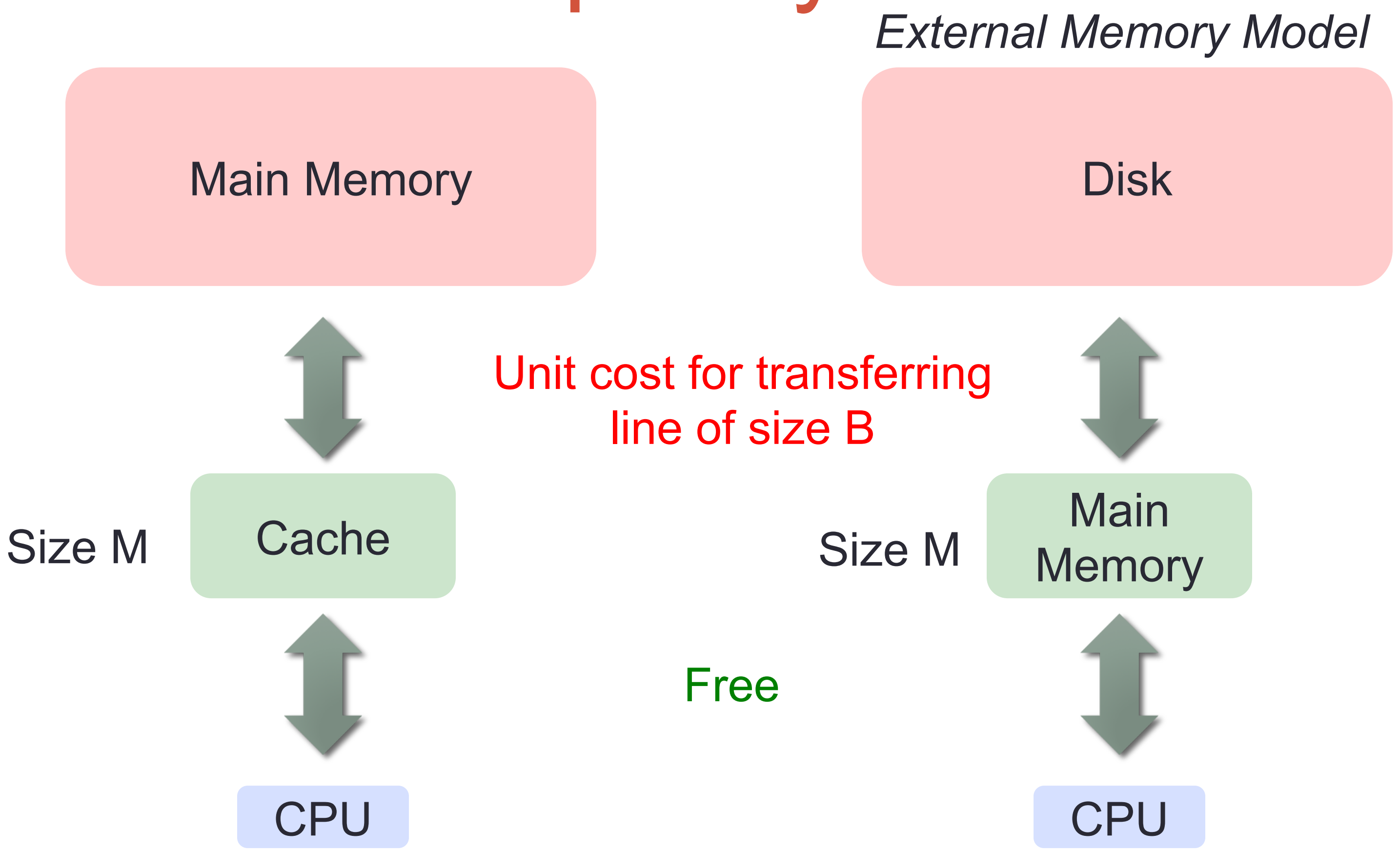
Low Depth Cache-Oblivious Algorithms

Authors: Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri

Presented by Julian Shun

Cache Complexity Model



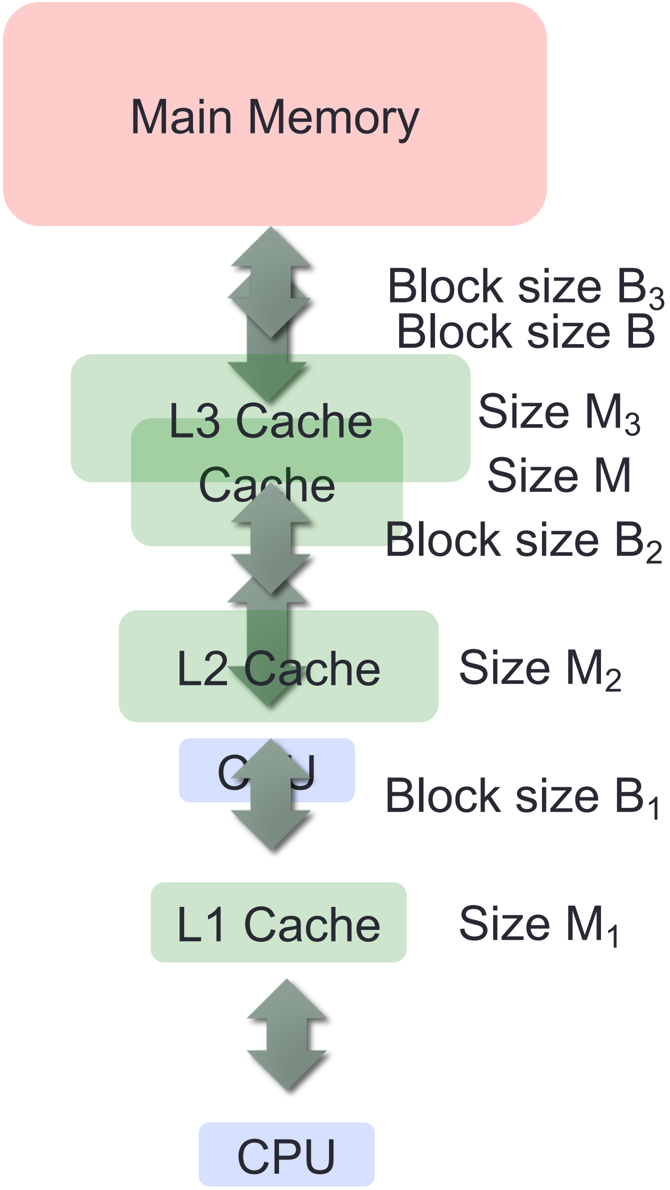
Complexity = # ~~cache misses~~ disk accesses

Cache-aware (external-memory) algorithms: have knowledge of M and B

Cache-oblivious algorithms: no knowledge of parameters

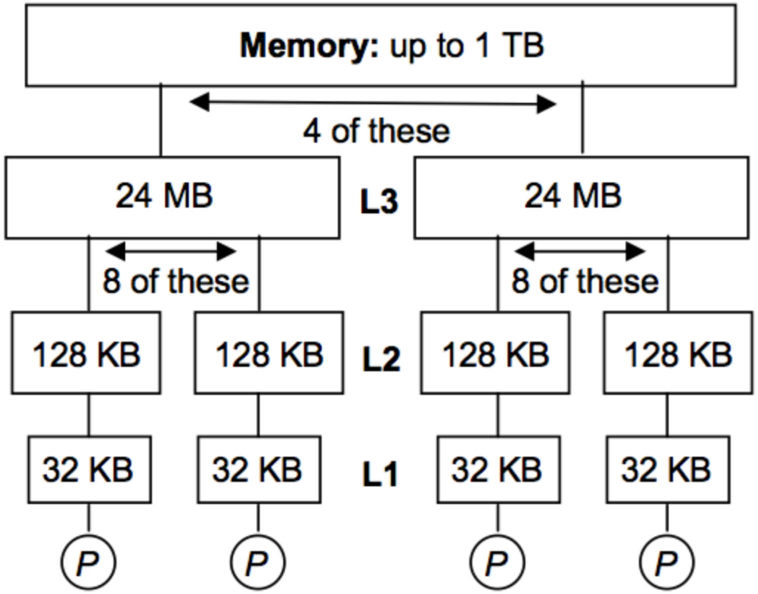
Cache Oblivious Model [Frigo et al. '99]

- Algorithm works well regardless of cache parameters
- Works well on multi-level hierarchies
- Simplifies algorithm design and implementation due to not having to tune for specific machine parameters
- Implementations are portable across different machines



Parallel Cache Oblivious Model

- Parallel Cache Oblivious Model for hierarchies of shared and private caches [Blelloch et al. '11]
- Parallel programs are often memory bound
- Even harder to manually tune algorithms for parameters of parallel machines
- Existing parallel cache bounds:
  - $Q_p(n; M, B) < Q(n; M, B) + O(pMD/B)$ for private caches using work-stealing scheduler
  - $Q_p(n; M+pDB, B) < Q(n; M, B)$ for shared cache using parallel depth-first (PDF) scheduler
- Recipe for parallel cache-oblivious algorithms:
  - Design low-depth algorithms with low sequential cache complexity



Algorithms

Primitive	Work	Depth	Cache Complexity
Scan/filter/merge	O(n)	O(log n)	O(n/B)
Sort	O(n log n)	$O(\log^2 n)$	$O((n/B)\log_{(M/B)}(n/B))$
Matrix Transpose	O(nm)	O(log(n+m))	O(nm/B)
SpMV (n^{ε}-separator)	O(m)	$O(\log^2 n)$	$O(m/B+n/M^{1-\varepsilon})$
Many graph algorithms	$O(W_{sort}\text{polylog}(m))$	$O(D_{sort}\text{polylog}(m))$	$O(Q_{sort}\text{polylog}(m))$

Merge and Mergesort

- Input: arrays A and B where $|A|+|B|=n$
- For $k\in[1,\ldots,n^{1/3}]$ pick pivots such that $a_k+b_k=kn^{2/3}$ and $A[a_k] \leq B[b_k+1]$ and $B[b_k] \leq A[a_k+1]$ using dual binary search*
- Recursively merge each of the $n^{1/3}$ subproblems created by the pivots until reaching base case
- $W(n) = n^{1/3}W(n^{2/3})+O(n^{1/3}\log n) = O(n)$
- $D(n) = D(n^{2/3})+O(\log n) = O(\log n)$
- $Q(n; M, B) \leq O(n^{1/3} (\log(n/B)+Q(n^{2/3}; M, B)))$ if $n > cM$
  $\leq O(n/B)$ otherwise (base case)
- This solves to $Q(n; M, B) = O(n/B)$
- Plug this in to obtain cache-oblivious mergesort with $O(\log^2 n)$ depth and $O((n/B) \log_2(n/M))$ cache misses, which is sub-optimal

* http://blog.jzhanson.com/blog/practice/code/2018/01/08/algos-1.html

Deterministic Samplesort

1. Divide input into $\sqrt{n}$ subarrays of size $\sqrt{n}$ and sort them recursively
2. Choose every (log n)-th element from each subarray as a sample and sort the O(n/log n) samples using mergesort
3. Pick $\sqrt{n}$ evenly spaced keys from sorted samples to determine bucket boundaries and split subarrays according to bucket boundaries
4. Use prefix sums and matrix transpose to determine offsets into buckets
5. Move keys into buckets using B-TRANSPOSE
6. Recursively sort each bucket

Deterministic Samplesort

1. Divide input into $\sqrt{n}$ subarrays of size $\sqrt{n}$ and sort them recursively
2. Choose every (log n)-th element from each subarray as a sample and sort the O(n/log n) samples using mergesort
3. Pick $\sqrt{n}$ evenly spaced keys from sorted samples to determine bucket boundaries and split subarrays according to bucket boundaries
4. Use prefix sums and matrix transpose to determine offsets into buckets
5. Move keys into buckets using B-TRANSPOSE
6. Recursively sort each bucket

Work and depth: O((n/log n)*log n)=O(n) work, $O(\log^2 n)$ depth,
Cache complexity: $O(((n/\log n)/B) \log_2(n/M)) = O(n/B)$

Deterministic Samplesort

1. Divide input into √n subarrays of size √n and sort them recursively
2. Choose every (log n)-th element from each subarray as a sample and sort the O(n/log n) samples using mergesort
3. Pick √n evenly spaced keys from sorted samples to determine bucket boundaries and split subarrays according to bucket boundaries
4. Use prefix sums and matrix transpose to determine offsets into buckets
5. Move keys into buckets using B-TRANSPOSE
6. Recursively sort each bucket

Split by merging subarray with array of pivots
Work and depth: O(n/B) work and O(log n) depth
Cache complexity: O(n/B)

Deterministic Samplesort

1. Divide input into $\sqrt{n}$ subarrays of size $\sqrt{n}$ and sort them recursively
2. Choose every (log n)-th element from each subarray as a sample and sort the O(n/log n) samples using mergesort
3. Pick $\sqrt{n}$ evenly spaced keys from sorted samples to determine bucket boundaries and split subarrays according to bucket boundaries
4. Use prefix sums and matrix transpose to determine offsets into buckets
5. Move keys into buckets using B-TRANSPOSE
6. Recursively sort each bucket

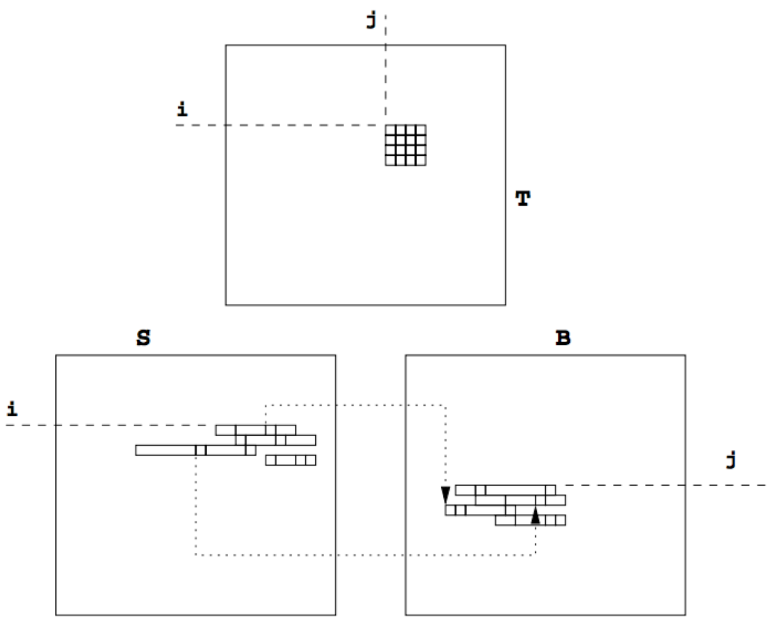
Work and depth: O(n/B) work and O(log n) depth
Cache complexity: O(n/B)

B-TRANSPOSE

- Naïvely moving elements into buckets can incur one cache miss per transfer, for a total of O(n)
- B-TRANSPOSE: cache-oblivious divide-and-conquer method for transferring keys into the appropriate buckets

Algorithm B-TRANSPOSE(S, B, T, i_s, i_b, n)
if $(n = 1)$ **then**
  Copy $S_{i_s}[T_{i_s,i_b}\langle 1\rangle : T_{i_s,i_b}\langle 1\rangle + T_{i_s,i_b}\langle 3\rangle)$
  to $B_{i_b}[T_{i_s,i_b}\langle 2\rangle : T_{i_s,i_b}\langle 2\rangle + T_{i_s,i_b}\langle 3\rangle)$
else
  B-TRANSPOSE($S, B, T, i_s, i_b, n/2$)
  B-TRANSPOSE($S, B, T, i_s, i_b + n/2, n/2$)
  B-TRANSPOSE($S, B, T, i_s + n/2, i_b, n/2$)
  B-TRANSPOSE($S, B, T, i_s + n/2, i_b + n/2, n/2$)
end if

- Lemma: B-TRANSPOSE takes O(n) work, O(log n) depth, and O(n/B) cache misses



Bucket transpose diagram: The 4x4 entries shown for T dictate the mapping from the 16 depicted segments of S to the 16 depicted segments of B. Arrows highlight the mapping for two of the segments.

Deterministic Samplesort

1. Divide input into $\sqrt{n}$ subarrays of size $\sqrt{n}$ and sort them recursively
2. Choose every (log n)-th element from each subarray as a sample and sort the O(n/log n) samples using mergesort
3. Pick $\sqrt{n}$ evenly spaced keys from sorted samples to determine bucket boundaries and split subarrays according to bucket boundaries
4. Use prefix sums and matrix transpose to determine offsets into buckets
5. Move keys into buckets using B-TRANSPOSE
6. Recursively sort each bucket

Work and depth: O(n/B) work and O(log n) depth
Cache complexity: O(n/B)

Deterministic Samplesort

1. Divide input into $\sqrt{n}$ subarrays of size $\sqrt{n}$ and sort them recursively
2. Choose every (log n)-th element from each subarray as a sample and sort the $O(n/\log n)$ samples using mergesort
3. Pick $\sqrt{n}$ evenly spaced keys from sorted samples to determine bucket boundaries and split subarrays according to bucket boundaries
4. Use prefix sums and matrix transpose to determine offsets into buckets
5. Move keys into buckets using B-TRANSPOSE
6. Recursively sort each bucket

Can show that buckets will have size at most $2\sqrt{n}\log n$

Deterministic Samplesort

Using the fact that bucket sizes are at most 2√n log n

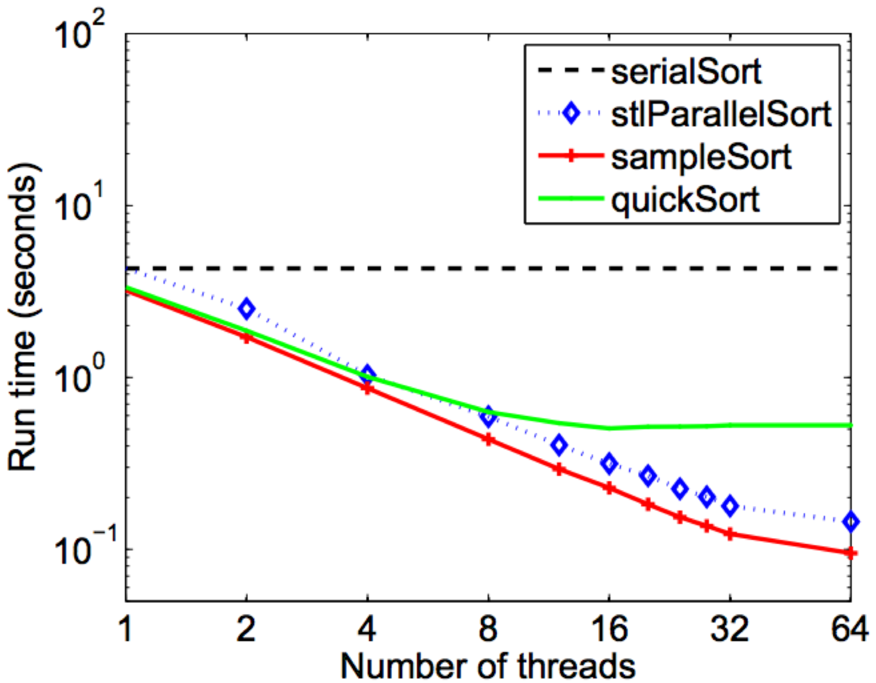
$$W(n) = O(n) + \sqrt{n}W(\sqrt{n}) + \sum_{i=1}^{\sqrt{n}} W(n_i) \quad = O(n \log n)$$

$$D(n) = O(\log^2 n) + \max_{i=1}^{\sqrt{n}}\{D(n_i)\} \quad + D(\sqrt{n}) = O(\log^2 n)$$

$$Q(n; M, B) = O\left(\left\lceil \frac{n}{B} \right\rceil\right) + \sqrt{n}Q(\sqrt{n}; M, B) + \sum_{i=1}^{\sqrt{n}} Q(n_i; M, B)$$

$$= O((n/B)\log_M n)$$

Randomized Samplesort Performance



(a) comparison sorting algorithms with a **trigram string of length** 10^7

- 32 cores with hyper-threading
- Cache-oblivious sample sort gets near linear speedup and outperforms stlParallelSort by 1.2 to 2.4x