Processor-oblivious parallel algorithms with provable performances

Applications

Jean-Louis Roch

Lab. Informatique Grenoble, INRIA, France

Interactive parallel computation?

Any application is "parallel": composition of several programs / library procedures (possibly concurrent);
each procedure written independently and also possibly parallel itself.

Parallelism induces overhead:
e.g. Parallel prefix on fixed architecture

- Prefix problem:
  input: a_0, a_1, ..., a_n
  output: s_0, s_1, ..., s_n
- Sequential algorithm:
  \( s_j = \bigoplus_{i=0}^{j} a_i \)
- Fine grain optimal parallel algorithm:
  \( s_j = \bigoplus_{i=0}^{j} a_i \) performs only \( 2n \) operations
- Tight lower bound on \( p \) identical processors:
  \( T_p = 2n(1 + \frac{1}{p}) \) performs \( 2n \) \( p(1 + \frac{1}{p}) \) operations

Overview

- Introduction: interactive computation, parallelism and processor oblivious
- Overhead of parallelism: parallel prefix
- Machine model and work-stealing
  - Scheme 1: Adaptive parallel algorithms
  - Scheme 2: Amortizing the overhead of synchronization (Nano-loop)
  - Scheme 3: Amortizing the overhead of parallelism (Macro-loop)
- Putting things together: processor-oblivious prefix computation

New parallel supports

from small to large

- Parallel chips & multi-core architectures:
  - MPSoCs (Multi-Processor Systems-on-Chips)
  - GPU: graphics processors (parallelizable: shaders, Compute API)
  - Dual-Core processors (Opteron, Itanium, etc.)
  - Heterogeneous multi-cores: CPU+GPU+DSPs+FPGAs (Cell)
- Commodity SMPs:
  - 4 way PCs equipped with multi-core processors (AMD Hypertransport) + 0 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 Scalability
- Dedicated platforms: e.g. Virtual Reality/Visualization Clusters:
  - Scientific Visualization and Computational Steering
  - PC clusters + graphics cards + multiple X3 servers
  - (perform, 3D trackers, multi-projector displays)

Lower bound(s) for the prefix

Prefix circuit of depth \( d \)

\[ \text{operations} > 2n - d \]

Parallel time \( \geq \frac{2n}{(p+1)\tau_{\text{min}}} \)
The problem
To design a single algorithm that computes efficiently prefix(a) on an arbitrary dynamic architecture

Processor-oblivious algorithms
Dynamic architecture: non-fixed number of resources, variable speeds eg: grid, SMP server in multi-users mode. . .

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: N-Queens on an heterogeneous grid

Heterogeneous processors, work and depth
Processor speeds are assumed to change arbitrarily and adversarially:
model (precise): \( R_p(t) = \text{instantaneous speed of processor } i \text{ at time } t \) (in task operations per second)

Assumption: \( \beta_{min} R_p(t) < \text{const} \cdot \beta_{max} R_p(t) \)

Def. for a computation with duration \( T \)
- total speed: \( \sum_{i=1}^{p} \beta_i R_i \)
- average speed per processor: \( \frac{\sum_{i=1}^{p} \beta_i R_i}{p} \)

"Work" \( W = \# \text{total operations performed} \)
"Depth" \( D = \# \text{operations on a critical path} \)
(parallel"time" on = resource)

For any greedy maximum utilization schedule:

\[
\text{makespan} = \frac{W}{p} \cdot \sum_{i=1}^{p} \frac{1}{R_i} = \frac{W}{\beta_{min}} \cdot \sum_{i=1}^{p} \frac{1}{R_i}
\]

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 job on a remote, non-idle, victim processor (randomly chosen)
- Theorem: With good probability,
  - Attack \( < p \cdot D \)
  - Execution time \( \frac{W}{p} \cdot \sum_{i=1}^{p} \frac{1}{R_i} \)

- Interest: if \( W \) independent of \( p \) and \( D \) is small, work stealing achieves near-optimal schedule
Work stealing implementation

- Difficult in general (coarse grain)
- But easy if D is small
- But small overhead if a small number of tasks

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

**Implementation:** since all tasks but a few are executed in the local slack, overhead of task creation should be as close as possible as sequential function call

At any time on any non-tille processor, efficient local degeneration of the parallel program in a sequential execution

---

Work-stealing implementations following the work-first principle: Cilk

- Cilk-5 is an efficient sequential algorithm:
  - Parallel Divide & Conquer computations
  - ... But, this may not be general in practice

---

Work-stealing implementations following the work-first principle: KAAPI

- KAAPI / Athapascan
- ... But, this may not be general in practice

---

N-queens: Takaken C sequential code parallelized in C++/KAAPI


---

Experimental results on SOFA

- Kaapi (C++, ~300 lines)
- Cilk (C, ~240 lines)

---

3. Work-first principle and adaptability

- Work-first principle: explicit - 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 in Work
- WS adapts parallelism to processors with practical provable performances
  - Parallel, Divisible & Computations

The choice is justified only when the sequential execution of the parallel algorithm is an efficient sequential algorithm:

- Parallel, Divisible & Computations

-> But, this may not be general in practice
How to get both optimal work \( W_1 \) and \( W_c \) small?

- General approach: to mix both
  - a sequential algorithm with optimal work \( W_1 \)
  - and a fine grain parallel algorithm with minimal critical time \( W_c \)

- Fork technique: parallel, than sequential
  - Parallel algorithm until a certain grain - then use the sequential one
  - Drawback: \( W_c \) increases \( \Delta \), and also, the number of steals

- Work-preemining speed-up technique (parallel, then sequential, then parallel
  Cascading: work)
  - Careful interplay of both algorithms to build one with both \( W_1 \) small and \( W_c \leq W_{opt} \)
  - Use 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, \( \Delta \)

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 used only if a processor becomes idle (as work-stealing)
- \( W_{opt} \) from work-stealing

Assumption: two concurrent algorithms that are complementary:
- One sequential, \( \text{SeqCompute} \) (always performed, the priority)
- The other parallel, fine grain / \( \text{FastPortComposition} \) (often not performed)

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 \( \Pi_{av} \), it outputs in a fixed time \( T \)
with the same quality than a sequential processor with speed \( \Pi_{seq} \) in time \( p \Pi_{seq} \)

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**  
[L. Soares 06]  
A classical recursive anytime 3D modeling algorithm.

- Standard algorithms with time control:
  - (a)
  - (b)
  - (c)
  - (d)
  - (e)
  - (f)
  - (g)

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 \((\text{e.g. } D = 8, W = 164,000)\)
- 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 overhead)  
- Time deadline \((30 \text{ ms})\) managed by signal protocol

**Results**  
[L. Soares 06]  
- 16 core Opteron machine, 64 images  
- Sequential: 293 ms, 16 Cores: 24 ms  
- 8 cores: about 100 steals \((187,000 \text{ gray cells})\)

**Preliminary result: CPUs+GPU**
- 1 GPU = 16 CPUs  
- GPU programmed in OpenGL  
- Efficient coupling till 8 but does not scale

---

**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

\(\text{Eg (C++ STL): } \text{find}_\text{if} \text{(first, last, predicate)}\)

Locates the first element in \((\text{First, Last})\) 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 increases 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 remaining 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.

\(\text{Eg: } \text{find}_\text{if} \text{(first, last, predicate)}\)
- Only the work already performed is known \((\text{on-line})\)  
- Then prevent to assign more than \(e(W_{\text{max}})\) operations to work-stealers
- Choices for \(g(n)\):
  - \(n^2\): similar to Floyd’s iteration \((\text{approximation ratio } = 2)\)
  - \(n/\log^* n\): to ensure optimal usage of the work-stealers

---

**Results on find_if**  
[L. Gauthier]

\(N\text{ doubles: time predicate } = 0.31 \text{ ms}\)

With no amortization macroloop

With amortization macroloop
5. Putting things together
process oblivious prefix computation

Parallel algorithm based on:
- compute-seq / extract-par scheme
- nano-loop for compute-seq
- macro-loop for extract-par
Parallel Code coupling

P-Oblivious Prefix on 3 proc.

Parallel
Sequential
Implicit critical path on the sequential process

Project 4

Analysis of the algorithm

• Execution time ≤ \( \frac{2n}{(p+1)P_{ave}} + O \left( \frac{\log n}{P_{ave}} \right) \)

• 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 performs \( X \) operations on other processors
    - Dynamic splitting always possible (for finest grain GBT local sequential)
    - Dynamic splitting possible if not \( S \) local sequential
  - \( T_p = \frac{S}{X} \) (constant time that can possibly split on optimal path)
  - \( T_\text{opt} = \frac{S}{X} + O \left( \frac{\log n}{P_{ave}} \right) \)

Results 1/2

[D Traore]

Prefix sum of 8.10^6 double on a SMP 8 proc (IA64 1.5GHz/linu)

Single-user context:
- Processor-oblivious prefix achieves user-optimal performance:
  - Close to the lower bounds on 1 proc and 8 p processors.
- Less sensitive to system overhead - even better than the theoretically "optimal" off-line parallel algorithm on p processors.

Multi-user context:
- Processor-oblivious prefix on 8 x-8 processors:
  - Dynamic coupling of two algorithms that complete simultaneously:
  - Dynamic splitting always possible (for finest grain GBT local sequential)

Results 2/2

[D Traore]

Prefix sum of 8.10^6 double on a SMP 8 proc (IA64 1.5GHz/linu)

Multi-user context:
- Additional external charge (9-p) additional external dummy processes are concurrently executed

Conclusion

• Fine grain parallelism enables efficient execution on a small number of processors:
  - Interest: portability; modularization of code;
  - Drawback: needs work-first principle \( \Rightarrow \) algorithm design

• Efficiency of classical work stealing relies on work-first principle:
  - Implicitly deforms a parallel algorithm into a sequential efficient ones:
  - Assumes that parallel and sequential algorithms perform the same amount of operations

• Processor Obvious algorithms based on work-first principle
  - Based on enroute extraction of parallelism from any sequential algorithm (may execute different amount of operations)
  - Obvious: near-optimal whatever the execution context is.

• Generic scheme for stream computations:
  - Parallelism introduces a copy overhead from local buffers to the output

Kaapi (kaapi.gforge.inria.fr)
- [FlowVR (flowvr.snet)]
  - Dedicated to interactive applications
  - Static Macro-dataflow
  - Parallel Code coupling

[2 Raffin, MAIS & E Boyer, PERCEPTION]

Thank you!
Back slides

The Prefix race: sequential/parallel fixed/ adaptive

Adaptive prefix: some experiments

With * = double sum ( \( r[i][k] = r[i][k] + x[i] \) )

The Moais Group

Moais Platforms

- [1] Solaris 2:
  - 110 dual Itanium in-processors with Myrinet network
- Grifimage ("Griffe" and Images):
  - Camera Network
  - 54 processors (dual processor cluster)
  - Dual gigabit network
  - 18 projectors display wall
- Grids:
  - Regional Center
  - National Grid5000
  - Cedelectro CS experiments
- SMPs:
  - 8-way Intel (Bull nanoscale)
  - 8-way dual-core Opteron + 2 GPUs
- MPSoCs:
  - Collaborations with ST Microelectronics on STi
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