Parallel Algorithms

Design
and
Implementation

Jean-Louis.Roch at imag.fr

MOAIS / Lab. Informatique Grenoble, INRIA, France

Overview

- Machine model and work-stealing
- Work and depth
- Fundamental theorem : Work-stealing theorem
- Parallel divide & conquer
- Examples
    - Accumulate
    - Monte Carlo simulations

- Part2: Work-first principle - Amortizing the overhead of parallelism
- Prefix/partial sum
    - Sorting and merging

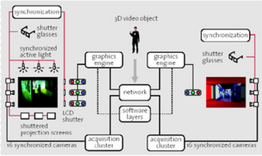
- Part3: Amortizing the overhead of synchronization and communications
- Numerical computations : FFT, marix computations; Domain decompositions

Interactive parallel computation?

Any application is "parallel":

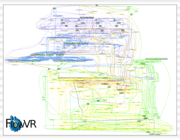
•composition of several programs / library procedures (possibly concurrent) ;

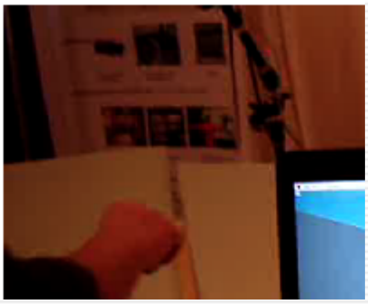
•each procedure written independently and also possibly parallel itself.



Interactive Distributed Simulation

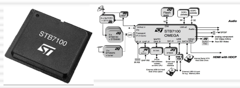
3D-reconstruction
+ simulation
+ rendering
[B Raffin &E Boyer]
- 1 monitor
- 5 cameras,
- **6** PCs

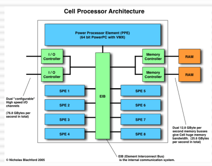




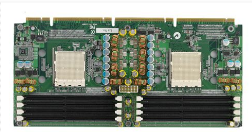
New parallel supports from small too large

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

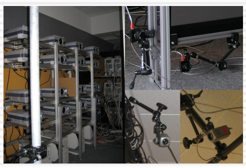




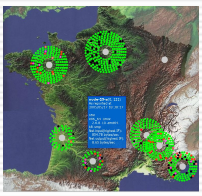
- **Commodity SMPs**:
  - 8 way PCs equipped with multi-core processors (AMD Hypertransport) + 2 GPUs



- **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

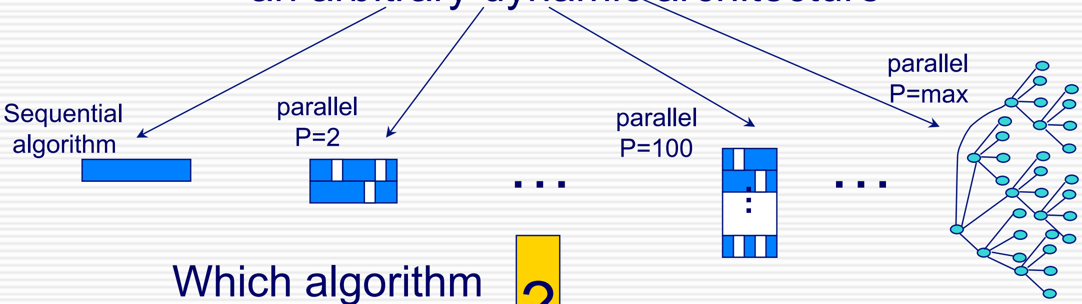


- **Dedicated platforms**: eg Virtual Reality/Visualization Clusters:
  - Scientific Visualization and Computational Steering
  - PC clusters + graphics cards + multiple I/O devices (cameras, 3D trackers, multi-projector displays)

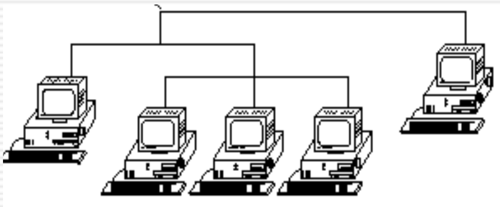
Grimage platform

The problem

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



Which algorithm to choose ?



Heterogeneous network

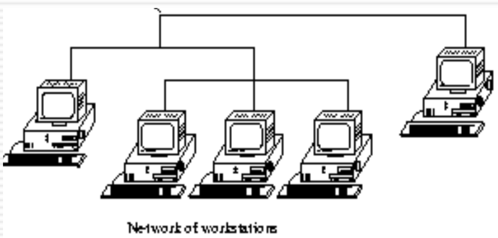
Multi-user SMP server

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:

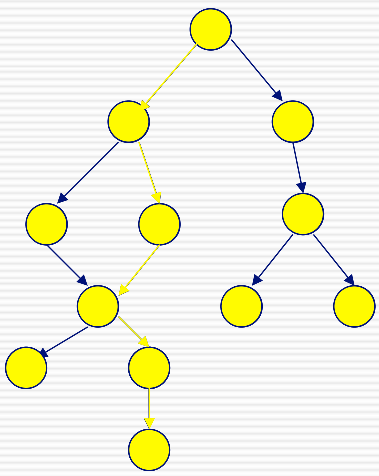
model [Bender,Rabin 02] $\Pi_i(t)$ = ***instantaneous speed*** *of processor i at time t*

(in #unit operations per second)

Assumption : $\mathrm{Max}_{i,t}\{\,\Pi_i(t)\,\} <$ *constant .* $\mathrm{Min}_{i,t}\{\,\Pi_i(t)\,\}$

Def: for a computation with duration T

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

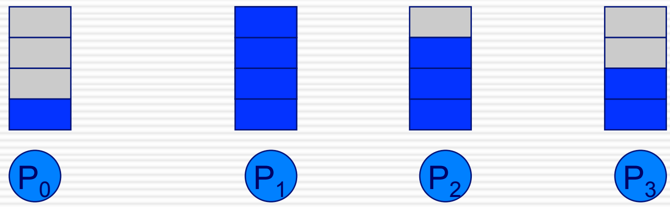
For any greedy *maximum utilization* schedule:

[Graham69, Jaffe80, Bender-Rabin02]

$$\textit{makespan} \leq \frac{W}{p\,\Pi_{ave}} + \left(1 - \frac{1}{p}\right)\frac{D}{\Pi_{ave}}$$

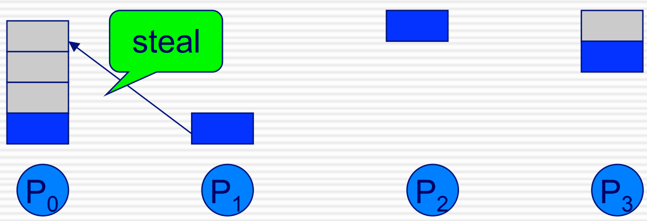
The work stealing algorithm

- **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 = O(*p.D*)** **and 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

Proof

- **Any parallel execution can be represented by a binary tree:**
  - Node with 0 child = TERMINATE instruction
    - End of the current thread
  - Node with 1 son = sequential instruction
  - Node with 2 sons: parallelism = instruction that
    - Creates a new (ready) thread
      - eg fork, thread_create, spawn, …
    - Unblocks a previously blocked thread
      - eg signal, unlock, send

Proof (cont)

- Assume the local ready task queue is stored in an array: each ready task is stored according to its depth in the binary tree
- **On processor i at top t :**
  - $H_i(t)$ = the index of the oldest ready task
- **Prop 1:** When non zero, $H_i(t)$ is increasing
- **Prop 2: H(t) =** $\text{Min}_{(i \text{ active at } t)}\{ H_i(t) \}$ is increasing
- **Prop 3: Each steal request on i makes H_i strictly increases**.
- **Corollary:** if at each steal, the victim is a processor i with minimum H_i then
  #steals ≤ (p-1).Height(tree) ≤ (p-1).D

Proof (randomized, general case)

- **Group the steal operations in blocks of consecutive steals:** [Coupon collector problem]
  - Consider p.log p consecutive steals requests after top t, Then with probability > ½, **any** active processor at t have been victim of a steal request.
    - Then $\text{Min}_i\ H_i$ has increased of at least 1
- **In average, after (2.p.log p.M) consecutive steals requests, $\mathbf{Min_i\ H_i \geq M}$**
  - Thus, in average, after (2.p.log p.D) steal requests, the execution is completed !
- [Chernoff bounds] **With high probability (w.h.p.),**
  - **#steal requests = O(p.log p.D)**

Proof (randomized, additional hyp.)

- **With additional hypothesis:**
    - Initially, only one active processor
    - When several steal requests are performed on a same victim processor at the same top,
      only the first one is considered (others fail)
  - [Balls&Bins] Then **#steal requests = O(p.D)** w.h.p.
- **Remarks:**
  - This proof can be extended to
    - asynchronous machines (synchronization = steal)
    - Other steal policies then steal the "topmost=oldest" ready tasks, but with impact on the bounds on the steals

Steal requests and execution time

- **At each top, a processor j is**
  - Either active: performs a “work” operation
    - Let wj be the number of unit work operations by j
  - Either idle: performs a steal requests
    - Let sj be the number of unit steal operations by j

- **Summing on all p processors :**

$$\text{Execution time} \leq \frac{W}{p\Pi_{ave}} + O\left(\frac{D}{\Pi_{ave}}\right)$$

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

```
cilk int fib (int n)
{
     if (n < 2) return n;
     else
     {
        int x, y;

        x = spawn fib (n-1);
        y = spawn fib (n-2);

        sync;

        return (x+y);
     }
}
```

```
int fib (int n)
{
    fib_frame *f;                 frame pointer
    f = alloc(sizeof(*f));        allocate frame
    f->sig = fib_sig;             initialize frame
    if (n<2) {
        free(f, sizeof(*f));      free frame
        return n;
    }
    else {
        int x, y;
        f->entry = 1;             save PC
        f->n = n;                 save live vars
        *T = f;                   store frame pointer
        push();                   push frame
        x = fib (n-1);            do C call
        if (pop(x) == FAILURE)    pop frame
            return 0;             frame stolen
        ...                       second spawn
        ;                         sync is free!
        free(f, sizeof(*f));      free frame
        return (x+y);
    }
}
```

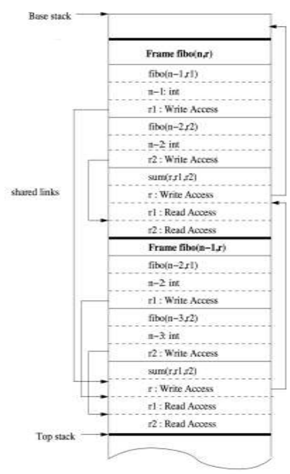
- *won the 2006 award "Best Combination of Elegance and Performance" at HPC Challenge Class 2, SC'06, Tampa, Nov 14 2006 [Kuszmaul] on SGI ALTIX 3700 with 128 bi-Ithanium]*

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)

```
struct sum {
   void operator()(Shared_r < int > a,
                   Shared_r < int > b,
                   Shared_w < int > r )
    { r.write(a.read() + b.read()); }
 } ;

 struct fib {
  void operator()(int n, Shared_w<int> r)
  { if (n <2) r.write( n );
    else
    { int r1, r2;
      Fork< fib >() ( n-1, r1 ) ;
      Fork< fib >() ( n-2, r2 ) ;
      Fork< sum >() ( r1, r2, r ) ;
    }
  }
} ;
```

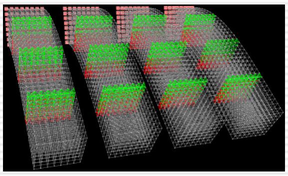


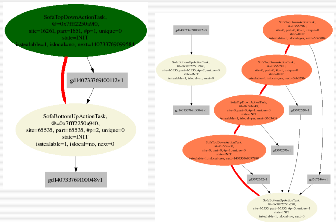
(b) Shared links

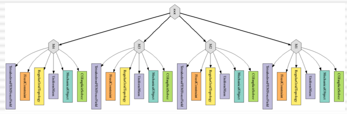
- *Kaapi won the 2006 award "Prix special du Jury" for the best performance at NQueens contest, Plugtests-Grid&Work'06, Nice, Dec.1, 2006 [Gautier-Guelton] on Grid'5000 1458 processors with different speeds.*

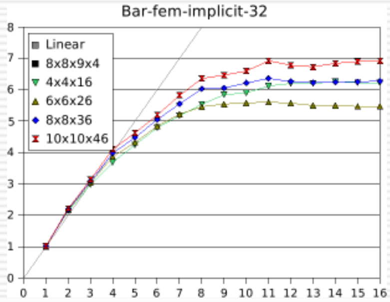
Experimental results on SOFA [CIMIT-ETZH-INRIA]

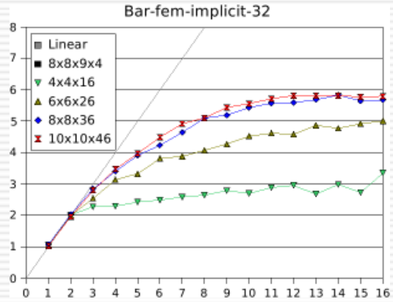
[Allard 06]







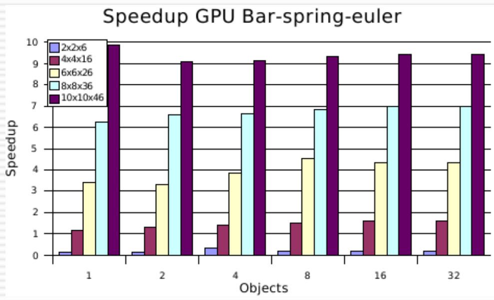




Kaapi (C++, ~500 lines) Cilk (C, ~240 lines)

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



Algorithm design

$$\text{Execution time} \leq \frac{W}{p.\Pi_{ave}} + O\left(\frac{D}{\Pi_{ave}}\right)$$

- **From work-stealing theorem, optimizing the execution time by building a parallel algorithm with both**
  - **$W = T_{seq}$**

  and

  - small depth D

- Double criteria
  - **Minimum work W** (ideally T_{seq})
  - **Small depth D**: ideally polylog in the work: $= \log^{O(1)} W$

Examples

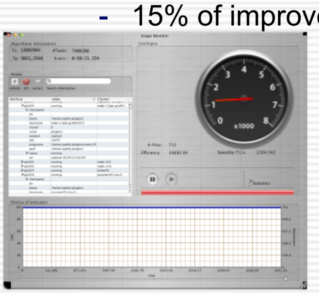
- Accumulate

- => Monte Carlo computatons

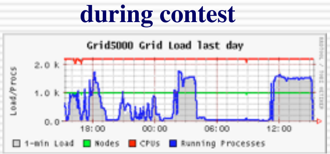
Example: Recursive and Monte Carlo computations

- *X Besseron, T. Gautier, E Gobet, &G Huard won the nov. 2008 Plugtest-Grid&Work'08 contest – Financial mathematics application (Options pricing)*

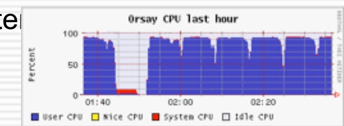
- In 2007, the team won the Nqueens contest; Some facts [on *on Grid'5000*, a grid of processors of heterogeneous speeds]
  - NQueens(21) in 78 s on about 1000 processors
  - Nqueens (22) in 502.9s on 1458 processors
  - Nqueens(23) in 4435s on 1422 processors [~24.10^{33} solutions]
  - 0.625% idle time per processor
  - < 20s to deploy up to 1000 processes on 1000 machines [Taktuk, Huard]
  - 15% of improvement of the sequential due to C++ (te



Grid'5000 utilization during contest



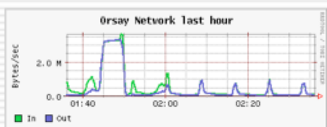
Competitor X
Competitor Y
Competitor Z
Grid'5000 free
N-Queens(23)



CPU

6 instances Nqueens(22)

Network



Algorithm design

- **Cascading divide & Conquer**

  - $W(n) \le a.W(n/K) + f(n)$ with $a>1$
    - If $f(n) << n^{\log_K a}$ => $W(n) = O(n^{\log_K a})$
    - If $f(n) >> n^{\log_K a}$ => $W(n) = O(f(n))$
    - If $f(n) = \Theta(n^{\log_K a})$ => $W(n) = O(f(n) \log n)$
  - $D(n) = D(n/K) + f(n)$
    - If $f(n) = O(\log^i n)$ => $D(n) = O(\log^{i+1} n)$
  - $D(n) = D(\mathrm{sqrt}(n)) + f(n)$
    - If $f(n) = O(1)$ => $D(n) = O(\log\log n)$
    - If $f(n) = O(\log n)$ => $D(n) = O(\log n)$!!

Examples

- Accumulate
- Monte Carlo computations
- **Maximum on CRCW**
- Matrix-vector product – Matrix multiplication -- Triangular matrix inversion
- **Exercise: parallel merge and sort**
- Next lecture: Find, Partial sum, adaptive parallelism, communications

Algorithm design

$$\text{Execution time} \leq \frac{W}{p.\Pi_{ave}} + O\left(\frac{D}{\Pi_{ave}}\right)$$

- **From work-stealing theorem, optimizing the execution time by building a parallel algorithm with both**
  - **$W = T_{seq}$**

  and

  - small depth D

- Double criteria
  - **Minimum work W** (ideally T_{seq})
  - **Small depth D**: ideally polylog in the work: $= \log^{O(1)} W$