Provably Efficient Scheduling of Dynamically Allocating Programs on Parallel Cache Hierarchies

Guy E. Blelloch*, Phillip B. Gibbons†, and Harsha Vardhan Simhadri‡,
* *Carnegie Mellon University, Email:guyb@cs.cmu.edu*
† *Carnegie Mellon University, Email: gibbons@cs.cmu.edu*
‡ *Microsoft Research India, Email: harsha.v.simhadri@gmail.com*

***Abstract*—Thread schedulers are designed to dynamically map parallel programs to processors to optimize performance metrics including memory footprint, number of cache misses at each cache level, and load balance, so as to minimize the total running time of the program. Programs with *dynamic memory allocation* pose particular challenges for thread schedulers, and indeed prior schedulers that are provably cache- and time-efficient on multi-level cache hierarchies require *static* memory allocation. Not only do many thread schedulers fail to reuse memory effectively, but there is often an inherent tradeoff between parallelism and memory use in algorithms.**

In this paper, we present the first runtime thread scheduler for multi-level cache hierarchies, called the *space-bounded recursive-PDF scheduler*, that is provably space-, cache-, and time-efficient for parallel programs that dynamically allocate memory. Our bounds hold for nested parallel programs with good *regularity* as measured by the *effective cache complexity*—a program-centric metric. The cache and time bounds are asymptotically optimal, while the space bound is asymptotically optimal for highly parallel and regular programs.

***Keywords*-Thread schedulers, Memory allocation, Space-bounded schedulers, Work stealing, Cache hierarchies**

I. Introduction

A popular and effective paradigm for programming shared memory parellel processors is to use a language supporting fine-grained nested parallel programming (e.g., Cilk++ [1], X10 [2], NESL [3], Fork-Join Java [4], Habanero C [5]) in conjunction with a smart runtime thread scheduler that maps program tasks to processing cores as the computation unfolds. There has been significant work over the past 20 years on designing runtime thread schedulers and studying their performance. Theoretical guarantees on running time are typically provided via bounds on the program's memory footprint, number of cache misses, and load balance. While early work often focused on multicores with a single level of cache, recent work has focused on multi-level cache hierarchies used in modern multicore processors (see Fig. 3c).

Challenges of Dynamic Memory Allocation. Programs that dynamically allocate memory, either explicitly (e.g., malloc/free) or implicitly (in garbage collected languages), pose particular challenges for thread schedulers. Consider a nested-parallel version of the Quicksort algorithm in Fig. 1 written in the NESL language [3], [6]. At each level of

```
function Quicksort(A) =
  if (#A < 2) then A
  else let pivot = A[#A/2];
         ls    = {e in A| e < pivot};
         eq    = {e in A| e == pivot};
         gr    = {e in A| e > pivot};
         ret   = {Quicksort(v):v in [ls,gr]};
       in ret[0] ++ eq ++ ret[1];
```

Figure 1: NESL code for dynamically allocated Quicksort

recursion, array `A` is split into three smaller arrays, `ls`, `eq` and `gr`, which are recursively sorted (except for `eq`), and put together before returning. Whereas a naive memory allocation scheme might assign each of the $O(n \log n)$ variables a different location, $O(n)$ locations are sufficient to execute this in parallel. The locations assigned to `A` and temporary variables can be recycled after `A` is split, and hence, one can use two arrays of size $O(n)$ to hold subarrays `A` and `ls`, `eq` and `gr` across all levels of recursion so that they can be executed in parallel. At each level of recursion the input subarray would be mapped to one chunk, and the splits to the other chunk. At the next level of recursion, the mapping to the memory chunks is reversed.

A smart thread scheduler should enable such an allocation scheme that minimizes the memory footprint and uses the reduced footprint to **minimize cache misses**. Namely, once a subcomputation fits into a cache at a given level of the multicore cache hierarchy, it should be executed by the subset of the cores that share the cache incurring misses at this cache only for the initial load. For example, if 15 cores shared a 30MB L3 cache in Fig. 3c as in an Intel Xeon E7-8870, the scheduler could map a subtask of quicksort with 1.5M integers — which requires two 6GB chunks for the data, and three 6GB chunks for auxiliary index structures.

Moreover, for some algorithms, the appropriate memory allocation can be part of a **space vs. parallelism trade-off**. Consider a nested-parallel matrix multiplication of two $n \times n$ matrices in which eight recursive calls to smaller matrix multiplications on $n/2 \times n/2$ matrices can be invoked in parallel. This maximizes the available parallelism, but could require up to $\Theta(n^3)$ space as up to n^3 partial results may be live concurrently, e.g., if a breadth-first schedule were used. An alternative is to limit the parallelism as in Figure 2.

```
function MM(A,B,C) =
  if (nr(A)<=k & nc(A)<=k & nc(B)<=k) then MM-Seq(A,B,C)
  else if (nr(A)>=nc(A) & nr(A)>=nc(B))
    then let (A1,A2) = split-rows(A);
             (C1,C2) = split-rows(C);
           C1 += MM(A1,B) || C2 += MM(A2,B)
        in join-rows(C1,C2)
  else if (nc(B)>=nr(A) & nc(B)>=nc(A))
    then let (B1,B2) = split-cols(B);
             (C1,C2) = split-cols(C);
           C1 += MM(A,B1) || C2 += MM(A,B2)
        in join-cols(C1,C2)
  else let (A1,A2) = split-cols(A); // NO PARALLELISM
           (B1,B2) = split-rows(B);
      in C += MM(A1,B1); C += MM(A2,B)
```

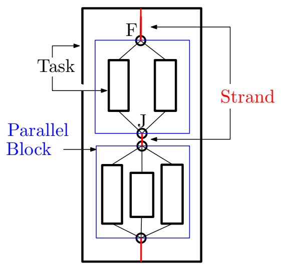
Figure 2: Pseudocode for nested-parallel Matrix Multiplication where minimizing space is prioritized over exposing parallelism. Function calls separated by "||" are executable in parallel, while ";" represents sequential composition.

Prior Work Falls Short. Our goal is to design a runtime thread scheduler for multi-level parallel cache hierarchies that is provably space-, cache-, and time-efficient for parallel programs that dynamically allocate memory. Prior work has achieved this for multicore processors with a single level of private caches [7]–[9] or a single shared cache [10], [11]. For multi-level parallel cache hierarchies, prior work [12]–[16] has provided provably cache- and time-efficient thread schedulers only for programs that *statically allocate* memory, i.e., programs that do all heap allocation at the start of execution and manage its thread-safety internally.

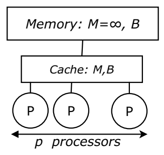
Among these, *space-bounded schedulers* [14], [15] for parallel cache hierarchies (Fig. 3c) and *parallel depth-first (PDF) schedulers* [10], [12] for machines with a single shared cache (Fig.3b) are relevant to our context. The idea of space-bounded schedulers is to "anchor" a set of tasks to a cache that (just) fits the set, and then schedule all subtasks on caches or cores below it in the hierarchy.

The PDF scheduler executes the DAG greedily prioritizing instructions based on their order in a depth-first order. Therefore, its space requirement is closely related to the depth-first schedule, and exceeds it by the amount of space required by the ***premature*** nodes that are executed out of the depth-first order. If each instruction takes one step to execute, the number of premature nodes is no more than $(p-1)d$ for a depth d DAG on a p processor machine [10] (see Fig 4). This limits the space of the PDF schedule relative to the depth-first schedule if we limit the amount of space dynamically allocated per instruction. For highly parallel algorithms with low depth, such bounds on space are very competitive. However, the amount of time an instruction in a multi-level cache hierarchy takes depends on the location of its data in the hierarchy. Therefore, **a more refined notion of the depth of a DAG** is needed to bound the space requirements of a PDF-like scheduler.

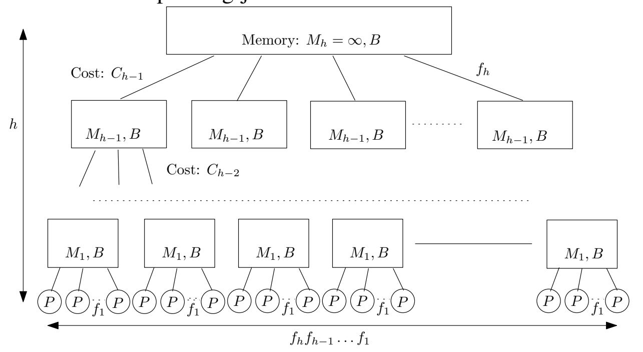
Our Results. In this paper, we present the first runtime thread scheduler achieving our goal of provably space-,



(a) A nested-parallel DAG. The circles represent the fork and join points. F is a fork, and J its corresponding join.



(b) One-level shared cache machine model.



(c) A Parallel Memory Hierarchy [17], modeled by an h-level tree of caches, with $\prod_{i=1}^{h} f_i$ processors.

Figure 3: Programming and Machine models.

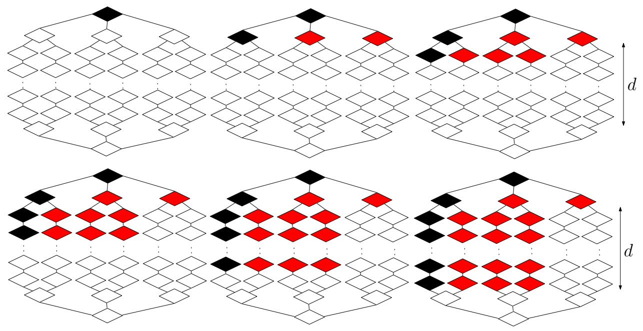


Figure 4: A four processor PDF schedule on a DAG of depth $d+4$ at different points in the execution. The maximum continuous segment of instructions in the depth-first schedule executed by the PDF schedule at each point are marked in black. Premature nodes executed out of the depth-first order are shown in red and are no more than $3d+2$ in number.

cache-, and time-efficient scheduling for both (i) multi-level parallel cache hierarchies and (ii) dynamic memory allocation. Our scheduler uses a novel combination of ideas from prior work on space-bounded schedulers and PDF schedulers. The idea of our new scheduler, which we call the *space-bounded recursive-PDF scheduler*, is when anchoring a task to a level i cache, we identify which of its subtasks just fit in a level $i-1$ cache (as distinguished from those subtasks that fit in levels $\leq i-2$) and then schedule those subtasks in a PDF order. This can be viewed as extending the *controlled-PDF scheduler* [12] to more than two levels (that paper [12], however, did not consider space bounds).

We present bounds on parallel space in terms of the "effective depth" of a program (see Sec. II-D for definition) that takes into consideration the non-uniformity of time required to access data on the critical path. The effective depth is adapted from a set of program-centric metrics called the parallel cache-complexity (PCC) framework [18] that also includes the "effective cache complexity". We show that our scheduler leads to good space and cache bounds, as well as good time bounds, for nested-parallel programs that obey a regularity criteria quantified in the effective cache complexity. The regularity criterion basically requires that when forking a set of parallel subtasks, the subtasks have approximately the same relationship between work and space. This allows for subtasks with vastly different input sizes, as long as the computation performed within each subtask has a fixed relationship between input size and space. This criterion is satisfied not only by Quicksort and Matrix Multiplication algorithms described here, but also many others including scans, basic graph and combinatorial algorithms (see Tables $6.1, 6.2$ of [18]). A key feature of our metrics and the regularity criterion is that they are program-centric (or multicore-oblivious [14]), i.e., defined solely in terms of the program without reference to scheduler or hardware parameters.

Contributions.

- A class of space-bounded schedulers with good bounds on space, cache misses, and time. Our bounds hold for nested parallel programs and a general allocation scheme that allows for allocations and frees at arbitrary points in the program. Allocations and frees may be implicit, e.g., in a garbage-collected language where the "free" occurs implicitly after the last reference.
- The first effective bounds on the extra space required at each level of the cache to schedule a program on a tree of caches, in terms of the program's "parallelizability" and effective cache complexity. When programs that are "regular" and highly parallel are mapped on to machines whose parallelism does not exceed the parallelizability of the program, the space bound as well as the cache miss and run time bounds are asymptotically optimal.

One class of programs for which our scheduler guarantees asymptotically optimal bounds is certain well-structured, highly parallel, divide-and-conquer routines. Call a divide-and-conquer routine a type-0 growth-$(\log_b a)$ routine if it consists of recursively solving a size-n/b subproblems in parallel, for integer constants $a, b > 1$, and additionally performs $O(\text{polylog(n)})$ sequential work. These routines have $O(n^{\log_b a})$ work and polylogarithmic depth. Examples of type-0 growth-1 (linear work) routines include reductions on arrays and Matrix Addition. More generally, a type-i growth-$\log_b a$ may additionally perform a constant number of size-n calls to lower-type divide-and-conquer subroutines with growth at most $(\log_b a)$. The 8-way Matrix Multiplication is an example of a type-1 routine, with growth $\log_4 8 = 1.5$, because it relies on the type-0 routine of Matrix Addition. Similarly, if Quicksort were always provided with the median element as pivot, it would be a growth-1 type-1 routine. For constant type, a routine with growth c has polylogarithmic depth and $O(n^c \text{polylog(n)})$ work. For realistic machine parameters, our scheduler guarantees asymptotically optimal space bounds for constant-type routines with growth at least 1.

Note that our results, like much prior work, assume a *dag consistent* memory model for the caches [19], in which concurrent subtasks competing for disjoint pieces of the same cache line can each operate on their distinct piece in parallel even if some of the subtasks are writing, and then the state of the cache line is resolved at the common synchronization point for the subtasks.[1]

Road Map. Section II describes the computational model, the allocation model, the machine model, and the parallel cache complexity framework used in this paper. Section III presents our space-bounded recursive-PDF scheduler. Section IV presents our main theorem on the performance of such a scheduler and Section V discusses the results.

II. Programming and Machine Models

We consider scheduling dynamically allocated nested-parallel programs on parallel cache hierarchies. Programs are represented by DAGs that unfold dynamically, i.e., parts of the graph need not be specified until their predecessors have been executed. The nodes in the DAG are either (a) arithmetic or control instructions such as add, jump if equal, (b) Fork or Join instructions that help express parallelism, and (c) **alloc** and **free** instructions that allocate and free memory respectively. We consider DAGs that are structurally deterministic. That is, multiple executions of the DAG, given the same input (and random seed, if required), must unfold to the same DAG with the same set of instructions (nodes) and precedence constraints (edges). The programming model, allocation model and machine model are discussed in Sections II-A, II-B, and II-C respectively. The metrics needed to quantify the cost of the DAGs are in Section II-D. All of the notation used in this Section is summarized in Table I.

A. Nested-Parallel DAGs: Tasks, Strands, Parallel Blocks

We consider nested-parallel programs allowing arbitrary dynamic nesting of fork-join constructs but no other synchronizations. Fork-join constructs may be used to implement parallel loops. This corresponds to the class of algorithms with series-parallel dependence graphs (see Fig. 3a).

[1]Such so-called *false sharing* of cache blocks would cause additional misses in today's multicores. If the cache line size B is known, then padding to cache line boundaries solves this problem. See [20] for a machine-oblivious approach (i.e., B is unknown).

Following the notation from [15], series-parallel DAGs can be decomposed into "tasks", "parallel blocks" and "strands" recursively as follows. As a base case, a ***strand*** is a serial sequence of instructions not containing any parallel constructs or subtasks. A ***task*** is formed by serially composing $k \geq 1$ strands interleaved with $(k-1)$ "parallel blocks" (denoted by $\mathsf{t} = \mathsf{l}_1; \mathsf{b}_1; \ldots; \mathsf{l}_k$). 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 (denoted by $\mathsf{b} = \mathsf{t}_1 \| \mathsf{t}_2 \| \ldots \| \mathsf{t}_k$). A parallel block can be, for example, a parallel loop or some constant number of recursive calls. We do allow a single task in a parallel block, and in the PCC framework (Section II-D), this is different from continuing with a strand. If a task occurs in this serial interleaving, it is implicitly understood to be wrapped inside a parallel block, allowing the above definition of a task as the serial interleaving of only strands and parallel blocks. We refer to the top-level task as the ***computation***.

All strands share a single memory. We say two strands are ***concurrent*** if they are not ordered in the dependence graph. We assume that the program is free from data races [21], [22]. Concurrent reads (i.e., concurrent strands reading the same memory location) are permitted, but not concurrent writes (i.e., concurrent strands that read or write the same location with at least one write).

B. Dynamic Allocation and Space Bounds

Programs are allowed to allocate and free variables on demand. The runtime system handles the mapping from variables to memory locations during program execution. Memory locations are provided on demand from a pool of free locations and recycled back when no longer needed. We assume the following natural restrictions on the dependencies between nodes that allocate a new variable v (**alloc(v)**), deallocate v (**free(v)**) and reference v:[2] (i) **free(v)** must succeed the corresponding **alloc(v)** in the DAG; (ii) v cannot be referenced by a node that precedes or is concurrent with **alloc(v)**; and (iii) v cannot be referenced by a node that succeeds or is concurrent with **free(v)**. Note that these restrictions do not force a variable to be allocated and freed at the same level of nesting. This is an important relaxation, because many programs violate the same-nesting-level requirement, e.g., the space-efficient Quicksort in Section I.

For programs such as the QuickSort in Fig. 1 where the allocation and the deallocation of variables are implicit, we can construct the equivalent DAG by placing **alloc** and **free** nodes appropriately. An **alloc** node is placed at the first instance where a variable is defined, and **free** is placed at the earliest position that follows (in DAG order) the allocation and all references.

[2]Our results generalize somewhat beyond these restrictions, e.g., allowing the last of a set of concurrent references to free a variable, as would be done in a garbage-collected setting.

n	problem size
D	span (a.k.a. depth) of computation
p	number of processing cores
S_1, S_p	space of sequential, and parallel executions
M, M_i	cache size (at level i)
B	memory block size (a.k.a. cache line size)
C_i	cost of cache miss at level i
f_i	num. of level $i-1$ caches per level i cache
c, t, b, l	computation, task, parallel block, strand
$Q_1, Q^*, \widehat{Q}_\alpha$	sequential, parallel and effective cache complexity
$S_1(\mathsf{t}; B)$	space of sequential exec. of t in words
$s_1(\mathsf{t}; B)$	space of sequential exec. of t in memory blocks
$loc(\mathsf{c})$	Set of memory blocks (locations) touched by c
$\left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}; x, B)}{s_1^\alpha(\mathsf{t}; B)} \right\rceil$	Effective depth of t for parameter α

Table I: Summary of Notation

We assume that the machines on which the programs are mapped have their memory and cache organized into *memory blocks* of a fixed size B. The **alloc** call can allocate only memory segments that are integral multiples of B. This is not a severe restriction as the sequential base case of the recursion that expresses parallelism is normally designed to use $\gg B$ words. Similar to prior work [10], we assume that an allocation or freeing of an array that has a length of k memory blocks is modeled in the DAG as k allocations of a single block. An allocation of $j > 1$ blocks is modeled as a parallel block of j tasks each of which allocates 1 memory block. This would not change the depth of the DAG significantly (at most a factor of 3), but it reflects the fact that allocating j memory blocks might need j units of *work*.

Space of a task. The number of active locations at an instant during execution—locations to which some live program variable is mapped—depends on the execution order of the instructions in the program. The ***space of a task*** in a program with respect to a schedule at a certain point t is the sum of the space of all previously allocated memory blocks referenced by the task until t plus the difference between allocations and deallocations made in the task until point t.

Space Requirement. The high water mark for the number of active locations during an execution represents the ***space requirement of the schedule*** for the computation or a task within the computation. We denote the space required by a task t (computation c) under the sequential left-to-right depth-first (1DF) schedule in terms of the number of memory blocks by $s_1(\mathsf{t}; B)$ ($s_1(\mathsf{c}; B)$). The number of words is denoted by $S_1(\mathsf{t}; B) := s_1(\mathsf{t}; B) \times B$. We denote the set of memory blocks touched by a task t by $loc(\mathsf{t}; B)$. For example, $S_1(\mathsf{c}_n; B) = O(n^2)$ for $n \times n$ matrix multiplication in Fig. 2, and $S_1(\mathsf{c}_n; B) = O(n)$ for Quicksort in Fig. 1.

C. Machine Model: Parallel Memory Hierarchy

Following prior work addressing multi-level parallel cache hierarchies, we model parallel machines using a tree-of-caches model. For concreteness, we will use, as in [15], a symmetric variant of the PMH model [17]. A PMH consists of a height-h tree of memory units, called caches (see Fig. 3c). We assume that each cache is an *ideal cache* [23], which has an optimal replacement policy. (See [11], [24], [25] for implications of realistic replacement policies.) The leaves of the tree are at level-0 and any internal node has a level one greater than its children. The leaves (level-0 nodes) are processors (cores), 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. Each level in the tree is parameterized by three parameters: M_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 memory blocks (cache lines) of size B. A level-$(i+1)$ cache miss occurs whenever a level-i cache miss occurs and the requested memory block is not resident in the parent level-$(i+1)$ cache; once the data becomes resident in the level-$(i+1)$ cache, a level-i cache request may be serviced by loading the memory block into the level-i cache. The cost of a level-i cache miss, denoted by $C_i \geq 1$, is the amount of time to load the corresponding memory block into the level-i cache under full load. Thus, C_i models both the latency and the bandwidth constraints. The number of level-$(i-1)$ caches below a level-i cache is denoted f_i, and typically $M_i > f_i M_{i-1}$.

D. The Parallel Cache Complexity Framework

This section develops on the parallel cache complexity (PCC) framework [18], and builds the definitions of the interrelated ***effective cache complexity*** and ***effective depth*** metrics, which are used for the analysis of the scheduler, via an intermediate **parallel cache complexity** metric. These "program-centric" metrics measure the anticipated costs of the program a PMH. The effective cache complexity quantifies the amount of time a program might spend accessing data from caches of a certain level in the PMH. It also captures the cost of load-balancing an irregular program. Just as the depth metric quantifies the limit of parallelizing the work in a DAG on the PRAM machine model, the effective depth quantifies the limit of parallelizability of the effective cache complexity on the PMH machine model. There are two keys issues that motivate the definition.

Parallel Cache Complexity. The first issue is that previous measures of cache complexity such as the one defined in [23] are for sequential programs. They quantify the number of cache misses a program would incur on an ideal cache [23] of a certain size. While this measure can be adapted to parallel programs as the "sequential cache complexity" Q_1 of the the 1DF schedule over the DAG, it may reward artificial sharing that occurs across parallel tasks. There are computations for which a sequential execution can reuse data and have low cache cost, whereas it is impossible to achieve this degree of reuse when tasks are scheduled in parallel [15, Th. 1]. To cope with this issue, the parallel cache complexity metric (i) ignores data reuse among parallel subtasks, and (ii) assumes an initially empty cache for any task larger than the cache size. For many algorithms, the existing analyses already ignore this artificial reuse, and hence the resulting bounds match the sequential cache complexity [18, Ch. 6].

Definition 1 (Parallel Cache Complexity)**.** *For cache parameters M and B the **parallel cache complexity** of a strand* l, *parallel block* b, *or task* t *starting at state κ is defined as:*
strand: $Q^*(\mathsf{l}; M, B; \kappa) = Q(\mathsf{l}; M, B; \kappa)$
parallel block: *For* $\mathsf{b} = \mathsf{t}_1 \| \mathsf{t}_2 \| \ldots \| \mathsf{t}_k$,
$Q^*(\mathsf{b}; M, B; \kappa) = \sum_{i=1}^{k} Q^*(\mathsf{t}_i; M, B; \kappa)$
task: *For* $\mathsf{t} = \mathsf{c}_1; \mathsf{c}_2; \ldots; \mathsf{c}_k$,
$Q^*(\mathsf{t}; M, B; \kappa) = \sum_{i=1}^{k} Q^*(\mathsf{c}_i; M, B; \kappa_{i-1})$,
where $\kappa_i = \emptyset$ if $S_1(\mathsf{t}; B) > M$,
and $\kappa_i = \kappa \cup_{j=1}^{i} loc(\mathsf{c}_j; B)$ if $S_1(\mathsf{t}; B) \leq M$.

Formally, the parallel cache complexity $Q^*(\mathsf{c}; M, B)$ is recursively defined for a computation c based on the composition rules in Sec. II-A. The present definition differs from the original definition of Q^* [15]—which assumed static allocation—by replacing all space terms by S_1. Here, the parameter κ is used to denote the starting state of a cache for each subcomputation, or $\kappa = \emptyset$ implicitly if the parameter is omitted. We denote the work (flops and other instructions) in c by $Q^*(\mathsf{c}; 0, 1)$. Let $Q(\mathsf{l}; M, B; \kappa)$ denote the sequential cache complexity of a strand l in the ideal cache model [23] when starting with cache state κ.

Balance Operation. The second issue is that for any cache in a PMH, the size of the cache and the number of processing units clustered below that cache are correlated. It therefore stands to reason that tasks can only be scheduled efficiently if they have a good ratio of parallelism to space requirements. In fact, we previously proved that for certain sufficiently parallel computations, it is impossible to schedule them without either sacrificing parallelism or consuming more space or suffering more cache misses [15, Theorem 4].

To cope with this second issue, we extend parallel cache complexity Q^* to the effective cache complexity to capture space-parallelism imbalance of the algorithm through a balance operation. The effective cache complexity of a computation c is denoted by $\widehat{Q}_\alpha(\mathsf{c}; M, B)$, where M is the cache size and B is the block size, and $\alpha \geq 0$ is an analytic parameter that models parallelism. The balance operation relates the amount of parallelism that can be utilized to the memory footprint of the computation—the operation models the effect of running a size-S computation on $O((S/B)^\alpha)$ processors, where $\alpha \geq 0$. We adopt the convention that the space of strands nested directly in task t is equal to $S_1(\mathsf{t}; B)$. This is purely for convenience and does not restrict the program in any way.

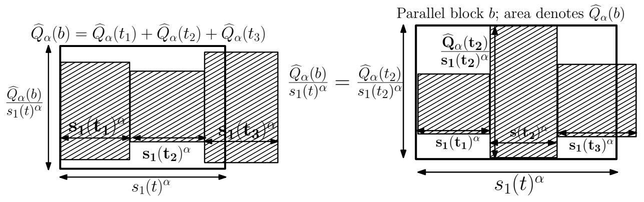

$\widehat{Q}_\alpha(b) = \widehat{Q}_\alpha(t_1) + \widehat{Q}_\alpha(t_2) + \widehat{Q}_\alpha(t_3)$
$\frac{\widehat{Q}_\alpha(b)}{s_1(t)^\alpha}$
$\mathbf{s_1(t_1)^\alpha}$
$\mathbf{s_1(t_2)^\alpha}$
$\mathbf{s_1(t_3)^\alpha}$
$s_1(t)^\alpha$
Parallel block b; area denotes $\widehat{Q}_\alpha(b)$
$\frac{\widehat{Q}_\alpha(b)}{s_1(t)^\alpha} = \frac{\widehat{Q}_\alpha(t_2)}{s_1(t_2)^\alpha}$
$\frac{\widehat{\mathbf{Q}}_\alpha(\mathbf{t_2})}{\mathbf{s_1(t_2)^\alpha}}$
$\mathbf{s(t_2)^\alpha}$
$\mathbf{s_1(t_1)^\alpha}$
$\mathbf{s_1(t_3)^\alpha}$
$s_1(t)^\alpha$

Figure 5: Work-dominated (left) and depth-dominated (right) parallel blocks. Height represents effective depth and the area represents effective cache complexity.

Definition 2 (Effective cache complexity)**.** *For cache parameters M and B, and $\alpha > 0$, the **effective cache complexity** of a strand l, parallel block b, or a task t is defined as.*

strand: $\widehat{Q}_\alpha(\mathsf{l}; M, B) = Q^*(\mathsf{l}; M, B) \times s_1^\alpha(\mathsf{l}; B)$

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

$$\frac{\widehat{Q}_\alpha(\mathsf{b}; M, B)}{s_1^\alpha(\mathsf{t}; B)} = \max \begin{cases} \max_i \left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}_i; M, B)}{s_1^\alpha(\mathsf{t}_i; B)} \right\rceil & \textit{(depth dominated)} \\ \frac{\sum_i \widehat{Q}_\alpha(\mathsf{c}_i; M, B)}{s_1^\alpha(\mathsf{t}; B)} & \textit{(work dominated)} \end{cases}$$

task: *For* $\mathsf{t} = c_1; c_2; \ldots; c_k$,
$\widehat{Q}_\alpha(\mathsf{t}; M, B) = \sum_{i=1}^k \widehat{Q}_\alpha(\mathsf{c}_i; M, B)$.

The multiplication by $s_1^\alpha(\mathsf{l}; B)$ for the strand reflects the fact that while one processor executes the sequential strand, all the remaining $s_1^\alpha(\mathsf{l}; B) - 1$ processors wait for it to finish. The ratio $\left\lceil \widehat{Q}_\alpha(\mathsf{t}_i) / s_1^\alpha(\mathsf{t}_i; B) \right\rceil$ defines the ***effective depth*** for the subtask t_i for a parameter α. This effective depth metric functions as a proxy for the depth of the algorithm. When the *depth-dominated* term applies, the effective cache complexity of the parallel block b nested directly inside task t is the maximum effective depth of its subtasks multiplied by the number of processors for the parallel block b, which is $s_1^\alpha(\mathsf{t}; B)$ (see Fig. 5). When α increases, the depth-dominated term captures the effect of using $s_1^\alpha(\mathsf{t}_i; B)$ processors to execute t_i. Setting $\alpha = 0$ is analogous to computing the effective cache complexity on 1 processor, and hence the work term always dominates. We say that an algorithm is ***α-efficient*** if $\widehat{Q}_\alpha(\mathsf{c}; M, B) = O(\widehat{Q}_0(\mathsf{c}; M, B))$, where $\widehat{Q}_0$ (i.e., $\widehat{Q}_\alpha$ with $\alpha := 0$) is Q^* by definition. This α-efficiency occurs trivially if the work term always dominates, but can also happen if sometimes the depth term dominates.

Parallelizability. The least upper bound on the set of α for which an algorithm is α-efficient (as $n \to \infty$) specifies the ***parallelizability of the algorithm***. For example, the Quicksort algorithm in Fig. 1 is α-efficient for all $\alpha \in [0, 1)$ and not for any value of $\alpha \geq 1$. The 8-fold recursive matrix multiplication algorithm has parallelizability of 1.5 and the limited parallelism version in Fig. 2 has a parallelizability of only 1. Analogous to the parallelizability of an algorithm, the ***parallelism of the machine*** is defined to be the least value of β for which $f_i \leq \left(\frac{M_i}{M_{i-1}}\right)^\beta$ for all $i > 1$, $f_1 \leq (M_1/3B)^\beta$.

III. The space-bounded recursive-PDF scheduler

Chowdhury et. al. introduced the idea of *space-bounded schedulers* for use in trees of caches [13], [14]. The schedulers assume the memory size of a task is known when the task is scheduled and therefore dynamic memory allocation significantly complicates the story. The problem is that memory allocation and cache bounds interact. In particular, to bound the number of cache misses at a particular cache it is helpful to bound the memory footprint of the task that is scheduled on that cache—hence the name "space-bounded". However the footprint depends on the amount of data that are dynamically allocated and simultaneously live during the task, which in turn depends on how subtasks are scheduled within the task. To avoid this problem, previous work [14], [15] has assumed that all memory is preallocated by the user (*static* allocation). Such an assumption limits the kind of programs that can be implemented in the framework, and even for programs that can be implemented, it is a significant inconvenience for the programmer.

In [15], we designed a class of "space-bounded" schedulers [14] parameterized by a global ***dilation parameter*** $0 < \sigma \leq 1$, machine parameters $\{M_i, B, f_i\}$ to map nested-parallel programs to PMH. 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., $S_1(\mathsf{t}; B) \leq \sigma M_i$ and $S_1(\mathsf{t}; B) > \sigma M_{i-1}$. 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 schedule satisfies two properties:

- *Anchored [14]*: Each task is anchored to a smallest possible cache that is bigger than the task—strands within the task can only be scheduled on processors in the tree rooted at the cache.
- *Bounded*: A maximal live task "occupies" a cache X if it is either (i) anchored to X, or (ii) anchored to a cache in a subcluster below X while its parent is anchored above X. A live strand occupies cache X if it is live on a processor beneath cache X and the strand's task is anchored to an ancestor cache of X. The sum of sizes of live tasks and strands that occupy a cache is restricted by the scheduler to be less than the size of the cache.

These two conditions are sufficient to imply good bounds on the number of cache misses.

Theorem 3 (Theorem 3 [15])**.** *Let* t *be a level-i task. The number of level-j cache misses incurred by executing* t *with any space-bounded scheduler is at most* $Q^*(\mathsf{t}; \sigma M_j, B_j)$ *for all cache levels* $j \leq i$.

The scheduler operates by anchoring the root task to the memory, and exploring the task till it finds a collection of subtasks and strands that fit in the next highest cache level $(h-1)$. It then anchors and executes these subtasks on the $(h-1)$ level cache in keeping with the space and processing constraints. The subtasks anchored at level $(h-1)$ are recursively explored and so on. Since processors are tied to caches, a space-bounded scheduler would need to carefully allocate caches and processors to tasks based on their size. This is done by keeping track of processor utilization and through a "desire" function $g_i(S)$ that specifies for each level i task t_i the number of $(i-1)$-level caches (***subclusters***) allocated to t_i as a function of its space $S_1(\mathsf{t}_i; B)$. At least $(1/(1-k))^{i-1}$ fraction of processors, where k is constant in $(0,1)$, on each of the $g_i(S)$ level-$(i-1)$ caches assigned to t_i are required to be dedicated to t_i. (see [15, Sec.7] for more details including the definition of the utilization function μ.) The choice of the desire function is critical to obtaining good running time bounds for the scheduler. Using the desire function $g_i(S) = \min\{f_i, \max\{1, \lfloor f_i(3S/M_i)^{\alpha'} \rfloor\}\}$ where $\alpha' \geq 0$ is a parameter smaller than both 1 and α the parallelizability of the algorithm, we showed the following bound on runtime.

Theorem 4 (Reworded lemma 11 from [15])**.** *Consider an h-level PMH with parallelism β and a computation to schedule with parallelizability α such that $\alpha > \beta$. Let α' be a non-negative parameter such that $\beta < \alpha' \leq \min\{\alpha, 1\}$. Let N_i be a task or strand which has been assigned a set $\mathcal{U}_t$ of $q \leq g_i(S(N_i; B))$ level-$(i-1)$ subclusters by the scheduler. Letting $\sum_{V \in \mathcal{U}_t}(1-\mu(V)) = r$ (by definition, $r \leq |\mathcal{U}_t| = q$), the running time of N_i is at most $\frac{\sum_{j=0}^{h-1} C_j \cdot \widehat{Q}_\alpha(N_i; M_j, B)}{r p_{i-1}} \cdot v_i$, where $v_i = 2\prod_{j=1}^{i-1}\left(\frac{1}{k} + \frac{f_j}{(1-k)(M_j/M_{j-1})^{\alpha'}}\right)$ is overhead.*

Key ideas in this paper. Any space-bounded schedule that follows the anchoring and processor allocation rules specified in Section 7 of [15] has good bounds on time, irrespective of the order in which subtasks of a maximal level-i task are scheduled at level-$(i-1)$ subclusters. But to achieve good bounds on space, subtasks must be scheduled in an order that is close to the depth-first order. Therefore, we use an adaptation of the Parallel Depth-First (PDF) scheduler to schedule subtasks with in a task. A p-processor PDF schedule $\mathcal{S}$ based on a depth-first order $\mathcal{S}_1$ is defined as follows. Label each node in the DAG with its order of execution in $\mathcal{S}_1$ and prioritize nodes by this order. Construct $\mathcal{S}$ by allowing any idle processor to pick the ready node (a ready node has all its predecessors in DAG completed) in the DAG with the highest priority according to the order of execution in $\mathcal{S}_1$.

We organize each maximal level-i task t_i into a collection of maximal tasks of any level less than i, which we refer to as *super-nodes*, and nodes which do not belong to any maximal subtask, which we refer to as *glue-nodes*. When t_i is anchored to a level-i cluster, the glue and super nodes in t_i are assigned to the subclusters in the PDF order based on the left-to-right sequential depth-first order, according to same allocation policy and desire function as earlier based on $S_1(\mathsf{t}; B)$ (instead of $S(\mathsf{t}; B)$ as in [15]). A subcluster that starts a super node completes it before seeking more work in the PDF order. Each super-node anchored to a subcluster is in turn recursively scheduled using the PDF order on its glue and super-nodes. In the next section, we will prove that the parallel execution that follows this recursive PDF order subject to space-bounded constraints has low space requirements for highly parallel programs.

IV. Bounds on space

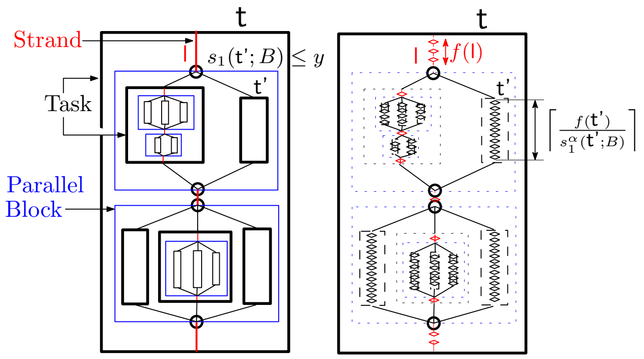
Our goal is to bound the space used by a space-bounded recursive-PDF scheduler. The observation is that, relative to the sequential execution, a parallel execution might have larger quantities of data that are "live" at the same time, and hence require more memory. Thus, as in prior work for other schedulers, we seek to bound the space for parallel execution relative to the space for sequential execution.

In the space-bounded recursive-PDF schedule on a PMH based on the depth-first schedule $\mathcal{S}_1$, additional space is required by super-nodes and glue-nodes at each level in the hierarchy that are ***premature*** with respect to $\mathcal{S}_1$, i.e., nodes that are executed before their turn in the sequential $\mathcal{S}_1$ has arrived. Our main result is an upper bound on the size of premature nodes at each level in the cache hierarchy.

Theorem 5. *Let t_i be a level-i task. Let $\alpha > 0$. Let $\mathcal{S}$ be a space-bounded recursive-PDF schedule that, based on the sequential depth first schedule $\mathcal{S}_1$, maps t_i to a PMH. The combined sequential space of all level-$(i-1)$ supernodes and glue nodes that are premature in $\mathcal{S}$ with respect to $\mathcal{S}_1$ at any point does not exceed $O(d_\alpha \times f_i M_{i-1})$, where*

$$d_\alpha := \sum_{j=0}^{i-1} \frac{C_j}{C_{i-1}} \left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}; M_j, B)}{s_1(\mathsf{t}, B)^\alpha} \right\rceil.$$

The space requirement of the parallel schedule is bounded by the sum of the space requirement of $\mathcal{S}_1$ and the sum of maximum size of premature nodes at each level in the hierarchy. One might note that this bound is similar to the bound on the parallel space of the PDF schedule on a PRAM schedule; $S_p - S_1 \leq (p-1)d$, where p is the number of processors, d is the depth of the DAG, and $(p-1)d$ is the maximum number of premature nodes. The term d_α, which is analogous to d, is the sum of the effective depths of the algorithm with respect to each level in the hierarchy weighted by the cost of a cache miss at the corresponding level. The fan-out f_i is analogous with the number of processors p. The M_{i-1} represents the maximum possible sequential space of level-$(< i)$ supernode.



t
Strand
l
s_1(t'; B) $\leq y$
Task
t'
Parallel Block
t
l
f(l)
t'
$\left\lceil \frac{f(\mathsf{t}')}{s_1^\alpha(\mathsf{t}';B)} \right\rceil$

Figure 6: Shortened DAG of task t with respect to function f at threshold $y \leq S_1(\mathsf{t}; B)$. The diamonds represent shortened nodes. Each task on the left is a maximal subtask of t with size $< y$.

We now build up some definitions and lemmas to prove this result. The first among these allows us to simplify a mixture of supernodes and glue nodes into a more uniform DAG that is simpler to analyze (see Fig. 6).

Definition 6 (α-Shortened DAG). *Let f be a function that assigns to each task and strand a non-negative real number. Let $\alpha \geq 0$. The **α-shortened DAG** $G^f_{\alpha,y}(\mathsf{t})$ of task t evaluated with respect to f at threshold $y \leq S_1(\mathsf{t}, B)$ is constructed based on the decomposition of the DAG G corresponding to t into super-nodes of size at most y and glue-nodes. Replace the super-node corresponding to every maximal subtask t′ of sequential size $s_1(\mathsf{t}'; B) \leq y$ by a chain of $\lceil f(\mathsf{t}')/s_1^\alpha(\mathsf{t}'; B)\rceil$ nodes in $G^x_{\alpha,y}$. For every maximal sequential chain of glue-nodes l, add $f(\mathsf{l})$ nodes in $G^f_{\alpha,y}(\mathsf{t})$. Precedence constraints between the nodes in $G^f_{\alpha,y}(\mathsf{t})$ are identical to those between the super-nodes and glue-nodes they represent in G (Fig. 6).*

Note that α is a free parameter in this definition just as in the definition of the effective cache complexity. The α-shortened DAGs for different values of f and y are just convenient notions for proving our claims and are not used by or known to the space-bounded recursive-PDF scheduler. The α-shortened DAG is constructed deliberately so as to be related to effective depth.

Lemma 7. *Let $f(\mathsf{t}) := \widehat{Q}_\alpha(\mathsf{t}; x, B)$ be the effective cache complexity for some cache size $x > 0$. Fix $\alpha > 0$ and $y \leq S_1(\mathsf{t}, B)$. The depth of the α-shortened DAG $G^f_{\alpha,y}(\mathsf{t})$ of a task t is at most $\left\lceil \widehat{Q}_\alpha(\mathsf{t}; x, B)/s_1^\alpha(\mathsf{t}; B)\right\rceil$.*

Proof: This follows from the composition rules for effective cache complexity. We will fix y, and prove the claim by induction on the composition rules of the task t.

If $s_1^\alpha(\mathsf{t}; B) = y/B$, the claim follows immediately. For a task $\mathsf{t} = \mathsf{l}_1; \mathsf{b}_1; \mathsf{l}_2; \mathsf{b}_2; \ldots; \mathsf{l}_k$ with sequential space greater than y, the depth of the α-shortened graph corresponding to t is the sum of the depths of the α-shortened subgraphs corresponding to $\mathsf{l}_1, \mathsf{t}_1; \mathsf{l}_2; \mathsf{t}_2; \ldots; \mathsf{t}_k$ where t_i is the parallel task in block b_i with the largest effective depth.

If we inductively assume that the depth of the α-shortened DAG of each t_i with $s_1(\mathsf{t}_i; B) \geq y/B$ is $\leq \left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}_i; x, B)}{s_1(\mathsf{t}_i, B)^\alpha} \right\rceil$, then the lemma follows by induction because the composition rules for effective depth imply $\left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}; x, B; \kappa)}{s_1^\alpha(\mathsf{t}; B)} \right\rceil \geq \sum_{i=1}^k Q^*(\mathsf{l}_i; x, B) + \sum_{i=i}^{k-1} \left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}_i; x, B; \kappa)}{s_1^\alpha(\mathsf{t}_i; B)} \right\rceil$. ■

Since the total work done by a scheduler is a mixture of cache misses at different levels, we would like to bound the depth of an α-shortened DAG with respect to a function that captures work across all levels of hierarchy. For this, we borrow the definition of ***latency added effective work*** from [15]. We assume that allocations in a maximal level-$(i-1)$ task incur the cost of cache miss to be fetched from level i cache, i.e., they incur C_{i-1} cost. Let the latency cost $\rho(x)$ of an instruction x accessing location m be $\rho(x) = Q^*(x; 0, 1) + \sum_{k=1}^{j-1} C_k$ if the scheduler causes the instruction x to fetch m from a level j cache on the given PMH. Using this per-instruction cost, we can define the latency added cost of a task, and apply the balance operation to define the effective work $\widehat{W}^*_\alpha(\cdot)$, just as the balance operation was applied to parallel cache complexity to define effective cache complexity. Let $N(\mathsf{t}) = \widehat{W}^*_\alpha(\mathsf{t})/C_{i-1}$ be the effective work normalized by C_{i-1} for a level-i task t.

Lemma 8. *Fix $\alpha > 0$ and let $y \leq S_1(\mathsf{t}, B)$. The depth of the α-shortened DAG $G^N_{\alpha,y}(\mathsf{t}) \leq \sum_{j=0}^{i-1} \frac{C_j}{C_{i-1}} \left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}; M_j, B)}{s_1^\alpha(\mathsf{t}; B)} \right\rceil$.*

Proof: The separation lemma from [15] allows effective work of any level-i task t′ to bounded by a summation of effective cache complexities: $\widehat{W}^*_\alpha(\mathsf{t}') \leq \sum_{j=0}^{i-1} C_j \widehat{Q}_\alpha(\mathsf{t}'; M_j, B)$. Using this inequality, lemma 7 and rearranging the ceiling functions, we have for any level-i task $\left\lceil \frac{N(\mathsf{t}')}{s_1^\alpha(\mathsf{t}'; B)} \right\rceil \leq \left\lceil \sum_{j=0}^{i-1} \frac{C_j}{C_{i-1}} \frac{\widehat{Q}_\alpha(\mathsf{t}'; M_j, B)}{s_1^\alpha(\mathsf{t}'; B)} \right\rceil \leq \sum_{j=0}^{i-1} \frac{C_j}{C_{i-1}} \left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}'; M_j, B)}{s_1^\alpha(\mathsf{t}'; B)} \right\rceil$. Just as in the case of previous lemma, this lemma follows by induction along the lines of the composition rules of the effective cache complexity applied to x in $\widehat{Q}_\alpha(\mathsf{t}; x, B)$ for each value in $\{M_j\}_{j=0}^{i-1}$. ■

Theorem 5 follows directly from lemma 9 which bounds the size of premature nodes in terms of depth of α-shortened DAG and lemma 8 which bounds the depth of α-shortened DAG in terms of effective depth.

Lemma 9. *Let t be a level-i task. Let $\alpha > 0$. Let $\mathcal{S}$ be a space-bounded recursive-PDF schedule that, based on the sequential depth first schedule $\mathcal{S}_1$, maps t to a PMH.*

Denote the depth of the α-shortened DAG $G^N_{\alpha,M_{i-1}}(\mathsf{t}_i)$ by d_α. The combined sequential space of all level-$(i-1)$ supernodes and glue nodes that are premature in $\mathcal{S}$ with respect to $\mathcal{S}_1$ at any point does not exceed $O\left(d_\alpha \times f_i M_{i-1}\right)$.

Proof: Let $H_\alpha := G^N_{\alpha,M_{i-1}}(\mathsf{t})$. Let $\alpha' = \min\{1, \alpha\}$.

We adapt the arguments in the proof of [10, Theoremm 2.3], which bounds the number of premature nodes in a PDF schedule, to analyze H_α. For this, we use the following convention: for a maximal task t' replaced by $l > 1$ nodes in H_α, we say that t' has been completed to depth $k < l$ in H_α in a space-bounded schedule if at least $k \times C_{i-1} \times s_1^{\alpha'}(\mathsf{t}'; B)$ effective work has been done on instructions in t'. Task t' is completed to depth l if it has been completed.

Partition wall-clock time into continuous phases of $\tau_i := \left\lceil v_i \times C_{i-1} \times \frac{(M_{i-1}/B)^\alpha}{p_{i-1}} \right\rceil$ clock cycles, where v_i is the overhead mentioned in Theorem 4. This provides sufficient time for the scheduler to complete one more depth level on any level-j $(j < i)$ subtask t_j.

Therefore, if a super-node is completed to depth i at the beginning of a phase and yet to be completed, it will be completed to depth $i+1$ by the end of the phase because of the allocation policy and theorem 4. Put another way, each phase provides ample time for a $(i-1)$-level subcluster to completely execute all the instructions in any set of nodes from H_α corresponding to maximal tasks or glue nodes that fit in M_{i-1} space.

Consider a snapshot of the execution and let $\mathcal{C}$ denote the set of nodes of H_α that have been executed and let $\mathcal{C}_1$ denote the longest sequential prefix of the sequential execution of t contained in $\mathcal{C}$. A phase is said to complete level l in the H_α if it is the earliest phase in which all nodes at depth l in $\mathcal{C}_1$ have been completed. We will bound the number of phases that complete a level $l \le d_\alpha$ by arguing that if a *new* premature node is started in phase i, either phase i or $i+1$ completes a level.

Suppose that a node of H_α premature with respect to $\mathcal{C}_1$ was started in phase i. Let l be the lowest level in H_α completed in $\mathcal{C}_1$ at the start of phase i. Then, at the beginning of phase i, all nodes in $\mathcal{C}_1$ at level $l+1$ are either executed, under execution or ready to be executed (denote these three sets by $\mathcal{C}_{l+1,i,d}$, $\mathcal{C}_{l+1,i,e}$, and $\mathcal{C}_{l+1,i,r}$ respectively). A premature node with respect to $\mathcal{C}_1$ can not be started unless all of the nodes in $\mathcal{C}_{l+1,i,r}$ have been started as they are ready for execution and have a higher priority in the PDF order on H_α. Therefore, if a premature node is started in phase i, all nodes in $\mathcal{C}_{l+1,i,r}$ have been started by the end of phase i, which implies they will be completed by phase $i+1$. Nodes in $\mathcal{C}_{l+1,i,e}$ will be completed by phase i. This proves our claim that if a premature node is started in phase i, a new level of H_α in phase i or $i+1$.

There are at most d_α nodes in which a level of $\mathcal{C}_1$ in H_α is completed. Since a premature node can be executed only in a phase that completes a level or in the phase before it, the number of phases that start a new premature node with respect to $\mathcal{C}_1$ is at most $2d_\alpha$. We will bound the additional space that premature nodes take up in each such phase. Note that there will be phases in which premature nodes are executed but not started. We account for the space added by each such premature node in the phase that started it.

Suppose a phase contained new premature nodes. A premature super-node in the decomposition of t that increases the space requirement by M units at some point during its execution must have at least M cache misses or allocations. It costs at least $M \times C_{i-1}$ processor cycles as every unit of extra space is paid for with C_{i-1} processor cycles. Therefore, the worst case scenario in terms of extra space added by premature nodes is the following. Every processor allocates an unit of space in a premature node every cycle until the last cycle. In the last cycle of the phase, an additional set of premature nodes of the largest possible size are started. The contribution of all but the last cycle of the phase to extra space is at most the number of cycles per phase multiplied by the number of processors and block size and divided by C_{i-1}, i.e., $\tau_{i-1} \times \frac{1}{C_{i-1}} \times p_i B = \left\lceil v_i \times f_i B \left(M_{i-1}/B\right)^{\alpha'} \right\rceil$. In the last cycle of phase, the costliest way to schedule premature nodes is to schedule a M_{i-1} size super-node at each level-$(i-1)$ subcluster for a total of $f_i M_{i-1}$ space. Adding together the extra space contributed by premature nodes across all phases gives an upper bound of $2d_\alpha \times \left(\left\lceil v_i \times f_i B (M_{i-1}/B)^{\alpha'} \right\rceil + f_i M_{i-1}\right) = O\left(d_\alpha \times f_i M_{i-1}\right)$. ■

V. Interpretation and Connections

To interpret Theorem 5, fix α to be any constant, preferably the largest for which the computation being mapped is α-efficient according to definition of effective cache complexity. Consider a highly parallel and regular algorithm like the 8-way recursive matrix multiplication algorithm with $\widehat{Q}_\alpha(\mathsf{t}_n; M, B) = O(\lceil n/M \rceil^{1.5} \lceil M/B \rceil)$ for all $\alpha < 3/2 - 1/2p$, $p = \log_{\lceil M/B \rceil} \lceil n/B \rceil$. The work exponent 1.5 closely matches the parallelizability $1.5 - 1/2p$ [18]. For such algorithms, the effective depth is poly-logarithmic or sub-polynomial ($o(n^c)$ for all $c > 0$) in input size. The extra space needed by the premature nodes in a level-i task t_i of size $S_1(\mathsf{t}_i, B) = M_i$ can be obtained by setting α to $1.5 - 0.5(\log_{\lceil M_i/B \rceil} \lceil M_{i-1}/B \rceil) - \epsilon$ in $O\left(\sum_{j=1}^{i-1} \frac{C_j}{C_{i-1}} \left\lceil \frac{\widehat{Q}_\alpha(\mathsf{t}_i; M_j, B)}{M_i^\alpha} \right\rceil \times f_i M_{i-1}\right)$, which results in $O\left(\left(\sum_{j=1}^{i-1} \frac{C_j}{C_{i-1}} \left\lceil \frac{M_{i-1}}{M_j} \right\rceil^{0.5} \times M_{i-1}^\epsilon\right) f_i M_{i-1}\right)$. Under the reasonable assumption that $\frac{C_{i-1}}{C_j} \ge \left\lceil \frac{M_{i-1}}{M_j} \right\rceil^{0.5}$ (cache access latency grows with physical dimension of cache), the last expression reduces to $O\left((i-1) \times M_{i-1}^\epsilon \times f_i M_{i-1}\right)$.

For realistic cache hierarchies, the machine parallelism $\beta < 1$, i.e., $f_i M_{i-1} = O(M_i^c)$ for some $c < 1$. Therefore, as $\epsilon \to 0$, the entire term for space of premature nodes is sublinear in M_i. In addition to algorithms with poly-logarithmic effective depth, these sort of bounds can also be shown for algorithms whose DAGs can be constructed as a sequence of tasks with poly-logarithmic effective depth, e.g., the recursive matrix inversion algorithm [18, Sec. 4.3.4].

Cache Bounds. To obtain good cache bounds, not just good space bounds, we will assume that the memory allocator maintains a pool of free memory for each cache to service the allocation requests by tasks anchored to that cache. If the task including its premature nodes fits in the cache, there will always be free memory available in that cache's pool. This avoids any additional cache misses when the same space is reused by distinct program variables.

One way to fit the parallel execution of a task in a cache is to augment each cache with the space required for premature nodes as indicated by Theorem 5. We can retain the same asymptotically optimal bounds on communication cost and time as in Theorems 3 and 4. When the extra space required at each level-i cache for a highly parallel and regular algorithm is sub-linear in M_i, the cache augmentation necessary is very small. Alternatively, since cache space available at each level is fixed, the space-bounded scheduler would need to anchor tasks to cache leaving enough margin for the extra space required to support parallelism. Suppose that partitioning level-i caches into two components of size M_i' and $M_i - M_i'$ for the in-order and premature nodes suffices. Then Theorems 3 and 4 would hold with M_i' substituted for M_i. For highly parallel and regular algorithms, even when the scheduler conservatively anchors tasks to caches to accommodate for the (sublinear) extra space, replacing M_i' for M_i does not asymptotically change the communication cost and runtime bounds.

Total Space. From Theorem 5, it follows that the sum of sequential space of premature nodes at levels in the hierarchy is sublinear in S_1 for highly parallel and regular algorithms when machine parallelism $\beta < 1$. The following lemma follows from the earlier observation about sublinear premature space and the fact that $f_i M_{i-1} = o(M_i)$ for all levels $i < h$ in the hierarchy.

Lemma 10. *Consider a level-h task* t *with parallelizability* α *whose effective depth with respect to* α *is always subpolynomial in n: $o(n^c)$ for all $c > 0$. Suppose that a space-bounded recursive-PDF scheduler maps this to a h-level PMH with parallelism $\beta < \min\{1, \alpha\}$. The extra space required at all levels of caches for allocations by premature nodes in addition to $S_1(\mathsf{t}; B)$ is sublinear in terms of $S_1(\mathsf{t}; B)$.*

References

[1] C. E. Leiserson, "The Cilk++ concurrency platform," *J. Supercomputing*, vol. 51, 2010.

[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," ser. OOPSLA '05.

[3] G. E. Blelloch, "NESL: A nested data-parallel language," CMU, Tech. Rep. CMU-CS-92-103, 1992.

[4] D. Lea, "A java fork/join framework," in *ACM Java Grande*, 2000.

[5] S. Chatterjee, S. Tasrlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with MPI," in *IPDPS'13*.

[6] Scandal, "Irregular algorithms in NESL language," 1994.

[7] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *JACM*, vol. 46, no. 5, 1999.

[8] D. J. Simpson and F. W. Burton, "Space efficient execution of deterministic parallel programs," *IEEE Trans. Softw. Eng.*, vol. 25, no. 6, 1999.

[9] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," *Theory Comp. Sys.*, vol. 35, no. 3, 2002.

[10] G. E. Blelloch, P. B. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," *JACM*, vol. 46, no. 2, 1999.

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

[12] 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.

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

[14] R. A. Chowdhury, V. Ramachandran, F. Silvestri, and B. Blakeley, "Oblivious algorithms for multicores and networks of processors," *JPDC*, vol. 73, no. 7, 2013.

[15] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri, "Scheduling irregular parallel computations on hierarchical caches," in *SPAA*, 2011.

[16] H. V. Simhadri, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and A. Kyrola, "Experimental analysis of space-bounded schedulers," in *SPAA*, 2014.

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

[18] H. V. Simhadri, "Program-centric cost models for locality and parallelism," Ph.D. dissertation, CMU, 2013.

[19] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, "An analysis of dag-consistent distributed shared-memory algorithms," in *SPAA*, 1996.

[20] R. Cole and V. Ramachandran, "Analysis of randomized work stealing with false sharing," in *IPDPS*, 2013.

[21] S. V. Adve and M. D. Hill, "Weak ordering – a new definition," in *ISCA*, 1990.

[22] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *ISCA*, 1990.

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

[24] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *CACM*, vol. 28, no. 2, 1985.

[25] A. K. Katti and V. Ramachandran, "Competitive cache replacement strategies for shared cache environments," in *IPDPS*, 2012.