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, 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.

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 (Optronis, 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

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 \( \Pi_i(t) = \text{speed of processor } i \text{ at time } t \) 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 \( \Pi_i(t) = \text{instantaneous speed of processor } i \text{ at time } t \)

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

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

“Work” \( W = \#\text{total number operations performed} \)
“Depth” \( D = \#\text{operations on a critical path} \)
(“parallel “time” on = resources)

For any greedy maximum utilization schedule:
\[ \text{makespan} = \frac{W}{P \Pi_{ave}} \cdot \left( 1 - \frac{1}{p} \right) \frac{D}{\Pi_{ave}} \]
The work stealing algorithm

- A distributed and randomized algorithm that computes a greedy schedule:
  - Each processor manages a local task (depth-first execution)

![Diagram of processors and tasks]

Theorem: With good probability, 

\[ \text{steals} < \frac{W}{p \Pi_{\text{ave}}} + \left( \frac{D}{\Pi_{\text{ave}}} \right) \]

Interest:

- If \( W \) independent of \( p \) and \( D \) is small, work stealing achieves near-optimal schedule.

Work stealing implementation

Efficient policy

Scheduling

Control of the policy

Difficult in general (coarse grain)

But easy if \( D \) is small (work-stealing)

 narrow overhead in a small number of tasks

Expensive in general (fine grain)

Execution time

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

(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 \( f(x); \) sync (mere-parallel programs)
  - Requires a shared-memory machine
  - Depth-first execution with synchronization (on sync) with the end of a task:
    - Spawmed 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)

01 {
02     if (n < 2) return n;
03     else {
04         int x, y;
05         x = spawn fib(n-1);
06         y = spawn fib(n-2);
07         sync;
08         return (x+y);
09     }
10 }

Cilk

- won the 2006 award “Best Combination of Elegance and Performance” at HPC Challenge Class 2, SC06, Tampa, Nov 14 2006 [Announced on SGI Altix 3700 with 128 bi-Itanium]
Work-stealing implementations following the work-first principle: KAAPI

- Kaapi / Athapascan [http://kaapi.gforge.inria.fr] C++ library
  - Fork<>()(a, ...) with access mode to parameters (value;read;write;rw;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)


Experimental results on SOFA [CIMIT-ETZH-INRIA]

Kaapi (C++, ~500 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
  - \( W(n) = a \cdot W(n/K) + f(n) \)
  - \( D(n) = D(n/K) + f(n) \)
  - \( D(n) = D(\sqrt{\text{sqrt}(n)}) + \log n \)

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 502.9 s on 1458 processors
  - Nqueens(23) in 4435 s on 1422 processors [-24.10^3 solutions]
  - 0.625% idle time per processor
  - < 20 s to deploy up to 1000 processes on 1000 machines (Taktuk, Huard)
  - 15% of improvement of the sequential due to C++

Parallelism induces overhead:

- **Prefix problem:**
  - Input: a_n, a_{n-1}, ..., a_0
  - Output: \( \pi_n, \pi_{n-1}, ..., \pi_0 \)

- **Sequential algorithm:**
  - for \( i = 0 \) to \( n \)
  - \( \pi_i = a_i \)

- **Fine grain optimal parallel algorithm:**
  - Critical time = \( 2 \cdot \log n \)
  - Optimal time: \( \pi_i = a_i \cdot \prod_{k=0}^{i-1} a_k \)
  - \( \pi_i = a_i \cdot \prod_{k=0}^{i-1} a_k \)

- **Tight lower bound on \( n \) identical processors:**
  - \( \pi_i = a_i \cdot \prod_{k=0}^{i-1} a_k \)
  - \( \pi_i = a_i \cdot \prod_{k=0}^{i-1} a_k \)

- **Performs only \( n \) operations**

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