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
  - Parallel divide & conquer
  - Examples
    - Accumulate
    - Monte Carlo simulations
    - Prefix/partial sum

- Work-stealing theorem

- Course 2: Work-first principle - Amortizing the overhead of parallelism
  - Sorting and merging

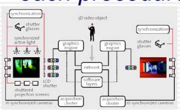
- Course 3: Amortizing the overhead of synchronization and communications
  - Numerical computations : FFT, marix computations; Domain decompositions

Interactive parallel computation?

Any application is "parallel":

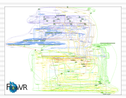
•composition of several programs / library procedures (possibly concurrent) ;

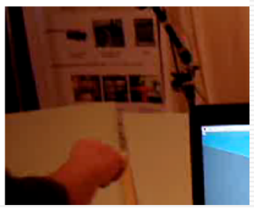
•each procedure written independently and also possibly parallel itself.



Interactive Distributed Simulation

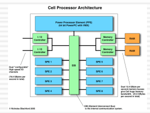
3D-reconstruction
+ simulation
+ rendering
[B Raffin &E Boyer]
- 1 monitor
- 5 cameras,
- **6** PCs





New parallel supports from small too large

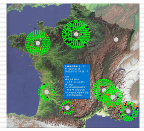
- **Parallel chips & multi-core architectures**:
  - **MPSoCs** (Multi-Processor Systems-on-Chips)
  - **GPU** : graphics processors (and programmable: Shaders; Cuda SDK)
  - Dual Core processors (Opterons, Itanium, etc.)
  - Heteregeneous multi-cores : **CPUs + GPUs + DSPs+ FPGAs** (Cell)



- **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

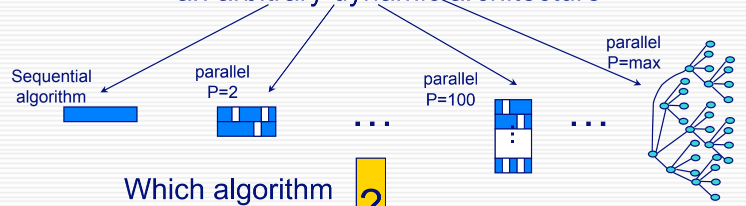


- **Dedicated platforms**: eg Virtual Reality/Visualization Clusters:
  - Scientific Visualization and Computational Steering
  - PC clusters + graphics cards + multiple I/O devices (cameras, 3D trackers, multi-projector displays)

Grimage platform

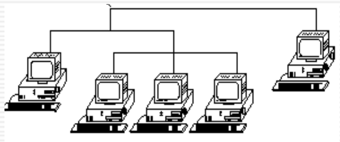
The problem

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



Which algorithm to choose ?

?



Heterogeneous network

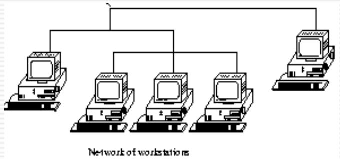
Multi-user SMP server

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

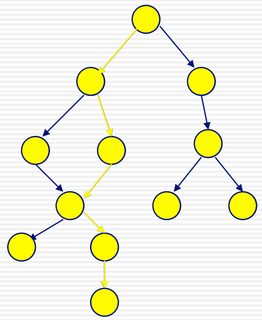
- Heterogeneous machine model and work-depth framework
- Distributed work stealing
- Work-stealing implementation : work first principle
- Examples of implementation and programs:
  Cilk , Kaapi/Athapascan
- Application: Nqueens on an heterogeneous grid

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

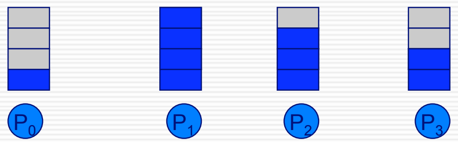
"Depth" D = #operations on a critical path
(~parallel "time" on ∞ resources)

For any greedy *maximum utilization* schedule:
[Graham69, Jaffe80, Bender-Rabin02]

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

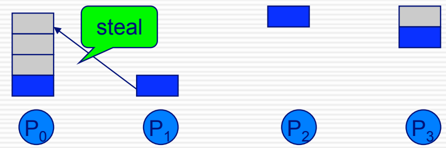
The work stealing algorithm

- **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 < $p.D$
  - 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 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

```
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);
    }
}
```

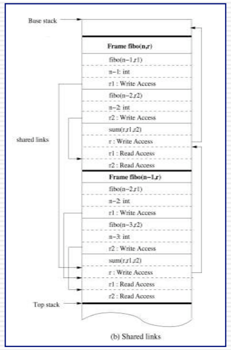
```
int fib (int n)
{
    fib_frame *f;                    frame pointer
    f = alloc(sizeof(*f));           allocate frame
    f->sig = fib_sig;                initialize frame
    if (n<2) {
        free(f, sizeof(*f));         free frame
        return n;
    }
    else {
        int x, y;
        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)

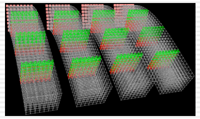
```
struct sum {
   void operator()(Shared_r < int > a,
                   Shared_r < int > b,
                   Shared_w < int > r )
   { r.write(a.read() + b.read()); }
 } ;

 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 ) ;
    }
  }
} ;
```

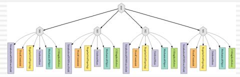


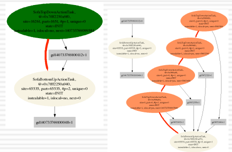
- *Kaapi won the 2006 award "Prix special du Jury" for the best performance at NQueens contest, Plugtests-Grid&Work'06, Nice, Dec.1, 2006 [Gautier-Guelton] on Grid'5000 1458 processors with different speeds.*

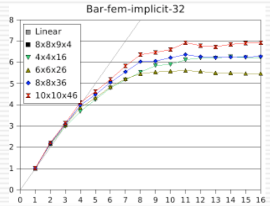
Experimental results on SOFA [CIMIT-ETZH-INRIA]

[Allard 06]

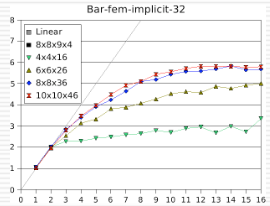








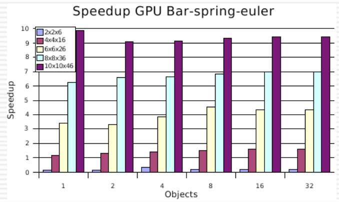
Kaapi (C++, ~500 lines)



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

- **Cascading divide & Conquer**
  - $W(n) = a.W(n/K) + f(n)$
  - $D(n) = D(n/K) + f(n)$
  - $D(n) = D(\sqrt{n}) + \log n$

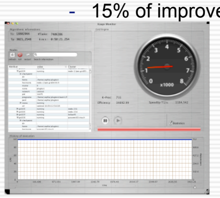
Examples

- **Accumulate:**
  - Sequential
  - Parallel
- **Matrix-vector product – Matrix multiplication**
- **Triangular matrix inversion**
- **Maximum on CRCW**
- **Partial sum**

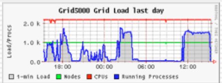
Example: Recursive and Monte Carlo computations

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

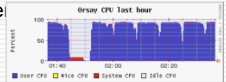
- In 2007, the team won the Nqueens contest; Some facts [on *on Grid'5000*, a grid of processors of heterogeneous speeds]
  - NQueens(21) in 78 s on about 1000 processors
  - Nqueens (22) in 502.9s on 1458 processors
  - Nqueens(23) in 4435s on 1422 processors [~24.10^33 solutions]
  - 0.625% idle time 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



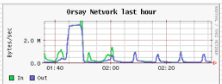
Competitor X
Competitor Y
Competitor Z
Grid'5000 free
N-Queens(23)



CPU

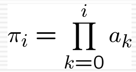
6 instances Nqueens(22)

Network



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



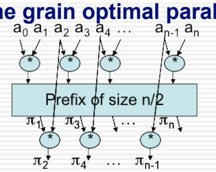
$$\pi_i = \prod_{k=0}^{i} a_k$$

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

performs only **n** *operations*

- **Fine grain optimal parallel** algorithm :

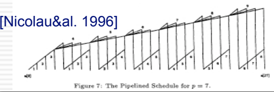
[Ladner-Fisher-81]



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

- 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**

Parallel requires ***twice more*** *operations than sequential !!*