Note Taker Checklist Form -*MSRI*

Name : Rob Stapleton

E-mail Address/ Phone #: jrstaple@ncsu.edu

Talk Title and Workshop assigned to:
Interactive Parallel Computation in Support of Research in Algebra, Geometry and Number Theory

Lecturer (Full name): Jean-Louis Roch

Date & Time of Event: 11:30 a.m. Jan 29, 2007

Check List:

(✓) Introduce yourself to the lecturer prior to lecture. Tell them that you will be the note taker, and that you will need to make copies of their own notes, if any.

(✓) Obtain all presentation materials from lecturer (i.e. Power Point files, etc). This can be done either before the lecture is to begin or after the lecture; please make arrangements with the lecturer as to when you can do this.

(✓) Take down all notes from media provided (blackboard, overhead, etc.)

(✓) Gather all other lecture materials (i.e. Handouts, etc.)

(✓) Scan all materials on PDF scanner in 2nd floor lab (assistance can be provided by Computing Staff) – Scan this sheet first, then materials. In the subject heading, enter the name of the speaker and date of their talk.

Please do **NOT** use **pencil** or colored pens other than black when taking notes as the scanner has a difficult time scanning pencil and other colors.

Please fill in the following after the lecture is done:

1. List 6-12 lecture keywords: work-share, work-first, work-steal, parallel, processor-oblivious, task farm, KAAPI, Athapascan

2. Please summarize the lecture in 5 or less sentences.
With the proper scheme, many sequential algorithms can be parallelised. Additional work-first and work-stealing schemes can be added to increase the performance of sequential-but-parallelizable and parallelized algorithms.

Once the materials on check list above are gathered, please scan ALL materials and send to the Computing Department. Return this form to Larry Patague, Head of Computing (rm 214)

11:30 a.m.
Processor-Oblivious Parallel Processing with Provable Performances
Jean-Louis Roch

Overview:
-Introduction
-Machine model and work-stealing
-Cheme 1 Adaptive Parallel algorithms
-Scheme 2 Nano-loop
-Scheme 3 Amoritizing the overhead of parallelism
-Putting things together

Interactive Parallel Computation:
Any application is "parallel":
-composition of several programs / library procedures (possibly concurrent)
-each procedure written independently and also possibly parallel, itself

Parallel Interactive Application:
-Human in the loop
-Prallel machines (cluster) to enable large interactive applications
-Two main performance criteria:
	-Frequency (refresh rate)
		-Visualization: 30-60Hz
		-Haptic: 1000Hz
	-Latency (makespan for one iteration)
		-object handling: 75 ms

New Parallel Supports (from small to large)
-Multi-core architectures
	-Dual Core procssors
	-Dual Core graphics processors
	-Heterogeneous multi-cores
	-MPSoCs
-Commodity SMPs
	-8-way PCs equipped with multicore processors (AMD Hypertransport) + 2 GPUs
-Clusters
	-72% of top 500 machines
	-Trends: more processing units, faster networks (PCI-Express)
	-Heterogeneous (CPUs, GPUs, FPGAs)
-Grids
	-Heterogeneous networks
	-Heterogeneous administration policies
	-Resource volatility
-Virtual Reality / Visualization Clusters
	-Virtual Reality, Scientific Visualization and Computational Steering
	-PC clusters + graphics cards + multiple I/O devices (cameras, 3D trackres, etc.)

Parallel induces overhead: E.G. Parallel prefix on fixed architecture
Prefix problem: Input a_0, ..., a_n. Output : sequential products.
Serial performs only n opreations, serial performs 2n
but 2*log(n) time.
Optimal time T_p = 2n/(p+1) but performs 2np/(p+1) operations.

Lower Bound for the Prefix: look at the multiplication circuit as a binary tree

The problem: To design a single algorithm that computes efficiently

prefix (a) on an aritrary dynamic architecture

Dynamic Architecutre: non-fixed number of resources, varible speeds.
e.g. grid, ... but not only: SMP server in multi-users mode.

Processors-Oblivious Algorithms -- that's what what we want (?)

Machine Model and Work Stealing
-Heterogeneous machine model and work-depth framework
-Distributed work stealing

Heterogenous Processors, work and depth:
Processor speeds are assumed to change arbitrarily and adversarily.
Model [Bender, Rabin 02] PI_i(t) = instantaneous speed
of processor i at time t (in #unit operations per second). Assumption:
PI_{max}< C*PI_min(t)
Definition: for a computation with duration T:
Total speeD: PI_{tot} = sum(sum(PI_i(t), t=0..T), i=0..P)
Average Speed per processor: PI_{ave} = PI_{tot}/P
Work: W = #total number of operations performed
Depth: D = $operations on a critical path (~parallel "time" on
infinite resources)
For any greedy maximum utilization schedule: makespan <= W/(p*PI_{ave})
+ (1-1/p)*(D/PI_{ave})

The Work Stealing Algorithm:
A distributed and randomized algorithm that computes a greedy schedule:
-Each processor manages a local stack (depth-first execution)
-When idle, a processor steals the topmost task on a remote non-idle
victim processor (randomly chosen)
-Theorem: With good probability,
-#steals < P*D
-execution time <= W/(p*PI_{ave}) + O(D/Pi_{ave})
-Interest: If W indepenent of p and D is small, work stealing achieves
near-optimal schedule.

Work Stealing Implementation:
efficient policy (close to optimal) <---- scheduling ---> control of the
policy (realisation)
Difficult in general (coarse grain) Expensive in
general (fine grain)
But easy if D is small But small overhead
if small number of tasks
Execution time as above (fine grain)
(coarse grain)

If D is small, a work stealing algorithm performs a small number of
steals
=> Work-first principle: "Scheduling overheads should be borne by
the critical path of the computation"[Firggo 98]
Implementation: since all tasks but a few are executed in the local
stack, overhead of task creation should be as close as
possible as sequentail function call
At any time on any non-idle processor, efficient local degeneration of
the parallel program in a sequential execuation

Work Stealing implementaions following the work-first principle: Cilk
-Cilk-5 http://supertech.csail.mit.edu/cilk: C extension
-Spawn f(a); sync (serie-parallel programs)
-Requires a shared-memory machine
-Depth-first execution with synchronization (on sync) with the end

```
of a task:
            -spawned tasks are pushed in double-ended queue
      -"Two-clone" compliation strategy [Frigo-Leiserson-Randal 98]
            -On a successful steal, a thief executes the continuation on
the topmost ready task
            -When the continuation hasn't been stolen, "sync" = nop;
else synchronization with its thief
      -Won the 2006 award "Best Combination of Elegance and Performance"

Work Stealing implementaions following the work-first principle:  KAAPI
Kaapi/Athapascan  http://kaapi.gforge.inria  C++ library
      -Fork<f>()(a,...) with access mode to parameters (value, read,
write, r/w, cw) specified in f prototype (macro
dataflow programs)
      -Supports distributed and shared memory machines; heterogeneous
processors
      -Depth-first (reference order) execution with synchronization on
data access
            -Double-end queue ( mutual exclusion with compare-and-swap )
            -more

N-queens:  Takaken C sequential code parallelized in C++/Kaapi
-T. Gautier & S. Guelton won the 2006 "Prix special du Jury" for the
best performance at NQueens contest.
-Some facts[on Grid'5000, a grid of processors of heterogeneous speeds]
      -NQueens(21) in 78s on about 1000 processors
      -NQueens(22) in 502.9s on 1458 processors
      -NQueens(23) in 4435s on 1422 processors [~24 * 10^33 solutions]
      -0.625% idle time per processor
      -<20s to deploy up to 1000 processes on 1000 machine [Taktuk,
Huard]
      -15% more

Work first principle and adaptability
-Work-first principle:  Implicit dynamic choice between two executions:
      -a sequential "depth-first" execution of the parallel algorithm
(local, default)
      -a parallel "breadth-first" one

Extended work-stealing:  How do we get W_1 and W_{infinity} small?
Concurrently sequential and parallel
Based on the work-stealing and the work-first principle:
      Instead of optimizing the sequential execution of the best
parallel algorithm, let's optimize the parallel execution
      of the best sequential algorithm
Excecute always a sequential algorithm to reduce prallelism overhead
      parallel algorithm is used only if a processor becomes idle (i.e.
workstealing) to extract parallelism from the
      remaining work a sequential computation
Assumption: Two concurrent algorithms that are complimentary:
      -one sequenial (always performed, the priority
      -the other parallel

Extended work-stealing and granulairy
-Scheme of the seuqential process: nanoloop
      While (not completed(Wrem) ) and (next_operation hasn't been
stole){
            atomic {extract_next k operatoins ; Wrem -= k ;}
            process the k operations extracted ;
      }
-Processor-oblivious algorithm:
```

-Whatever p is, it performs O(p*D) preemption operations (continuation faults) -> D should be as small as possible to maximize both speed-up and locality

Interactive application with time ocnstraint
-Anytime algorithm:
    -an be stopped at any time (with a result)

Amortizing the arithmetic of parallelism
Adaptive scheme: extract_seq/nanoloop // extract_par
    -ensures an optimal number of operations on 1 processor but no guarantee of the work preformed on p processors

E.G. (C++ STL): find_if(fist, last, predicate)
locates the first element in [First,Last) verifying the predicate

This may be a drawback:
    -unneded processor usage
    -undesirable for a library code that may be used in a complex application with many components
    -(or not fair with other users)
    -increases the time of the application: any parallelism that increases execution time should be avoided

Similar to the nano-loop for the sequential processes: that balances the -atomic- local work by the depth of the remaining one.
Here, by amortizing the work induced by the extract_par operation, ensuring this work to be small enough:
    -either wrt the useful work already performed
    -or with respect to the useful work yet to be performed (if known)
    -or both

E.G.: find_if(First, Last, predicate):
    -only the work already performed is known (on-line)
    -then prevent to assign more than alpha(W_{done}) operations ot work-stealers
    -Choices for alpha(n):
        -n/2 similar to Floyd's iteration (approximation ratio = 2)
        -n/log(n) : to ensure optimal usage of the work-stealers

Putting things together: Processor -oblivious prefix computation

The critical path is put onto the parallel algorithm
Analysis:
Execution Time <= 2n/((p+1)*PI_{ave}) + O(log(n)/PI_{ave})

Conclusion:
-fine grain parallelism enables efficient execution on a small number of processors
-Efficiency of classical work stealing relies on work-first principle
-Processor Oblivious algorithms based on work-stealing/Work-first principle
    -Based on anytime extraction of parallelism from any sequential algorithm (may execute different amts of operations)
    -Oblivious: near optimal whatever the execution context is.
-Generic scheme for stream computations:
    -parallelism introduce a copy overhead from local buffers to the output gzip/compression, MPEG-4/H264

Processor-oblivious parallel algorithms with provable performances

-

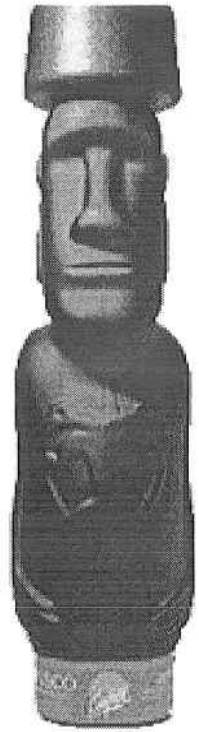
Applications

Jean-Louis Roch

MOAIS

Lab. Informatique Grenoble, INRIA, France

Overview

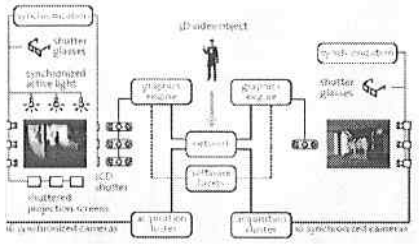


- Introduction : interactive computation, parallelism and processor oblivious
  - Overhead of parallelism : parallel prefix
- Machine model and work-stealing
- Scheme 1: Adaptive parallel algorithms
- Scheme 2: Amortizing the overhead of synchronization (Nano-loop)
- Scheme 3: Amortizing the overhead of parallelism (Macro-loop)
- Putting things together: processor-oblivious prefix computation

Interactive parallel computation?

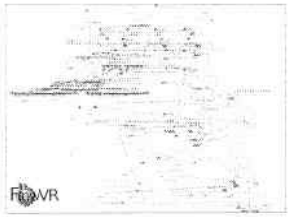
Any application is "parallel":

- *composition of several programs / library procedures (possibly concurrent,*
- *each procedure written independently and also possibly parallel itself.*



Interactive Distributed Simulation

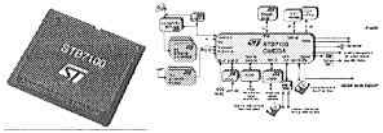
3D-reconstruction
+ simulation
+ rendering
[B Raffin &E Boyer]
- 1 monitor
- 5 cameras,
- **6** PCs

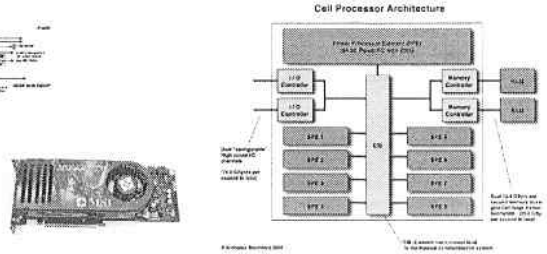


QuickTime[a] et un
dŽcompresseur codec YUV420
sont requis pour visionner cette image.

New parallel supports from small too large

Parallel chips & multi-core architectures:



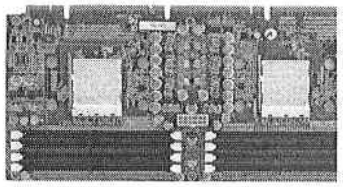


- **MPSoCs** (Multi-Processor Systems-on-Chips)
- **GPU** : graphics processors (and programmable: Shaders; Cuda SDK)

- Dual Core processors (Opterons, Itanium, etc.)
- Heteregoneous multi-cores : **CPUs + GPUs + DSPs+ FPGAs** (Cell)

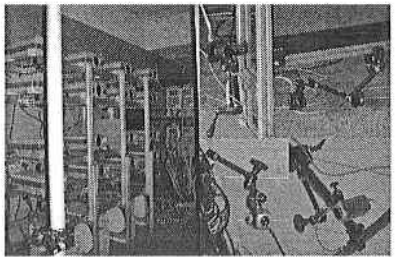
Commodity SMPs:

- 8 way PCs equipped with multi-core processors (AMD Hypertransport) + 2 GPUs



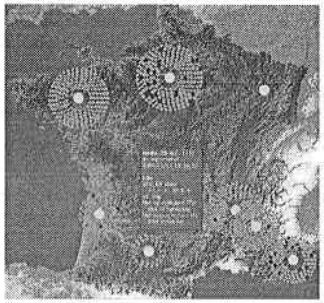
Clusters:

- 72% of top 500 machines
- Trends: more processing units, faster networks (PCI- Express)
- Heterogeneous (CPUs, GPUs, FPGAs)



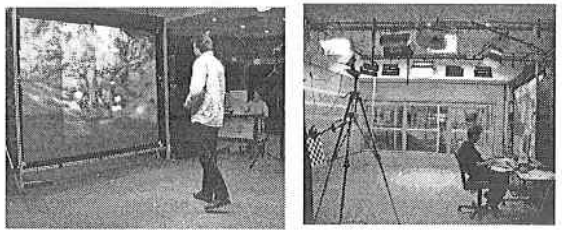
Grids:

- Heterogeneous networks
- Heterogeneous administration policies
- Resource Volatility



Dedicated platforms: eg Virtual Reality/Visualization Clusters:

- Scientific Visualization and Computational Steering
- PC clusters + graphics cards + multiple I/O devices
  (cameras, 3D trackers, multi-projector displays)



Grimage platform

Parallelism induces overhead : e.g. Parallel prefix on fixed architecture

- **Prefix problem** :
  - input : $a_0, a_1, \ldots, a_n$
  - output : $\pi_1, \ldots, \pi_n$ with

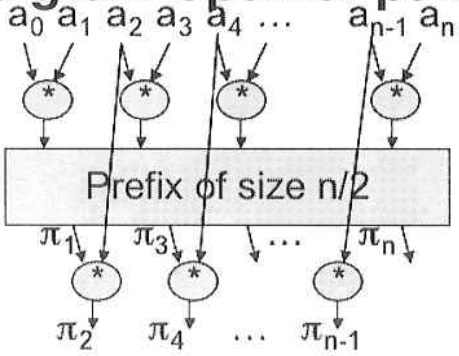
$$\pi_i = \prod_{k=0}^{i} a_k$$

- **Sequential** algorithm :
  - for ($\pi[0] = a[0]$, i = 1 ; i <= n; i++) $\pi[\,i\,] = \pi[\,i-1\,] * a[\,i\,]$

performs only n operation

- **Fine grain optimal parallel** algorithm :

ıdner-
ıher-81]



ɔal time = 2. *log* n
performs **2.n ops**

Parallel requires twice more operations than sequential !!

- Tight lower bound on **p identical processors**:

olau&al. 1996]

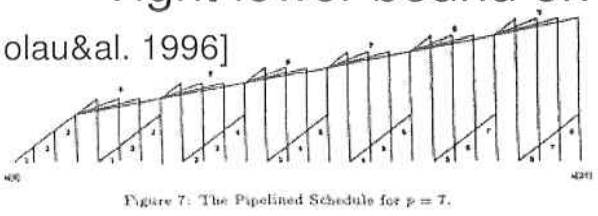


Figure 7: The Pipelined Schedule for $p = 7$.

Optimal time T_p = 2n / (p+1)
but performs **2.n.p/(p+1) ops**

Lower bound(s) for the prefix

Prefix circuit of depth d

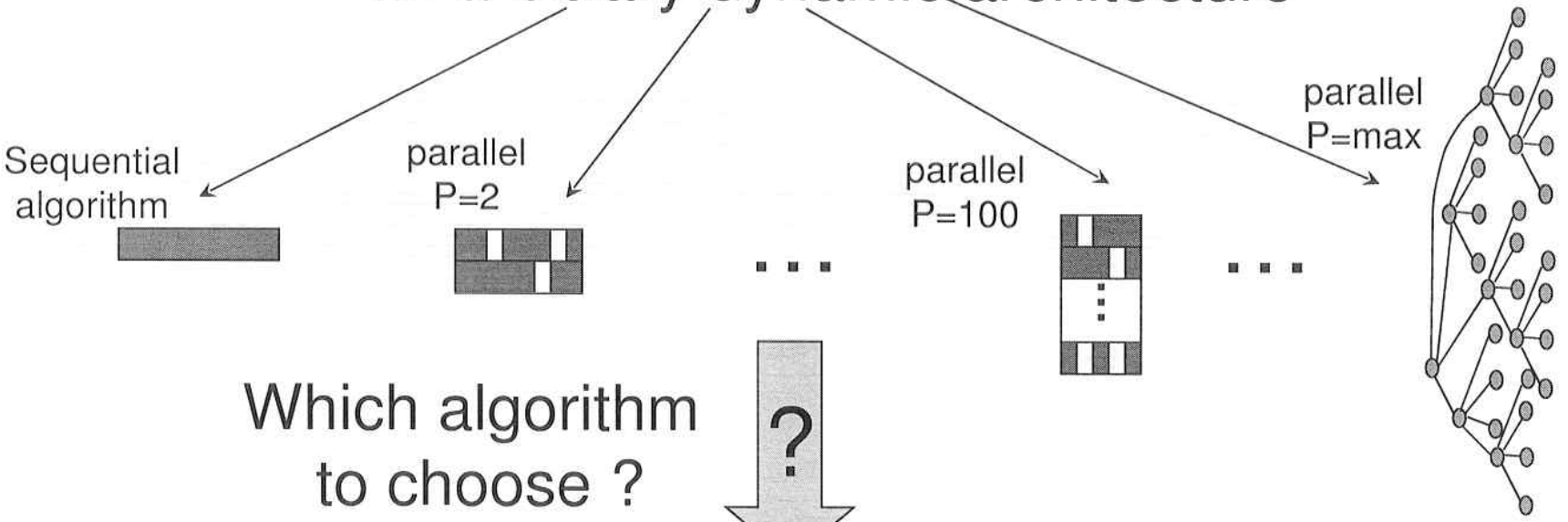
$\Downarrow$ [Fitch80]

#operations > $2n - d$

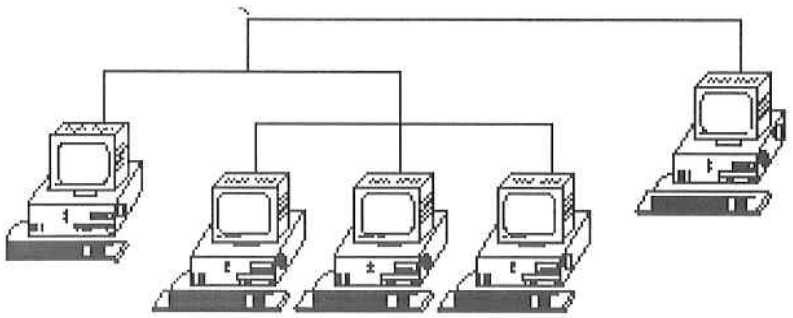
$$\text{parallel time} \geq \frac{2n}{(p+1).\Pi_{ave}}$$

he problem

› design a single algorithm that computes efficiently prefix(a) c
an arbitrary dynamic architecture



Which algorithm
to choose ?



Heterogeneous network

Multi-user SMP server

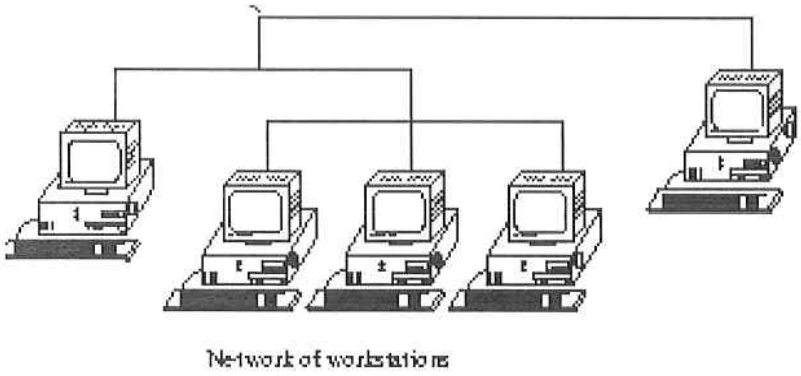
Grid

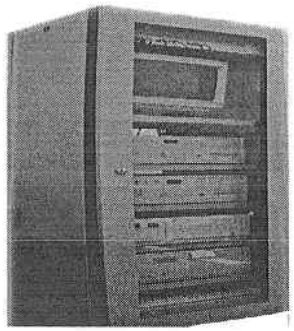
Dynamic architecture : non-fixed number of resources, **variable speeds**
eg: *grid*, … but not only: *SMP server in multi-users mode*

Processor-oblivious algorithms

Dynamic architecture : non-fixed number of resources, variable speeds

eg: grid, SMP server in multi-users mode,....





> motivates the design of «**processor-oblivious**» parallel algorithm that:

\+ is **independent** from the underlying architecture:

no reference to p nor $\Pi_i(t)$ = *speed of processor i at time t nor …*

\+ on a given architecture, has **performance guarantees** :

behaves as well as an optimal (off-line, non-oblivious) one

2. Machine model and work stealing

- Heterogeneous machine model and work-depth framework
- Distributed work stealing

- Work-stealing implementation : work first principle

- Examples of implementation and programs:
  Cilk , Kaapi/Athapascan

- Application: Nqueens on an heterogeneous grid

Heterogeneous processors, work and depth

Processor speeds are assumed to change arbitrarily and adversarially:

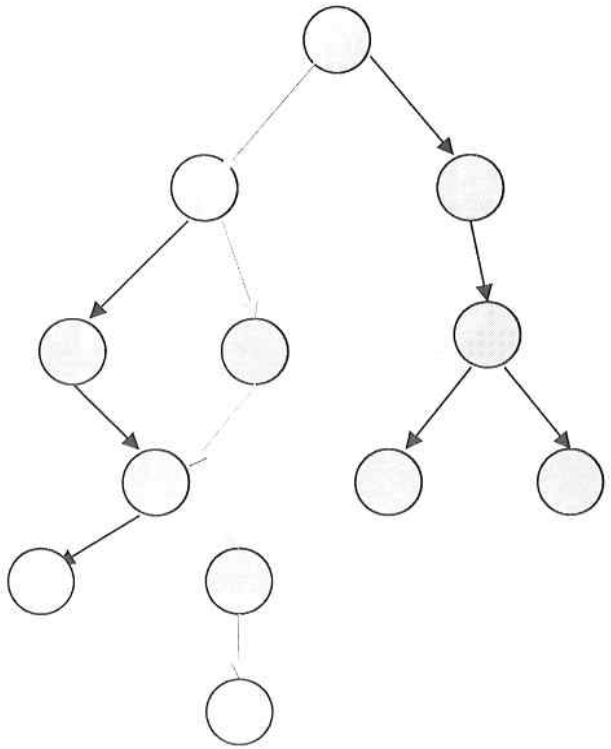
model [Bender,Rabin 02] $\Pi_i(t)$ = ***instantaneous speed*** *of processor i at time t*

(in #unit operations per second)

Assumption : $\Pi_{max}(t) < constant \cdot \Pi_{min}(t)$

Def: for a computation with duration T

- **total speed:** $\Pi_{tot} = \Sigma_{i=0,..,P} \Sigma_{t=0,..,T} \Pi_i(t)$
- **average speed** per processor: $\Pi_{ave} = \Pi_{tot} / P$



"**Work**" W = #total number operations performed

"**Depth**" **D** = #operations on a critical path

(~parallel "time" on ∞ resources)

For any greedy *maximum utilization* schedule:

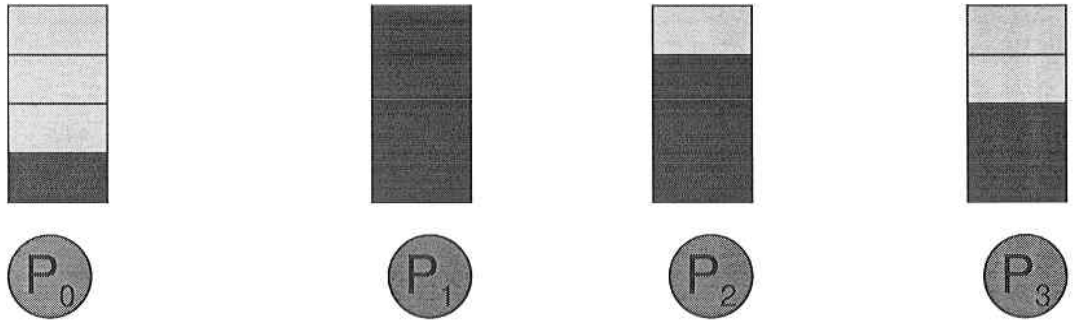
[Graham69, Jaffe80, Bender-Rabin02]

$$makespan \le \frac{W}{P_{ave}} + \left(1 - \frac{1}{P}\right) \frac{D}{P_{ave}}$$

The work stealing algorithm

A distributed and randomized algorithm that computes a greedy schedule :

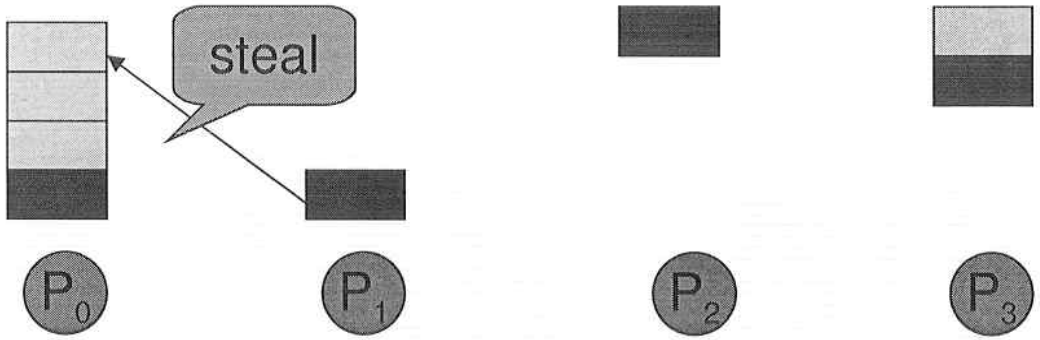
- Each processor manages a local task (depth-first execution)



The work stealing algorithm

A distributed and randomized algorithm that compute a greedy schedule :

- Each processor manages a local stack (depth-first execution)



- When idle, a processor steals the topmost task on a remote -non idle- victim processor (randomly chosen)
- **Theorem**: With good probability, [Acar,Blelloch, Blumofe02, BenderRabin02]
  - #steals < $p.D$
  - execution time $\leq \frac{W}{p.P_{ave}} + O\left(\frac{D}{P_{ave}}\right)$

- **Interest**:
  if ***W* independent of *p*** and ***D* is small**, work stealing achieves **near-optimal** schedule

Vork stealing implementation

Scheduling

efficient policy
(close to optimal)

control of the policy
(realisation)

Difficult in general (coarse grain)

But easy if *D* is small [Work-stealing]

Execution time $\leq \frac{W}{p.\mathrm{P}_{ave}} + O\left(\frac{D}{\mathrm{P}_{ave}}\right)$

(fine grain)

Expensive in general (fine grain)

But small overhead if a small number of tasks

(coarse grain)

*f D is small, a work stealing algorithm performs a **small number of steals***

=> ***Work-first principle:*** "scheduling overheads should be borne by the critical path f the computation" [Frigo 98]

mplementation: since all tasks but a few are executed in the local stack, overhead f task creation should be as close as possible as sequential function call

t any time on any non-idle processor,
efficient local *degeneration* of the *parallel* program in a *sequential execution*

Work-stealing implementations following the work-first principle : Cilk

Cilk-5 http://supertech.csail.mit.edu/cilk/ **: C extension**

- **Spawn** f (a) ; **sync** (serie-parallel programs)
- Requires a shared-memory machine
- Depth-first execution with synchronization (on sync) with the end of a task :
  - Spawned tasks are pushed in double-ended queue
- "Two-clone" compilation strategy [Frigo-Leiserson-Randall98] :
  - on a successfull steal, a thief executes the continuation on the topmost ready task ;
  - When the continuation hasn't been stolen, "sync" = nop ; else synchronization with its thief

```
01 cilk int fib (int n)
02 {
03     if (n < 2) return n;
04     else
05     {
06         int x, y;
07
08         x = spawn fib (n-1);
09         y = spawn fib (n-2);
10
11         sync;
12
13         return (x+y);
14     }
15 }
```

```
1   int fib (int n)
2   {
3       fib_frame *f;                  frame pointer
4       f = alloc(sizeof(*f));         allocate frame
5       f->sig = fib_sig;              initialize frame
6       if (n<2) {
7           free(f, sizeof(*f));       free frame
8           return n;
9       }
10      else {
11          int x, y;
12          f->entry = 1;              save PC
13          f->n = n;                  save live vars
14          *T = f;                    store frame pointer
15          push();                    push frame
16          x = fib (n-1);             do C call
17          if (pop(x) == FAILURE)     pop frame
18              return 0;              frame stolen
19          ...                        second spawn
20          ;                          sync is free!
21          free(f, sizeof(*f));       free frame
22          return (x+y);
23      }
24  }
```

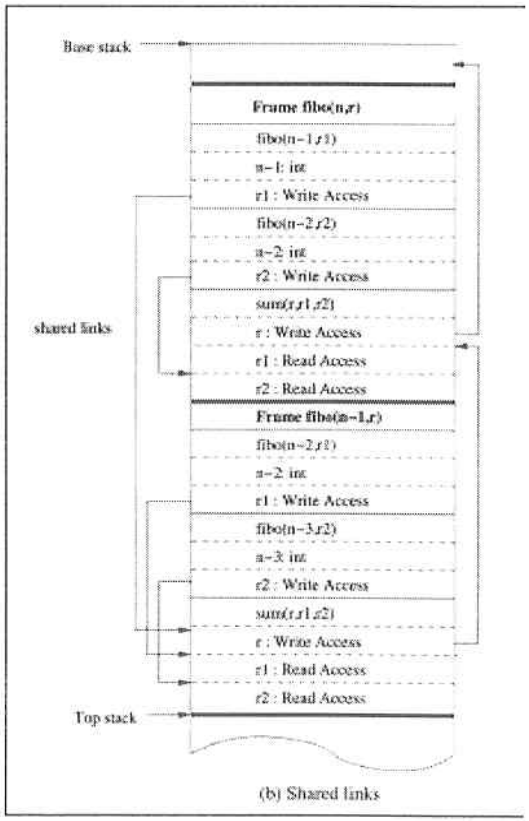
- *won the 2006 award "Best Combination of Elegance and Performance" at HPC Challenge Class 2, SC'06, Tampa, Nov 14 2006 [Kuszmaul] on SGI ALTIX 3700 with 128 bi-Ithanium]*

Work-stealing implementations following the work-first principle : KAAPI

Kaapi / Athapascan http://kaapi.gforge.inria : C++ library

- **Fork**<f>()(a, …) with **access mode** to parameters (value;read;write;r/w;cw) **specified in f prototype** (macro dataflow programs)
- Supports distributed and shared memory machines; heterogeneous processors
- Depth-first (*reference order*) execution with synchronization on data access :
  - Double-end queue (mutual exclusion with compare-and-swap)
  - on a successfull steal, one-way data communication (write&signal)

```
 1  struct sum {
 2     void operator()(Shared_r < int > a,
 3                     Shared_r < int > b,
 4                     Shared_w < int > r )
 5     { r.write(a.read() + b.read()); }
 6   } ;
 7
 8   struct fib {
 9    void operator()(int n, Shared_w<int> r)
10    { if (n <2) r.write( n );
11      else
12      { int r1, r2;
13        Fork< fib >() ( n-1, r1 ) ;
14        Fork< fib >() ( n-2, r2 ) ;
15        Fork< sum >() ( r1, r2, r ) ;
16      }
17    }
18  } ;
```

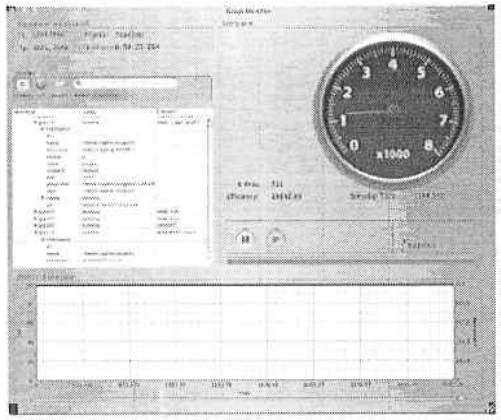


- *Static scheduling won the 2006 award "Prix special du Jury" for the best performance at NQueens contest, Plugtests-Grid&Work'06, Nice, Dec.1, 2006 [Gautier-Guelton] on Grid'5000 1458 processors with different speeds.*

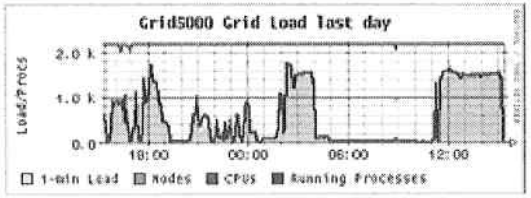
-queens: Takaken C sequential code arallelized in C++/Kaapi

- *T. Gautier&S. Guelton won the 2006 award "Prix special du Jury" for the best performance at NQueens contest, Plugtests- Grid&Work'06, Nice, Dec.1, 200*

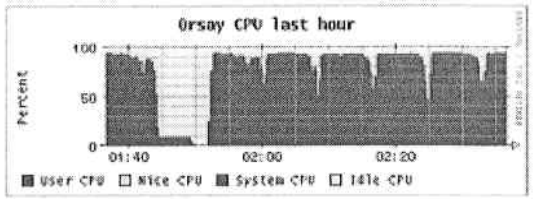
- Some facts [on *on Grid'5000*, a grid of processors of heterogeneous speeds]
  - NQueens(21) in 78 s on about 1000 processors
  - Nqueens (22) in 502.9s on 1458 processors
  - Nqueens(23) in 4435s on 1422 processors [~24.10^{33} solutions]
  - 0.625% idle time per processor
  - < 20s to deploy up to 1000 processes on 1000 machines [Taktuk, Huard]
  - 15% of improvement of the sequential due to C++ (template)



Grid'5000 utilization during contest



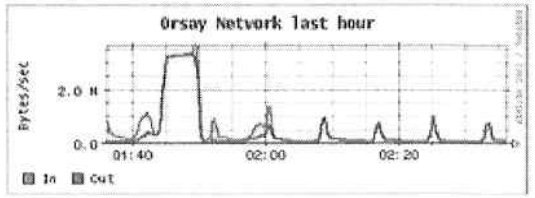
Competitor X
Competitor Y
Competitor Z
Grid'5000 free
N-Queens(23)



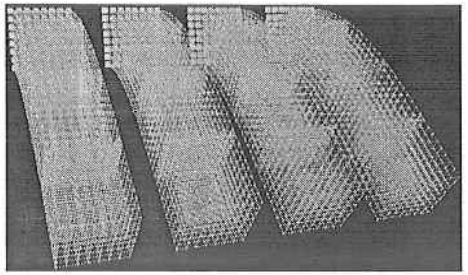
CPU

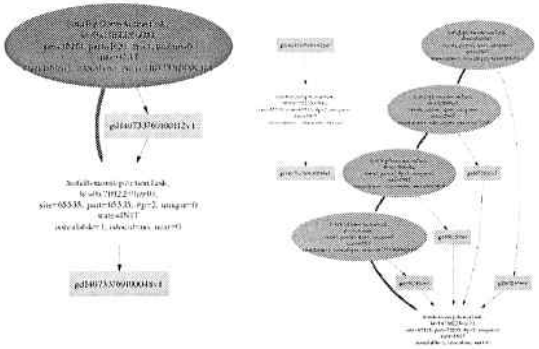
6 instances Nqueens(22)

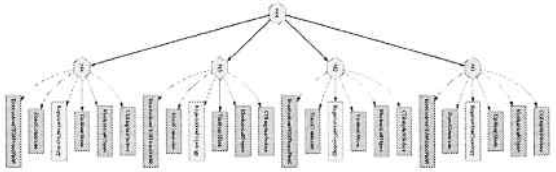
Network



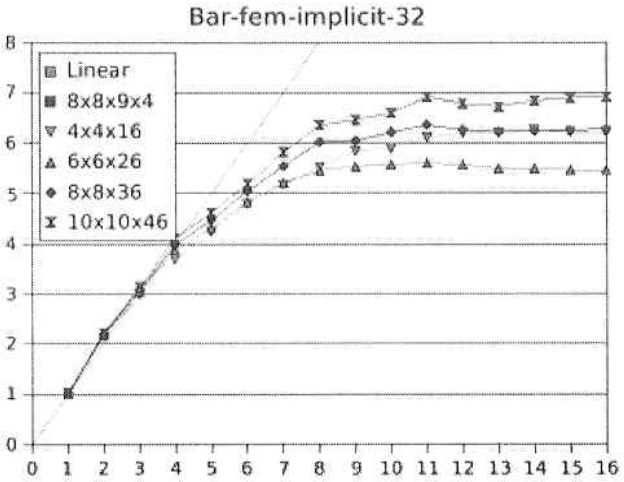
Experimental results on SOFA [CIMIT-ETZH-INRI



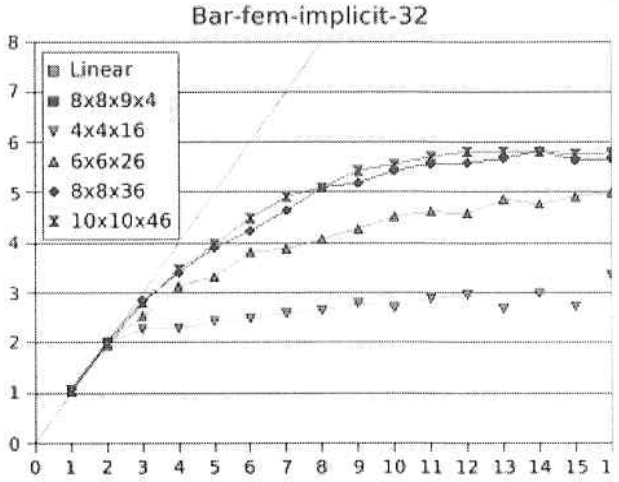




[Allard 06]



Kaapi (C++, ~500 lines)

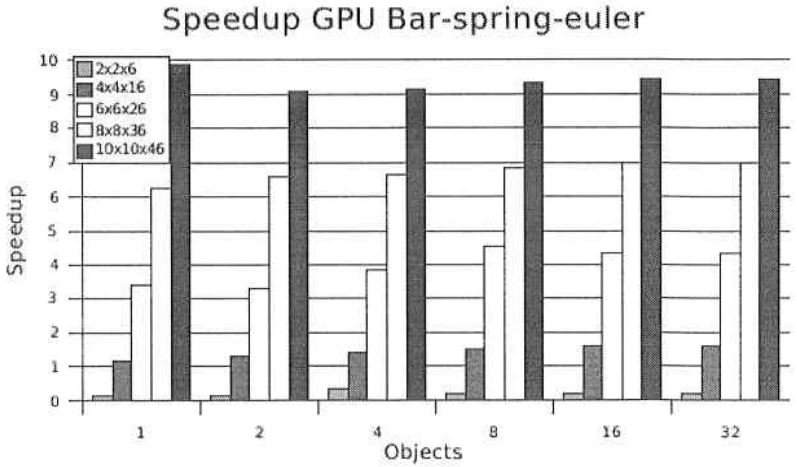


Cilk (C, ~240 lines)

eliminary results on GPU NVIDIA 8800 GTX

peed-up ~9 on Bar 10x10x46 to Athlon64 2.4GHz

- 128 "cores" in 16 groups
- CUDA SDK : "BSP"-like, 16 X [16 .. 512] threads
- Supports most operations available on CPU
- ~2000 lines CPU-side + 1000 GPU-side



3. Work-first principle and adaptability

- **Work-first principle**: -implicit- dynamic choice between two executions :
  - a **sequential** ***"depth-first"*** execution of the parallel algorithm (local, default) ;
  - a **parallel** ***"breadth-first"*** one.
- Choice is performed at runtime, depending on resource idleness:
  rare event if Depth is small to Work
- **WS adapts parallelism to processors with practical provable performance**
  - Processors with changing speeds / load (data, user processes, system, users,
  - Addition of resources (fault-tolerance [Cilk/Porch, Kaapi, …])

- **The choice is justified only when the sequential execution of the parallel algorithm is an efficient sequential algorithm:**
  - Parallel Divide&Conquer computations
  - …

  -> But, this may not be general in practice

low to get both optimal work W_1 and W_∞ small?

- General approach: **to** mix **both**
  - a **sequential** algorithm with optimal work W_1
  - and a fine grain **parallel** algorithm with minimal critical time W_∞

- Folk technique **:** *parallel, than sequential*
  - Parallel algorithm until a certain « grain »; then use the sequential one
  - Drawback : W_∞ increases ;o) …and, also, the number of steals

- *Work-preserving speed-up technique* [Bini-Pan94] *sequential, then parallel* **Cascading** [Jaja92] **: Careful interplay of both algorithms to build one with both**

$$W_\infty \textbf{ small} \quad \text{and} \quad W_1 = \mathbf{O}(\, W_{seq}\,)$$

  - Use the work-optimal sequential algorithm to reduce the size
  - Then use the time-optimal parallel algorithm to decrease the time
  - Drawback : sequential at coarse grain and parallel at fine grain ;o(

Extended work-stealing: concurrently sequential and parall

*Based on the work-stealing and the **Work-first** principle* :

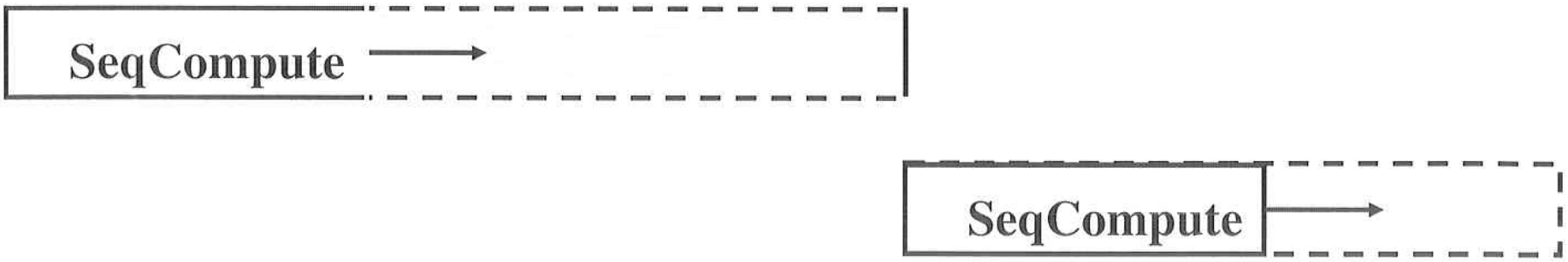
Instead of optimizing the **sequential execution** of the **best parallel** algorithm,
let optimize the **parallel execution** of the ***best sequential*** algorithm

Execute always a sequential algorithm to reduce parallelism overhead

⇒ parallel algorithm is used only if a processor becomes **idle** (ie *workstealing*) [Roch&al2005,..
to ***extract parallelism*** from the remaining work a sequential computation

Assumption : two concurrent algorithms that are complementary:

- - one sequential : ***SeqCompute*** (always performed, the priority)
  - the other parallel, fine grain : ***LastPartComputation*** (often not performed)



Extended work-stealing : concurrently sequential and parall

*Based on the work-stealing and the **Work-first** principle* :

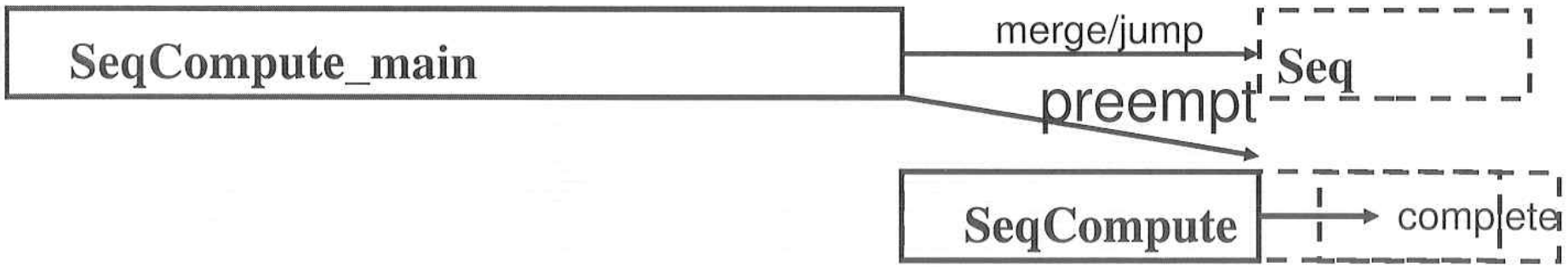
Instead of optimizing the **sequential execution** of the **best parallel** algorithm,
let optimize the **parallel execution** of the ***best sequential*** algorithm

Execute always a sequential algorithm to reduce parallelism overhead

⇒ parallel algorithm is used only if a processor becomes **idle** (ie *workstealing*) [Roch&al2005,..
to ***extract parallelism*** from the remaining work a sequential computation

Assumption : two concurrent algorithms that are complementary:

- - one sequential : ***SeqCompute*** (always performed, the priority)
  - the other parallel, fine grain : ***LastPartComputation*** (often not performed)



Note:

- **merge and jump** operations to ensure non-idleness of the victim
- Once *SeqCompute_main* completes, it becomes a work-stealer

Extended work-stealing and granularity

- **Scheme of the sequential process : nanoloop**

```
While (not completed(Wrem) ) and (next_operation hasn't been stolen)
{
    atomic { extract_next k operations ; Wrem -= k  ; }
    process the k operations extracted ;
}
```

- ***Processor-oblivious* algorithm**
  - Whatever *p* is, it performs O(*p.D*) preemption operations (« continuation faults »)
    -> ***D*** should be as small as possible to maximize both speed-up and locality
  - If no steal occurs during a (sequential) computation, then its *arithmetic work* is optim to the one W_{opt} of the sequential algorithm (no spawn/fork/copy)
    -> ***W*** should be as close as possible to W_{opt}
- Choosing **k = Depth(W_{rem})** does not increase the depth of the parallel algorithm while ensuring ***O(W / D)*** *atomic operations :*
  since $D > log_2 W_{rem}$, **then if *p = 1*:** $W \sim W_{opt}$
- **Implementation :** atomicity in nano-loop based on efficient local lock
- ***Self-adaptive granularity***

nteractive application with time constraint

Anytime Algorithm:

- Can be stopped at any time (with a result)
- Result quality improves as more time is allocated

In Computer graphics, anytime algorithms are common:

Level of Detail algorithms (time budget, triangle budget, etc…)

Example: Progressive texture loading, triangle decimation (Google Earth)

Anytime processor-oblivious algorithm:

On p processors with average speed Π_{ave}, it outputs in a fixed time T a result with the same quality than a sequential processor with speed Π_{ave} in time $p.\Pi_{ave}$.

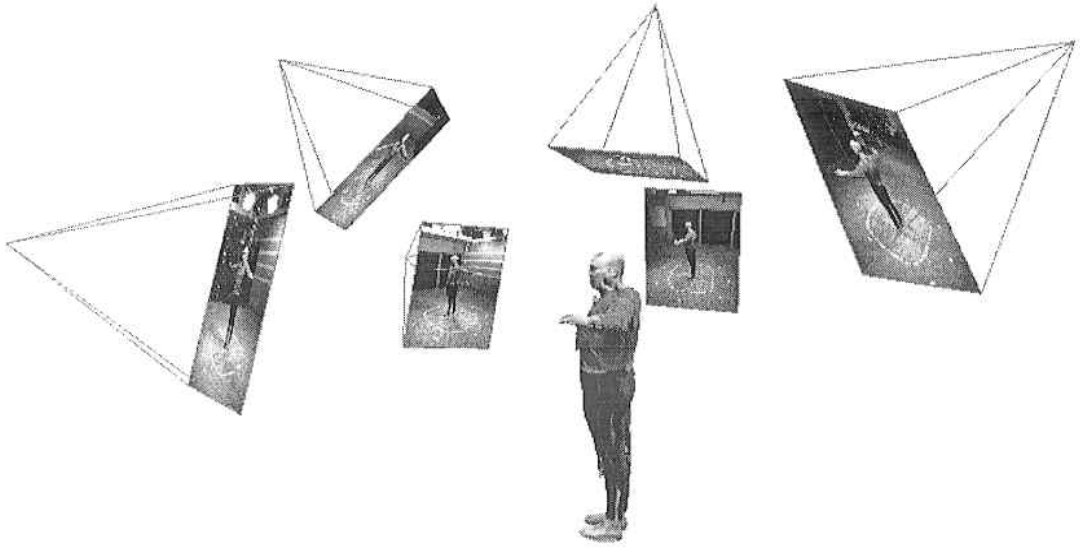
Example: Parallel Octree computation for 3D Modeling

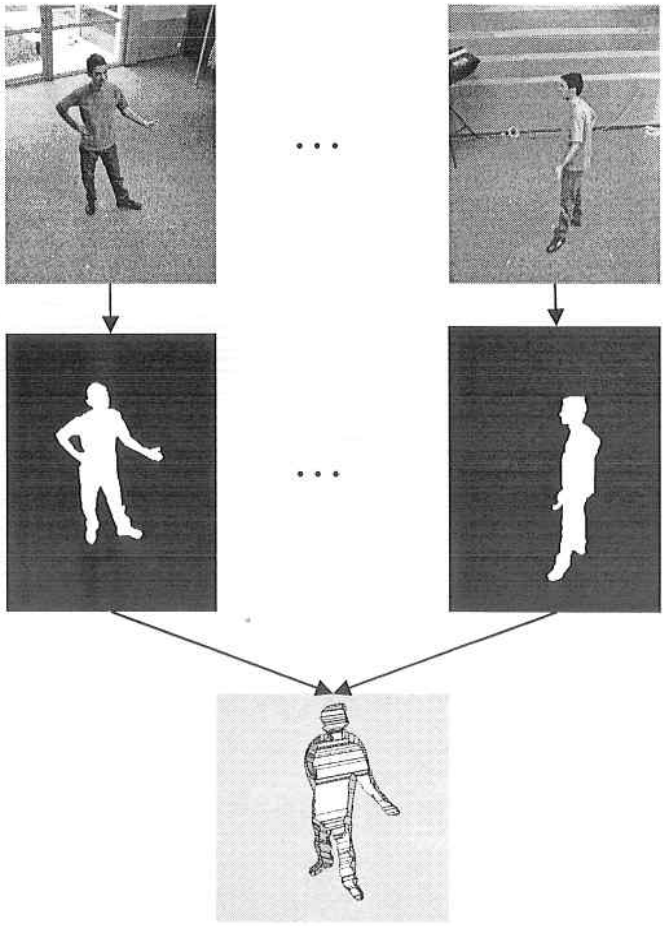
Parallel 3D Modeling

3D Modeling :

build a 3D model of a scene from a set of calibrated images

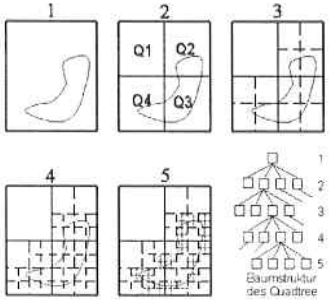
On-line 3D modeling for interactions: 3D modeling from multiple video streams (30 fps)

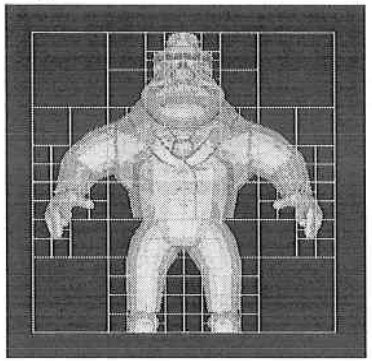


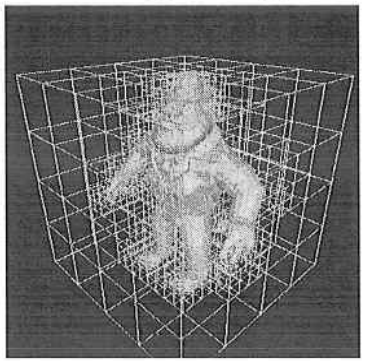


Octree Carving [L. Soares 06]

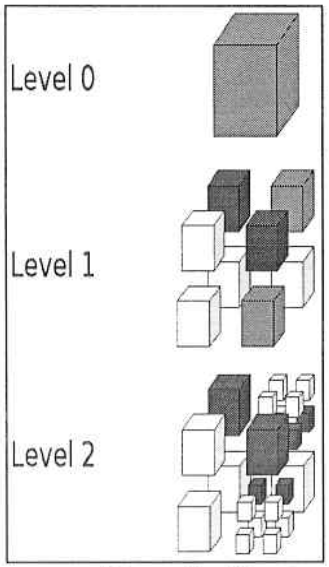
A classical recursive anytime 3D modeling algorithm.





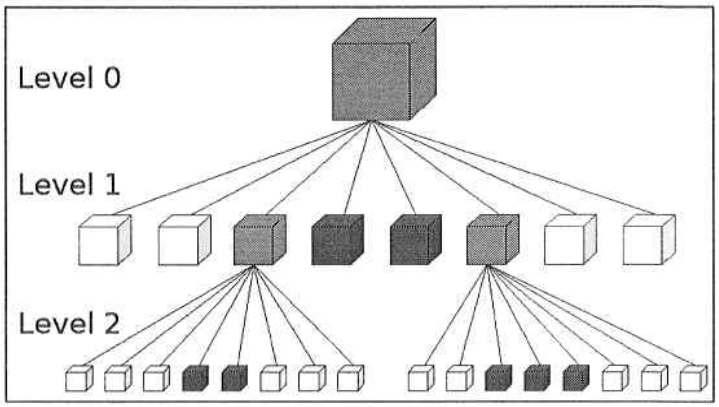


Standard algorithms with time control:



Depth first
+ iterative deepening

State of a cube:
- Grey: mixed => split
- Black: full : stop
- White: empty : stop



Width first

At termination: quick test to decide all grey cubes time control

Width first parallel octree carving

ell suited to work-stealing

-Small critical path, while huge amount of work (eg. D = 8, W = 164 000)

- non-predictable work, non predictable grain :

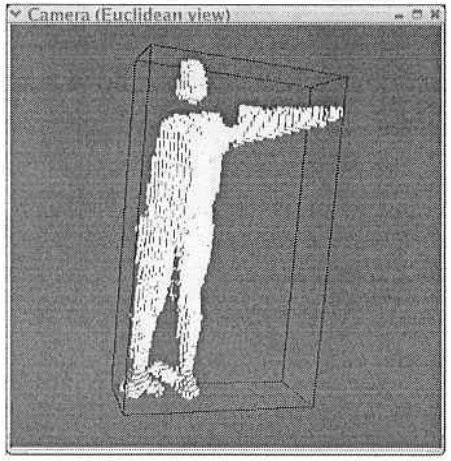
)r cache locality, each level is processed by a self-adaptive grain :

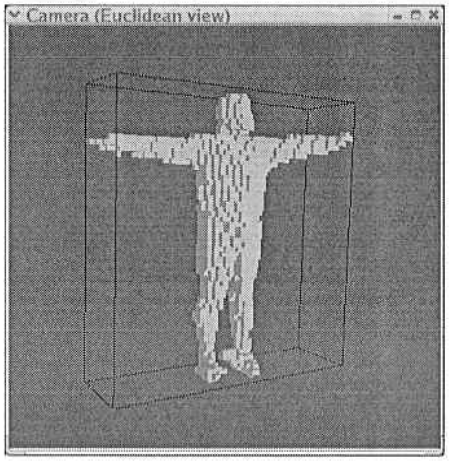
"sequential iterative" / "parallel recursive split-half"

:tree needs to be "balanced" when stopping:

- Serially computes each level *(with small overlap)*
- Time deadline (30 ms) managed by signal protocol

Unbalanced





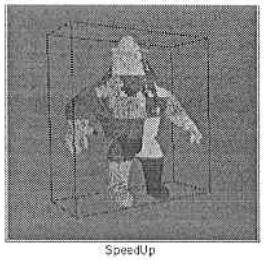
Balanced

ıeorem: W.r.t the adaptive in time T on p procs., the sequential algorithm:

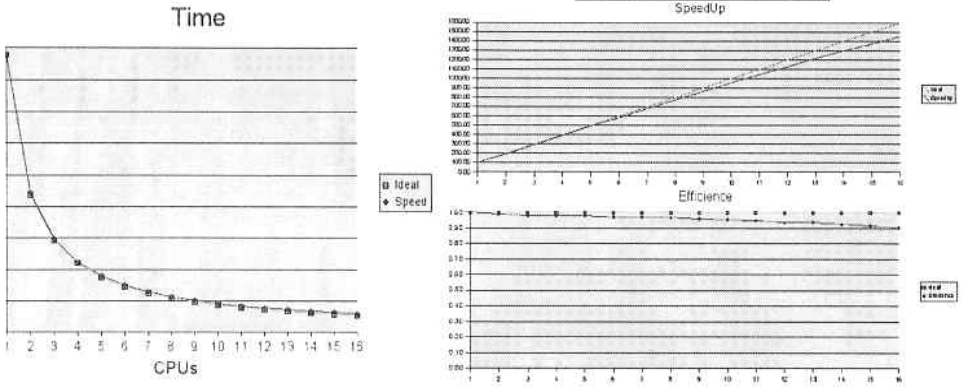
- goes at most one level deeper : $| d_s - d_p | \leq 1$;
- computes at most : $n_s \leq n_p + O(\log n_s)$.

Results

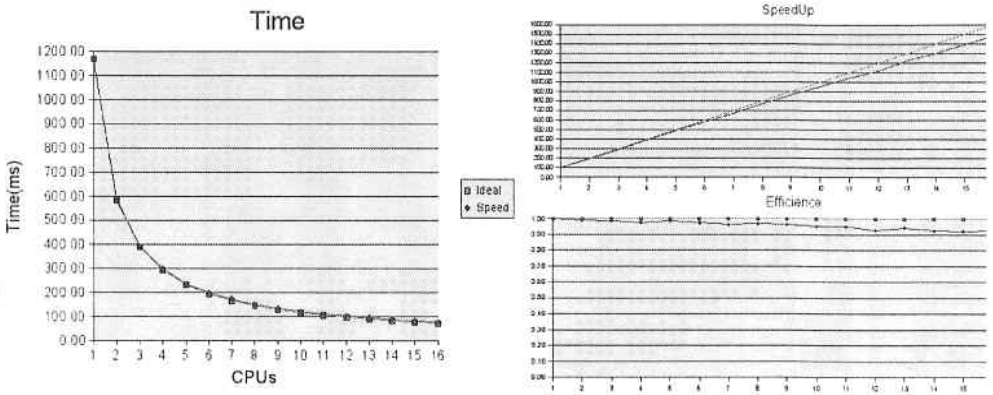
[L. Soares 06]



- 16 core Opteron machine, 64 images
- Sequential: 269 ms, 16 Cores: 24 ms
- 8 cores: about 100 steals (167 000 grey cells)



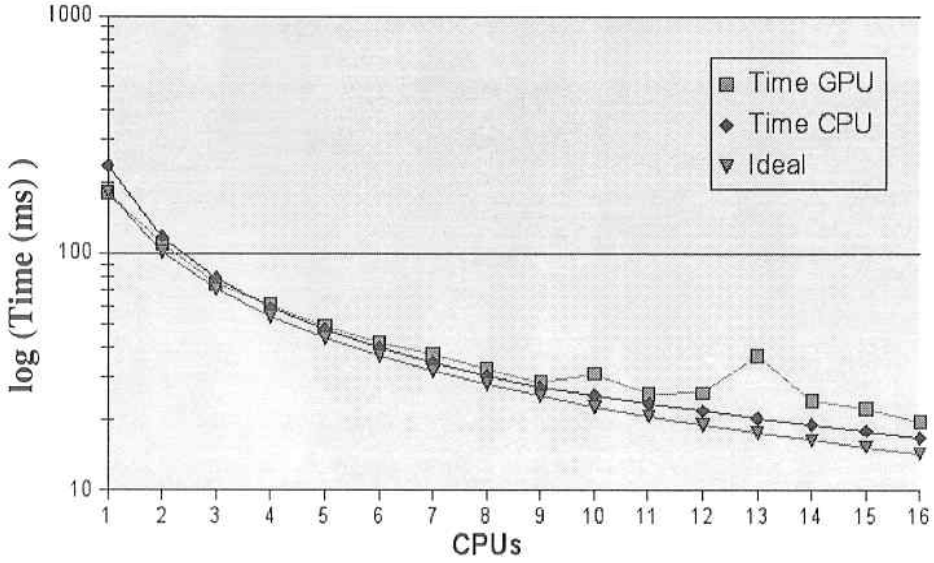
8 cameras, levels 2 to 10



64 cameras, levels 2 to 7

Preliminary result: CPUs+GPU

- 1 GPU + 16 CPUs
- GPU programmed in OpenGL
- efficient coupling till 8 but does not scale



4. Amortizing the arithmetic overhead of parallelism

Adaptive scheme : extract_seq/nanoloop // extract_par

- ensures an optimal number of operation on 1 processor
- but no guarantee on the work performed on p processors

Eg (C++ STL): find_if (first, last, predicate)

locates the first element in [First, Last) verifying the predicate

This may be a drawback :

- unneeded processor usage ;
- undesirable for a library code that may be used in a complex application, with many components
- (or not fair with other users)
- increases the time of the application :
  - any parallelism that increases the execution time should be avoided

Motivates the building of **work-optimal** parallel adaptive algorithm (processor oblivious)

4. Amortizing the arithmetic overhead of parallelism (cont'd)

Similar to nano-loop for the sequential process :

- that balances the -atomic- local work by the depth of the remaindering one

Here, by **amortizing** the work induced by the extract_par operation, ensuring this **work to be *small*** enough :

- Either w.r.t the -useful- **work already performed**
- Or with respect to the - useful - **work yet to performed** (if known)
- or both.

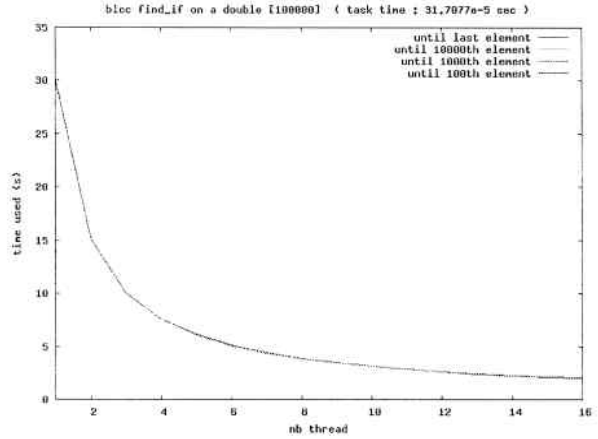
Eg : find_if (first, last, predicate) :

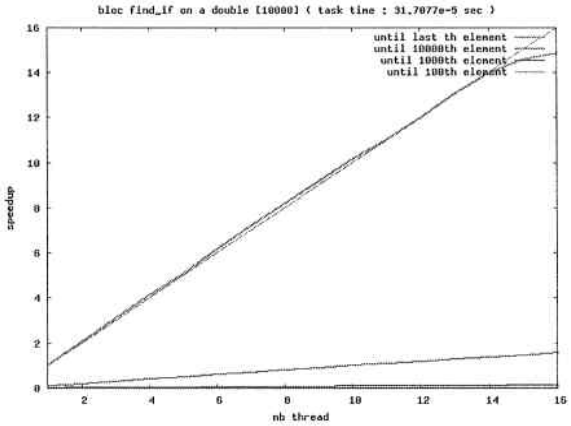
- only the work already performed is known (on-line)
- then prevent to assign more than $\alpha(W_{done})$ operations to work-stealers
- Choices for $\alpha(n)$:
  - **n/2 : similar to Floyd's iteration** (approximation ratio = 2)
  - **n/log* n :** to ensure optimal usage of the work-stealers

Results on find_if

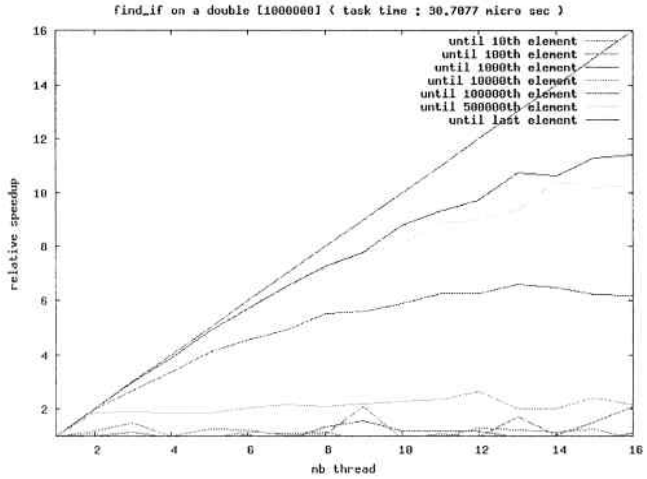
[S. Guelton]

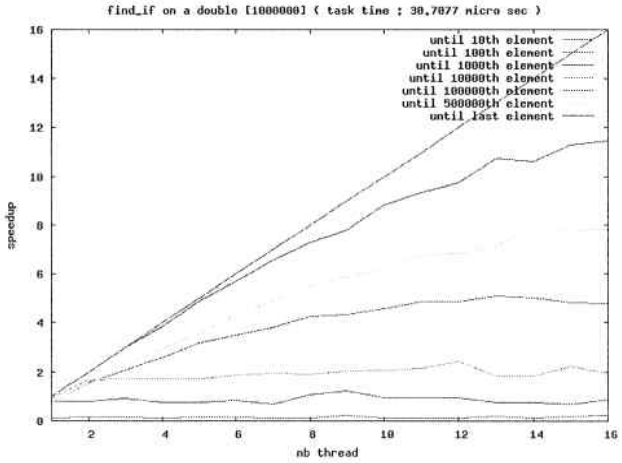
N doubles : time predicate ~ 0.31 ms





With no amortization macroloop





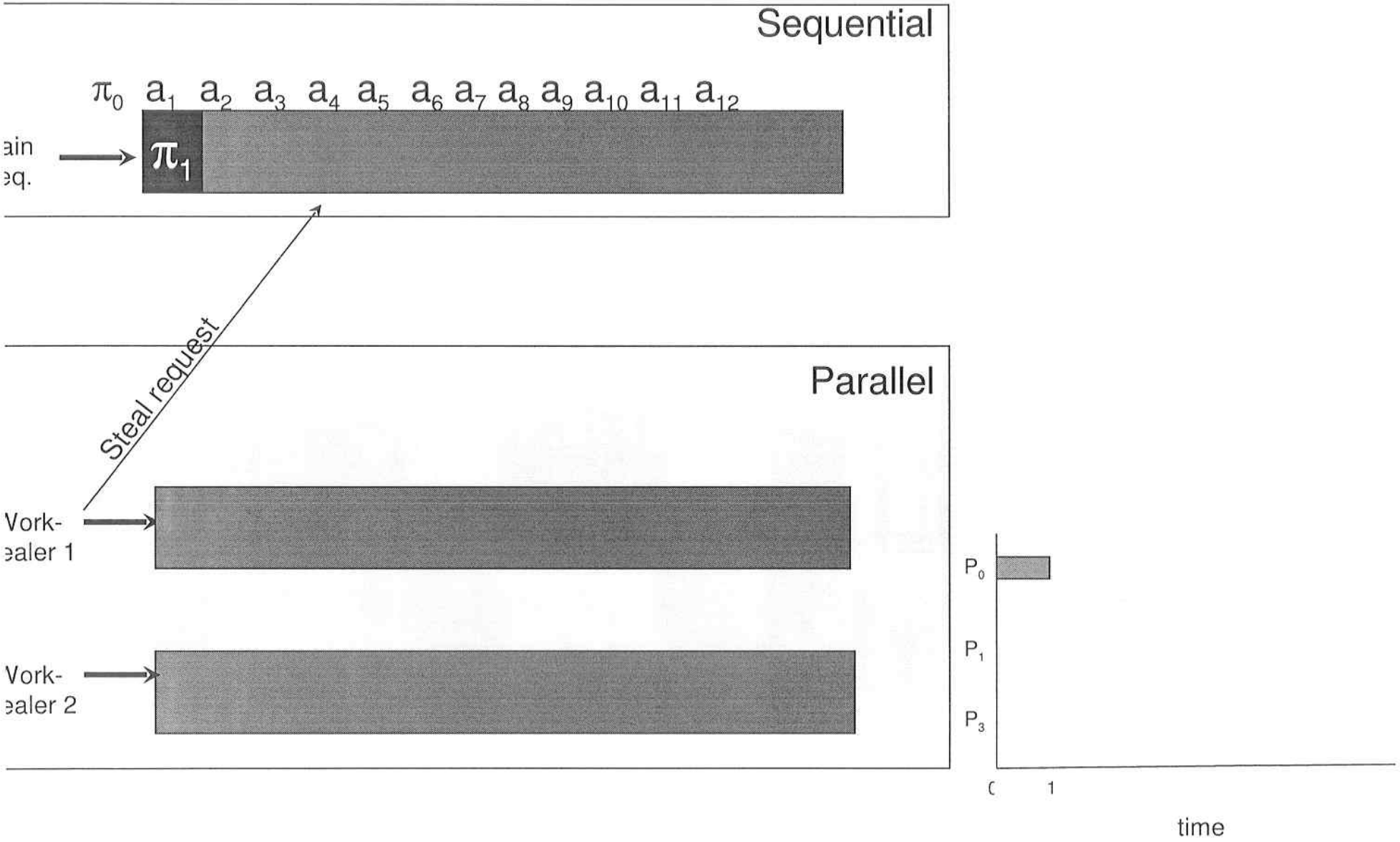
With amortization macroloop

5. Putting things together
processor-oblivious prefix computation

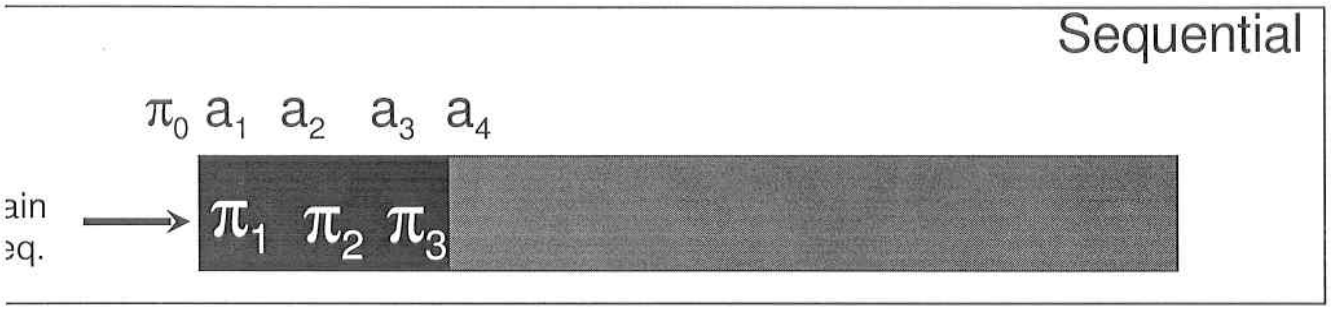
Parallel algorithm based on :

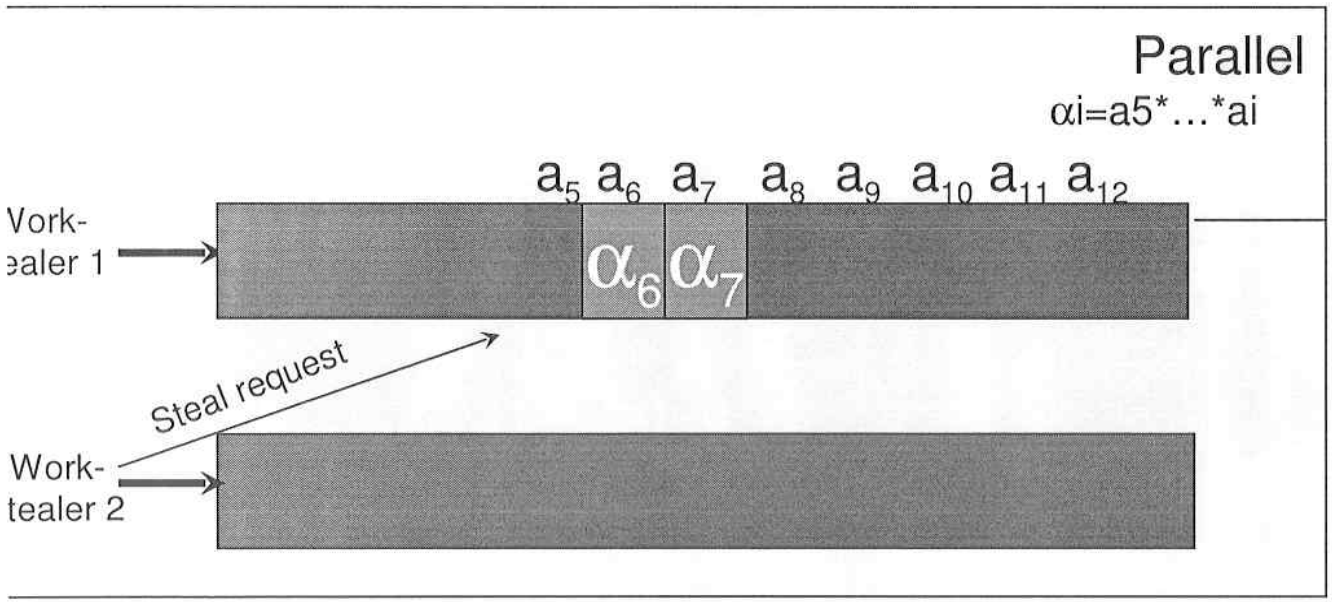
- **compute-seq / extract-par scheme**
- nano-loop for compute-seq
- macro-loop for extract-par

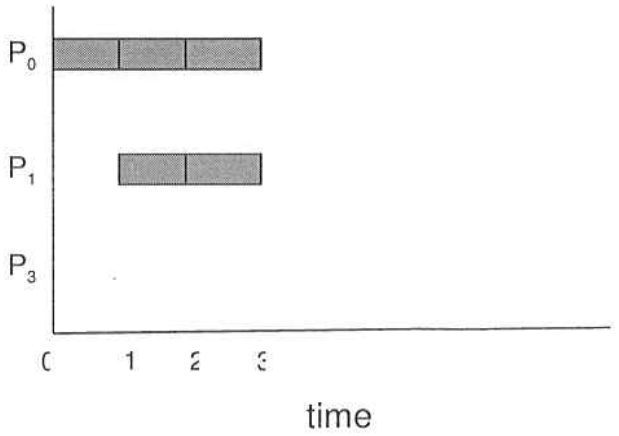
P-Oblivious Prefix on 3 proc.



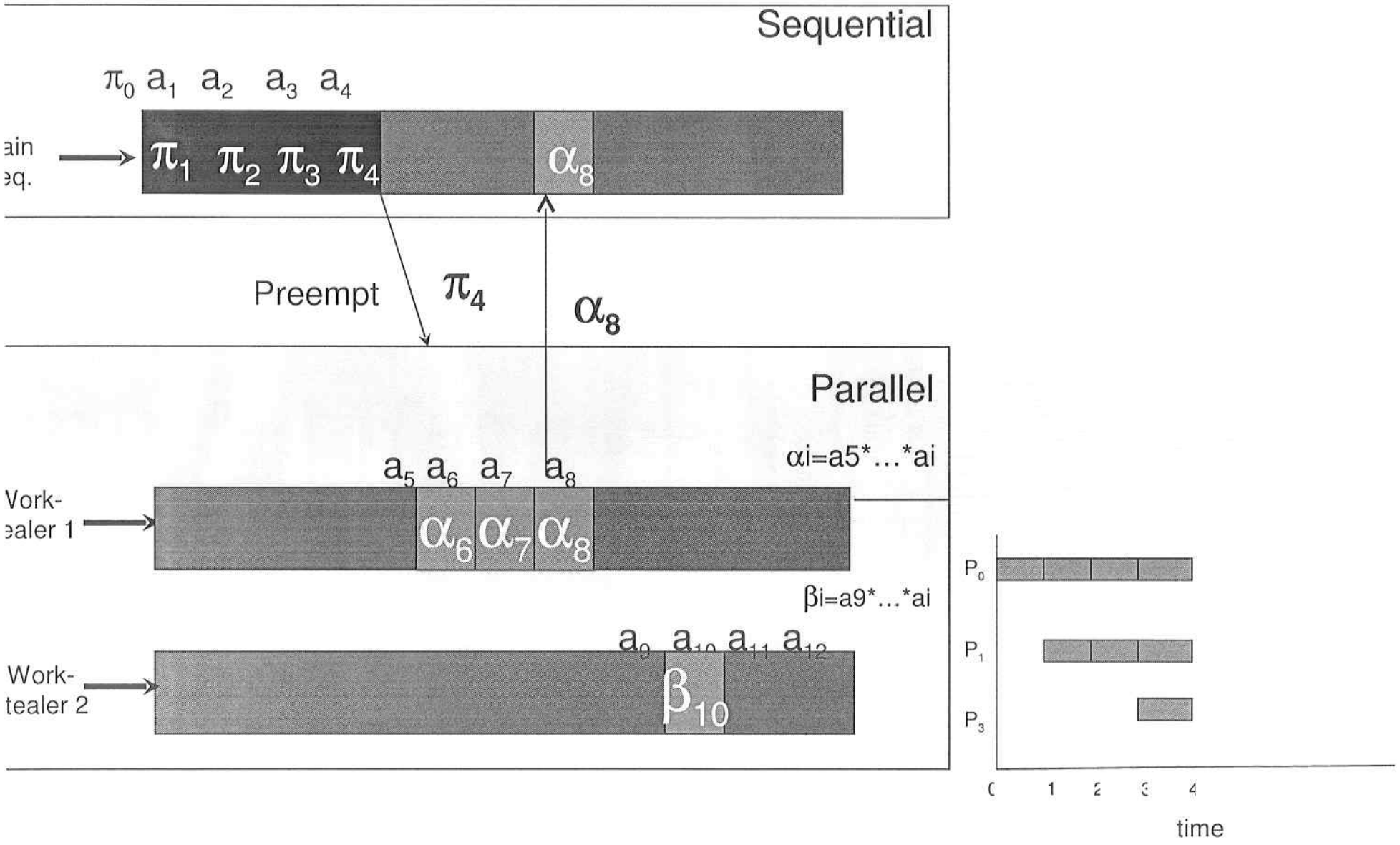
P-Oblivious Prefix on 3 proc.



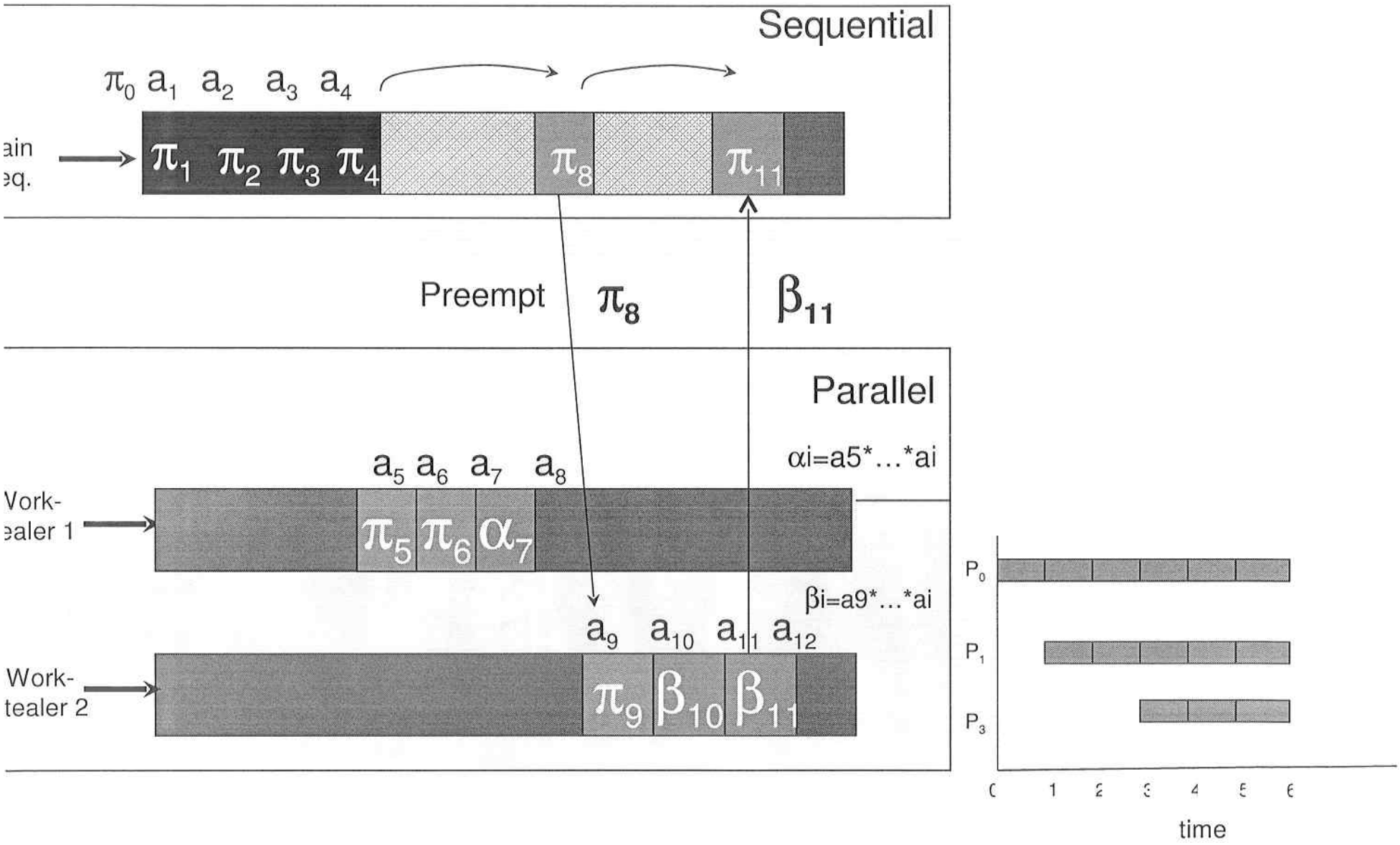




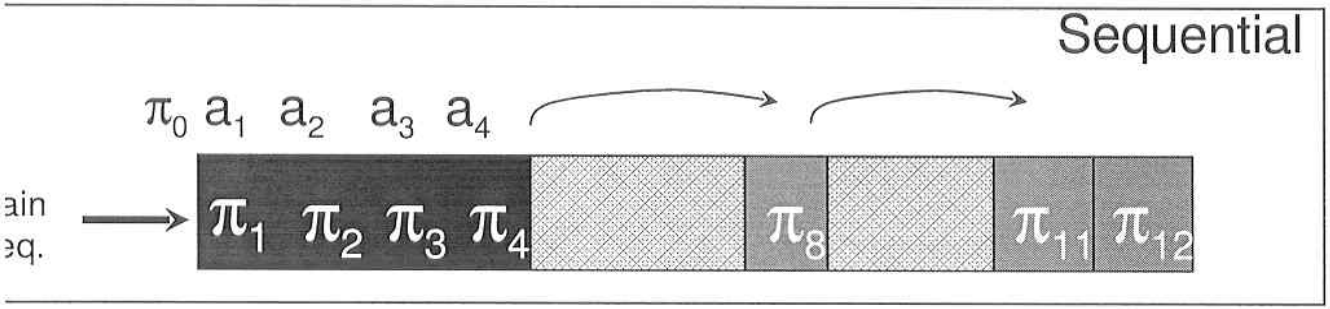
P-Oblivious Prefix on 3 proc.

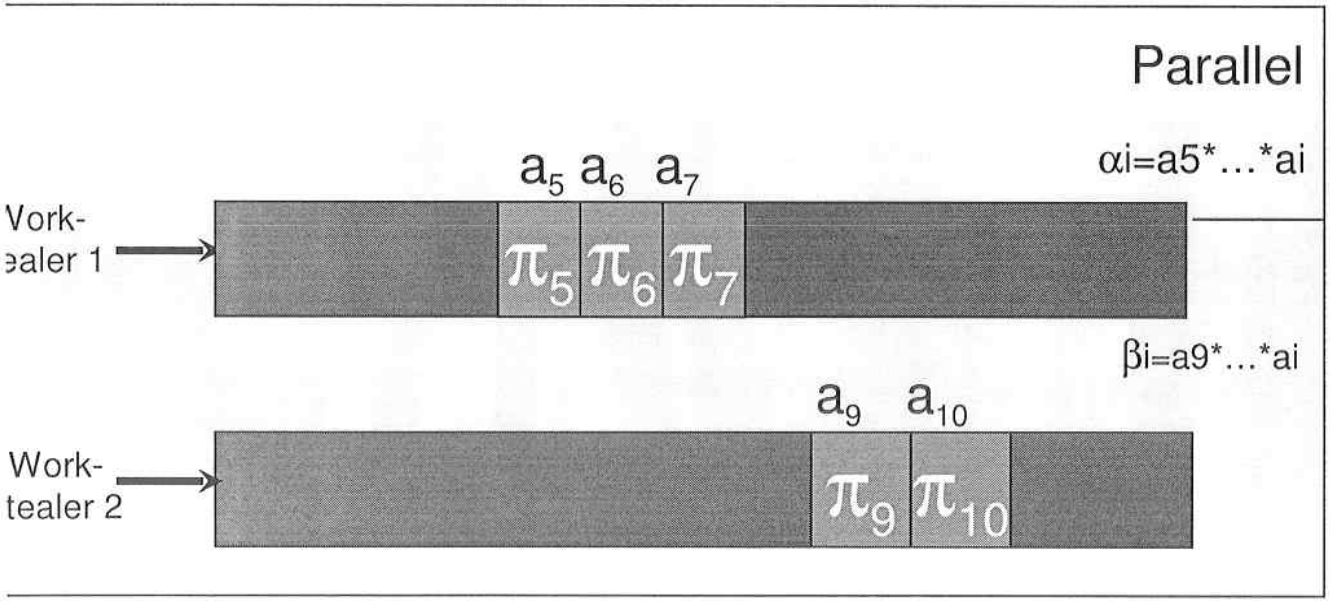


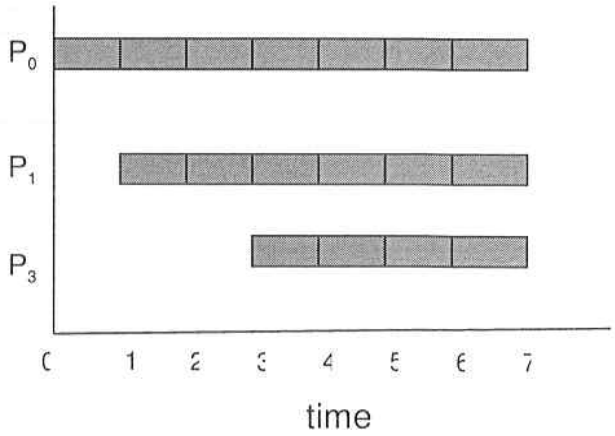
P-Oblivious Prefix on 3 proc.



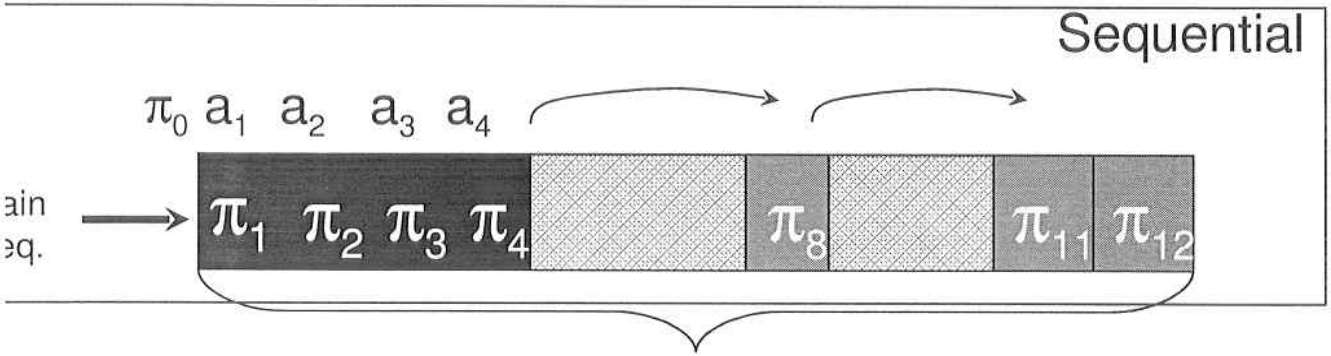
P-Oblivious Prefix on 3 proc.





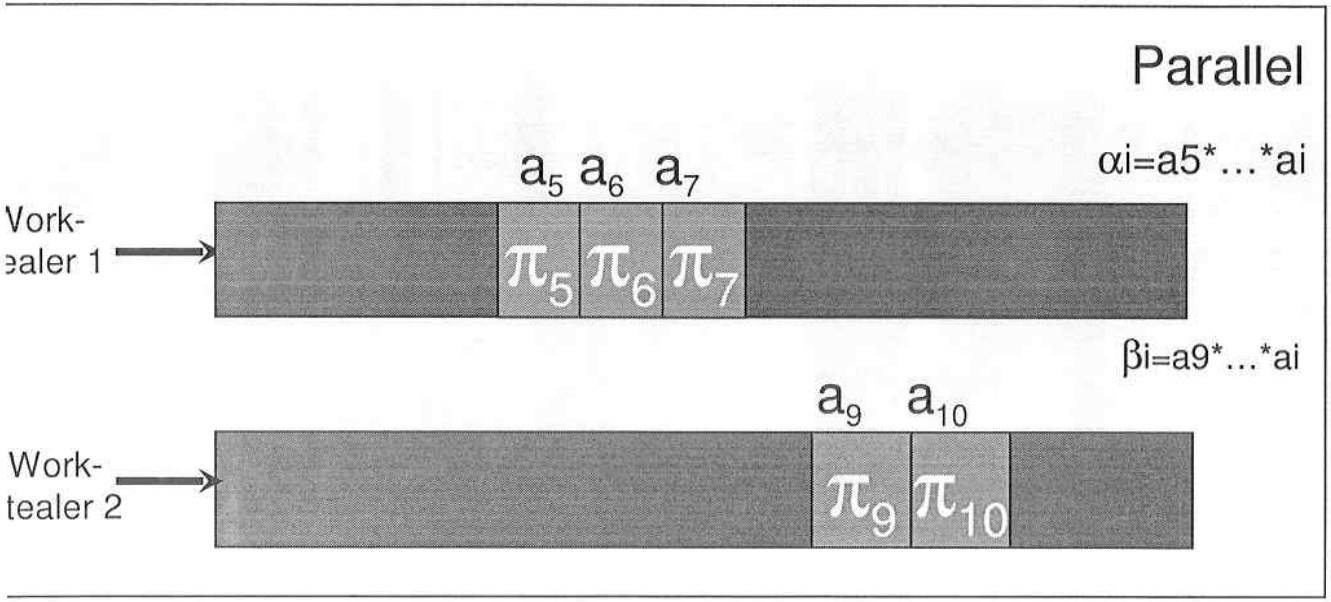


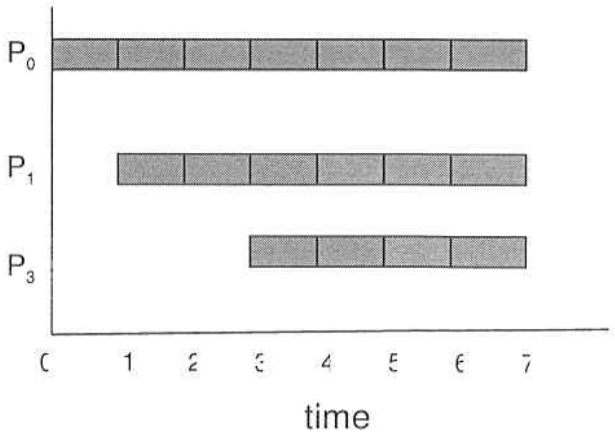
P-Oblivious Prefix on 3 proc.



Implicit critical path on the sequential process

$T_p = 7$ $T_p^* = 6$





Analysis of the algorithm

$$\text{Execution time} \leq \text{Lower bound} + O\left(\frac{\log n}{\Pi_{ave}}\right)$$

Sketch of the proof :

Dynamic coupling of two algorithms that complete simultaneously:

- Sequential: (optimal) number of operations S on one processor
- Extract_par : work stealer perform X operations on other processors
  - dynamic splitting always possible till finest grain BUT local sequential
    - Critical path small (eg : log X with a W= n / log* n macroloop)
    - Each non constant time task can potentially be splitted (variable speeds)

$$T_s = \frac{S}{\Pi_{ave}} \text{ and } T_p = \frac{X}{(p-1).\Pi_{ave}} + O\left(\frac{\log X}{\Pi_{ave}}\right)$$

- Algorithmic scheme ensures $T_s = T_p + O(\log X)$

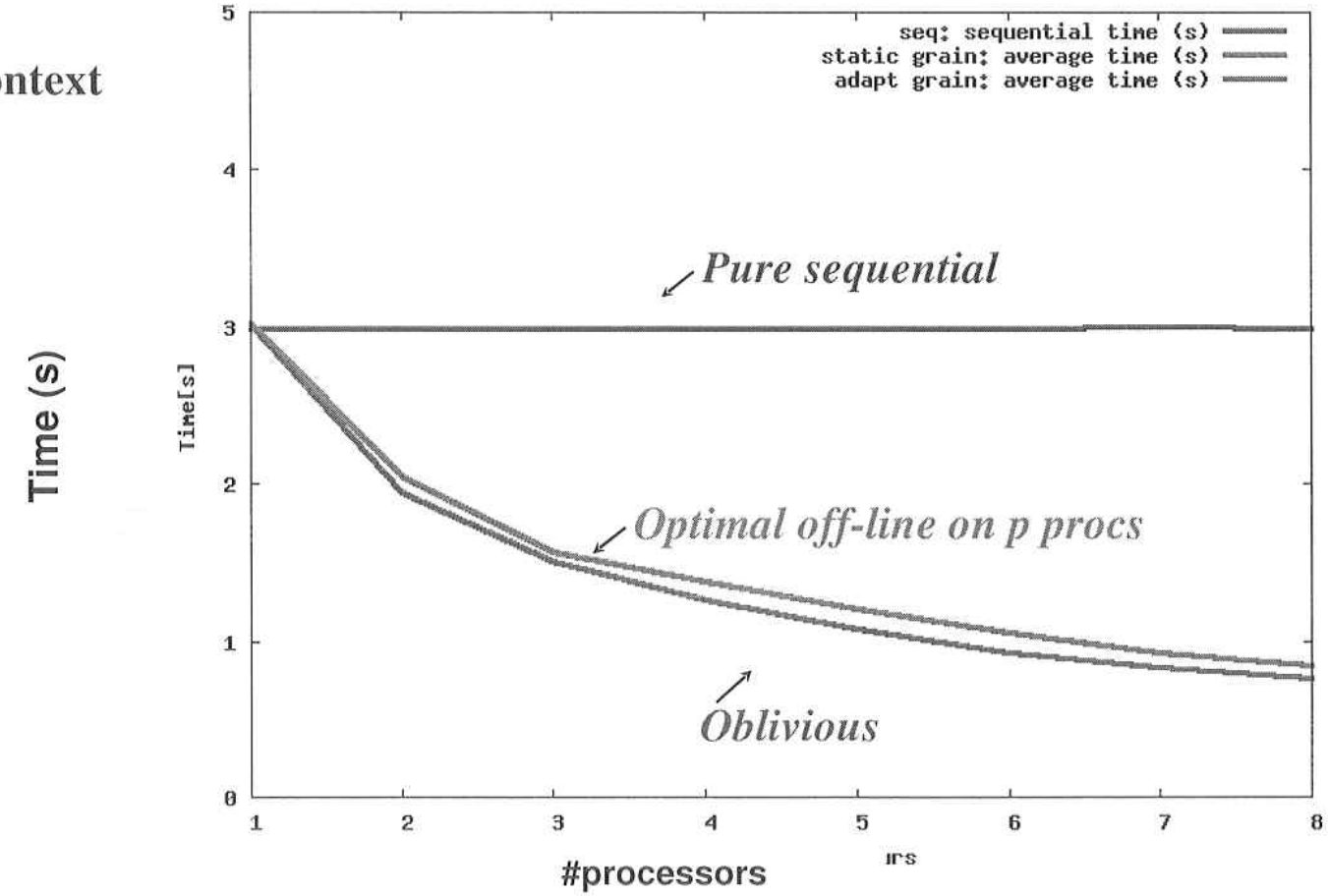
=> enables to bound the whole number X of operations performed
and the overhead of parallelism = (s+X) - #ops_optimal

Results 1/2

[D Traore]

Prefix sum of 8.10^6 double on a SMP 8 procs (IA64 1.5GHz/ linux)

Single user context



Single-usercontext : processor-oblivious prefix achieves near-optimal performance :

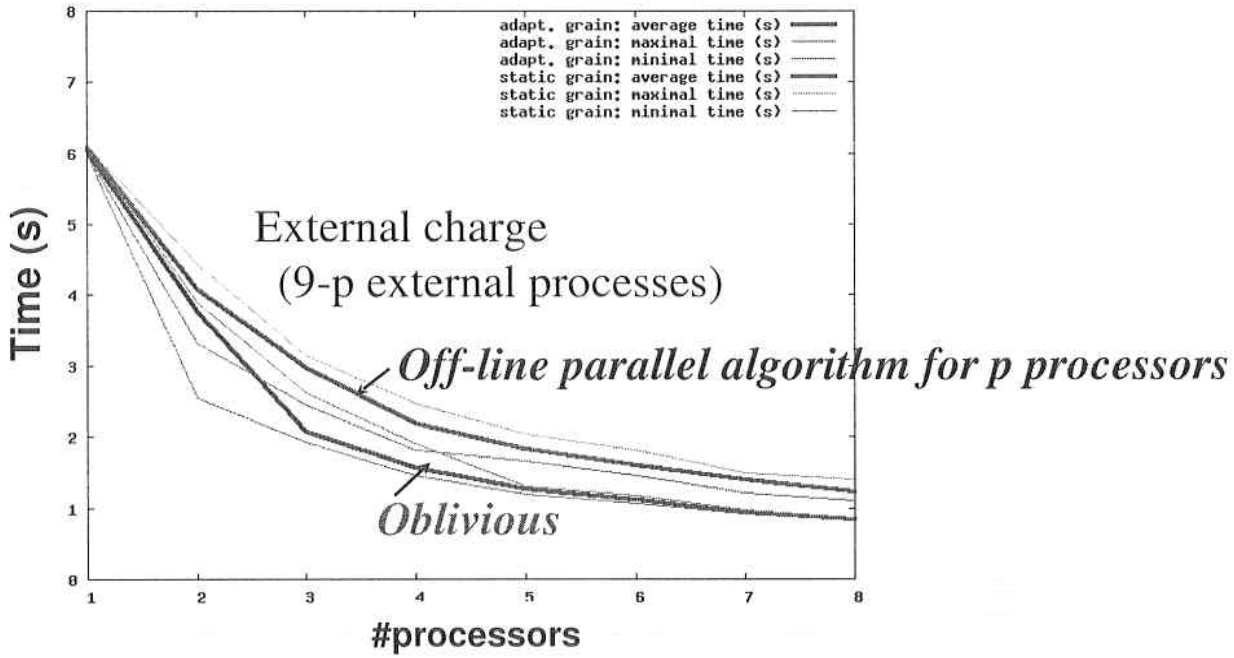
- close to the lower bound both on 1 proc and on p processors

- Less sensitive to system overhead : even better than the theoretically "optimal" off-line parallel algorithm on p proc

Results 2/2 [D Traore]

Prefix sum of 8.10^6 double on a SMP 8 procs (IA64 1.5GHz/ linux)

Multi-user context :



Multi-user context :

Additional external charge: (9-p) additional external dummy processes are concurrently executed

Processor-oblivious prefix computation is always the fastest

15% benefit over a parallel algorithm for p processors with off-line schedule,

Conclusion

- **Fine grain parallelism enables efficient execution on a small number of processors**
  - Interest : portability ; mutualization of code ;
  - Drawback : needs work-first principle => algorithm design

- **Efficiency of classical work stealing relies on *work-first principle* :**
  - Implicitly defenerates a parallel algorithm into a sequential efficient ones ;
  - Assumes that parallel and sequential algorithms perform about the same amount of operations

- **Processor Oblivious algorithms based on *work-first principle***
  - Based on anytime extraction of parallelism from any sequential algorithm (may execute different amount of operations) ;
  - Oblivious: near-optimal whatever the execution context is.

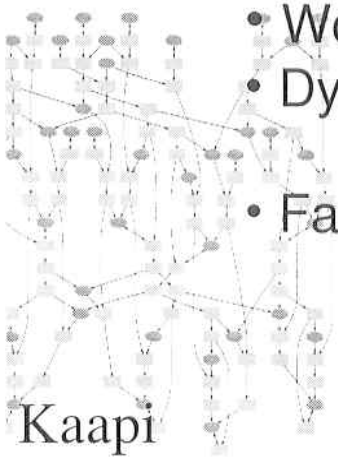
- **Generic scheme for stream computations :**

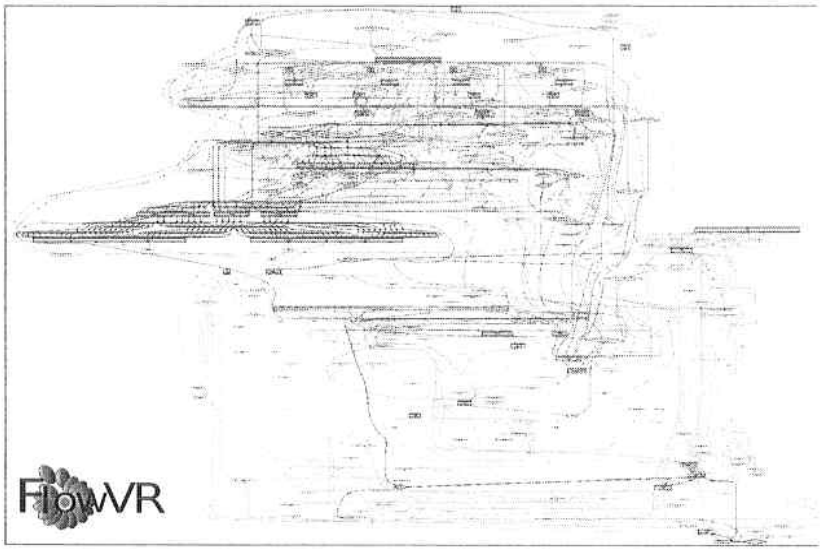
  parallelism introduce a copy overhead from local buffers to the output

  gzip / compression, MPEG-4 / H264

Kaapi (kaapi.gforce.inria.fr)

- Work stealing / work-first principle
- Dynamics Macro-dataflow :
  partitioning (Metis, …)
- Fault Tolerance (add/del resources)





FlowVR (flowvr.sf.net)

- Dedicated to interactive applicatio
- Static Macro-dataflow
- Parallel Code coupling

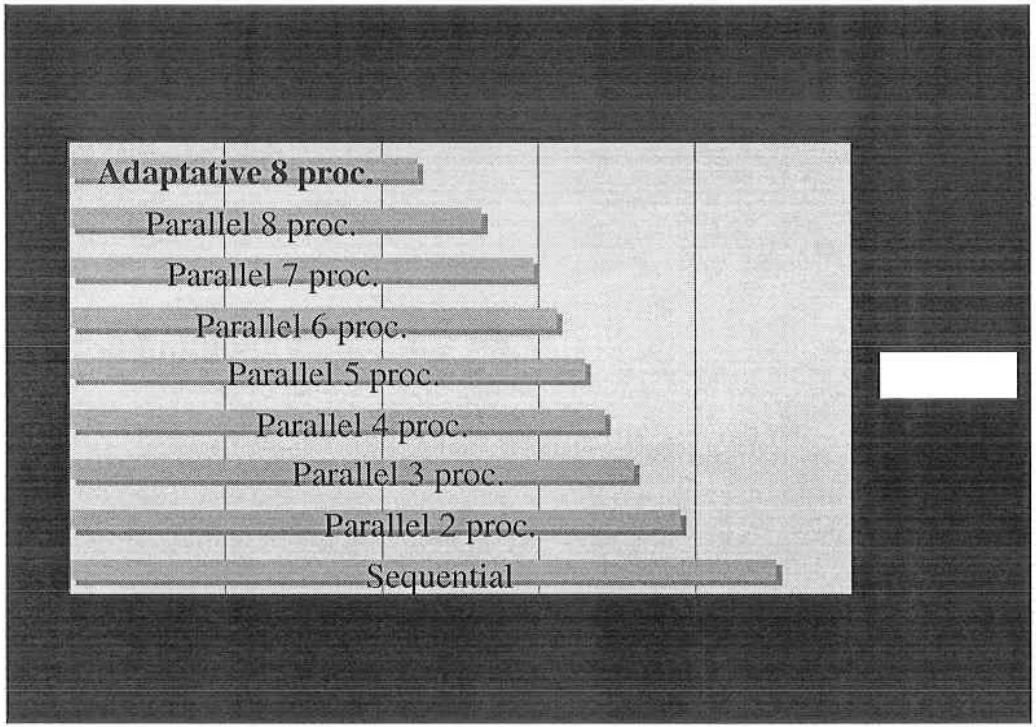
QuickTime™ et un
dŽcompresseur codec YUV420
sont requis pour visionner cette image.

[E Boyer, B Raffin 2006]

Thank you !

Back slides

he Prefix race: equential/parallel fixed/ adaptive

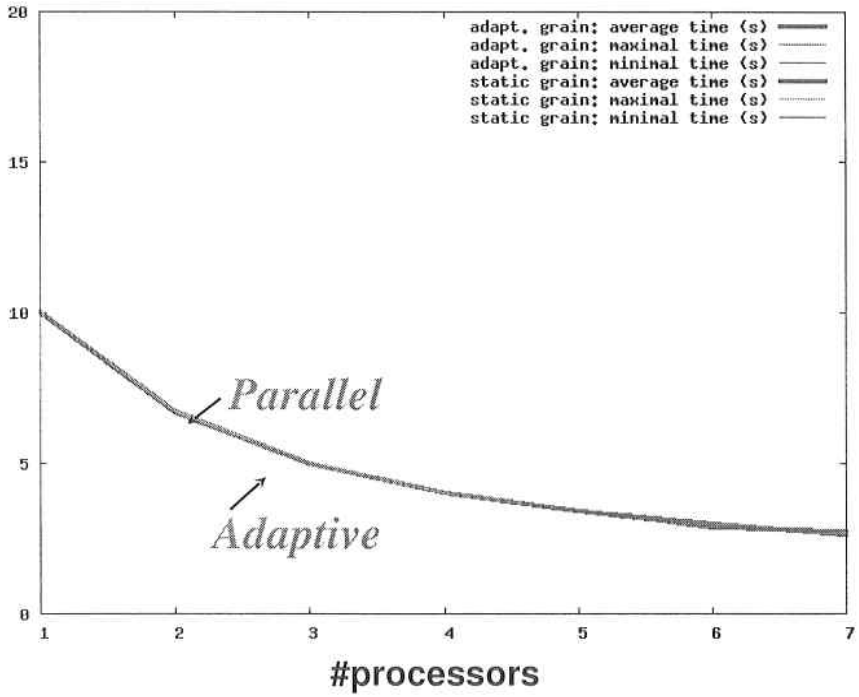


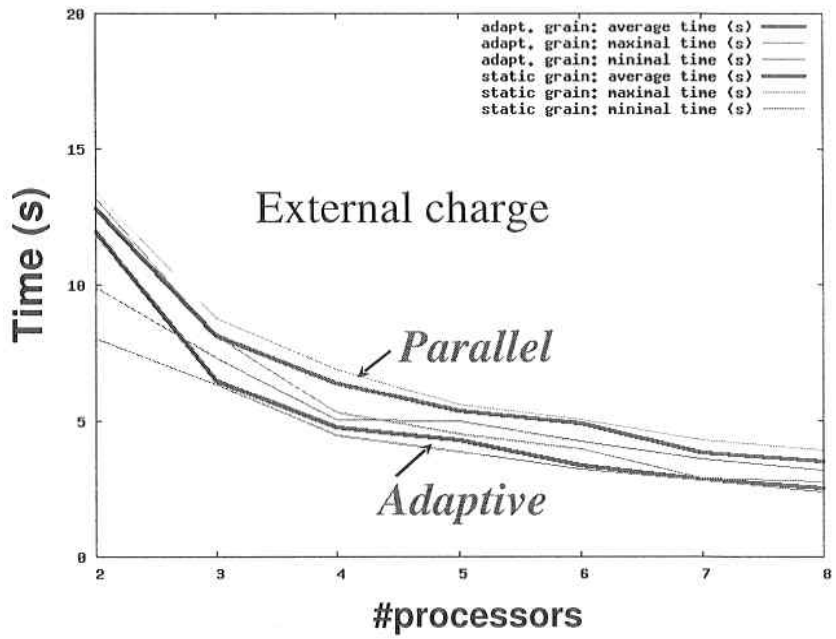
	Sequentiel	Statique					Adaptatif
		p=2	p=4	p=6	p=7	p=8	p=8
Minimum	21,83	18,16	15,89	14,99	13,92	12,51	8,76
Maximum	23,34	20,73	17,66	16,51	15,73	14,43	12,70
Moyenne	22,57	19,50	17,10	15,58	14,84	13,17	11,14
Mediane	22,58	19,64	17,38	15,57	14,63	13,11	11,01

On each of the 10 executions, adaptive completes first

Adaptive prefix : some experiments

Prefix of 10000 elements on a SMP 8 procs (IA64 / linux)





Single user context

Adaptive is equivalent to:

- sequential on 1 proc
- optimal parallel-2 proc. on 2 processors
- ...
- optimal parallel-8 proc. on 8 processors

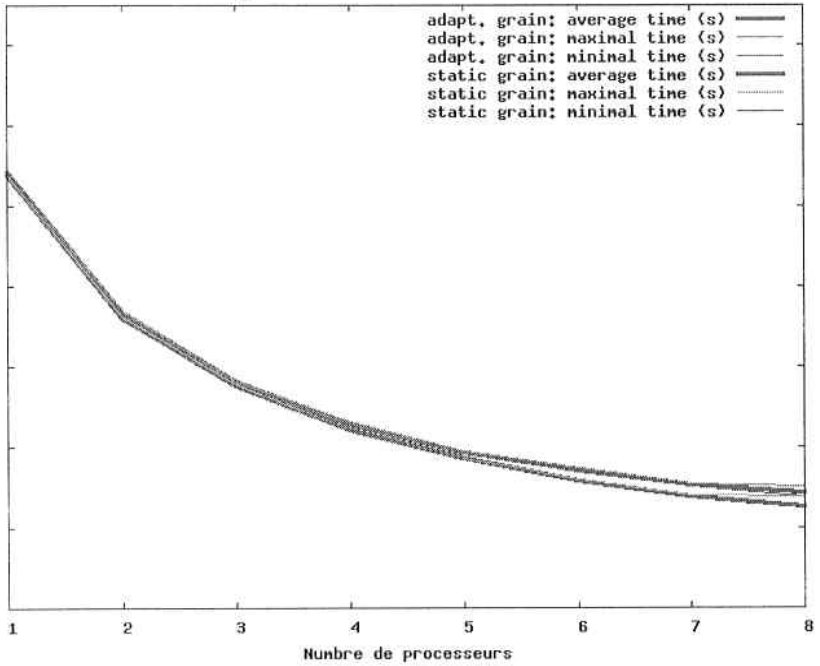
Multi-user context

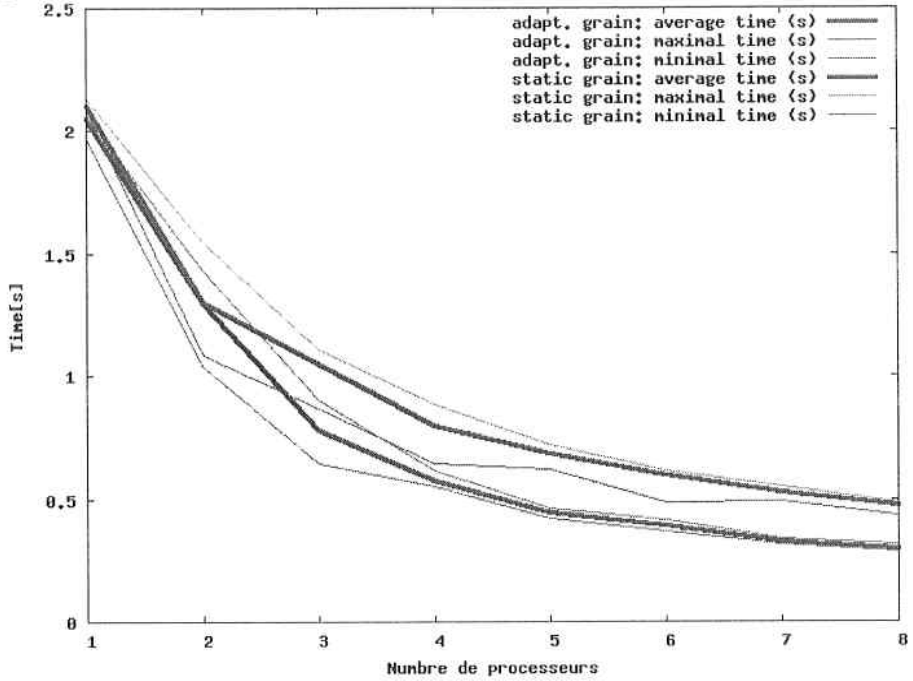
Adaptive is the fastest

15% benefit over a static grain algorithm

With * = double sum (r[i]=r[i-1] + x[i])

Finest “grain” limited to 1 page = 16384 octets = 2048 double





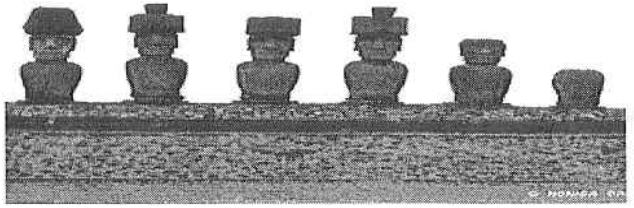
Single user

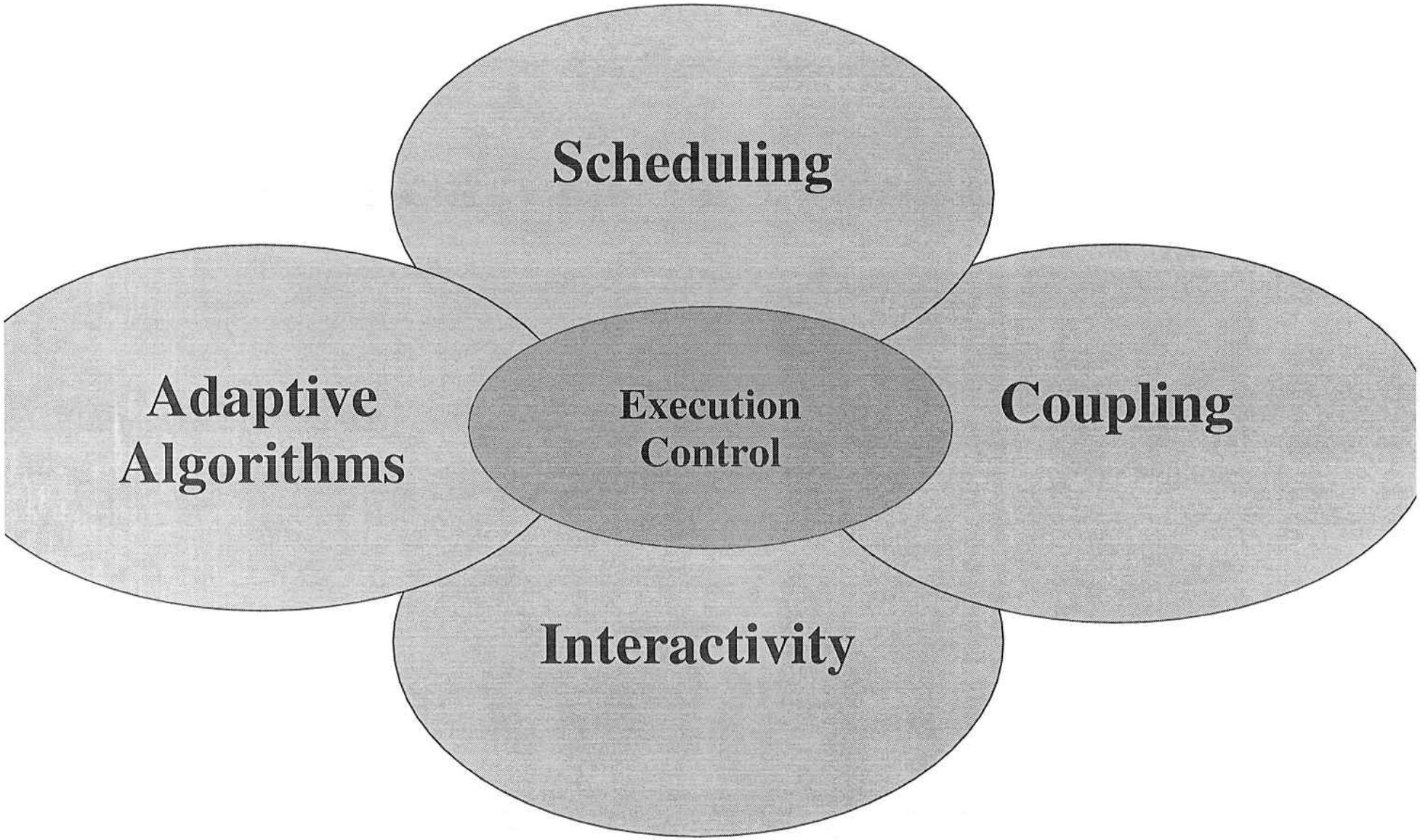
Processors with variable speeds

Remark for n=4.096.000 doubles :

- “pure” sequential : 0,20 s
- minimal ”grain” = 100 doubles : 0.26s on 1 proc
  and 0.175 on 2 procs (close to lower bound)

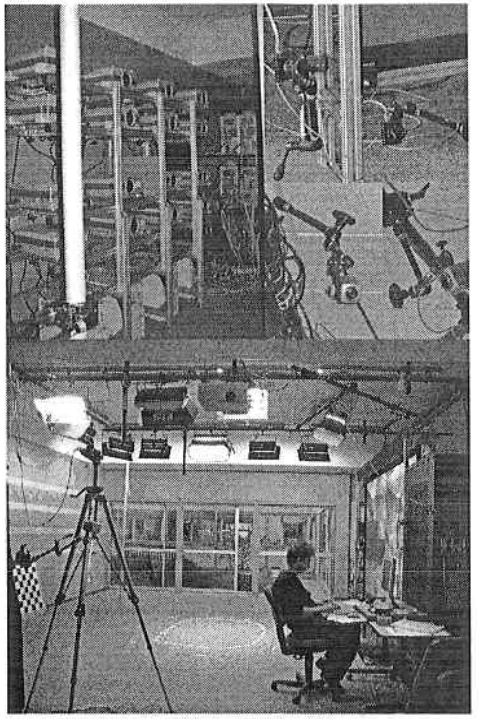
The Moais Group

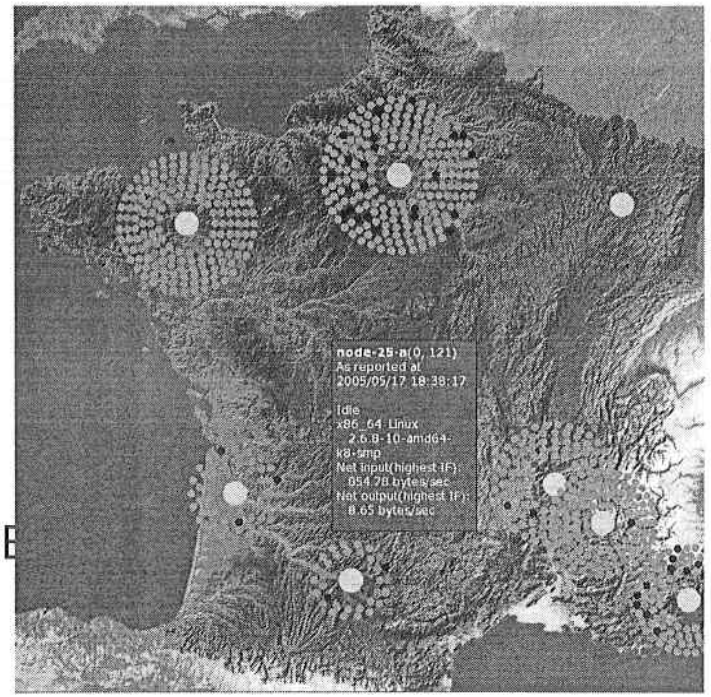




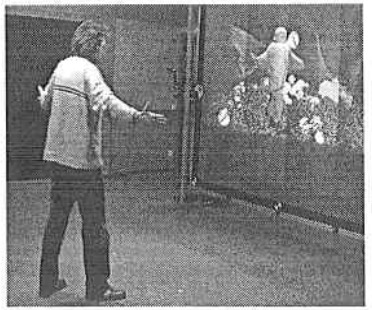
Moais Platforms

- Icluster 2 :
  - 110 dual Itanium bi-processors with Myrinet network
- GrImage ("Grappe" and Image):
  - Camera Network
  - 54 processors (dual processor cluster)
  - Dual gigabits network
  - 16 projectors display wall
- Grids:
  - Regional: Ciment
  - National: Grid5000
    - Dedicated to CS experiments
- SMPs:
  - 8-way Itanium (Bull novascale)
  - 8-way dual-core Opteron + 2 GPUs
- MPSoCs
  - Collaborations with ST Microelectronics on STE

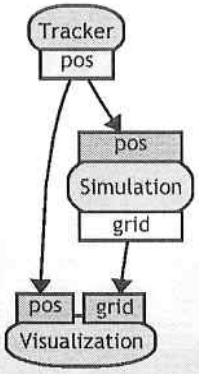




Parallel Interactive App.



- Human in the loop
- Parallel machines (cluster) to enable large interactive applications
- Two main performance criteria:
  - Frequency (refresh rate)
    - Visualization: 30-60 Hz
    - Haptic : 1000 Hz
  - Latency (makespan for one iteration)
    - Object handling: 75 ms



- A classical programming approach: data-flow model
  - Application = static graph
    - Edges: FIFO connections for data transfert
    - Vertices: tasks consuming and producing data
    - Source vertices: sample input signal (cameras)
    - Sink vertices: output signal (projector)
- One challenge:

  Good mapping and scheduling of tasks on processors