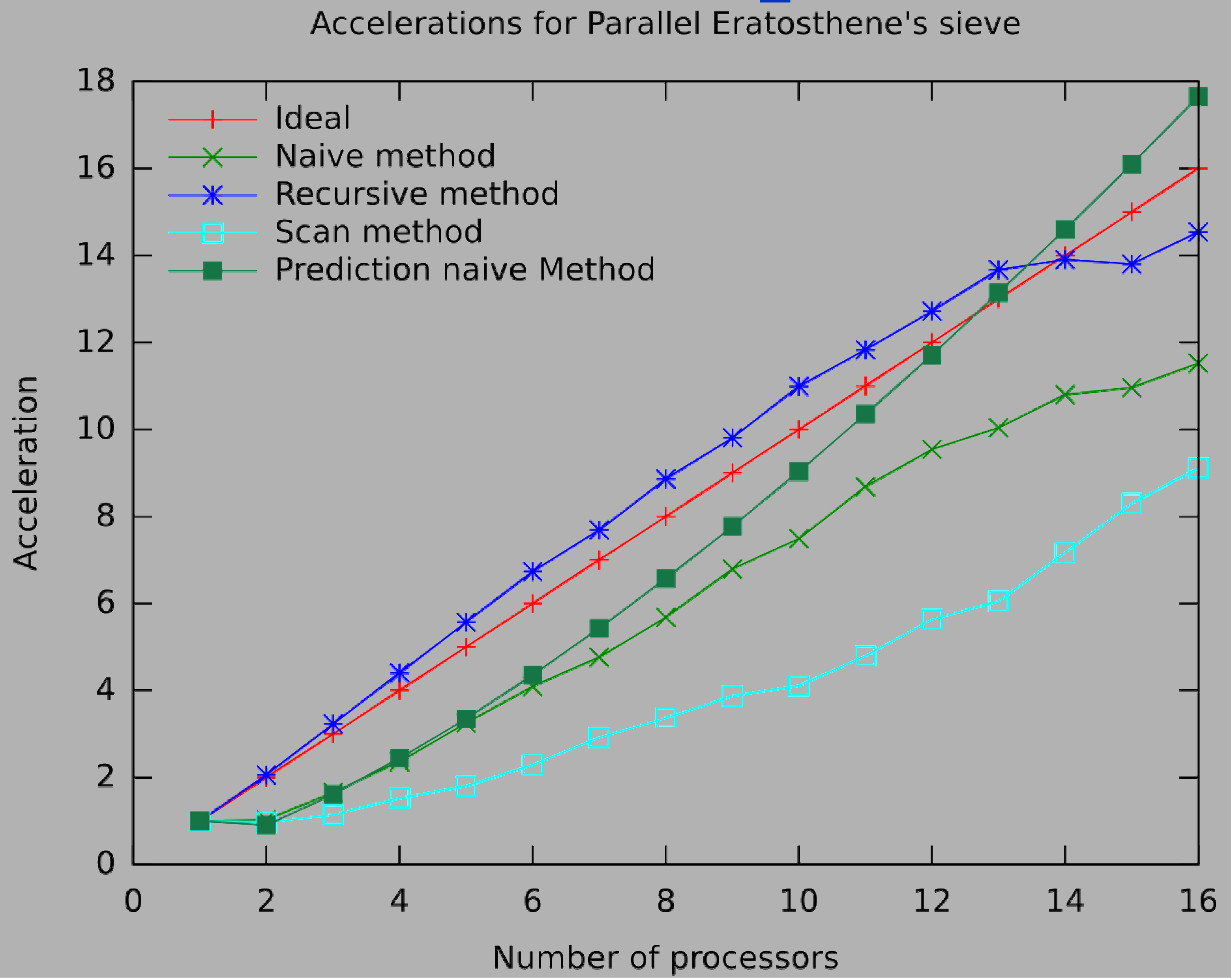
Benchs and BSP predictions



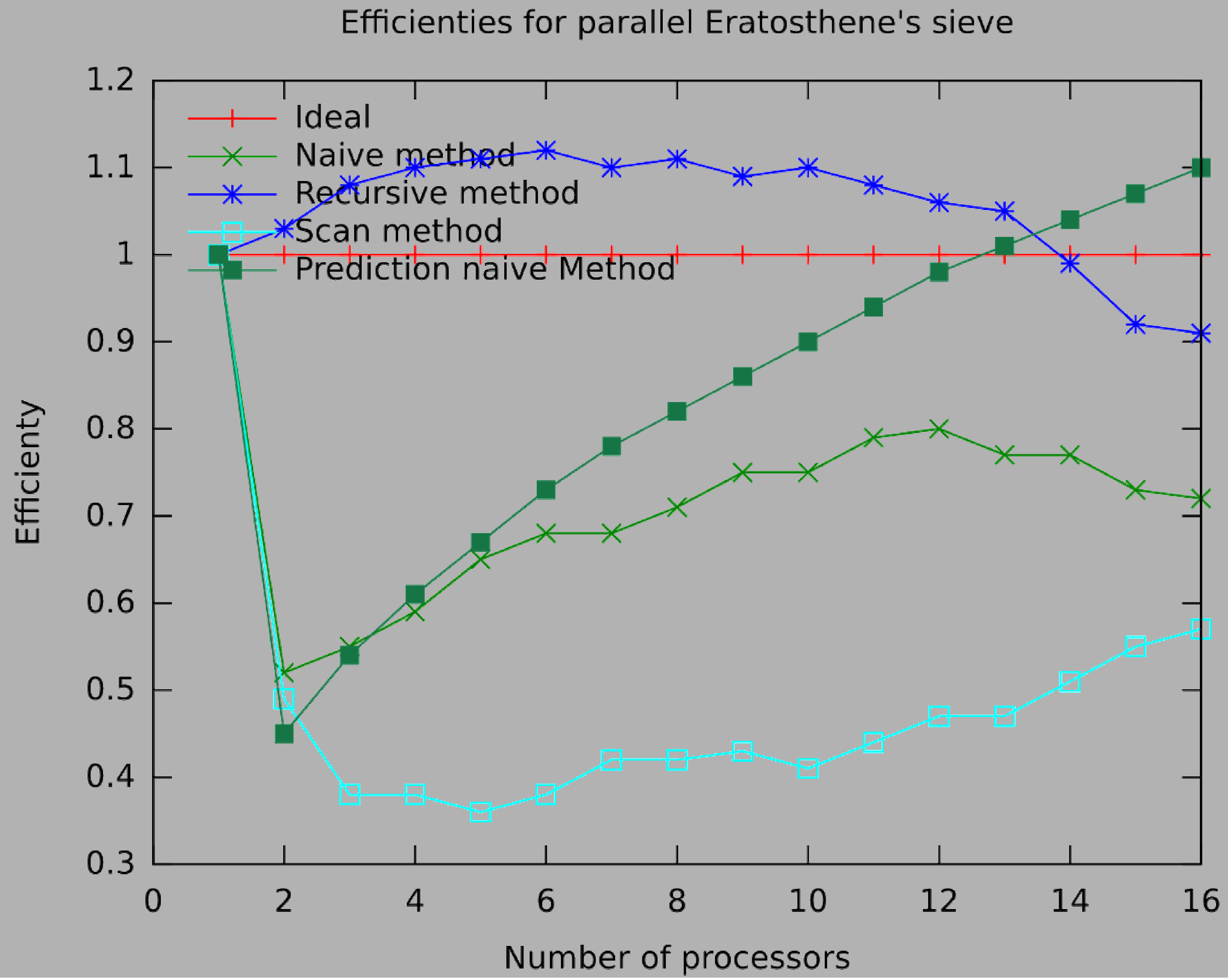
Bench and BSP predictions



Benchs and BSP predictions



Bench and BSP predictions

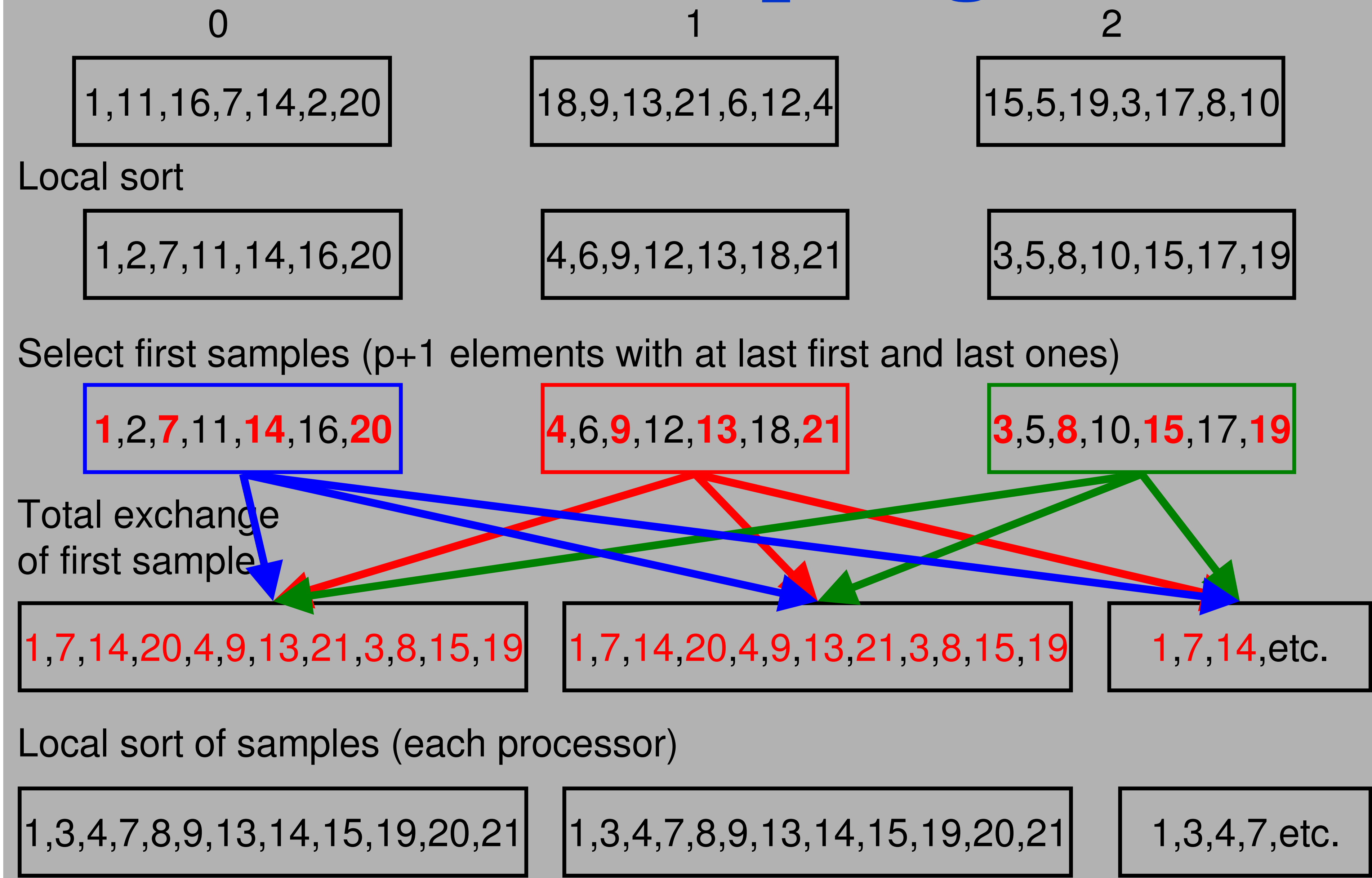


Parallel sample sorting

Presentation

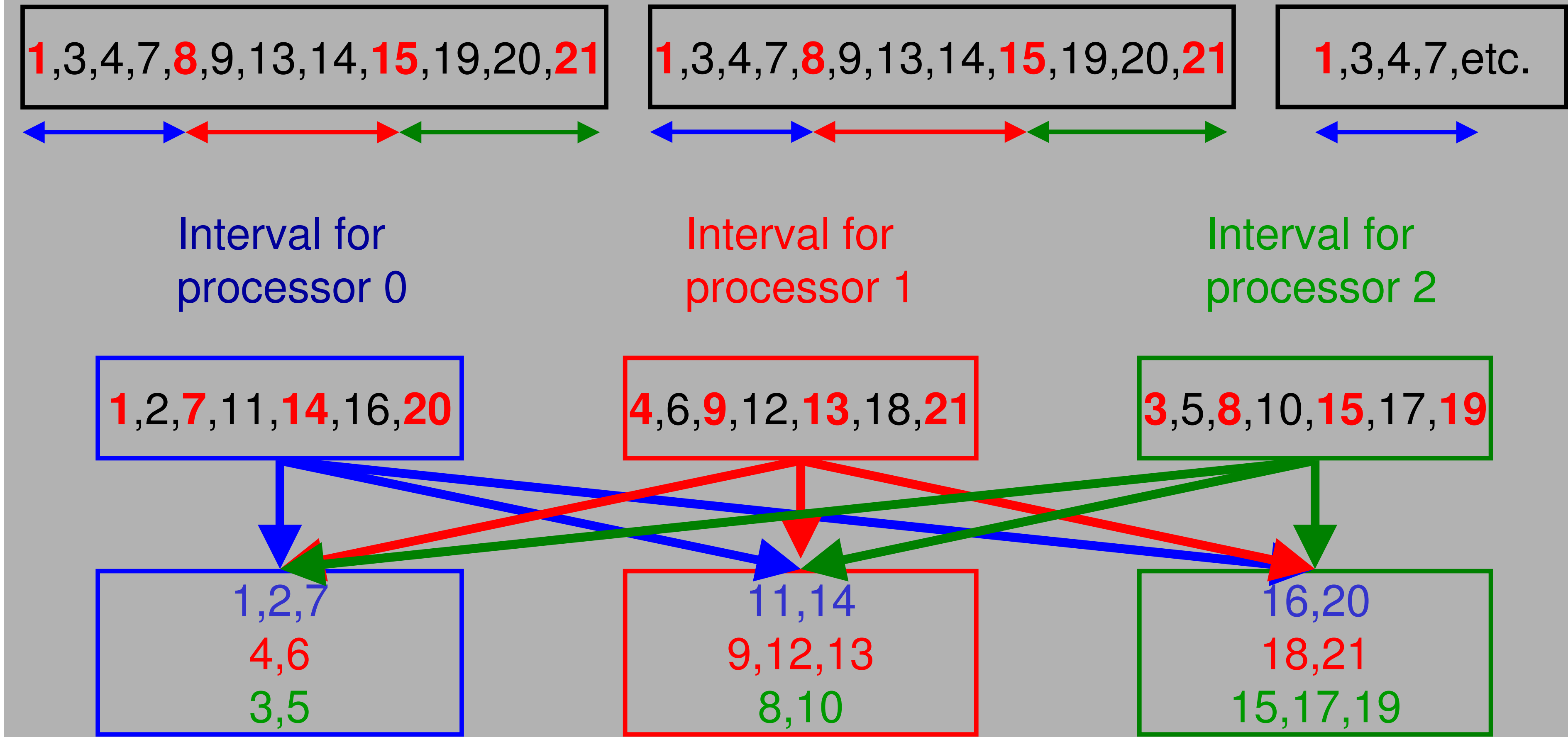
- Each processor has listed set of data (array, list, *etc.*)
- The goal is that :
  - data on each processor are ordored.
  - data on processor *i* are smaller than data on processor *i+1*
  - good balancing
- Parallel sorting is not very efficient due to too many communications
- But usefull and more efficient than gather all the data in one processor and then sort them

Tiskin's Sampling Sort



Tiskin's Sampling Sort

Select second samples (p+1 elements with at last first and last ones)



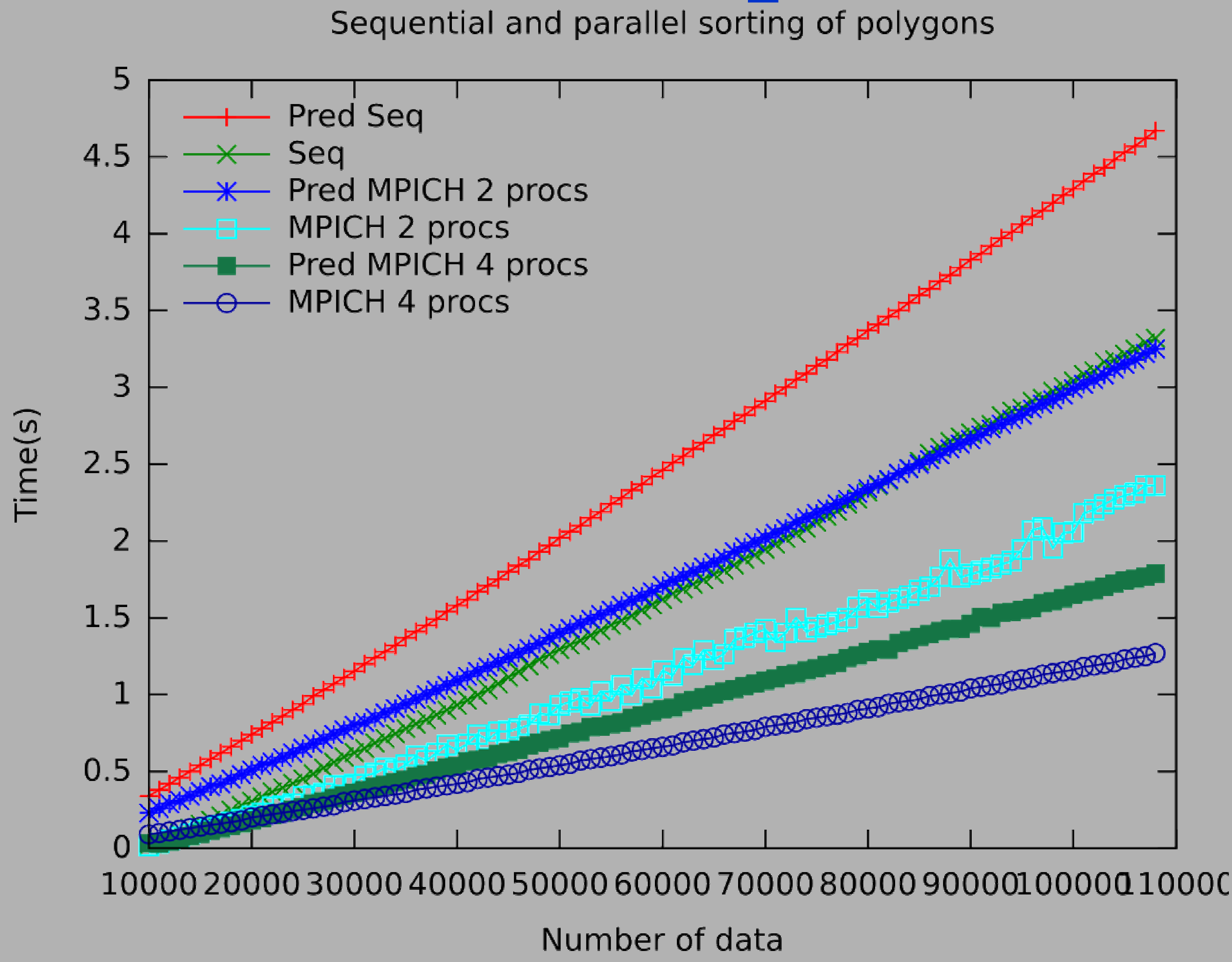
Fusion of receveid and sorted elements

1,2,3,4,5,6,7	8,9,10,11,12,13,14	15,16,17,18,19,20,21

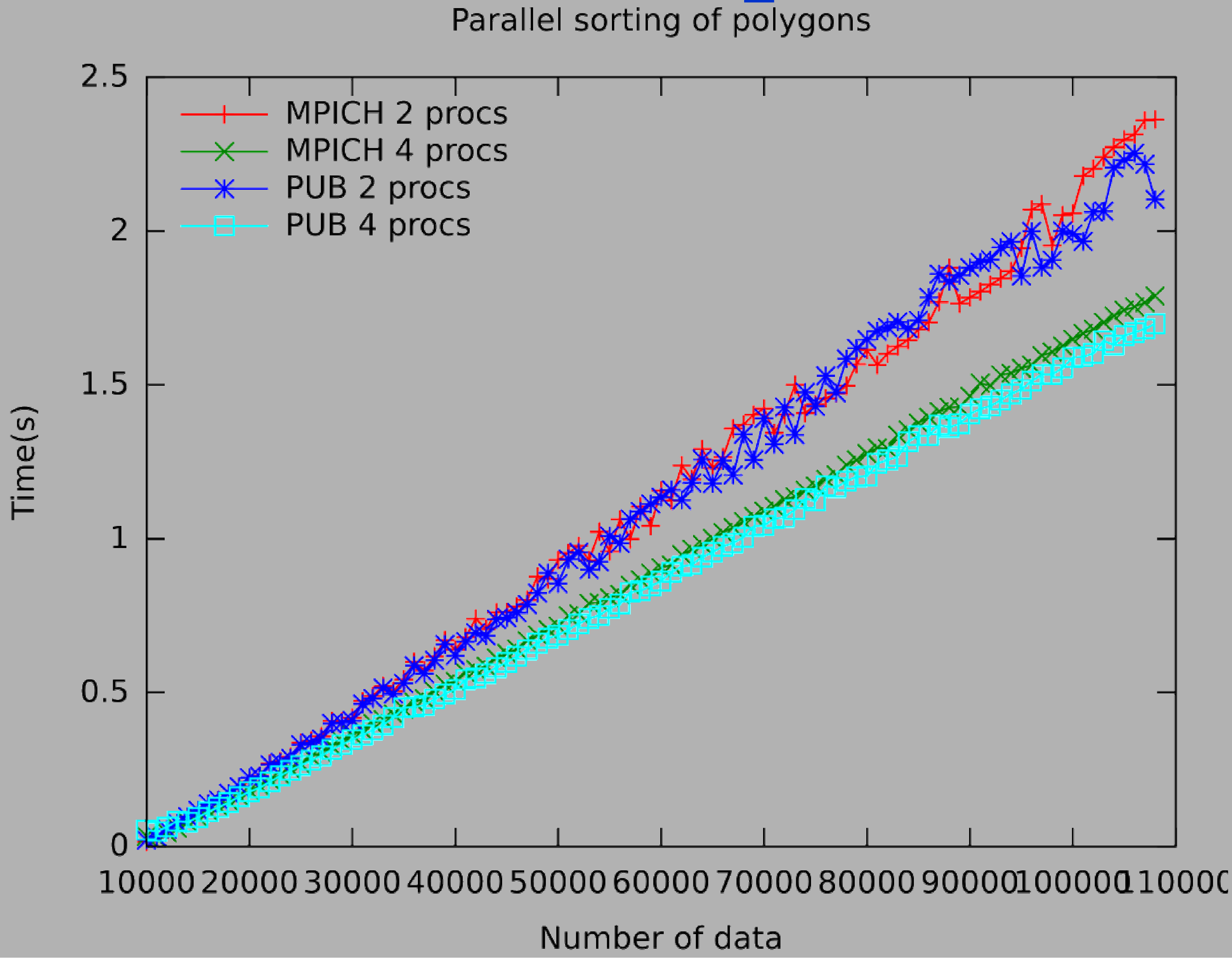
Parallel Sorting in BSML

```
let bsp_sample_sort compare seq_sort select merge_samples to_be_send
                                        get merge_block vec =
 let p=bsp_p() in
 (* merge the sending blocks at the end *)
 let final_merge f =
  let rec final n tmp =
    if n=p then tmp else final (n+1) (merge_block compare tmp (f n))
   in final 1 (f 0)
 in
  (* Super−step 1 *)
  let vec_sort = parfun (seq_sort compare) vec in
  let primary_sample = parfun (select p) vec_sort in
  let totex_prim_sample = replicate_total_exchange primary_sample in
  (* Super−step 2 *)
  let scd_sample = select p (merge_samples compare totex_prim_sample) in
  let elts_to_send = parfun (to_be_send compare p scd_sample) vec_sort in
  let to_send = put (parfun get elts_to_send) in
  (* Super−step 3 *)
   parfun final_merge to_send
```

Benchs and BSP predictions



Bench and BSP predictions



Matrix multiplication

Naive parallel algorithm

- We have two matrices A and B of size n×n
- We supose $p = \sqrt{p} \times \sqrt{p}$
- Each matrice is distributed by blocs of size $m = \frac{n}{\sqrt{p}}$
- That is, element A(i,j) is on processor $(\frac{j}{m}) \times \sqrt{p} + \frac{i}{m}$
- Algorithm :

$$
\begin{array}{l}
\textbf{begin } \mathrm{Mult(C,A,B)} \\
\quad \textbf{let } m = \frac{n}{\sqrt{p}} \textbf{ in} \\
\quad \textbf{let } p_i = pid \mod \sqrt{p} \textbf{ and } p_j = \frac{pid}{\sqrt{p}} \textbf{ and } C_q = [0] \textbf{ in} \\
\quad \textbf{for } 0 \leq l < \sqrt{p} \textbf{ do} \\
\quad\quad \textbf{begin} \\
\quad\quad\quad \textbf{let } a = \boxed{A_{((p_i+p_j+l) \mod \sqrt{p}) \times \sqrt{p}+p_i}} \\
\quad\quad\quad \textbf{and } b = \boxed{B_{((p_i+p_j+l) \mod \sqrt{p})+p_j \times \sqrt{p}}} \textbf{ in} \\
\quad\quad\quad C_{pid} \leftarrow C_{pid} \oplus a \otimes b \\
\quad\quad \textbf{end} \\
\textbf{end } \mathrm{Mult}
\end{array}
$$

Each processor reads twice one bloc from another processor

Two gets

- Read data from another processor :

```
(* get_from : (int →int) →α par →α par *)
let get_from f parv =
 let comms = put(apply (mkpar (fun me v pid →if me=(f pid) then Some v else None)) parv) in
  apply (mkpar (fun me rcv →match (rcv (f me)) with
                                   None →failwith "Cas␣impossible␣!"
                                 | Some v →v)) comms
```

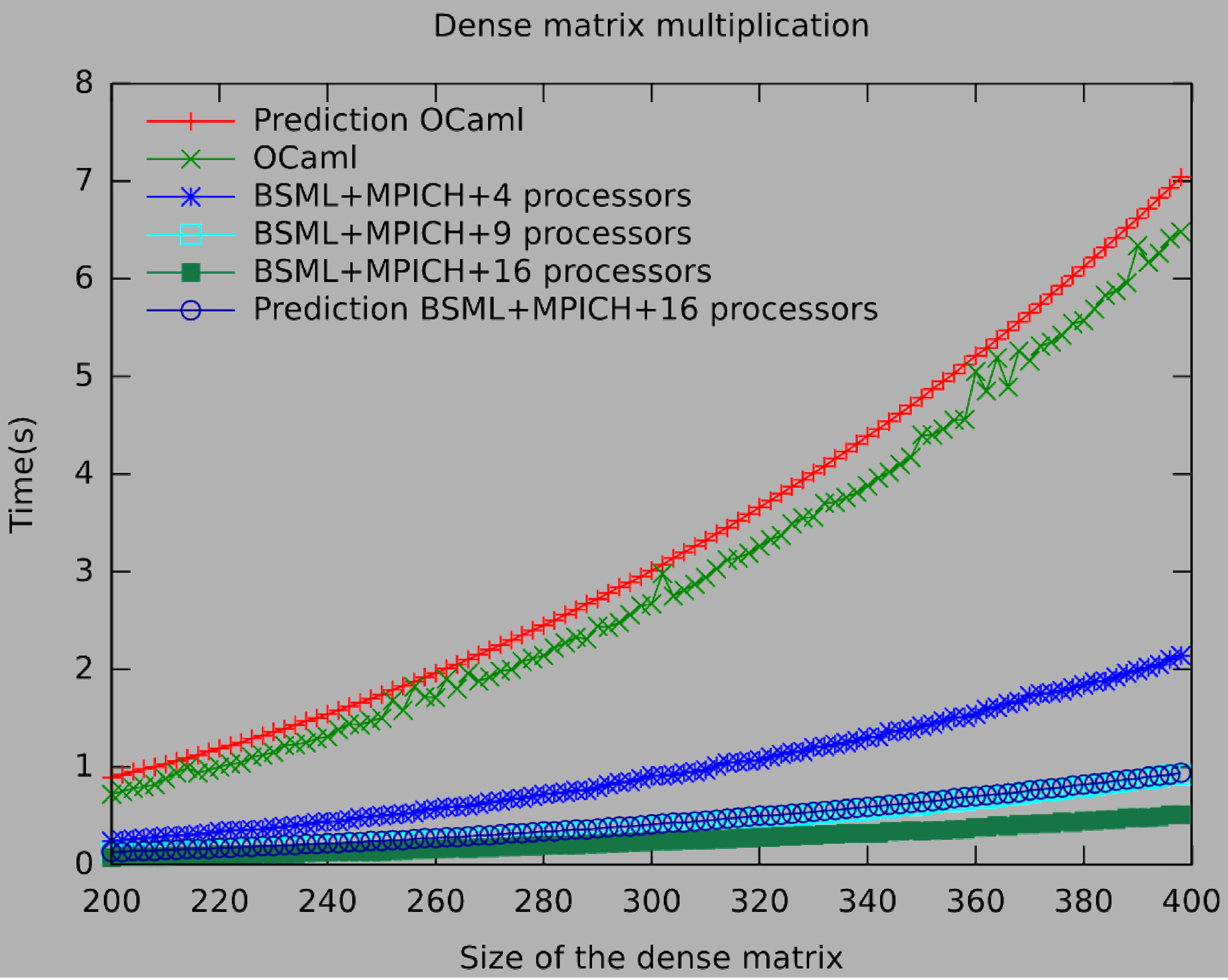
- Read twice = just a superposition of 2 get_from

Mult in BSML

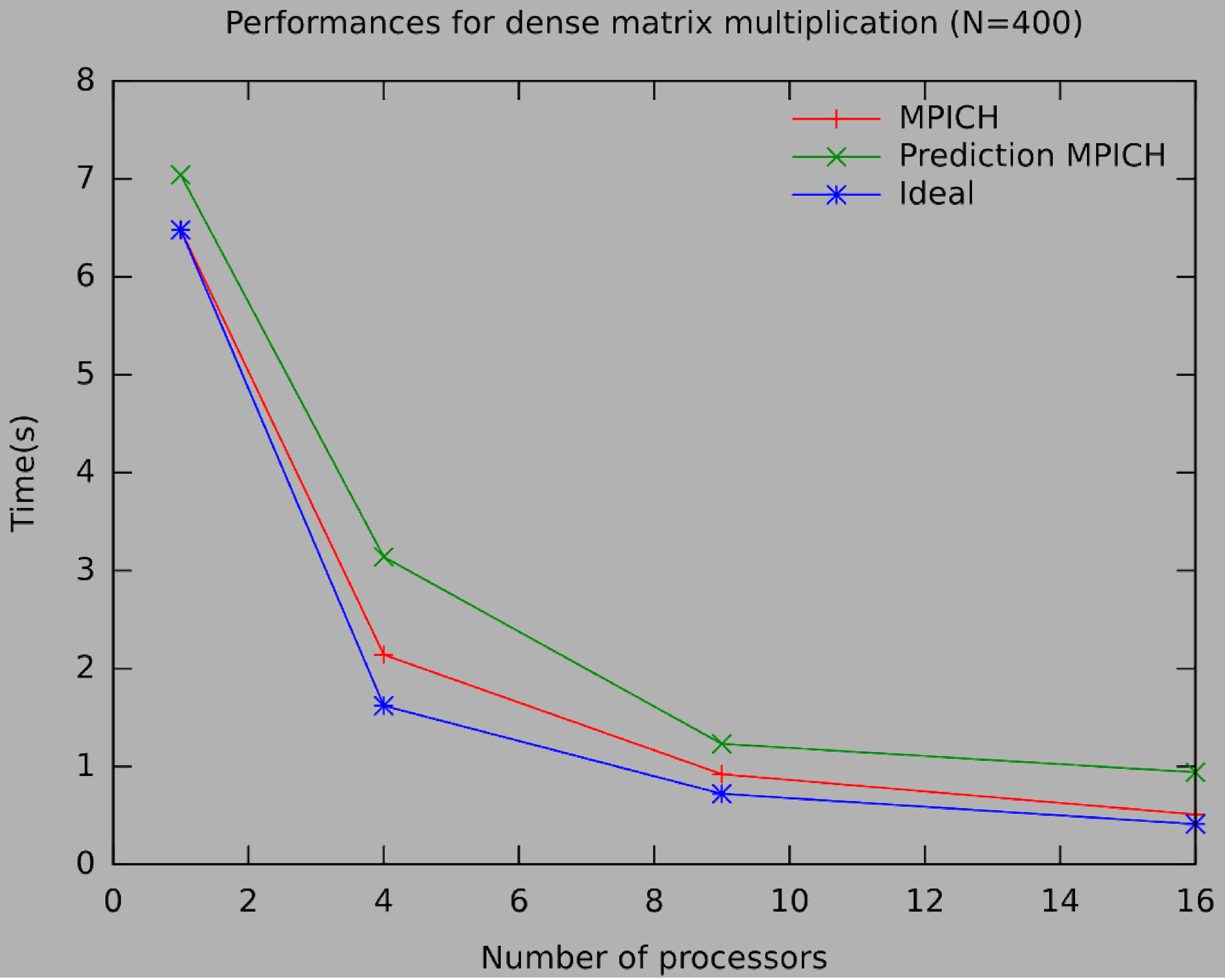
```
(* multiply_par:α →int→(β →α →α )→(α →α →α )→α  array par→β  array par→α  array par→unit *)
let multiply_par e n mult plus parA parB parC =
 let sqrt_p=int_of_float(sqrt(float_of_int (bsp_p()))) in
 let ni=n/sqrt_p in
 let pids_to_sendA l pid =
  let ppi=pid mod sqrt_p and ppj=pid / sqrt_p in
  let from=(ppi+ppj+l) mod sqrt_p in from*sqrt_p+ppi
 and pids_to_sendB l pid =
  let ppi=pid mod sqrt_p and ppj=pid / sqrt_p in
  let from=(ppi+ppj+l) mod sqrt_p in from+ppj*sqrt_p
  in for l=0 to sqrt_p−1 do
       let rcvpA,rcvpB =super (fun ()→ get_from (pids_to_sendA l) parA)
                                 (fun ()→ get_from (pids_to_sendB l) parB) in
        ignore(parfun3 (fun a b c →multiplication e ni mult plus a b c) rcvpA rcvpB parC)
     done
```

begin Mult(C,A,B)
  let $m = \frac{n}{\sqrt{p}}$ **in**
  let $p_i = pid \mod \sqrt{p}$ **and** $p_j = \frac{pid}{\sqrt{p}}$ **and** $C_q = [0]$ **in**
  for $0 \leq l < \sqrt{p}$ **do**
   begin
    let $a = A_{((p_i+p_j+l) \mod \sqrt{p})\times\sqrt{p}+p_i}$
    and $b = B_{((p_i+p_j+l) \mod \sqrt{p})+p_j\times\sqrt{p}}$ **in**
    $C_{pid} \leftarrow C_{pid} \oplus a \otimes b$
   end
end Mult

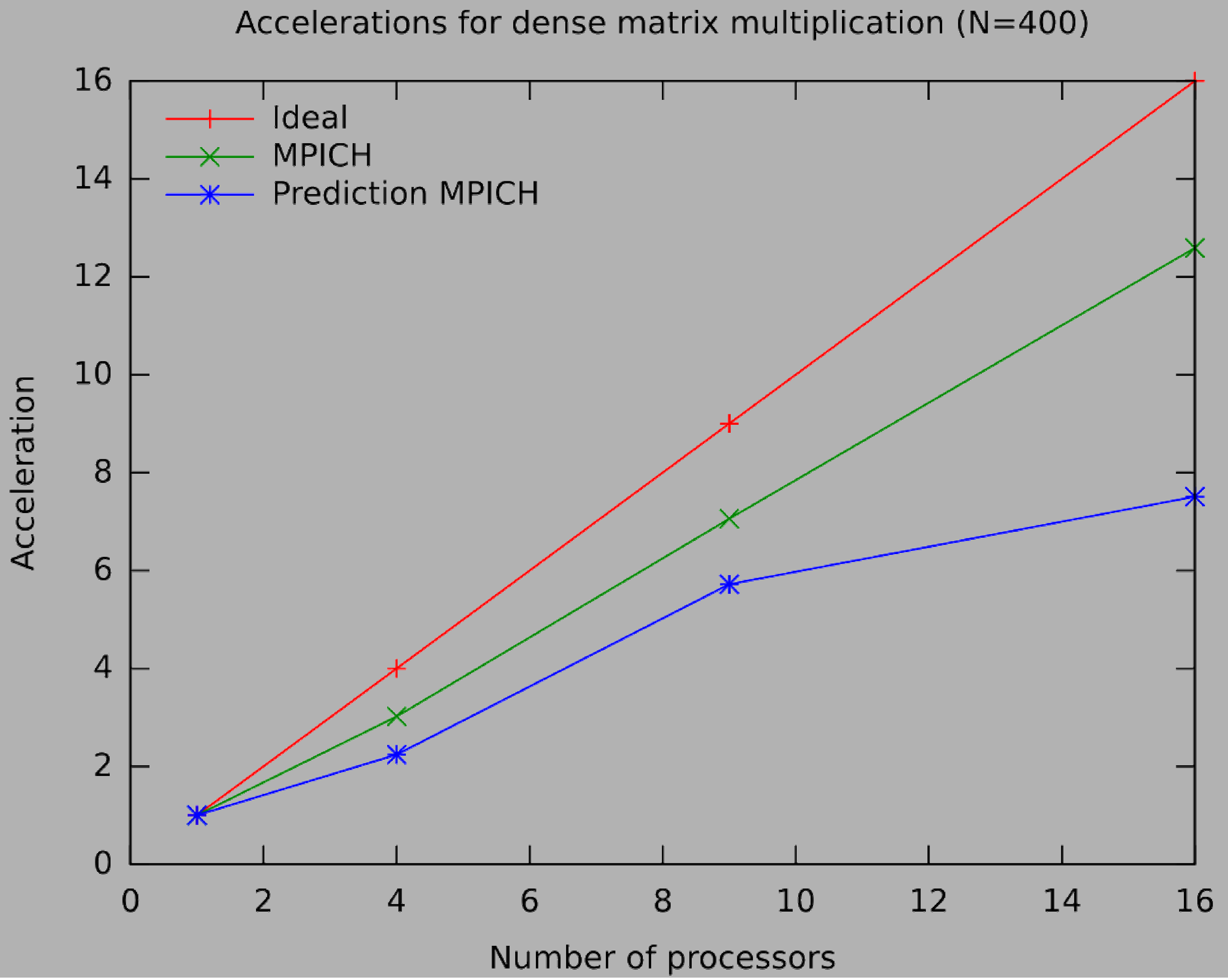
Bench and BSP predictions



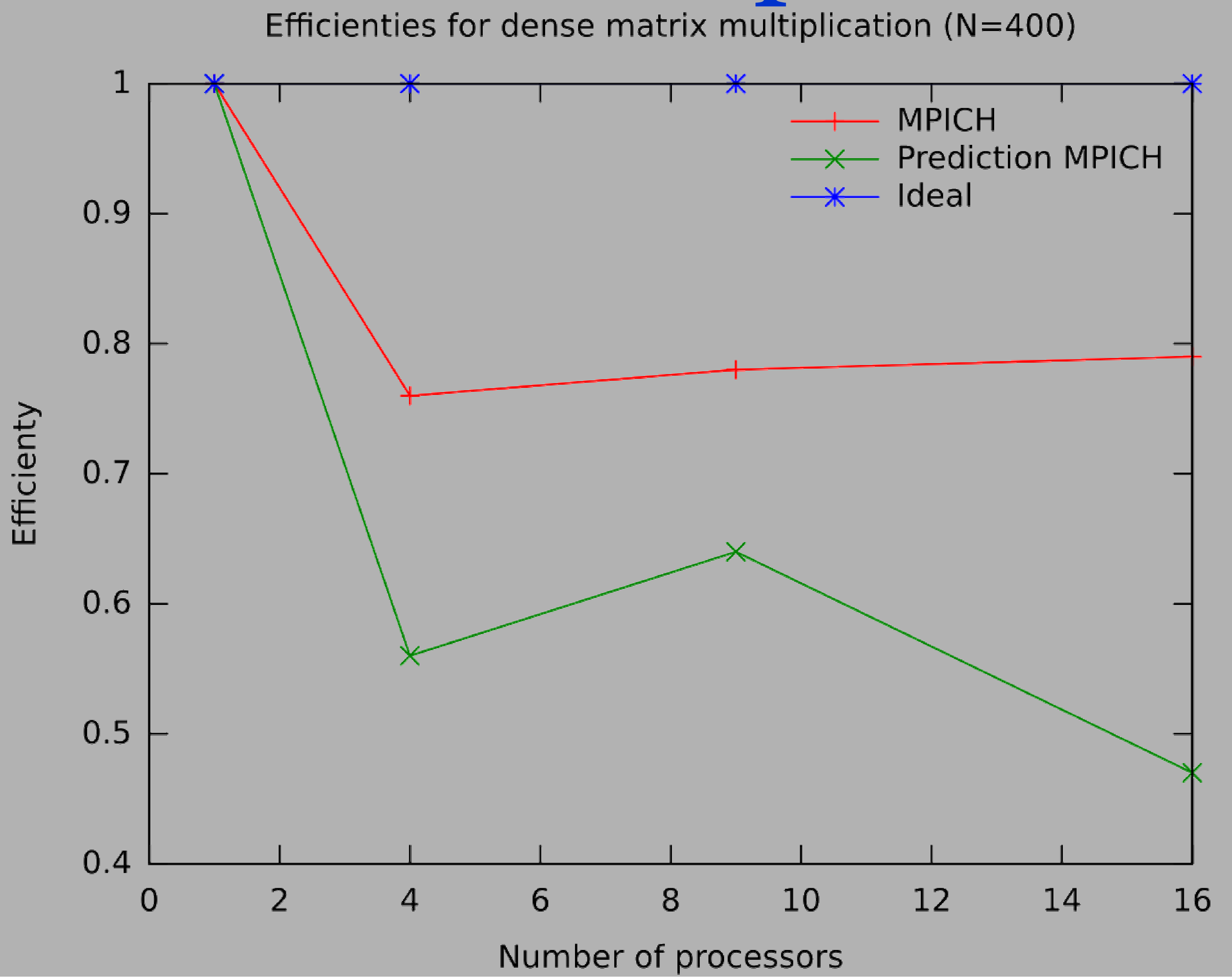
Bench and BSP predictions



Bench and BSP predictions



Bench s and BSP predictions



Data-Parallel Skeletons

Algorithm Skeletons

- Skeletons encapsulate basic parallel programming patterns in a well understood and structured approach
- Thus, skeletons are a set of functions which have 2 semantics : sequential and parallel ones.
- In general, skeletons work one list of data : a stream in the parallel semantics
- Typical examples : pipeline, farm, *etc.*
- Data-parallel skeletons are design for work on data and not one the stream of data
- Data-parallel skeletons has been design for lists, trees, *etc.*

Our Skeletons

- Work on lists : each processor has a sub-list
- Map : application of a function on list of data :

$$map\ f\ [x_1, x_2, \ldots, x_n] = [(f\ x_1), (f\ x_2), \ldots, (f\ x_n)]$$

$$mapidx\ f\ [x_1, x_2, \ldots, x_n] = [(f\ 1\ x_1), (f\ 2\ x_2), \ldots, (f\ n\ x_n)]$$

- Zip : combines elements of two lists of equal length with a binary operation :

$$zip\ \oplus\ [x_1, \ldots, x_n]\ [y_1, \ldots, y_n] = [x_1 \oplus y_1, \ldots, x_n \oplus y_n]$$

- Reduce and scan
- Rpl : creates a new list containing n times element x

Distributable Homomorphism

- Dh : know as butterfly skeleton and used to express a special class of divide-and-conquer algorithms
- Récursive définition :

$$dh \ \oplus \ \otimes \ [x_1, \ldots, x_n] = [y_1, \ldots, y_n]$$

- where :

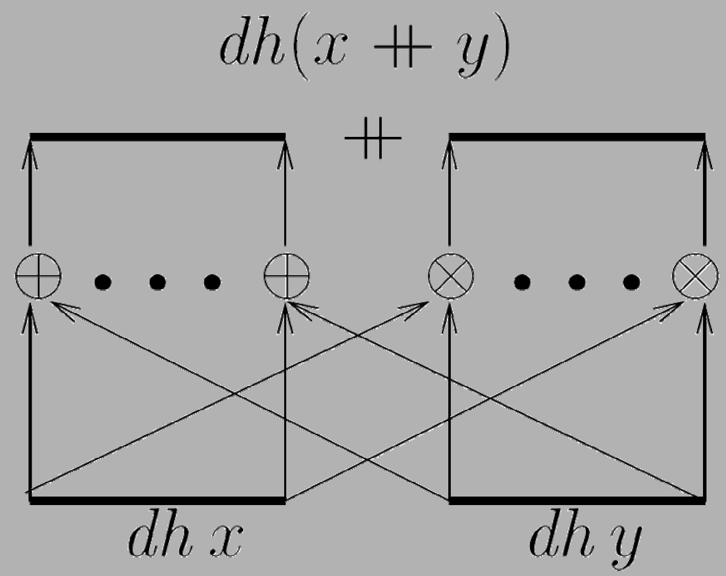
$$y_i = \begin{cases} u_i \oplus v_i & \text{if } i \leq n/2 \\ u_{i-n/2} \otimes v_{i-n/2} & \text{otherwise} \end{cases}$$

- and

$$u = dh \ \oplus \ \otimes \ [x_1, \ldots, x_{n/2}]$$
$$v = dh \ \oplus \ \otimes \ [x_{n/2+1}, \ldots, x_n]$$

Distributable Homomorphism

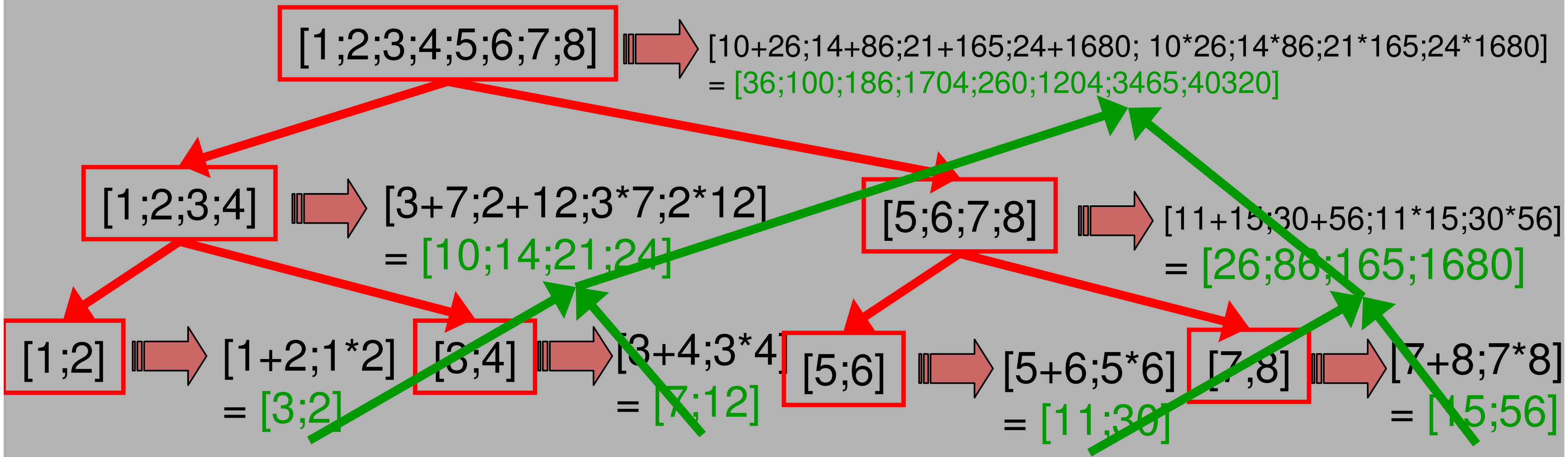
- The butterfly (if x and y lists of data) :



- Which is also a parallel point of view…

Example dh

dh + × [1;2;3;4;5;6;7;8]



Parallel Implementation

- Currently, naive implementation : suppose 2^l processors (even, you need to manage bording data)
- Recursive implementation using superposition
- BSP Cost = logarithmic number of super-step with at most 2^(l-p) data communicated
- Application : Fast Fourier Transformation (FFT) and Tridiagonal System Solver (TDS)

Code of Dh

```
let dh oplus omult fl =
 let rec tmp n1 n2 n vec =
  if n=1 then vec else
   let n'=n/2 in
   let n1'=n1+n' and n2'=n1+n'−1 in
   let vec'= super_mix (n1'−1) (super (fun () →tmp n1 n2' n' vec)
                                          (fun () →tmp n1' n2 n' vec)) in
   let msg=mkpar (fun pid v →
                   if pid<n1'
                    then (fun dst →if dst=(pid+n') then Some v else None)
                   else (fun dst →if dst=(pid−n') then Some v else None))
 in
   let send=put (apply msg vec') in
   let rcv = mkpar (fun pid f a →if pid<n1'
                          then match (f (pid+n')) with
                                 Some b →List.map2 oplus a b
                               | None →a
                           else match (f (pid−n')) with
                                 Some b →List.map2 omult b a
                               | None →a) in
   apply2 rcv send vec'
  in (tmp 0 (bsp_p()−1) (bsp_p()) (parfun (local_dh oplus omult) fl
```

Fast Fourier Transformation

Presentation

- Usefull in many numeric applications
- Définition (n=2^l) :

$$\mathbf{fft}\ [x_1, \ldots, x_n] = [y_1, \ldots, y_n]$$

$$\text{where } y_i = \sum_{k=0}^{n-1} x_k \omega_n^{ki}$$

$$\text{and } \omega_n = e^{2\pi\sqrt{-1/n}}$$

Skeleton implementation

- Recursive computation :

$$y_i = (\mathbf{FFT}\ x)_i = \begin{cases} (\mathbf{FFT}\ u)_i \ \oplus_{i,n}\ (\mathbf{FFT}\ v)_i & \text{if } i < n/2 \\ (\mathbf{FFT}\ u)_{i-n/2}\ \otimes_{i-n/2,n}\ (\mathbf{FFT}\ v)_{i-n/2} & \text{otherwise} \end{cases}$$

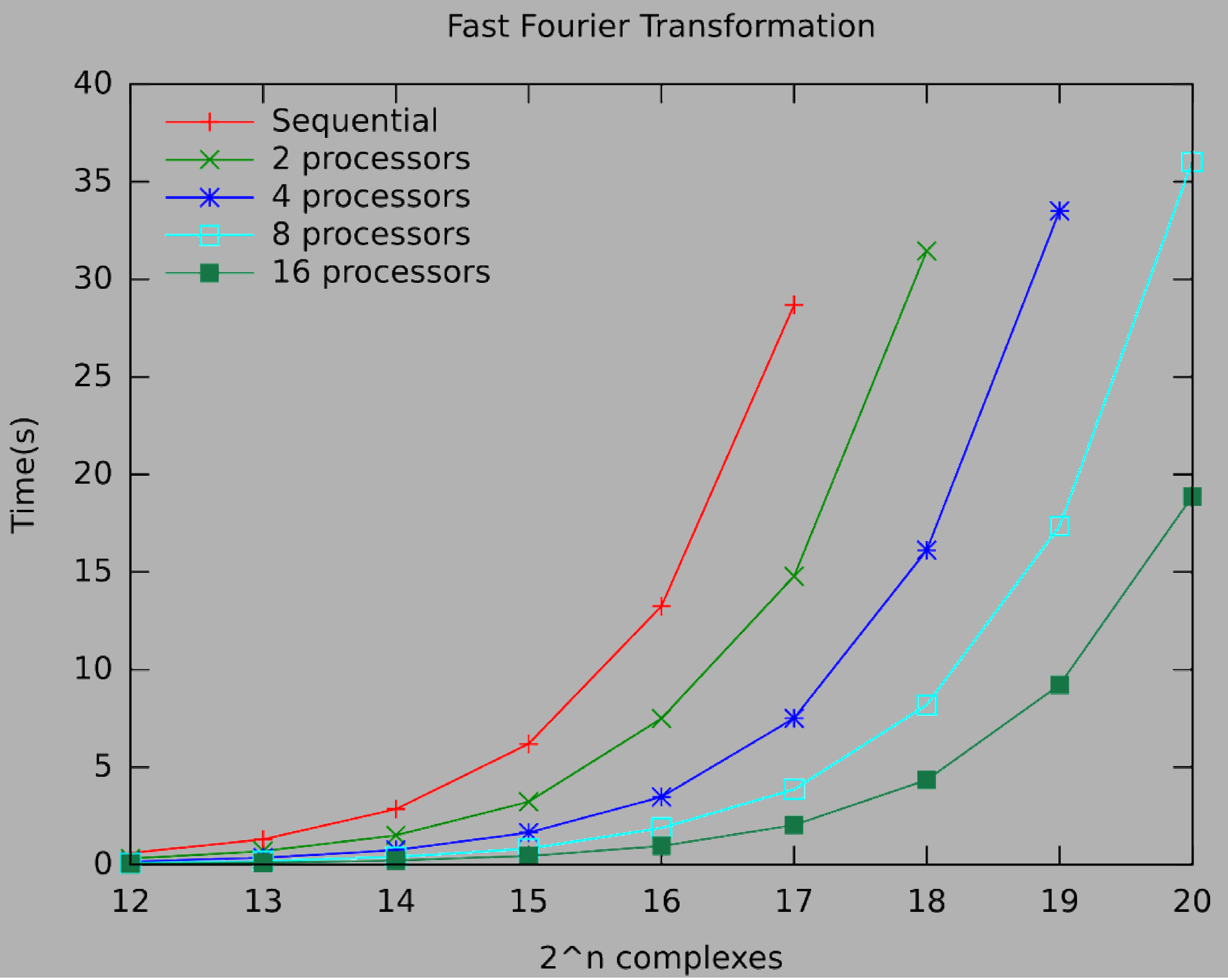
where $a \oplus_{i,n} b = a + \omega_n^i b$ and $a \otimes_{i,n} b = a - \omega_n^i b$

- where $u = [x_0, x_2, \ldots, x_{n-2}]$ and $v = [x_1, x_3, \ldots, x_{n-1}]$
- Operator : $\begin{pmatrix} x_1 \\ i_1 \\ n_1 \end{pmatrix} \oplus \begin{pmatrix} x_2 \\ i_2 \\ n_2 \end{pmatrix} = \begin{pmatrix} x_1 \oplus_{i_1,n_1} x_2 \\ i_1 \\ 2n_1 \end{pmatrix}$
- Skeleton code :

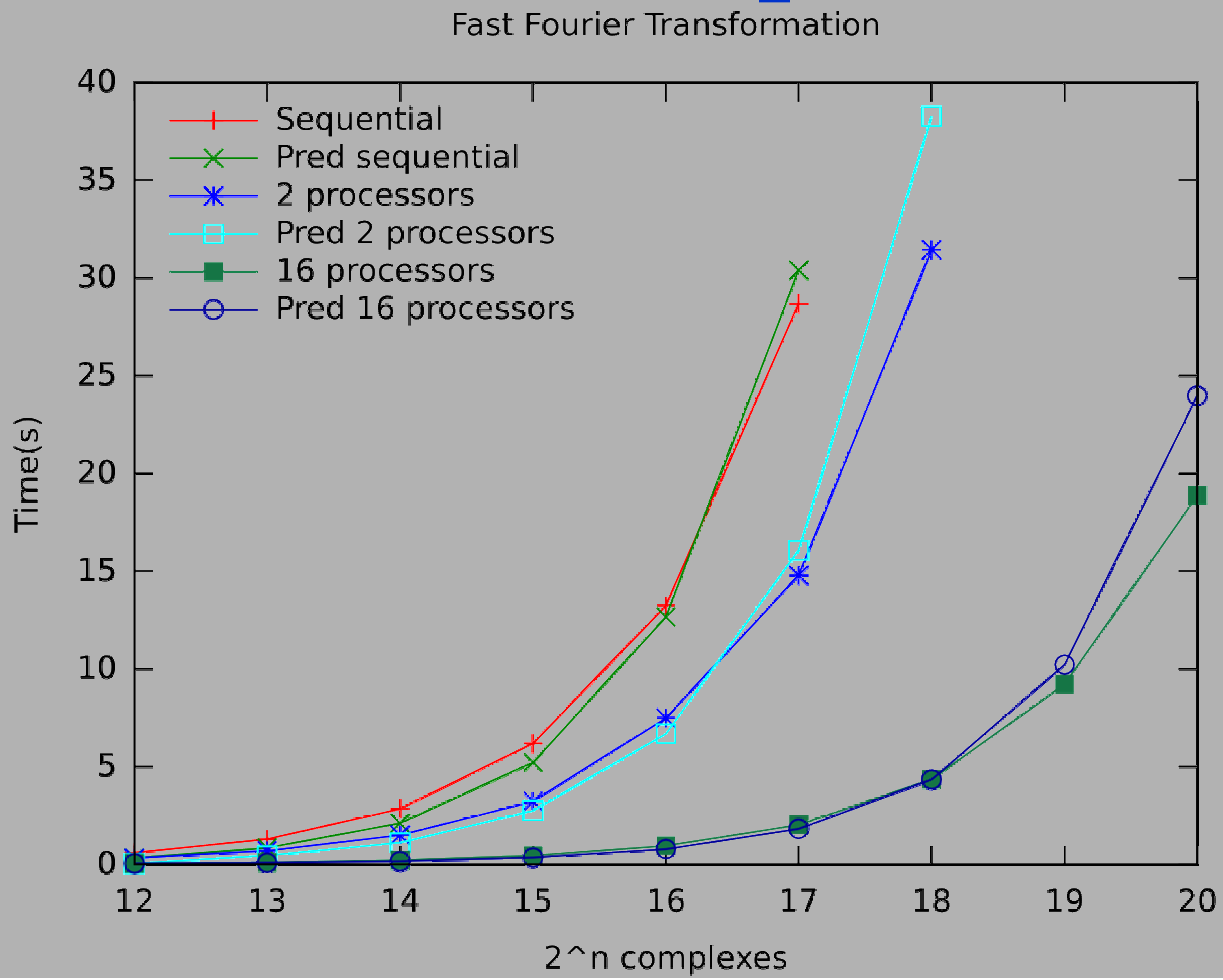
$$\mathbf{let\ fft\ l} = map\ \mathbf{fst}\ (dh \oplus \otimes\ (mapidx\ \mathbf{triple}\ l))$$

- where : $\mathbf{triple}\ x_i = (x_i, i, 1)$

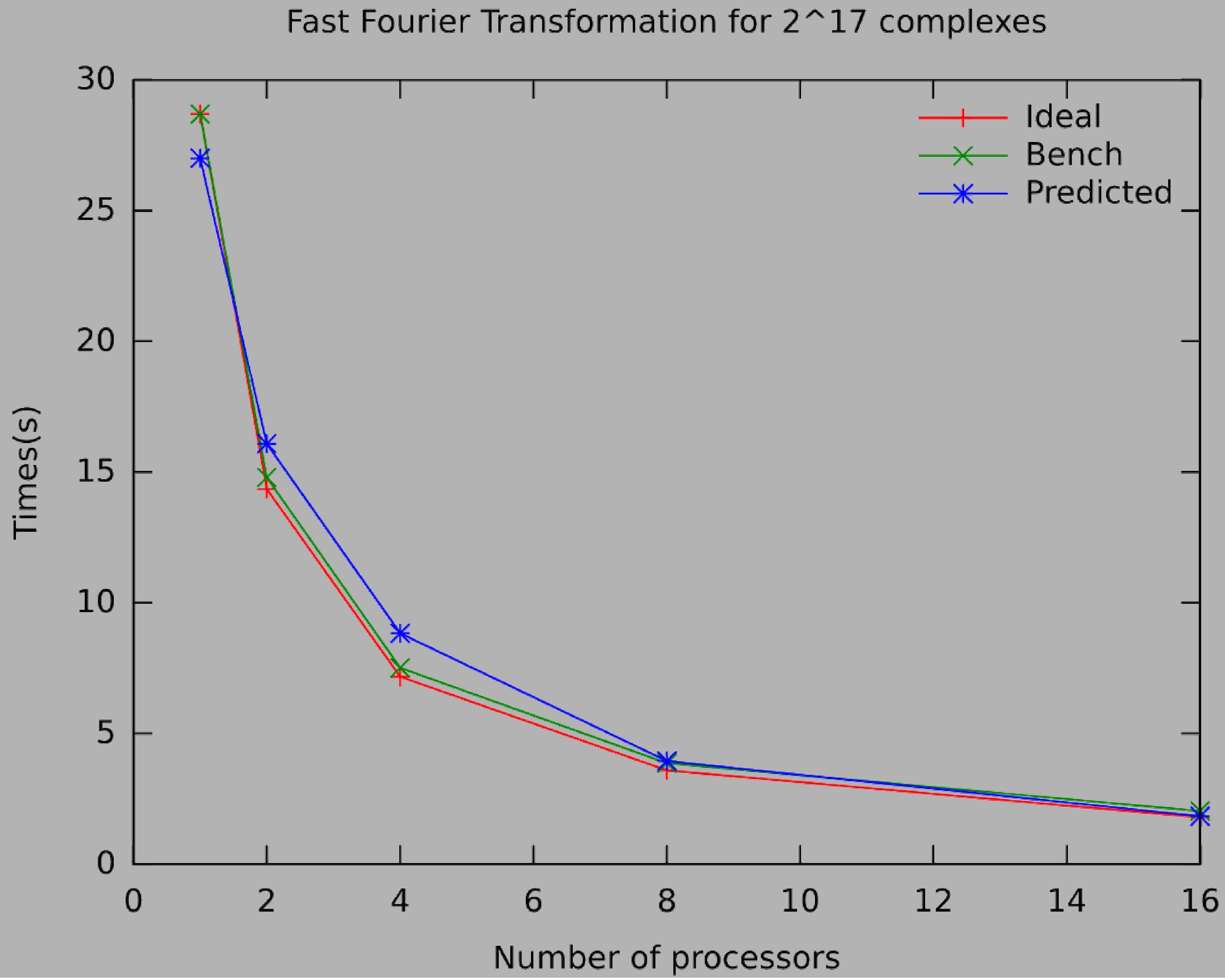
Benchs and BSP predictions



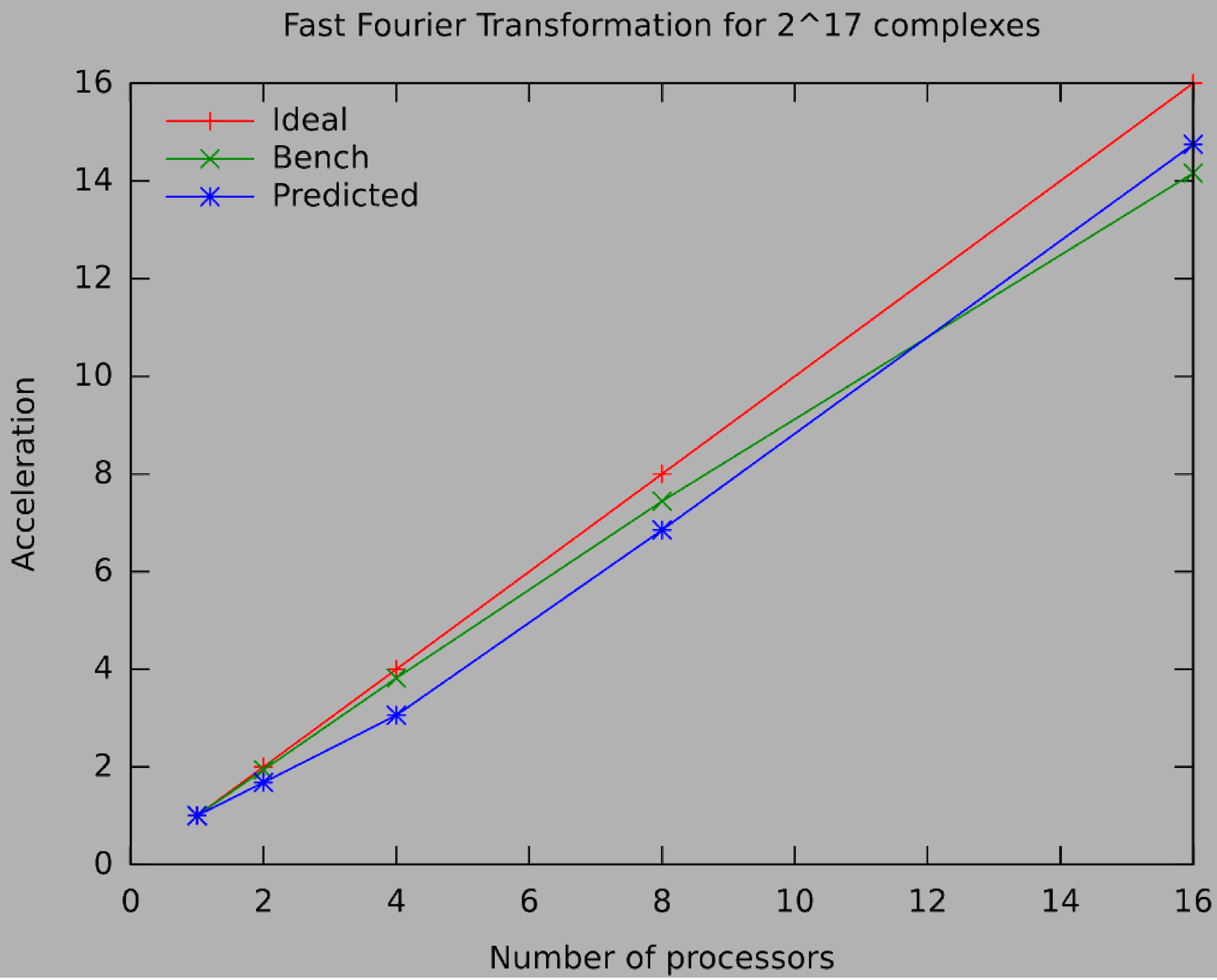
Benchs and BSP predictions



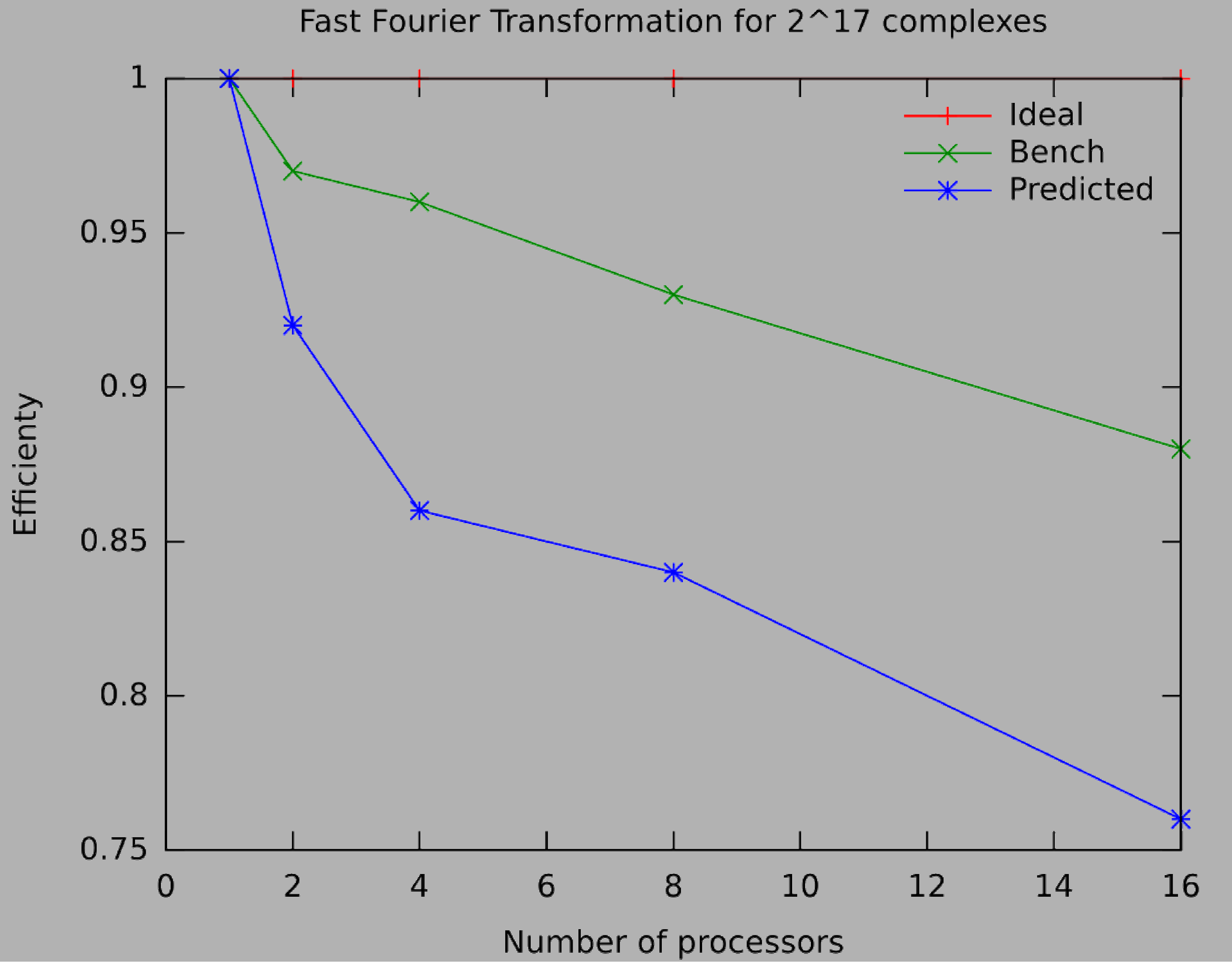
Bench and BSP predictions



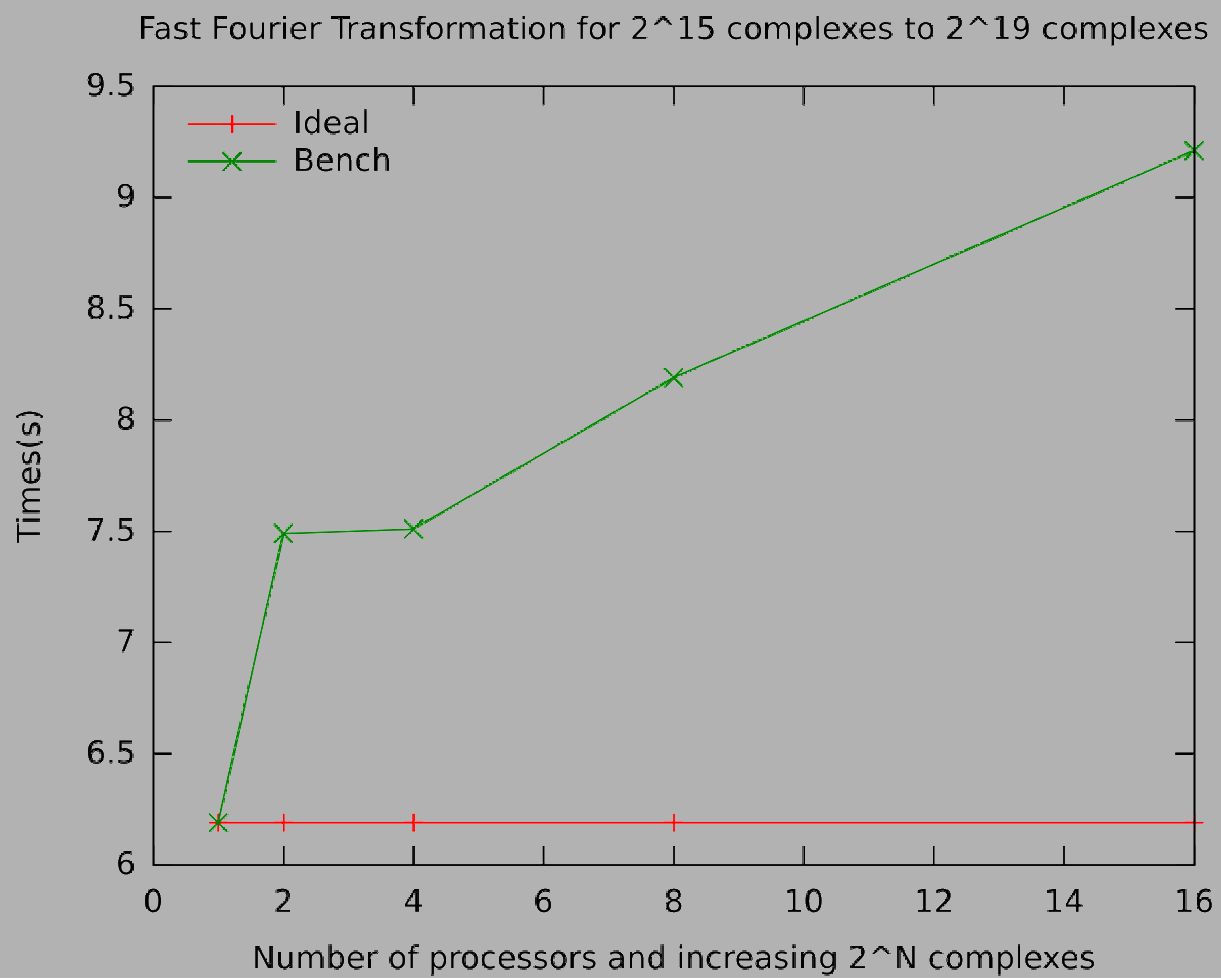
Benchs and BSP predictions



Benchs and BSP predictions



Bench and BSP predictions

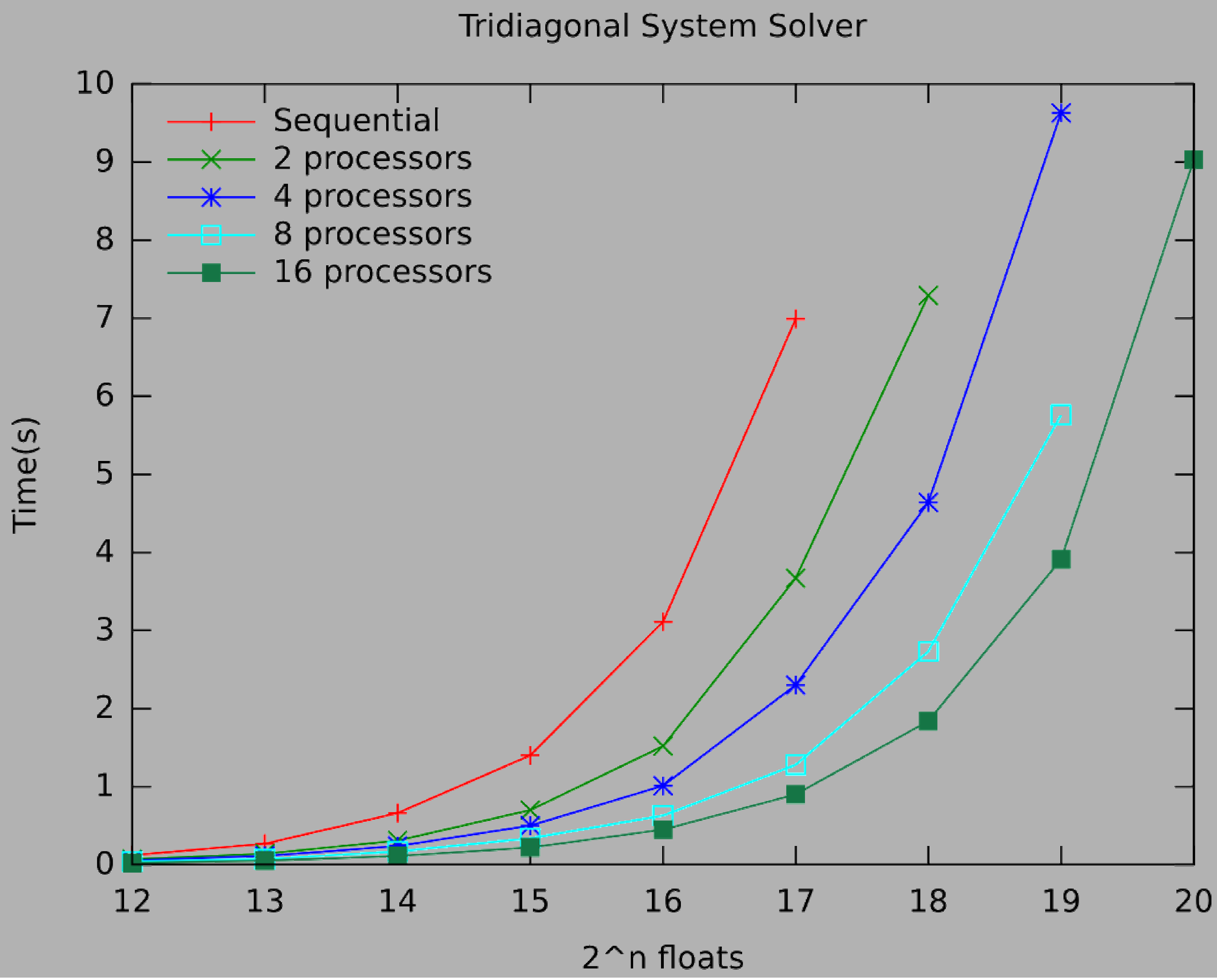


Tridiagonal System Solver

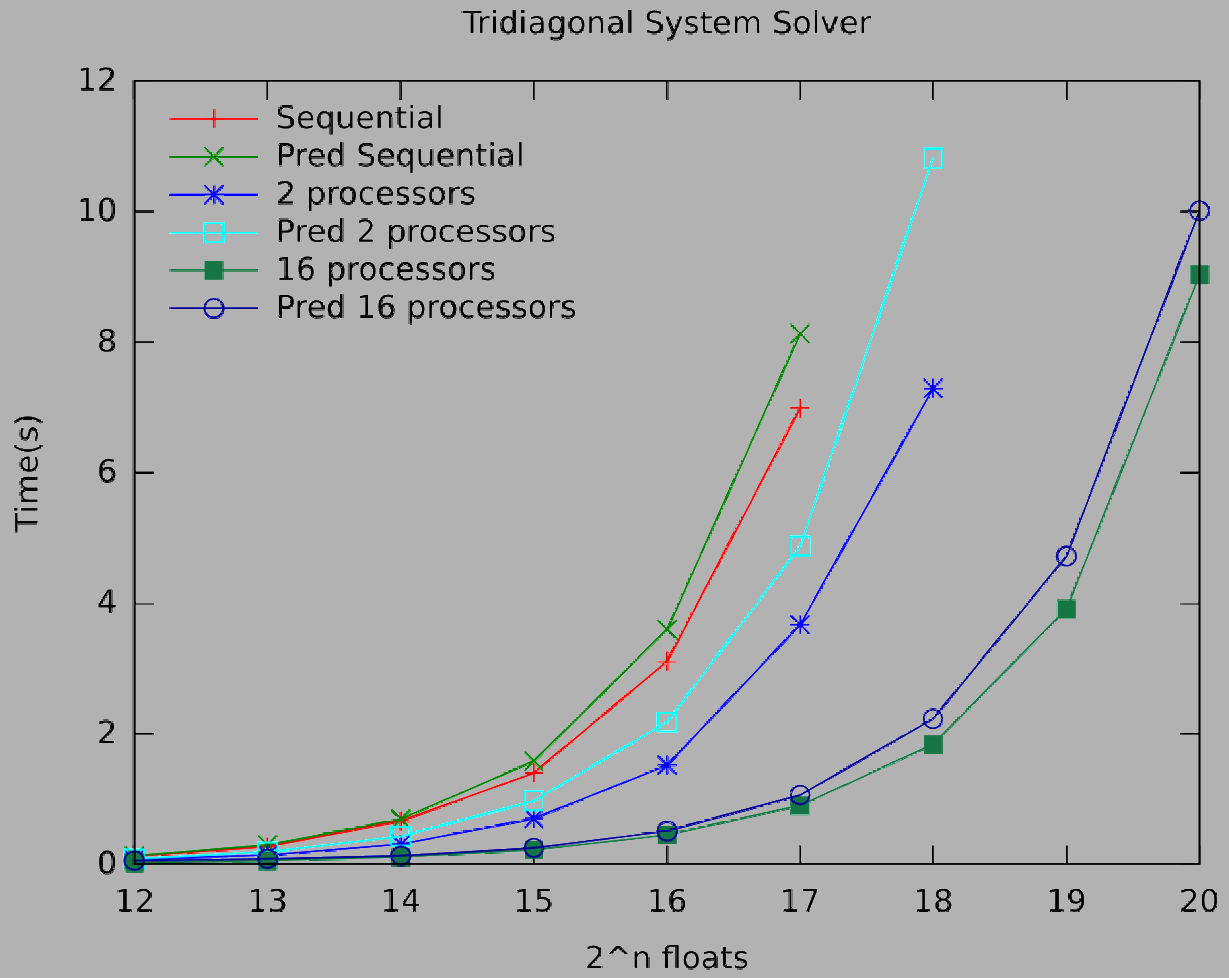
Presentation

- Usefull in many applications
- A×x = b where A is sparce matrix representing coefficients, x a vector of unknowns and b a right-hand-side vector.
- The only values of A unequal to 0 are on the main diagonal, as well as directly above and below it
- Can be implemented using dh as FFT but just other operators…

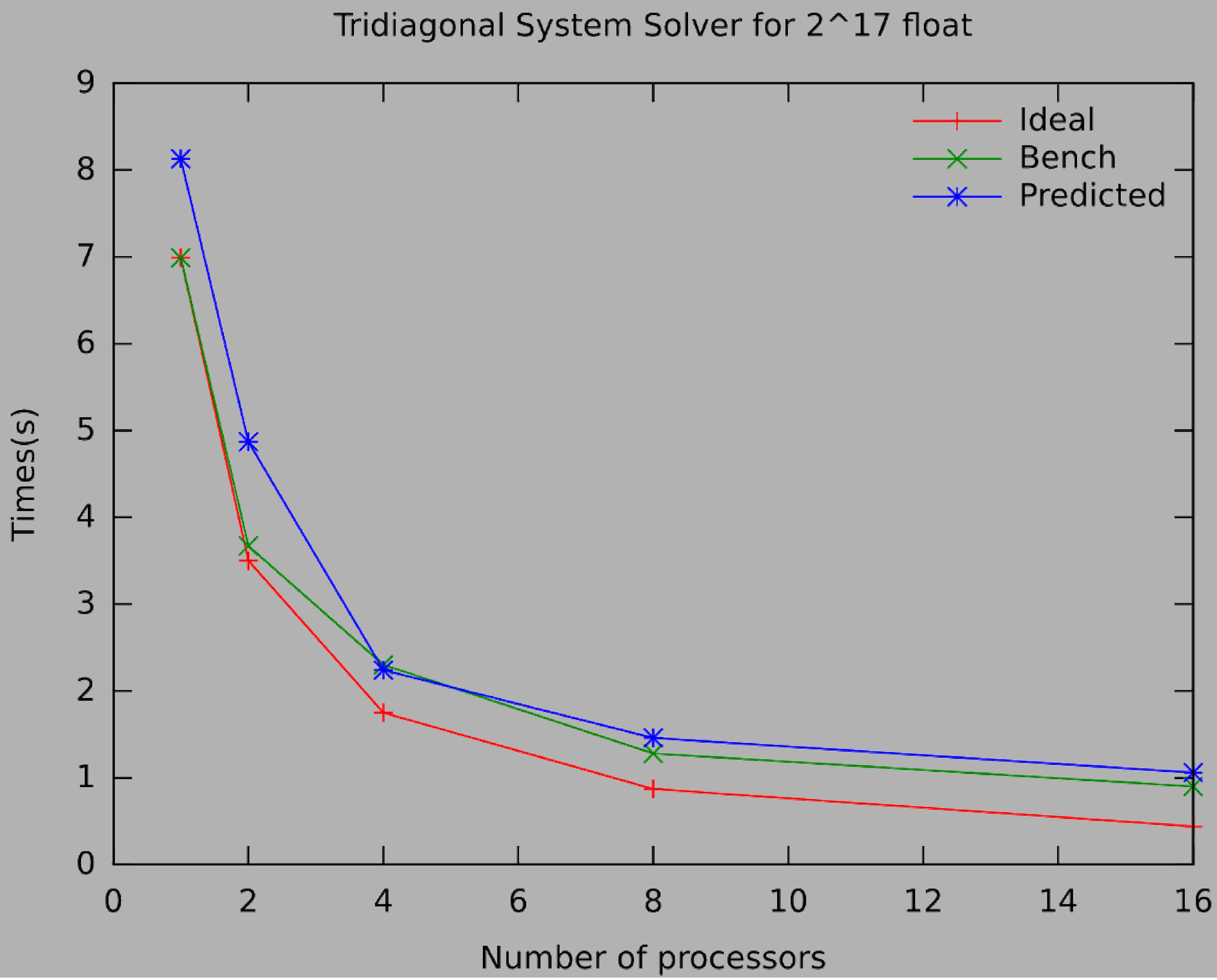
Bench and BSP predictions



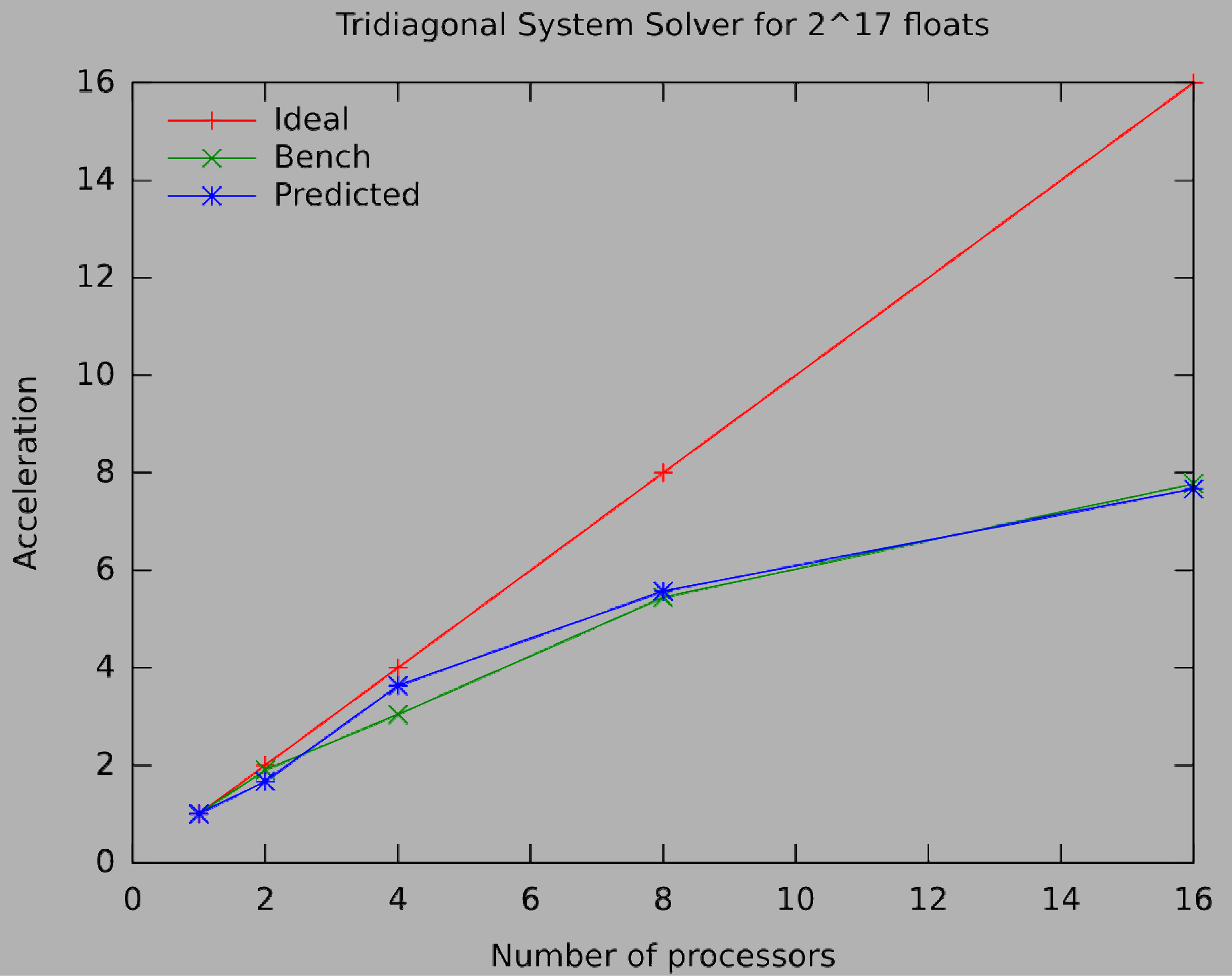
Benchs and BSP predictions



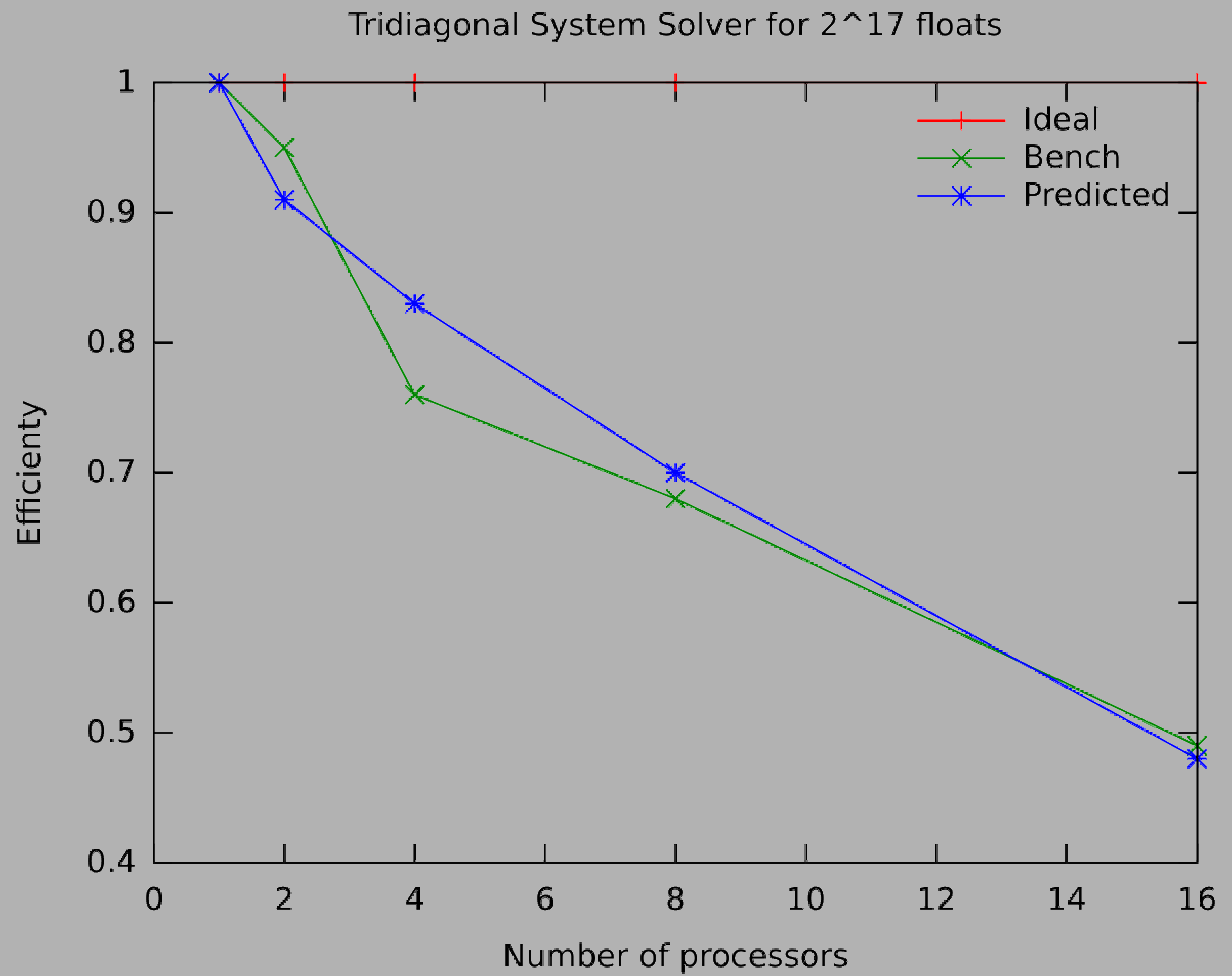
Bench and BSP predictions



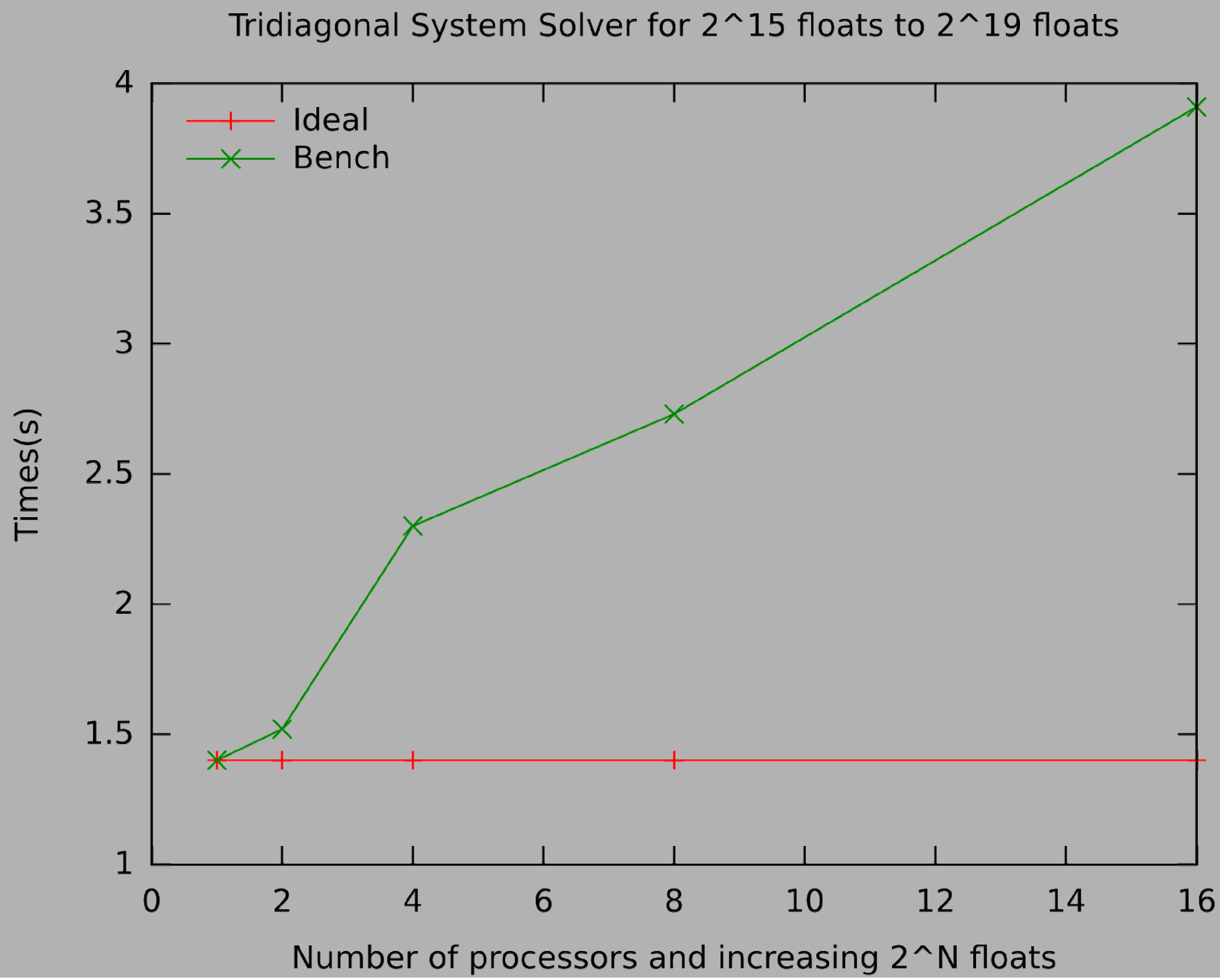
Benchs and BSP predictions



Benchs and BSP predictions



Bench and BSP predictions



Conclusion and future works

Conclusion

- BSML=BSP**+**ML
  - **safe high-level** parallel language
  - **Easy** to write parallel programs
  - allow a **cost based methodology**
- Some **typical example**, data-parallel **skeletons** and **benchs**
- Many work on **operational semantics** to ease properties

What is « off »

- As a library for O'Caml, BSML has many lacks of safety :
  - nested of parallel vector is allow
  - Problem of determinism with some side effects
  - some functions of O'Caml standard library can break the model of execution
  - …
- Need of a full language :
  - new type system (ongoing work)
  - Implementation using **continuation** (transformation of source's code with the help of a type checker) for the superposition (ongoing work)
  - create our own standard library to delete « dangerous functions » (easy but boring work)

Future works

- Implementation of parallel skeletons (management of tasks) using the superposition ?

- **Implementation** of bigger algorithms for better benchmarks of BSML

  - BSP **model-checking** of high-level **Petri-nets** (M-nets). The main difficult : find a non-trivial algorithm as the community of concurrent programming does. Possible but need more theoretical optimisations…

  - Libraries for matrices (by Sovanna Tan) and graphs (ongoing work)

  - More symbolic computations...(Knuth-Bendix, on going work)

- **PROPAC ("PROgrammation PArallèle Certifiée")**

Thanks for your attention