Processor-oblivious parallel algorithms with provable performances
-
Applications

Jean-Louis Roch

MOAIS

Lab. Informatique Grenoble, INRIA, France

Laboratoire Informatique et Distribution — CNRS Centre National de la Recherche Scientifique — Institut National Polytechnique de Grenoble INPG — INRIA Institut National de Recherche en Informatique et en Automatique — Grenoble 1 Université Joseph Fourier Sciences Technologie Médecine

Overview

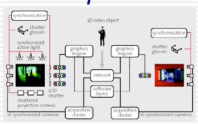
- Introduction : interactive computation, parallelism and processor oblivious
  - Overhead of parallelism : parallel prefix
- Machine model and work-stealing
- Scheme 1: Adaptive parallel algorithms
- Scheme 2: Amortizing the overhead of synchronization (Nano-loop)
- Scheme 3: Amortizing the overhead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

Interactive parallel computation?

Any application is "parallel":

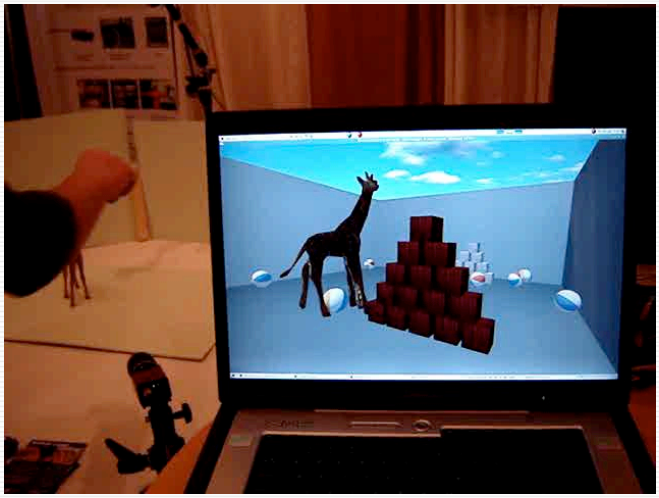
•composition of several programs / library procedures (possibly concurrent) ;

•each procedure written independently and also possibly parallel itself.



Interactive Distributed Simulation

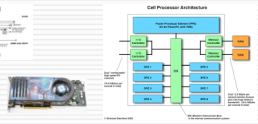
3D-reconstruction
+ simulation
+ rendering
[B Raffin &E Boyer]
- 1 monitor
- 5 cameras,
- **6** PCs



New parallel supports

from small too large

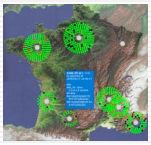
- **Parallel chips & multi-core architectures**:
  - **MPSoCs** (Multi-Processor Systems-on-Chips)
  - **GPU** : graphics processors (and programmable: Shaders; Cuda SDK)
  - Dual Core processors (Opterons, Itanium, etc.)
  - Heteregoneous multi-cores : **CPUs + GPUs + DSPs+ FPGAs** (Cell)



- **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

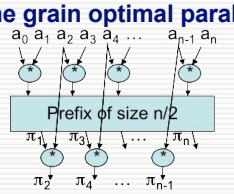
$$\pi_i = \prod_{k=0}^{i} a_k$$

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

performs only **n** *operations*

- **Fine grain optimal parallel** algorithm :

[Ladner-Fisher-81]

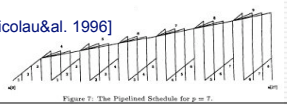


Critical time = 2. *log* n
but performs **2.n ops**

Parallel requires twice more operations than sequential !!

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

[Nicolau&al. 1996]



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

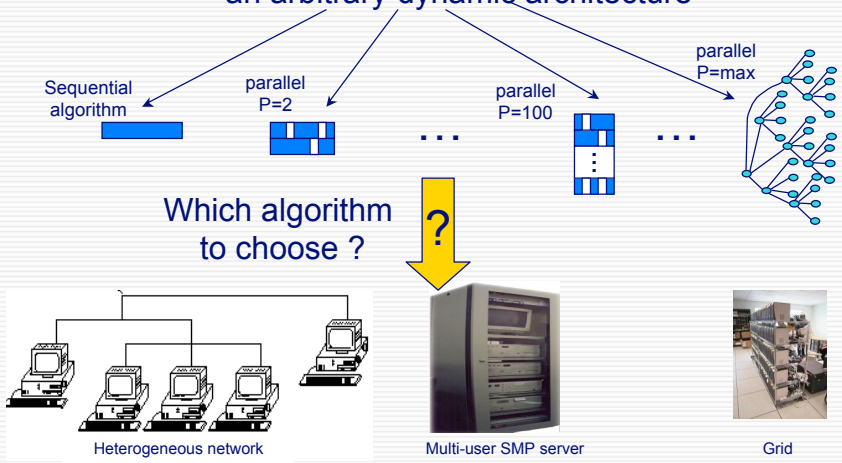
⇓ [Fitch80]

#operations > $2n - d$

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

The problem

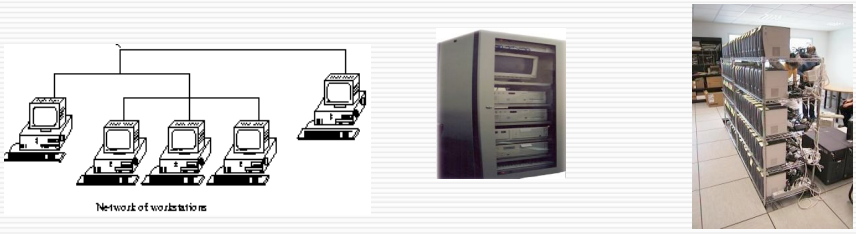
To design a single algorithm that computes efficiently prefix(a) on an arbitrary dynamic architecture



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

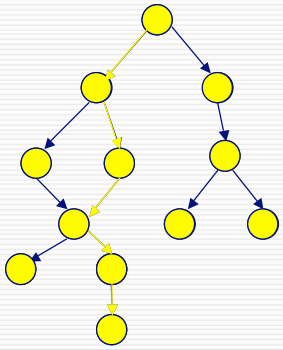
Processor speeds are assumed to change arbitrarily and adversarially:

model [Bender,Rabin 02] $\Pi_i(t)$ = ***instantaneous speed*** *of processor i at time t*
(in #unit operations per second)

Assumption : $\text{Max}_{i,t}\{ \Pi_i(t) \} < \text{constant} \cdot \text{Min}_{i,t}\{ \Pi_i(t) \}$

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)

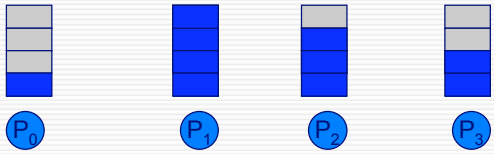
For any greedy *maximum utilization* schedule:

[Graham69, Jaffe80, Bender-Rabin02]

$$\text{makespan} \le \frac{W}{p.\Pi_{ave}} + \left(1 - \frac{1}{p}\right)\frac{D}{\Pi_{ave}}$$

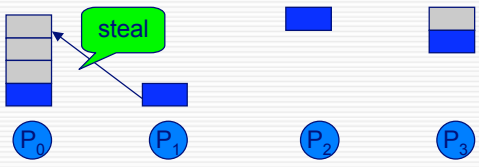
The work stealing algorithm

- **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 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, [Acar,Blelloch, Blumofe02, BenderRabin02]
    - #steals < $p.D$
    - execution time $\leq \frac{W}{p.\Pi_{ave}} + O\left(\frac{D}{\Pi_{ave}}\right)$
  - **Interest**:
    if ***W* independent of *p*** and ***D* is small**, work stealing achieves **near-optimal** schedule

Work 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 $\leq \frac{W}{p.\Pi_{ave}} + O\left(\frac{D}{\Pi_{ave}}\right)$ **(fine grain)**

Expensive in general (fine grain)
But small overhead if a small number of tasks
(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" [Frigo 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 sequential function call

At 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 }
```

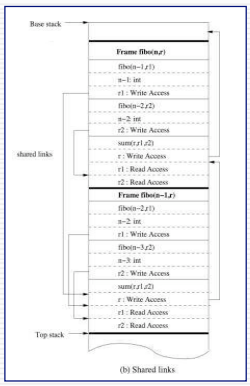
```
1   int fib (int n)
2   {
3       fib_frame *f;                  frame pointer
4       f = alloc(sizeof(*f));         allocate frame
5       f->sig = fib_sig;              initialize frame
6       if (n<2) {
7           free(f, sizeof(*f));       free frame
8           return n;
9       }
10      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.fr : 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 successful 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  } ;
```

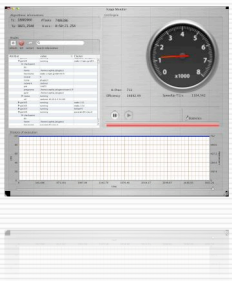


- *Kaapi won the 2006 award "Prix special du Jury" for the best performance at NQueens contest, Plugtests-Grid&Work'06, Nice, Dec.1, 2006 [Gautier-Guelton] on Grid'5000 1458 processors with different speeds.*

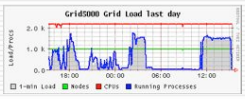
N-queens: Takaken C sequential code parallelized in C++/Kaapi

- *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, 2006*

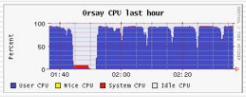
- Some facts [on *on Grid'5000*, a grid of processors of heterogeneous speeds]
  - NQueens(21) in 78 s on about 1000 processors
  - Nqueens (22) in 502.9s on 1458 processors
  - Nqueens(23) in 4435s on 1422 processors [~24.10³³ solutions]
  - 0.625% idle time per processor
  - < 20s to deploy up to 1000 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)

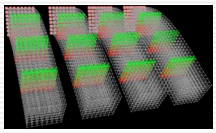


CPU

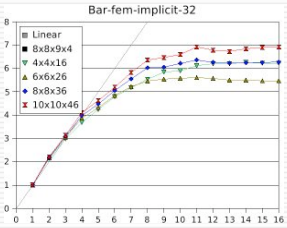
6 instances Nqueens(22)

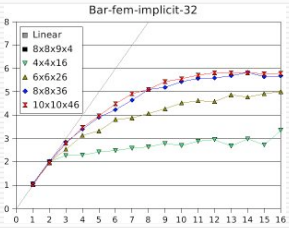
Network

Experimental results on SOFA [CIMIT-ETZH-INRIA]



[Allard 06]



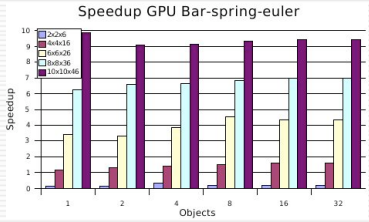


Kaapi (C++, ~500 lines)

Cilk (C, ~240 lines)

Preliminary results on GPU NVIDIA 8800 GTX

- speed-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



Overview

- Introduction : interactive computation, parallelism and processor oblivious
  - Overhead of parallelism : parallel prefix
- Machine model and work-stealing
- **Scheme 1: Extended work-stealing : concurently sequential and parallel**

19

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 performances**
  - 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

20

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

- General approach**: to** mix **both**
  - a **sequential** algorithm with optimal work $\boldsymbol{W_1}$
  - and a fine grain **parallel** algorithm with minimal critical time $\boldsymbol{W_\infty}$
- Folk technique **:** *parallel, than sequential*
  - Parallel algorithm until a certain « grain »; then use the sequential one
  - Drawback : $\boldsymbol{W_\infty}$ 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_∞ **small** and $W_1 = O(W_{seq})$
  - 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(

21

Extended work-stealing: concurrently sequential and parallel

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

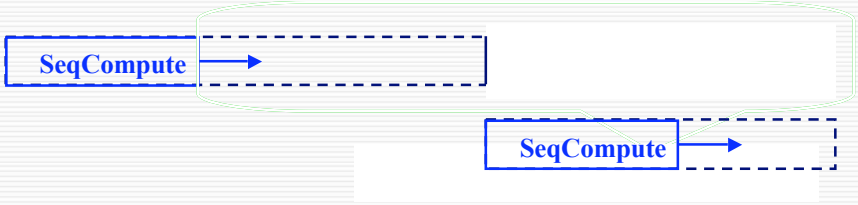
Instead of optimizing the **sequential execution** of the **best parallel** algorithm, let optimize the **parallel execution** of the ***best sequential*** algorithm

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)



22

Extended work-stealing : concurrently sequential and parallel

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

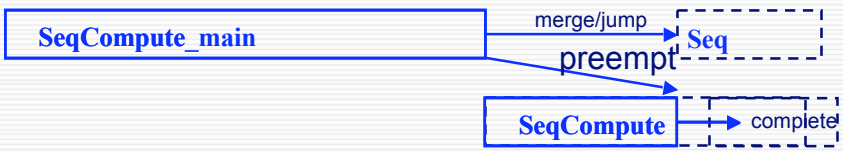
Instead of optimizing the **sequential execution** of the **best parallel** algorithm, let optimize the **parallel execution** of the ***best sequential*** algorithm

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

Overview

- Introduction : interactive computation, parallelism and processor oblivious
  - Overhead of parallelism : parallel prefix
- Machine model and work-stealing
- Scheme 1: Extended work-stealing : concurently sequential and parallel
- **Scheme 2: Amortizing the overhead of synchronization (Nano-loop)**

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 optimal 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 without lock
  - Efficient mutual exclusion between sequential process and parallel work-stealer
- ***Self-adaptive granularity***

25

Interactive 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

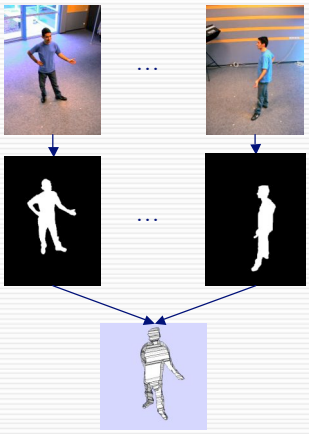
26

Parallel 3D Modeling

3D Modeling :

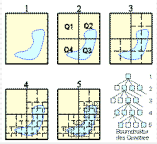
build a 3D model of a scene from a set of calibrated images

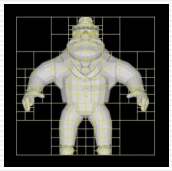
On-line 3D modeling for interactions: 3D modeling from multiple video streams (30 fps)

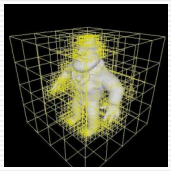


Octree Carving [L. Soares 06]

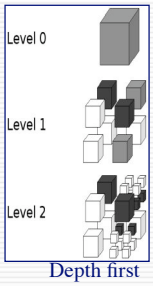
A classical recursive anytime 3D modeling algorithm.





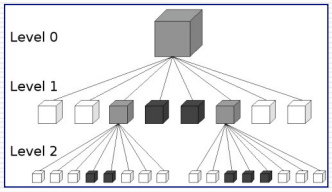


Standard algorithms with time control:



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

28

Width first parallel octree carving

Well suited to work-stealing

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

- non-predictable work, non predictable grain :

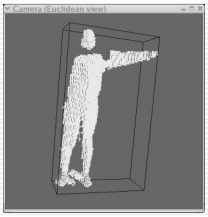
For cache locality, each level is processed by a self-adaptive grain :

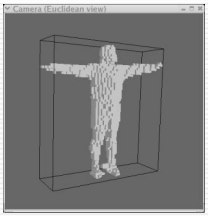
"sequential iterative" / "parallel recursive split-half"

Octree needs to be "balanced" when stopping:

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

Unbalanced





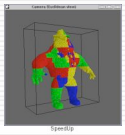
Balanced

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

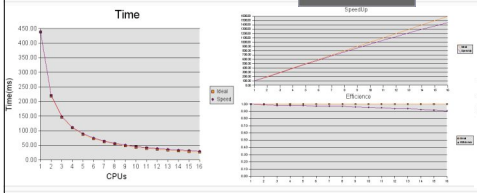
- goes at most one level deeper : $\mathbf{l\ d_s - d_p\ l \leq 1}$;
- computes at most : $\mathbf{n_s \leq n_p + O(\log n_s)}$.

Results

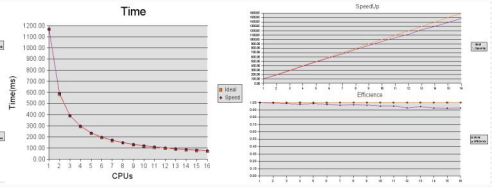
[L. Soares 06]



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



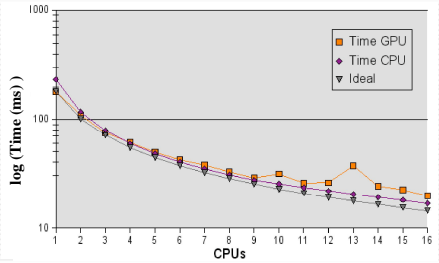
8 cameras, levels 2 to 10



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



Overview

- Introduction : interactive computation, parallelism and processor oblivious
  - Overhead of parallelism : parallel prefix
- Machine model and work-stealing
- Scheme 1: Extended work-stealing : concurently sequential and parallel
- Scheme 2: Amortizing the overhead of synchronization (Nano-loop)
- **Scheme 3: Amortizing the overhead of parallelism (Macro-loop)**

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 may increase 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

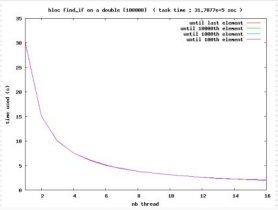
Here, by **amortizing** the work induced by the extract_par operation, ensuring this **work to be *small*** enough :
• Either w.r.t the -useful- **work already performed**
• Or with respect to the - useful - **work yet to performed** (if known)
• or both.

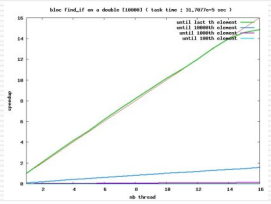
Eg : find_if (first, last, predicate) :
• only the work already performed is known (on-line)
• then prevent to assign more than $\alpha(W_{done})$ operations to work-stealers
• Choices for $\alpha(n)$ **:**
  • **n/2 : similar to Floyd's iteration** (approximation ratio = 2)
  • **n/log* n :** to ensure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

Results on find_if

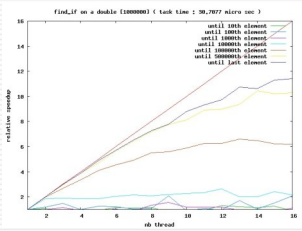
[S. Guelton]

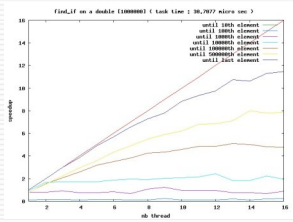
N doubles : time predicate ~ 0.31 ms





With no amortization macroloop





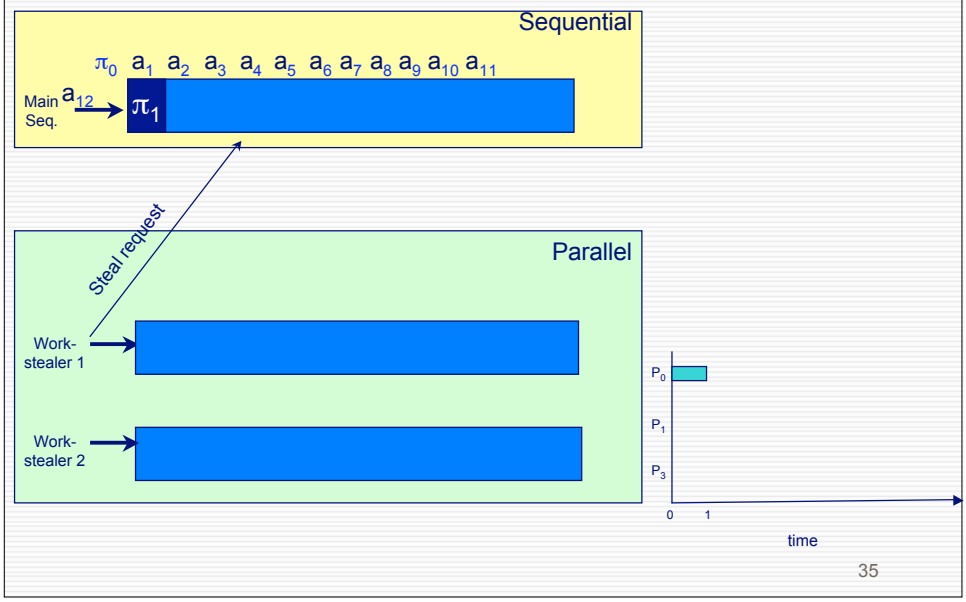
With amortization macroloop

5. Putting things together
processor-oblivious prefix computation

Parallel algorithm based on :

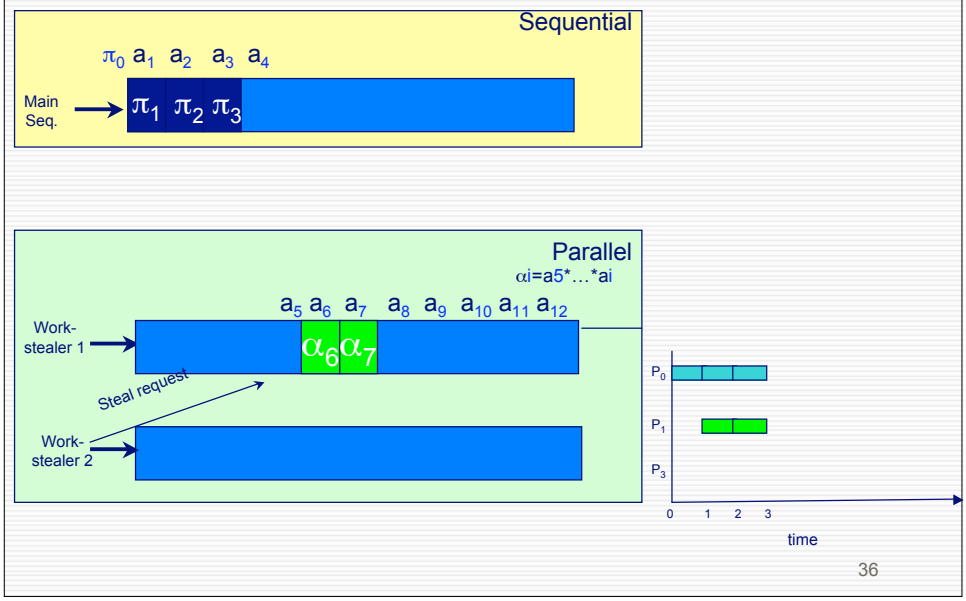
- **compute-seq / extract-par scheme**
- nano-loop for compute-seq
- macro-loop for extract-par

P-Oblivious Prefix on 3 proc.



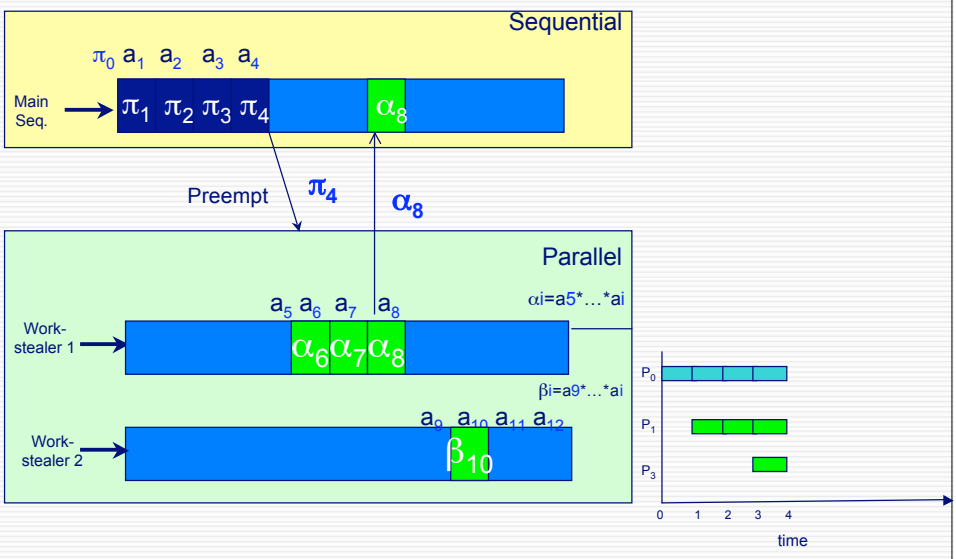
35

P-Oblivious Prefix on 3 proc.



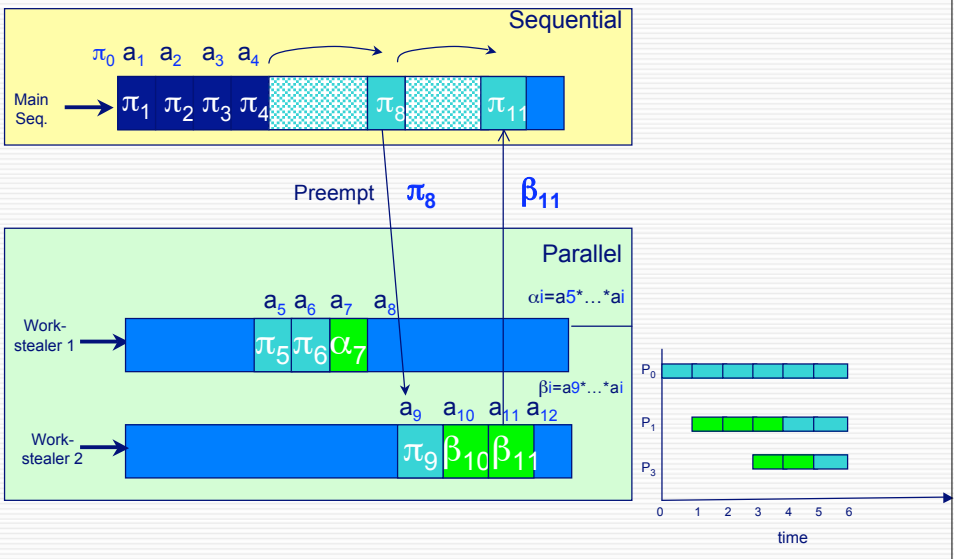
36

P-Oblivious Prefix on 3 proc.



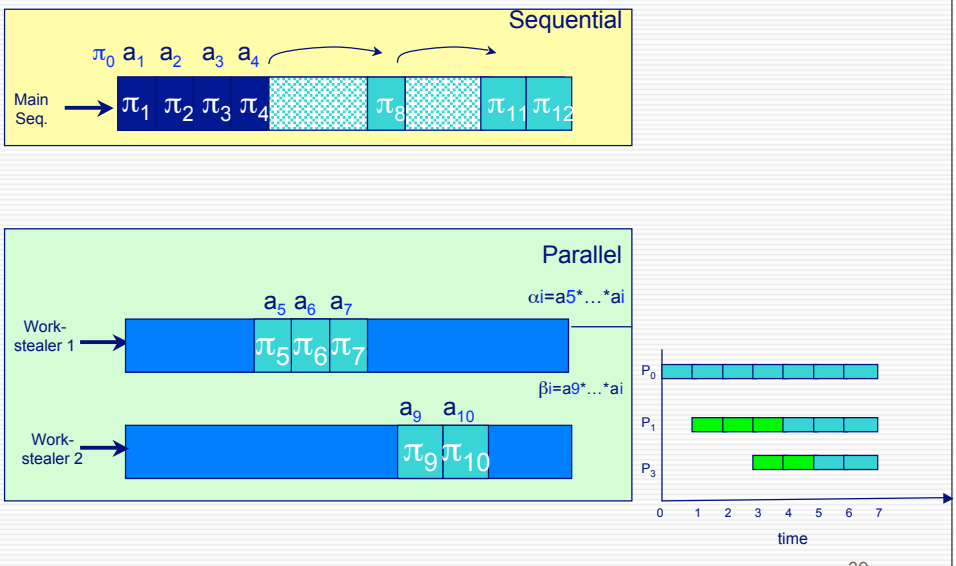
37

P-Oblivious Prefix on 3 proc.



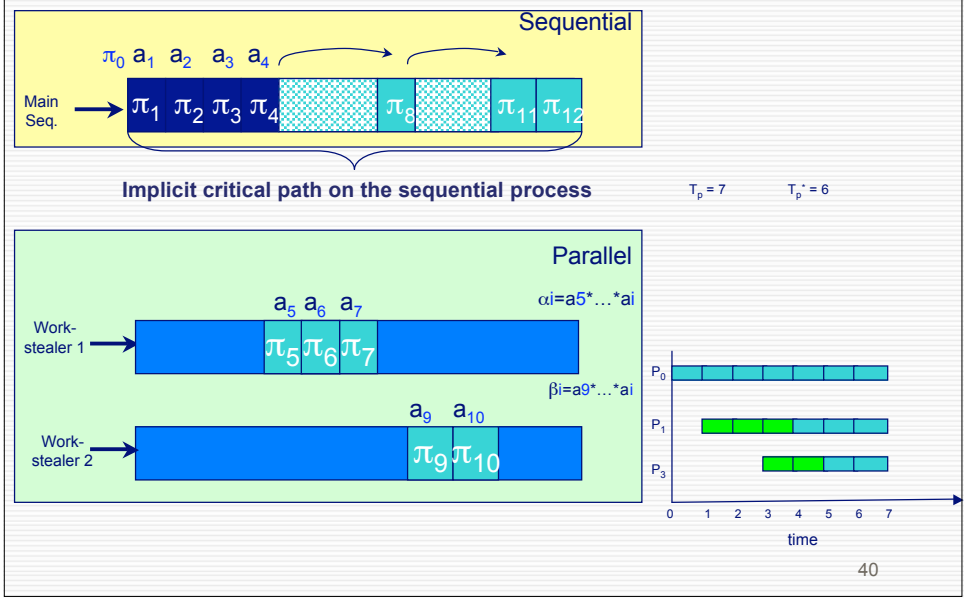
38

P-Oblivious Prefix on 3 proc.



39

P-Oblivious Prefix on 3 proc.



40

Analysis of the algorithm

- Execution time $\leq \frac{2n}{(p+1).\Pi_{ave}} + O\left(\frac{\log n}{\Pi_{ave}}\right)$

  Lower bound

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

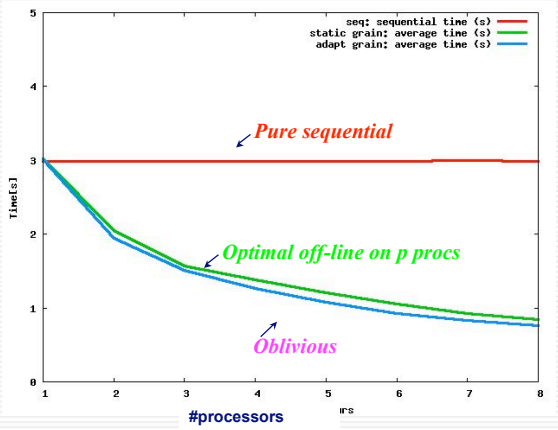
    => enables to bound the whole number X of operations performed and the overhead of parallelism = (s+X) - #ops_optimal

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

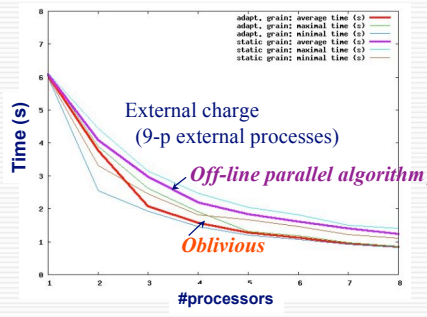
- Less sensitive to system overhead : even better than the theoretically "optimal" off-line parallel algorithm on p processors

Results 2/2

[D Traore]

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

Multi-user context :



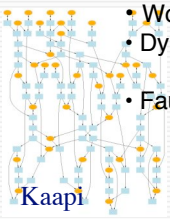
Multi-user context :
Additional external charge: (9-p) additional external dummy processes are concurrently executed

Processor-oblivious prefix computation is always the fastest
15% benefit over a parallel algorithm for p processors with off-line schedule,

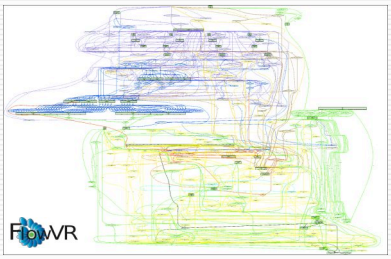
Conclusion

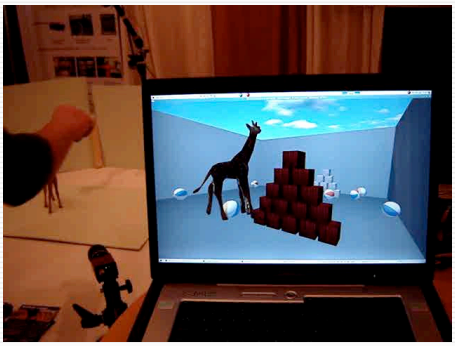
- **Fine grain parallelism enables efficient execution on a small number of processors**
  - Interest : portability ; mutualization of code ;
  - Drawback : needs work-first principle => algorithm design
- **Efficiency of classical work stealing relies on *work-first principle* :**
  - Implicitly defenerates 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.gforge.inria.fr)



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





[E Boyer, B Raffin 2006]

FlowVR (flowvr.sf.net)

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

Thank you !