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, matrix 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**

- **Parallel chips & multi-core architectures:**
  - MPSoCs (Multi-Processor Systems-on-Chips)
  - GPU: graphics processors (and programmable: Shaders; Cuda SDK)
  - MultiCore processors (Opterons, Itanium, etc.)
  - Heterogeneous 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:** e.g. Virtual Reality/Visualization Clusters:
  - Scientific Visualization and Computational Steering
  - PC clusters + graphics cards + multiple I/O devices
    (cameras, 3D trackers, multi-projector displays)
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

Which algorithm to choose?

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 \( II_i(t) = \text{speed of processor } i \text{ at time } t \) 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: $\operatorname{Max}_{i,t}\{\Pi_i(t)\} < \text{constant} \cdot \operatorname{Min}_{i,t}\{\Pi_i(t)\}$

Def: for a computation with duration $T$
- total speed: $\Pi_{\text{tot}} = (\sum_{i=0}^{P} \sum_{t=0}^{T} \Pi_i(t)) / T$
- average speed per processor: $\Pi_{\text{ave}} = \frac{\Pi_{\text{tot}}}{P}$

“Work” $W =$ total number operations performed

“Depth” $D =$ operations on a critical path

($\sim$ parallel “time” on \(\propto\) resources)

For any greedy maximum utilization schedule:

$[\text{Graham69, Jaffe80, Bender-Rabin02}]$

makespan $\leq W \cdot \frac{p}{P \cdot \Pi_{\text{ave}}} + \left(1 - \frac{1}{p}\right) \frac{D}{\Pi_{\text{ave}}}$
The work stealing algorithm

- A distributed and randomized algorithm that computes a greedy schedule:
  - Each processor manages a local task (depth-first execution)
  - When idle, a processor steals the topmost task on a remote -non idle- victim processor (randomly chosen)

Theorem: With good probability, \[ \#\text{steals} = O(pD) \] and execution time \[ \leq \frac{W}{p \Pi_{\text{ave}}} + O\left(\frac{D}{\Pi_{\text{ave}}}\right) \]

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: \( H(t) = \min_{(i \text{ active at } t)} \{ H_i(t) \} \) 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 \leq (p-1).\text{Height(tree)} \leq (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 $> \frac{1}{2}$, any active processor at $t$ have been victim of a steal request.
    - Then $\min_i H_i$ has increased of at least 1
  - In average, after $(2p \log p M)$ consecutive steals requests, $\min_i H_i \geq M$
    - Thus, in average, after $(2p \log p D)$ steal requests, the execution is completed!
  - [Chernoff bounds] With high probability (w.h.p.),
    - $\#\text{steal requests} = O(p \log p D)$

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 $\#\text{steal requests} = O(p D)$ 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 \( w_j \) be the number of unit work operations by \( j \)
  - Either idle: performs a steal requests
    - Let \( s_j \) be the number of unit steal operations by \( j \)

- Summing on all \( p \) processors:

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

Work stealing implementation

Difficult in general (coarse grain)
- But easy if \( D \) is small [Work-stealing]

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

Expensive in general (fine grain)
- But small overhead if a small number of tasks

If \( D \) is small, a work stealing algorithm performs a small number of steals

\( \Rightarrow \) 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

```
   1  #include <cilk.h>
   2  int fib (int n)
   3  { if (n < 2) return n;
   4      else
   5      { int x, y;
   6          x = spawn fib (n-1);
   7          y = spawn fib (n-2);
   8          sync;
   9          return (x+y); }
 10  }
```

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

**Experimental results on SOFA**  
[CIMIT-ETZH-INRIA]  

[Allard 06]

**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**

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

- From work-stealing theorem, optimizing the execution time by building a parallel algorithm with both
  - \( W = T_{\text{seq}} \)
  - small depth \( D \)

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

- Accumulate

- => Monte Carlo computations

**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 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^{13} 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](image)

**Grid’5000 free**

**Competitor X**

**Competitor Y**

**Competitor Z**

**N-Queens(23)**

**CPU**

6 instances Nqueens(22)

**Network**
Algorithm design

- Cascading divide & Conquer
  
  - $W(n) \leq a \cdot W(n/K) + f(n)$ with $a>1$
    
    - If $f(n) \ll n^{\log K a}$ $\Rightarrow$ $W(n) = O( n^{\log K a} )$
    
    - If $f(n) \gg n^{\log K a}$ $\Rightarrow$ $W(n) = O( f(n) )$
    
    - If $f(n) = \Theta( n^{\log K a} )$ $\Rightarrow$ $W(n) = O( f(n) \log n )$

  
  - $D(n) = D(n/K) + f(n)$
    
    - If $f(n) = O( \log^i n)$ $\Rightarrow$ $D(n) = O( \log^{i+1} n )$

  
  - $D(n) = D( \sqrt{n} ) + f(n)$
    
    - If $f(n) = O(1)$ $\Rightarrow$ $D(n) = O( \log \log n )$
    
    - If $f(n) = O( \log n)$ $\Rightarrow$ $D(n) = O( \log n )$ !!

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

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

- 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: \( = \log^{O(1)} W \)