CACHE-OBLIVIOUS ALGORITHMS

Lecture based on [Kumar03]

Slides partially based on
Cache Oblivious Algorithms – Theory and Practice
by Piyush Kumar
(with slides by Harold Prokop)
This lecture

• Motivation
• The cache oblivious model
• Cache oblivious searching
• Cache oblivious sorting
• Discussion of the model
Brief History

Cache-obliviousness is a **hot** new idea for algorithms dealing with massive data sets.

- Frigo, Leiserson, Prokop, Ramachandran (FOCS 99): **Cache oblivious algorithms**
- Bender, Demaine, Farch-Colton (FOCS 00) **Cache oblivious B-trees**
- ...
- Arge et al. (STOC 02) **Cache oblivious priority queue**
Memory of a modern computer

A hierarchy of memory levels. Lower levels are increasingly bigger, but also increasingly slower.
Two consecutive levels of the hierarchy
Block transfers between levels

- Until now we have considered only two levels of memory: RAM and disk.
- However, I/Os may not always be the dominant cost (see [Sanders03]).
- We would like to consider all levels of the memory hierarchy.
- There is a lot of interest in this among algorithms and database researchers.
PROBLEM SESSION

- Consider a memory hierarchy with three levels (e.g., processor cache, RAM and disk).
- Suppose that block transfers between the lower levels are on $B_1$ words, and on $B_2$ words between the upper levels.
- How would one design a “B-tree” that takes both $B_1$ and $B_2$ into account?
Caching in the memory hierarchy

• A level in the memory hierarchy serves as a cache for the next level.

• **Example**: Main memory is used to cache a number of disk blocks (pages).

• The caching may be controlled by hardware or in software.

• Common caching strategies: LRU, FIFO.
Limitations on caches

- Hardware caches often use restricted versions of LRU or FIFO.
- In particular, limited **associativity** is common: The number of possible locations in cache for a given block is some (small) number $k$. 
Ideal cache model

Features:
- Two-level hierarchy.
- Cache of size $M$.
- Cache-line length $B$.
- Fully associative.
- Optimal, omniscient replacement.

Measures:
- Cache misses $Q$.
- Work $W$. 
Assumptions of the model

- Two Levels of Memory
- Tall Cache Assumption: $M = \Omega(B^2)$
- Optimal Cache Replacement (!)
  Not as unreasonable as it seems:
  - Fully-associative LRU can be used instead of optimal replacement with no asymptotic loss of performance.
  - Fully-associative LRU caches can be maintained in ordinary memory with constant slowdown in expected performance.
  - See [Kumar03, Lemma 9.1] for details.
Cache obliviousness

- **Cache oblivious algorithm** = Algorithm in the ideal cache model which does **not** “know” M and B.
- In the **analysis** of a cache oblivious algorithm, M and B may be used.
- Ideally, we obtain the same complexity as can be obtained in the external memory model (where parameters M and B are known to the algorithm).
PROBLEM SESSION

• Argue that an optimal cache-oblivious algorithm (in the ideal cache model) is automatically optimal for all levels in a memory hierarchy (with ideal caches).

• Which of the algorithms that we have seen are cache oblivious?
Cache oblivious B-trees

- B-trees were derived from binary search trees by **blocking** nodes in groups of size roughly $B$.
- Can we obtain the same effect without using any knowledge of $B$?
- We now consider the **static** case, where there are no updates, only searches.
Static searches

- Suppose we have a **perfectly balanced** binary tree.
- Assume there are no insertions and deletions.
- How do we search with few cache misses?

Can we speed it up?
Choosing the memory layout

- **Layout** = Mapping of nodes of a tree to memory cells
- Different kinds of layouts
  - In-order
  - Post-order
  - Pre-order
  - Van Emde Boas
- **Main Idea** : Store recursive subtrees in contiguous memory
Van Emde Boas layout example

Actual Layout of Tree in memory:

1, 2, 3, 4, 8, 9, 5, 10, 11, 6, 12, 13, 7, 14, 15
Another view
Search complexity

Analysis:

- Recursive subtrees of size at most $B$ span at most two contiguous blocks.
- Thus, at most two cache misses for each recursive subtree of height $\log B$.
- Total number of cache misses when searching down all $\log n$ levels is therefore:

$$2 \log n / \log B = 2 \log_B n.$$
Performance in practice

- Windows notebook/512MB/PIII 1Gz/256 byte nodes
Performance in practice II

- Windows notebook/512MB/PIII 1Gz/32 byte nodes
Performance in practice III

- Linux/Itanium/2GB/g++ -O3/ 48 byte nodes
Cache oblivious sorting

In contrast to internal sorting, only a few optimal methods known:

- Funnel sort (Modified merge sort)
- Column sort
- Distribution sort (next ⑥)

(Modified sample sort - we discuss a randomized version)
Distribution sort

Basic approach:

- Partition A into \( \sqrt{n} \) sub-arrays each of size \( \sqrt{n} \); Sort recursively
- Distribute into \( \sqrt{n} \) ordered buckets
- Sort buckets recursively
- Copy buckets to output
Recursive sorting of subarrays

$n$ input elements, partitioned into $\sqrt{n}$ contiguous subarrays of size $\sqrt{n}$:

Recursive sorted arrays:

Order:
**Distribution step**

**Recursively sorted arrays:**

- Buckets:
  - Pivots:

**Distribute step**

**Order:**
The distribution step

- Has to distribute subarrays into buckets $B_1, B_2, B_3...B_q$
- Each bucket contains elements from a range delimited by two **pivot** elements.
- The set of pivot elements can be chosen by taking a **random sample** of $O(\sqrt{n})$ elements. (This works well with high probability)
Recursive bucketing

**Idea:** Solve 4 bucketing subproblems recursively.
PROBLEM SESSION

How many cache misses does the distribution algorithm just presented incur?
Recursive sorting of buckets

After distribution step:

Recursively sort each bucket.
The algorithm

Algorithm 9.7.1 procedure Sort$(A)$

Require: An input array $A$ of Size $N$

1: Partition $A$ into $\sqrt{N}$ sub arrays of size $\sqrt{N}$. Recursively sort each subarray.
2: $R \leftarrow \text{ComputeSplitters}(A, \sqrt{N})$
3: Sort$(R)$ recursively. $R = \{r_0, r_1, \ldots, r_{\sqrt{N}}\}$.
4: Calculate counts $c_i$ such that $c_i = |\{x \mid x \in A \text{ and } r_i \leq x < r_{i+1}\}|$
5: $c_{1+\sqrt{N}} = |A| - \sum_i c_i$.
6: Distribute $A$ into buckets $B_0, B_1, \ldots B_{1+\sqrt{N}}$ where last element of each bucket is $r_i$ except the last bucket. For the last bucket, the last element is maximum element of $A$. Note that $|B_i| = c_i$.
7: Recursively sort each $B_i$
8: Copy sorted buckets back to $A$. 
Analysis of distribution sort

• Assume for simplicity that the pivots are chosen such that each bucket has size \( \sqrt{n} \).

• Disregarding the \( 2\sqrt{n} \) recursive calls to subproblems of size \( \sqrt{n} \), the algorithm has at most \( 8n/B \) cache misses.

• The recursive calls at the first level incur at most \( 2\sqrt{n} \cdot 8\sqrt{n}/B = 16n/B \) cache misses.

• At the third level there are at most \( 32n/B \) cache misses.
Analysis part 2

- At some level $i$ of the recursion, subproblems have size $n_i < M$.
- By the tall cache assumption, these subproblems have size at least $B$.
- Since we have an ideal cache, these subproblems are solved in $O(n_i/B)$ cache misses.
- The dominant number of cache misses will be at recursive level $i-1$. 
Analysis part 3

- The size of a subproblem at recursive level $j$ is $n^{1/2^j}$.
- Therefore $i = \log_2 (\log_2 n / \log_2 M)$.
- The number of cache misses at level $i-1$ is $O(2^{i-1} n/B) = O(n/B \cdot \log_M n)$.
- Same bound as in the external memory model (recall the tall cache assumption).
Distribution sort in practice

**Implementation:**

- Only one level of recursion (below which a standard sorting algorithm is used)
- Uses random sampling for pivots
- Uses a cache oblivious counting phase to compute bucket sizes
Experimental results

$$\text{Base} = 2^{14} = 16384$$
Is the model oversimplified?

- **Associativity** (Not fully associative)
- **Complicated algorithms** (Asymptotics hides disasters!)
- **Unified caches** (Instruction caches)
- **TLB** (not tall)
- **Concurrency** (Coherence misses: Xeon)
- **Replacement policy** (4Gb limit)
- **Multiple disks** (Can increase I/O speed)
- **Write through caches** (Causes misses even if problem fits into cache)
**DO’s and DON'Ts**

**DO's**
- Scans
- D&C, recursion
- Blocking
- Inline judiciously
- Local access
- Experiment with different algorithms

**DON'Ts**
- Non-local access
- Believe asymptotics
- Ignore the platform/compiler
- Believe models always
- Believe simulation results

**Sometimes**
- Sub-optimal can be better than optimal
- Cache aware might be the way to go
- Several optimal cache oblivious algorithm are available and only one is good among them
Optimal cache oblivious algorithms are automatically optimal at all levels of the memory hierarchy. A number of cache oblivious generalizations of external memory algorithms are known. A number of issues not covered by the cache oblivious model are important in real-world settings. • Summary
Talk on cache obliviousness

- On Wednesday April 30.
- **Speaker**: Gerth Brodal, BRICS, Århus.
- **Topic**: Cache oblivious sorting and searching (survey + recent research).
- **Place**: Will be announced by e-mail.