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

\[ n_i = l - f_1 \]

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

![Graph showing time vs. size](image-url)
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.

  ![Diagram of parallel processors and index](image)

  - Sequential time is $T_{\text{seq}} = 2$
  - Parallel time = time of the last processor to complete: here, on 4 processors: $T_4 = 6$

Amortizing Parallel Arithmetic overhead: example: find_if

- To adapt with provable performances ($W_{\text{par}} \sim W_{\text{seq}}$): 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

  ![Diagram of parallel processors and n cuckoos](image)

- Example: find_if
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:** $10^5$;
**Time STL:** 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 tim 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**

$\Rightarrow$ parallel algorithm is used only if a processor becomes idle (i.e., workstealing) [Roch\&al2005,…] to extract parallelism from the remaining work a sequential computation

Assumption: two concurrent algorithms that are complementary:
- one sequential: $\text{SeqCompute}$ (always performed, the priority)
- the other parallel, fine grain: $\text{LastPartComputation}$ (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&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)

---

**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)

Note:
- **merge and jump** operations to ensure non-idleness of the victim
- Once **SeqCompute_main** completes, it becomes a work-stealer
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 »)
    
    $\Rightarrow$ $D$ 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)
    
    $\Rightarrow$ $W$ 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:

State of a cube:
- Grey: mixed => split
- Black: full : stop
- White: empty : stop

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

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)

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: 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 \( \alpha(W_{done}) \) 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

---

**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**:
  - Critical time = \( 2 \cdot \log n \)
  - but performs \( 2 \cdot n \) ops

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

---

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

Prefix circuit of depth \( d \)

\[ \downarrow \quad [\text{Fitch80}] \]

#operations > \( 2n - d \)

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.

Steal request

Parallel

Work-stealer 1

\( \Pi_1 \)

Work-stealer 2

\( P_0 \ P_1 \ P_2 \ P_3 \)

time
P-Oblivious Prefix on 3 proc.

Sequential

\[ \pi_0 \ a_1 \ a_2 \ a_3 \ a_4 \]

Main Seq.

Parallel

\[ \alpha_i = a_5^* \ldots ^* a_i \]

Work-stealer 1

Steal request

Work-stealer 2

P-Oblivious Prefix on 3 proc.

Sequential

\[ \pi_0 \ a_1 \ a_2 \ a_3 \ a_4 \]

Preempt \( \pi_4 \)

Parallel

\[ \alpha_i = a_5^* \ldots ^* a_i \]

\[ \beta_i = a_9^* \ldots ^* a_i \]
P-Oblivious Prefix on 3 proc.

Sequential

Parallel

\( \pi_0 a_1 a_2 a_3 a_4 \rightarrow \pi_1 \pi_2 \pi_3 \pi_4 \rightarrow \pi_8 \rightarrow \pi_{11} \)

Preempt \( \pi_3 \rightarrow \beta_{11} \)

Work-stealer 1

Work-stealer 2

\( \alpha = a_5 \ldots a_i \)

\( \beta = a_9 \ldots a_i \)

P-Oblivious Prefix on 3 proc.

Sequential

Parallel

\( \pi_0 a_1 a_2 a_3 a_4 \rightarrow \pi_1 \pi_2 \pi_3 \pi_4 \rightarrow \pi_8 \rightarrow \pi_{11} \rightarrow \pi_{12} \)

Preempt \( \pi_3 \rightarrow \beta_{11} \)

Work-stealer 1

Work-stealer 2

\( \alpha = a_5 \ldots a_i \)

\( \beta = a_9 \ldots a_i \)
P-Oblivious Prefix on 3 proc.

Implicit critical path on the sequential process

Parallel

Sequential

Analysis of the algorithm

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

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

  $\Rightarrow$ enables to bound the whole number $X$ of operations performed
  and the overhead of parallelism = (s+X) - 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:

- Processor-oblivious prefix computation 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 $8 \times 10^6$ double on a SMP 8 procs (IA64 1.5GHz/ linux)

Multi-user context:

- 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 deferenates 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!
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 \( * = \text{double sum} \) \( (r[i]=r[i-1] + x[i]) \)

Finest “grain” limited to 1 page = 16384 octets = 2048 double

Remark for \( n=4,096,000 \) doubles:
- “pure” sequential: 0.20 s
- minimal “grain” = 100 doubles: 0.26 s on 1 proc
  and 0.175 on 2 procs (close to lower bound)
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