Low Depth Cache-Oblivious Algorithms

Guy E. Blelloch[*] **Phillip B. Gibbons**[†]
Harsha Vardhan Simhadri[**]

July 2009
CMU-CS-09-134

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[*] email: `guyb@cs.cmu.edu`
[†] Intel Labs Pittsburgh. email: `phillip.b.gibbons@intel.com`
[**] email: `harshas@cs.cmu.edu`

Blelloch and Simhadri are supported by IBM, Intel, and the Microsoft-sponsored Center for Computational Thinking.

Keywords: Cache-oblivious algorithms, sorting, sparse-matrix vector multiply, graph algorithms, parallel algorithms, multi-core, multiprocessors, schedulers, parallel tree-of-caches

Abstract

In this paper we explore a simple and general approach for developing parallel algorithms that lead to good cache complexity on a variety of parallel cache architectures. 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. Our sorting algorithm yields the first cache-oblivious algorithms with polylogarithmic depth and low sequential cache complexities for list ranking, Euler tour tree labeling, tree contraction, least common ancestors, graph connectivity, and minimum spanning forest.

Using known mappings, our results lead to low cache complexities on multi-core processors (and shared-memory multiprocessors) with a single level of private caches or a single shared cache. We generalize these mappings to a multi-level *parallel tree-of-caches* model that reflects current and future trends in multi-core cache hierarchies—these new mappings imply 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.

1 Introduction

The *cache-oblivious model* (*ideal-cache model*) [29] is a two-level model of computation comprised 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 (cache misses)). 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 Z_i and L_i at each level i [29]. Researchers have developed cache-oblivious algorithms for a variety of problems [6, 18, 26].

The cache complexity $Q_1(n; Z, L)$ for a natural sequential execution of a parallel program (on input of size n) can also be used to bound the cache complexity $Q_P(n; Z, L)$ for the same program on P-processor parallel machines with a shared memory and a single level of cache(s) [1, 14]. In particular, for a parallel machine with private caches (each processor has its own cache of size Z) using a work-stealing scheduler [17],

$$Q_P(n; Z, L) < Q_1(n; Z, L) + O(ZPD/L) \text{ with probability } 1 - \delta \text{ [1]},^{1} \tag{1}$$

and for a parallel machine with a shared cache using a parallel depth-first (PDF) scheduler [15],

$$Q_P(n; Z + PLD, L) \leq Q_1(n; Z, L) \text{ [14]}, \tag{2}$$

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 computations are series-parallel.) The "natural" sequential execution is simply one that runs each call in a fork to completion before starting the next. The *depth* of a computation (also known as the span or the critical path length) is defined recursively by taking the maximum of the depths of the forked threads at each fork-join construct. It is the length of the longest path in the computation dag. We assume concurrent threads can read the same location.

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 parallel machines with common configurations of *multi-level* cache hierarchies (namely, a *parallel tree-of-caches*). In particular, we show that a work-stealing scheduler achieves

$$Q_P(n; Z_i, L_i) < Q_1(n; Z_i, L_i) + O(Z_i PD/L_i) \text{ with probability } 1 - \delta \text{ for each level } i,$$

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 [29], $Q_1(n; Z, L) = O(n^{\lg 7}/(L\sqrt{Z}))$. Thus, we have that $Q_P(n; Z, L) < Q_1(n; Z, L) + O(ZP \log^2(n)/L)$ for a single level of private caches (Equation 1)

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

and $Q_P(n; Z + PL\log^2 n, L) \leq Q_1(n; Z, L) = O(n^{\lg 7}/(L\sqrt{Z}))$ for a shared cache (Equation 2). 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). Similarly good bounds are obtained for multi-level caches, using our results for such hierarchies.

Although some cache-oblivious algorithms are naturally parallel and have low depth (*e.g.*, matrix multiply, matrix transpose [29] and FFT [29]), many are not. For example, prior cache-oblivious sorting algorithms with optimal sequential cache complexity [19, 20, 21, 27, 29] 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 $Z = \Omega(L^2)$ [29], our (deterministic) sorting algorithm has cache complexity $Q_1(n; Z, L) = O(\frac{n}{L}\log_Z n)$ and work $W = O(n \log n)$, which are optimal, and depth $D = O(\log^2 n)$. The depth can be improved using randomization. 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; the algorithm is optimal work, $O(\log^2 n)$ depth, and its sequential cache complexity improves upon the previous best sequential algorithm for general sparse matrices 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.* [30] for a class of algorithms with a regularity condition, by Blelloch *et al.* [13] for a class of binary divide-and-conquer algorithms, and by Chowdhury and Ramachandran [23, 24] 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 range of algorithms and a variety of parallel machine configurations; 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, 8, 12], which demonstrate 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 private caches. Also their algorithms are described for a fixed number of processors P while in our approach the algorithms are described at a higher-level that is oblivious of P (the scheduler is responsible for mapping tasks onto processors). In fact, it is this flexibility that enables the approach to be applied to a variety of cache hierarchies.

2 Low-Depth Cache-Oblivious 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 [19, 20, 21, 27, 29] have $\Omega(\sqrt{n})$ depth.

Our sorting algorithm uses known algorithms for prefix sums and merging as subroutines. A simple variant of the standard parallel prefix-sums algorithm has logarithmic depth and cache complexity $O(n/L)$. The only adaptation is that the input has to be laid out in memory such that any subtree of the balanced binary tree over the input (representing the divide-and-conquer recursion) is contiguous. For completeness, the precise algorithm and analysis are in Section A.1 of the appendix. Likewise, a simple variant of a standard parallel merge algorithm also has logarithmic depth and cache complexity $O(n/L)$, as described next.

```
Algorithm MERGE($(A, s_A, l_A), (B, s_B, l_B), (C, s_C)$)
if $l_B = 0$ then
    Copy $A[s_A : s_A + l_A)$ to $C[s_C : s_C + l_A)$
else if $l_A = 0$ then
    Copy $B[s_B : s_B + l_B)$ to $C[s_C : s_C + l_B)$
else
    $\forall k \in [1 : \lfloor n^{1/3} \rfloor]$, 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 $B[s_B + b_k] \leq A[s_A + a_k + 1]$.
    $\forall k \in [1 : \lfloor n^{1/3} \rfloor]$, MERGE($(A, s_A + a_k, a_{k+1} - a_k), (B, s_B + b_k, b_{k+1} - b_k), (C, s_C + a_k + b_k)$)
end if
```

Figure 1: Merge $A[s_A : s_A + l_A)$ and $B[s_B : s_B + l_B)$ into array $C[s_C : s_C + l_A + l_B)$

2.1 Merge

To make the usual merging algorithm cache oblivious, we use divide-and-conquer with a branching factor of $n^{1/3}$. (We need a $o(\sqrt{n})$ branching factor in order to achieve optimal cache complexity.) 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 elements ranked $\{n^{2/3}, 2n^{2/3}, 3n^{2/3}, \dots\}$ among the set of keys from both arrays, and recurse on each pair of subarrays. This takes $n^{1/3} \cdot \log n$ work, $\log n$ depth and at most $O(n^{1/3} \log{(n/L)})$ 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. See Algorithm MERGE in Figure 1.

The cache complexity of Algorithm MERGE can be expressed using the recurrence

$$Q(n; Z, L) \leq n^{1/3}(\log{(n/L)} + Q(n^{2/3}; Z, L)), \tag{3}$$

where the base case is $Q(n; Z, L) = O(\lceil n/L \rceil)$ when $n \leq cZ$ for some positive constant c. When $n > cZ$, Equation 3 solves to

$$Q(n; Z, L) = O(n/L + n^{1/3} \log{(n/L)}). \tag{4}$$

Because $Z = \Omega(L^2)$ and $n > cZ$, the $O(n/L)$ term in Equation 4 is asymptotically larger than the $n^{1/3} \log{(n/L)}$ term, making the second term redundant. Therefore, in all cases, $Q(n; Z, L) = O(\lceil n/L \rceil)$. The recurrence relation for the 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.

The costs of Algorithm MERGE are summarized in Figure 2.

Mergesort. 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 $O((n/L) \log_2{(n/Z)})$, which is not optimal. Blelloch *et al.* [13] analyze similar merge and mergesort algorithms with the same (suboptimal) cache complexities but with larger depth. Next, we present a sorting algorithm with optimal cache complexity (and low depth).

2.2 Deterministic Sorting

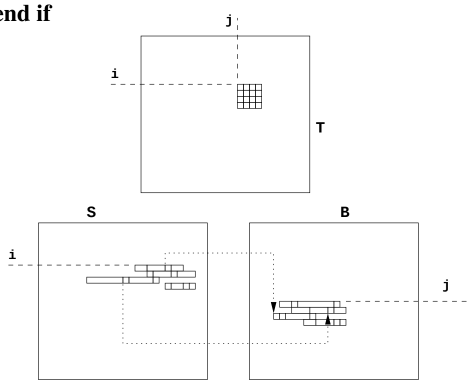
Our parallel sorting algorithm is based on a version of sample sort [28, 36]. Sample sorts first use a sample to select a set of pivots that partition the keys into buckets, then route all the keys to their appropriate buckets, and finally sort within the buckets. Compared to prior cache-friendly sample sort algorithms [2, 33], which incur $\Omega(\sqrt{n})$ 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.

Problem	Depth	Cache Complexity	Section
Prefix Sums	$O(\log n)$	$O(\lceil n/L \rceil)$	A.1
Merge	$O(\log n)$	$O(\lceil n/L \rceil)$	2.1
Sort (deterministic)*	$O(\log^2 n)$	$O(\lceil n/L \rceil \lceil \log_Z n \rceil)$	2.2
Sort (randomized; bounds are w.h.p.)*	$O(\log^{1.5} n)$	$O(\lceil n/L \rceil \lceil \log_Z n \rceil)$	2.3
Sparse-Matrix Vector Multiply (m non-zeros, n^ϵ separators)*	$O(\log^2 n)$	$O(\lceil m/L + n/Z^{1-\epsilon} \rceil)$	4
Matrix Transpose ($n \times m$ matrix)	$O(\log (n+m))$	$O(\lceil nm/L \rceil)$	[29]

Figure 2: 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 $Z = \Omega(L^2)$.

```
**Algorithm** COSORT($A$, $n$)
if $n \le 10$ 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, $h$)
$\forall i \in [1:h]$, $B'_i \leftarrow$ 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 $B_{b_s}[T_{i_s,i_b}\langle 2 \rangle : T_{i_s,i_b}\langle 2 \rangle + T_{i_s,i_b}\langle 3 \rangle)$
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 3: Cache-Oblivious Sorting and Bucket-Transpose Algorithms

Our sorting algorithm (Algorithm COSORT in Figure 3) 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 incurs $O(\lceil n/L \rceil)$ 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 $O(\lceil n/L \rceil)$ 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. We can use the standard divide-and-conquer matrix-transpose algorithm [29], which is work optimal, has logarithmic depth and has optimal cache complexity when $Z = \Omega(L^2)$ (details in Figure 2). The mapping of segments to bucket locations 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 3) 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 and j.

Lemma 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 $O(\lceil n/L \rceil)$ sequential cache complexity.*

Proof. It is easy to see that the work is $O(n)$ because there is only constant work for each of the $O(\log n)$ levels of recursion and each key is copied exactly once. Similarly, the depth is $O(\log n)$ because we can use prefix sums to do the copying whenever a segment is larger than $O(\log n)$.

To analyze the cache complexity, we use the following definitions. 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 $s(v) < Z/100$, $w(v) < Z/10$, and its parent node is of size $\geq Z/100$.
2. Light-2 nodes: A node v is light-2 if $s(v) < Z/100$, $w(v) < Z/10$, and its parent node is of weight $\geq Z/10$.
3. Heavy leaves: A leaf v is heavy if $w(v) \geq Z/10$.

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 Z/L cache misses. It can also be seen that light-1 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-1 nodes is no more than $O(\lceil n/L \rceil)$.

Light-2 nodes are similar to light-1 nodes in that their target data fits into a cache of size Z. If we assume that they have a combined weight of $n - W$, then there are no more than $4(n-W)/(Z/10)$ of them, putting the aggregate cache complexity for their subtrees 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 $W/L + 10W/Z < 11W/L$. Therefore, the cache complexities of light-2 nodes and heavy leaves add up to another $O(\lceil n/L \rceil)$.

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 $O(\sqrt{n} \log n)$. □

Theorem 1 *On an input of size n, the deterministic COSORT has $Q(n; Z, L) = O(\lceil n/L \rceil \lceil \log_Z n \rceil)$ 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 $O(\lceil n/L \rceil)$. 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:

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

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 $n \leq cZ$ for some constant c. Solving these recurrences proves the theorem. □

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 and has cache complexity $O(n/L)$ and the probability that the largest of the resultant buckets are larger than $c\sqrt{n} \log n$ is not more $1 - 1/n$ for a certain constant c. When one of the buckets is too large ($> c\sqrt{n} \log n$), 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.

Theorem 2 *On an input of size n, the randomized version of COSORT has, with probability greater than $1 - 1/n$, $Q(n; Z, L) = O(\lceil n/L \rceil \lceil \log_Z n \rceil)$ sequential cache complexity, $O(n \log n)$ work, and $O(\log^{1.5} n)$ depth.*

Proof. 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 $O(\lceil n/L \rceil)$ 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 $n_i < 2\sqrt{n} \log n$ and the n_is add up to n. This implies that $\mathbb{E}[W(n)] = O(n \log n)$. Similarly for cache complexity we have $\mathbb{E}[Q(n; Z, L)] = O(n/L) + \sqrt{n}\,\mathbb{E}[Q(\sqrt{n}; Z, L)] + \sum_{i=1}^{\sqrt{n}} \mathbb{E}[Q(n_i; Z, L)]$, which solves to $\mathbb{E}[Q(n; Z, L)] = O((n/L) \log_{\sqrt{Z}} n) = O((n/L) \log_Z n)$. To show the high probability bounds for work and cache complexity, we can use Chernoff bounds because the fan-out at each level of recursion is high.

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	$O(D_{LR}(n) \log n)$	$O(Q_{sort}(\|E\|) \log(\|V\|/\sqrt{Z}))$
Minimum Spanning Forest	$O(D_{LR}(n) \log n)$	$O(Q_{sort}(\|E\|) \log(\|V\|/\sqrt{Z}))$

Figure 4: Low-depth cache-oblivious graph algorithms. All algorithms are deterministic. The bounds assume $Z = \Omega(L^2)$. D_{sort} and Q_{sort} are the depth and cache complexity of cache-oblivious sorting.

Proving probability bounds on the depth is more involved. We prove that the sum of depths of all nodes at level d in a recursion tree is $O(\log^{1.5} n / \log\log n)$ with probability at least $1 - 1/n^{\log^2 \log n}$. We also show that the recursion tree has at most $O(\log\log n)$ levels and the number of instances of recursion tree is $n^O(1.1 \log\log n)$. This implies that the depth of the critical path computation is at most $O(\log^{1.5} n)$ with high probability. The details of this depth analysis are in Section A.2 of the appendix. □

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, 8] 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 their 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 [22] 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 [32] 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 [32]: use two rounds of Cole and Vishkin's deterministic coin tossing [25] 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 on $O(n/\log n)$ elements in order 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 *The deterministic list ranking outlined above has $Q(n; Z, L) = O(\lceil n/L \rceil \lceil \log_Z n \rceil)$ 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 4). The algorithms that lead to these complexities are straightforward adaptations of known

PRAM algorithms (as described for the external memory model in [22]). 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 [22]: $\log(|V|/\sqrt{Z})$ versus $\log(|V|/Z)$. While [22] uses knowledge of Z 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

In this section, we consider the problem of multiplying a sparse $n \times n$ matrix with m non-zeros by a dense vector (SpMV-multiply). 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 $O(m/L + n/Z^{1/d})$ [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 $Z \geq L^d$ (*e.g.*, the tall-cache assumption in 2 dimension), the cache complexity reduces to $O(m/L)$ which is as fast as scanning memory.

The cache-efficient layout and bounds for well-shaped meshes can be extended to graphs with good edge separators [13]. 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 is linear work, $O(\log^2 n)$ depth, and $O(m/L + n/Z^{1-\epsilon})$ 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 $G = (V, E)$ 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 $\{(u, v) \in E | (u \in V_a \wedge v \in V_b) \vee (u \in V_b \wedge v \in V_a)\} = \emptyset$. In our presentation we assume the matrix has symmetric non-zero structure (but not necessarily symmetric values (weights)); 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 [35]. The algorithm for building the tree is defined by Algorithm BuildTree in Figure 5. 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 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 of the partition.

Every vertex in our original graph of degree Δ corresponds to a binary tree embedded in the separator

```
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 $\leftarrow x[T.v$.index$] \otimes T.v$.weight
    $T.v$.value $\leftarrow x[T.u$.index$] \otimes T.u$.weight
    {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$\rightarrow$value $\oplus$ $v$.right$\rightarrow$value)
    end for
end if
```

Figure 5: Cache-Oblivious Algorithms for Building a Separator Tree and for Sparse-Matrix Vector Multiply

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 the SpMV-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 SpMV-multiply is described in Algorithm SparseMxV in Figure 5. 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 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. 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 *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 $O(\lceil m/L + n/Z^{1-\epsilon}\rceil)$ sequential cache complexity.*

Proof. For a constant k we say a vertex copy is heavy if appears in a separator node with size (memory usage) larger than Z/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 $O(n/Z^{1-\epsilon})$ 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 given by the recurrence relation:

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

This recurrence solves to $X(n) = k(n - n^{\epsilon})$, $k = \beta/(\alpha^{\epsilon} + (1-\alpha)^{\epsilon} - 1)$. Therefore, there exists a positive constant c such that for $n \leq cZ$, the subtree rooted at a 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:

$$H(n) \quad = \begin{cases} \max_{\alpha \leq \alpha' \leq \frac{1}{2}} \{H(\alpha' n) + H((1-\alpha')n) + \beta n^{\epsilon}\} & \text{if } n > cZ \\ 0 & \text{otherwise} \end{cases}$$

This recurrence relation solves to $H(n) = k(n/(cZ)^{1-\epsilon} - \beta n^{\epsilon}) = O(n/Z^{1-\epsilon})$.

Now we note that if a vertex is not heavy (*i.e.*, light) it is used only 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 $O(n/L)$. We note that the edges are traversed in order so they only incur $O(m/L)$ 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 do not fit) 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 $O(m/L + n/Z^{1-\epsilon})$ 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 (both $|V_{a,sep}|$ and $|V_{b,sep}|$ are at most $\alpha n + \beta n^{\epsilon}$). Since the branching of the **for all** takes $O(\log n)$ depth, the total depth is bounded by $O(\log^2 n)$. □

5 Parallel Multi-level Hierarchies

In this section, we show how algorithms designed for the cache-oblivious model can be scheduled on a class of parallel multi-level cache hierarchy models, such that we can upper bound the parallel cache complexity and the parallel time. We define the class of hierarchies we consider (Section 5.1), then present our bounds (Section 5.2).

5.1 Tree-of-Caches Hierarchy

We consider the family of parallel cache hierarchies depicted in Figure 6(left), which we call the *Parallel Tree-of-Caches (PToC)*. Each of the p processors is connected to a private level-one cache. Disjoint (not necessarily equal-sized) groups of processors each share a larger level-two cache, and so on, forming a tree of caches of k levels. All computation by a processor P occurs on data in P's 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 $i + 1$. Tree of caches models [3, 4, 12, 34, 37, 39] reflect current and future trends in many-core cache hierarchies [31, 34, 37], with increasing numbers of cores and new memory technologies (Flash, PCM, etc.) increasing the numbers of leaves and levels in the hierarchy.

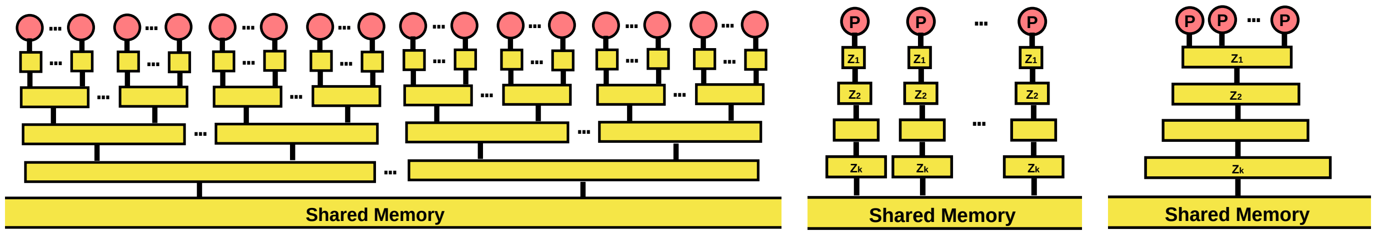


Figure 6: *Left:* Parallel Tree-of-Caches Hierarchy (PToC). Cache sizes and fan-out need not be regular. *Center:* Parallel Multi-level Distributed Hierarchy (PMDH). *Right:* Parallel Multi-level Shared Hierarchy (PMSH).

As in the cache-oblivious model, each cache is assumed to be fully associative and use an optimal replacement policy (an alternative LRU replacement policy is discussed in Section A.3 of the appendix). We further 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). However, depending on the consistency model (discussed next), parent caches may have stale data, in order to avoid updating k ancestor caches on each write to a level-one cache.

As in the two-level private cache model studied by Frigo *et al.* [30], the multi-level PToC assumes a variant of the *dag consistency* cache consistency model [16] that uses an optimal replacement policy instead of LRU replacement. Caches are *non-interfering* in that the cache misses of one processor can be analyzed independently of other processors. To maintain this property, Frigo *et al.* use the BACKER protocol [16]. This protocol manages caches so that if an instruction j is a descendant of instruction i in the computation DAG, then values written to memory objects 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.

A PToC has *intra-level regularity* if all the caches in any given level i

- have the same line size L_i,
- take the same time (latency cost) C_i to fetch a line from the cache (and populate all intervening caches), where $C_1 < \cdots < C_k$, and
- are of size $p_{i,j} \cdot Z_i$, where $p_{i,j}$ is the number of processors sharing the jth cache at level i.

This third condition allows for irregular fan-out in the PToC, while assuming *proportionate caches*, i.e., the sizes of the caches at a given level are proportional to the number of processors sharing the cache. Intra-level regularity is a natural restriction in practice and it simplifies the presentation of our bounds; the results can be readily extended to more general families of PToCs.

A specific instance of the PToC with intra-level regularity is the *Parallel Multi-level Distributed Hierarchy (PMDH)* shown in Figure 6(center). In the PMDH with p processors, $p_{i,j} = 1$ for $i \in [1..k]$ and $j \in [1..p]$. We prove our bounds first for the simpler PMDH, and then discuss implications for the PToC.

We focus on nested-parallel computations that are scheduled using a standard (randomized) work-stealing scheduler [17], in which idle processors steal work from random other processors. We will also consider a centralized work-stealing scheduler, as well as a parallel depth-first (PDF) scheduler [15].

5.2 Complexity Bounds

We now derive bounds for the PToC such that the only algorithm-specific metrics are W, D and Q. Given a particular execution X of a computation on some parallel machine M, let $W_M^{lat}(X)$ be $W + \sum_i (C_i \cdot F_{M,i}(X))$, where $F_{M,i}(X)$ is the number of fetches from a cache at level i. The *latency-added work*, W_M^{lat}, of a computation is the maximum of $W_M^{lat}(X)$ over all executions X. The *latency added depth* D_M^{lat} can be defined similarly. We note that $W \cdot C_k$ and $D \cdot C_k$ are pessimistic upper bounds on the latency-added work and depth for any machine.

5.2.1 Bounds for the PMDH

Theorem 5 (Upper Bounds) *For any $\delta > 0$, when a cache-oblivious nested-parallel computation with binary forking, sequential cache complexity $Q(Z, L)$, work W, and depth D is scheduled on a PMDH P of p processors using work stealing:*

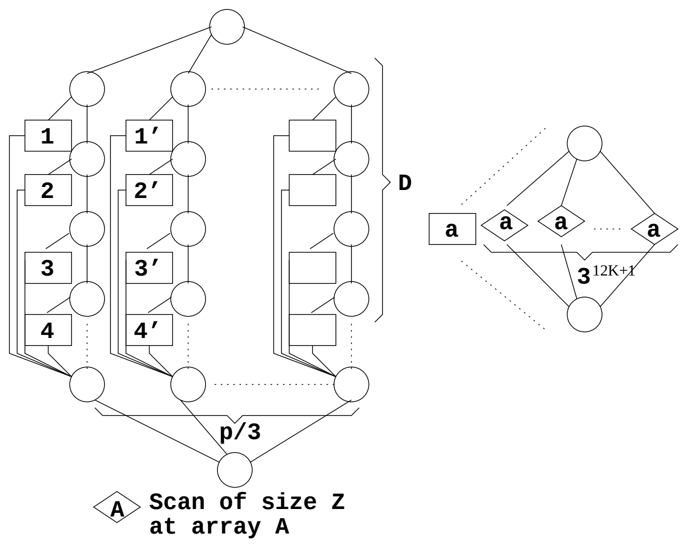
- *The number of steals is $O(p(D_P^{lat} + \log 1/\delta))$ with probability at least $1 - \delta$.*
- *All the caches at level i incur a total of less than $Q(Z_i, L_i) + O(p(D_P^{lat} + \log 1/\delta) Z_i/L_i)$ cache misses with probability at least $1 - \delta$.*
- *The computation completes in time not more than $W_P^{lat}/p + D_P^{lat} < W/p + O((DC_k + \log 1/\delta) C_k Z_k/L_k) + O(\sum_i C_i(Q(Z_{i-1}, L_{i-1}) - Q(Z_i, L_i)))/p$ with probability at least $1 - \delta$.*

Proof. Results from [9] imply the statement about the number of steals. Because the schedule involves not more than $O(p(D_P^{lat} + \log 1/\delta))$ steals with probability at least $1 - \delta$, all the caches at level i incur a total of at most $Q(Z_i, L_i) + O(p(D_P^{lat} + \log 1/\delta) Z_i/L_i)$ cache misses with probability at least $1 - \delta$. To compute the running time of the algorithm, we count the time spent by a processor waiting upon a cache miss towards the work and use the same proof as in [9]. In other words, we use the notion of latency-added work (W_P^{lat}) defined above. Because this is not more than $W + O(p(D_P^{lat} + \log 1/\delta) C_k Z_k/L_k + \sum_i C_i(Q(Z_{i-1}, L_{i-1}) - Q(Z_i, L_i)))$ with probability at least $1 - \delta$, the claim about the running time follows. □

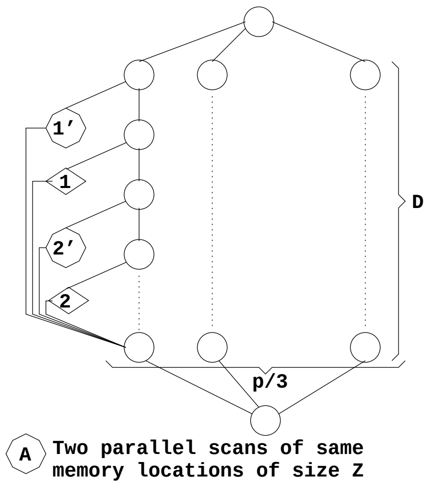
Thus, for constant δ, the parallel cache complexity at level i exceeds the sequential cache complexity by $O(pD_P^{lat} Z_i/L_i)$ with probability $1 - \delta$. This matches the bound in Equation 1 (Section 1) for a single level of private caches.

We can also consider a *centralized* work stealing scheduler. In *centralized work stealing*, processors deterministically steal a node of least depth in the DAG; this has been shown to be a good choice for reducing the number of steals [9]. The bounds in Theorem 5 carry over to centralized work stealing without the δ terms, e.g., the parallel cache complexity exceeds the sequential cache complexity by $O(pD_P^{lat} Z_i/L_i)$.

Theorem 6 (Lower Bound) *For a PMDH P with any given number of processors $p \geq 4$, cache sizes $Z_1 < \cdots < Z_k \leq Z/3$ for some a priori upper bound Z, cache line sizes $L_1 \leq \cdots \leq L_k$, and cache latencies $C_1 < \cdots < C_k$, and for any given depth $D' \geq 3(\log p + \log Z) + C_k + O(1)$, 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 $Q(Z_i, L_i)$ by $\Omega(pD_P^{lat} Z_i/L_i)$*



(a) Randomized work stealing



(b) Centralized work stealing

Figure 7: DAGs used in the lower bounds for randomized and centralized work stealing.

when scheduled using randomized work stealing. Such a computation DAG can also be constructed for centralized work stealing.

Proof. Such a construction is shown in Figure 7(a) for randomized work stealing. Based on the earlier lemma, we know that there exist a constant K such that the number of steals is at most KpD^{lat} with probability at least $1-(1/D^{lat})$. We construct the DAG such that it consists of a binary fanout to $p/3$ spines of length $D = D' - 2(K + \log(p/3) + \log Z)$ each. Each of the first $D/2$ nodes on the spine forks off a subdag that consists of $3^{(12K+1)}$ identical parallel scan structures of length Z each. A scan structure is a binary tree forking out to Z parallel nodes that collectively read a block of Z consecutive locations. The remaining $D/2$ nodes on the spine are the joins back to the spine of these forked off subdags. Note that $D_P^{lat} = D' + C_k$ because each path in the DAG contains at most one memory request.

For any Z_i and L_i, the sequential cache complexity $Q(Z_i, L_i) = (p/3)(Z_i + L + i + 3^{(12K+1)}(Z - Z_i + L_i))(D/2)/L_i$ because the sequential execution executes the subdags one by one and can reuse a scan segment of length Z_i/L_i for all the identical scans to avoid repeated misses on a set of locations. In other words, sequential execution gets $(p/3)(3^{(12K+1)} - 1)(Z_i - L_i))(D/2)/L_i$ cache hits because it executes identical scans one after the other.

We argue that in the case of randomized work stealing, there are a large number of subdags such that the probability that at least two scans from the subdag are repeated are executed by disjoint set of processors is greater than some positive constant. This implies that the cache complexity is $\Theta(pDZ_i/L_i)$ higher that the sequential cache complexity.

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 with get a spine node and hence a fresh subdag 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 $1/2$. This follows from the last observation.

4. If processor p stole a fresh subdag A and started the scans in it, the probability that work from the subdag 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 event that p currently chose a fresh subdag is not correlated with any event in the history, and therefore, with probability at least $1/2$, more than $p/2$ processors did not steal a fresh subdag in the latest steal. This means that these processors which stole a stale subdag (a subdag already being worked on by other processors) got less than $2/3$-rd fraction of the subdag 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. Since these steals are randomly distributed over all processors, there is a probability of at least $1/16$ that two of these steals where from p. Two steals from p would cause p to lose work from it's fresh subdag.

5. Since there are at most KpD steals with high probability, there no more $pD/6$ subdags which incur more than $12K$ steals. Among nodes with fewer than $12K$ steals, consider those described in the previous scenario where the processor p that started the subdag has work stolen from it before p executes $2/3$-rd of the subdag. At least $(1/16)(5pD/6)$ such subdags are expected in a run. Because there are 3^{12K+1} identical scans in each such subdag, at least one processor apart from p that has stolen work from this subdag gets to execute one complete scan. This means that the combined cache complexity at the i-th cache level for each subdag is at least Z_i/L_i greater than the sequential cache complexity, proving the lemma.

The construction for centralized work stealing is shown in Figure 7(b). The DAG ensures that the least-depth node at 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. $\square$

5.2.2 Implications for the Parallel Tree-of-Caches

The bounds in Theorem 5 hold for any Parallel Tree-of-Caches with intra-level regularity: Simply view the tree-of-caches as a PMDH by partitioning each proportionate cache shared by $p_{i,j}$ processors into $p_{i,j}$ disjoint caches of size Z_i.

5.2.3 Extending Shared Cache Results to Multiple Levels

Finally, we note that the bound in Equation 2 (Section 1) for a single level of cache shared by all the processors can be generalized to the *Parallel Multi-level Shared Hierarchy (PMSH)*, a special case of the PToC in which at each level i there is a cache of size Z_i shared by all the processors; see Figure 6(right):

Theorem 7 *When a cache-oblivious nested-parallel computation with sequential cache complexity $Q(Z, L)$, 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 $Q(Z_i - pL_iD_P^{lat}, L_i)$ cache misses. Moreover, the computation completes in time not more than $W_P^{lat}/p + D_P^{lat}$.*

Proof. The cache bound follows because (i) inclusion implies that hits/misses/evictions at levels $< i$ do not alter the number of misses at level i, (ii) 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 (iii) the key property of PDF schedulers, $Q_P(Z + pLD_P^{lat}, L) \leq Q_1(Z, L)$ (Equation 2), holds at each level i of a PMSH. The time bound follows because the schedule is greedy. □

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.

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Theory of Computing Systems*, pages 1–12, 2000.

[2] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 305–314, New York, NY, USA, 1987. ACM.

[3] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12:72–109, 1994.

[4] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *Proc. 1993 Conference 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 *Proceedings of the 34th ACM Symposium on Theory of Computing (STOC 2002)*, pages 268–276, Montréal, Québec, Canada, May 19–21 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 *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 197–206, New York, NY, USA, 2008. ACM.

[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 *In Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Puerto Vallarta*, pages 119–129, 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 *SPAA '07: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, 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] B. Blakeley and V. Ramachandran. Graph algorithms for multicores with multilevel caches. Manuscript, 2009.

[13] 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 *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 501–510, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[14] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 235–244, New York, NY, USA, 2004. ACM.

[15] 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.

[16] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 132–141, Honolulu, Hawaii, Apr. 1996.

[17] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999.

[18] G. S. Brodal. Cache-oblivious algorithms and data structures. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*, pages 3–13, 2004. LNCS, vol. 3111. Springer.

[19] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. 29th International Colloquium on Automata, Languages, and Programming*, pages 426–438, 2002. LNCS, vol. 2380. Springer.

[20] G. S. Brodal, R. Fagerberg, and G. Moruz. Cache-aware and cache-oblivious adaptive sorting. In *Proc. 32nd International Colloquium on Automata, Languages, and Programming*, pages 576–588, 2005. LNCS, vol. 3580. Springer.

[21] G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. *ACM Journal of Experimental Algorithmics*, 12, 2008. Article No. 2.2.

[22] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 139–149, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[23] R. A. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 71–80, New York, NY, USA, 2007. ACM.

[24] R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 207–216, New York, NY, USA, 2008. ACM.

[25] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 206–219, New York, NY, USA, 1986. ACM.

[26] E. D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive Data Sets*, Lecture Notes in Computer Science. Springer-Verlag, BRICS, University of Aarhus, Denmark, June 27–July 1 2002.

[27] G. Franceschini. Proximity mergesort: Optimal in-place sorting in the cache-oblivious model. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms*, pages 291–299, 2004.

[28] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the Association for Computing Machinery*, 17(3):496–507, 1970.

[29] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, New York, Oct. 1999.

[30] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 271–280, New York, NY, USA, 2006. ACM.

[31] Intel multi-core technology. www.intel.com/multi-core/, 2009.

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

[33] P. Kumar. Cache oblivious algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*, pages 193–212. Springer, 2003.

[34] E. Ladan-Mozes and C. E. Leiserson. A consistency architecture for hierarchical shared caches. In *Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 11–22, 2008.

[35] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36:177–189, 1979.

[36] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989.

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

[38] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[39] L. G. Valiant. A bridging model for multicore computing. In *Proc. 16th European Symposium on Algorithms*, pages 13–28, 2008.

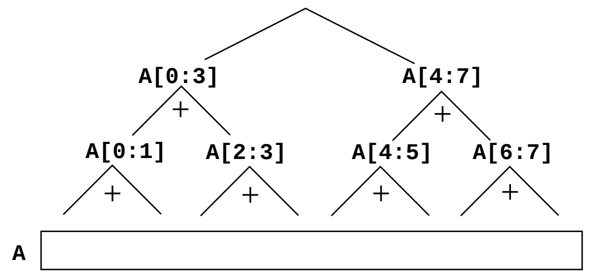


Figure 8: Prefix Sums. The first phase involves recursively summing up the elements and storing the intermediate values.

A Appendix

A.1 Prefix Sums

Consider a balanced binary tree laid over the input (see Figure 8). The algorithm works in two phases that traverse the tree: the first calculates for each node the sum of the left subtree and the second pushes values down the tree based on these partial sums. The recursive codes for the two phases are shown in Algorithms 1 and 2, respectively. The code uses an input array A of length n, a temporary array T of length $n-1$, and an output array R. The only adaptation to the standard prefix sums algorithm is that we lay out the tree in infix order in the array T, so as to make the algorithm cache-efficient. The down phase basically takes the input s, passes it to the left call and passes $s \oplus root(T)$ to the right call. To compute prefix-sum of A, we call the function UP$(A, T, 0, n)$ followed by DOWN$(R, T, 0, n, 0)$.

Algorithm 1 UP(A, T, i, n)

```
if $n = 1$ then
    return $A[i]$
else
    $k \leftarrow i + n/2$
    $T[k] \leftarrow$ UP$(A, T, i, n/2)$
    right $\leftarrow$ UP$(A, T, k, n/2)$
    return $T[k] \oplus$ right
end if
```

Algorithm 2 DOWN(R, T, i, n, s)

```
if $n = 1$ then
    $R[i] \leftarrow s$
else
    $k \leftarrow i + n/2$
    DOWN$(R, T, i, n/2, s)$
    DOWN$(R, T, k, n/2, s \oplus T[k])$;
end if
```

Lemma 2 *The prefix sum of an array of length n can be computed in $W(n) = O(n)$, $D(n) = O(\log n)$ and $Q(n; Z, L) = O(\lceil n/L \rceil)$.*

Proof. There exists a positive constant c such that for a call to the function UP with $n \leq cZ$, all the memory locations accessed by the function fit into the cache. Thus, $Q(n; Z, L) \leq Z/L$ for $n \leq cZ$. Also $Q(n; Z, L) = O(1) + 2 \cdot Q(n/2; Z, L)$ for $n > cZ$. Therefore, the sequential cache complexity for UP is $O(n/Z + (n/(cZ)) \cdot (Z/L)) = O(\lceil n/L \rceil)$. A similar analysis for the cache complexity of DOWN shows that $Q(n; Z, L) = O(\lceil n/L \rceil)$. Both the algorithms have depth $O(\log n)$ because the recursive calls can be made in parallel. □

We note that the temporary array T can also be stored in preorder or postorder with the same bounds, but that the index calculations are then a bit more complicated. What matters is that any subtree is contiguous in the array. The standard heap order (level order) does not give the same result.

A.2 Depth of randomized sorting algorithm

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 probability at each level we need to execute the outer loop more times toward the leaves where the 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 $1 - 1/m$.

To prove the required bounds on depth, we prove that the sum of depths of all nodes at level d in the recursion tree is at most $O(\log^{3/2} n / \log \log n)$ with probability at least $1 - 1/n^{O(\log^2 \log n)}$ and that the recursion tree is at most $1.1 \log_2 \log_2 n$ levels deep. This will prove that the depth of the recursion tree is $O(\log^{3/2} n)$ with probability at least $1 - (1.1 \log_2 \log_2 n)/n^{O(2 \log^2 \log n)}$. Since the actual depth is a maximum over all "critical" paths which we will argue are not more than $n^{O(1.1 \log \log n)}$ in number (critical paths are shaped liked the recursion tree), we can conclude that the depth is $O(\log^{3/2} n)$ with high probability.

The maximum number of levels in the recursion tree can be bounded using the recurrence relation $X(n) = 1 + X(\sqrt{n} \log n)$ and $X(10) = 1$. Using induction, it is straightforward to show that this solves to $X(n) < 1.1 \log_2 \log_2 n$. Similarly the number of critical paths $C(n)$ can be bounded using the relation $C(n) < (\sqrt{n} X(\sqrt{n}))(\sqrt{n} X(\sqrt{n} \log n))$. Again, using induction, this relation can be used to show that $C(n) = n^{O(1.1 \log \log n)}$.

To compute the sum of the depth of nodes at level d in the recursion tree, we consider two cases: (1) when $d > 10 \log \log \log n$ and (2) otherwise.

Case 1: The size of a node one level deep in the recursion tree is at most $O(\sqrt{n} \log n) = O(n^{1/2+r})$ for any $r > 0$. Also, the size of a node which is d levels deep is at most $O(n^{(1/2+r)^d})$, each costing $O((1/2 + r)^d \log n)$ depth per trial. Since there are 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 $(2^d \cdot \log^{1/2} n)/((1 + 2r)^d \cdot \log \log n)$ loops is small. Since we are estimating the sum of 2^d independent variables, we use Chernoff bounds of the form:

$$Pr[X > (1 + \delta)\mu] \leq e^{-\delta^2 \mu}, \tag{5}$$

with $\mu = (2 \cdot 2^d)$, $\delta = (1/2)(\log^{1/2} n/((1 + 2r)^d \cdot \log \log n)) - 1$. The resulting probability bound is asymptotically lesser than $1/n^{O(\log^2 \log n)}$ for $d > 10 \log \log \log n$. Therefore, the contribution of nodes at level d in the recursion tree to the depth of recursion tree is at most $2^d \cdot (1/2 + r)^d \log n \cdot \log^{1/2} n/((1 + 2r)^d \cdot \log \log n) = \log^{3/2} n / \log \log n$ with probability at least $1 - 1/n^{O(\log^2 \log n)}$.

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 $2^d < (\log \log n)^5$. Consider the

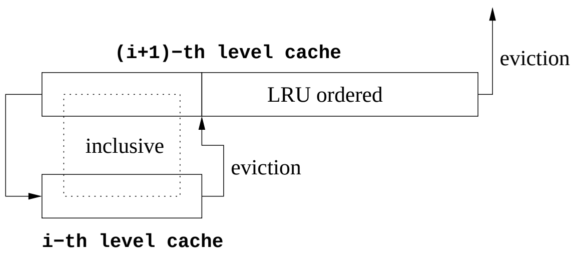


Figure 9: Multi-level LRU, depicting a parent at level $i+1$ and one of its children at level i

small nodes. Each small node can contribute a depth of at most $2 \log \log n$ to the recursion tree and there are at most $(\log \log n)^1 0$ of them. Therefore, their contribution to depth of the recursion tree at level d is asymptotically lesser than $\log n$.

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 $1/n^{O(\log^2 \log n)}$, 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 in the form presented above with the above μ, and $\delta = (0.9 \cdot \log^2 n - 1)$, we infer that this probability is asymptotically less than $1/n^{O(\log^2 \log n)}$. Since $j < 2^d$, the depth contributed by the larger nodes is at most $2^d(1/2 + r)^d \log n$, asymptotically smaller than $\log^{3/2} n/\log \log n$.

The above analysis shows that the combined depth of all nodes at level d is $O(\log^{3/2} n/\log \log n)$ with high probability. Since there are only $1.1 \log_2 \log_2 n$ levels in the recursion tree, this completes the proof.

A.3 LRU replacement

Thus far, we have assumed that each cache in a Parallel Tree-of-Caches uses an optimal replacement policy. In this section, we consider instead a more practical LRU replacement policy.

Frigo *et al.* [29] motivate the use of an optimal replacement policy in the cache-oblivious model by appealing to the fact that LRU and optimal differ by at most a constant factor. Namely, Sleator and Tarjan [38] have shown that a cache of size $2Z$ using LRU incurs at most twice as many misses as a cache of size Z using optimal. This result extends to each cache of the PToC as well.

Alternatively, we can directly consider a PToC in which each cache uses a multi-level inclusive LRU replacement policy. Assuming that a cache line evicted at level i is sent to its parent at level $i+1$ and that any access to a memory location not at a level i cache is serviced by passing it from higher levels in the memory hierarchy through its parent cache at level $i+1$, the cache at level $i+1$ knows exactly what elements are contained in its children caches at level i. From the order in which cache lines were evicted by level i, a cache at level $i+1$ can fill up the rest of its slots and order them in the LRU order. See Figure 9.