Cache-Efficient Parallel Algorithms for Scientific Visualization

Marc Tchiboukdjian  Vincent Danjean
Jean-Philippe Nominé  Bruno Raffin
Leverage the Power of Modern Processors

Modern Processors

- Increasing number of cores
- Caches hierarchy
- Complex architecture with many parameters

Challenge: efficient use of hardware

- Take advantage of caches (memory access locality)
- Take advantage of multiple cores (parallel programs)
- Performance guarantee
Impact on Algorithm Design

New machine models

- Need models taking into account characteristics of hardware (e.g. number of cores, caches)
- Design and analyze algorithms within these models

The "Oblivious" Approach

*Design algorithms without using parameters from the hardware as fast as aware algorithms with full knowledge of the hardware.*

Advantages of oblivious algorithms

- Portable
- Can adapt dynamically to execution environment
- No need for parameter tuning
Outline

Introduction

Processor-Oblivious Algorithms

Cache-Oblivious Algorithms

Scientific Visualization

Cache-Efficient Parallel Visualization

Conclusion
Processor-Oblivious Algorithms

Processor-Oblivious

Construct parallel algorithms/programs without using the number of processors p or their speeds.

Advantages

- Composition of parallel computations
- Processors with varying speeds

Method

1. Programmer expresses parallelism at fine grain
2. Runtime maps computations on hardware
Work and Depth Model

Directed Acyclic Graph

- **Nodes**: Tasks
  - basic units of computation
- **Arcs**: Precedences $u \rightarrow v$
  - $u$ must be executed before starting $v$

Work and Depth

- **Work $W$**: total number of operations of all tasks
- **Depth $D$**: number of operations of tasks on the critical path
- **Execution time on $p$ processors**: $T_p \geq \frac{W}{p}$ and $T_p \geq D$
Work Stealing Scheduler

**Worker algorithm**
- Each thread has its own work queue
- Pop a task from the queue and execute it
- Push newly created tasks in the queue
- Push/pop at the bottom of the queue

**Thief algorithm**
- When queue is empty, select a victim at random
- Steal the task at the top of the queue

**Concurrent queue**
- Thieves and worker operate on different sides of the queue
- No synchronization operation for the worker
Performance Guarantee on the Execution Time

- Using a work stealing scheduler with random victim selection on a DAG of work $W$ and depth $D$, the expected time on $p$ processors is
  \[ T_p = \frac{W}{p} + O(D) \]

- The expected number of steals is $S = O(p \cdot D)$

- Design parallel algorithms with small depth $D$

*Gantt chart: work in white, steal in grey.*
Example: Parallel Loops

Examples of parallel loops

- OpenMP `#pragma omp parallel for`
- TBB `parallel_for`
- Cilk `parallel_for`
- Parallel STL `for_each` or `transform`

Problem characteristics

- Schedule $n$ iterations on $p$ cores
- Iterations can be processed independently
- Time to process one iteration can vary
Static Scheduling of Parallel Loops

Static Scheduling

- Allocate \( \frac{n}{p} \) iterations to each core
- ex: OpenMP static scheduling

Characteristics

- low overhead mechanism
- bad load balancing if workload is irregular
Dynamic Scheduling of Parallel Loops

OpenMP dynamic scheduling

- Allocate iterations in chunks of size $g$
- All chunks are stored in a centralized list
- Each thread remove a chunk from the list and process it

Characteristics

- **Good load balancing**
- **Contention** on the list
- **Chunk creation overhead**
Scheduling Parallel Loops with Work Stealing

Work Stealing

- Each thread has its own list of tasks (\(\equiv\) chunks)
- If list is empty, steal tasks in a randomly selected list
- Binary tree of tasks to minimize number of steals:
  one steal \(\Leftrightarrow\) half of the iterations

Characteristics

- Good load balancing
- Contention is reduced
- **Task creation overhead**
Scheduling Parallel Loops with Work Stealing

- terminated tasks
- ready tasks
- running tasks
Scheduling Parallel Loops with Work Stealing

- terminated tasks
- ready tasks
- running tasks

steal
Parallel Loops with **Xkaapi**

**Xkaapi**

- Work stealing library
- **Tasks are created on a steal:** reduce task creation overhead
- Aggregation of steal requests: reduce contention on the work queues
- Better load balancing

**Characteristics**

- Good load balancing
- Low overhead mechanism

```c
typedef struct {
    InputIterator ibeg;
    InputIterator iend;
} Work_t; // Task

void parallel_for (...) {
    while (iend != ibeg)
        do_work ( ibeg++ );
} // no more work -> become a thief

void splitter ( num_req ) {
    i = 0;
    size = victim.iend - victim.ibeg;
    bloc = size / ( num_req + 1 );
    local_end = victim.iend;
    while ( num_req > 0 ) {
        thief->iend = local_end;
        thief->ibeg = local_end - bloc;
        local_end -= bloc;
        --num_req;
    }
} // victim + thieves -> parallel_for
```
Outline

Introduction

Processor-Oblivious Algorithms

Cache-Oblivious Algorithms

Scientific Visualization

Cache-Efficient Parallel Visualization

Conclusion
Characteristics

- **Automatically** managed by the CPU
- Transfer by **blocks** or cache lines (generally 64B)
- When a data is not in cache: **cache miss**
- Pseudo-LRU replacement:
  
  evict the least recently used cache line
The Cache-Oblivious Model

- Performance: number of block transfers (cache misses)
- Model locality of memory accesses
- Architecture independent

Marc Tchiboukdjian
Cache-Efficient Parallel Algorithms for Scientific Visualization
Cache-Oblivious Algorithms

Cache-Oblivious

*Construct algorithms/programs without using the cache size, the line size and the number of cache levels.*

Advantages

- Portable
- Optimal on 2 levels $\rightarrow$ optimal on all levels

Methods

- Divide and Conquer
- Recursive data layout
Example: Matrix Multiplication

Naive algorithm

- Memory accesses in $B$ are suboptimal
- $Q(N) = O\left(\frac{N}{B} + N\right) \cdot N^2$
- $Q(N) = O(N^3)$
Example: Matrix Multiplication

Algorithm by blocks

- 3 blocks fit in cache
- \( Q(N) = \mathcal{O}\left( \frac{M}{B} \right) \cdot \frac{N}{\sqrt{M/3}} \cdot \frac{N^2}{M/3} \)
- \( Q(N) = \mathcal{O}\left( \frac{N^3}{B\sqrt{M}} \right) \)
- Cache-aware
Example: Matrix Multiplication

D&C algorithm

- Recursively divide in 4
- Eventually, 3 blocks fit in cache
- Cache-Oblivious
- Same number of cache misses
Outline

Introduction

Processor-Oblivious Algorithms

Cache-Oblivious Algorithms

Scientific Visualization

Cache-Efficient Parallel Visualization

Conclusion
Scientific Visualization

- Precise simulations $\rightarrow$ huge data sets
- Bottleneck for visualization filters: data transfers
Mesh Data Structure

- Main data structure for visualization filters
- Points array contains coordinates and attributes of points
- Cells array contains indices of points for each cell
Common Mesh Access Patterns of Visualization Filters

Visualization Filter
A function $f$ to apply to all or a subset of points or cells of the mesh

Cell Neighborhood

Point Neighborhood

Cell Attributes

Point Attributes

Store elements connected in the mesh close in memory
Cache-Oblivious Mesh Layout: FastCOL

FastCOL Algorithm

- Recursively cut the mesh while minimizing the cut
- Store contiguously elements in the same node of the BSP tree
- Layout computation: $O(n \log n)$
Cache-Oblivious Mesh Layout: FastCOL

The FastCOL layout guarantees a cache-efficient traversal for spatially coherent filters whatever the cache size.

\[ Q_{B,M}(S) = \frac{S}{B} + O\left(\frac{S}{M^{1/3}}\right) \]

- \( Q_{B,M}(S) \): mesh read overhead
- \( B \): cache line size
- \( M \): cache size
- \( S \): mesh size

Example of layouts

(Cells close in memory have the same color)
Cache-Oblivious Mesh Layout: experimental results

- Speedup $\frac{t_{ori}}{t_{opt}}$
- Cache misses ratio $\frac{c_{ori}}{c_{opt}}$

- Non modified VTK filters except Iso (home-made)
- Speedup of the CO mesh compared to the original mesh
Outline

Introduction

Processor-Oblivious Algorithms

Cache-Oblivious Algorithms

Scientific Visualization

Cache-Efficient Parallel Visualization

Conclusion
Keep Good Cache Performance in Parallel

8MB

256KB  256KB  256KB  256KB
32KB   32KB   32KB   32KB
Core 0 Core 1 Core 2 Core 3

Nehalem (Xeon E5530)
Keep Good Cache Performance in Parallel

Nehalem (Xeon E5530)

- Take advantage of multiple cores
- No extra cache misses
- Cores share the last cache level
Classical Scheme: NoWindow

NoWindow Strategy

- Divide $n$ tets into $p$ chunks
- Cores compete for shared cache space
Shared Cache Aware Scheme: StaticWindow

StaticWindow Strategy
▶ Divide into chunks fitting in the shared cache
▶ Cores work in parallel inside a chunk
▶ Cores benefit from data cached by others

StaticWindow vs NoWindow
▶ Less cache misses
▶ More synchronizations
Synchronization overhead

Speedup on 4 cores on Opteron

- Pthread < TBB < Xkaapi

On Opteron: no shared cache ⇒ Window < NoWindow
more synchronizations without gain in cache misses
Window Size $m$

- Static
- Seq.
- No

- On 4 cores of Nehalem
- Shared 8MB $L_3$ cache

Graph showing the number of $L_3$ cache misses for different window sizes $m$.
Speedup on Nehalem

- 4 cores of Nehalem with 8MB of shared cache $L_3$
- Best performance: StaticWindow
Optimized Implementation using Xkaapi: SlidingWindow

- Processing iteration $i$ enables iteration $i + m$
- Master thread is at the beginning of the sequence
- On a steal, the master can give work
  - In the interval $[ibeg, iend[$ like the other workers
  - In the interval $[ilast, ibeg + m[$ enabled since the last steal

```
typedef struct {
    InputIterator ibeg;
    InputIterator iend;
} Work_t ; // Task
```

```
typedef struct {
    InputIterator ibeg;
    InputIterator iend;
    InputIterator ilast;
} Master_Work_t ; // Master Task
```
Speedup and Cache Misses on Nehalem

- 4 cores of Nehalem with 8MB of shared cache \( L_3 \)
- Best performance: SlidingWindow
Conclusion

Design Oblivious Algorithms

► Processor-Oblivious Algorithms
  ▶ Work and depth model
  ▶ Work stealing scheduler
  ▶ Optimized implementation for parallel loops

► Cache-Oblivious Algorithms
  ▶ Cache-Oblivious model
  ▶ Divide and Conquer technique
  ▶ Cache-Oblivious Mesh Layout

Combine both approaches

► Shared cache aware scheduler for parallel loops
► Efficient implementation using work stealing
► Need cache size to find window size $m$
  Cache-oblivious version?
Some References

- Have a look at my webpage
  http://moais.imag.fr/membres/marc.tchiboukdjian/
- M. Frigo, C. E. Leiserson, H. Prokop and S. Ramachandran
  *Cache-Oblivious Algorithms*
- R. D. Blumofe and C. E. Leiserson
  *Scheduling multithreaded computations by work stealing*
- N. S. Arora, R. D. Blumofe and C. G. Plaxton
  *Thread scheduling for multiprogrammed multiprocessors*
- M. Frigo, C. E. Leiserson and K. H. Randall
  *The implementation of the Cilk-5 multithreaded language*
  *Deque-free work-optimal parallel STL algorithms*
- X. Besseron, C. Laferriere, D. Traore and T. Gautier
  *XKaapi : Une nouvelle implémentation eXtrême du vol de travail*