Brief Announcement: Low Depth Cache-Oblivious Sorting

Guy E. Blelloch
Carnegie Mellon University
Pittsburgh, PA USA
guyb@cs.cmu.edu

Phillip B. Gibbons
Intel Reseach Pittsburgh
Pittsburgh, PA USA
phillip.b.gibbons@intel.com

Harsha Vardhan Simhadri
Carnegie Mellon University
Pittsburgh, PA USA
harshas@cs.cmu.edu

ABSTRACT

Cache-oblivious algorithms have the advantage of achieving good sequential cache complexity across *all* levels of a multi-level cache hierarchy, regardless of the specifics (cache size and cache line size) of each level. In this paper, we describe cache-oblivious sorting algorithms with optimal work, optimal cache complexity and polylogarithmic depth. Using known mappings, these lead to low cache complexities on shared-memory multiprocessors with a single level of private caches or a single shared cache. Moreover, the low cache complexities extend to shared-memory multiprocessors with common configurations of multi-level caches. The key factor in the low cache complexity on multiprocessors is the low depth of the algorithms we propose.

Categories and Subject Descriptors: F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems – *Sorting and Searching*. F.1.2 [Computation by abstract devices]: Modes of Computation – *Parallelism and Concurrency*. D.1.3 [Programming Techniques]: Concurrent Programming – *Parallel programming*.

General Terms: Algorithms, Performance, Theory

Keywords: Cache-oblivious algorithms, sorting, parallel algorithms, multiprocessors, schedulers

1. INTRODUCTION

The *cache-oblivious model* (*ideal-cache model*) [12] is a two-level model of computation composed of an unbounded memory and a cache of size Z. Data are transferred between the two levels using cache lines of size L; all computation occurs on data in the cache. Both Z and L are unknown to the algorithm, and the goal is to minimize an algorithm's *work* (number of operations) and *cache complexity* (number of cache lines transferred). Sequential algorithms designed for this model have the advantage of achieving good sequential cache complexity across *all* levels of a multi-level cache hierarchy, regardless of the values of Z_i and L_i at each level i [12]. Researchers have developed cache-oblivious algorithms for a variety of problems [10].

The cache complexity $Q_1(Z, L)$ for a natural sequential execution of a parallel program can also be used to bound the cache complexity $Q_P(Z, L)$ for the same program on certain P-processor parallel machines with a single level of cache(s) [1, 5]. In particular, for a shared-memory parallel machine with private caches (each processor has its own cache) using a work-stealing scheduler, $Q_P(Z, L) < Q_1(Z, L) + O(ZPD/L)$ with high probability [1], and for a shared cache using a PDF scheduler, $Q_P(Z + PD, L) \leq Q_1(Z, L)$ [5], where D is the depth of the computation. These results apply to nested-parallel computations—computations starting with a single thread and using (nested) fork-join parallelism—that use binary forking (spawning) of threads. The "natural" sequential execution is simply one that runs each call in a fork to completion before starting the next. The depth of a fork-join construct is determined by taking the maximum of the depths of the forked threads.

These results suggest a simple approach for developing cache-efficient parallel algorithms: Develop a nested-parallel algorithm with (1) low cache-oblivious complexity for the sequential ordering, and (2) low depth; then use the results above to bound the cache complexity on a parallel machine. Low depth is important because D shows up in the term for additional misses for private caches, and additional cache size for a shared cache. Moreover, we show that algorithms designed with this approach can also achieve good parallel cache complexity on parallel machines with common configurations of *multi-level* cache hierarchies.

As an example of the approach consider Strassen's matrix multiply. It is nested-parallel since the seven recursive calls can be made in parallel and the matrix addition can be implemented by forking off a tree of parallel calls. For $n \times n$ matrices the total depth is $O(\log^2 n)$—$O(\log n)$ levels of recursion, each with $O(\log n)$ depth for the additions. As shown in the original paper on cache-oblivious algorithms [12] $Q_1(n; Z, L) = n^{\lg 7}/(L\sqrt{Z})$. Therefore we have that $Q_P(n; Z, L) < n^{\lg 7}/(L\sqrt{Z}) + O(ZP\log^2(n)/L)$ for private caches and $Q_P(Z + P\log^2 n, L) \leq n^{\lg 7}/(L\sqrt{Z})$ for a shared cache. For practical parameters these bounds indicate either only marginally more total misses than the sequential version (private caches) or only marginally larger cache size (shared cache).

Although several known cache-oblivious algorithms are naturally parallel and have low depth (*e.g.*, matrix multiply, matrix transpose, FFT), others are not. In particular the cache-oblivious algorithms for sorting [12] are not parallel. This paper presents a low-depth cache-oblivious sorting algorithm. It has cache complexity $Q_1(n; Z, L) = O((n/L)\log_Z n)$ and work $W = O(n \log n)$, which are optimal, and depth $D = O(\log^2 n)$. The depth can be improved using randomization.

Other work on parallel cache-oblivious algorithms has concentrated on bounding cache misses for particular classes of algorithms. This includes results by Frigo *et al.* [13] for a class of algorithms with a regularity condition, by Blelloch *et al.* [4] for a class of binary divide-and-conquer algorithms, and by Chowdhury and Ramachandran [8, 9] for a class of dynamic programming and Gaussian elimination-style problems. Our design motive is to have a generic approach that works for a wide-class of algorithms and a variety of parallel machine configurations; we study sorting as a specific instance of our approach. Our work may also be contrasted

SPAA'09, August 11–13, 2009, Calgary, Alberta, Canada.
ACM 978-1-60558-606-9/09/08.

Table 1: Algorithmic complexity (assuming $Z = \Omega(L^2)$). All algorithms are work optimal. (*new algorithms)

Problem	Depth	Cache-complexity
Matrix Transpose	$O(\log(n+m))$	$O(\lceil nm/L \rceil)$
Prefix Sum	$O(\log n)$	$O(\lceil n/L \rceil)$
Merge	$O(\log n)$	$O(\lceil n/L \rceil)$
Sort (randomized)*	$O(\log^{3/2} n)$	$O(\lceil n/L \rceil \lceil \log_Z n \rceil)$
Sort (deterministic)*	$O(\log^2 n)$	$O(\lceil n/L \rceil \lceil \log_Z n \rceil)$

with that of [3], which demonstrates cache-efficient algorithms for private caches, the major difference being that their algorithms are not cache-oblivious and are tuned specifically for one level of the cache.

2. SORTING

We use known algorithms for matrix transpose, prefix sums, and merging as subroutines. The costs are summarized in Table 1. The standard divide-and-conquer matrix-transpose algorithm [12] is work optimal, has logarithmic depth and has optimal cache complexity when $Z = \Omega(L^2)$. A simple variant of the standard parallel prefix-sums algorithm has logarithmic depth and cache complexity $O(n/L)$ even with only a single cache block (*i.e.*, $L = Z$).

To merge two arrays A and B of sizes l_A and l_B ($l_A + l_B = n$), conduct a dual binary search of the arrays to find the keys with ranks $\{n^{2/3}, 2n^{2/3}, 3n^{2/3}, \ldots\}$ among the set of keys from both arrays. Each dual binary search takes $O(\log n)$ work and depth and incurs $O(\log\lceil n/L \rceil)$ cache misses. Once the locations of pivots have been identified, the subarrays which are of size $n^{2/3}$ each can be recursively merged and appended. When $Z = \Omega(L^2)$, this algorithm can be shown to have $O(\lceil n/L \rceil)$ cache complexity, and it has $O(\log n)$ depth (see the full version of the paper [6]). Using this merge in a mergesort in which the two recursive calls are parallel gives an algorithm with depth $O(\log^2 n)$ and cache complexity $O((n/L)\log(n/Z))$, which is not optimal. Blelloch *et al.* [4] analyze similar merge and mergesort algorithms with the same cache complexities but with larger depth.

Our parallel sorting algorithm is based on a version of sample sort [11, 15], and has optimal cache complexity. Sample sorts use a sample to select a set of pivots that partition the keys into buckets, then route all the keys to their appropriate buckets, and then to sort within the buckets. Kumar [14] presents a version of sample sort in which this key distribution is done sequentially (without cost analysis). Our algorithm is also similar to that of [2].

The algorithm (Algorithm 1) first splits the set of elements into $\sqrt{n}$ subarrays of size $\sqrt{n}$ and recursively sorts each of the subarrays. Then, we deterministically choose samples to determine pivots: we choose every $(\log n)$-th element from each of the subarrays as a sample. The sample set, which is smaller than the given data set by a factor of $\log n$, is then sorted using mergesort. Because mergesort is reasonably cache-efficient, using it on a set slightly smaller than the input set is not too costly in terms of cache complexity. More precisely, this mergesort does not incur more than $O(\lceil n/L \rceil)$ cache misses. We can then pick $\sqrt{n}$ evenly spaced keys from the sample set as pivots to determine bucket boundaries. To determine the bucket boundaries, the pivots are used to split each subarray using the cache-oblivious merge procedure. This procedure also takes no more than $O(\lceil n/L \rceil)$ cache misses.

Once the subarrays have been split, parallel prefix and matrix transpose operations can be used to determine the precise location in the buckets where each segment of the subarray is to be sent.

Algorithm 1 COSORT(A, n)

1: **if** $n < 10$ **then**
2: **return** Sort A sequentially
3: **end if**
4: $h \leftarrow \lceil \sqrt{n} \rceil$
5: $\forall i \in [1:h]$, Let $A_i \leftarrow A[h(i-1)+1 : hi]$
6: $\forall i \in [1:h]$, $S_i \leftarrow$ COSORT(A_i, h)
7: $X \leftarrow$ Pick every $(\log n)$-th element of each of the A_is
8: $Y \leftarrow$ MERGESORT(X)
9: $Z \leftarrow$ Pick every $(\sqrt{n}/\log n)$-th element of Y
10: $\forall i \in [1:h]$, $M_i \leftarrow$ SPLIT(S_i, Z)
{Each array M_i contains for each bucket j a start location in S_i for bucket j and a length of how many entries are in that bucket, possibly 0.}
11: Let L be the $h \times h$ matrix formed by rows M_i with just the lengths.
12: $L^T \leftarrow$ TRANSPOSE(L)
13: $\forall i \in [1:h]$, $O_i \leftarrow$ PREFIX-SUM(L_i^T)
14: $O^T \leftarrow$ TRANSPOSE(O)
15: $\forall i, j \in [1:n]$, $T_{i,j} \leftarrow \langle M_{i,j}\langle 1\rangle, O^T_{i,j}, M_{i,j}\langle 2\rangle\rangle$
{Each triple corresponds to an offset in row i for bucket j, an offset in bucket j for row i and the length to copy.}
16: Let $B_1, B_2, \ldots, B_h$ be arrays (buckets) of size $2h$ each
17: B-TRANSPOSE(S, B, T, 1, 1, h)
18: $\forall i$, $SB_i \leftarrow$ COSORT(B_i, length(B_i))
19: **return** $SB_1 || SB_2 || \ldots || SB_h$

This mapping information is stored in a matrix T of size $\sqrt{n} \times \sqrt{n}$. Note that none of the buckets will be loaded with more than $2\sqrt{n}\log n$ keys because of the way we select pivots.

Algorithm 2 B-TRANSPOSE(S, B, T, i_s, i_b, n)

Copy from arrays $S_{i \in [i_s : i_s+n)}$ to buckets $B_{j \in [i_b : i_b+n)}$ using map $T_{i \in [i_s : i_s+n), j \in [i_b : i_b+n)}$.
1: **if** $(n = 1)$ **then**
2: 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 $B_{b_s}[T_{i_s,i_b}\langle 2\rangle : T_{i_s,i_b}\langle 2\rangle + T_{i_s,i_b}\langle 3\rangle)$
3: **else**
4: B-TRANSPOSE(S, B, T, i_s, i_b, $n/2$)
5: B-TRANSPOSE(S, B, T, i_s, $i_b + n/2$, $n/2$)
6: B-TRANSPOSE(S, B, T, $i_s + n/2$, i_b, $n/2$)
7: B-TRANSPOSE(S, B, T, $i_s + n/2$, $i_b + n/2$, $n/2$)
8: **end if**

Once the bucket boundaries have been determined, the keys need to be transferred to the buckets. Although a naive algorithm to do this is not cache-efficient, we show that the bucket transpose algorithm (Algorithm 2) algorithm is. The bucket transpose is a four way divide-and-conquer procedure on the (almost) square matrix T which indicates a set of segments of subarrays (segments are contiguous in each subarray) and their target locations in the bucket. The matrix T is cut in half vertically and horizontally and separate recursive calls are assigned the responsibility of transferring the keys specified in each of the four parts.

LEMMA 2.1. *The algorithm B-TRANSPOSE transfers a matrix of $\sqrt{n} \times \sqrt{n}$ keys into bucket matrix B according to offset T in $O(n)$ work, $O(\log n)$ depth, and $O(\lceil n/L \rceil)$ sequential cache complexity.*

PROOF. (*outline*): For each node n in the recursion tree of bucket transpose, we define the node's size $s(n)$ to be the size of the matrix T and the node's weight $w(n)$ to be the number of keys that T

is responsible for transferring. We identify three classes of nodes in the recursion tree:

1. Light nodes: A node n is light if $s(n) < Z/100$, and $w(n) < Z/10$, and its parent node is of size greater than $Z/100$.
2. Heavy leaves: A leaf n is heavy if $w(n) \geq Z/10$.
3. Heavy nodes: An interior node n is heavy if $s(n) < Z/100$, $w(n) < Z/10$, and its parent node is of weight larger than $Z/10$.

The union of these three sets covers the responsibility for transferring all the keys.

From the definition of a light node, it can be argued that all the keys that a light node is responsible for fit inside a cache, implying that each light node cannot incur more than Z/L cache misses. It can also be seen that light nodes can not be greater than $4n/(Z/100)$ in number leading to the fact that the sum of cache complexities of all the light nodes is no more than $O(\lceil n/L \rceil)$.

Heavy nodes are similar to light nodes in that their target data fits into a cache. If we assume that they have combined weight of $n-W$, then there no more than $4(n-W)/(Z/10)$ of them, putting their aggregate cache complexity at $40(n-W)/L$.

A heavy leaf of size w incurs $\lceil w/L \rceil$ cache misses. There are no more than $W/(Z/10)$ of them, implying that their aggregate cache complexity is $W/L + 10W/Z$. Therefore, the cache complexities of heavy nodes and leaves adds up to another $O(\lceil n/L \rceil)$. □

THEOREM 2.2. *On an input of size n, the algorithm COSORT incurs $O(n \log n)$ work and $Q(n; Z, L) = O(\lceil n/L \rceil \lceil \log_Z n \rceil)$ sequential cache complexity, and has $O(\log^2 n)$ depth.*

PROOF. All the subroutines other than recursive calls to COSORT have linear work and cache complexity $O(\lceil n/L \rceil)$. Also, the subroutine with the maximum depth is the mergesort used to find pivots; its depth is $O(\log^2 n)$. Therefore, the recurrence relations for the work, depth, and cache complexity are as follows:

$$\begin{aligned} W(n) &= O(n) + \sqrt{n}W(\sqrt{n}) + \sum_{i=1}^{\sqrt{n}} W(n_i) \\ D(n) &= O(\log^2 n) + \max_{i=1}^{\sqrt{n}}\{D(n_i)\} \\ Q(n; Z, L) &= O\left(\left\lceil \frac{n}{L} \right\rceil\right) + \sqrt{n}Q(\sqrt{n}; Z, L) + \sum_{i=1}^{\sqrt{n}} Q(n_i; Z, L), \end{aligned}$$

where the n_is are such that their sum is n and none individually exceed $2\sqrt{n}\log n$. The base case for the recursion for cache complexity is $Q(n; Z, L) = O(\lceil n/L \rceil)$ for $n < cZ$ for some constant c. Solving these recurrences proves the theorem. □

The depth of the above algorithm can be slightly improved by using a randomized method to select the pivots: we pick $\sqrt{n}$ random keys from A, and use a brute force sort to determine the relative order of the sample set. Because this procedure does not always yield balanced buckets, we need to repeat pivot selection until all buckets are of size less than $\sqrt{n}\log n$. It can be argued that each iteration of the loop exits successfully with probability at least $1-1/n$. This version of the algorithm can be shown to have a depth of $O(\log^{3/2} n)$ with high probability (see [6]).

3. MULTI-LEVEL HIERARCHIES

We highlight one of our results for mapping low-depth cache-oblivious algorithms to shared-memory parallel machines with *multi-level* caches. Consider the *Parallel Tree-of-Caches (PToC)* family of parallel cache hierarchies, with *intra-level regularity*. Each of P processors is connected to a private level-one cache of size Z_1. Disjoint, equal-sized groups of P_2 processors each share a level-two cache of size $Z_2 \cdot P_2$, and so on, forming a tree of caches of k levels. Cache lines (or blocks) are of size L_i at level i in the hierarchy, and $\forall i, Z_i \geq L_i^2$. The cache hierarchy is inclusive: each cached word at level $i < k$ is also cached in its "parent" cache at level $i+1$. Moreover, each cache is fully associative and supports a variant of the *dag consistency* cache consistency model [7] that uses an optimal replacement policy. Theorem 3.1 generalizes the prior bound on the cache complexity for single level private caches (recall Section 1).

THEOREM 3.1. *When a computation with sequential cache complexity $Q_1(Z, L)$, work W, and depth D is scheduled on a P-processor PToC with intra-level regularity using work stealing, all the caches at level i incur a total of $Q_1(Z_i, L_i) + O(Z_i PD/L_i)$ cache misses with high probability.*

See [6] for other cache configurations and further details, including bounds for PDF schedulers.

Acknowledgments. This work was funded in part by IBM, Intel, and the Microsoft-sponsored Center for Computational Thinking.

4. REFERENCES

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory Comput. Syst.*, 35(3), Springer, 2002.
[2] A. Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. In *IEEE FOCS'87*, 1987.
[3] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *ACM SPAA'08*, 2008.
[4] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *ACM-SIAM SODA'08*, 2008.
[5] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *ACM SPAA'04*, 2004.
[6] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. Tech. Rep. CMU-CS-09-134, Carnegie Mellon University, 2009. http://reports-archive.adm.cs.cmu.edu /anon/2009/CMU-CS-09-134.pdf.
[7] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In *IEEE IPPS'96*, 1996.
[8] R. A. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. In *ACM SPAA'07*, 2007.
[9] R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *ACM SPAA'08*, 2008.
[10] E. D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*, LNCS. Springer-Verlag, 2002.
[11] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3), 1970.
[12] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *IEEE FOCS'99*, 1999.
[13] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *ACM SPAA'06*, 2006.
[14] P. Kumar. Cache oblivious algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*. Springer, 2003.
[15] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3), 1989.