Scheduling Irregular Parallel Computations on Hierarchical Caches

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

December 2010
CMU-CS-10-154

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: jfineman@cs.cmu.edu
[**] email: harshas@cs.cmu.edu

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

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

Abstract

Making efficient use of cache hierarchies is essential for achieving good performance on multicore and other shared-memory parallel machines. Unfortunately, designing algorithms for complicated cache hierarchies can be difficult and tedious. To address this, recent work has developed high-level models that expose locality in a manner that is oblivious to particular cache or processor organizations, placing the burden of making effective use of a parallel machine on a runtime *task scheduler* rather than the algorithm designer/programmer.

This paper continues this line of work by (i) developing a new model for parallel cache cost, (ii) developing a task scheduler for irregular tasks on cache hierarchies, and (iii) proving that the scheduler assigns tasks to processors in a work-efficient manner (including cache costs) relative to the model. As with many previous models, our model allows algorithms to be analyzed using a single level of cache with parameters M (cache size) and B (cache-line size), and algorithms can be written cache obliviously (with no choices made based on machine parameters). Unlike previous models, our cost $\widehat{Q}_\alpha(n; M, B)$, for problem size n, captures costs due to work-space imbalance among tasks, and we prove a lower bound that shows that some sort of penalty is needed to achieve work efficiency. Nevertheless, for many algorithms, $\widehat{Q}_\alpha()$ is asymptotically equal to the standard sequential cache cost $Q()$.

Our task scheduler is a specific "space-bounded scheduler," which assigns subtasks to caches based on their space usage. Our scheduler extends prior work by efficiently scheduling "irregular" computations with arbitrary *work imbalance* among parallel subtasks, reflected by the $\widehat{Q}_\alpha()$ cost. Moreover, in addition to proving bounds on cache complexity, we also provide a bound on the total running time of an program execution using our scheduler. Specifically, our scheduler executes a program on a homogeneous h-level parallel memory hierarchy having p processors in time $O\left((v_h/p)\sum_{i=0}^{h}\widehat{Q}_\alpha(n; M_i, B)\cdot C_i\right)$, where M_i is the size of the level-i cache, B is the cache-line size, C_i is the cost of level-i cache miss, and v_h is an overhead defined in the paper. Ignoring the overhead v_h, which may be small (or even constant) when certain side conditions hold, this bound is optimal whenever $\widehat{Q}_\alpha()$ matches the sequential cache complexity—at every level of the hierarchy it is dividing the total cache cost by the total number of processors.

1 Introduction

Because of limited bandwidths on real parallel machines locality is critical to achieving good performance for parallel programs. To account for this in the design of algorithms, over the years many locality-aware parallel models have been suggested [21, 22, 16, 2, 17, 6]. 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 cache hierarchy with a factor of over a hundred 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 hierarchy for the Xeon based Nehalem architecture, the current generation of desktop architecture from Intel. Correspondingly there has been significant recent work on parallel cache based locality [1, 8, 5, 19, 11, 7, 4, 12, 23, 9, 13]. The work has fallen into two main classes. One class involves designing algorithms directly for the machine. This includes the work by Arge et al. on designing algorithms directly for a p-processor machine with one layer of private caches (the PEM) [4], and by Valiant [23] 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 a scheduler is responsible for mapping this onto the processors.

Dynamic parallelism can be further divided between approaches in which the user analyzes their algorithm in an abstract model that knows nothing about the scheduler and its interaction with the machine [1, 8, 19, 11, 9], and approaches in which the analysis requires an integrated analysis [7, 12, 13, 15, 14]. In the first class the scheduler is required to supply a general mapping from an abstract cost model to the particular machine models. A pair of common abstract measures are the number of misses given a sequential ordering of a parallel computation [1, 8, 19, 9] and the depth of the computation. For example, one can show that any nested-parallel computation with depth D and sequential complexity Q_1 will have at most $Q_1 + O(PDM/B)$ misses on a parallel machine with P processors, each with a private cache of size M and block size B, when using a work-stealing scheduler [1]. Quicksort, for example, has depth $D(n) = O(\log^2 n)$ and $Q_1 = Q_1(n; M, B) = O((n/B)\log(n/M))$, and hence suffers $O((n/B)\log(n/M) + (PM\log^2 n)/B)$ misses on a P processor private-cache machine with cache size M (all w.h.p.). For current machines and reasonable sized problems the first term dominates and the total number of misses is asymptotically the same as on a serial machine.

Dynamic parallelism has important advantages over programming directly for the machine, including being much simpler (*e.g.*, consider analyzing quicksort by having to map all recursive subproblems to different parts of a memory hierarchy by hand), and being much closer to how users actually code on these machines using languages such as Open MP, Cilk++, Intel TBB, and the Microsoft Task Parallel Library. The existing approaches mentioned above, however, have some important limitations. They either apply to hierarchies of only private or only shared caches [1, 8, 19, 15, 9], require some strict balance criteria [7, 14], or require a per algorithm analysis [13]. In this paper we present a model and corresponding scheduler that enables an analysis that is independent of the scheduler, allows for arbitrary imbalance among tasks, and works on hierarchies of shared and private caches (as in Fig. 1). The approach is limited to nested-parallel computations without race conditions, but this includes a very broad set of algorithms, certainly including all CREW PRAM-like algorithms.

The approach is based on three components. The first is a cache cost model (***Parallel Cache-Oblivious (PCO)***) that diverges from the serial cache cost used in previous work. The problem with the serial model is that it captures incidental cache reuse among parallel subcomputations, which makes it hard to prove strong bounds for parallel schedulers, especially in the hierarchical setting. The idea in the extension is

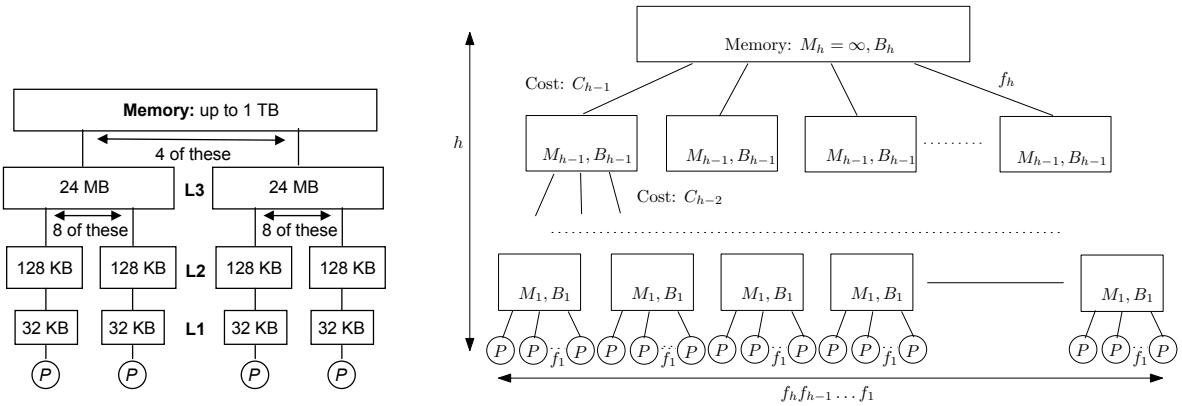

Figure 1: (left) Memory hierarchy of the Intel Xeon 7500. (right) An example abstract parallel memory hierarchy: the PMH model of [3] (details in Section 2). Each cache (rectangle) is shared by all processors (circles) in its subtree.

to ignore the data reuse between parallel subtasks—instead, the model accounts for reuse only when there is a serial relationship between instructions accessing the same data. Thus, the cache cost in the PCO model may be higher than the serial cache cost. 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 gives needed flexibility to the scheduler.

The second is a 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 too small.

The third is a new "space-bounded scheduler" that extends recent work of Chowdhury et al. [13]. In particular our scheduler allows parallel subtasks to be scheduled on different levels in the memory hierarchy, thus allowing significant imbalance in the sizes of tasks. 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 allowed to migrate across caches. We show that the 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 PCO model with problem size n, cache size M_i, and cache-line size B_i. Finally, we show that a particular 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. In particular the runtime can be determined by considering each cache level independently and calculating the time simply by multiplying the number of misses for the cache parameters at that level by the cost of a miss at that level and dividing by the total number of processors (see Theorem 5).

2 Preliminaries: Computation and Machine Models

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."

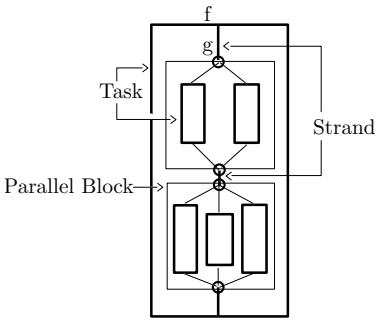

Figure 2: Decomposing the computation: tasks, strands and parallel blocks

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: Applying case 1 of equation 1 of the PCO model. $Q^*(\mathsf{t}; M, B; \kappa) = 4$.

A ***parallel block*** is formed by composing in parallel one or more tasks with a fork point before all of them and a join point after. 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. A parallel block is initiated by a *fork* instruction which takes a counter of how many tasks to fork and a *continuation strand* to execute when all tasks return. Each task receives the same state except for an index that indicates which task it is. It is easy to implement parallel loops or simple parallel recursive calls (*e.g.*, recursive calls in a divide-and-conquer algorithm) with such a construct. A *join* instruction finishes the task. When all tasks of a parallel block are finished, then the continuation strand for the block can be executed. 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 [3,11,7,12,23,9,13], 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), although our results can be readily adapted to other models. 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 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. The cost of a "miss" in a level-i cache is denoted by C_i, 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 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 to denote the number of level-$(i-1)$ caches below a single level-i cache, also called the ***fanout***. We assume the model maintains DAG consistent shared memory with the BACKER algorithm [10]. This is a weak consistency model and assumes that cache lines are merged on writing back to memory thus avoiding false sharing.

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 [18], in the ***Parallel Cache-Oblivious (PCO) model*** there are 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*.

In order to extend the CO model to parallel computations, PCO 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 level of private caches, as the following theorem demonstrates for the CO model:

Theorem 1 *Consider a PMH comprised of $p \geq 2$ processors, each with a private cache of szie $M \geq 1$, and a shared memory (i.e., $h = 2$). Then there exists a parallel block such that for any greedy scheduler,[2] the number of 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 tasks, each consisting of a strand reading the same set of M memory locations from M/B blocks. The last task has an additional read to a different memory location x. On the CO model with cache size M and block size B, after the first M/B misses, all other accesses are hits except for the read of x. Thus, the cache cost is $M/B + 1$ on the CO model.

Any greedy schedule on p processors executes each strand on a distinct processor, and hence must load the M locations into each processor's private cache of size M. Thus, the block incurs $p(M/B) + 1$ misses on the PMH model with private caches. □

On the other hand, note that in the PCO model we define, the cache complexity $Q^*(M, B)$ for this particular construction matches the number of PMH misses. In general, we show that for any nested parallel computation and any "space-bounded" scheduler (defined in Section 4), the number of misses at each level i of an arbitrary PMH hierarchy is upper bounded by $Q^*(M_i, B_i)$ of the PCO model (see Theorem 2).

The gap arises because a sequential ordering accounts for significant reuse among the subtasks in the block, but there is no such reuse when each subtask is scheduled on its own private cache.

To overcome this difficulty, we instead use the novel 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). We separate two cases: when a task t fits within a cache (*i.e.*, $S(\mathsf{t}; B) \leq M$) and when it does not. In the first case the cache state is inherited by subtasks at a fork, and merged (union) at the corresponding join point. In the second case we assume that any fork or join point clears the cache. More formally we denote the ***cache***

[1] Two prior works not using the sequential ordering are the *concurrent cache-oblivious model* [5] and the *ideal distributed cache model* [19], 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. We later extend this theorem to "greedy" space-bounded schedulers, where processors can also idle to avoid violating space constraints.

Problem	Span	Cache Complexity Q^*
Scan (prefix sums, etc.)	$O(\log n)$	$O(\lceil n/B \rceil)$
Matrix Transpose ($n \times m$ matrix) [18]	$O(\log(n+m))$	$O(\lceil nm/B \rceil)$
Matrix Multiplication ($\sqrt{n} \times \sqrt{n}$ matrix) [18]	$O(\sqrt{n})$	$O(\lceil n^{1.5}/B \rceil / \sqrt{M+1})$
Matrix Inversion ($\sqrt{n} \times \sqrt{n}$ matrix)	$O(\sqrt{n})$	$O(\lceil n^{1.5}/B \rceil / \sqrt{M+1})$
Quicksort [20]	$O(\log^2 n)$	$O(\lceil n/B \rceil \log(n/(M+1)))$
Sample Sort (randomized) [9]	$O(\log^{1.5} n)$	$O(\lceil n/B \rceil \lceil \log_{M+2} n \rceil)$
Sparse-Matrix Vector Multiply [9] (m nonzeros, n^ϵ separators)	$O(\log^2 n)$	$O(\lceil m/B + n/(M+1)^{1-\epsilon} \rceil)$
Convex Hull (e.g., see Fig. 6)	$O(\log^2 n)$	$O(\lceil n/B \rceil \lceil \log_{M+2} n \rceil)$
Barnes Hut tree (e.g., see Fig. 6)	$O(\log^2 n)$	$O(\lceil n/B \rceil \log(n/(M+1)))$

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.

complexity (*i.e.*, number of cache misses) of a task, strand or parallel block c as $Q^*(\mathsf{c}; M, B; \kappa)$ where κ is the initial cache state and use compositional rules to define it. For a strand s, $Q^*(\mathsf{s}; M, B; \kappa)$ is the same as the ideal cache model starting at state κ.[3] For a parallel block b in task t comprised of k subtasks $\mathsf{t}_1, \mathsf{t}_2, \ldots, \mathsf{t}_k$,

$$Q^*(\mathsf{b}; M, B; \kappa), \kappa' \quad = \quad \begin{cases} (\sum_i Q^*(\mathsf{t}_i; M, B; \kappa)), \kappa \cup_i loc(\mathsf{t}_i) & S(\mathsf{t}; B) \leq M \\ (\sum_i Q^*(\mathsf{t}_i; M, B; \emptyset)), \emptyset & \text{otherwise} \end{cases} \tag{1}$$

where $(Q^*(\cdot), \kappa')$ represents the cache complexity and new cache state at the join point (see Fig. 3). Note that the above rule for carrying cache state also defines the rule for sequential composition since it specifies the input (κ) and output (κ') state. 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 $Q^*(\mathsf{c}; 0, 1, \kappa)$ to denote the computational work. Note that by setting M to 0, we implcit force the analysis to count every instruction including those that are not explicit memory accesses (like fork and join instructions) to be counted towards work.

Comments on the definition: Because each subtask is analyzed based on its independent version of the cache starting from either the cache state at the fork point (case 1) or an empty cache (case 2), data reuse among parallel subtasks (*e.g.*, the pair of accesses to l_4 in Fig. 3) is ignored. Whenever a task does not fit in cache (case 2), its subtasks may compete for cache space; emptying the cache at its fork and join points provides a simple upper bound on the resulting misses. Whenever there is a maximal task that fits in cache (*i.e.*, a task whose parent task did not fit), it starts from an empty cache and only case 1 will be applied until the task completes; thus the union of all the locations accessed by its constituent strands and parallel blocks will indeed fit in the cache, so κ' in equation 1 is a legal cache state. Moreover, because the task fits in cache, we have space to *preload* before forking any cache line (such as l_4) that will be used by multiple subtasks, thereby eliminating all but one miss for the line.

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 Appendix A). Moreover, as we will show in Section 4, 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

[3]While it might seem that the furthest in the future eviction policy may lead to divergence between the two models even on a strand, note that in case 1 all accesses within the task will fit and in case 2 the cache is cleared.

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 The Basic Space-Bounded Scheduler

In this section we describe a class of schedulers, called space-bounded schedulers, and show (Theorem 2) 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. [13], but their paper does not use the PCO model and hence cannot show the same kind of optimality as Theorem 2. This section also describes a specific online space-bounded scheduler that performs very well in terms of runtime on very balanced computations, and use it to highlight some of the difficulties in designing a scheduler (such as the one in Section 6) 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 [13] 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 program 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 2 *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. Since the space-bounded scheduler awards σM_j capacity of a level-$(j < i)$ cache to the level-i strand, a level-i strand indeed executes as though on a serial level-$(i-1)$ memory hierarchy with σM_j memory. 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.

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, we have level-$j$ cache complexity for t 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. □

This theorem shows that the greedy scheduler to be described next with dilation $\sigma = 1$ exhibits optimal cache performance with respect to the PCO model. If σ is a constant, this bound is also asymptotically optimal for cache-oblivious algorithms.

A Simple Space-Bounded Scheduler and its Limitations. While all space-bounded schedulers achieve optimal cache complexity, they vary in total running time. 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. We consider a simple, online, greedy space-bounded scheduler that is similar to one presented in [13], except for some key low-level details (e.g., scheduling of strands, which is omitted from [13]). To avoid any ambiguity, we have included an operational description of our greedy space-bounded scheduler. At a high level, the 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.

Operational details Before discussing how the scheduler operates, we address state maintained by the scheduler. We maintain for each level-i cache U the total space used by maximal tasks anchored to U.

We maintain for each anchored maximal level-i task t two lists. The ***ready-strand list*** $\mathcal{S}(\mathsf{t})$ contains the ready level-i strands of t. The ***ready-task list*** $\mathcal{R}(\mathsf{t})$ contains the ready, unexecuted (and hence unanchored), maximal level-$(j < i)$ subtasks of t. The ready-task list contains only maximal tasks, as tasks which are not maximal are implicitly anchored to the same cache as t.[4]

A strand or task is assigned to the appropriate list when it first becomes ready. We use $parent[\mathsf{t}]$ to denote the nearest containing task of task/strand t, and $maximal[\mathsf{t}]$ to denote the nearest maximal task containing task/strand t ($maximal[\mathsf{t}] = \mathsf{t}$ if t is maximal). When the execution of a strand reaches a fork, we also keep a count on the number of outstanding subtasks, called the join counter.

Initially, the main task t is anchored at the smallest cache U in which it fits, both $\mathcal{R}(\mathsf{t})$ and $\mathcal{S}(\mathsf{t})$ are created, and t is allocated all subclusters of U (ignore for the greedy scheduler). The leading strand of the task is added to the ready-strand list $\mathcal{S}(\mathsf{t})$, and all other lists are empty.

The scheduler operates in parallel, invoking either of the two following scheduling rules when appropriate. We ignore the precise implementation of the scheduler, and hence merely assume that the invocation of each rule is atomic. For both rule descriptions, let t be a maximal level-i task, let U be the cache to which it

[4]For this greedy scheduler, a pair of lists for each cache is sufficient, *i.e.*, $\mathcal{R}(U) = \bigcup_{\mathsf{t} \text{ anchored to } U} \mathcal{R}(\mathsf{t})$, and similarly for $\mathcal{S}$, but for consistency of notation with later schedulers we continue with a pair of lists per task here.

is anchored, and let $\mathcal{U}_t$ be all the children of U for the greedy scheduler.

strands We say that a cache U_j is ***strand-ready*** if there exists a cache-to-processor path of caches $U_j, U_{j-1}, \ldots, U_1, p$ descending from U_j down to processor P such that: 1) each $U_k \in \{U_j, \ldots, U_1\}$ has σM_k space available, and 2) P is idle. We call $U_j, U_{j-1}, \ldots, U_1, p$ the ***ready path***. If $\mathcal{S}(\mathsf{t})$ is not empty and $V \in \mathcal{U}_t$ is strand-ready, then remove a strand s from $\mathcal{S}(\mathsf{t})$. Anchor s at all caches along the ready path descending from V, and decrease the remaining capacity of each U_k on the ready path by σM_k. Execute the strand on the ready path's processor.

tasks Suppose there exists some level-$(j-1) < (i-1)$ strand-ready descendant cache U_{j-1} of $V \in \mathcal{U}_t$, and let U_j be its parent cache. If there exists level-j subtask $\mathsf{t}' \in \mathcal{R}(\mathsf{t})$ such that U_j has $S(\mathsf{t}'; B_j)$ space available, then remove t' from $\mathcal{R}(\mathsf{t})$. Anchor t' at U_j, create $\mathcal{R}(\mathsf{t}')$ and $\mathcal{S}(\mathsf{t}')$, and reduce U_j's remaining capacity appropriately. Then schedule and execute the first strand of t' as above.

Observe that the implementation of these rules can either be top-down with caches pushing tasks down or bottom up with idle processors "stealing" and pulling tasks down. To be precise about where pieces of the computation execute, we've chosen both scheduling rules to cause a strand to execute on a processor. This choice can be relaxed without affecting the bounds.

When P has been given a strand s to execute (by invoking either scheduling rule), it executes the strand to completion. Let $\mathsf{t} = parent[\mathsf{s}]$ be the containing task. When s completes, there are two cases:
Case 1 (fork). When P completes the strand and reaches a fork point, t's join counter is set to the number of forked tasks. Any maximal subtasks are inserted into the ready-task list $\mathcal{R}(maximal[\mathsf{t}])$, and the lead strand of any other (non maximal) subtask is inserted into the ready-strand list $\mathcal{S}(maximal[\mathsf{t}])$.
Case 2 (task end). If P reaches the end of the task t, then the containing parent task $parent[\mathsf{t}]$'s join counter is decremented. If $parent[\mathsf{t}]$'s join counter reaches zero, then the subsequent strand in $parent[\mathsf{t}]$ is inserted into $maximal[parent[\mathsf{t}]]$'s ready-strand list. Moreover, if t is a maximal level-i task, the total space used in by the cache to which t was anchored is decreased by $S(\mathsf{t}; B_i)$ to reflect t's completion.
In either of these cases, P becomes idle again, and scheduling rules may be invoked.

Space-bounded schedulers like this greedy variant perform well for computations that are very balanced. Chowdhury et al. [13] present analyses of a similar space-bounded scheduler (that includes minor enhancements violating the greedy principle). These analyses are algorithm specific and rely on the balance of the underlying computation. We prove a theorem (Th. 3) to provides run time bounds of the same flavor for the greedy space-bounded scheduler, leveraging many of the same assumptions on the underlying algorithms.

The following are the types of informal structural restrictions imposed on the underlying algorithms to guarantee efficient scheduling with our greedy scheduler and previous work.

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 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.

We now formalize the structural restrictions described above and analyse a greedy space-bounded scheduler with ideal dilation $\sigma = 1$. This analysis is intended both as an outline for the main theorem in Section 6 and also to understand the limitations of the simple scheduler that we are trying to ameliorate. These balance restrictions are necessary in order to prove good bounds for the greed scheduler, and that are relaxed in Section 6.

For conciseness, define the ***total work*** of a maximal level-i task t for the given memory hierarchy as $TW(\mathsf{t}) = \sum_{j=0}^{i} C_j Q^*(\mathsf{t}; M_j, B_j)$. For s a level-i strand or task that is not maximal, $TW(\mathsf{s}) = \sum_{j=0}^{i-1} C_j Q^*(\mathsf{s}; M_j, B_j)$, *i.e.*, assume all memory locations already reside in the level-i cache.

We say that a task t is ***recursively large*** if any child subtask t' of t (those nested directly within t) uses at least half the space used by t, *i.e.*, $S(\mathsf{t}'; B) \geq S(\mathsf{t}; B)/2$ for all B, and t' is also recursively large. Recursively large is quite restrictive, but it greatly simplifies both the analysis and further definitions here, as we can assume all subtasks of a level-i task are at least level-$(i-1)$.

Consider a level-i task t. We say that t is ***γ_i parallel at level-i*** if no more than a $1/\gamma_i$ fraction of the total work from all descendant maximal level-i strands of t fall along a single path in t's subdag, and no more than a $1/\gamma_i$ fraction of total work from all descendant level-$(i-1)$ maximal subtasks falls along a single path among level-$(i-1)$ tasks[5] in t's subdag. Moreover, we say that t is ***λ_i task heavy*** if the total work from strands comprises at most a $1/\lambda_i$ factor of the total work from level-$(i-1)$ subtasks.

The following theorem bounds the total running time of an algorithm using the greedy space-bounded scheduler.

Theorem 3 *Consider a PMH and dilation parameter $\sigma = 1$. Let t be a maximal level-i task, and suppose that*

- *t is recursively large, and*
- *all memory used by each maximal level-j (sub)task t' can be identified and preloaded into the level-j cache in parallel (in time $C_j S(\mathsf{t}'; B_j)/p_j B_j$) before executing any strands in t'.*

For each level-j, let γ_j and λ_j be values such that all maximal level-j descendent subtasks are γ_j parallel at level j and λ_j task heavy, respectively. Then t executes in $(TW(\mathsf{t})/p_i) \prod_{j=1}^{i}(1 + f_j/\gamma_j)(1 + p_{j-1}/\lambda_j)$ time on a greedy space-bounded scheduler.

Proof. To start with, observe that any maximal level-k task t_k uses at least $M_k/2$ space, and hence the scheduler assigns at most one level-k task to each level-k cache at any time. This fact follows because t_k's parent task t_{k+1} uses strictly more than M_k space, (as otherwise t_k is not maximal), and hence $S(\mathsf{t}_k; B_k) \geq S(\mathsf{t}_{k+1}; B_k)/2 > M_k/2$.

Suppose that the theorem holds for all maximal level-$(k-1)$ tasks, and inductively prove for a maximal level-k task t_k.

Since each level-$(k-1)$ cache is working on at most one subtask or thread at any time, we instead interpret our level-k computation as running at a courser granularity on a machine consisting of only a

[5]That is, ignore the parallelism within the level-$(i-1)$ subtasks — treat each subtask as an indivisible entity, forming a dag over level-$(i-1)$ subtasks.

single level-k cache with f_k processors (in place of the level-$(k-1)$ caches). We then interpret each level-$(k-1)$ task t_{k-1} as a serial computation that has work $(TW(\mathsf{t}_{k-1})/p_{k-1})\prod_{j=1}^{k-1}(1+f_j/\gamma_j)(1+p_{j-1}/\lambda_j)$ (*i.e.*, takes this amount of time when run on a single "processor"). Similarly, each level-k strand s is a serial computation having $TW(\mathsf{s})$ work. We thus interpret t_k as a computation having

$$\begin{aligned}
&\sum_{\text{subtasks } \mathsf{t}_{k-1}} \frac{TW(\mathsf{t}_{k-1})}{p_{k-1}} \prod_{j=1}^{k-1}\left(1+\frac{f_j}{\gamma_j}\right)\left(1+\frac{p_{j-1}}{\lambda_j}\right) + \sum_{\text{level-}k \text{ strands } \mathsf{s}} TW(\mathsf{s}) \\
\leq\ & \left(\frac{TW(\mathsf{t}_k) - C_k S(\mathsf{t}_k; B_k)/B_k}{p_{k-1}}\right) \prod_{j=1}^{k-1}\left(1+\frac{f_j}{\gamma_j}\right)\left(1+\frac{p_{j-1}}{\lambda_j}\right) + \sum_{\text{level-}k \text{ strands } \mathsf{s}} TW(\mathsf{s}) \\
\leq\ & \left(1+\frac{p_{k-1}}{\lambda_k}\right)\left(\frac{TW(\mathsf{t}_k) - C_k S(\mathsf{t}_k; B_k)/B_k}{p_{k-1}}\right) \prod_{j=1}^{k-1}\left(1+\frac{f_j}{\gamma_j}\right)\left(1+\frac{p_{j-1}}{\lambda_j}\right)
\end{aligned}$$

work after preloading the cache, where the first step of the derivation follows from the definition of Q^* and TW, and the second follows from t_k being λ_k task heavy. Since t_k is also γ_k parallel, it has a critical-path length that is at most a $1/\gamma_k$ factor of the work. Observing that the space-bounded scheduler operates as a greedy scheduler with respect to the level-k cache, we apply Brent's theorem to conclude that the space bounded scheduler executes this computation in

$$\begin{aligned}
&\left(\frac{1}{f_k}+\frac{1}{\gamma_k}\right)\left(1+\frac{p_{k-1}}{\lambda_k}\right)\left(\frac{TW(\mathsf{t}_k) - C_k S(\mathsf{t}_k; B_k)/B_k}{p_{k-1}}\right) \prod_{j=1}^{k-1}\left(1+\frac{f_j}{\gamma_j}\right)\left(1+\frac{p_{j-1}}{\lambda_j}\right) \\
=\ & \left(\frac{TW(\mathsf{t}_k) - C_k S(\mathsf{t}_k; B_k)/B_k}{p_k}\right) \prod_{j=1}^{k}\left(1+\frac{f_j}{\gamma_j}\right)\left(1+\frac{p_{j-1}}{\lambda_j}\right)
\end{aligned}$$

time on the f_k processors. Adding the $C_k S(\mathsf{t}_k; B_k)/p_k B_k$ time necessary to load the level-k cache completes the proof □

The $(1+f_j/\gamma_j)$ overheads arise from load imbalance and the recursive application of Brent's theorem, whereas the $(1+p_{j-1}/\lambda_j)$ overheads stem from the fact that strands block other tasks. For sufficiently parallel algorithms with short enough strands, the product of these overheads reduces to $O(1)$. This bound is then optimal whenever $Q^*(\mathsf{t}; M_i, B_i) = O(Q(\mathsf{t}; M_i, B_i))$ for all i.

Our new scheduler in Section 6 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 5 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.

5 Extending PCO with an Imbalance Metric

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 $nC/(C + p) \geq \frac{1}{2}\min(n, nC/p)$.*

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 $1/p$ of the instructions are misses in Q^*, an optimal scheduler either uses only one processor at a time (if $C \geq p$) or incurs a cache miss on almost every read or write (if $C < p$).

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 6, 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. 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. We use composition rules to accumulate the effective work across sequential ordering and parallel blocks. For a strand s with cache complexity $\widehat{Q}_\alpha(\mathsf{s})$ in task t, define $\widehat{Q}_\alpha(\mathsf{s}; M, B) = Q^*(\mathsf{s}; M, B) \times s(\mathsf{t}; B)^\alpha$. For a task $\mathsf{t} = \mathsf{s}_1; \mathsf{b}_1; \mathsf{s}_2; \mathsf{b}_2; \ldots; \mathsf{s}_k$, which by definition is the sequential composition alternating strands and parallel blocks, $\widehat{Q}_\alpha(\mathsf{t}; M, B) = \sum_{i=1}^{k-1} \left[\widehat{Q}_\alpha(\mathsf{s}_i; M, B) + \widehat{Q}_\alpha(\mathsf{b}_i; M, B)\right] + \widehat{Q}_\alpha(\mathsf{s}_k; M, B)$, the rules for forwarding cache state being the

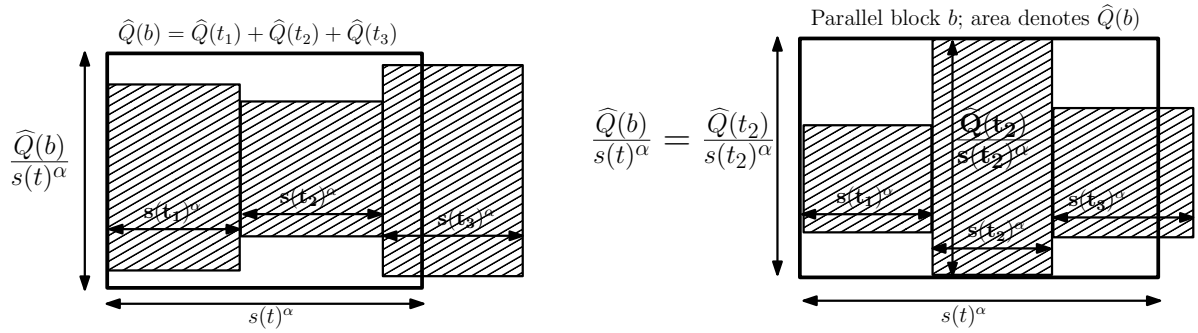

Figure 4: Two examples of equation 2 applied to a parallel block $\mathsf{b} = \mathsf{t}_1 \parallel \mathsf{t}_2 \parallel \mathsf{t}_3$ belonging to task t. The shaded rectangles represent the subtasks and the white rectangle represents the parallel block b. Subtask rectangles have fixed area ($\widehat{Q}_\alpha(\mathsf{t}_i)$, determined recursively) and *maximum* width $s(t_i)^\alpha$. (left) Cache complexity term dominates. In this example, the total area of b's subtasks is larger than any balance term, and determines the area $\widehat{Q}_\alpha(\mathsf{b})$. The height of the rectangle corresponding to b is set so that $\widehat{Q}_\alpha(\mathsf{b}) = \widehat{Q}_\alpha(\mathsf{t}_1) + \widehat{Q}_\alpha(\mathsf{t}_2) + \widehat{Q}_\alpha(\mathsf{t}_3)$. (right) Balance term dominates. Here, the height of a subtask t_2 determines the height of b; the other subtasks' area can be packed within the rectangle by reducing their width while preserving their area.

same as for Q^*. For a parallel block b in task t consisting of $j \geq 1$ subtasks $t_1, t_2, \ldots, t_j$,

$$\widehat{Q}_\alpha(\mathsf{b}; M, B; \kappa) \quad = \quad \max \begin{cases} s(\mathsf{t}; B)^\alpha \max_i \left\{ \frac{\widehat{Q}_\alpha(\mathsf{t}_i; M, B; \kappa')}{s(\mathsf{t}_i; B)^\alpha} \right\} & \text{depth dominated} \\ \sum_i \widehat{Q}_\alpha(\mathsf{t}_i; M, B; \kappa') & \text{work dominated} \end{cases} \tag{2}$$

where $\kappa' = \kappa$ if t fits in the cache (*i.e.*, $S(t; B) \leq M$) and $\emptyset$ otherwise. We refer to the term $s(\mathsf{t})^\alpha \max_i \{\widehat{Q}_\alpha(\mathsf{t}_i)/s(\mathsf{t}_i)^\alpha\}$ as the ***balance term***. The other term is the standard definition of cache complexity. The balance 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 an algorithm on input size n is α-*efficient* if $Q^*(n; M, B) = O(\widehat{Q}_\alpha(n; M, B))$, which happens if the second term in equation 2 dominates at most levels of recursion. The maximum α for which an algorithm is α-efficient specifies the ***effective parallelism*** s^α. $\widehat{Q}_\alpha(\cdot)$ is an attribute of an algorithm, and as such can be analyzed irrespective of the machine and the scheduler. Appendix B illustrates the analysis for $\widehat{Q}_\alpha(\cdot)$ and effective parallelism for several algorithms. Note that, as illustrated in Fig. 4(left) and these algorithms, good effective parallelism can be achieved even when there is significant work imbalance among subtasks. Finally, the balance term implicitly includes the span so we do not need a separate span (depth) cost in our model.

6 Scheduler

This section modifies the space-bounded scheduler to address some of the balance concerns discussed in Section 4. 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. 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, we have $f_i \leq (M_i/M_{i-1})^\beta$. This definition is analogous to the effective parallelism α of the algorithm. Aside from the overhead v_h (defined in the theorem), this bound is optimal in the PCO model for a PMH with $1/3$ 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 Intel Xeon 7500 from Figure 1), or if $\alpha \gg \beta$.

Theorem 5 *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 $\alpha' = \min\{\alpha, 1 - \log_f 2\}$, $f = max\{3, min_i\{f_i\}\}$. The runtime of t is no more than:*

$$\frac{\sum_{j=0}^{h-1} \widehat{Q}_\alpha(\mathsf{t}; M_j/3, B_j) \cdot C_j}{p_h} \cdot v_h, \quad \textit{where overhead } v_h \textit{ is } \; v_h = 2\prod_{j=1}^{h-1}\left(\frac{1}{k} + \frac{f_j}{(1-k)(M_j/M_{j-1})^{\alpha'}}\right).$$

As much of the scheduler matches the greedy scheduler from Section 4, only the differences are highlighted here. There are three main difference between this scheduler and greedy scheduler from Section 4. First, we fix the dilation to $\sigma = 1/3$ instead of $\sigma = 1$. Whereas reducing σ worsens the bound in Theorem 2 (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 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-$s = S(\mathsf{t}; B_i)$ task t ***desires*** $g_i(s)$ level-$(i-1)$ clusters, g_i to be specified later. When anchoring 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 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.

Operational details In order to prove performance bounds, we need to make changes to the greedy scheduler to restrict scheduling possibilities based on the above notions of saturation and allocation. We use much the same data structures from Section 4 In addition, for each of anchored anchored, maximal tasks t, we

also maintain the set of level-$(i-1)$ subclusters allocated to t, denoted by $allocated[\mathsf{t}]$. Also, when t is a maximal level-i task anchored to U, let $\mathcal{U}_t = allocated[\mathsf{t}]$ (instead of all the children of U for the greedy scheduler). The scheduling rules modify as follows:

strands Cache U_j is ***strand-ready*** if there exists a cache-to-processor path of caches $U_j, U_{j-1}, \ldots, U_1, p$ descending from U_j down to processor P such that: 1) $U_j, \ldots, U_1$ are unsaturated (additional restriction), 2) $U_{j-1}, \ldots, U_1$ are unallocated (additional restric), 3) each $U_k \in \{U_j, \ldots, U_1\}$ has σM_k space available, and 4) P is idle. All other definitions and actions remain the same as earlier, except that s is allocated all caches along the ready path except for V itself.

tasks Suppose there exists some level-$(j-1) < (i-1)$ strand-ready descendant cache U_{j-1} of $V \in \mathcal{U}_t$, and let U_j be its parent cache. Suppose also that no cache between V and U_j is saturated (inclusive of V and U_j), and no cache between V and U_{j-1} is allocated (exclusive of V but inclusive of U_{j-1}). If there exists level-j subtask $\mathsf{t}' \in \mathcal{R}(\mathsf{t})$ such that U_j has $S(\mathsf{t}'; B_j)$ space available, then remove t' from $\mathcal{R}(\mathsf{t})$. Anchor t' at U_j, allocate subclusters to t' according to Section 6, create $\mathcal{R}(\mathsf{t}')$ and $\mathcal{S}(\mathsf{t}')$, and reduce U_j's remaining capacity appropriately. Then schedule and execute the first strand of t' as above.

Note that we have chosen our definition of saturation so that these rules can be implemented from top-down locally without considering the entire hierarchy — specifically, if a cache is unsaturated, then it has at least one unsaturated, unallocated child cache with at least a σ-fraction of the capacity available. Finally, operations performed when a strand completes remain the same.

Scheduler details. We now describe the two missing details of the scheduler, namely the notion 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:

$$\mu(U) = \begin{cases} \min\left\{1, \frac{1}{kf_i}\sum_{i=1}^{f_i} \mu'(V_i)\right\} & \text{if is a level-}(\geq 2)\text{ cluster} \\ \min\{1, \frac{x}{f_1 k}\} & \text{if } U \text{ level-1 cluster with } x \text{ allocated processors} \end{cases} \tag{3}$$

and

$$\mu'(V) = \begin{cases} 1 & \text{if } V \text{ is allocated} \\ \mu(V) & \text{otherwise} \end{cases}, \tag{4}$$

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 choose $g_i(s) = \min\{f_i, \max\{1, \lfloor f(3s/M_i)^{\alpha'} \rfloor\}\}$, where $\alpha' = \min\{\alpha, 1 - \log_f 2\}$, $f = \max\{3, \min_i\{f_i\}\}$ [6]. What this says is that a maximal level-i task is allocated one subcluster when it has size $s(t; B_i) = \frac{M_i}{3B_i f_i^{1/\alpha'}}$, 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 $s(t; B_i) = \frac{M_{i-1}}{3B_i}$. We define $g(s) = g_i(s)p_{i-1}$ if $s \in (M_{i-1}/3, M_i/3]$. We also assume that $\alpha \geq \log_2(1 + 1/f_i)$.

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.

We now proceed to the analysis of this scheduler, proving several lemmas leading up to Theorem 5, which is similar in form to Theorem 3. First, the following lemma implies that the capacity restriction of each cache is subsumed by the scheduler only assigning tasks to unallocated, unsaturated clusters. It follows that if U is allocated to t, and U is unsaturated, then the ready-task and ready-strand lists, $\mathcal{R}(\mathsf{t})$ and $\mathcal{S}(\mathsf{t})$, are empty.

Lemma 6 *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 U has at least $M_i/3$ capacity remaining when $f_i > 3$. The cases of $f_i = 1, 2$ are trivial. A task t anchored to U with desire x has size at most $(M_i/3)((x+1)/f_i)^{1/\alpha'}$, where $\alpha' = \min\{\alpha, 1\} \leq 1$. Therefore, the ratio of space of t to its desire x is at most $(M_i/3x)((x+1)/f_i)^{1/\alpha'} \leq \max\{(M_i/3)((f_i+1)/f_i)^{1/\alpha'}/f_i, (M_i/3)2^{1/\alpha'}/f_i^{1/\alpha'}\} \leq \max\{(1+1/f_i)^{1/\alpha'}, 2^{1/\alpha'} f_i^{1-1/\alpha'}\} \times (M_i/3f_i)$. The first term inside the max expression is at most 2 as $\alpha' \geq \log_2(1 + 1/f_i)$. The second is at most 1 as $\alpha' \leq 1 - \log_{f_i} 2$. Therefore, the whole expression is at most $(2M_i/3f_i)$. Thus, if the number of allocated subclusters is less than f_i, then the capacity used is less than $(2/3)M_i$. □

Latency added cost. Section 5 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

[6] Note that scheduling decisions are trivial when $f_i = 1, 2$

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 ($\rho(\cdot)$) 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 the effective work of a strand s directly enclosed in task t to be

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

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

$$\widehat{W}^*_\alpha(\mathsf{t}) = \sum_{i=1}^{k} \widehat{W}^*_\alpha(c_i) \quad \text{(Serial composition).} \tag{5}$$

For a parallel block $\mathsf{b} = \mathsf{t}_1 \| \mathsf{t}_2 \| \ldots \| \mathsf{t}_k$ immediately inside task t',

$$\widehat{W}^*_\alpha(\mathsf{b}) = \max \left\{ s(\mathsf{t}'; B)^\alpha \max_i \left\{ \frac{\widehat{W}^*_\alpha(\mathsf{t}_i)}{s(\mathsf{t}_i; B)^\alpha} \right\}, \sum_i \widehat{W}^*_\alpha(\mathsf{t}_i) \right\} \text{ (Parallel composition).} \tag{6}$$

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 $\widehat{W}^{(i)}_\alpha(\cdot)$ 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 $i \in [h-1]$, $\widehat{W}^{(i)}_\alpha(\mathsf{c})$ 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 $\widehat{W}^{(0)}_\alpha(\mathsf{c})$ 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 7 *For an h-level PMH with $B = B_j$ for all $1 \leq j \leq h$ and computation A, we have*

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

Proof. The proof is based on induction on the structure of the computation (in terms of it's 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

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

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

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

For a parallel block b inside task t consisting of tasks $\{\mathsf{t}_i\}_{i=1}^m$, consider the equation 6 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 6 for evaluating each of $\widehat{W}^{(l)}_\alpha(\mathsf{b})$ and suppose that the k_l-th term (denoted by $T^{(l)}_{k_l}$) on the right hand side determines the value of $\widehat{W}^{(l)}_\alpha(\mathsf{b})$. Then,

$$\frac{\widehat{W}^*_\alpha(\mathsf{b})}{s(\mathsf{t}; B)^\alpha} = T_k \leq \sum_{l=0}^{h-1} T^{(l)}_k \leq \sum_{l=0}^{h-1} T^{(l)}_{k_l} = \frac{\sum_{l=0}^{h-1} \widehat{W}^{(l)}_\alpha(\mathsf{b})}{s(\mathsf{t}; B)^\alpha}, \tag{7}$$

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 5, 6. ρ could have been replaced with any other kind of work and ρ_i with it's 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 8 *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 7 then implies that for any computation c:

$$\widehat{W}^*_\alpha(\mathsf{c}) \leq \sum_{i=0}^{h-1} \widehat{Q}_\alpha(\mathsf{c}; M_i/3, B) \cdot C_i. \tag{8}$$

□

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 also consistent with Theorem 3 for the simple greedy scheduler, here we address complications that arise due to partially allocated caches and subtasks skipping levels in the hierarchy.

Lemma 9 *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 $\alpha' = \min\{\alpha, 1 - \log_f 2\}$, $f = \max\{3, \min_i\{f_i\}\}$. 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:*

$$\frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}} \cdot v_i, \quad \textit{where overhead } v_i \textit{ is } \; v_i = 2\prod_{j=1}^{i-1}\left(\frac{1}{k} + \frac{f_i}{(1-k)(M_i/M_{i-1})^{\alpha'}}\right).$$

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 run 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

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

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

$$\sum_{x\in\chi(N_i)} \frac{\widehat{W}^*_\alpha(x)}{(1-k)g(S(x;B))}\cdot v_{i-1} \qquad = \sum_{x\in\chi(N_i)} \frac{\widehat{W}^*_\alpha(x)}{s(x;B)^\alpha}\frac{s(x;B)^\alpha}{(1-k)g(S(x;B))}\cdot v_{i-1}$$

$$\leq \left(\sum_{x\in\chi(N_i)} \frac{\widehat{W}^*_\alpha(x)}{s(x;B)^\alpha}\right)\frac{(M_{i-1}/3B)^\alpha}{(1-k)p_{i-1}}\cdot v_{i-1} \quad \leq \frac{\widehat{W}^*_\alpha(N_i)}{s(N_i;B)^\alpha}\cdot\frac{(M_{i-1}/3B)^\alpha}{(1-k)p_{i-1}}\cdot v_{i-1} \text{ (by the defn. of } \widehat{W}^*_\alpha())$$

$$= \frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}}\cdot\frac{r(M_{i-1}/3B)^\alpha}{s(N_i;B)^\alpha}\cdot\frac{v_{i-1}}{1-k} \qquad \leq \frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}}\cdot\frac{q(M_{i-1}/3B)^\alpha}{s(N_i;B)^\alpha}\cdot\frac{v_{i-1}}{1-k}$$

$$\leq \frac{\widehat{W}^*_\alpha(N_i)}{rp_{i-1}}\cdot\frac{f_i}{(1-k)(M_i/M_{i-1})^{\alpha'}}\cdot v_{i-1} \qquad \text{(by the definition of function } g).$$

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

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

□

Theorem 5 follows from Lemmas 8 and 5, starting on a system with no utilization.

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*, pages 1–10, 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 and P. B. Gibbons. Effectively sharing a cache among threads. In *SPAA*, 2004.

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

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

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

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

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

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

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

[16] 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.

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

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

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

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

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

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

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

Appendix

Section A demonstrates how to analyze for Q^* exhibits that $Q^* = O(Q)$ for some example algorithms. Section B analyzes many algorithms with respect to the effective cache complexity.

A Example PCO Algorithm 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 example algorithm that is easy to analyse in this model. A quicksort on size $n > M$ input involves one cache miss for finding a pivot, $\lceil n/B \rceil$ misses to filter the input and $Q^*(i; M, B) + Q^*(n - i; M, B)$ cache misses for the subtasks (assuming pivot is the i-th element). Solving this recurrence relation (such as in lemma 9.2 of [20]) shows that the expected value of $Q^*(n; M, B)$ is $O(\lceil n/B \rceil \log n)$. In [9], we present a sample sort that can also be analyzed in the PCO model and has $Q^*(n; M, B) = O(\lceil n/B \rceil \log_M n)$. Note that in the original analysis in both papers, the cache complexities of the parallel subtasks were already analyzed independently assuming no data reuse.

The rest of the algorithms in Table 1 can be similarly analyzed without difficulty.

B Examples of algorithms and their complexities

In this section, we will illustrate how to estimate α, the degree of parallelism and compute $\widehat{Q}_\alpha()$ metrics as a function of α with the help of a few algorithms. We will start by computing the effective cache complexity of a simple scan over an array, as introduced in section A, which touches each element by touching the middle element and then recursively splitting the array in half around this element. For this algorithm,

$$\widehat{Q}_\alpha(s; M, B; \kappa) \quad = \quad \begin{cases} O(1) & s \leq B, M > 0 \\ \max\left\{ (\lceil s/B \rceil)^\alpha \, \frac{\widehat{Q}_\alpha(\frac{s}{2}; M, B; \kappa)}{\lceil s/2B \rceil^\alpha}, \; 2\widehat{Q}_\alpha(\frac{s}{2}; M, B; \kappa) \right\} + O(\lceil s/B \rceil^\alpha), & otherwise. \end{cases}$$

$\widehat{Q}_\alpha(n) = O(1)$ for $s < B, M > 0$ because only the first level of recursion after $s < B$ (and one other node in the recursion tree rooted at this level, in case this call accesses locations across a boundary) incurs at most one cache miss (which costs $O(1^\alpha)$) and the cache state, which includes this one cache line, is carried forward to lower levels preventing them from any more cache misses. For $\alpha < 1, M > 0$, the summation term dominates (second term in the max) and the recursion solves to $O(\lceil s/B \rceil)$, which is optimal. If $M = 0$, then $O(s)$, for $\alpha < 1$.

We will see the analysis for some matrix operations. Consider a recursive version of matrix addition where each quadrant of the matrix is added in parallel. Here, a task (corresponding to a recursive addition) comprises a strand for the fork and join points and a parallel block consisting of four smaller tasks on a matrix one fourth the size. So,

$$\widehat{Q}_\alpha(s;M,B) \;=\; \begin{cases} O(1), & s \leq B, M > 0 \\ O(\lceil s/B \rceil^\alpha) + 4\widehat{Q}_\alpha(s/4;M,B), & B < s \end{cases}$$

which implies $\widehat{Q}_\alpha(s;M,B) = O(\lceil s/B \rceil)$ if $\alpha < 1, M > 0$, and $\widehat{Q}_\alpha(s) = O(\lceil s/B \rceil^\alpha)$ if $\alpha < 1, M > 0$. These bounds imply that matrix addition is work effecient only when parallelism is limited to $\alpha < 1$. Further, when $M = 0$, $\widehat{Q}_\alpha(s;M,B) = O(s)$ for $\alpha < 1$.

Matrix multiplication, as described in Figure 5 consists of fork and join points and two parallel blocks. Each parallel block is a set of 4 parallel tasks each, each of which is a matrix multiplication on matrices four times smaller. Note that if at some level of recursion, the first parallel block fits in cache, then the next parallel block can reuse the same set of locations. Therefore, when $M > 0$,

$$\widehat{Q}_\alpha^{Mult}(s;M,B;\kappa) \;=\; \begin{cases} 0, \quad \text{if } \kappa \text{ includes all relevant locations} & \\ O(1) \quad \text{if } \kappa \text{ does not include the relevant cache lines,} & s \leq B \\ O(\lceil s/B \rceil^\alpha) + 4\widehat{Q}_\alpha^{Mult}(s/4;M,B;\kappa), \kappa_p + 4\widehat{Q}_\alpha^{Mult}(s/4;M,B;\kappa_p), & B < s \leq M \\ O(\lceil s/B \rceil^\alpha) + 8\widehat{Q}_\alpha^{Mult}(s/4;M,B;\emptyset), & otherwise, \end{cases}$$

which implies $\widehat{Q}_\alpha^{Mult}(s) = O(\lceil s/M \rceil^{3/2} \lceil M/B \rceil)$ for all $\alpha < 1$. Note that a different version of matrix multiplication with eight parallel recursive calls may have greater parallelism, but may require superlinear stack space. Further, for $M = 0$, $\widehat{Q}_\alpha^{Mult}(s) = O(s^{3/2})$ when $\alpha < 1$.

Matrix Inversion, as described in Figure 5, consists of fork and join points, and 10 parallel blocks (each containing only on subtask). Two of these blocks are matrix inversions on matrices four times smaller, six are multiplications on matrices of the same size and two are additions on matrices of the same size. The effective cache complexity of the parallel blocks is dominated by the balance term only in the case of the matrix inversion (because they operate on much smaller matrices). The effective cache complexity is

$$\widehat{Q}_\alpha^{Inv}(s;M,B) = O(\lceil s/B \rceil^\alpha) + 2\left(\lceil s/B \rceil\right)^\alpha \left(\frac{\widehat{Q}_\alpha^{Inv}(s/4;M,B;\emptyset)}{(\lceil s/4B \rceil)^\alpha} \right) + 6\widehat{Q}_\alpha^{Mult}(s/4;M,B;\emptyset) + 2\widehat{Q}_\alpha^{Add}(s/4;M,B;\emptyset),$$

when $s > M > 0$ (other cases have been left out for brevity). It can be shown that $\widehat{Q}_\alpha^{Inv}(s) = O(\lceil s/M \rceil^{3/2} \lceil M/B \rceil)$ when $\alpha < 1, M > 0$, indicating that inversion has a parallelism of up to $\alpha \to 1$. For $M > 0$, $\widehat{Q}_\alpha^{Inv}(s;M,B) = O(s^{3/2})$ for $\alpha < 1$. The gap of about $s^{1/2}$ between effective work and parallelism as indicated by the maximum α is consistent with the fact that under the conventional definitions, this inversion algorithm has a depth of about $s^{1/2}$.

In all the above examples, calls to individual tasks in a parallel block are relatively balanced. We now show an example – parallel deterministic cache oblivious sorting from [9] (the algorithm is also described in http://reports-archive.adm.cs.cmu.edu/anon/2009/CMU-CS-09-134.pdf) outlined in Figure 5 as sample sort – where this is not the case. This sorting uses five primitives: prefix sums, matrix transpose, bucket transpose, merging and simple merge sort. We will first analyze these five.

The prefix sum algorithm works as follows: imagine a binary tree over the array. The algorithm involves two phases: one in which elements are summed bottom-up recursively along this tree and the second phase in which these partial sums are passed down and the answer computed. Each of these phases involves a

simple balanced recursion where each task of size s adds two integers apart from two recrsive tasks on an array half the size. Therefore, the efective work is simply $\widehat{Q}_\alpha^{PS}(s; M, B) = O(\lceil s/B \rceil)$ for $\alpha < 1, M > 0$.

Bucket tranpose involves work that is linear in terms of space, but the recursion is highly imbalanced. A call (task) to bucket transpose with space s invokes four parallel bucket tranposes on sizes s_1, s_2, s_3, s_4 such that $\sum_{i=1}^{4} s_i = s$. The only other work here is fork and join points. When this recursion reaches the leaf (of size s), an array copy (completely parallel) whose effective work is s (for $\alpha < 1$) is done. For an internal node of this recursion, when $s > B, M > 0$,

$$\widehat{Q}_\alpha^{BT}(s; M, B; \emptyset) = O(\lceil s/B \rceil^\alpha) + \max \left\{ \max_i \left\{ \widehat{Q}_\alpha^{BT}(s_i; M, B; \emptyset) \frac{\lceil s/B \rceil^\alpha}{\lceil s_i/B \rceil^\alpha} \right\}, \sum_{i=1}^{4} \widehat{Q}_\alpha^{BT}(s_i; M, B; \emptyset) \right\}.$$

If we inductively assume that for $\alpha < 1$, $\widehat{Q}_\alpha^{BT}(s_i; M, B; \emptyset) \leq k(\lceil s_i/B \rceil - \lceil s_i/B \rceil^{(1+\alpha)/2})$, then the i-th balance term is at most $k(\lceil s/B \rceil^\alpha \lceil s_i/B \rceil^{1-\alpha} - \lceil s_i/B \rceil)$, which is dominated by the summation on the right, when $\alpha < 1$. Further the summation is also less than $k(\lceil s/B \rceil - \lceil s/B \rceil^{(1+\alpha)/2})$, when $s > k_1 B$ for some constant k_1. Filling in other cases of the recurrence will show that $\widehat{Q}_\alpha^{BT}(s; M, B) = O(\lceil s/B \rceil)$, for $\alpha < 1, M > 0$ (also $\widehat{Q}_\alpha^{BT}(s; 0, B) = O(s)$ for $\alpha < 1$). Like wise, matrix tranpose has $\widehat{Q}_\alpha^{MT}(s; M > 0, B) = O(\lceil s/B \rceil)$.

Merging two arrays of combined size s can be shown to have $\widehat{Q}_\alpha^{Merge}(s; M, B) = O(\lceil s/B \rceil)$, and merge sort on array of size s has $\widehat{Q}_\alpha^{MS}(s; M, B) = O(\lceil s/B \rceil \log s/M)$ for $\alpha < 1$. Now we are ready to analyze the complexity of the cache oblivious sample sort COSORT. Each call to such a COSORT with input size s starts with a completely balanced parallel block of $\sqrt{s}$ recursive calls to COSORT with input size $\sqrt{s}$. After this $(s/\log s)$ pivots are picked and merge sorted, the subarrays are split (same complexity as merging) with the pivots, offsets in to target buckets computed with prefix sum (and matrix tranpose operations) and the elements are sent in to buckets using bucket transpose. At the end, there is a parallel block sorting the $\sqrt{s}$ buckets (not all of same size) using COSORT. All the operations except the recursive COSORT have effective work that sums up $O(s)$, and cache complexity that sums up $O(\lceil s/B \rceil)$ (if starting with empty cache) when $\alpha < 1$. Therefore, for $\alpha < 1$, $s > M$,

$$\widehat{Q}_\alpha^{SS}(s; M, B) = O\left(\lceil \frac{s}{B} \rceil\right) + \sqrt{s}\widehat{Q}_\alpha^{SS}(\sqrt{s}; M, B) + \max \left\{ \max_i \{\widehat{Q}_\alpha^{SS}(s_i; M, B) \frac{\lceil s/B \rceil^\alpha}{\lceil s_i/B \rceil^\alpha}\}, \sum_{s_i \leq s \log s, \sum_i s_i = s} \widehat{Q}_\alpha^{SS}(s_i; M, B) \right.$$

Just as in the case of bucket tranpose, the balance terms are dominted by the summation on the right .The base case for this recursion is $\widehat{Q}_\alpha^{SS}(s; M, B) = O(\lceil s/B \rceil)$ for $s < M/k$ for some small constant k. This recursion implies $\widehat{Q}_\alpha^{SS}(s) = O(\lceil s/B \rceil \log_{M+2}(s))$.

function **QuickSort**(A)
if ($\#A \leq 1$) then return A
p = A[rand($\#A$)] ;
Les = **QuickSort**($\{a \in A | a < p\}$) $\|$
Eql = $\{a \in A | a = p\}$ $\|$
Grt = **QuickSort**($\{a \in A | a > p\}$) ;
return Les ++ Eql ++ Grt

$$\widehat{Q}_\alpha(n; M, B) \quad = \quad O((n/B)\log(n/(M+1)))$$

In the code both $\|$ and $\{\ \}$ indicate parallelism. The bounds are expected case.

function **SampleSort**(A)
if ($\#A \leq 1$) then return A
parallel for $i \in [0, \sqrt{n}, \ldots, n)$
  SampleSort($A[i, ..., i + \sqrt{n})$) ;
$P[0, 1, \ldots, \sqrt{n}) =$ findPivots(A) ;
$B[0, 1, \ldots, \sqrt{n}) =$ bucketTrans(A, P) ;
parallel for $i \in [0, 1, \ldots, \sqrt{n})$
  SampleSort(B_i) ;
return flattened B

$$\widehat{Q}_\alpha(n; M, B) \quad = \quad O((n/B)\log_{M+2} n)$$

This version is asymptotically optimal for cache misses. The pivots partition the keys into buckets and bucketTrans places the keys from each sorted subset in A into the buckets B. Each bucket ends up with about $\sqrt{n}$ elements [9].

function **MM**(A,B,C)
if ($\#A \leq k$) then return MMsmall(A,B)
$(A_{11}, A_{12}, A_{21}, A_{22}) =$ split(A) ;
$(B_{11}, B_{12}, B_{21}, B_{22}) =$ split(B) ;
$C_{11} =$ **MM**(A_{11}, B_{11}) $\|$ $C_{21} =$ **MM**(A_{21}, B_{11}) $\|$
$C_{12} =$ **MM**(A_{11}, B_{12}) $\|$ $C_{22} =$ **MM**(A_{21}, B_{12}) ;
$C_{11} +=$ **MM**(A_{12}, B_{21}) $\|$ $C_{21} +=$ **MM**(A_{22}, B_{21}) $\|$
$C_{12} +=$ **MM**(A_{12}, B_{22}) $\|$ $C_{22} +=$ **MM**(A_{22}, B_{22}) ;
return join$(C_{11}, C_{12}, C_{21}, C_{22})$

$$\widehat{Q}_\alpha(n; M, B) \quad = \quad O((n^{1.5}/B)/\sqrt{M+1})$$

Multiplying two $\sqrt{n} \times \sqrt{n}$ matrices. The split operation has to touch any of the four elements at the center of the matrices A, B. The eight recursive subtasks are divided in to two parallel blocks of four tasks each. Can easily be converted to Strassen with $\widehat{Q}_\alpha(n; M, B) = (n^{(\log_2 7)/2}/B)/\sqrt{M+1}$ and the same depth.

function **MatInv**(A)
if ($\#a \leq k$) then InvertSmall(A)
$(A_{11}, A_{12}, A_{21}, A_{22}) =$ split(A) ;
$A_{22}^{-1} =$ **MatInv**(A_{22}) ;
$S = A_{11} - A_{12}A_{22}^{-1}A_{21}$;
$C_{11} =$ **MatInv**(S) ;
$C_{12} = C_{11}A_{12}A_{22}^{-1}$; $C_{21} = -A_{22}^{-1}A_{21}C_{11}$;
$C_{22} = A_{22}^{-1} + A_{22}^{-1}A_{21}C_{11}A_{12}A_{22}^{-1}$;
return join$(C_{11}, C_{12}, C_{21}, C_{22})$

$$\widehat{Q}_\alpha(n; M, B) \quad = \quad O((n^{1.5}/B)/\sqrt{M+1})$$

Inverting a $\sqrt{n} \times \sqrt{n}$ matrix using the Schur complement (S). The split operation has to touch any of the four elements at the center of matrix A. The depth is $O(\sqrt{n})$ since the two recursive calls cannot be made in parallel. The parallelism comes from the matrix multiplies.

Figure 5: Examples of quicksort, sample sort, matrix multiplication and matrix inversion. All the results in this table are for $0 < \alpha < 1$. Therefore, these algorithms have parallelism of up to $\alpha \to 1$.

function **BHT**$(P, (x_0, y_0, s))$
 if $(\#P = 0)$ then return EMPTY
 if $(\#P = 1)$ then return LEAF($P[0]$)
 $x_m = x_0 + s/2$; $y_m = y_0 + s/2$;
 $P_1 = \{(x, y, w) \in P | x < x_m \wedge y < y_m\}$ ||
 $P_2 = \{(x, y, w) \in P | x < x_m \wedge y \geq y_m\}$ ||
 $P_3 = \{(x, y, w) \in P | x \geq x_m \wedge y < y_m\}$ ||
 $P_4 = \{(x, y, w) \in P | x \geq x_m \wedge y \geq y_m\}$ ||
 $T_1 =$ **BHT**$(P_1, (x_0, y_0, s/2))$ ||
 $T_2 =$ **BHT**$(P_2, (x_0, y_0 + y_m, s/2))$ ||
 $T_3 =$ **BHT**$(P_3, (x_0 + x_m, y_0, s/2))$ ||
 $T_4 =$ **BHT**$(P_4, (x_0 + x_m, y_0 + y_m, s/2))$ ||
 $C =$ CenterOfMass(T_1, T_2, T_3, T_4) ;
 return NODE(C, T_1, T_2, T_3, T_4)

$$\widehat{Q}_\alpha(n; M, B) \quad = \quad O((n/B)\log(n/(M+1)))$$

The two dimensional Barnes Hut code for constructing the quadtree. The bounds make some (reasonable) assumptions about the balance of the tree.

---

function **ConvexHull**(P)
 $P' =$**SampleSort**(P by x coordinate) ;
 return **MergeHull**(P')

function **MergeHull**(P)
 if $(\#P = 0)$ then return EMPTY
 Touch $P[n/2]$;
 $H_L =$ **MergeHull**$(P[0, \ldots, n/2))$ ||
 $H_R =$ **MergeHull**$(P[n/2, \ldots, n))$;
 return bridgeHulls(H_L, H_R)

$$\widehat{Q}_\alpha(n; M, B) \quad = \quad O((n/B)\log_{M+2} n)$$

The two dimensional Convex Hull code (for finding the upper convex hull). The bridgeHulls routine joins two adjacent hulls by doing a dual binary search and only requires $O(\log n)$ work. The cost is dominated by the sort.

---

function **SparseMxV**(A, x)
 parallel for $i \in [0, \ldots, n)$
  $r_i =$ sum$(\{v \times x_j | (v, j) \in A_i\})$
 return r

$$\widehat{Q}_\alpha(n, m; M, B) \quad = \quad O(m/B + n/(M+1)^{1-\gamma})$$

Sparse vector matrix multiply on a matrix with n rows and m non-zeros. We assume the matrix is in compressed sparse row format and A_i indicates the i^{th} row of A. The bound assumes the matrix has been diagonalized using recursive separators and the edge-separator size is $O(n^\gamma)$ [7]. The parallel for loop at the top should be done in a divide and conquer fashion by cutting the index space in half. At each such fork, the algorithm should touch at least one element from the middle row. The sum inside the loop can also be executed in parallel.

Figure 6: Examples of Barnes Hut tree construction, convex hull, and a sparse matrix by dense vector multiply. All the results in this table are for $0 < \alpha < 1$. Therefore, these algorithms have parallelism of up to $\alpha \to 1$.