Scheduling Irregular Parallel Computations on Hierarchical Caches

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Jeremy T. Fineman
Carnegie Mellon University
jfineman@cs.cmu.edu

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

Harsha Vardhan Simhadri
Carnegie Mellon University
harshas@cs.cmu.edu

ABSTRACT

For nested-parallel computations with low depth (span, critical path length) analyzing the work, depth, and *sequential* cache complexity suffices to attain reasonably strong bounds on the *parallel* runtime and cache complexity on machine models with either shared or private caches. These bounds, however, do not extend to general hierarchical caches, due to limitations in (i) the cache-oblivious (CO) model used to analyze cache complexity and (ii) the schedulers used to map computation tasks to processors. This paper presents the *parallel cache-oblivious (PCO)* model, a relatively simple modification to the CO model that can be used to account for costs on a broad range of cache hierarchies. The first change is to avoid capturing artificial data sharing among parallel threads, and the second is to account for parallelism-memory imbalances within tasks. Despite the more restrictive nature of PCO compared to CO, many algorithms have the same asymptotic cache complexity bounds.

The paper then describes a new scheduler for hierarchical caches, which extends recent work on "space-bounded schedulers" to allow for computations with arbitrary *work imbalance* among parallel subtasks. This scheduler attains provably good cache performance and runtime on parallel machine models with hierarchical caches, for nested-parallel computations analyzed using the PCO model. We show that under reasonable assumptions our scheduler is "work efficient" in the sense that the cost of the cache misses are evenly balanced across the processors—*i.e.*, the runtime can be determined within a constant factor by taking the total cost of the cache misses analyzed for a computation and dividing it by the number of processors. In contrast, to further support our model, we show that no scheduler can achieve such bounds (optimizing for both cache misses and runtime) if work, depth, and sequential cache complexity are the only parameters used to analyze a computation.

Categories and Subject Descriptors

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

General Terms

Algorithms, Theory

Keywords

Parallel hierarchical memory, Cost models, Schedulers, Analysis of parallel algorithms, Cache complexity

1. INTRODUCTION

Because of limited bandwidths on real parallel machines locality can be critical to achieving good performance for parallel programs. To account for this in the design of algorithms, many locality-aware parallel models have been suggested [2, 6, 18, 19, 23, 24]. This work has contributed significantly to our understanding of locality in parallel algorithms.

With the advent of multicores most computer users have a parallel machine on their desk or lap, and these are all based on a multi-level cache hierarchy with a 50–200X factor difference between the access time to the first level cache and main memory (whether used sequentially or in parallel). Fig. 1 shows, for example, the memory hierarchies for the current generation desktop/servers from Intel, AMD, and IBM. Correspondingly there has been significant recent work on parallel cache based locality [1, 4, 5, 7, 9, 10, 12–14, 21, 25]. The work has fallen into two main classes. One class involves designing algorithms directly for the machine and having the algorithm designer explicitly allocate tasks to processors. This includes the work by Arge et al. [4] on designing algorithms directly for a p-processor machine with one layer of private caches (the PEM), and by Valiant [25] on algorithms for a hierarchical cache with unit-size cache lines (the Multi-BSP). The other class involves dynamic parallelism in which the algorithm designer specifies the full parallelism of the computation, typically much more than is available on a real machine, and analyzes the cache cost in an abstract cost model not directly corresponding to a parallel machine. A scheduler is then responsible for dynamically mapping the computation onto the processors in a manner that bounds the cost as a function of the analyzed costs. Dynamic parallelism has important advantages, including being much simpler, potentially machine independent, and much closer to how users actually code on these machines using languages such as OpenMP, Cilk++, Intel TBB, and the Microsoft Task Parallel Library. However, the abstraction makes it harder to achieve good performance.

A pair of common abstract measures for capturing parallel cache based locality are the number of misses given a sequential ordering of a parallel computation [1, 9, 10, 21], and the depth (span, critical path length) of the computation. The cache-oblivious (CO) model (a.k.a., the ideal cache model) [20] can be used for analyzing the misses in the sequential ordering, giving a cache complexity $Q(n; M, B)$ where n is the size of the problem, M is a single cache size and B a single block size. One can show, for example, that any nested-parallel computation with sequential cache complexity Q and depth D will cause at most $Q + O(pDM/B)$ total misses when run with an appropriate scheduler on p processors, each with a private cache of size M and block size B [1]. Unfortunately, current dynamic parallelism approaches have important limitations: they either apply to hierarchies of only private or only shared caches [1,9,10,16,21], require some strict balance criteria [7,15], or require a joint algorithm/scheduler analysis [7,13–16].

In this paper we present a model and a scheduler that ***enable an algorithm analysis that is independent of both the parallel machine and the scheduler, allow for irregular computations with arbitrary imbalance among tasks, and work on hierarchies of shared and private caches*** (as in Fig. 1). The approach is limited to nested-parallel computations, but this includes a very broad set of algorithms, including most divide-and-conquer, data-parallel, and CREW PRAM-like algorithms.

The approach is based on three components. The first is a cache cost model (the ***Parallel Cache-Oblivious (PCO)*** model). As with the standard CO model the cache cost is derived in terms of a single cache size M and a single block size B giving a cache complexity $Q^*(n; M, B)$ independent of the number of processors. The model for a sequential strand of computation remains the same. When a task t forks a set of child tasks, however, the child tasks start with the same cache state; this contrasts with the standard CO analysis based on a sequential ordering of the child tasks. In particular if t fits in M all child tasks start with the cache state of the parent at the fork point, and at the join point the union of their locations are included in the cache state of the parent. If the task does not fit in M then the cache state is emptied at the fork and join points. This model ignores (incidental) data reuse among parallel subcomputations and accounts for reuse only when there is a serial relationship between instructions accessing the same data. As we show, this enables tighter bounds when mapping computations onto, for example, shared caches. For the same M and B the cache cost in the PCO model may be higher than in the CO model. For a variety of fundamental parallel algorithms, however, including quicksort, sample sort, matrix multiplication, matrix inversion, sparse-matrix multiplication, and convex hulls, the asymptotic bounds are not affected, while the higher baseline enables a provably efficient mapping to parallel hierarchies for arbitrary nested-parallel computations.

The second is a new cost metric that penalizes large imbalance in the ratio of space to parallelism in subtasks. We present a lower bound that indicates that some form of parallelism-space imbalance penalty is required. Intuitively this is because on any given parallel memory hierarchy as depicted in Fig. 1, the cache resources are linked to the processing resources: each cache is shared by a fixed number of processors. Therefore any large imbalance between space and processor requirements will require either processors to be under-utilized or caches to be over-subscribed. As in the basic PCO model, the cost $\widehat{Q}_\alpha(n; M, B)$ for inputs of size n is asymptotically equal to that of the standard sequential cache cost $Q(n; M, B)$ for many problems.

The third is a new "space-bounded scheduler" that extends recent work of Chowdhury et al. [14]. A space-bounded scheduler accepts dynamically parallel programs that have been annotated with space requirements for each recursive subcomputation called a "task." These schedulers run every task in a cache that just fits it (*i.e.*, no lower cache will fit it), and once assigned, tasks are not migrated across caches. We show that any space-bounded scheduler guarantees that the number of misses across all caches at each level i of the machine's hierarchy is at most $Q^*(n; M_i, B_i)$, where $Q^*(n; M_i, B_i)$ is the cost in the basic PCO model with problem size n, cache size M_i, and cache-line size B_i.

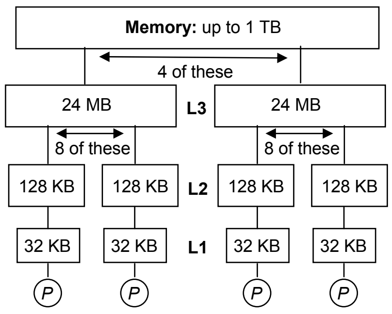
In contrast to previous work, we describe a space-bounded scheduler that allows parallel subtasks to be scheduled on different levels in the memory hierarchy, thus allowing significant imbalance in the sizes of tasks. Furthermore, we show that our space-bounded scheduler achieves efficient total running time, as long as the parallelism of the machine is sufficient with respect to the parallelism of the algorithm. Specifically, we show that our scheduler executes a cache-oblivious computation on a homogeneous h-level parallel memory hierarchy having p processors in time:

$$O\left(\frac{v_h \sum_{i=0}^{h} \widehat{Q}_\alpha(n; M_i, B) \cdot C_i}{p}\right),$$

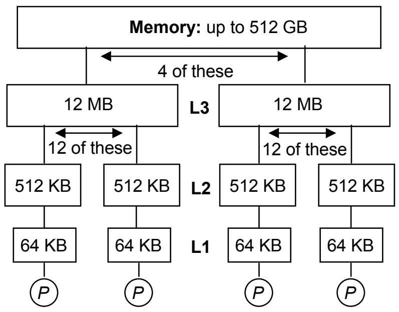
where M_i is the size of each level-i cache, B is the uniform cache-line size, C_i is the cost of a level-i cache miss, and v_h is an overhead defined in Theorem 6. For any algorithms where $\widehat{Q}_\alpha(n; M, B)$ is asymptotically equal to the optimal sequential cache-oblivious cost $Q(n; M, B)$ for the problem, and under conditions where v_h is constant, this is optimal across all levels of the cache. For example, a parallel sample sort (that uses imbalanced subtasks) gives $\widehat{Q}_\alpha(n; M, B) = O((n/B) \log_M(n/B))$, which matches the optimal sequential cache complexity for sorting, implying optimality on parallel cache hierarchies using our scheduler.

2. PRELIMINARIES

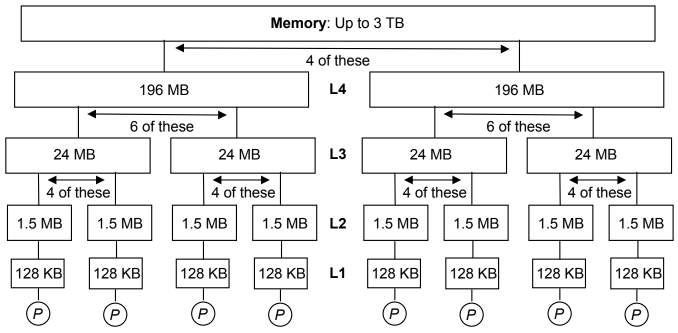
Computation Model. As in most of the prior work cited in Section 1, this paper considers algorithms with nested parallelism, allowing arbitrary dynamic nesting of parallel loops and fork-join constructs but no other synchronizations. This corresponds to the class of algorithms with series-parallel dependence graphs (see Fig. 2). Computations can be decomposed into "tasks", "parallel blocks" and "strands" recursively as follows. As a base case, a ***strand*** is a serial sequence of instructions not containing any parallel constructs or subtasks. A ***task*** is formed by serially composing $k \geq 1$ strands interleaved with $(k-1)$ "parallel blocks" (denoted by $\mathsf{t} = \mathsf{s}_1; \mathsf{b}_1; \ldots; \mathsf{s}_k$). A ***parallel block*** is formed by composing



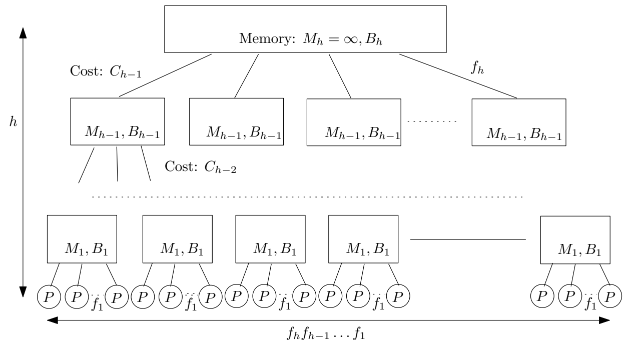
(a) 32-processor Intel Xeon 7500



(b) 48-processor AMD Opteron 6100



(c) 96-processor IBM z196



(d) PMH model of [3]

Figure 1: Memory hierarchies of current generation architectures from Intel, AMD, and IBM, plus an example abstract parallel hierarchy model. Each cache (rectangle) is shared by all processors (circles) in its subtree.

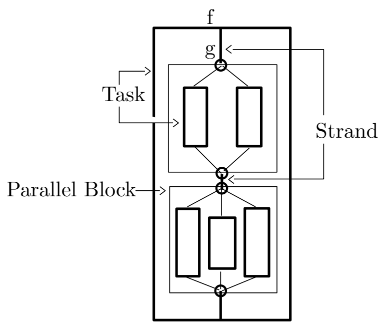


Figure 2: Decomposing the computation: tasks, strands and parallel blocks

in parallel one or more tasks with a fork point before all of them and a join point after (denoted by $\mathsf{b} = \mathsf{t}_1 \| \mathsf{t}_2 \| \dots \| \mathsf{t}_k$). A parallel block can be, for example, a parallel loop or some constant number of recursive calls. The top-level computation is a task. The ***span*** (a.k.a., ***depth***) of a computation is the length of the longest path in the dependence graph.

The nested-parallel model assumes all strands share a single memory. We say two strands are ***concurrent*** if they are not ordered in the dependence graph. Concurrent reads (*i.e.*, concurrent strands reading the same memory location) are permitted, but not data races (*i.e.*, concurrent strands that read or write the same location with at least one write).

Machine Model: The Parallel Memory Hierarchy model. Following prior work addressing multi-level parallel hierarchies [3,7,10,12–14,25], we model parallel machines using a tree-of-caches abstraction. For concreteness, we use a symmetric variant of the parallel memory hierarchy (PMH) model [3] (see Fig. 1(d)), which is consistent with many other models [7, 10, 12–14]. A PMH consists of a height-h tree of memory units, called ***caches***. We assume that each cache is an ideal cache. The leaves of the tree are at level-0 and any internal node has level one greater than its children. The leaves (level-0 nodes) are processors, and the level-h root corresponds to an infinitely large main memory. We do not assume inclusive caches, meaning that a memory location may be stored in a low-level cache without being stored at all ancestor caches. We can extend the model to support inclusive caches, but then we must assume larger cache sizes to accommodate the inclusion.

Each level in the tree is parameterized by four parameters: M_i, B_i, C_i, and f_i. We denote the ***capacity*** of each level-i cache by M_i. Memory transfers between a cache and its child occur at the granularity of ***cache lines***. We use $B_i \geq 1$ to denote the ***line size*** of a level-i cache, or the size of contiguous data transferred from a level-$(i+1)$ cache to its level-i child. If a processor accesses data that is not resident in its level-1 cache, a level-1 ***cache miss*** occurs. More generally, a ***level-$(i+1)$ cache miss*** occurs whenever a level-i cache miss occurs and the requested line is not resident in the parent level-$(i+1)$ cache; once the data becomes resident in the level-$(i+1)$ cache, a level-i cache request may be serviced by loading the size-B_{i+1} line into the level-i cache. The cost of a level-i cache miss is denoted by $C_i \geq 1$, where this cost represents the amount of time to load the corresponding line into the level-i cache under full load. Thus, C_i models both the latency and the bandwidth constraints

Task t forks subtasks t_1 and t_2,
with $\kappa = \{l_1, l_2, l_3\}$

t_1 accesses l_1, l_4, l_5 incurring 2 misses
t_2 accesses l_2, l_4, l_6 incurring 2 misses

At the join point: $\kappa' = \{l_1, l_2, l_3, l_4, l_5, l_6\}$

Figure 3: Example applying the PCO model (Definition 2) to a parallel block. Here, $Q^*(\mathsf{t}; M, B; \kappa) = 4$.

of the system (whichever is worse under full load). The cost of an access at a processor that misses at all levels up to and including level-j is thus $C'_j = \sum_{i=0}^{j} C_i$. We use $f_i \geq 1$ to denote the number of level-$(i-1)$ caches below a single level-i cache, also called the ***fanout***. As in [1], we assume the model maintains DAG consistent shared memory with the BACKER algorithm [11]. This is a weak consistency model and assumes that cache lines are merged on writing back to memory thus avoiding "false sharing" issues.

We assume that the number of lines in any nonleaf cache is greater than the sums of the number of lines in all its immediate children, *i.e.*, $M_i/B_i \geq f_i M_{i-1}/B_{i-1}$ for $1 < i \leq h$, and $M_1/B_1 \geq f_1$. The miss cost C_h and line size B_h are not defined for the root of the tree as there is no level-$(h+1)$ cache. The leaves (processors) have no capacity ($M_0 = 0$), and they have $B_0 = C_0 = 1$. Also, $B_i \geq B_{i-1}$ for $0 < i < h$. Finally, we call the entire subtree rooted at a level-i cache a ***level-i cluster***, and we call its child level-$(i-1)$ clusters ***subclusters***. We use $p_i = \prod_{j=1}^{i} f_j$ to denote the total number of processors in a level-i cluster.

3. THE PCO MODEL

In this section, we present the Parallel Cache-Oblivious model, a simple, high-level model for algorithm analysis. As in the sequential cache-oblivious (CO) model [20], in the ***Parallel Cache-Oblivious (PCO) model*** there is a memory of unbounded size and a single cache with size M, line-size B (in words), and optimal (i.e., furthest into the future) replacement policy. The cache state κ consists of the set of cache lines resident in the cache at a given time. When a location in a non-resident line l is accessed and the cache is full, l replaces in κ the line accessed furthest into the future, incurring a *cache miss*.

To extend the CO model to parallel computations, one needs to define how to analyze the number of cache misses during execution of a parallel block. Analyzing using a sequential ordering of the subtasks in a parallel block (as in most prior work[1]) is problematic for mapping to even a single shared cache, as the following theorem demonstrates for the CO model:

THEOREM 1. *Consider a PMH comprised of a single cache shared by $p > 1$ processors, with cache-line size B, cache size $M \geq pB$, and a memory (i.e., $h = 2$). Then there exists a parallel block such that for any greedy scheduler[2] the number of cache misses is nearly a factor of p larger than the cache complexity on the CO model.*

PROOF. Consider a parallel block that forks off p identical tasks, each consisting of a strand reading the same set of M memory locations from M/B blocks. In the CO model, after the first M/B misses, all other accesses are hits, yielding a total cost of M/B misses in the CO model.

Any greedy schedule on p processors executes all strands at the same time, incurring simultaneous cache misses (for the same line) on each processor. Thus, the parallel block incurs $p(M/B)$ misses. ☐

The gap arises because a sequential ordering accounts for significant reuse among the subtasks in the block, but a parallel execution cannot exploit reuse unless the line has been loaded earlier.

To overcome this difficulty, we instead use an approach of (i) ignoring any data reuse among the subtasks and (ii) flushing the cache at each fork and join point of any task that does not fit within the cache, as follows. Let $loc(\mathsf{t}; B)$ denote the set of distinct cache lines accessed by task t, and $S(\mathsf{t}; B) = |loc(\mathsf{t}; B)| \cdot B$ denote its size (also let $s(\mathsf{t}; B) = |loc(\mathsf{t}; B)|$ denote the size in terms of number of cache lines). Let $Q(\mathsf{c}; M, B; \kappa)$ be the cache complexity of c in the sequential CO model when starting with cache state κ.

DEFINITION 2. *[Parallel Cache-Oblivious Model] For cache parameters M and B the* ***cache complexity*** *of a strand* s, *parallel block* b, *or task* t *starting at state κ is defined as:*
strand:

$$Q^*(\mathsf{s}; M, B; \kappa) = Q(\mathsf{s}; M, B; \kappa)$$

parallel block: *For* $\mathsf{b} = \mathsf{t}_1 \| \mathsf{t}_2 \| \ldots \| \mathsf{t}_k$,

$$Q^*(\mathsf{b}; M, B; \kappa) = \sum_{i=1}^{k} Q^*(\mathsf{t}_i; M, B; \kappa)$$

task: *For* $\mathsf{t} = \mathsf{c}_1; \mathsf{c}_2; \ldots; \mathsf{c}_k$,

$$Q^*(\mathsf{t}; M, B; \kappa) = \sum_{i=1}^{k} Q^*(\mathsf{c}_i; M, B; \kappa_{i-1}) \ ,$$

where $\kappa_i = \emptyset$ if $S(\mathsf{t}; B) > M$, and $\kappa_i = \kappa \cup_{j=1}^{i} loc(\mathsf{c}_j; B)$ if $S(\mathsf{t}; B) \leq M$.

We use $Q^*(\mathsf{c}; M, B)$ to denote a computation c starting with an empty cache, $Q^*(n; M, B)$ when n is a parameter of the computation, and $Q^*(\mathsf{c}; 0, 1)$ to denote the computational work. Note that by setting M to 0, we force the analysis to count every instruction that touches even a register and hence effectively corresponds to instruction count.

Comments on the definition: Since a task t alternates between strands and parallel blocks the definition effectively clears the cache at every fork and join point in t when $S(\mathsf{t}; B) > M$. This is perhaps more conservative than required but leads to a simple model and does not seem to affect bounds. Since in a parallel block all subtasks start with the same cache state, no sharing is assumed among parallel blocks. If an algorithms wants to share a value loaded from memory, then the load should occur before the fork. The notion of furthest in the future for Q in a strand might seem ill-defined since the future might entail parallel tasks. However, all future references fit into cache until reaching a supertask that does not fit in cache, at which point the

[1]Two prior works not using the sequential ordering are the *concurrent cache-oblivious model* [5] and the *ideal distributed cache model* [21], but both design directly for p processors and consider only a single level of private caches.

[2]In a *greedy* scheduler, a processor remains idle only if there is no ready-to-execute task.

Problem	Span	Cache Complexity Q^*
Scan (prefix sums, etc.)	$O(\log n)$	$O(\lceil n/B \rceil)$
Matrix Transpose ($n \times m$ matrix) [20]	$O(\log(n+m))$	$O(\lceil nm/B \rceil)$
Matrix Multiplication ($\sqrt{n} \times \sqrt{n}$ matrix) [20]	$O(\sqrt{n})$	$O(\lceil n^{1.5}/B \rceil/\sqrt{M+1})$
Matrix Inversion ($\sqrt{n} \times \sqrt{n}$ matrix)	$O(\sqrt{n})$	$O(\lceil n^{1.5}/B \rceil/\sqrt{M+1})$
Quicksort [22]	$O(\log^2 n)$	$O(\lceil n/B \rceil(1+\log\lceil n/(M+1) \rceil))$
Sample Sort [10]	$O(\log^2 n)$	$O(\lceil n/B \rceil \lceil \log_{M+2} n \rceil)$
Sparse-Matrix Vector Multiply [10] (m nonzeros, n^ϵ edge separators)	$O(\log^2 n)$	$O(\lceil m/B + n/(M+1)^{1-\epsilon} \rceil)$
Convex Hull (e.g., see [8])	$O(\log^2 n)$	$O(\lceil n/B \rceil \lceil \log_{M+2} n \rceil)$
Barnes Hut tree (e.g., see [8])	$O(\log^2 n)$	$O(\lceil n/B \rceil(1+\log\lceil n/(M+1) \rceil))$

Table 1: Cache complexities of some algorithms analyzed in the PCO model. The bounds assume $M = \Omega(B^2)$. All algorithms are work optimal and their cache complexities match the best sequential algorithms.

cache is assumed to be flushed. Thus, there is no need to choose cache lines to evict. For a single strand the model is equivalent to the cache-oblivious model.

We believe that the PCO model is a simple, effective model for the cache analysis of parallel algorithms. It retains much of the simplicity of the ideal cache model, such as analyzing using only one level of cache. It ignores the complexities of artificial locality among parallel subtasks. Thus, it is relatively easy to analyze algorithms in the PCO model (examples are given in Section 4). Moreover, as we will show in Section 5, PCO bounds optimally map to cache miss bounds on each level of a PMH. Finally, although the PCO bounds are upper bounds, for many fundamental algorithms, they are tight: they asymptotically match the bounds given by the sequential ideal cache model, which are asymptotically optimal. Table 1 presents the PCO cache complexity of a few such algorithms, including both algorithms with polynomial span (matrix inversion) and highly imbalanced algorithms (the block transpose used in sample sort).

4. EXAMPLE PCO ANALYSIS

It is relatively easy to analyze algorithms in the PCO model. Let us consider first a simple map over an array which touches each element by recursively splitting the array in half until reaching a single element. If the algorithm for performing the map does not touch any array elements until recursing down to a single element, then each recursive task begins with an empty cache state, and hence the cache performance is $Q^*(n; M, B) = n$. An efficient implementation would instead load the middle element of the array before recursing, thus guaranteeing that a size-$\Theta(B)$ recursive subcomputation begins with a cache state containing the relevant line. We thus have the recurrence

$$Q^*(n; M, B) \quad = \quad \begin{cases} 2Q^*(\frac{n}{2}; M, B) + O(1) & n > B \\ O(1) & n \leq B \end{cases},$$

which implies $Q^*(n; M, B) = O(n/B)$, matching the sequential cache complexity.

Quicksort is another algorithm that is easy to analyze in this model. A standard quicksort analysis for work [17] observes that all work can be amortized against comparisons of keys with a pivot. The probability of comparing keys of rank i and $j > i$ is at most $2/(j-i)$, *i.e.*, the probability of selecting i or j as a pivot before any element in between. The expected work is thus $\sum_{i=1}^{n} \sum_{j>i}^{n} 2/(j-i) = \Theta(n \log n)$. Extending this analysis to either the CO or the PCO models is identical—comparisons become free once the corresponding subarray fits in memory. Specifically, for nearby keys i and $j < i + M/3$, no paid comparison occurs if a key between $i - M/3$ and $i-1$ is chosen before i and if $j+1$ to $j + M/3$ is chosen before j. Summing over all keys gives expected number of paid comparisons $\sum_{i=1}^{n} \sum_{j>i+M/3}^{n} 2/(j-1) + \sum_{i=1}^{n} \sum_{j<i+M/3} 6/M = \Theta(n \log\lceil n/(M+1) \rceil + n)$. Completing the analysis (dividing this cost by B) entails observing that each recursive quicksort scans the subarray in order, and thus whenever a comparison causes a cache miss, we can charge $\Theta(B)$ comparisons against the same cache miss.

The rest of the algorithms in Table 1 can be similarly analyzed without difficulty, observing that for the original CO analyses, the cache complexities of the parallel subtasks were already analyzed independently assuming no data reuse.

5. BASIC SPACE-BOUNDED SCHEDULER

In this section we describe a class of schedulers, called space-bounded schedulers, and show (Theorem 3) that such schedulers have cache complexity on the PMH machine model that matches the PCO cache complexity. Space-bounded schedulers were introduced by Chowdhury et al. [14], but their paper does not use the PCO model and hence cannot show the same kind of optimality as Theorem 3. This section briefly describes a "greedy-space-bounded" scheduler that performs very well in terms of runtime on very balanced computations, and uses it to highlight some of the difficulties in designing a scheduler (such as the one in Section 7) that permits imbalance.

Space-Bounded Schedulers. A "space-bounded scheduler" is parameterized by a global ***dilation*** parameter $0 < \sigma \leq 1$ and machine parameters $\{M_i, B_i, C_i, f_i\}$. Given these parameters, we define a ***level-i task*** to be a task that fits within a σ fraction of the level-i cache, but not within a σ fraction of the level-$(i-1)$ cache, *i.e.*, $S(\mathsf{t}; B_i) \leq \sigma M_i$ and $S(\mathsf{t}; B_{i-1}) > \sigma M_{i-1}$. We call t a ***maximal level-i task*** if it is a level-i task but its parent (*i.e.*, minimal containing) task is not. The top level task (no parent) is considered maximal. We call a strand a ***level-i strand*** if its minimal containing task is a level-i task.

A ***space-bounded*** scheduler [14] is one that limits the migration of tasks across caches and the number of outstanding subtasks as follows. Consider any level-i task t. Once any of t is executed by some processor below level-i cache U_i, all remaining strands of t must be executed by the same level-i cluster. We say that t is ***anchored*** at U_i. Moreover, at any point in time, consider the maximal level-i tasks $\mathsf{t}_1, \mathsf{t}_2, \ldots, \mathsf{t}_k$ anchored to level-i cache U_i. Then $\sum_{j=1}^{k} S(\mathsf{t}_j; B_i) \leq M_i$. That is to say, the total space used by tasks anchored to U_i does not exceed U_i's capacity. Finally, we consider strands. Whereas a task is anchored to a single cache, a level-i strand is anchored to caches along a level-i to level-1 path in the memory hierarchy. When a level-i strand is anchored to a level-$j < i$ cache, it is treated as a task that takes σM_j space, thereby preventing (many) other tasks/strands from being anchored at the same cache.

We relax the usual definition of greedy scheduler in the following: A ***greedy-space-bounded scheduler*** is a space-bounded scheduler in which a processor remains idle only if there is no ready-to-execute strand that can be anchored to the processor (and appropriate ancestor caches) without violating the space-bounded constraints.

Cache Bounds: PCO Cache Complexity is Optimal For Space-Bounded Schedulers. The following theorem implies that a nested-parallel computation scheduled with any space-bounded scheduler achieves optimal cache performance, with respect to the PCO model. A main idea of the proof is that each task reserves sufficient cache space and hence never needs to evict a previously loaded cache line.

THEOREM 3. *Consider a PMH and any dilation parameter $0 < \sigma \leq 1$. Let* t *be a level-i task. Then for all memory-hierarchy levels $j \leq i$, the number of level-j cache misses incurred by executing* t *with any space-bounded scheduler is at most $Q^*(\mathsf{t}; \sigma M_j, B_j)$.*

PROOF. Let U_i be the level-i cache to which t is assigned. Observe that t uses space at most σM_i. Moreover, by definition of the space-bounded scheduler, the total space needed for tasks assigned to U_i is at most M_i, and hence no line from t need ever be evicted from U's level-i cache. Thus, an instruction x in t accessing a line ℓ does not exhibit a level-i cache miss if there is an earlier-executing instruction in t that also accesses ℓ. Any instruction serially preceding x must execute earlier than x. Hence, the parallel cache complexity $Q^*(\mathsf{t}; \sigma M_i, B_i)$ is an upper bound on the actual number of level-i cache misses.

We next extend the proof for lower-level caches. First, let us consider a level-i strand s belonging to task t. The PCO model states that for any $M_{j<i}$, the cache complexity of a level-i strand matches the serial cache complexity of the strand beginning from an initially empty state. Consider each cache partitioned such that a level-i strand can use only the σM_j capacity of a level-$(j < i)$ cache awarded to it by the space-bounded scheduler. Then the number of misses is indeed as though the strand executed on a serial level-$(i-1)$ memory hierarchy with σM_j cache capacity at each level j. Hence, $Q^*(\mathsf{s}; \sigma M_j, B_j)$ is an upper bound on the actual number of level-j cache misses incurred while executing the strand s. (The actual number may be less because an optimal replacement policy may not partition the caches and the cache state is not initially empty.)

Finally, to complete the proof for all memory-hierarchy levels j, we assume inductively that the theorem holds for all maximal subtasks of t. The PCO model assumes an empty initial level-j cache state for any maximal level-j subtask of t, as $S(\mathsf{t}; B_j) > \sigma M_j$. Thus, the level-$j$ cache complexity for t is defined as $Q^*(\mathsf{t}; \sigma M_j, B_j) = \sum_{\mathsf{t}' \in A(\mathsf{t})} Q^*(\mathsf{t}'; \sigma M_j, B_j, \emptyset)$, where $A(\mathsf{t})$ is the set of all level-i strands and nearest maximal subtasks of t. Since the theorem holds inductively for those tasks and strands in $A(\mathsf{t})$, it holds for t. □

In contrast, there is no such optimality result for the CO model: Theorem 1 (showing a factor of p gap) readily extends to any greedy-space-bounded scheduler, using the same proof.

Runtime Bounds: A Simple Space-Bounded Scheduler and its Limitations. While all space-bounded schedulers achieve optimal cache complexity, they vary in total running time. Greedy-space-bounded schedulers, like the scheduler in [14], perform well for computations that are very well balanced. At a high level, a greedy-space-bounded scheduler operates on tasks anchored at each cache. These tasks are "unrolled" to produce maximal tasks, which are in turn anchored at descendant caches. If a processor P becomes idle and a strand is ready, we assume P begins working on a strand immediately (i.e., we ignore scheduler overheads). If multiple strands are available, one is chosen arbitrarily. Our main scheduler is based on a greedy-space-bounded scheduler, and an operational description of both is included in [8].

Chowdhury et al. [14] present analyses of a (nearly) greedy-space-bounded scheduler (which includes minor enhancements violating the greedy principle). These analyses are algorithm specific and rely on the balance of the underlying computation. A more general performance theorem is included in the associated technical report [8]. Along with our main theorem (Theorem 6), these analyses all use recursive application of Brent's theorem to obtain a total running time: small recursive tasks are assumed inductively to execute quickly, and the larger tasks are analyzed using Brent's theorem with respect to a single-level machine of coarser granularity.

The following are the types of informal structural restrictions imposed on the underlying algorithms to guarantee efficient scheduling with a greedy-space-bounded scheduler and previous work. For more precise, sufficient restrictions, see the technical report [8].

1. *When multiple tasks are anchored at the same cache, they should have similar structure and work. Moreover, none of them should fall on a much longer path through the computation.* If this condition is relaxed, then some anchored task may fall on the critical path. It is important to guarantee each task a fair share of processing resources without leaving many processors idle.
2. *Tasks of the same size should have the same parallelism.*
3. *The nearest maximal descendant tasks of a given task should have roughly the same size.* Relaxing this condition allows two or more tasks at different levels of the memory hierarchy to compete for the same resources. Guaranteeing that each of these tasks gets enough processing resources becomes a challenge.

In addition to these balance conditions, the previous analyses exploit preloading of tasks: the memory used by a task is assumed to be loaded (quickly) into the cache before executing the task. For array-based algorithms preloading is a reasonable requirement. When the blocks to be loaded are not contiguous, however, it may be computationally challenging to determine which blocks should be loaded. Removing the preloading requirement complicates the analysis, which then must account for high-level cache misses that may occur as a result of tasks anchored at lower-level caches.

Our new scheduler in Section 7 relaxes all of these balance conditions, allowing for more asymmetric computations. Moreover, we do not assume preloading. To facilitate analysis of less regular computations, we first define a more holistic measure of the balance of the algorithm in Section 6 and then prove our performance bounds with respect to this metric. This balance metric has the added benefit of separating the algorithm analysis from the scheduler.

6. EXTENDING PCO FOR IMBALANCE

In the PMH (or any machine with shared caches), all caches are associated with a set of processors. It therefore stands to reason that if a task needs memory M but does not have sufficient parallelism to make use of a cache of appropriate size, that either processors will sit idle or additional misses will be required. This might be true even if there is plenty of parallelism on average in the computation. The following lower-bound makes this intuition more concrete.

THEOREM 4. *(Lower Bound) Consider a PMH comprised of a single cache shared by $p > 1$ processors with parameters $B = 1$, M and C, and a memory (i.e., $h = 2$). Then for all $r \geq 1$, there exists a computation with $n = rpM$ memory accesses, $\Theta(n/p)$ span, and $Q^*(M,B) = pM$, such that for any scheduler, the runtime on the PMH is at least $nC/(C+p) \geq (1/2)\min(n, nC/p)$.*

PROOF. Consider a computation that forks off $p > 1$ parallel tasks. Each task is sequential (a single strand) and loops over touching M locations, distinct from any other task (*i.e.*, a total of Mp locations are touched). Each task then repeats touching the same M locations in the same order a total of r times, for a total of $n = rMp$ accesses. Because M fits within the cache, only a task's first M accesses are misses and the rest are hits in the PCO model. The total cache complexity is thus only $Q^*(M,B) = Mp$ for $B = 1$ and any $r \geq 1$.

Now consider an execution (schedule) of this computation on a shared cache of size M with p processors and a miss cost of C. Divide the execution into consecutive sequences of M timesteps, called ***rounds***. Because it takes 1 (on a hit) or $C \geq 1$ (on a miss) units of time for a task to access a location, no task reads the same memory location twice in the same round. Thus, a memory access costs 1 only if it is to a location in memory at the start of the round and C otherwise. Because a round begins with at most M locations in memory, the total number of accesses during a round is at most $(Mp{-}M)/C{+}M$ by a packing argument. Equivalently, in a full round, M processor steps execute at a rate of 1 access per step, and the remaining $Mp - M$ processor steps complete $1/C$ accesses per step, for an average "speed" of $1/p + (1 - 1/p)/C < 1/p + 1/C$ accesses per step. This bound holds for all rounds except the first and last. In the first round, the cache is empty, so the processor speed is $1/C$. The final round may include at most M fast steps, and the remaining steps are slow. Charging the last round's fast steps to the first round's slow steps proves an average "speed" of at most $1/p{+}1/C$ accesses per processor timestep. Thus, the computation requires at least $n/(p(1/p{+}1/C)) = nC/(C{+}p)$ time to complete all accesses. When $C \geq p$, this time is at least $nC/(2C) = n/2$. When $C \leq p$, this time is at least $nC/(2p)$. □

The proof shows that even though there is plenty of parallelism overall and a fraction of at most $1/r$ of the accesses are misses in Q^*, an optimal scheduler either executes tasks (nearly) sequentially (if $C \geq p$) or incurs a cache miss on (nearly) every access (if $C \leq p$).

This indicates that some cost must be charged to account for the space-parallelism imbalance. We extend PCO with a cost metric that charges for such imbalance, but does not charge for imbalance in subtask size. When coupled with our scheduler in Section 7, the metric enables PCO bounds to effectively map to PMH runtime, even for highly-irregular computations.

The metric aims to estimate the degree of parallelism that can be utilized by a symmetric hierarchy as a function of the size of the computation. Intuitively, a computation of size S with "parallelism" $\alpha \geq 0$ should be able to use $p = O(S^\alpha)$ processors effectively. This intuition works well for algorithms where parallelism is polynomial in the size of the problem.

More formally, we define a notion of ***effective cache complexity*** $\widehat{Q}_\alpha(\mathsf{c})$ for a computation c based on the definition of Q^*. Just as for Q^*, $\widehat{Q}_\alpha()$ for tasks, parallel blocks and strands is defined inductively based on the composition rules described in section 2 for building a computation. (Note that since work is just a special case of Q^*, obtained by substituting $M = 0$, the following metric can be used to compute effective work just like effective cache complexity).

DEFINITION 5. *[PCO extended for imbalance] For cache parameters M and B and parallelism α, the* ***effective cache complexity*** *of a strand s, parallel block b, or task t starting at cache state κ is defined as:*

strand: *Let t be the nearest containing task of strand s*

$$\widehat{Q}_\alpha(\mathsf{s}; M, B; \kappa) = Q^*(\mathsf{s}; M, B; \kappa) \times s(\mathsf{t}; B)^\alpha$$

parallel block: *For $\mathsf{b} = \mathsf{t}_1 \| \mathsf{t}_2 \| \ldots \| \mathsf{t}_k$ in task t,*

$$\widehat{Q}_\alpha(\mathsf{b}; M, B; \kappa) =$$

$$\max \left\{ \begin{array}{ll} s(\mathsf{t}; B)^\alpha \max_i \left\{ \frac{\widehat{Q}_\alpha(\mathsf{t}_i; M, B; \kappa)}{s(\mathsf{t}_i; B)^\alpha} \right\} & \textit{(depth dominated)} \\ \sum_i \widehat{Q}_\alpha(\mathsf{t}_i; M, B; \kappa) & \textit{(work dominated)} \end{array} \right.$$

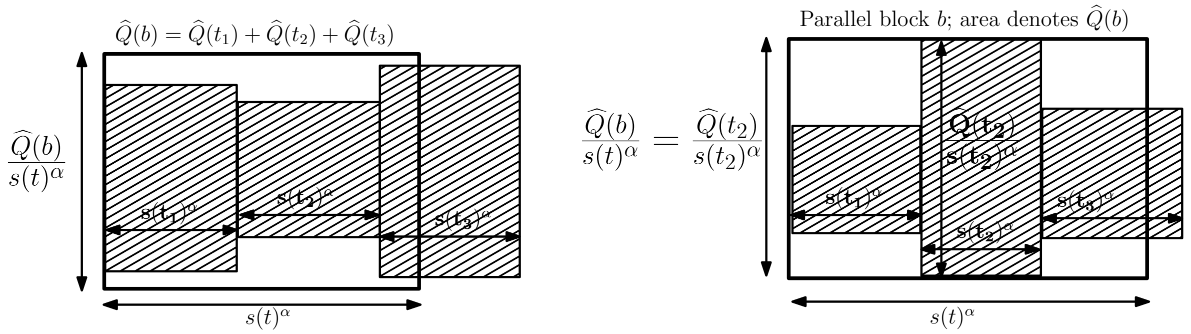
task: *For $\mathsf{t} = c_1; c_2; \ldots; c_k$,*

$$\widehat{Q}_\alpha(\mathsf{t}; M, B; \kappa) = \sum_{i=1}^{k} \widehat{Q}_\alpha(c_i; M, B; \kappa_i) \ ,$$

where κ_i is defined as in Definition 2.

In the rule for parallel block, the *depth dominated* term corresponds to limiting the number of processors available to do the work on each subproblem t_i to $s(\mathsf{t}_i)^\alpha$. This throttling yields a span (depth) $\widehat{Q}_\alpha(\mathsf{t}_i)/s(\mathsf{t}_i)^\alpha$ for each task and the effective cache complexity is then the maximum of the spans over the subtasks multiplied by the number of processors for the parallel block b, which is $s(\mathsf{t}; B)^\alpha$ (see Fig. 4).

We say that an algorithm is *α-efficient* if $Q^*(n; M, B) = O(\widehat{Q}_\alpha(n; M, B))$, where n denotes the input size. This α-efficiency occurs trivially if the work term always dominates, but can also happen if sometimes the depth term dominates. The maximum α for which an algorithm is α-efficient specifies the ***effective parallelism***.



$\widehat{Q}(b) = \widehat{Q}(t_1) + \widehat{Q}(t_2) + \widehat{Q}(t_3)$
$\frac{\widehat{Q}(b)}{s(t)^\alpha}$
$s(t_1)^\alpha$
$s(t_2)^\alpha$
$s(t_3)^\alpha$
$s(t)^\alpha$
Parallel block b; area denotes $\widehat{Q}(b)$
$\frac{\widehat{Q}(b)}{s(t)^\alpha} = \frac{\widehat{Q}(t_2)}{s(t_2)^\alpha}$
$\frac{\widehat{Q}(t_2)}{s(t_2)^\alpha}$
$s(t_1)^\alpha$
$s(t_2)^\alpha$
$s(t_3)^\alpha$
$s(t)^\alpha$

Figure 4: Two examples of Definition 5 applied to a parallel block $\mathsf{b} = \mathsf{t}_1 \parallel \mathsf{t}_2 \parallel \mathsf{t}_3$ belonging to task t. The shaded rectangles represent the subtasks and the white rectangle represents the parallel block b. Subtask rectangles have fixed area ($\widehat{Q}_\alpha(\mathsf{t}_i)$, determined recursively) and *maximum* width $s(t_i)^\alpha$. The left example is work dominated: the total area of b's subtasks is larger than any depth subterms, and determines the area $\widehat{Q}_\alpha(\mathsf{b}) = \widehat{Q}_\alpha(\mathsf{t}_1) + \widehat{Q}_\alpha(\mathsf{t}_2) + \widehat{Q}_\alpha(\mathsf{t}_3)$. The right example is depth dominated: the height of a subtask t_2 determines the height of b and hence the area is $(s(t)/s(t_2))^\alpha \widehat{Q}_\alpha(\mathsf{t}_2)$.

$\widehat{Q}_\alpha(\cdot)$ is an attribute of an algorithm, and as such can be analyzed irrespective of the machine and the scheduler. Appendix B of the associated report [8] illustrates the analysis for $\widehat{Q}_\alpha(\cdot)$ and effective parallelism for several algorithms. Note that, as illustrated in Fig. 4(left) and the analysis of algorithms in the report, good effective parallelism can be achieved even when there is significant work imbalance among subtasks. Finally, depth dominated term implicitly includes the span so we do not need a separate span (depth) cost in our model.

7. SCHEDULER

This section modifies the space-bounded scheduler to address some of the balance concerns discussed in Section 5. These modification restrict the space bounded scheduler, potentially forcing more processors to remain idle. These restriction, nevertheless, allow nicer provable performance guarantees.

The main performance theorem for our scheduler is the following, which is proven in Section 8. This theorem does not assume any preloading of the caches, but we do assume that all block sizes are the same (except at level 0). Here, the machine parallelism β is defined as the minimum value such that for all hierarchy levels $i > 1$, we have $f_i \leq (M_i/M_{i-1})^\beta$, and $f_1 \leq (M_1/3B_1)^\beta$. Aside from the overhead v_h (defined in the theorem), this bound is optimal in the PCO model for a PMH with 1/3-rd the given memory sizes. Here, k is a tunable constant scheduler parameter with $0 < k < 1$, discussed later in this section. Observe that the v_h overhead reduces significantly (even down to a constant) if the ratio of memory sizes is large but the fanout is small (as in the machines in Figure 1), or if $\alpha \gg \beta$.[3]

THEOREM 6. *Consider an h-level PMH with $B = B_j$ for all $1 \leq j \leq h$, and let t be a task such that $S(\mathsf{t}; B) > f_h M_{h-1}/3$ (the desire function allocates the entire hierarchy to such a task) with effective parallelism $\alpha \geq \beta$, and let $\alpha' = \min\{\alpha, 1\}$. The runtime of t is no more than:*

$$\frac{\sum_{j=0}^{h-1} \widehat{Q}_\alpha(\mathsf{t}; M_j/3, B_j) \cdot C_j}{p_h} \cdot v_h, \quad \textit{where overhead } v_h \textit{ is}$$

$$v_h = 2 \prod_{j=1}^{h-1} \left(\frac{1}{k} + \frac{f_j}{(1-k)(M_j/M_{j-1})^{\alpha'}} \right).$$

[3]For example, $v_h < 10$ on the Xeon 7500 as $\alpha \to 1$.

Since much of the scheduler matches the greedy-space-bounded scheduler from Section 5, only the differences are highlighted here. An operational description of the scheduler can be found in the associated technical report [8].

There are three main differences between this scheduler and greedy-space-bounded scheduler from Section 5. First, we fix the dilation to $\sigma = 1/3$ instead of $\sigma = 1$. Whereas reducing σ worsens the bound in Theorem 3 (only by a constant factor for cache-oblivious algorithms), this factor of 1/3 allows us more flexibility in scheduling.

Second, to cope with tasks that may skip levels in the memory hierarchy, we associate with each cache a notion of how busy the descending cluster is, to be described more fully later. For now, we say that a cluster is ***saturated*** if it is "too busy" to accept new tasks, and ***unsaturated*** otherwise. The modification to the scheduler here is then restricting it to anchor maximal tasks only at *unsaturated* caches.

Third, to allow multiple differently sized tasks to share a cache and still guarantee fairness, we partition each of the caches, awarding ownership of specific subclusters to each task. Specifically, whenever a task t is anchored at U, t is also ***allocated*** some subset $\mathcal{U}_t$ of U's level-$(i-1)$ subclusters, essentially granting ownership of the clusters to t. This allocation restricts the scheduler further in that now t may execute only on $\mathcal{U}_t$ instead of all of U. This allocation is exclusive in that a cluster may be allocated to only one task at a time, and no new tasks may be anchored at any cluster $V \in \mathcal{U}_t$ except descendent tasks of t. Moreover, tasks may not skip levels through V, *i.e.*, a new level-$(j < i-1)$ subtask of a level-$k > i$ task may not be anchored at any descendent cache of V. Tasks that skipped levels in the hierarchy before V was allocated may have already been anchored at or below V — these tasks continue running as normal, and they are the main reason for our notion of saturation.

A level-i strand is allocated every cache to which it is anchored, *i.e.*, exactly one cache at every level below i. In contrast, a level-i task $\mathtt{t}$ is anchored only to a level-i cache and allocated potentially many level-$(i-1)$ subclusters, depending on its size. We say that the size-$s = S(\mathtt{t}; B_i)$ task $\mathtt{t}$ ***desires*** $g_i(s)$ level-$(i-1)$ clusters, g_i to be specified later. When anchoring $\mathtt{t}$ to a level-i cache U, let q be the number of unsaturated and unallocated subclusters of U. Select the most unsaturated $\min\{q, g_i(s)\}$ of these subclusters and allocate them to $\mathtt{t}$.

For each cache, there may be one anchored maximal task that is ***underallocated***, meaning that it receives fewer subclusters than it desires. The only underallocated task is the most recent task that caused the cache to transition from being unsaturated to saturated. Whenever a subcluster frees up, allocate it to the underallocated task. If assigning a subcluster causes the underallocated task to achieve its desire, it is no longer underallocated, and future free subclusters become available to other tasks.

Scheduler details. We now describe the two missing details of the scheduler, namely the notion of saturation, as well as the desire function g_i, which specifies for a particular task size the number of desired subclusters.

One difficulty is trying to schedule tasks with large desires on partially assigned clusters. We continue assigning tasks below a cluster until that cluster becomes saturated. But what if the last job has large desire? To compensate, our notion of saturation leaves a bit of slack, guaranteeing that the last task scheduled can get some minimum amount of computing power. Roughly speaking, we set aside a constant fraction of the subclusters at each level as a reserve. The cluster becomes saturated when all other subclusters have been allocated. The last task scheduled, the one that causes the cluster to become saturated, may be allocated subclusters from the reserve.

There is some tradeoff in selecting the reserve constant here. If a large constant is reserved, we may only allocate a small fraction of clusters at each level, thereby wasting a large fraction of all processing power at each level. If, on the other hand, the constant is small, then the last task scheduled may run too slowly. Our analysis will count the first against the work of the computation and the second against the depth.

Designing a good function to describe saturation and the reserved subclusters is complicated by the fact that task assignments may skip levels in the hierarchy. The notion of saturation thus cannot just count the number of saturated or allocated subclusters — instead, we consider the degree to which a subcluster is utilized. For a cluster U with subclusters $V_1, V_2, \ldots, V_{f_i}$ ($f_i > 1$), define the utilization function $\mu(U)$ as follows:

$$\mu(U) = \begin{cases} \min\left\{1, \frac{1}{kf_i}\sum_{i=1}^{f_i} \mu'(V_i)\right\} & \text{if } U \text{ is a level-}(\geq 2) \text{ cluster} \\ \min\{1, \frac{x}{f_1 k}\} & \text{if } U \text{ is a level-1 cluster with } x \text{ allocated processors} \end{cases}$$

and

$$\mu'(V) = \begin{cases} 1 & \text{if } V \text{ is allocated} \\ \mu(V) & \text{otherwise} \end{cases},$$

where $k \in (0,1)$, the value $(1-k)$ specifying the fraction of processors to reserve. For a cluster U with just one subcluster V, $\mu(U) = \mu(V)$. To understand the remainder of this section, it is sufficient to think of k as $1/2$. We say that U is saturated when $\mu(U) = 1$ and unsaturated otherwise.

It remains to define the desire function g_i for level i in the hierarchy. A natural choice for g_i is $g_i(S) = \lceil S/(M_i/f_i)\rceil = \lceil Sf_i/M_i\rceil$. That is, associate with each subcluster a $1/f_i$ fraction of the space in the level-i cache — if a task uses x times this fraction of total space, it should receive x subclusters. It turns out that this desire does not yield good scheduler performance with respect to our notion of balanced cache complexity. In particular it does not give enough parallel slackness to properly load-balance subtasks across subclusters.

Instead, we use $g_i(S) = \min\{f_i, \max\{1, \lfloor f(3S/M_i)^{\alpha'}\rfloor\}\}$, where $\alpha' = \min\{\alpha, 1\}$. What this says is that a maximal level-i task is allocated one subcluster when it has size $S(t; B_i) = M_i/(3f_i^{1/\alpha'})$, and the number of subclusters allocated to t increases by a factor of 2 whenever the size of t increases by a factor of $2^{1/\alpha'}$. It reaches the maximum number of subclusters when it has size $S(t; B_i) = M_{i-1}/3$. We define $g(S) = g_i(S)p_{i-1}$ if $S \in (M_{i-1}/3, M_i/3]$.

For simplicity we assumed in our model that all memory is preallocated, which includes stack space. This assumption would be problematic for algorithms with $\alpha > 1$ or for algorithms which are highly dynamic. However, it is easy to remove this restriction by allowing temporary allocation inside a task, and assume this space can be shared among parallel tasks in the analysis of Q^*. To make our bounds work this would require that for every cache we add an additional number of lines equal to the sum of the sizes of the subclusters. This augmentation would account even for the very worst case where all memory is temporarily allocated.

The analysis of this scheduler is in Section 8, summarized by Theorem 6. There are a couple of challenges that arise in the analysis. First, while it is easy to separate the run time of a task on a sequential machine in to a sum of the cache miss costs for each level, it is not as easy on a parallel machine. Periods of waiting on cache misses at several levels at multiple processors can be interleaved in a complex manner. Our **separation lemma** (lemma 9) addresses this issue by bounding the run time by the sum of its cache costs at different levels ($\widehat{Q}_\alpha(\mathtt{t}; M_, B_i) \cdot C_i$).

Second, whereas a simple greedy-space-bounded scheduler applied to *balanced* tasks lends itself to an easy analysis through an inductive application of Brent's theorem, we have to tackle the problem of subtasks skipping levels in the hierarchy and partially allocated caches. At a high level, the analysis of Theorem 6 recursively decomposes a maximal level-i task into its nearest maximal descendent level-$j < i$ tasks. By inductively assuming that these tasks finish "quickly enough," we combine the subproblems with respect to the level-i cache analogous to Brent's theorem, arguing that a) when all subclusters are busy, a large amount of productive work occurs, b) and when subclusters are idle, all tasks have been allocated sufficient resources to progress at a sufficiently quick rate. Our carefully planned allocation and reservations of clusters as described earlier in this section are critical to this proof.

8. ANALYSIS OF THE SCHEDULER

This section presents the analysis of our scheduler, proving several lemmas leading up to Theorem 6. First, the following lemma implies that the capacity restriction of each cache is subsumed by the scheduling decision of only assigning tasks to unallocated, unsaturated clusters.

Lemma 7. *Any unsaturated level-i cluster U has at least $M_i/3$ capacity available and at least one subcluster that is both unsaturated and unallocated.*

Proof. The fact that an unsaturated cluster has an unsaturated, unallocated cluster follows from the definition. Any saturated or allocated subcluster V_i has $\mu'(V_i) = 1$. Thus, for unsaturated cluster U with subclusters $V_1, \ldots, V_{f_i}$, we have $1 > (1/kf_i)\sum_{j=1}^{f_i} \mu'(V_i) \geq (1/f_i)\sum_{j=1}^{f_i} \mu'(V_i)$, and it follows that some $\mu'(V_i) < 1$.

We now argue that if U is unsaturated, then it has at least $M_i/3$ capacity remaining. This fact is trivial for $f_i = 1$, as in that case at most one task is allocated. Suppose that tasks $\mathsf{t}_1, \mathsf{t}_2, \ldots, \mathsf{t}_k$ are anchored to an unsaturated cluster and have desires $x_1, x_2, \ldots, x_k$. Since U is unsaturated $\sum_{i=1}^{k} x_i \leq f_i - 1$, which implies $x_i \leq f_i - 1$ for all i. We will show that the ratio of space to desire, $S(\mathsf{t}_i; B)/x_i$, is at most $2M_i/3f_i$ for all tasks anchored to U, which implies $\sum_{i=1}^{k} S(\mathsf{t}_i; B) \leq 2M_i/3$.

Since a task with desire $x \in \{1, 2, \ldots, f_i - 1\}$ has size at most $(M_i/3)((x+1)/f_i)^{1/\alpha'}$, where $\alpha' = \min\{\alpha, 1\} \leq 1$, the ratio of its space to its desire x is at most $(M_i/3x)((x+1)/f_i)^{1/\alpha'}$. Letting $q = 1/\alpha' \geq 1$, we have the space-to-desire ratio r bounded by

$$\begin{aligned} r &\leq \frac{M_i}{3} \cdot \frac{(x+1)^q}{x} \cdot \frac{1}{f_i^q} \leq \frac{2M_i}{3} \cdot \frac{(x+1)^q}{x+1} \cdot \frac{1}{f_i^q} \\ &\leq \frac{2M_i}{3f_i} \cdot \frac{(x+1)^{q-1}}{f_i^{q-1}} \leq \frac{2M_i}{3f_i} \end{aligned}$$

□

Latency added cost. Section 6 introduced effective cache complexity $\widehat{Q}_\alpha(\cdot)$, which is algorithmic measure. To analyze the scheduler, however, it is important to consider when cache misses occur. To factor in the effect of the cache miss costs, we define the latency added effective work, denoted by $\widehat{W}_\alpha^*(\cdot)$, of a computation with respect to the particular PMH. Latency added effective work is only for use in the analysis *of the scheduler*, and does not need to be analyzed by an algorithm designer.

The ***latency added effective work*** is similar to the effective cache complexity, but instead of counting just instructions, we add the cost of cache misses at each instruction. The cost $\rho(x)$ of an instruction x accessing location m is $\rho(x) = W(x) + C_i'$ if the scheduler causes the instruction x to fetch m from a level i cache on the given PMH. Using this per-instruction cost, we define effective work $\widehat{W}_\alpha^*(.)$ of a computation using structural induction in a manner that is deliberately similar to that of $\widehat{Q}_\alpha(.)$.

Definition 8 (Latency added cost). *For cost $\rho(x)$ of instruction x, the **latency added effective work** of a task t, or a strand s or parallel block b nested inside t is defined as:*

strand:

$$\widehat{W}_\alpha^*(\mathsf{s}) = s(\mathsf{t}; B)^\alpha \sum_{x \in \mathsf{s}} \rho(x).$$

parallel block: *For* $\mathsf{b} = \mathsf{t}_1 \| \mathsf{t}_2 \| \ldots \| \mathsf{t}_k$,

$$\widehat{W}_\alpha^*(\mathsf{b}) = \max\left\{ s(\mathsf{t}; B)^\alpha \max_i \left\{ \frac{\widehat{W}_\alpha^*(\mathsf{t}_i)}{s(\mathsf{t}_i; B)^\alpha} \right\}, \sum_i \widehat{W}_\alpha^*(\mathsf{t}_i) \right\}. \tag{1}$$

task: *For* $\mathsf{t} = c_1; c_2; \ldots; c_k$,

$$\widehat{W}_\alpha^*(\mathsf{t}) = \sum_{i=1}^{k} \widehat{W}_\alpha^*(c_i). \tag{2}$$

Because of the large number of parameters involved ($\{M_i$, B, $C_i\}_i$ etc.), it is undesirable to compute the latency added work directly for an algorithm. Instead, we will show a nice relationship between latency added work and effective work.

We first show that $\widehat{W}_\alpha^*(\cdot)$ (and $\rho(\cdot)$, on which it is based) can be decomposed into a per (cache) level costs $\widehat{W}_\alpha^{(i)}(\cdot)$ that can each be analyzed in terms of that level's parameters ($\{M_i, B, C_i\}$). We then show that these costs can be put together to provide an upper bound on $\widehat{W}_\alpha^*(\cdot)$. For $i \in [h-1]$, $\widehat{W}_\alpha^{(i)}(\mathsf{c})$ of a computation c is computed exactly like $\widehat{W}_\alpha^*(\mathsf{c})$ using a different base case: for each instruction x in c, if the memory access at x costs at least C_i', assign a cost of $\rho_i(x) = C_i$ to that node. Else, assign a cost of $\rho_i(x) = 0$. Further, we set $\rho_0(x) = W(x)$, and define $\widehat{W}_\alpha^{(0)}(\mathsf{c})$ in terms of $\rho_o(\cdot)$. It also follows from these definitions that $\rho(x) = \sum_{i=0}^{h-1} \rho_i(x)$ for all instructions x.

Lemma 9. ***Separation Lemma:*** *For an h-level PMH with $B = B_j$ for all $1 \leq j \leq h$ and computation A, we have*

$$\widehat{W}_\alpha^*(A) \leq \sum_{i=0}^{h-1} \widehat{W}_\alpha^{(i)}(A).$$

Proof. The proof is based on induction on the structure of the computation (in terms of its decomposition in to block, tasks and strands). For the base case of the induction, consider the sequential thread (or strand) s at the lowest level in the call tree. If $S(\mathsf{s})$ denotes the space of task immediately enclosing s, then by definition

$$\begin{aligned} \widehat{W}_\alpha^*(\mathsf{s}) &= \left(\sum_{x \in \mathsf{s}} \rho(x)\right) \cdot s(\mathsf{s}; B)^\alpha \leq \left(\sum_{x \in \mathsf{s}} \sum_{i=0}^{h-1} \rho_i(x)\right) \cdot s(\mathsf{s}; B)^\alpha \\ &= \sum_{i=0}^{h-1} \left(\sum_{x \in \mathsf{s}} \rho_i(x) \cdot s(\mathsf{s}; B)^\alpha\right) = \sum_{i=0}^{h-1} \widehat{W}_\alpha^{(i)}(\mathsf{s}). \end{aligned}$$

For a series composition of strands and blocks with in a task $\mathsf{t} = x_1; x_2; \ldots; x_k$,

$$\widehat{W}_\alpha^*(\mathsf{t}) \ = \ \sum_{i=1}^{k} \widehat{W}_\alpha^*(x_i) \leq \ \sum_{i=1}^{k} \sum_{l=0}^{h} \widehat{W}_\alpha^{(h)}(x_i) \ = \ \sum_{l=0}^{h} \widehat{W}_\alpha^{(l)}(x)$$

For a parallel block b inside task t consisting of tasks $\{\mathsf{t}_i\}_{i=1}^{m}$, consider the equation 1 for $\widehat{W}_\alpha^*(\mathsf{b})$ which is the maximum of $m+1$ terms, the $(m+1)$-th term being a summation. Suppose that of these terms, the term that determines $\widehat{W}_\alpha^*(\mathsf{b})$ is the k-th term (denote this by T_k). Similarly, consider the equation 1 for evaluating each of $\widehat{W}_\alpha^{(l)}(\mathsf{b})$ and suppose that the k_l-th term (denoted by $T_{k_l}^{(l)}$) on the right hand side determines the value of $\widehat{W}_\alpha^{(l)}(\mathsf{b})$. Then,

$$\frac{\widehat{W}_\alpha^*(\mathsf{b})}{s(\mathsf{t}; B)^\alpha} = T_k \leq \sum_{l=0}^{h-1} T_k^{(l)} \leq \sum_{l=0}^{h-1} T_{k_l}^{(l)} = \frac{\sum_{l=0}^{h-1} \widehat{W}_\alpha^{(l)}(\mathsf{b})}{s(\mathsf{t}; B)^\alpha}, \tag{3}$$

which completes the proof. Note that we did not use the fact that some of the components were work or cache complexities. The proof only depended on the fact that $\rho(x) =$

$\sum_{i=0}^{h-1} \rho_i(x)$ and the structure of the composition rules given by equations 2, 1. ρ could have been replaced with any other kind of work and ρ_i with its decomposition. □

The previous lemma indicates that the latency added work can be separated into costs per cache level. The following lemma then relates these separated costs to effective cache complexity $\widehat{Q}_\alpha(\cdot)$.

LEMMA 10. *Consider an h-level PMH with $B = B_j$ for all $1 \leq j \leq h$ and a computation* c. *If* c *is scheduled on this PMH using a space-bounded scheduler with dilation $\sigma = 1/3$, then $\widehat{W}^*_\alpha(\mathsf{c}) \leq \sum_{i=0}^{h-1} \widehat{Q}_\alpha(\mathsf{c}; M_i/3, B) \cdot C_i$.*

PROOF. (*Sketch*) The function $\widehat{W}^{(i)}_\alpha(\cdot)$ is monotonic in that if it is computed based on function $\rho'_i(\cdot)$ instead of $\rho_i(x)$, where $\rho'_i(x) \leq \rho_i(x)$ for all instructions x, then the former estimate would be no more than the latter. It then follows from the definitions of $\widehat{W}^{(i)}_\alpha(\cdot)$ and $\rho_i(\cdot)$, that $\widehat{W}^{(i)}_\alpha(\mathsf{c}) \leq \widehat{Q}_\alpha(\mathsf{c}; M_i/3, B) \cdot C_i$ for all computations c, $i \in \{0, 1, \ldots, h-1\}$. Lemma 9 then implies that for any computation c: $\widehat{W}^*_\alpha(\mathsf{c}) \leq \sum_{i=0}^{h-1} \widehat{Q}_\alpha(\mathsf{c}; M_i/3, B) \cdot C_i$. □

Finally, we prove the main lemma, bounding the running time of a task with respect to the remaining utilization the clusters it has been allocated. At a high level, the analysis recursively decomposes a maximal level-i task into its nearest maximal descendent level-$j < i$ tasks. We assume inductively that these tasks finish "quickly enough." Finally, we combine the subproblems with respect to the level-i cache analogous to Brent's theorem, arguing that a) when all subclusters are busy, a large amount of productive work occurs, b) and when subclusters are idle, all tasks make sufficient progress. Whereas this analysis outline is consistent with a simple analysis of the greedy scheduler and that in [14], here we address complications that arise due to partially allocated caches and subtasks skipping levels in the hierarchy.

LEMMA 11. *Consider an h-level PMH with $B = B_j$ for all $1 \leq j \leq h$ and a computation to schedule with $\alpha \geq \beta$, and let $\alpha' = \min\{\alpha, 1\}$. Let N_i be a task or strand which has been assigned a set $\mathcal{U}_t$ of $q \leq g_i(S(N_i; B))$ level-$(i-1)$ subclusters by the scheduler. Letting $\sum_{V \in \mathcal{U}_t}(1 - \mu(V)) = r$ (by definition, $r \leq |\mathcal{U}_t| = q$), the running time of N_i is at most:*

$$\frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}} \cdot v_i, \quad \text{where overhead } v_i \text{ is}$$

$$v_i = 2\prod_{j=1}^{i-1}\left(\frac{1}{k} + \frac{f_i}{(1-k)(M_i/M_{i-1})^{\alpha'}}\right).$$

PROOF. We prove the claim on run time using induction on the levels.

Induction: Assume that all child maximal tasks of N_i have run times as specified above. Now look at the set of clusters $\mathcal{U}_t$ assigned to N_i. At any point in time, either:

1. all of them are saturated.

2. at least one of the subcluster is unsaturated and there are no jobs waiting in the queue $R(N_i)$. More specifically, the job on the critical path $(\chi(N_i))$ is running. Here, critical path $\chi(N_i)$ is the set of strictly ordered immediate child subtasks that have the largest sum of effective depths. We would argue in this case that progress is being made along the critical path at a reasonable rate.

Assuming $q > 1$, we will now bound the run time required to complete N_i by bounding the number of cycles the above two phases use. Consider the first phase. A job $x \in C(N_i)$ (subtasks of N_i) when given an appropriate number of processors (as specified by the function g) can not have an overhead of more than v_{i-1}, i.e., it uses at most $\widehat{W}^*_\alpha(x)v_{i-1}$ individual processor clock cycles. Since in the first phase, at least k fraction of available subclusters under $\mathcal{U}_t$ are always allocated (at least rp_{i-1} clock cycles put together) to some subtask of N_i, it can not last for more than

$$\sum_{x \in C(N_i)} \frac{1}{k}\frac{\widehat{W}^*_\alpha(x)}{rp_{i-1}} \cdot v_{i-1} < \frac{1}{k}\frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}} \cdot v_{i-1} \text{ number of cycles.}$$

For the second phase, we argue that the critical path runs fast enough because we do not underallocate processing resources for any subtask by more than a factor of $(1-k)$ as against that indicated by the g function. Specifically, consider a job x along the critical path $\chi(N_i)$. Suppose x is a maximal level-$j(x)$ task, $j(x) < i$. If the job is allocated subclusters below a level-$j(x)$ subcluster V, then V was unsaturated at the time of allocation. Therefore, when the scheduler picked the $g_{j(x)}(S(x; B))$ most unsaturated subclusters under V (call this set $\mathcal{V}$), $\sum_{v \in \mathcal{V}} \mu(v) \geq (1-k)g_{j(x)}(S(x; B))$. When we run x on V using the subclusters $\mathcal{V}$, its run time is at most

$$\begin{aligned}\frac{\widehat{W}^*_\alpha(x)}{(\sum_{v \in \mathcal{V}} \mu(v))p_{j(x)-1}} \cdot v_{j(x)-1} &< \frac{\widehat{W}^*_\alpha(x) \quad \cdot \quad v_{j(x)-1}}{(1-k)g(S(x; B_{j(x)}))} \\ &= \frac{\widehat{W}^*_\alpha(x)}{s(x; B)^\alpha}\frac{s(x; B)^\alpha}{g(S(x; B_{j(x)}))}\frac{v_{j(x)-1}}{1-k}\end{aligned}$$

time. Amongst all subtasks x of N_i, the ratio $\frac{s(x;B)^\alpha}{g(S(x;B_{j(x)}))}$ is maximum when when $S(x; B) = M_{i-1}/3$, where the ratio is $(M_{i-1}/3B)^\alpha/p_{i-1}$. Summing the run times of all jobs along the critical path would give us an upper bound for time spent in phase two. This would be at most

$$\begin{aligned}&\sum_{x \in \chi(N_i)} \frac{\widehat{W}^*_\alpha(x)}{(1-k)g(S(x; B))} \cdot v_{i-1} \\ &= \sum_{x \in \chi(N_i)} \frac{\widehat{W}^*_\alpha(x)}{s(x; B)^\alpha} \cdot \frac{s(x; B)^\alpha}{g(S(x; B))} \cdot \frac{v_{i-1}}{1-k} \\ &\leq \left(\sum_{x \in \chi(N_i)} \frac{\widehat{W}^*_\alpha(x)}{s(x; B)^\alpha}\right) \cdot \frac{(M_{i-1}/3B)^\alpha}{p_{i-1}} \cdot \frac{v_{i-1}}{1-k} \\ &\leq \frac{\widehat{W}^*_\alpha(N_i)}{s(N_i; B)^\alpha} \cdot \frac{(M_{i-1}/3B)^\alpha}{p_{i-1}} \cdot \frac{v_{i-1}}{1-k} \qquad \text{(by defn. of } \widehat{W}^*_\alpha()) \\ &= \frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}} \cdot \frac{r(M_{i-1}/3B)^\alpha}{s(N_i; B)^\alpha} \cdot \frac{v_{i-1}}{1-k} \\ &\leq \frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}} \cdot \frac{q(M_{i-1}/3B)^\alpha}{s(N_i; B)^\alpha} \cdot \frac{v_{i-1}}{1-k} \\ &\leq \frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}} \cdot \frac{f_i}{(1-k)(M_i/M_{i-1})^{\alpha'}} \cdot v_{i-1} \qquad \text{(by defn. of } g).\end{aligned}$$

Putting together the run times of both the phases, we have an upper bound of

$$\frac{\widehat{W}_\alpha^*(N_i)}{rp_{i-1}} v_{i-1} \cdot \left(\frac{1}{k} + \frac{f_i}{(1-k)(M_i/M_{i-1})^{\alpha'}}\right) = \frac{\widehat{W}_\alpha^*(N_i)}{rp_{i-1}} \cdot v_i.$$

If $q = 1$, N_i would get allocated just one $(i-1)$-subcluster V, and of course, all the (yet unassigned) $(i-2)$ subclusters $\mathcal{V}$ below V. Then, we can view this scenario as N_i running on the $(i-1)$-level hierarchy. Memory accesses and cache latency costs are charged the same way as before with out modification so that the effective work of N_i would still be $\widehat{W}_\alpha^*(N_i)$. By inductive hypothesis, we know that the run time of N_i would be at most

$$\frac{\widehat{W}_\alpha^*(N_i)}{(\sum_{V \in \mathcal{V}}(1-\mu(V)))p_{i-2}} \cdot v_{i-1}$$

which is at most $\frac{\widehat{W}_\alpha^*(N_i)}{rp_{i-1}} \cdot v_i$ since $\sum_{V \in \mathcal{V}}(1-\mu(V)) \geq rf_{i-1}$ and $v_{i-1} < v_i$.

Base case ($i = 1$): N_1 has $q = r$ processors available, all under a shared cache. If $q = 1$, the claim is clearly true. If $q > 1$, since there is no further anchoring beneath the level-1 cache (since $M_0 = 0$), we can use Brent's theorem on the latency added effective work to bound the run time: $\frac{\widehat{W}_\alpha^*(N_1)}{r}$ added to the critical path length, which is at most $\frac{\widehat{W}_\alpha^*(N_1)}{s(N_1;B)^\alpha}$. This sum is at most

$$\begin{aligned}
\frac{\widehat{W}_\alpha^*(N_1)}{r}\left(1 + \frac{q}{s(N_1;B)^\alpha}\right) &\leq \frac{\widehat{W}_\alpha^*(N_1)}{r}\left(1 + \frac{g(S(N_1;B))}{s(N_1;B)^\alpha}\right) \\
&\leq \frac{\widehat{W}_\alpha^*(N_1)}{r}\left(1 + \frac{S(N_1;B)^\alpha}{s(N_1;B)^\alpha} \cdot \frac{f_1}{(M_1/3)^\alpha}\right) \\
&\leq \frac{\widehat{W}_\alpha^*(N_1)}{r} \times 2.
\end{aligned}$$

□

Theorem 6 follows from Lemmas 10 and 11, starting on a system with no utilization.

9. CONCLUSION

The paper described models that capture the "locality" of an algorithm independently of the machine or how the computation is mapped onto the machine either by hand or by a scheduler. In particular the models are just based on the structure of the program: they make no reference to processors, and use only two simple cache parameters, one capturing temporal locality (M) and one spatial locality (B), and one parallelism parameter (α). The models modify the sequential cache-oblivious model to avoid capturing false dependences and to account for memory-parallelism imbalances. The paper also developed a scheduler that can guarantee strong bounds when mapping the costs analyzed in the model to cache misses and runtime on parallel machines with tree-of-caches hierarchies. We expect the model can also be used for other types of machines with hierarchical locality.

Acknowledgments. This work is partially supported by the National Science Foundation under grant number CCF-1018188, as well as under the NSF/CRA sponsored CIFellows program. We are grateful to Intel, IBM, and Microsoft for generous gifts that have helped support this work.

10. REFERENCES

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

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

[3] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *Programming Models for Massively Parallel Computers*, 1993.

[4] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *SPAA*, 2008.

[5] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious B-trees. In *SPAA*, 2005.

[6] G. Bilardi, A. Pietracaprina, G. Pucci, and F. Silvestri. Network-oblivious algorithms. In *IPDPS*, 2007.

[7] 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*, 2008.

[8] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. A cache-oblivious model for parallel memory hierarchies. Technical Report CMU-CS-10-154, Computer Science Department, Carnegie Mellon University, 2010.

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

[10] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low-depth cache oblivious algorithms. In *SPAA*, 2010.

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

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

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

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

[15] R. Cole and V. Ramachandran. Efficient resource oblivious scheduling of multicore algorithms. manuscript, 2010.

[16] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. In *ICALP*, 2010.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd Edition*. MIT Press, 2001.

[18] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7), 1993.

[19] P. de la Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. In *Euro-Par, Vol. II*, 1996.

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

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

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

[23] C. E. Leiserson. Fat-Trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C–34(10), 1985.

[24] L. G. Valiant. A bridging model for parallel computation. *CACM*, 33(8), 1990.

[25] L. G. Valiant. A bridging model for multi-core computing. In *ESA*, 2008.