Low Depth Cache-Oblivious Algorithms

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

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

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

ABSTRACT

In this paper we explore a simple and general approach for developing parallel algorithms that lead to good cache complexity on parallel machines with private or shared caches. The approach is to design nested-parallel algorithms that have low depth (span, critical path length) and for which the natural sequential evaluation order has low cache complexity in the cache-oblivious model. We describe several cache-oblivious algorithms with optimal work, polylogarithmic depth, and sequential cache complexities that match the best sequential algorithms, including the first such algorithms for sorting and for sparse-matrix vector multiply on matrices with good vertex separators.

Using known mappings, our results lead to low cache complexities on shared-memory multiprocessors with a single level of private caches or a single shared cache. We generalize these mappings to multi-level cache hierarchies of private or shared caches, implying that our algorithms also have low cache complexities on such hierarchies. The key factor in obtaining these low parallel cache complexities is the low depth of the algorithms we propose.

Categories and Subject Descriptors

F.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Algorithms, Theory

Keywords

Cache-oblivious algorithms, sorting, sparse-matrix vector multiply, graph algorithms, parallel algorithms, multiprocessors, schedulers.

1. INTRODUCTION

Due to the physical realities of building machines it seems likely that locality will always play a role in designing efficient algorithms for parallel machines. Indeed many parallel models have been designed to take account of locality on both shared [4, 48, 7] and distributed memory machines [48, 33, 12]. These models, however, assume a fixed number of processors for which the algorithm designer or programmer have to map their algorithms onto. What seems to be emerging instead as the dominant programming paradigm for shared memory parallel machines is one based on dynamic parallelism. In such models the programmer expresses the full parallelism without concern of how it maps onto processors. The runtime system then supplies a scheduler that maps this dynamic parallelism onto the processors of the machine. A common form of programming in this model is based on nested parallelism—consisting of nested parallel loops and/or fork-join constructs [13, 26, 20, 35, 44]. If locality is not of concern, performance costs in such models can be calculated in terms of *work* (number of operations) and *depth* (also known as the span or the critical path length) and can be mapped onto runtime on a fixed number of processors. This can greatly simplify how programmers think about parallelism. It is not clear, however, how to capture locality in these models in a high-level way.

In this paper we are interested in analyzing the locality of algorithms written with dynamic nested parallelism. We consider a paradigm based on analyzing the cost of an algorithm using in addition to work and depth the cache complexity in the sequential cache-oblivious model. The *cache-oblivious model* (*ideal-cache model*) [38] is a two-level model of computation comprised of an unbounded memory and a cache of size M. Data are transferred between the two levels using cache lines of size B; all computation occurs on data in the cache. Both M and B are unknown to the algorithm, and the goal is to minimize an algorithm's *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 (single processor) multi-level cache hierarchy, regardless of the values of M_i and B_i at each level i [38]. Researchers have developed cache-oblivious algorithms for a many problems [6, 22, 34].

The cache complexity $Q(n; M, B)$ for a natural sequential execution of a parallel program (on input of size n) can be used to bound the cache complexity $Q_p(n; M, B)$ for the same program on certain p-processor parallel machines with a single level of cache(s) [1, 15]. In particular, for a shared-memory parallel machine with private caches (each processor has its own cache) using a work-stealing scheduler, $Q_p(n; M, B) < Q(n; M, B) + O(pMD/B)$ with probability $1-\delta$ [1],[1] and for a shared cache using a parallel-depth-first (PDF) scheduler, $Q_p(n; M+pBD, B) \leq Q(n; M, B)$ [15], 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. (When viewed as a computation dag where the nodes are constant-work tasks and the edges are dependences between tasks, the dags for such compu-

[1]In this paper, δ is an arbitrarily small positive constant.

tations are series-parallel.) The "natural" sequential execution is simply one that runs each call in a fork to completion before starting the next.

These results for a single level of cache(s) 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 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 *multi-level* private or shared caches. For example, we show that for a work-stealing scheduler on a multi-level private cache hierarchy $Q_p(n; M_i, B_i) < Q(n; M_i, B_i) + O(pM_iD/B_i)$ with probability $1-\delta$ for each level i, and that this bound is tight.

As an example of the approach consider Strassen's matrix multiply. It is nested-parallel because 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 [38], the sequential cache complexity is $Q(n; M, B) = O(n^{\lg 7}/(B\sqrt{M}))$. Thus, we have that $Q_p(n; M, B) < Q(n; M, B) + O(pM\log^2(n)/B)$ for a single level of private caches and $Q_p(n; M + pB\log^2 n, B) \leq Q(n; M, B) = O(n^{\lg 7}/(B\sqrt{M}))$ for a shared cache. For practical parameters these bounds indicate either only marginally more total misses than the sequential version for private caches or only a marginally larger cache size for shared caches. Similarly good bounds are obtained for multi-level hierarchies of private or shared caches, using our results for such hierarchies.

Although matrix multiply and some other known cache-oblivious algorithms are naturally parallel with low depth (*e.g.*, matrix transpose and FFT [38]), many are not. Importantly, prior cache-oblivious sorting algorithms with optimal sequential cache complexity [23, 24, 25, 36, 38] are not parallel. This paper presents the first low (*i.e.*, polylogarithmic) depth cache-oblivious sorting algorithm with optimal cache complexity. Under the standard "tall cache" assumption $M = \Omega(B^2)$ [38], our (deterministic) sorting algorithm has cache complexity $Q(n; M, B) = O(\frac{n}{B}\log_M n)$ and work $W = O(n\log n)$, which are optimal, and depth $D = O(\log^2 n)$. We improve the depth for a randomized version. In contrast, parallelizing the prior algorithms using known techniques would result in depth at least $\Omega(\sqrt{n})$. We illustrate how our sorting algorithm can be used to construct the first polylogarithmic depth, cache-oblivious, optimal cache complexity algorithms for other important problems such as list ranking and tree contraction. Finally, we present the first cache-oblivious, low cache complexity algorithm for sparse-matrix vector (SpMV) multiply on matrices with good vertex separators (roughly speaking a sparse matrix has good separators if the corresponding graph can be partitioned by removing a reasonably small set of vertices and their incident edges so no partition is too large). All planar graphs have good separators. The SpMV algorithm is optimal work, $O(\log^2 n)$ depth, and its sequential cache complexity improves upon the previous best sequential algorithm and is optimal for planar graphs.

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.* [39] for a class of algorithms with a regularity condition, by Blelloch *et al.* [14] for a class of binary divide-and-conquer algorithms, and by Chowdhury and Ramachandran [28, 29] for a class of dynamic programming and Gaussian elimination-style problems. Recent work by Chowdhury et. al. [31] has studied cache oblivious algorithms for a parallel model with a tree of caches. Our design motive is to have a generic approach that enables one to analyze an algorithm independently of the model in a simple way and then map onto different machines; we study SpMV-multiply, sorting and related algorithms as specific instances of our general approach. Our work may also be contrasted with that of [7], which presents cache-efficient algorithms for private caches but the algorithms are not cache oblivious and are based on a fixed number of processors p.

A preliminary version of this paper appeared as a three page brief announcement in SPAA'09 [17].

Algorithm 1 MERGE$((A, s_A, l_A), (B, s_B, l_B), (C, s_C))$

```
Merges $A[s_A : s_A + l_A)$ and $B[s_B : s_B + l_B)$
into array $C[s_C : s_C + l_A + l_B)$
1: if $l_B = 0$ then
2:     Copy $A[s_A : s_A + l_A)$ to $C[s_C : s_C + l_A)$
3: else if $l_A = 0$ then
4:     Copy $B[s_B : s_B + l_B)$ to $C[s_C : s_C + l_B)$
5: else
6:     $\forall k \in [1 : \lfloor n^{1/3} \rfloor]$, find pivots $(a_k, b_k)$ such that $a_k + b_k = k\lceil n^{2/3} \rceil$ and $A[s_A + a_k] \leq B[s_B + b_k + 1]$ and $B[s_B + b_k] \leq A[s_A + a_k + 1]$.
7:     $\forall k \in [1 : \lfloor n^{1/3} \rfloor]$, MERGE$((A, s_A + a_k, a_{k+1} - a_k), (B, s_B + b_k, b_{k+1} - b_k), (C, s_C + a_k + b_k))$
8: end if
```

2. SORTING

In this section, we present the first cache-oblivious sorting algorithm that achieves optimal work, polylogarithmic depth, and good sequential cache complexity. Prior cache-oblivious algorithms with optimal cache complexity [23, 24, 25, 36, 38] have $\Omega(\sqrt{n})$ depth.

2.1 Algorithm Preliminaries

Our sorting algorithm uses known algorithms for matrix transpose, prefix sum and merging as subroutines. We first describe the exact variants of these algorithms that the sorting algorithm uses. The costs are summarized in Figure 1. The standard divide-and-conquer matrix-transpose algorithm [38] is work optimal, has logarithmic depth and has optimal cache complexity when $M = \Omega(B^2)$. A simple variant of the tree-based parallel prefix-sums algorithm has logarithmic depth and cache complexity $O(n/B)$. As usual the algorithm works in two phases generating partial sums in a tree in one phase and propagating results down in the next. Each phase is implemented using divide-and-conquer over the tree. For cache efficiency, the tree of partial sums is laid out in the infix order. This gives an algorithm that runs with cache complexity $O(n/b)$ and depth $O(\log n)$ even if the cache only has a single cache line.

Algorithm 1 merges two arrays A and B of sizes l_A and l_B ($l_A + l_B = n$). The pivots ranked $\{n^{2/3}, 2n^{2/3}, \ldots\}$ can be found using a dual binary search on the arrays. This takes $O(n^{1/3} \cdot \log n)$ work, $O(\log n)$ depth and at most $O(n^{1/3}\log(n/B))$ cache misses. Once the locations of pivots have been identified, the subarrays which are of output size $n^{2/3}$ each can be recursively merged and appended. The recursive relation for the cache complexity is

$$Q(n; M, B) \leq \begin{cases} k_1 n^{1/3}(\log(n/B) + Q(n^{2/3}; M, B)) & n > cM \\ k_2 n/B + 1 & \text{otherwise} \end{cases}$$

for some positive constants c, k_1 and k_2. When $n > cM$, this

Problem	Depth	Cache Complexity	Section
Matrix Transpose ($n \times m$ matrix)	$O(\log(n+m))$	$O(\lceil nm/B \rceil)$	[38]
Prefix Sums	$O(\log n)$	$O(\lceil n/B \rceil)$	2.1
Merge	$O(\log n)$	$O(\lceil n/B \rceil)$	2.1
Sort (deterministic)*	$O(\log^2 n)$	$O(\lceil n/B \rceil \lceil \log_M n \rceil)$	2.2
Sort (randomized; bounds are w.h.p.)*	$O(\log^{1.5} n)$	$O(\lceil n/B \rceil \lceil \log_M n \rceil)$	2.3
Sparse-Matrix Vector Multiply (m nonzeros, n^ϵ separators)*	$O(\log^2 n)$	$O(\lceil m/B + n/M^{1-\epsilon} \rceil)$	4

Figure 1: Low-depth cache-oblivious algorithms. New algorithms are marked (*). All algorithms are work optimal and their cache complexities match the best sequential algorithms. The bounds assume $M = \Omega(B^2)$.

recursion satisfies

$$Q(n; M, B) = O(n/B + n^{1/3} \log(n/B)).$$

If $M = \Omega(B^2)$ and $n > cM$, then the first term in the expression for cache complexity $O(n/B)$ is asymptotically larger than $n^{1/3} \log(n/B)$, making the second term redundant. Therefore, in all cases, $Q(n; M, B) = O(\lceil n/B \rceil)$. The recurrence relation for depth is:

$$D(n) \leq \log n + D(n^{2/3}),$$

which solves to $D(n) = O(\log n)$. It is easy to see that the work involved is linear.

Using this merge algorithm in a mergesort in which the two recursive calls are parallel gives an algorithm with depth $O(\log^2 n)$ and cache complexity $O((n/B) \log_2(n/M))$, which is not optimal. Blelloch *et al.* [14] analyze similar merge and mergesort algorithms with the same (suboptimal) cache complexities but with larger depth.

2.2 Deterministic Sorting

Our parallel sorting algorithm is based on a version of sample sort [37, 45], and has optimal cache complexity. Sample sorts first 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 finally sort within the buckets. Compared to prior cache-friendly (sequential) sample sort algorithms [2, 41], which with slight modification can be improved to $\Omega(\sqrt{n})$ depth, our cache-oblivious algorithm uses (and analyzes) a new parallel bucket-transpose algorithm for the key distribution phase, in order to achieve $O(\log^2 n)$ depth.

The algorithm (Algorithm COSORT in Figure 2) first splits the set of elements into $\sqrt{n}$ subarrays of size $\sqrt{n}$ and recursively sorts each of the subarrays. Then, samples are chosen to determine pivots. This step can be done either deterministically or randomly. We first describe a deterministic version of the algorithm for which the **repeat** and **until** statements are not needed; Section 2.3 will describe a randomized version that uses these statements. For the deterministic version, 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 the mergesort algorithm outlined above. 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 $O(\lceil n/B \rceil)$ cache misses. We can then pick $\sqrt{n}$ evenly spaced keys from the sample set $\mathcal{P}$ 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 $O(\lceil n/B \rceil)$ cache misses.

Once the subarrays have been split, prefix sums 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. 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.

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 B-TRANSPOSE in Figure 2) 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. Note that ordinary matrix transpose is the special case of $T_{i,j} = \langle j, i, 1 \rangle$ for all i, j.

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

PROOF. (sketch) For each node v in the recursion tree of bucket transpose, we define the node's size $s(v)$ to be n^2, the size of its submatrix T, and the node's weight $w(v)$ to be the number of keys that T is responsible for transferring. We identify three classes of nodes in the recursion tree:

1. Light-1 nodes: A node v is light-1 if $s(v) < M/100$, $w(v) < M/10$, and its parent node is of size $\geq M/100$.
2. Light-2 nodes: A node v is light-2 if $s(v) < M/100$, $w(v) < M/10$, and its parent node is of weight $\geq M/10$.
3. Heavy leaves: A leaf v is heavy if $w(v) \geq M/10$.

The union of these three sets covers the responsibility for transferring all the keys, *i.e.*, all leaves are accounted for in the subtrees of these nodes.

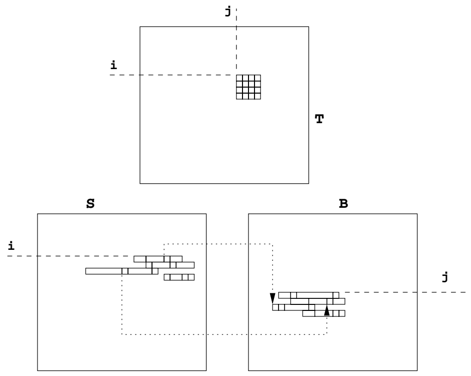
From the definition of a light-1 node, it can be argued that all the keys that a light-1 node is responsible for fit inside a cache, implying that the subtree rooted at a light-1 node cannot incur more than M/B cache misses. It can also be seen that light-1 nodes can not be greater than $4n/(M/100)$ in number leading to the fact that the sum of cache complexities of all the light-1 nodes is no more than $O(\lceil n/B \rceil)$.

Light-2 nodes are similar to light-1 nodes in that their target data fits into a cache of size M. If we assume that they have combined weight of $n - W$, then there are no more than $4(n-W)/(M/10)$ of them, putting the aggregate cache complexity for their subtrees at $40(n-W)/B$.

A heavy leaf of weight w incurs $\lceil w/B \rceil$ cache misses. There are no more than $W/(M/10)$ of them, implying that their aggregate cache complexity is $W/B + 10W/M < 11W/B$. Therefore, the cache complexities of light-2 nodes and heavy leaves adds up to

```
Algorithm COSORT($A, n$)
if $n < 10$ then
    return Sort $A$ sequentially
end if
$h \leftarrow \lceil\sqrt{n}\rceil$
$\forall i \in [1:h]$, Let $A_i \leftarrow A[h(i-1)+1 : hi]$
$\forall i \in [1:h]$, $S_i \leftarrow$ COSORT($A_i, h$)
repeat
    Pick an appropriate sorted pivot set $\mathcal{P}$ of size $h$
    $\forall i \in [1:h]$, $M_i \leftarrow$ SPLIT($S_i, \mathcal{P}$)
    {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.}
    $L \leftarrow h \times h$ matrix formed by rows $M_i$ with just the lengths
    $L^T \leftarrow$ TRANSPOSE($L$)
    $\forall i \in [1:h]$, $O_i \leftarrow$ PREFIX-SUM($L_i^T$)
    $O^T \leftarrow$ TRANSPOSE($O$)   {$O_i$ is the $i$th row of $O$}
    $\forall i, j \in [1:h]$, $T_{i,j} \leftarrow \langle M_{i,j}\langle 1\rangle, O^T_{i,j}, M_{i,j}\langle 2\rangle\rangle$
    {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.}
until No bucket is too big
Let $B_1, B_2, \ldots, B_h$ be arrays (buckets) of sizes dictated by $T$
B-TRANSPOSE($S, B, T, 1, 1, h$)
$\forall i$, $B_i' \leftarrow$ COSORT($B_i$, length($B_i$))
return $B_1' || B_2' || \ldots || B_h'$
```

```
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
```



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.

Figure 2: Cache-Oblivious Sorting and Bucket-Transpose Algorithms

another $O(\lceil n/B\rceil)$. We also note that the validity of this proof does not depend on the size of the individual buckets. The statement of the lemma holds even for the case where each of the buckets is as large as $O(\sqrt{n}\log n)$. $\square$

THEOREM 2.2. *On an input of size n, the deterministic COSORT has $Q(n; M, B) = O(\lceil n/B\rceil\lceil\log_M n\rceil)$ sequential cache complexity, $O(n\log n)$ work, and $O(\log^2 n)$ depth.*

PROOF. All the subroutines other than recursive calls to COSORT have linear work and cache complexity $O(\lceil n/B\rceil)$. 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:

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

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

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

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 $Q(n; M, B) = O(\lceil n/B\rceil)$ for $n < cM$ for some constant c. Solving these recurrences proves the theorem. $\square$

2.3 Randomized Sorting

A simple randomized version of the sorting algorithm is to randomly pick $\sqrt{n}$ elements for pivots, sort them using brute force (compare every pair) and using the sorted set as the pivot set $\mathcal{P}$. This step takes $O(n)$ work, $O(\log n)$ depth (let c_n be a constant such that depth is at most $c_n\log(n)$) and has cache complexity $O(n/B)$ and the probability that the largest of the resultant buckets are larger than $3\sqrt{n}\log n$ is not more $1 - 1/n$. When one of the buckets is too large ($> 3\sqrt{n}\log n$), the process of selecting pivots and recomputing bucket boundaries is repeated. Because the probability of this happening repeatedly is low, the overall depth of the algorithm is small. Further, the recursion is stopped when the problem size reduces to 2^{40}.

THEOREM 2.3. *On an input of size n, the randomized version of COSORT has, with probability greater than $1-1/n$, $Q(n; M, B) = O(\lceil n/B\rceil\lceil\log_M n\rceil)$ sequential cache complexity, $O(n\log n)$ work, and $O(\log^{1.5} n)$ depth.*

PROOF. (sketch) In a call to randomized COSORT with input size n, the loop terminates with probability $1 - 1/n$ in each round and takes less than 2 iterations on average to terminate. Each iteration of the while loop, including the brute force sort requires $O(n)$ work and incurs at most $O(\lceil n/B\rceil)$ cache misses with high probability. Therefore,

$$\mathbb{E}[W(n)] = O(n) + \sqrt{n}\,\mathbb{E}[W(\sqrt{n})] + \sum_{i=1}^{\sqrt{n}} \mathbb{E}[W(n_i)],$$

where each $n_i < 3\sqrt{n}\log n$ and $\sum_{i=1}^{\sqrt{n}} n_i = n$. This implies that $\mathbb{E}[W(n)] = O(n\log n)$. Similarly for cache complexity we have

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

which implies $\mathbb{E}[Q(n; M, B)] = O\left(\frac{n}{B}\log_{\sqrt{M}} n\right) = O\left(\frac{n}{B}\log_M n\right)$. To show the high probability bounds for work and cache complexity, we can use Chernoff bounds since the fan out at each level of recursion is high.

To analyze the depth of the dag, we obtain high probability bounds on the depth of each level of recursion tree (we assume that the levels are numbered starting with the root at level 0). To get sufficient confidence bounds at each level we might have to execute the outer loop more times toward the leaves where the problem size is small. Each iteration of the outer loop at node N of input size m at level k in the recursion tree has depth $\log m$ and the termination probability of the loop is $1 - 1/m$.

We first show probability bounds on the depth of a maximal path in the computation dag. We represent the path as a recursion tree and show that the sum of depths of all nodes at level d in the recursion tree is at most $c_d \log^{3/2} n / \log_2 \log_2 n$ with probability at least $1 - 1/n^{(\log_2 \log_2 n)^2}$ (for some constant c_d to be defined shortly) and that the recursion tree is at most $1.5 \log_2 \log_2 n$ levels deep. This will prove that the depth of the recursion tree (*i.e.*the path) is $1.5 c_d \log^{3/2} n$ with probability at least

$$1 - (1.5 \log_2 \log_2 n)/n^{\log_2^2 \log_2 n}.$$

Since each of the paths is a candidate for the critical path, the actual depth is a maximum over all such paths. We will argue that there are not more than $C(n) = n^{1.5 \log_2 \log_2 n}$ such paths. Then, by the union bound, it follows that the probability that any single path is longer than $1.5 c_d \log^{3/2} n$ is less than

$$1.5 \log \log n / n^{(\log_2 \log_2 n)^2 - 1.5 \log_2 \log_2 n}$$

(high probability bound).

The maximum number of levels in the recursion tree can be bounded using the recurrence relation $X(n) \geq 1 + X(3\sqrt{n} \log n)$ and $X(2^{40}) = 1$. Using induction, it is straightforward to show that this solves to $X(n) < 1.5 \log_2 \log_2 n$. Similarly the number of paths $C(n)$ can be bounded using the relation $C(n) > (\sqrt{n} C(\sqrt{n}))(\sqrt{n} C(2\sqrt{n} \log n))$. Again, using induction, this relation can be used to show that $C(n) = n^{1.5 \log_2 \log_2 n}$.

To compute the sum of the depth of nodes at level d in the recursion tree, we consider two cases: (1) when $d > 80 \log_2 \log_2 \log_2 n$ and (2) otherwise.

Case 1: The size of a node one level deep in the recursion tree is at most $3\sqrt{n} \log n \leq n^{1/2+r}$ for $r = 1/6$. Also, the size of a node which is d levels deep is at most $n^{(1/2+r)^d}$, each costing $c_n (1/2 + r)^d \log n$ depth per trial. Since there are at most 2^d nodes at level d in the recursion tree, and the failure probability of a loop in any node is no more than $1/2$, we show that the probability of having to execute more than $(2^d \cdot \log^{1/2} n)/((1 + 2r)^d \cdot \log_2 \log_2 n)$ loops is small. Since we are estimating the sum of 2^d independent variables, we use Chernoff bounds of the form:

$$Pr[X > (1 + \delta)\mu] \leq e^{-\delta^2 \mu}, \qquad (1)$$

with $\mu = (2 \cdot 2^d)$, $\delta = (1/2)(\log^{1/2} n/((1+2r)^d \cdot \log_2 \log_2 n)) - 1$. The resulting probability bound is less than $1/n^{\log_2^2 \log_2 n}$ for $d > 80 \log_2 \log_2 \log_2 n$. Therefore, the contribution of nodes at level d in the recursion tree to the depth of recursion tree is at most $2^d \cdot (1/2 + r)^d c_n \log n \cdot \log^{1/2} n/((1 + 2r)^d \cdot \log_2 \log_2 n) = c_n \log^{3/2} n / \log_2 \log_2 n$ with probability at least $1 - 1/n^{\log_2^2 \log_2 n}$.

Case 2: We classify all nodes at level d in to two kinds, the large ones with size greater than $\log^2 n$ and the smaller ones with size at most $\log^2 n$. The total number of nodes is at most $2^d < (\log_2 \log_2 n)^{80}$. Consider the small nodes. Each small node can contribute a depth of at most $2c_n \log_2 \log_2 n$ to the recursion tree and there are at most $(\log_2 \log_2 n)^{80}$ of them. If we set c_d to be the minimum number such that $c_d \log^{3/2} n > c_n \log_2 \log_2^{82} n$, then the contribution of small nodes to depth of the recursion tree at level d is lesser than $c_d \log^{3/2} n / \log_2 \log_2 n$.

We use Chernoff bounds to bound the contribution of large nodes to the depth of the recursion tree. Suppose that there are j large nodes. We show that with probability not more than $1/n^{\log^2 \log n}$, it takes more than $10 \cdot j$ loop iterations at depth d for j of them to succeed. For this, consider $10 \cdot j$ random independent trials with success probability at least $1 - 1/\log^2 n$ each. The expected number of failures is no more than $\mu = 10 \cdot j / \log^2 n$. We want to show that the probability that there are greater than $9 \cdot j$ failures in this experiment is tiny. Using Chernoff bounds with the above μ, and $\delta = (0.9 \cdot \log^2 n - 1)$, we infer that this probability is less than $1/n^{\log_2^2 \log_2 n}$. Since $j < 2^d$, the depth contributed by the larger nodes is at most $2^d (1/2 + 1/6)^d \log n < c_d \log^{3/2} n / \log_2 \log_2 n$.

We have shown that each level in the recursion tree adds at most $c_d \log^{3/2} n / \log_2 \log_2 n$ depth to a path with probability at least $1 - 1/n^{\log_2^2 \log_n n}$. The proof follows from the union bound described earlier. □

3. APPLICATIONS OF SORTING: GRAPH ALGORITHMS

In this section, we make use of the fact that the PRAM algorithms for many problems can be decomposed into primitive operations such as scans and sorts. Our approach is similar to that of [7] except that we use the cache-oblivious model instead of the parallel external memory model. Arge *et al.* [5] demonstrate a cache-oblivious algorithm for list ranking using priority queues and use it to construct other graph algorithms. But their algorithms have $\Omega(n)$ depth because list ranking uses a serially-dependent sequence of priority queue operations to compute independent sets. Our parallel algorithms derived from sorting are the same as in [27] except that we use different algorithms for the primitive operations scan and sort, which suit our cache-oblivious framework. Moreover, a careful analysis (using standard techniques) is required to prove our sequential cache complexity and depth bounds under this framework.

List Ranking. A basic strategy for list ranking [40] is the following: (i) shrink the list to size $O(n/\log n)$, and (ii) apply pointer jumping on this shorter list. Stage (i) is achieved through finding independent sets in the list of size $\Theta(n)$ and removing them to yield a smaller problem. This can be done randomly using the random mate technique in which case, $O(\log \log n)$ rounds of such reduction would suffice. Alternately, we could use the deterministic technique described in section 3.1 of [40]: use two rounds of Cole and Vishkin's deterministic coin tossing [32] to find a $O(\log \log n)$-ruling set and then convert the ruling set to an independent set of size at least $n/3$ in $O(\log \log n)$ rounds. Arge *et al.* [8] show how this conversion can be made cache-efficient, and it is straightforward to change this algorithm to a cache-oblivious one. Stage (ii) uses $O(\log n)$ rounds of pointer jumping, each round essentially involving a sort operation to figure out the next level of pointer jumping. Thus, the cache complexity of this stage is asymptotically the same as sorting and its depth is $O(\log n)$ times the depth of sorting:

THEOREM 3.1. *The deterministic list ranking outlined above has $Q(n; M, B) = O(\lceil n/B \rceil \lceil \log_M n \rceil)$ sequential cache complexity, $O(n \log n)$ work, and $O(D_{sort}(n) \log n)$ depth.*

Graph Algorithms. We tabulate the complexity measures of basic graph algorithms on bounded degree graphs (Figure 3). The al-

Problem	Depth	Cache Complexity
List Ranking	$D_{LR}(n) = O(D_{sort}(n) \log n)$	$O(Q_{sort}(n))$
Euler Tour on Trees	$O(D_{LR}(n))$	$O(Q_{sort}(n))$
Tree Contraction	$O(D_{LR}(n) \log n)$	$O(Q_{sort}(n))$
Least Common Ancestors (k queries)	$O(D_{LR}(n))$	$O(\lceil k/n \rceil Q_{sort}(n))$
Connected Components	$O(D_{LR}(n) \log n)$	$O(Q_{sort}(\lvert E\rvert) \log(\lvert V\rvert/\sqrt{M}))$
Minimum Spanning Forest	$O(D_{LR}(n) \log n)$	$O(Q_{sort}(\lvert E\rvert) \log(\lvert V\rvert/\sqrt{M}))$

Figure 3: Low-depth cache-oblivious graph algorithms. All algorithms are deterministic. The bounds assume $M = \Omega(B^2)$. D_{sort} and Q_{sort} are the depth and cache complexity of cache-oblivious sorting.

gorithms that lead to these complexities are straightforward cache-oblivious adaptations of known PRAM algorithms (as described for the external memory model in [27]) using primitives from earlier sections. For instance, finding an Euler Tour involves scanning the input to compute the successor function for each edge and running a list ranking. Tree contraction involves constructing an Euler tour of the tree, finding an independent set on it and contracting the tour to recursively solve a smaller problem. Finding Least Common Ancestors of a set of vertex pairs in a tree involves computing the Euler tour and reducing the problem to a range minima query problem (which is solved with search trees). Deterministic algorithms for Connected Components and Minimum Spanning Forest are similar and use tree contraction as their basic idea; the cache bounds are slightly worse than those in [27]: $\log(|V|/\sqrt{M})$ versus $\log(|V|/M)$. While [27] uses knowledge of M to transition to a different approach once the *vertices* in the contracted graph fit within the cache, cache-obliviously we need for the *edges* to fit before we stop incurring misses.

4. SPARSE-MATRIX VECTOR MULTIPLY

We consider the problem of multiplying a sparse $n \times n$ matrix with m non-zeros by a dense vector. For general sparse matrices Bender *et al.* [10] show a lower bound on cache complexity, which for $m = O(n)$ matches the sorting lower bound. However, for certain matrices common in practice the cache complexity can be improved. For example, for matrices with non-zero structure corresponding to a well-shaped d-dimensional mesh, a multiply can be performed with cache complexity $O(m/B + n/M^{1/d})$ [11]. This requires pre-processing to lay out the matrix in the right form. However, for applications that run many multiplies over the same matrix, as with many iterative solvers, the cost of the pre-processing can be amortized. Note that for $M \geq B^d$ (*e.g.*, the tall-cache assumption in 2 dimension), the cache complexity reduces to $O(m/B)$ which is asymptotically the same as scanning memory.

The cache-efficient layout and bounds for well-shaped meshes can easily be extended to graphs with good edge separators [14]. The layout and algorithm, however, is not efficient for graphs such as planar graphs or graphs with constant genus that have good vertex separators but not necessarily any good edge separators. In this paper we generalize the results to graphs with good vertex separators and present the first cache-oblivious, low cache complexity algorithm for the sparse-matrix multiplication problem on such graphs. We do not analyze the cost of finding the layout, which involves the recursive application of finding vertex separators, as it can be amortized across many solver iterations. Our algorithm for matrices with n^ϵ separators has linear work, $O(\log^2 n)$ depth, and $O(m/B + n/M^{1-\epsilon})$ sequential cache complexity.

Let S be a class of graphs that is closed under the subgraph relation. We say that S satisfies a $f(n)$-*vertex separator theorem* if there are constants $\alpha < 1$ and $\beta > 0$ such that every graph

```
Algorithm BuildTree($V, E$)
if $|E| = 1$ then
  return $V$
end if
$(V_a, V_{sep}, V_b) \leftarrow$ FindSeparator$(V, E)$
$E_a \leftarrow \{(u, v) \in E | u \in V_a \vee v \in V_a\}$
$E_b \leftarrow E - E_a$
$V_{a,sep} \leftarrow V_a \cup V_{sep}$
$V_{b,sep} \leftarrow V_b \cup V_{sep}$
$T_a \leftarrow$ BuildTree$(V_{a,sep}, E_a)$
$T_b \leftarrow$ BuildTree$(V_{b,sep}, E_b)$
return SeparatorTree$(T_a, V_{sep}, T_b)$
```

```
Algorithm SparseMxV($x$,$T$)
if isLeaf($T$) then
  $T.u$.value $\leftarrow x[T.v.\text{index}] \otimes T.w_{vu}$
  $T.v$.value $\leftarrow x[T.u.\text{index}] \otimes T.w_{uv}$
  {Two statements for the two edge directions}
else
  SparseMxV($T$.left) and SparseMxV($T$.right)
  for all $v \in T$.vertices do
    $v$.value $\leftarrow$ ($v$.left→value $\oplus$ $v$.right→value)
  end for
end if
```

Figure 4: Cache-Oblivious Algorithms for Building a Separator Tree and for Sparse-Matrix Vector Multiply

$G = (V, E)$ in S with n vertices can be partitioned into three sets of vertices V_a, V_s, V_b such that $|V_s| \leq \beta f(n)$, $|V_a|, |V_b| \leq \alpha n$, and $\{(u, v) \in E | (u \in V_a \wedge v \in V_b) \vee (u \in V_b \wedge v \in V_a)\} = \emptyset$ [43]. In our presentation we assume the matrix has symmetric non-zero structure (if it is asymmetric we can always add zero weight reverse edges while at most doubling the number of edges).

We now describe how to build a separator tree assuming we have a good algorithm FindSeparator for finding separators. For planar graphs this can be done in linear time [43]. The algorithm for building the tree is defined by Algorithm BuildTree in Figure 4. At each recursive call it partitions the edges into two subsets that are passed to the left and right children. All the vertices in the separator are passed to both children. Each leaf corresponds to a single edge. We assume that FindSeparator only puts a vertex in the separator if it has an edge to each side and always returns a separator with at least one vertex on each side unless the graph is a clique. If the graph is a clique, we assume the separator contains all but one of the vertices, and that the remaining vertex is on the left side (V_a) of the partition. By convention we place any edges between vertices in the separator in E_b.

Every vertex of degree Δ in our original graph corresponds to a binary tree embedded in the separator tree with Δ leaves, one for each of its incident edges. To see this consider a single vertex. Every time it appears in a separator, its edges are partitioned into two sets, and the vertex is copied to both recursive calls. Because the vertex will appear in Δ leaves, it must appear in $\Delta - 1$ separators, so it will appear in $\Delta - 1$ internal nodes of the separator tree. We refer to the tree for a vertex as the *vertex tree*, each appearance of a vertex in the tree as a *vertex copy*, and the root of each tree as the *vertex root*. The tree is used to sum the values for matrix vector multiply.

We reorder the rows/columns of the matrix based on a preorder traversal of their root locations in the separator tree (*i.e.*, all vertices in the top separator will appear first). This is the order we will use for the input vector x and output vector y when calculating $y = Ax$. We keep a vector R in this order that points to each of the corresponding roots of the tree. The separator tree is maintained as a tree T in which each node keeps its copies of the vertices in its separator. Each of these vertex copies will point to its two children in the vertex tree. Each leaf of T is an edge and includes the indices of its two endpoints and its weight. In all internal vertex copies we keep an extra value field to store a temporary variable, and in the leaves we keep two value fields, one for each direction. Finally we note that all data for each node of the separator tree is stored adjacently (*i.e.*, all its vertex copies are stored one after the other), and the nodes are stored in preorder. This is important for cache efficiency.

Our algorithm for sparse-matrix vector multiplication is described in Algorithm SparseMxV in Figure 4. This algorithm will take the input vector x and leave the results of the matrix multiplication in the root of every vertex. To gather the results up into a result vector y we simply use the root pointers R to fetch each root. The algorithm does not do any work beyond the recursive calls on the way down the recursion, but when it gets to a leaf the edge multiplies its two endpoints by its weight putting the result in its temporary value. If the matrix is symmetric then only one weight is needed. Then on the way back up the recursion the algorithms sums these values. In particular whenever it gets to an internal node of a vertex tree it adds the two children. Since the algorithm works bottom up the values of the children are always ready when the parent reads them.

THEOREM 4.1. *Let $\mathcal{M}$ be a class of matrices for which the adjacency graphs satisfy an n^ϵ-vertex separator theorem. Algorithm SparseMxV on an $n \times n$ matrix $A \in \mathcal{M}$ with $m \geq n$ non-zeros has $O(m)$ work, $O(\log^2 n)$ depth and $O(\lceil m/B + n/M^{1-\epsilon} \rceil)$ sequential cache complexity.*

PROOF. For a constant k we say a vertex copy is heavy if it appears in a separator node with size (memory usage) larger than M/k. We say a vertex is heavy if it has any heavy vertex copies. We first show that the number of heavy vertex copies for any constant k is bounded by $O(n/M^{1-\epsilon})$ and then bound the number of cache misses based on the number of heavy copies.

For a node of n vertices, the size $X(n)$ of the tree rooted at the node is bounded by the separator condition giving the recurrence relation:

$$X(n) \quad \leq \quad \max_{1/2 \geq \alpha' \geq \alpha} \{X(\alpha' n) + X((1-\alpha')n) + \beta n^\epsilon\}$$

This recurrence is satisfied by $X(n) = k(n - n^\epsilon)$, $k = \beta/(\alpha^\epsilon + (1-\alpha)^\epsilon - 1)$. Therefore, there exists a constant c such that for $n < cM$, the subtree rooted at node of n vertices fits into the cache. We use this to count the number of heavy vertex copies $H(n)$. The recurrence relation for $H(n)$ is:

$$H(n) \leq \max_{\alpha \leq \alpha' \leq \frac{1}{2}} \{H(\alpha' n) + H((1-\alpha')n) + \beta n^\epsilon\},$$

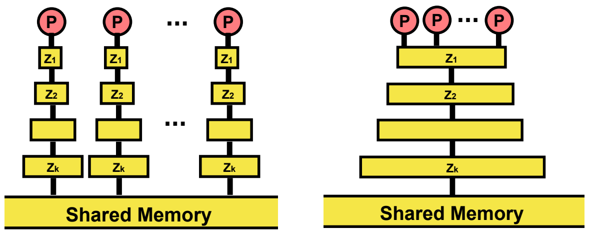


Figure 5: *Left:* Parallel Multi-level Distributed Hierarchy (PMDH). *Right:* Parallel Multi-level Shared Hierarchy (PMSH).

for $n > cM$ and 0 otherwise. This recurrence relation is satisfied by $H(n) = k(n/(cM)^{1-\epsilon} - \beta n^\epsilon) = O(n/M^{1-\epsilon})$.

Now we note that if a vertex is not heavy (*i.e.*, light) it is only used by a single subtree that fits in cache. Furthermore because of the ordering of the vertices based on where the roots appear, all light vertices that appear in the same subtree are adjacent. Therefore the total cost of cache misses for light vertices is $O(n/B)$. We note that the edges are traversed in order so they only incur $O(m/B)$ misses. Now each of the heavy vertex copies can be responsible for at most $O(1)$ cache misses. In particular reading each child can cause a miss. Furthermore reading the value from a heavy vertex (at the leaf of the recursion) could cause a miss since it is not stored in the subtree that fits into cache. But the number of subtrees that just fit into cache (*i.e.*, their parents don't) and read a vertex u is bounded by one more than the number of heavy copies of u. Therefore we can count each of those misses against a heavy copy. We therefore have a total of $O(m/B + n/M^{1-\epsilon})$ misses.

The work is simply proportional to the number of vertex copies, which is less than twice m and hence is bounded by $O(m)$. For the depth we note that the two recursive calls can be made in parallel and furthermore the **for all** statement can be made in parallel. Furthermore the tree is depth $O(\log n)$ because of the balance condition on separators. Since the branching of the **for all** takes $O(\log n)$ depth, the total depth is bounded by $O(\log^2 n)$. $\square$

5. MAPPING TO PARALLEL MULTI-LEVEL HIERARCHIES

In this section, we discuss how (low-depth) algorithms designed for the cache-oblivious model can be scheduled on natural multi-level generalizations of one-level private or shared cache machines, such that we can upper bound the parallel cache complexity and the parallel run time. We begin by defining the two models we consider.

5.1 PMDH and PMSH Models

We consider a *Parallel Multi-level Distributed Hierarchy (PMDH)* (Figure 5(left)), where each of the p processors has a multi-level private hierarchy and a *Parallel Multi-level Shared Hierarchy (PMSH)* (Figure 5(right)), where all the processors share a multi-level cache hierarchy. All computation by a processor p occurs on data in p's (private or shared) level-one cache. One or more cache lines of a given cache at level $i < k$ fit precisely in a cache line of its "parent" cache at level $i+1$. We assume the cache hierarchy is inclusive: each cached word at level $i < k$ is also cached in its parent cache at level $i+1$. A processor requesting a memory word *fetches* the cache line containing the word from the lowest-level ancestor cache containing the line (and populates all intervening caches). If the processor writes to the memory word, only the level-one cache line is updated and the line becomes *dirty*. Whenever a dirty line is evicted from a cache, its contents are written back to the corresponding line in its parent cache. Each cache is fully associative and uses an optimal replacement policy (within the constraints of being inclusive).

Cache Consistency in PDMH. In private cache models, the same memory word can be in the caches of multiple processors and these copies must be kept consistent. As in the two-level private cache model studied by Frigo *et al.* [39], the multi-level PMDH assumes a variant of the *dag consistency* cache consistency model [19] that uses an optimal replacement policy instead of LRU replacement. (We revisit LRU replacement in Section 6.) Caches are *non-interfering* in that the cache misses of one processor can be analyzed independent of other processors. To maintain this property, Frigo *et al.* use the BACKER protocol [19]. This protocol manages caches so that if an instruction j is a descendant of instruction i, then values written to memory words by i are reflected in j's memory accesses. However, concurrent writes to objects by instructions that do not have a path between them in the dag will not be communicated between processors executing these instructions. Such writes are *reconciled* to shared memory and reflected in other cache copies only when a descendant of the instruction that performed these writes tries to access them. Reconciliation of a memory block involves updating all written words within the block; the protocol must track all such writes. In case of multiple writes to the same word, an arbitrary write succeeds. Concurrent reads are permitted. We likewise assume the same non-interfering property, with the same reconciliation process.

5.2 Extending Private Cache Results to Multiple Levels

Because each processor in the PMDH model has its own private memory hierarchy, it is better to have each processor work on parts of computations that are as far apart as possible. The work stealing scheduler [21, 9, 1] is an ideal choice for such a system. In its simpler form, a work stealing scheduler maintains a task dequeue for each processor. When a processor spawns a new job, the new job is queued at the tail of its dequeue. When a processor runs out of work, it pulls out the job at the head of its task queue. If its own task queue is empty, the processors randomly picks another task queue to steal from. This version of the work stealing is referred to as *randomized work stealing*. Another (perhaps less practical) version of work stealing uses a single shared task dequeue for all processors; we refer to this as *centralized work stealing*.

We derive run time bound for algorithms under randomized work stealing for the PMDH such that the only algorithm-specific metrics in the bound are W, D and the sequential cache complexity Q. (To simplify notation, we will use $Q(M, B)$ instead of $Q(n; M, B)$ in the remainder of this section.) Given a particular execution X of a computation A on some parallel machine P, let $c(x)$ be the cost of instruction x. This cost includes the time for accessing the data used by x; if the access needs to fetch the data from level i cache, the cost is C_i (we view the shared memory as level $k+1$). The latency added work under execution X is $W^{lat}_{A,P}(X) = \sum_{x \in A} c(x)$. The *latency-added work*, $W^{lat}_{A,P}$, of a computation is the maximum of $W^{lat}_{A,P}(X)$ over all executions X. This can be bounded by $W + \sum_{i=2}^{k+1}(Q(M_{i-1}, B_{i-1}) - Q(M_i, B_i)) \cdot C_i + (\#S)(M_k/B_k)C_k$, where $\#S$ is the number of steals. The *latency added depth* $D^{lat}_{A,P}$ can be defined using $c(x)$ similarly: it is the maximum of $\sum_{x \in P} c(x)$ over all paths P in A. We note that $D \cdot C_{k+1}$ is a pessimistic upper bound on the latency-added depth for any machine.

THEOREM 5.1. **(Upper Bounds)** *For any $\delta > 0$, when a cache-oblivious nested-parallel computation A with binary forking, sequential cache complexity $Q(M, B)$, work W, and depth D is scheduled on a PMDH P of p processors using randomized work stealing:*

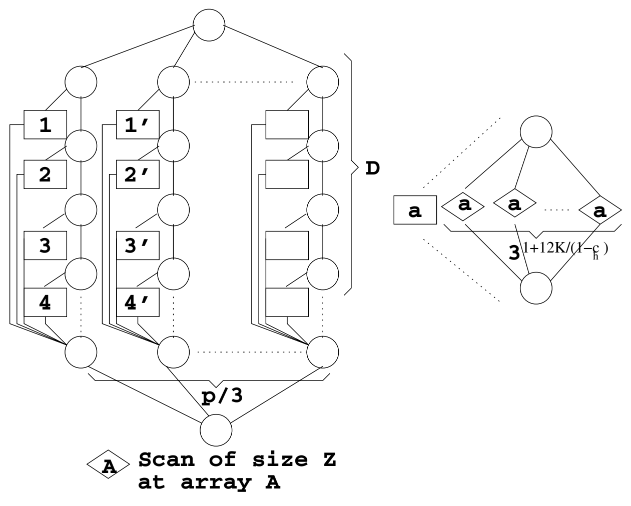
- *The number of steals is $O(p(D^{lat}_{A,P} + \log 1/\delta))$ with probability at least $1 - \delta$.*
- *All the caches at level i incur a total of less than $Q(M_i, B_i) + O(p(D^{lat}_{A,P} + \log 1/\delta)M_i/B_i)$ cache misses with probability at least $1 - \delta$.*
- *The computation completes in time not more than $W^{lat}_{A,P}/p + D^{lat}_{A,P} < W/p + O(p(DC_{k+1} + \log 1/\delta)C_{k+1}M_k/B_k + \sum_{i=1}^{k} C_i(Q(M_{i-1}, B_{i-1}) - Q(M_i, B_i)))/p + DC_{k+1}$ with probability at least $1 - \delta$.*

PROOF. We use Lemma 12 from [21] to bound the number of steals. Since that result uses a simpler model for the computation that does not charge cache miss costs towards run time, we reduce our dag to a simpler form on which the lemma can be applied directly. For each instruction in A, we replace the instruction by a chain of $c(x)$ (according to some execution) sequential instructions. Each of these replaced instructions take unit time. If x is a fork (join) point, the last (first) node in this chain does the equivalent fork (join). Since we assume a dag-consistent memory model, the run time of this modified computation A' is the same as that of A. Since the depth of A' under any execution does not exceed $D^{lat}_{A,P}$, the schedule involves not more than $O(p(D^{lat}_{A,P} + \log 1/\delta))$ steals with probability at least $1 - \delta$, all the caches at level i incur a total of at most $Q(M_i, B_i) + O(p(D^{lat}_{A,P} + \log 1/\delta)M_i/B_i)$ cache misses with probability at least $1 - \delta$. To bound the running time of the computation A' which has at most $W^{lat}_{A,P}$ instructions and $D^{lat}_{A,P}$ depth, we use Theorem 13 from [21]. Since $W^{lat}_{A,P} \leq W + \sum_i C_i \cdot (Q(M_{i-1}, B_{i-1}) - Q(M_i, B_i))$, the run time is at most $W + O(p(D^{lat}_{A,P} + \log 1/\delta)C_k M_k/B_k + \sum_i C_i(Q(M_{i-1}, B_{i-1}) - Q(M_i, B_i)))$ with probability at least $1 - \delta$, the claim about the running time follows. □

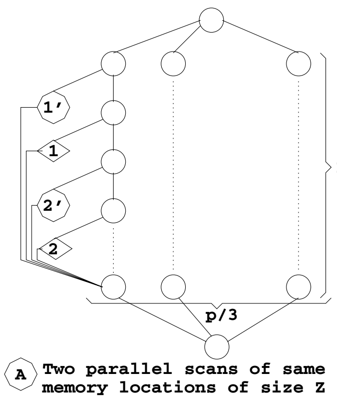
Thus, for constant δ, the parallel cache complexity at level i exceeds the sequential cache complexity by $O(pD^{lat}_{A,P}M_i/B_i)$ with probability $1 - \delta$. The bounds in Theorem 5.1 carry over to centralized work stealing without the δ terms, *e.g.*, the parallel cache complexity exceeds the sequential cache complexity by $O(pD^{lat}_{A,P}M_i/B_i)$. Throughout this section, the runtime bounds do not include scheduler overheads (which would increase the runtime by at most a small constant factor).

THEOREM 5.2. **(Lower Bound)** *For a PMDH P with any given number of processors $p = \Omega(\log D)$, cache sizes $M_1 < \cdots < M_k \leq M/3$ for some a priori upper bound M, cache line sizes $B_1 \leq \cdots \leq B_k$, and cache latencies $C_1 < \cdots < C_{k+1}$, and for any given depth $D' \geq 3(\log p + \log M) + C_{k+1} + c_0$ (for some constant c_0), we can construct a nested-parallel computation DAG with binary forking and depth D', whose (expected) parallel cache complexity on P, for all levels i, exceeds the sequential cache complexity $Q(M_i, B_i)$ by $\Omega(pD^{lat}_{A,P}M_i/B_i)$ when scheduled using randomized or centralized work stealing.*

PROOF. **Randomized work stealing**: Such a construction is shown in Figure 6(a). Based on the earlier lemma, we know that there exist a constant K such that the number of steals is at most KpD with probability at least $1 - (1/pD)$. We construct the DAG such that it consists of a binary fanout to $p/3$ spines of length $D =$



(a) Randomized work stealing



(b) Centralized work stealing

Figure 6: DAGs used in the proof of Theorem 5.2.

$D' - 2(12K/(1-c_h) + \log(p/3) + \log M)$ each ($c_h \in (0,1)$ is a constant that we will define shortly). Each of the first $D/2$ nodes on the spine forks off a "superscan" that consists of $3^{1+12K/(1-c_h)}$ identical parallel scans of length M each. A scan over an array A of size M is a binary tree forking out in to M parallel leaves, each leaf scanning one of the consecutive words in the array A. The remaining $D/2$ nodes on the spine are the joins corresponding to superscans forked up the spine. Note that $D^{lat}_{A,P} = D' + C_{k+1}$ because each path in the DAG contains at most one memory request.

In a sequential execution, a processor executes the superscans one by one and can reuse a subsequence of length M_i/B_i (at level i) for all the identical scans with in a superscan. In other words, sequential execution gets $(pD/6)(3^{1+12K/(1-c_h)}-1)\lceil(M_i-B_i)/B_i\rceil$ cache hits at level i cache, and the sequential cache complexity $Q(M_i,B_i)$ is $(pD/6)(\lceil M_i/B_i\rceil + 3^{1+12K/(1-c_h)}\lceil(M-M_i)/B_i\rceil)$.

We argue that in the case of randomized work stealing, there are a large number of superscans such that the probability that at least two scans from such superscans are executed by different processors is greater than some positive constant. This implies that the cache complexity is $\Theta(pDM_i/B_i)$ higher that the sequential cache complexity (claim A).

1. Once the $p/3$ spines have been forked, each spine is occupied by at least one processor till the stage where work along a spine has been exhausted. This property follows directly from the nature of the work stealing protocol.
2. In the early stages of computation after spines have been forked, but before the computation enters the join phase on the spines, exactly $p/3$ processors have a spine node on the head of their work queue. Therefore, the probability that a random steal will get a spine node and hence a fresh superscan is $1/3$.
3. At any moment during the computation, the probability that more than $p/2$ of the latest steals of the p processors found fresh spine nodes is exponentially small in terms of p and therefore less than $1/2$.
4. If processor p stole a fresh superscan A and started the scans in it, the probability that the work from the superscan A is not stolen by some other processor before p executes the first $2/3$-rd of the scan is at most a constant $c_h \in (0,1)$. This is because the probability that p currently got a fresh superscan does not depend on events in the history, and therefore, with probability at least $1/2$, more than $p/2$ processors did not steal a fresh superscan in the latest steal. This means that these processors which stole a stale superscan got less than $2/3$-rd fraction of the superscan to work on before they need to steal again. Therefore, by the time p finishes $2/3$-rd of the work, there would have been at least $p/2$ steal attempts and there is a probability of at least $1/16$ that two of these steals stole from p. Two steals from p would cause p to lose work from it's fresh superscan. In this scenario, p does not execute more than $5/6$-th of the scan even if it comes back to steal work from the higher instance of scan A.
5. Since there are at most KpD steals with high probability, there are no more than $(1-c_h)pD/12$ superscans which incur more than $12K/(1-c_h)$ steals. On an average, about $(1-c_h)pD/6$ superscans are stolen from before the first processor that touched the superscan executes $2/3$-rd of it. Therefore, on an average, there are no less than $(1-c_h)pD/12$ superscans which get fewer than $12K/1-c_h$ steals and are stolen from before one processor executes $2/3$-rd of it. Such superscans, by construction have at least two different processors execute a complete scan. This proves claim A.

Centralized work stealing: The construction for centralized work stealing is shown in Figure 6(b). The DAG ensures each steal causes a scan of a completely different set of memory locations. The bound follows from the fact that unlike the case in sequential computation, cache access overlap in the pairs of parallel scans are never exploited. □

Clearly this lower bound also applies to more general multi-level cache models such as studied in [3, 4, 30, 42, 46, 49].

5.3 Extending Shared Cache Results to Multiple Levels

Finally, we consider the PMSH model. For the case of a single level of shared cache, our previous work [15] showed that the parallel depth-first (PDF) scheduler was a good choice for mapping good sequential cache complexity to provably good parallel cache complexity. In the PDF scheduler [16, 15] tasks are prioritized according to their ordering in the natural sequential execution, *i.e.*, according to the ordering used to analyze the sequential cache complexity Q; the ith task in the sequential execution is given priority rank i. A processor completing a task is assigned the lowest ranked task among all the available tasks that are ready to execute. The relative ranking of available tasks can be efficiently determined on-the-fly without having to perform a sequential execution [16].

The results from [15] stated in Section 1 for a single level of shared cache can be generalized to the PMSH:

THEOREM 5.3. *When a cache-oblivious nested-parallel computation A with sequential cache complexity $Q(M, B)$, work W, and depth D is scheduled on a PMSH P of p processors using a PDF scheduler, then the cache at each level i incurs fewer than $Q(p(M_i - B_i D^{lat}_{A,P}), B_i)$ cache misses. Moreover, the computation completes in time not more than $W^{lat}_{A,P}/p + D^{lat}_{A,P}$.*

PROOF. (sketch) The cache bound follows because (1) inclusion implies that hits/misses/evictions at levels $< i$ do not alter the number of misses at level i, (2) caches sized for inclusion imply that all words in a line evicted at level $> i$ will have already been evicted at level i, and hence (3) the key property of PDF schedulers, $Q_p(M + pBD^{lat}_{A,P}, B) \leq Q(M, B)$, holds at each level i of a PMSH. The time bound follows because the schedule is greedy (and we are not accounting for scheduler overheads). □

Thus, our approach for developing cache-efficient parallel algorithms via (i) low cache-oblivious sequential cache complexity and (ii) low depth is validated for shared-cache hierarchies (and PDF schedulers) as well.

6. DISCUSSION

A goal of the work described in this paper is to develop a simple model for accounting for locality with dynamic parallelism—cleanly separating the cost-model from any particular machine model while still being useful in bounding costs on various machines. We believe the approach of analyzing cache complexity in the cache-oblivious model, and work and depth with dynamic nested parallelism as described in the paper achieves this goal. The approach, however, does have some limitations and ignores some details. We briefly describe these here. Firstly the general bounds on the parallel cache misses rely on low-depth (as the title of the paper implies). It seems that avoiding this would require a modified model for cache complexity, or taking into account particular properties of programs as studied in some previous work [39, 14, 29]. A more general approach to handle algorithms with higher depth would be useful.

Secondly, our scheduler results assume DAG consistency using the BACKER protocol, which at present is not implemented on real machines. The backer protocol avoids cache protocol misses due false sharing (multiple threads writing to different locations of a shared cache line) by resolving cache line conflict when writing back to memory. Strong consistency is not guaranteed and not needed by our DAG consistent algorithms. Maintaining strong consistency per cache line could create problems on the algorithms and scheduling techniques we described by forcing cache lines to "ping-pong" among processors. It would be interesting to develop a model and algorithms that avoid these problems, but ultimately if this makes the process of designing or analyzing algorithms more complicated, or breaks the abstraction between a high-level model and the machines below, this would be a argument to modify cache consistency protocols.

Thirdly, our scheduler results assume an optimal cache replacement policy. Note that for practical purposes, each level of cache could instead use a multi-level inclusive LRU replacement policy. Unlike in the case of optimal replacement, where a complete memory access profile may be needed a priori at all levels in order to compute what to replace, implementing a multi-level LRU replacement policy does not require that all levels of the cache hierarchy see the memory access profile. Assuming that a cache line evicted at level i is sent to level $i + 1$ and that any access to a memory location not at level i is serviced by passing it from higher levels in the memory hierarchy through cache level $i + 1$, it is possible for cache level $i + 1$ to know exactly what memory words are contained in the lower level cache. From the order in which cache lines were evicted by level i, cache level $i + 1$ can fill up the rest of its slots and order them in LRU order (see Figure 7). It follows from [47] that the number of cache misses at each level under the multi-level LRU policy is within a factor of two of the number of misses for a cache half the size running the optimal replacement policy. For example, under multi-level LRU, the upper bound on the cache misses in Theorem 5.1 becomes $2Q(M_i/2, B_i) + O(p(D^{lat}_{A,P} + \log 1/\delta)M_i/B_i)$.

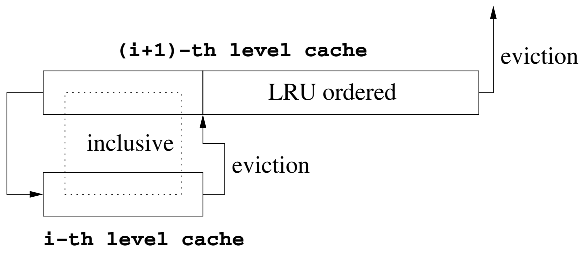


Figure 7: Multi-level LRU

Finally, our scheduler results are for multi-level hierarchies of private or shared caches. It would be interesting to extend these results to more general multi-level models [3, 4, 18, 30, 42, 46, 49], while preserving the goal of supporting a simple model for algorithm design and analysis.

Acknowledgements. We thank Vijaya Ramachandran for discussions on this work. This work was funded in part by IBM, Intel, and the Microsoft-sponsored Center for Computational Thinking.

7. REFERENCES

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3), 2002.
[2] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *ACM STOC '87*, 1987.
[3] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12, 1994.
[4] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *Proc. 1993 Conf. on Programming Models for Massively Parallel Computers*, 1993.
[5] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *ACM STOC '02*, 2002.
[6] L. Arge, G. S. Brodal, and R. Fagerberg. Cache-oblivous data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*. CRC Press, 2005.
[7] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *ACM SPAA '08*, 2008.
[8] L. Arge, M. T. Goodrich, and N. Sitchinava. Parallel external memory graph algorithms. Manuscript, 2009.
[9] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *ACM SPAA '98*, 1998.
[10] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and

E. Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. In *ACM SPAA '07*, 2007.

[11] M. A. Bender, B. C. Kuszmaul, S.-H. Teng, and K. Wang. Optimal cache-oblivious mesh layout. Computing Research Repository (CoRR) abs/0705.1033, 2007.

[12] G. Bilardi. Models for parallel and hierarchical computation. In *Proc. 4th ACM International Conf. on Computing Frontiers*, 2007.

[13] G. E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3), 1996.

[14] 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.

[15] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *ACM SPAA '04*, 2004.

[16] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2), 1999.

[17] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Brief announcement: Low-depth cache oblivious sorting. In *ACM SPAA '09*, 2009.

[18] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low-depth cache oblivious algorithms. Technical Report CMU-CS-TR-134, Computer Science Department, Carnegie Mellon University, 2009 http://reports-archive.adm.cs.cmu.edu/anon/2009/CMU-CS-09-134.pdf.

[19] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In *IPPS '96*, 1996.

[20] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1), 1996.

[21] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5), 1999.

[22] G. S. Brodal. Cache-oblivious algorithms and data structures. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*, 2004. LNCS, vol. 3111. Springer.

[23] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *ICALP '02*, 2002. LNCS, vol. 2380. Springer.

[24] G. S. Brodal, R. Fagerberg, and G. Moruz. Cache-aware and cache-oblivious adaptive sorting. In *ICALP '05*, 2005. LNCS, vol. 3580. Springer.

[25] G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. *ACM Journal of Experimental Algorithmics*, 12, 2008.

[26] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An object-oriented approach to non-uniform clustered computing. In *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming Languages and Applications*, 2005.

[27] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *ACM-SIAM SODA '95*, 1995.

[28] R. A. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. In *ACM SPAA '07*, 2007.

[29] R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *ACM SPAA '08*, 2008.

[30] R. A. Chowdhury, V. Ramachandran, and F. Silvestri. Oblivious algorithms for multicore, network, and petascale computing. Manuscript, 2009.

[31] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *IEEE IPDPS '10*, 2010.

[32] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *ACM STOC '86*, 1986.

[33] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *ACM PPOPP '93*, 1993.

[34] 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.

[35] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Supercomputing '06*, 2006.

[36] G. Franceschini. Proximity mergesort: Optimal in-place sorting in the cache-oblivious model. In *ACM-SIAM SODA '04*, 2004.

[37] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM*, 17(3), 1970.

[38] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *IEEE FOCS '99*, 1999.

[39] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *ACM SPAA '06*, 2006.

[40] J. Jaja. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[41] P. Kumar. Cache oblivious algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*. Springer, 2003.

[42] E. Ladan-Mozes and C. E. Leiserson. A consistency architecture for hierarchical shared caches. In *ACM SPAA '08*, 2008.

[43] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36, 1979.

[44] OpenMP Architecture Review Board. OpenMP application program interface. Technical Report Version 3.0, 2008.

[45] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3), 1989.

[46] J. E. Savage and M. Zubair. A unified model for multicore architectures. In *Proc. 1st International Forum on Next-Generation Multicore/Manycore Technologies*, 2008.

[47] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2), 1985.

[48] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8), 1990.

[49] L. G. Valiant. A bridging model for multicore computing. In *ESA*, 2008.