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, 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 from small to 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.)
  - 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:** eg 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
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 $\Pi_i(t) = \text{speed of processor } i \text{ at time } t$ 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
Processor speeds are assumed to change arbitrarily and adversarially:

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

\( (\text{in #unit operations per second}) \)

Assumption: \( \max_{i,t} \{ \Pi_i(t) \} < \text{constant} \cdot \min_{i,t} \{ \Pi_i(t) \} \)

Def: for a computation with duration \( T \)

- **total speed:** \( \Pi_{\text{tot}} = \left( \sum_{i=0,\ldots,P} \sum_{t=0,\ldots,T} \Pi_i(t) \right) / T \)
- **average speed per processor:** \( \Pi_{\text{ave}} = \Pi_{\text{tot}} / P \)

“Work” \( W = \text{#total number operations performed} \)

“Depth” \( D = \text{#operations on a critical path} \)

\( (\sim \text{parallel “time” on } \infty \text{ resources}) \)

For any greedy maximum utilization schedule:

\[ \text{makespan} \leq \frac{W}{p \cdot \Pi_{\text{ave}}} + \left( 1 - \frac{1}{p} \right) \frac{D}{\Pi_{\text{ave}}} \]

[Huffman69, Jaffe80, Bender-Rabin02]
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:** $H(t) = \min_{i \text{ active at } t}\{H_i(t)\}$ is increasing.

- **Prop 3:** Each steal request on i makes $H_i(t)$ strictly increase (i.e. $H_i(t+1) \geq H_i(t) + 1$).

- **Prop 4:** For all i and t: $H_i(t) \leq \text{Height(Tree)}$.

- **Corollary:** if at each steal, the victim is a processor i with minimum $H_i(t)$ then
  \[ \#\text{steals} \leq (p-1) \cdot \text{Height(tree)} \leq (p-1) \cdot D \]
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 #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

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

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

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

```c
01 cilk int fib (int n)
02 {
03     if (n < 2) return n;
04     else {
05         int x, y;
06         x = spawn fib (n-1);
07         y = spawn fib (n-2);
08         sync;
09         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, 
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  };
```
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
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: \( = \log^{0(1)} W \)
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 502.9 s on 1458 processors
  - NQueens(23) in 4435 s 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++ (templates)

![Graph showing Grid 5000 utilization during contest](image1.png)

![Graph showing Grid 5000 Grid Load last day](image2.png)

![Graph showing CPU utilization](image3.png)

<table>
<thead>
<tr>
<th>Competitor X</th>
<th>Competitor Y</th>
<th>Competitor Z</th>
<th>Grid 5000 Free</th>
<th>N-Queens(23)</th>
</tr>
</thead>
</table>

![Graph showing Network utilization](image4.png)
Algorithm design

- Cascading divide & Conquer

- $W(n) \leq a \cdot W(n/K) + f(n)$ with $a > 1$
  - If $f(n) << n^{\log_K a}$ $\Rightarrow$ $W(n) = O(n^{\log_K a})$
  - If $f(n) >> n^{\log_K a}$ $\Rightarrow$ $W(n) = O(f(n))$
  - 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 \( \leq \frac{W}{p.\Pi_{ave}} + O\left(\frac{D}{\Pi_{ave}}\right) \)

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

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

Design
and
Implementation

Lecture 2 – Processor oblivious algorithms

Jean-Louis.Roch at imag.fr

MOAIS / Lab. Informatique Grenoble, INRIA, France
Lecture 2

- Remind: Work W and depth D:
  - With work-stealing schedule:
    - #steals = O(pD)
    - Execution time on p procs = W/p + O(D) w.h.p.
    - Similar bound achieved with processors with changing speed or multiprogrammed systems.

- How to parallelize?
  - 1/ There exists a fine-grain parallel algorithm that is optimal in sequential
    - Work-stealing and Communications
  - 2/ Extra work induced by parallel can be amortized
  - 3/ Work and Depth are related
    - Adaptive parallel algorithms
First examples

- Put overhead on the steals:
  - Example Accumulate

- Follow an optimal sequential algorithm:
  - Example: Find_if
Adaptive coupling: Amortizing synchronizations (parallel work extraction)

Example: STL **transform** STL : loop with \( n \) independent computations

\[
f_1 \leftarrow f_2 \quad n_i = l - f_i
\]

\( \alpha \log(n_1) \quad \alpha \log(n_2) \)

Machine:
AMD Opteron Opteron 875
2.2 Ghz,
Compiler gcc, option -O2

![Graph showing time vs. size for transform and transform adapt]
Amortizing Parallel Arithmetic overhead: example: find_if

- For some algorithms:
  - $W_{\text{seq}}$ unknown prior to execution
  - Worst case work $W$ is not precise enough: we may have $W \gg W_{\text{seq}}$

- Example: find_if : returns the index of the first element that verifies a predicate.

Index of the matching element

- Sequential time is $T_{\text{seq}} = 2$
- Parallel time= time of the last processor to complete: here, on 4 processors: $T_4 = 6$
To adapt with provable performances ($W_{par} \sim W_{seq}$): compute in parallel no more work than the work performed by the sequential algorithm (Macro-loop [Danjean, Gillard, Guelton, Roch, Roche, PASCO’07]), Amortized scheme similar to Floyd’s algorithm.

Example: find_if

```
P_0, P_1, P_2   P_0, P_1, P_2   P_0, P_1, P_2
P_0, P_1, P_2
B_3
```
Amortizing Parallel Arithmetic overhead: example: `find_if` [Daouda Traore 2009]

- Example: `find_if` STL
  - Comparison with `find_if` parallel MPTL [Baertschiger 06]

**Machine:**
AMD Opteron (16 cœurs);
**Data:** doubles;
**Array size:** $10^6$;
**Position element:** $10^5$;
**TimeSTL:** 3.60 s;
**Predicate time** $\approx 36\mu$

Speed-down (speed-up $< 1$)
Amortizing Parallel Arithmetic overhead: example: find_if [Daouda Traore 2009]

- **Example : find_if STL**
  - Speed-up w.r.t. STL sequential time and the position of the matching element.

**Machine:**
AMD Opteron (16 cœurs);
**Data:** doubles;
**Size Array:** $10^6$;
**Predicate** time $\approx 36\mu$
Overview

• Introduction: interactive computation, parallelism and processor oblivious
  • Overhead of parallelism: parallel prefix

• Machine model and work-stealing

• Scheme 1: Extended work-stealing: concurrently sequential and parallel
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
How to get both optimal work $W_1$ and $D=W_\infty$ small?

- **General approach:** to mix both
  - a **sequential** algorithm with optimal work $W_1$
  - and a fine grain **parallel** algorithm with minimal depth $D = \text{critical time } W_\infty$

- **Folk technique:** parallel, than sequential
  - Parallel algorithm until a certain « grain »; then use the sequential one
  - Drawback: $W_\infty$ 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
  - $W_\infty$ 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(
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)
**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** (i.e. *workstealing*) [Roch & al 2005,…] 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: concurrently 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 \( k = \text{Depth}(W_{rem}) \) 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**
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 $\Pi_{ave}$, it outputs in a fixed time $T$
a result with the same quality than
a sequential processor with speed $\Pi_{ave}$ 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

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

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

[Octree Carving]

[L. Soares 06]
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: $|d_s - d_p| \leq 1$;
- 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

results: CPU + GPU

8 cameras, levels 2 to 7

log (Time (ms))
Overview

• Introduction: interactive computation, parallelism and processor oblivious
  • Overhead of parallelism: parallel prefix

• Machine model and work-stealing

• Scheme 1: Extended work-stealing: concurrently 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 \( \alpha(W_{\text{done}}) \) 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

N doubles : time predicate $\sim 0.31$ ms

With no amortization macroloop

With amortization macroloop
5. Putting things together

*processor-obliviou suffix computation*

Parallel algorithm based on:

- compute-seq / extract-par scheme
- nano-loop for compute-seq
- macro-loop for extract-par
Parallelism induces overhead: e.g. Parallel prefix on fixed architecture

- **Prefix problem:**
  - input: \( a_0, a_1, ..., a_n \)
  - output: \( \pi_1, ..., \pi_n \) with
  \[
  \pi_i = \prod_{k=0}^{i} a_k
  \]

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

- **Fine grain optimal parallel algorithm:**

- **Tight lower bound on** \( p \) **identical processors:**
  - Optimal time \( T_p = 2n / (p+1) \)
  - but performs \( 2np/(p+1) \) ops

---

**Perform only** \( n \) **operations**

**Parallel requires twice more operations than sequential!!**
Lower bound(s) for the prefix

Prefix circuit of depth $d$

\[ [\text{Fitch80}] \]

#operations $> 2n - d$

\[
\text{parallel time} \geq \frac{2n}{(p+1) \cdot \Pi_{ave}}
\]
P-Oblivious Prefix on 3 proc.

Sequential

\[ \pi_0 \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6 \ a_7 \ a_8 \ a_9 \ a_{10} \ a_{11} \ a_{12} \]

Main Seq. \[ \pi_1 \]

Parallel

\[ \pi_0 \ a_1 \ a_2 \ a_3 \ a_4 \ a_5 \ a_6 \ a_7 \ a_8 \ a_9 \ a_{10} \ a_{11} \ a_{12} \]

Steal from Work-stealer 1

Work-stealer 1

Steal from Work-stealer 2

Work-stealer 2

\[ P_0 \]

\[ P_1 \]

\[ P_3 \]

0 1
time
P-Oblivious Prefix on 3 proc.

Parallel
\[ \alpha_i = a_5^* \cdots a_i \]

Sequential

Main Seq.

Work-stealer 1

Work-stealer 2

Steal request

\[ \pi_0 a_1 a_2 a_3 a_4 \]

\[ \pi_1 \pi_2 \pi_3 \]

\[ a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} \]

\[ \alpha_6 \alpha_7 \]
P-Oblivious Prefix on 3 proc.
P-Oblivious Prefix on 3 proc.

Sequential

Main Seq.

Parallel

Work-stealer 1

Work-stealer 2

α_i = a_5 \cdots a_i

β_i = a_9 \cdots a_i

\pi_0 a_1 a_2 a_3 a_4

\pi_1 \pi_2 \pi_3 \pi_4

\pi_8 \pi_9

\pi_8 \pi_9

\pi_{11}

π_0 a_1 a_2 a_3 a_4

π_1 π_2 π_3 π_4

π_8 π_9

π_{11}

a_5 a_6 a_7 a_8

π_5 π_6 α_7

a_9 a_{10} a_{11} a_{12}

π_9 β_10 β_{11}

P_0

P_1

P_3

0 1 2 3 4 5 6

time
P-Oblivious Prefix on 3 proc.
**P-Oblivious Prefix on 3 proc.**

Implicit critical path on the sequential process

```
π₀ a₁ a₂ a₃ a₄
π₁ π₂ π₃ π₄ π₈
π₁₁ π₁₂
```

Parallel

```
a₅ a₆ a₇
π₅ π₆ π₇
```

Work-stealer 1

```
a₉ a₁₀
π₉ π₁₀
```

Work-stealer 2

```
αᵢ = a₅*…*aᵢ
βᵢ = a₉*…*aᵢ
```

Main Seq.

\[ T_p = 7 \]

\[ T_p^* = 6 \]
Analysis of the algorithm

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

- **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)

  \[
  T_s = \frac{S}{\Pi_{ave}} \quad \text{and} \quad T_p = \frac{X}{(p-1) \cdot \Pi_{ave}} + O \left( \frac{\log X}{\Pi_{ave}} \right)
  \]

- 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 = (\( s+X \)) - \( \# \text{ops\_optimal} \)
Results 1/2

Prefix sum of \(8 \times 10^6\) double on a SMP 8 procs (IA64 1.5GHz/ linux)

Single user context

Single-user context: 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

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

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)

**FlowVR** (flowvr.sf.net)
- Dedicated to interactive applications
- Static Macro-dataflow
- Parallel Code coupling

[Thank you !]

[E Boyer, B Raffin 2006]
Back slides
The Prefix race: sequential/parallel fixed/adaptive

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

Scheduling

Adaptive Algorithms

Execution Control

Interactivity

Coupling
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 STB
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 transfer
    • 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