Part 2, course 3: Parallel External Memory and Cache Oblivious Algorithms

CR10: Data Aware Algorithms

September 23-24, 2020
Outline

Parallel External Memory

Cache Complexity of Multithreaded Computations

Experiments with Matrix Multiplication
Outline

Parallel External Memory
- Model
- Prefix Sum
- Sorting
- List Ranking

Cache Complexity of Multithreaded Computations
- Multicore Memory Model
- Multithreaded Computations
- Parallel Scheduling of Multithreaded Computations
- Work Stealing Scheduler
- Conclusion

Experiments with Matrix Multiplication
- Model and Metric
- Algorithm and Data Layout
- Results
Outline

Parallel External Memory
  Model
  Prefix Sum
  Sorting
  List Ranking

Cache Complexity of Multithreaded Computations
  Multicore Memory Model
  Multithreaded Computations
  Parallel Scheduling of Multithreaded Computations
  Work Stealing Scheduler
  Conclusion

Experiments with Matrix Multiplication
  Model and Metric
  Algorithm and Data Layout
  Results
Parallel External Memory Model

Classical model of parallel computation: PRAM

- $P$ processor
- Flat memory (RAM)
- Synchronous execution
- Concurrency models: Concurrent/Exclusive Read/Write (CRCW, CREW, EREW)

Extension to external memory:

- Each processor has its own (private) internal memory, size $M$
- Infinite external memory
- Data transfers between memories by blocks of size $B$

PEM I/O complexity: nb of parallel block transfers
Other metrics: parallel computation time, total space
Outline

Parallel External Memory
  Model
  Prefix Sum
  Sorting
  List Ranking

Cache Complexity of Multithreaded Computations
  Multicore Memory Model
  Multithreaded Computations
  Parallel Scheduling of Multithreaded Computations
  Work Stealing Scheduler
  Conclusion

Experiments with Matrix Multiplication
  Model and Metric
  Algorithm and Data Layout
  Results
Prefix Sum in PEM

Definition (All-Prefix-Sum).

Given an ordered set $A$ of $N$ elements, compute an ordered set $B$ such that $B[i] = \sum_{k \leq i} A[i]$.

Theorem.

All-Prefix-Sum can be solved with optimal $O(N/PB + \log P)$ PEM I/O complexity.

Same algorithm as in PRAM:

1. Each processors sums $N/P$ elements
2. Compute partial sums using pointer jumping
3. Each processor distributes (adds) the results to its $N/P$ elements

Analysis:

- Phases 1 and 3: linear scan of the data $O(N/PB)$ I/Os
- Phase 2: at most $O(1)$ I/O per step: $O(\log P)$ I/Os
Outline

Parallel External Memory
  Model
  Prefix Sum
  Sorting
  List Ranking

Cache Complexity of Multithreaded Computations
  Multicore Memory Model
  Multithreaded Computations
  Parallel Scheduling of Multithreaded Computations
  Work Stealing Scheduler
  Conclusion

Experiments with Matrix Multiplication
  Model and Metric
  Algorithm and Data Layout
  Results
Sorting in PEM

Theorem (Mergesort in PEM).

We can sort $N$ items in the CREW PEM model using $P \leq N/B^2$ processors each having cache of size $M = B^{O(1)}$ in $O(N/P \log N)$ internal complexity with $O(N)$ total memory and a parallel I/O complexity of:

$$O \left( \frac{N}{PB} \log \frac{M}{B} \frac{N}{B} \right)$$

Proof: much more involved than the one for (sequential) external memory.
Outline

Parallel External Memory
  Model
  Prefix Sum
  Sorting
  List Ranking

Cache Complexity of Multithreaded Computations
  Multicore Memory Model
  Multithreaded Computations
  Parallel Scheduling of Multithreaded Computations
  Work Stealing Scheduler
  Conclusion

Experiments with Matrix Multiplication
  Model and Metric
  Algorithm and Data Layout
  Results
List Ranking and its applications

List ranking:
- Very similar to All-Prefix-Sum: compute sum of previous elements
- But initial data stored as linked list
- Not contiguous in memory 😞

Application:
- Euler tours for trees → Computation of depths, subtree sizes, pre-order/post-order indices, Lowest Common Ancestor, ...
- Many problems on graphs: minimum spanning tree, ear decomposition, ...
List Ranking in PEM

In PRAM: pointer jumping, but very bad locality 😞

Algorithm sketch for PEM:
1. Compute large independent set $S$
2. Remove node from $S$ (add bridges)
3. Solve recursively on remaining nodes
4. Extend to nodes in $S$

NB: Operations on steps 2 and 4 require only neighbors.

Lemma.
An operation on items of a linked list which require access only to neighbors can be done in $O(sort_P(N))$ PEM I/O complexity.
Computing an independent set 1/2

Objective:
- Independent set of size $\Omega(N)$
- Or bound on distance between elements

Problem: $r$-ruling set:
- There are at most $r$ items in the list between two elements of the set

Randomized algorithm
1. Flip a coin for each item: $c_i \in \{0, 1\}$
2. Select items such that $c_i = 1$ and $c_{i+1} = 0$

- Two consecutive items are not selected.
- On average, $N/4$ items are selected
Computing an independent set 1/2

Objective:
▶ Independant set of size $\Omega(N)$
▶ Or bound on distance between elements

Problem: $r$-ruling set:
▶ There are at most $r$ items in the list between two elements of the set

**Randomized algorithm**

1. Flip a coin for each item: $c_i \in \{0, 1\}$
2. Select items such that $c_i = 1$ and $c_{i+1} = 0$

▶ Two consecutive items are not selected.
▶ On average, $N/4$ items are selected
Deterministic coin flipping

1. Choose unique item IDs
2. Compute tag of each item: $2i + b$
   - $i$: smallest index of different bits in item ID and successor ID
   - $b$: this bit in the current item
3. Select items with (local) minimum tags

- Successive items have different tags
- At most $\log N$ tag values
  $\Rightarrow$ distance between minimum tags $\leq 2 \log N$
- To decrease this value, re-apply step 2 on tags ($tags$ of $tags$)
- Number of steps to get constant size $k = \log^* N$

PEM I/O complexity: $O(\text{sort}_P(N) \cdot \log^* N)$
Outline

Parallel External Memory
  Model
  Prefix Sum
  Sorting
  List Ranking

Cache Complexity of Multithreaded Computations
  Multicore Memory Model
  Multithreaded Computations
  Parallel Scheduling of Multithreaded Computations
  Work Stealing Scheduler
  Conclusion

Experiments with Matrix Multiplication
  Model and Metric
  Algorithm and Data Layout
  Results
Outline

Parallel External Memory
  Model
  Prefix Sum
  Sorting
  List Ranking

Cache Complexity of Multithreaded Computations
  Multicore Memory Model
  Multithreaded Computations
  Parallel Scheduling of Multithreaded Computations
  Work Stealing Scheduler
  Conclusion

Experiments with Matrix Multiplication
  Model and Metric
  Algorithm and Data Layout
  Results
Parallel Cache Oblivious Processing

In classical cache-oblivious setting:

- **Cache and block sizes** unknown to the algorithms
- **Paging mechanism:** loads and evicts blocks (based on $M$ and $B$)

When considering parallel systems:

- Same assumption on cache and block sizes
- Also **unknown number of processors** (or processing cores)
- **Scheduler:** (platform aware) places threads on processors
- **Paging mechanism:** as in sequential case

Focus on **dynamically unrolled multithreaded computations**.
Multicore Memory Hierarchy

Model of computation:

- $P$ processing cores (=processors)
- Infinite memory
- Shared L2 cache of size $C_2$
- Private L1 caches of size $C_1$, with $C_2 \geq P \cdot C_1$

When a processor reads the data:

- If in its own L1 cache: no i/O
- Otherwise, if in L2 cache, or in other L1 cache: L1 miss
- Otherwise: L2 miss

When a processor writes a data:

- Stored in its L1 cache, invalidated in other caches (thanks to cache coherency protocol)

Two I/O metrics:

- Shared cache complexity: number of L2 misses
- Distributed cache complexity: total number of L1 misses (sum)
Multicore Memory Hierarchy

Model of computation:
- \( P \) processing cores (\( = \)processors)
- Infinite memory
- Shared L2 cache of size \( C_2 \)
- Private L1 caches of size \( C_1 \), with \( C_2 \geq P \cdot C_1 \)
- When a processor reads the data:
  - if in its own L1 cache: no i/O
  - otherwise, if in L2 cache, or in other L1 cache: L1 miss
  - otherwise: L2 miss
- When a processor writes a data:
  Stored in its L1 cache, invalidated in other caches (thanks to cache coherency protocol)
- Two I/O metrics:
  - Shared cache complexity: number of L2 misses
  - Distributed cache complexity: total number of L1 misses (sum)
Multicore Memory Hierarchy

Model of computation:
- $P$ processing cores (processors)
- Infinite memory
- Shared L2 cache of size $C_2$
- Private L1 caches of size $C_1$, with $C_2 \geq P \cdot C_1$

When a processor reads the data:
- if in its own L1 cache: no i/O
- otherwise, if in L2 cache, or in other L1 cache: L1 miss
- otherwise: L2 miss

When a processor writes a data:
- Stored in its L1 cache, invalidated in other caches (thanks to cache coherency protocol)

Two I/O metrics:
- Shared cache complexity: number of L2 misses
- Distributed cache complexity: total number of L1 misses (sum)
Multicore Memory Hierarchy

Model of computation:

- $P$ processing cores (=processors)
- Infinite memory
- Shared L2 cache of size $C_2$
- Private L1 caches of size $C_1$, with $C_2 \geq P \cdot C_1$

When a processor reads the data:

- if in its own L1 cache: no i/O
- otherwise, if in L2 cache, or in other L1 cache: L1 miss
- otherwise: L2 miss

When a processor writes a data:

Stored in its L1 cache, invalidaded in other caches (thanks to cache coherency protocol)

Two I/O metrics:

- Shared cache complexity: number of L2 misses
- Distributed cache complexity: total number of L1 misses (sum)
Outline

Parallel External Memory
  Model
  Prefix Sum
  Sorting
  List Ranking

Cache Complexity of Multithreaded Computations
  Multicore Memory Model
  Multithreaded Computations
  Parallel Scheduling of Multithreaded Computations
  Work Stealing Scheduler
  Conclusion

Experiments with Matrix Multiplication
  Model and Metric
  Algorithm and Data Layout
  Results
Multithreaded computations 1/2

Threads:
- Sequential execution of instructions
- Each thread has its own activation frame (memory)
- May launch (spawn) other threads (children)
- Can wait for completion or messages from other threads
- DAG of instructions
  - Continue edges: within same thread
  - Spawn edges: to create new thread
  - Join edges: message to other threads/completion
- Dynamic behavior: may depend on the data
  (execution graph unknown before the computation)

Constraints:
- Strict computation: Join edges only directed to ancestors in the activation tree
- Fully strict computation: Join edges only directed to parent in the activation tree → Series-Parallel graph of instructions
Multithreaded computations 1/2

Threads:
- Sequential execution of instructions
- Each thread has its own activation frame (memory)
- May launch (spawn) other threads (children)
- Can wait for completion or messages from other threads
- DAG of instructions
  - Continue edges: within same thread
  - Spawn edges: to create new thread
  - Join edges: message to other threads/completion
- Dynamic behavior: may depend on the data
  (execution graph unknown before the computation)

Constraints:
- **Strict computation**: Join edges only directed to ancestors in the activation tree
- **Fully strict computation**: Join edges only directed to parent in the activation tree
  → Series-Parallel graph of instructions
Makespan Bound

Classical bound on total duration:

- **Work** $W = T_1$: total (weighted) number of instructions
- **Critical path** (or span) $T_\infty$: length of longest path
- **Greedy scheduling**: running time (makespan) bounded by $T_1/P + T_\infty$
- **Tight bound** (no better schedule) for some computations
In the sequential case:
- Natural order: **Depth-First traversal (1DF)**
- **Queue** (stack) of threads
- Whenever a thread is spawned:
  - **Current thread** put in the queue
  - **Newly created thread** executed
Outline

Parallel External Memory
  Model
  Prefix Sum
  Sorting
  List Ranking

Cache Complexity of Multithreaded Computations
  Multicore Memory Model
  Multithreaded Computations
  Parallel Scheduling of Multithreaded Computations
  Work Stealing Scheduler
  Conclusion

Experiments with Matrix Multiplication
  Model and Metric
  Algorithm and Data Layout
  Results
Parallel Depth First Scheduling (PDF)

Parallel adaptation of 1DF, targeting shared memory

- Global pool of ready threads
- Same behavior as 1DF when spawning threads
- When a processor is idle (current threads stalls or dies): it starts working on the next thread that would be activated by the 1DF sequential scheduler
- When thread enabled (unlocked from stall), put in the pool

Theorem (Shared cache complexity).
Let $C_1$ (resp. $C_P$) be the size of the cache for 1DF (resp. PDF). If $C_P \geq C_1 + PT_\infty$, then PDF does at most as many shared cache misses as 1DF.

Corollary (Memory Usage)
Assuming unlimited memory, if the sequential depth first schedule uses a memory of $M_1$, the work stealing execution uses at most a memory of $M_1 + PT_\infty$. 
Parallel Depth First Scheduling (PDF)

Parallel adaptation of 1DF, targeting shared memory

- Global pool of ready threads
- Same behavior as 1DF when spawning threads
- When a processor is idle (current threads stalls or dies): it starts working on the next thread that would be activated by the 1DF sequential scheduler
- When thread enabled (unlocked from stall), put in the pool

**Theorem (Shared cache complexity).**

Let $C_1$ (resp. $C_P$) be the size of the cache for 1DF (resp. PDF). If $C_P \geq C_1 + PT_\infty$, then PDF does at most as many shared cache misses as 1DF.

**Corollary (Memory Usage)**

Assuming unlimited memory, if the sequential depth first schedule uses a memory of $M_1$, the work stealing execution uses at most a memory of $M_1 + PT_\infty$. 
Scheduler for Multicore Memory Hierarchy

Contradictory objectives:
- Re-use data as much as possible in shared cache
- Work on disjoint datasets in private caches

Focus: divide-and-conquer algorithms

Simple recurrence relations:
\[
T(n) = t(n) + a \cdot T(n/b) \quad \text{(seq. time complexity)}
\]
\[
Q(M, n) = q(M, n) + q \cdot Q(M, n/b) \quad \text{(seq. cache complexity)}
\]

Hierarchical recurrence relations:
\[
T_k(n) = t_k(n) + a_k \cdot T_k(n/b_k) + \sum_{i<k} a_{k,i} \cdot T_i(n/b_i)
\]
\[
Q_k(M, n) = q_k(M, n) + a_k \cdot Q_k(M, n/b_k) + \sum_{i<k} a_{k,i} \cdot Q_i(M, n/b_i)
\]

Sequential space complexity: \(S(n)\)

\(r\): ratio between parallel and sequential space complexity
Scheduler for Multicore Memory Hierarchy

Contradictory objectives:
▶ Re-use data as much as possible in shared cache
▶ Work on disjoint datasets in private caches

Focus: divide-and-conquer algorithms
▶ Simple recurrence relations:

\[ T(n) = t(n) + a T(n/b) \] (seq. time complexity)
\[ Q(M, n) = q(M, n) + qQ(M, n/b) \] (seq. cache complexity)

▶ Hierarchical recurrence relations:

\[ T_k(n) = t_k(n) + a_k T_k(n/b_k) + \sum_{i<k} a_{k,i} T_i(n/b_i) \]
\[ Q_k(M, n) = q_k(M, n) + a_k Q_k(M, n/b_k) + \sum_{i<k} a_{k,i} Q_i(M, n/b_i) \]

▶ Sequential space complexity: \( S(n) \)
▶ \( r \): ratio between parallel and sequential space complexity
Scheduler for Multicore Memory Hierarchy

Contradictory objectives:
- Re-use data as much as possible in shared cache
- Work on disjoint datasets in private caches

Focus: divide-and-conquer algorithms
- Simple recurrence relations:
  \[ T(n) = t(n) + a T(n/b) \] (seq. time complexity)
  \[ Q(M, n) = q(M, n) + qQ(M, n/b) \] (seq. cache complexity)

- Hierarchical recurrence relations:
  \[ T_k(n) = t_k(n) + a_k T_k(n/b_k) + \sum_{i < k} a_{k,i} T_i(n/b_i) \]
  \[ Q_k(M, n) = q_k(M, n) + a_k Q_k(M, n/b_k) + \sum_{i < k} a_{k,i} Q_i(M, n/b_i) \]

- Sequential space complexity: \( S(n) \)
- \( r \): ratio between parallel and sequential space complexity
Controlled Parallel Depth-First

- **L1-supernodes**: of size $n_1 = S^{-1}(C_1)$
- **L2-supernodes**: of size $n_2 = S^{-1}(C_2/r)$
- Split recursion tree into L2 supernodes, executed one after the others
- Within a L2-supernode, distribute L1-supernodes to cores
- Optimal parallel speedup if enough L1-supernodes within one L2-supernode

---

**Theorem (Cache complexities).**

Asymptotically optimal L1 and L2 cache complexities:

$$Q_{L1}(n) = O(Q_k(C_1, n)) \quad \text{and} \quad Q_{L2}(n) = O(Q_k(C_2, n))$$
Outline

Parallel External Memory
  Model
  Prefix Sum
  Sorting
  List Ranking

Cache Complexity of Multithreaded Computations
  Multicore Memory Model
  Multithreaded Computations
  Parallel Scheduling of Multithreaded Computations
  Work Stealing Scheduler
  Conclusion

Experiments with Matrix Multiplication
  Model and Metric
  Algorithm and Data Layout
  Results
Work Stealing Scheduler

First ideas in the 1980s, formalised in the 1990s, now implemented in several thread schedulers (CILK, Java fork/join, Kaapi, etc.)

Distributed and dynamic scheduler:

▶ Each processor has its own local queue of ready threads
▶ Local queue stored as a deque (double-ended queue)
▶ When spawning a thread:
  ▶ Current thread placed at the bottom of the local queue
  ▶ Newly created thread executed
▶ When a processor is idle:
  ▶ If work in the local queue: pick thread at the bottom
  ▶ Otherwise, steal thread from the top of a random remote queue
▶ Thread enabled: put at the bottom of the local queue

NB: Do not rely on platform characteristics.
Work Stealing: Running Time Analysis

(Similar results for many platform/computation models)

Theorem (Running time).

For a computation with work $T_1$ and critical path $T_\infty$, the schedule obtained by work stealing has an expected duration of $T_1/P + O(T_\infty)$. Furthermore, the duration is bounded by $T_1/P + O(T_\infty + \log P + \log 1/\epsilon)$ with probability at least $1 - \epsilon$.

Theorem (Number of steals).

The number of steal attempts is bounded by $O(PT_\infty)$.

Theorem (Communication time).

The time spent in sending data among processors is bounded by $O(PT_\infty(1 + n_d)M_{\text{max}})$ where:

- $M_{\text{max}}$: maximal memory on a processor
- $n_d$: maximum number of join edges to parent
Work Stealing: Running Time Analysis

(Similar results for many platform/computation models)

Theorem (Running time).

For a computation with work $T_1$ and critical path $T_\infty$, the schedule obtained by work stealing has an expected duration of $T_1/P + O(T_\infty)$. Furthermore, the duration is bounded by $T_1/P + O(T_\infty + \log P + \log 1/\epsilon)$ with probability at least $1 - \epsilon$.

Theorem (Number of steals).

The number of steal attempts is bounded by $O(PT_\infty)$.

Theorem (Communication time).

The time spent in sending data among processors is bounded by $O(PT_\infty(1 + n_d)M_{\text{max}})$ where:

- $M_{\text{max}}$: maximal memory on a processor
- $n_d$: maximum number of join edges to parent
Work Stealing: Running Time Analysis

(Similar results for many platform/computation models)

**Theorem (Running time).**

For a computation with work $T_1$ and critical path $T_\infty$, the schedule obtained by work stealing has an expected duration of $T_1/P + O(T_\infty)$. Furthermore, the duration is bounded by $T_1/P + O(T_\infty + \log P + \log 1/\epsilon)$ with probability at least $1 - \epsilon$.

**Theorem (Number of steals).**

The number of steal attempts is bounded by $O(PT_\infty)$.

**Theorem (Communication time).**

The time spent in sending data among processor is bounded by $O(PT_\infty(1 + n_d)M_{\text{max}})$ where:

- $M_{\text{max}}$: maximal memory on a processor
- $n_d$: maximum number of join edges to parent
Working Stealing: Cache Complexity and Memory

**Theorem (Shared Cache Complexity).**
If the memory for the sequential depth first schedule is $M_1$ and work stealing is given a memory of $PM_1$, its shared cache complexity is in $O(Q_1)$, where $Q_1$ is the cache complexity of the sequential schedule.

**Corollary (Memory usage)**
*Assuming unlimited memory, if the sequential schedule uses a memory of $M_1$, the work stealing execution uses a memory of $PM_1$.*

**Theorem (Distributed Cache Complexity).**
For series-parallel computations, the distributed cache complexity of work stealing is bounded by $Q_1(Z) + O(ZPT_\infty)$ where $Z$ is the size of each distributed cache and $Q_1$ is the sequential cache complexity.

*NB: for non SP computations, unbounded dist. cache complexity*
Theorem (Shared Cache Complexity).
If the memory for the sequential depth first schedule is $M_1$ and work stealing is given a memory of $PM_1$, its shared cache complexity is in $O(Q_1)$, where $Q_1$ is the cache complexity of the sequential schedule.

Corollary (Memory usage)
Assuming unlimited memory, if the sequential schedule uses a memory of $M_1$, the work stealing execution uses a memory of $PM_1$.

Theorem (Distributed Cache Complexity).
For series-parallel computations, the distributed cache complexity of work stealing is bounded by $Q_1(Z) + O(ZPT_\infty)$ where $Z$ is the size of each distributed cache and $Q_1$ is the sequential cache complexity.

NB: for non SP computations, unbounded dist. cache complexity
Outline

Parallel External Memory
   Model
   Prefix Sum
   Sorting
   List Ranking

Cache Complexity of Multithreaded Computations
   Multicore Memory Model
   Multithreaded Computations
   Parallel Scheduling of Multithreaded Computations
   Work Stealing Scheduler

Conclusion

Experiments with Matrix Multiplication
   Model and Metric
   Algorithm and Data Layout
   Results
Conclusion on Schedulers

Parallel-Depth First:
▶ Bound for shared memory
▶ Adaptation to memory hierarchies: Controlled PDF

Work-Stealing:
▶ Very simple: amenable both to analysis and implementation
  ▶ Bounds on running time, number of steals, communications, etc. in various models
  ▶ Present in several real-world thread schedulers
▶ Bounds on shared and distributed cache complexities
▶ Data-locality problem for distributed platforms (clusters)
▶ Trade-off between:
  ▶ Fixed data distribution for (load balance and) locality
  ▶ Dynamic work-stealing for real-time load balance
Outline

Parallel External Memory
  Model
  Prefix Sum
  Sorting
  List Ranking

Cache Complexity of Multithreaded Computations
  Multicore Memory Model
  Multithreaded Computations
  Parallel Scheduling of Multithreaded Computations
  Work Stealing Scheduler
  Conclusion

Experiments with Matrix Multiplication
  Model and Metric
  Algorithm and Data Layout
  Results
Outline

Parallel External Memory
  Model
  Prefix Sum
  Sorting
  List Ranking

Cache Complexity of Multithreaded Computations
  Multicore Memory Model
  Multithreaded Computations
  Parallel Scheduling of Multithreaded Computations
  Work Stealing Scheduler
  Conclusion

Experiments with Matrix Multiplication
  Model and Metric
  Algorithm and Data Layout
  Results
Platform Model

- Multicore with \( p \) cores
- Different cache bandwidths
- New metric: data access time

\[
T_{\text{data}} = \frac{M_S}{\sigma_S} + \frac{M_D}{\sigma_D}
\]

- \( M_S \): nb of shared cache misses
- \( M_D \): nb of distributed cache misses
- Largest block size in shared cache: \( \lambda \times \lambda \)
- Largest block size in distributed cache: \( \mu \times \mu \)
Outline

Parallel External Memory
  Model
  Prefix Sum
  Sorting
  List Ranking

Cache Complexity of Multithreaded Computations
  Multicore Memory Model
  Multithreaded Computations
  Parallel Scheduling of Multithreaded Computations
  Work Stealing Scheduler
  Conclusion

Experiments with Matrix Multiplication
  Model and Metric
  Algorithm and Data Layout
  Results
Minimizing Data Access Time

- when $\alpha = \lambda$, we optimize for shared-memory
- when $\alpha^2 = p \times \lambda^2$, we optimize for distributed-memory
- Constraint: $2\alpha \times \beta + \alpha^2 \leq C_S$
- Minimize $T_{\text{data}} = \frac{1}{\sigma_S} (mn + \frac{2mnz}{\alpha}) + \frac{1}{\sigma_D} (\frac{mnz}{p\beta} + \frac{2mnz}{p\mu})$
Outline

Parallel External Memory
  Model
  Prefix Sum
  Sorting
  List Ranking

Cache Complexity of Multithreaded Computations
  Multicore Memory Model
  Multithreaded Computations
  Parallel Scheduling of Multithreaded Computations
  Work Stealing Scheduler
  Conclusion

Experiments with Matrix Multiplication
  Model and Metric
  Algorithm and Data Layout
  Results
Results on multicore CPU

- Intel Xeon E5520 processor (quad-core) running at 2.26 GHz.
- Shared L3 of 8MB (16-way associative)
- Distributed L2 256KB (8-way associative)
- All variants reach about 89% of GotoBlas2 (same for MKL)
- Our strategy perform less cache misses
- GotoBlas2: more regular memory accesses
  ⇒ automatic prefetch is much more efficient
Results on GPUs

GPU architecture: similar tradeoff

- Several Streaming Multiprocessor (many simple cores, SIMD)
- Limited GPU memory (at this time) \(\sim\) shared cache
- L1 \(\sim\) distributed cache
Results on GPUs

- Running times on GeForce GTX285 with 240 cores and 2GB global memory
- Results: depend on the matrix size
- Cublas uses different kernels depending on size
  Some kernels use GPU-specific features (texture units)
- On average Cublas performs 40% more shared cache misses and 90%–240% more distributed cache misses