Report Number: WUCSE-2011-64

2011

Hierarchical Scheduling for Multicores with Multilevel Cache Hierarchies

Kunal Agrawal and Jim Sukha

Cache-locality is an important consideration for the performance in multicore systems. In modern and future multicore systems with multilevel cache hierarchies, caches may be arranged in a tree of caches, where a level k cache is shared between Pk processors, called a processor group, and Pk increases with k. In order to get good performance, as much as possible, subcomputations that share more data should execute on processors which share a lower-level cache. Therefore, the number of cache misses in these systems depends on the scheduling decisions, and a scheduler is responsible for not just achieving good load-balance and... Read complete abstract on page 2.

Hierarchical Scheduling for Multicores with Multilevel Cache Hierarchies

Kunal Agrawal and Jim Sukha

Complete Abstract:

Cache-locality is an important consideration for the performance in multicore systems. In modern and future multicore systems with multilevel cache hierarchies, caches may be arranged in a tree of caches, where a level k cache is shared between Pk processors, called a processor group, and Pk increases with k. In order to get good performance, as much as possible, subcomputations that share more data should execute on processors which share a lower-level cache. Therefore, the number of cache misses in these systems depends on the scheduling decisions, and a scheduler is responsible for not just achieving good load-balance and low overheads, but also good cache complexity. However, these can be competing criteria. In this paper, we explore the tension between these criteria for online hierarchical schedulers. Formally, we consider a system with P processors, arranged in a multilevel hierarchy according to a hierarchy tree, where each of the P processors forms a leaf of the tree, and an internal node at level-k corresponds corresponds to a processor group. In addition, we assume that computations have locality regions, that represent parallel subcomputations that share data. Each locality region has a particular level, and the scheduler must ensure that a level-k locality region is executed by processors in the same level-k processor group, since they share a level k cache. Thus locality regions can improve cache performance. However, they may also impair load-balance and increase scheduling overheads since the scheduler must obey the restrictions posed by locality regions. In this paper, we present a framework of hierarchical computations, that is, computations with locality regions at multiple levels of nesting. We describe the hierarchical greedy scheduler, where each locality region is scheduled using a greedy scheduler which attempts to use as many processors as possible while obeying the restrictions posed by the locality regions. We derive a recurrence for the time complexity for a region in terms of its nested regions. We also describe how a more realistic hierarchical work-stealing scheduler can get the same bounds apart from constant factors for an important subclass of computations called homogenous computations. Finally, we also analyze the cache complexity of the hierarchical work-stealing scheduler for a system with a multilevel cache hierarchy.

Hierarchical Scheduling for Multicores with Multilevel Cache Hierarchies

Authors: Kunal Agrawal and Jim Sukha

Corresponding Author: kunal@cse.wustl.edu

Abstract: Cache-locality is an important consideration for the performance in multicore systems. In modern and future multicore systems with multilevel cache hierarchies, caches may be arranged in a tree of caches, where a level k cache is shared between Pk processors, called a processor group, and Pk increases with k. In order to get good performance, as much as possible, subcomputations that share more data should execute on processors which share a lower-level cache. Therefore, the number of cache misses in these systems depends on the scheduling decisions, and a scheduler is responsible for not just achieving good load-balance and low overheads, but also good cache complexity. However, these can be competing criteria. In this paper, we explore the tension between these criteria for online hierarchical schedulers.
Formally, we consider a system with P processors, arranged in a multilevel hierarchy according to a hierarchy tree, where each of the P processors forms a leaf of the tree, and an internal node at level-k corresponds corresponds to a processor group. In addition, we assume that computations have locality regions, that represent parallel subcomputations that share data. Each locality region has a particular level, and the scheduler must ensure that a level-k locality region is executed by processors in the same level-k processor group, since they share a level k cache. Thus locality regions can improve cache performance. However, they may also impair load-balance and increase scheduling overheads since the scheduler must obey the restrictions posed by locality regions.
In this paper, we present a framework of hierarchical computations, that is, computations with locality regions at multiple levels of nesting. We describe the hierarchical greedy scheduler, where each locality region is scheduled using a greedy scheduler which attempts to use as many processors as possible while obeying the restrictions posed by the locality regions. We derive a recurrence for the time complexity for a region in terms of its nested regions. We also describe how a more realistic hierarchical work-stealing scheduler can get the same bounds apart from constant factors for an important subclass of computations called homogenous computations. Finally, we also analyze the cache complexity of the hierarchical work-stealing scheduler for a system with a multilevel cache hierarchy.

Hierarchical Scheduling for Multicores with Multilevel Cache Hierarchies

Kunal Agrawal　　　Jim Sukha

August 2, 2011

Abstract

Cache-locality is an important consideration for the performance in multicore systems. In modern and future multicore systems with multilevel cache hierarchies, caches may be arranged in a tree of caches, where a level k cache is shared between P_k processors, called a processor group, and P_k increases with k. In order to get good performance, as much as possible, subcomputations that share more data should execute on processors which share a lower-level cache. Therefore, the number of cache misses in these systems depends on the scheduling decisions, and a scheduler is responsible for not just achieving good load-balance and low overheads, but also good cache complexity. However, these can be competing criteria. In this paper, we explore the tension between these criteria for online hierarchical schedulers.

Formally, we consider a system with P processors, arranged in a multilevel hierarchy according to a hierarchy tree, where each of the P processors forms a leaf of the tree, and an internal node at level-k corresponds corresponds to a ***processor group***. In addition, we assume that computations have ***locality regions***, that represent parallel subcomputations that share data. Each locality region has a particular level, and the scheduler must ensure that a level-k locality region is executed by processors in the same level-k processor group, since they share a level k cache. Thus locality regions can improve cache performance. However, they may also impair load-balance and increase scheduling overheads since the scheduler must obey the restrictions posed by locality regions.

In this paper, we present a framework of hierarchical computations, that is, computations with locality regions at multiple levels of nesting. We describe the ***hierarchical greedy scheduler***, where each locality region is scheduled using a greedy scheduler which attempts to use as many processors as possible while obeying the restrictions posed by the locality regions. We derive a recurrence for the time complexity for a region in terms of its nested regions. We also describe how a more realistic ***hierarchical work-stealing scheduler*** can get the same bounds apart from constant factors for an important subclass of computations called ***homogenous computations***. Finally, we also analyze the cache complexity of the hierarchical work-stealing scheduler for a system with a multilevel cache hierarchy.

Keywords: cache complexity, hierarchical scheduling, greedy scheduling, work-stealing

1 INTRODUCTION

To achieve good performance, programs must exploit ***locality*** in its memory references, i.e., it must utilize its caches effectively. For machines with a single processor or core, there is a rich history of work on external memory and hierarchical memory models and algorithms (e.g., [2, 5, 16, 18, 24]), all aimed at reducing the number of cache misses in a program. Exploiting locality is even more important for multicores; any increase in the number of cores on a chip puts pressure on the quantity of cache available for each core.

Unfortunately, multicore systems can have cache hierarchies that are much more complicated than the hierarchies for unicore systems. As shown in Figure 1, multicores may have shared caches, private caches, or a combination of shared and private caches (called ***hierarchical caches***) [9]. Small-depth cache hierarchies appear in existing multicore systems already; for example, one might have a quad-socket system with quad-core processor chips, for a total of 16 cores. Each code may have a private L1 cache, but the cores in each socket share L2 and L3 caches. As the number of cores in a system increase in the future, these hierarchies may become deeper. For example, researchers have been developing both caching models [25], and designing cache-coherence protocols for multicores with hierarchical caches [20].

Researchers have also explored how software might effectively utilize multicores with hierarchical caches. This work falls into two categories. The first category of results [7, 9, 13, 21] is *algorithms-focused*: this work aims to provide cache-optimal parallel algorithms for solving well-known problems such as sorting, Gaussian elimination and certain dynamic programming problems. The second category is *scheduling-focused*: this work aims to analyze the number of cache misses incurred by generic programs when using a particular scheduling policy, and to design new policies that reduce the number of misses. In [1], the authors prove a bound on the number of cache misses by a work-stealing scheduler where processors only have private caches (as in Figure 1(b)), while in [8], the authors prove a bound on the number of cache misses by a work-sharing scheduler, called the ***parallel depth-first scheduler*** when processors have only shared caches (as in Figure 1(c)). Finally, there has also been some recent empirical work on locality-aware schedulers [17, 19, 23].

Jim: I changed this reference. Is this the correct one?

In the analysis of parallel computations, theory usually considers two metrics: time complexity and cache complexity. The traditional objective for scheduling of a parallel computation is to minimize the ***time complexity***, i.e., to achieve good load-balance with small scheduling overhead. Alternatively, one can focus on minimizing the ***cache complexity***, i.e., the number of cache misses incurred when executing the program. Theoretical analyses often consider these metrics separately; in reality, the actual completion time of a program depends on both, since the number of cache misses has a direct impact on the running time and the time complexity bound often serves as a good indicator of load-balance and scheduling overheads. As described in [13], it is possible to describe algorithms which are optimal with respect to both time and cache complexity on systems with hierarchical caches. These algorithms require, however, idealized schedules which are tailored to a specific algorithm, and which incur little to no scheduling overheads (e.g., because a runtime can fix the schedules statically).

For generic parallel computations, there is a tension between the objectives of minimizing time and cache complexity. For example, consider a simple function `F` that reads in an input array `a` of length n, and computes an output array `b` of length n, with the computation of each element of b being independent. If terms of time complexity, we can execute this function on all 16 processors to achieve perfect load-balance. However, n may be small enough that `a`, `b`, and the working set of `F` fits into the cache shared by 4 cores which are on the same chip. In this case, executing the program on 4 processors reduces the number of cache misses and could improve performance, even though it does not provide perfect load-balance.

In this work, we explore how well a dynamic scheduler can do at minimizing both the cache and time

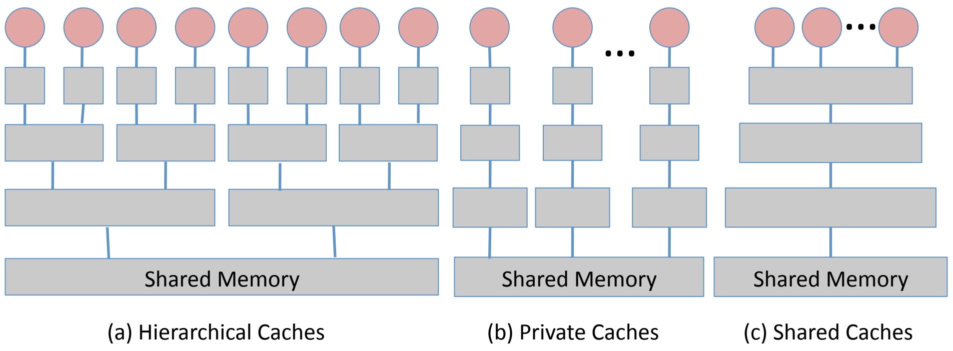

Figure 1: Some possible caching hierarchies in multicores.

complexity of a generic computation, with the goal of understanding the fundamental tradeoffs between these two objectives. In order to have a scheduler optimize for cache locality as well as load balance, it needs to have some information as to which parts of the computation share common data and should be executed using processors that share a cache. Thus, we consider computations which have ***locality regions***, i.e., sections of code that have the constraint that the runtime should schedule this computation on a set of processors which are "close" to each other. Adding locality regions to a computation essentially constrains the scheduler's decisions. If the locality regions are well specified, then this can lead to better cache performance than the same computation without locality regions. On the flip side, since the scheduler is constrained, it may not be able to provide the best possible load balance and the scheduling overhead increases. Computations with locality regions are ***hierarchical***; as a processor can access caches at multiple levels, it can also be executing in multiple locality regions, properly nested inside one another.

The contributions of this work are threefold:

- We describe an idealized hierarchical greedy scheduler (*HGS*) for executing computations with locality regions, and prove a completion time bound for *HGS*.
- For an important subset of applications, namely homogenous computations, we describe a more realistic hierarchical work-stealing scheduler (*HWS*), and show that its time complexity differs from that of *HGS* only in terms of constant factors.
- We analyze the cache-complexity of *HWS* for homogenous computations.

More specifically, we provide a recursive time bound for *HGS* for analyzing the completion time of locality regions at a given level of nesting. To provide intuition as to the meaning of this recursive formula, we solve this recurrence explicitly for a restricted class of homogenous computations generated by simple divide-and-conquer algorithms. From these results, we observe that hierarchical scheduling has a cost in terms of time complexity when we use dynamic schedulers with overheads, since constant factors on overhead in the bound at one level of nesting can multiply up the hierarchy, potentially increasing the cost exponentially with the number of levels in the caching hierarchy. In other words, if we choose to perform hierarchical scheduling in order to get cache benefits, then the cache benefits must be large enough to make up for diminishing load-balance.

The remainder of this report is organized as follows. In Section 2, we present the framework of hierarchical computations. In Section 3, we describe *HGS*, the idealized hierarchical greedy scheduler, and use it to illustrate the impact that locality regions have on time complexity. In Section 4, we describe *HWS*, a more realistic hierarchical scheduler that uses work-stealing to execute homogenous hierarchical computations.

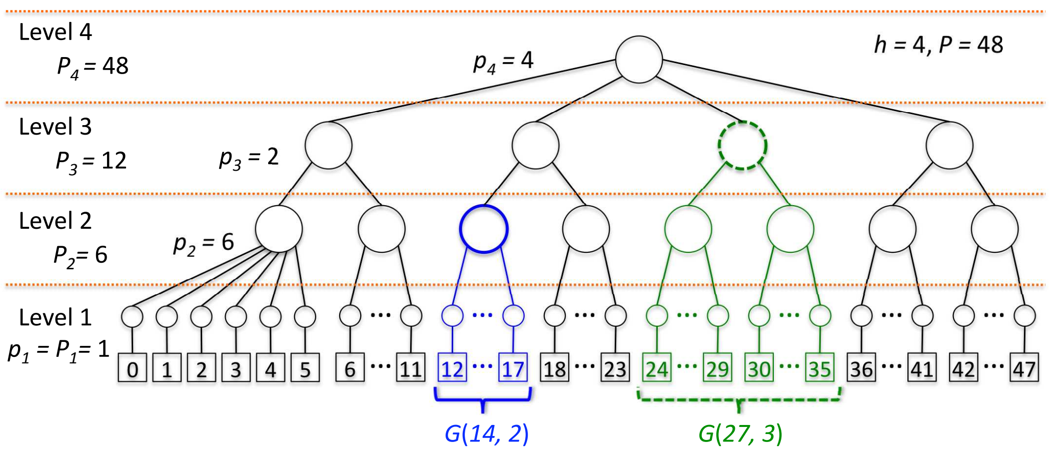

Figure 2: An example hierarchy tree $\mathcal{H}$ for a machine with $P = 48$ processors and $h = 4$ levels. The subtree of the dashed node at level 3 corresponds to the group $G(i,3)$ for $i \in \{24, 25, \ldots 35\}$.

In Section 5, we analyze the cache complexity of using *HWS* to execute arbitrary homogenous computations on a multilevel cache hierarchy, and then conclude in Section 7.

2 HIERARCHICAL COMPUTATIONS

In this section, we describe the model of ***hierarchical computations***, that is, computations designed for multicores with hierarchical caches and which contain "locality regions." We first describe the model for processor topology. Then we describe the notion of locality regions, explain the scheduling restrictions they impose, and discuss how they might be used to reduce the cache complexity of a parallel program.

Processor topology

In this work, we are concerned with processors arranged in the form of a hierarchy of h levels, according to a ***hierarchy tree***, denoted by $\mathcal{H}$. In $\mathcal{H}$, the P processors form the leaves of the tree (at height 0), and the root of the tree is a node at height h. All processors which are in the subtree of an internal node x_k at height $1 \le k \le h$ belong to the same ***processor group*** at level k and share a level-k cache of size C_k. In this paper, we only consider ***uniform hierarchy trees*** $\mathcal{H}$ — trees where each internal node in $\mathcal{H}$ at level k has exactly p_k children. Let $P_k = \prod_{i=1}^{k} p_k$ be the number of processors in a group at level k, and let the number of processors at the top level be $P = P_h \prod_{i=1}^{h} p_i$. Finally, for any processor i, let $G(i,k)$ be the set of all processors in the same level-k processor group as i. For convenience, we can also use $G(i,k)$ to identify a node in the hierarchy tree $\mathcal{H}$. Note that $G(i,k)$ corresponds to same node for P_k different values of i. Figure 2 shows an example hierarchy tree with $h = 4$.

Hierarchical Computations

Similar to [4] and [11], we model the execution of a program as a parallel traversal of a ***computation dag*** $\mathcal{E}$. Each node u in $\mathcal{E}$ represents a single unit of work, and an edge (u, v) in $\mathcal{E}$ represents the dependency that u must execute before v. We augment this basic computation model by adding ***locality regions*** which are subdags of $\mathcal{E}$. Each locality region X has a ***level***, denoted $\texttt{level}(X)$, associated with it. If $\texttt{level}(X) = k$, then at any time step, only processors belonging to the same level k group (elements of $G(i, k)$ for some i) can execute nodes inside X. For convenience, for a region X, we define $P(X) = P_{\texttt{level}(X)}$, i.e., the $P(X)$ is the maximum number of processors that can be used to execute X at any time step.

We assume that for well-formed hierarchical computations, locality regions satisfy additional structure. First, every region X is enclosed between a root node $\texttt{root}(X)$ and a final node $\texttt{final}(X)$. Second, since locality regions can be nested, locality regions satisfy ***proper nesting***, that is, if Y is nested inside region X, then $\texttt{level}(Y) \leq \texttt{level}(X)$.We say that a region X ***contains*** a node u if u is a node within the subdag for X, including nodes inside any regions Y nested inside X. In contrast, X ***owns*** a node u, or u ***belongs*** to X, only if u is in X's subdag, but not inside any of X's nested regions. Whenever a region Y is nested inside a region X, the predecessor of $\texttt{root}(Y)$ is a special ***region node*** v for Y which belongs to X. As we describe later, a hierarchical scheduler exhibits special behavior when it tries to execute a region node. A node u which is not a region node is called a ***regular node***.

Using locality regions to reduce the number of cache misses

Locality regions are meant to improve cache locality by enclosing a subdag X of the larger computation dag that accesses the same data. Intuitively, since a processor group $G(i, k)$ shares a level k cache, the number of cache misses can be reduced if that group is used to execute X and X's working set fits into a level-k cache. In this paper, we do not focus on how locality regions are prescribed, but are interested in how a scheduler can efficiently execute a computation once locality regions have already been specified. In practice, programmers may directly add locality regions, or the compiler or runtime may translate programmer annotations into locality regions. For example, one can use the space-bound scheduler hint of [12, 13]. The example for matrix multiplication is shown in Figure 3. The runtime system can then use these hints to automatically create locality regions to restrict execution of a subproblem X that fits into a level-k cache of size C_k to only the processors that share that cache. With locality regions, one can achieve the same good cache behavior of for any of the multicore-oblivious algorithms described in [13] which use the space-bound (SB) hints. Locality regions are not restricted to these applications, however, and can potentially be used in other, more irregular applications.

Hierarchical scheduling

A scheduling algorithm assigns processors to nodes that are ***ready*** to execute, i.e., whose predecessors in $\mathcal{E}$ have already been executed. For computations without locality regions, it is valid for a scheduler to assign any ready node to any processor. Locality regions impose additional restrictions on the scheduling. In this paper, we consider ***hierarchical schedulers***, where an independent scheduler operates in every region, and the scheduler for a particular region X assigns processors to only the nodes belonging to X. On each time step, X's scheduler can assign a processor to at most one ready node u (either a regular node or a region node) while obeying the following rules:

- At most one processor can be assigned to a regular node u that is ready. In the next time step, u completes and some successors of u may become ready.

```
 1  void MatMul(int* A, int* B, int* C, int N) SB(3*N*N) {
 2    if (N <= BASE_N) {
        // Base case multiplication.
 3      MatMulBase(A, B, C, N);
 4    }
 5    else {
        // Split each matrix into 4 submatrices.
 6      int *A11, *A12, *A21, *A22;
 7      int *B11, *B12, *B21, *B22;
 8      int *C11, *C12, *C21, *C22;
 9      split(A, N, &A11, &A12, &A21, &A22);
10      split(B, N, &B11, &B12, &B21, &B22);
11      split(C, N, &C11, &C12, &C21, &C22);

        // Spawn first 4 matrix multiplications, then sync.
12      spawn  MatMul(A11, B11, C11, N/2);
13      spawn  MatMul(A21, B11, C21, N/2);
14      spawn  MatMul(A11, B12, C12, N/2);
15      MatMul(A21, B12, C22, N/2);
16      sync;

        // Spawn 4 remaining multiplications.
17      spawn  MatMul(A12, B21, C11, N/2);
18      spawn  MatMul(A22, B21, C21, N/2);
19      spawn  MatMul(A12, B22, C12, N/2);
20      MatMul(A22, B22, C22, N/2);
21      sync;
22    }
23  }
```

Figure 3: A space bound hint for a matrix multiply-add, computing $C \leftarrow C + A \cdot B$. The programmer has specified the space bound as an annotation in line 1.

- Once a region node u for region Y nested inside X becomes ready, it remains ready until Y completes. On any step when X has assigned a processor to u, the scheduler for Y operates and Y is considered ***active***.
- Suppose u is a region node for nested region Y with $\texttt{level}(Y) = k$. Then X's scheduler must assign either 0 processors to u, or $P_k = P(Y)$ processors from a single level-k group to u.

Note that on any time step, a processor may be recursively assigned to many region nodes, corresponding to regions nested inside each other; each region Y has its own scheduler responsible for assigning nodes owned by Y. We impose the all-or-nothing condition in the last rule to avoid cache pollution for a locality region X; if X with $k = \texttt{level}(X)$ is being executed by a processor-group $G(i,k)$, we want the guarantee that no processor $j \in G(i,k)$ is working outside X and affecting the shared cache of $G(i,k)$.

3 HIERARCHICAL GREEDY SCHEDULING

In this section, we explain the greedy scheduling property, and introduce an idealized hierarchical greedy scheduler, *HGS*, for computations with locality regions. *HGS* is an idealized scheduler, in that on each step, it always finds a maximal assignment of processors obeying the scheduling restrictions described in Section 2. It is useful for illustrating the impact of locality regions on theoretical completion-time bounds.

Greedy Scheduling Property

A ***greedy scheduler*** is one that on each step, assigns processors to as many ready nodes as possible, i.e., it leaves a processor idle only if all ready nodes already have assigned processors. The goal of greedy scheduling is to minimize idleness, thereby achieving a small completion time. We will use this greedy scheduling property to provide intuition about the cost of using hierarchical scheduling. For computations without locality regions, any scheduler satisfying the greedy-scheduling property is known to provide a completion time that is within a constant factor of the optimal completion time. Thus, it is natural to ask, how much worse does greedy scheduling get if we impose the restrictions of locality regions?

To tackle this question, we design a ***hierarchical greedy scheduler (HGS)*** that extends the greedy-scheduling property to computations with locality regions, assuming an independent greedy scheduler is used for every region, each obeying the rules explained in Section 2. In this case, the definitions of idle processors change. A processor p is said to be ***idle with respect to a region*** Y if p has been assigned to Y (i.e., p is assigned to Y's region node by the scheduler operating at Y's parent X), and Y's scheduler does not assign it to any ready node. Note that in this case, p is idle with respect to Y, but it is *not* idle with respect to X. This definition allows the scheduler at each region to follow the greedy scheduling property independently, that is, each scheduler can assign as many processors to ready nodes as possible.[1]

Completion Time Bound for HGS

We can now prove a bound on the time that *HGS* requires to execute a region X. Define $T(X)$ as the number of time steps that region X is active, i.e., the number of time steps when $P(X)$ processors are assigned to the region node of X. If X is the global region for the entire computation, $T(X)$ is an upper bound on the time

[1]Note that for arbitrary computations, *finding* an assignment that satisfies the greedy-scheduling property efficiently in an online setting might be difficult. However, our purpose in describing greedy scheduling is to understand how much worse locality regions can make the completion time bounds, even using an idealized scheduling technique that is not practical to implement.

HGS requires to finish the computation. In this proof, we will bound $T(X)$ recursively using $T(Y)$ for all regions Y nested directly inside X.

For an execution dag $\mathcal{E}$ without regions, T_1 denotes the ***work*** of the computation (the number of nodes in the dag $\mathcal{E}$), and T_∞ denote the ***span*** (also called depth or critical-path length) of $\mathcal{E}$ (number of nodes on the longest path through $\mathcal{E}$). We extend these notations for computations with regions by saying that the work of a region X is $T_1(X)$ where all the nested regions are flattened into X. We also define analogous quantities that ignore the nested regions. Define the ***region work*** of X, denoted by $\tau_1(X)$, as the number of nodes belonging to X. Similarly, define the ***region span*** of X, denoted by $\tau_\infty(X)$, as the number of nodes on the longest path through X, counting only nodes owned by X. Note that the work, span, region work and region span are properties of the computation and do not depend on the scheduler.

In addition, for analysis purposes, for every region X, we consider a ***region execution dag***, denoted by $\mathcal{C}(X)$, which intuitively is the execution dag from the point of view of X's scheduler. Consider a region Y nested inside X. X's scheduler can not see inside Y. All it can do is assign $P(Y)$ processors to Y's region node for $T(Y)$ time steps. Therefore, for the region execution dag $\mathcal{C}(X)$, the subdag for every child region $Y \in N(X)$ (the dag between `root`(Y) and `final`(Y)) is replaced by $P(Y)$ serial chains, each of length $T(Y)$. Then, we define the ***execution work*** of X, $\widetilde{T}_1(X)$, as the number of nodes in $\mathcal{C}(X)$. Similarly, we define the ***execution span*** of X, denoted by $\widetilde{T}_\infty(X)$, as the number of nodes on the longest path through $\mathcal{C}(X)$.

Theorem 1 *Suppose HGS executes a region X in time $T(X)$. Then $T(X)$ satisfies*

$$T(X) \quad \leq \quad \frac{\tau_1(X)}{P(X)} + \widetilde{T}_\infty(X) + \sum_{Y \in N(X)} \left(\frac{P(Y)T(Y)}{P(X)} \right)$$

PROOF.

At any time step when X is active, its scheduler has $P(X)$ processors available to assign. Consider a time step when it assigns all these processors. Each of these processors is assigned to either a regular node or a region node. A regular node remains ready for at most 1 time step, and there are $\tau_1(X)$ of them. A region node for Y remains active for $T(Y)$ time and is assigned $P(Y)$ processors. Therefore, the total number of such time steps when all processors are assigned is at most $(\tau_1(X) + \sum_{Y \in N(X)} P(Y)T(Y))/P(X)$.

Considering the time steps when some processors are idle, it can only happen if all the ready nodes have been assigned enough processors. In all these time steps, the remaining execution critical path reduces by 1, therefore there are at most $\widetilde{T}_\infty(X)$ such time steps. $\square$

In order to understand this bound, let us compare it with the bound for greedy scheduling when there are no locality regions. For computations without regions, greedy scheduling guarantees a completion time of $T \leq T_1/P + T_\infty$, which is within a factor of 2 of the optimal completion time. Similarly, hand, Theorem 1 has the bound $\widetilde{T}_1(X)/P + \widetilde{T}_\infty(X)$. For the most leaf (most deeply nested) locality regions, we get the same bound, since $\widetilde{T}_1(X) = T_1(X)$ and $\widetilde{T}_\infty(X) = T_\infty(X)$.

For nonleaf regions, however, the bound in Theorem 1 can be significantly worse than the same computation without regions because locality regions in the computation could be poorly specified. For example, one can construct a Y which has a lot of parallelism, but is at a level k where P_k is small. A scheduler that respects locality regions has no choice but to use just P_k processors, effectively increasing Y's span. Similarly a region Y at a large level k could have little parallelism, wasting processor cycles and increasing completion time, since the scheduler is forced to allocate P_k processors a region that is mostly sequential.

Another cost of hierarchical scheduling is that inefficiencies in scheduling can potentially have a multiplicative effect at each level in the hierarchy tree.

Theorem 2 *For HGS, there exists a computation X with locality regions for which the worst-case greedy schedule is a factor of μ worse than the optimal (greedy) schedule, where*

$$\mu = \prod_{k=1}^{h} \left(2 - \frac{1}{p_k}\right).$$

PROOF. We can construct a family of worst-case computations X_k, with $k = \texttt{level}(X_k)$ recursively, out of worst-case regions X_{k-1} at level $k-1$. Let $T^*(X_k)$ be time required by an optimal schedule to complete X_k, and let $T(X_k)$ is the time required for our worst-case schedule using *HGS*.

Since $P_1 = p_1 = 1$, level-1 regions are always scheduled optimally; thus, for the base case, we first construct a worst-case region X_2 composed out of level-1 "regions." It is a simple exercise ([15], Exercise 27.1-4) to construct a DAG for which greedy scheduling is in the worst case, nearly a factor of 2 worse than optimal. Consider a parallel computation which has a single chain of cp_2 nodes, which can run in parallel with $c(p_2 - 1)p_2$ other nodes (all of which can run in parallel with each other). An optimal (greedy) scheduler will use one processor to execute the chain, and use the $p_2 - 1$ other processors to execute the other nodes. The one processor chain finishes in time cp_2 time, and the other processors also take cp_2 time to finish all the other work. On the other hand, the pessimal greedy schedule will use all p_2 processors to execute the $c(p_2 - 1)p_2$ work first, in $c(p_2 - 1)$ steps. Then, it will be left with a serial chain that takes cp_2 time to finish, taking a total of $c(2p_2 - 1)$ time. Thus, the worst-case greedy schedule requires spends time which is a factor of $2 - 1/p_2$ worse than the optimal schedule, i.e., $T(X_2) = (2 - 1/p_2)T^*(X_2)$.

Similarly, we can construct a level-3 region X_3 with out of copies of the level-2 region X_2: create a chain of p_2 level-2 regions, and $(p_2 - 1)p_2$ other level-2 regions. Assuming the same greedy scheduler is used to schedule each instance of X_2, then the worst-case greedy schedule for *HGS* at level 3 is a factor of $(2 - 1/p_3)$ slower than the optimal greedy schedule at level 3. For the combined computation, in the worst case, *HGS* could choose the pessimal schedule at both levels 1 and 2, while the overall optimal greedy schedule is to choose the optimal greedy schedule at both levels. Thus, at level 2, $T(X_3) = (2 - 1/p_3)(2 - 1/p_2)T^*(X_3)$.

Repeating this construction at all levels gives us the desired worst-case computation X_h. □

Theorem 2 demonstrates that in *HGS*, the inefficiency of greedy scheduling at each level in the hierarchy can have a multiplicative effect. In the worst case, if each $p_k = 2$, then $\mu = (3/2)^h = P^{(\lg 3)-1} \approx P^{0.585}$. Thus, *HGS* is getting sublinear speedup for this computation. Intuitively, since a processor idle at the deepest level is still considered part of an active processor group at the next level, in terms of scheduling, any inefficiency in a leaf region compounds as we go up the hierarchy tree. This seems to be a fundamental problem with hieararchical scheduling, and it is exacerbated when we consider a more realistic scheduler with even higher overheads in the next section.

Making the scheduler more realistic

HGS is an idealized scheduler that would likely have bad performance in practice, since it might repeatedly ***preempt*** a locality region, that is, remove a group from an active region before it finishes. More precisely, *HGS* is idealized in the following ways:

- *HGS* assumes that one can schedule a locality region Y with $P(Y) = 4$ on one 4-processor group on one time step and move it to an entirely different 4-processor group on the next. This preemption defeats the purpose of locality regions, since the cache advantages would be lost.

- The scheduler is obliged to make centralized decisions about processor assignments on each time step, introducing high scheduling overheads.

A practical hierarchical scheduler should avoid preemptions whenever possible. However, for computations with arbitrary regions, this goal is at odds with the desire to maintain the greedy scheduling property. For example, consider the case when three regions X_1, X_2 and Y are in parallel with each other, and $P(X_1) = P(X_2) = 4$ while $P(Y) = 8$. Say X_1 is executing on processors 0 through 3 and X_2 is executing on processors 8 through 11. Now when Y becomes ready, there is no free processor group with 8 processors that can be allocated to Y without preempting X_1 or X_2. Therefore, if we do not wish to preempt the other regions, we have two options: (1) Y can start executing with fewer than $P(Y)$ processors, or (2) Y can wait until one of the other regions finishes. Option 1 may lead to cache thrashing since in our model Y expects to have the entire level 3 cache to itself. On the other hand, option 2 may violate the greedy scheduling property since some processors may remain idle even though Y is ready.

If we wish to allow no preemptions of regions and try to be greedy, we can restrict ourselves to just ***homogenous computations***. A region X is ***homogenous*** if for any $Y_1, Y_2 \in N(X)$ which can execute in parallel in X, $\texttt{level}(Y_1) = \texttt{level}(Y_2)$. Given a homogenous computation, one can design a greedy scheduler where all regions finish to completion once started. This property should allow for good cache performance.

More formally:

Definition 1 *A region X is* ***homogenous*** *if any $Y_1, Y_2 \in N(X)$ which can execute in parallel in X, $\mathtt{level}(Y_1) = \mathtt{level}(Y_2)$. A computation is homogenous if all its regions are homogenous.*

For the rest of this paper, we only consider homogenous computations. For homogenous computations, we can construct greedy schedulers that never preempt a region. That is, once a region begins execution on a particular processor group, that particular processor group executes the region to completion before doing any other work. We call this scheduler a non-preemptive greedy scheduler. For the rest of this paper, we only consider non-preemptive schedulers.

4 A HIERARCHICAL WORK-STEALING SCHEDULER

In this section, we describe *HWS*, a hierarchical scheduler that uses work-stealing within regions to execute homogenous computations. *HWS* uses a more realistic scheduler than *HGS* described in Section 3, but is able to achieve analogous completion time bounds. We first give an overview the work-stealing scheduler in *HWS*, outlining the data structures used and focusing on the differences between *HWS* and traditional work-stealing schedulers. Then, we give a more formal description of the operational model for *HWS*. Finally, we state the completion time bound for *HWS*.

As we mentioned in the previous section, greedy schedulers can not be both greedy and non-preemptive at the same time for non-homogenous computations. Since work-stealing schedulers also strive to be greedy, work-stealing schedulers can not be noth greedy and non-preemptive at the same time. Therefore, we restrict our attention to homogenous computations.

Work-Stealing Schedulers

Work-stealing is a common scheduling technique used to execute ordinary, nonhierarchical computations. For a traditional work-stealing scheduler (e.g., as described in [4] and [11]), on a system with P processors, the runtime system maintains P ***deques***, one for each processor i. Normally, each processor i pushes and pops work from the bottom of its local deque, but i may steal work from the deques of other processors when its own deque is empty. To be more precise, when processor i spawns a function `G` from within a function `F`, i begins executing `G`, and pushes the continuation of `G` (in `F`) onto the bottom of its deque. When i stalls at a sync in the program, it first tries to pop work from the bottom of its deque. If i's deque is empty, then i chooses a victim processor j uniformly at random, and tries to steal work from the top of the j's deque.

Work-Stealing in HWS

Like hierarchical greedy scheduler, *HWS* is also hierarchical in that processors assigned to a particular region only operate within that region. However, unlike *HGS*, *HWS* is entirely distributed, and there is no central scheduler that assigns processors to nodes and regions. Instead, in *HWS*, processors themselves find work to do via work stealing, and join and operate within appropriate regions with the help of some data structures. For every active region X, *HWS* maintains a ***deque pool***, denoted by $\texttt{dqpool}(X)$, which is a collection of deques that contain exactly the work in X that is ready to execute.

At any instant in time, a processor i can be ***assigned*** to a chain of one or more regions, $X_0, X_1, \ldots X_d$, with i having a deque in the pool $\texttt{dqpool}(X_j)$ for each region X_j in this chain. The regions in this chain must correspond to properly nested locality regions, i.e., X_{j+1} is nested inside X_j (and thus, $P(X_j) > P(X_{j+1})$). The deepest region in this chain, X_d, is the ***active region*** for processor i. In *HWS*, processor i always works on the ready nodes of its active region X_d; when i runs out of work, it tries to randomly work-steal from deques in $\texttt{dqpool}(X_d)$.

The deque chain for a processor i can change when the following actions occur:

1. **Processor i starts a nested region Y.** When i starts a region Y nested inside X_d, processor i creates a new deque pool $\texttt{dqpool}(Y)$, which starts with a deque for i. Thus, Y is added to i's deque chain and Y becomes the active region for i.

2. **Processor $j \neq i$ starts a nested region Y.** A different processor j working in X_d may start a nested locality region Y, and we may have $i \in G(j, \texttt{level}(X_{d+1}))$, i.e., i belongs to j's group. In this case, once processor i notices that j has started Y, processor i ***suspends*** its current deque q_d in X_d, and then enters Y by creating a new deque q_{d+1} for itself in $\texttt{dqpool}(Y)$ and changing its active region to Y.[2]

3. **Processor i finishes working in a nested region Y.** In this case, processor i finishes working on its deque q_{d+1} in $\texttt{dqpool}(Y)$, removes q_{d+1} from the bottom of its deque chain, and then resumes execution of the bottom deque q_d on its chain (which was previously suspended).

Jim: This section is still a bit confusing. I made some claims about some properties following from homogeneity, but it isn't quite obvious to me why these things should be true from the description.

Note that in Case 2, whenever a processor $j \neq i$ tries to start a nested region Y, we know that Y is nested inside i's active region X_d by our restriction that *HWS* is executing a homogenous computation. By Definition 1, we know that at the time when j is starting Y nested inside X_{d+1}, that processor i can not be working in any locality region Z that does not also require processor j.

Synchronization for Locality Regions

When a processor i starts a locality region X at level k, then other processors in $G(i,k)$ must somehow find out and start working on X. In this section, we describe the protocol used by *HWS* to start locality regions.

To synchronize when starting regions, *HWS* maintains three atomic variables for every group $G(i,k)$ in the hierarchy tree $\mathcal{H}$: (1) a ***status field*** and (2) ***active region*** and (3) a ***waiting counter***. The group $G(i,k)$

[2]Note that although q_{d+1} is below q_d in i's deque chain, the work on these two deques are unrelated in terms of the call stack. More precisely, since $j \neq i$, i.e., i was not the "primary" worker that started Y, the work on deque q_d is not an ancestor of the work on q_{d+1} in the call chain.

has status of ACTIVE when it is executing a level-k region, INACTIVE when the processors in $G(i,k)$ are not working on a level-k region, and PENDING when some processor $j \in G(i,k)$ is trying to start a level-k region. When a processor $j \in G(i,k)$ tries to start a level-k region X, it tries to atomically switch the status of $G(i,k)$ from INACTIVE to PENDING. If it succeeds, it sets the active region field to X and resets the waiting counter of $G(i,k)$ to 1. Eventually, all other processors $j' \in G(i,k)$ notice that the status field is PENDING. At this point, j''s suspends its current work (if any), abandons its deque, increments the waiting counter of $G(i,k)$ and assigns itself to dqpool(X). When j notices the waiting counter equal to P_k, it can switch the status of $G(i,k)$ to ACTIVE, and the entire group $G(i,k)$ starts working in X.

There are two ways in which j' notices the PENDING status. First, in *HWS*, each processor $j \in G(i,k)$ is periodically polling the group status fields of $G(i,k)$ according to a ***polling frequency function*** $\delta(k)$. Second, if processor j' may notice when i tries to start another region Y. Since Y is in parallel with X and they are both nested inside the same outer region at level $k' > k$, we know that $\texttt{level}(Y) = k$. Therefore, j' will try to change the status field and notice that it is already pending. Again, it will suspend the region node for Y and join X.

In both cases above, j' might suspend its currently assigned node in order to enter X. This suspended node remains as the assigned node for j' in X's parent's deque pool. Eventually, i can resume it, when it returns from X). *HWS* also allows suspended nodes to be stolen; we refer to the steal of an assigned node as a ***mugging***.

Completion Time Bounds

For the *HWS* scheduler, one can prove a recursive completion time bound analogous to the result in Section 3. Note, however, that the completion time $T(Y)$ for a leaf region Y is a random variable, since scheduling inside Y is done using randomized work-stealing. Thus, $\widetilde{T}_1(X)$ and $\widetilde{T}_\infty(X)$ are also random variables for nonleaf regions X. We can prove the following theorem:

Theorem 3 *Suppose HWS executes a region X with* $\texttt{level}(X) = k$ *in time* $T(X)$*. At level k, let* s_k *be the number of time steps required for a steal or poll operation, and* $\delta(k)$ *be the polling interval at level k, with* $\delta(k) \geq 2hs_k$*. Then, for some constant* $c \leq 192$,

$$T(X) \quad \leq \quad \left(\frac{\tau_1(X)}{P(X)} + cs_k\widetilde{T}_\infty(X) + \frac{cs_k}{4}\ln\left(\frac{1}{\varepsilon}\right)\right)\left(1 + \frac{1}{\delta(k)}\right) + \sum_{Y \in N(X)} \left(\frac{P(Y)T(Y)}{P(X)}\right) + \delta(k) + s_k$$

with probability at least $1 - \varepsilon$.

PROOF. See Appendix B. □

Theorem 3 is analogous to Theorem 1 for greedy scheduling, except with additional terms to account for polling and work-stealing.

Implications of the completion time bound

As we saw in Section 3, in hieararchical scheduling, the scheduling overheads and inefficiencies are compounded as we go up the hierarchy. The same is true for the hiearchical work-stealing scheduler, but the problem may be worse since the scheduling overheads are larger.

Two factors come into play when trying to understand the cost of using locality regions. First, due to hierarchical scheduling, overheads can multiply up the hierarchy tree. In order to understand this, let us compare this bound with the bound for work-stealing when there are no locality regions. For computations without regions, work-stealing guarantees a completion time of $T \leq O(T_1/P + T_\infty) \leq K_1 T_1/P + K_2 T_\infty$, which is within a constant factor of the optimal completion time. On the other hand, Theorem 1 has the bound $K_1 \widetilde{T}_1(X)/P + K_2 \widetilde{T}_\infty(X)$. If we unroll this recursion, the running time at the highest level may not be within a constant factor of optimal. In fact, it is within $\prod_{1 \leq i \leq h} \rho_i$ factor of optimal where ρ_i is a parameter that depends on the structure and parallelism of level-i regions, with $\rho_i \leq K_1 + K_2$. Therefore, in the worst case, the completion time may increase exponentially with the depth of the hierarchy tree. Second, one can construct computations with poorly-specified locality regions, so that no hierarchical scheduler can provide good speedup, even in the absence of overheads. For example, one can construct a Y which has a lot of parallelism, but is at a level k where P_k is small. A scheduler that respects locality regions has no choice but to use just P_k processors, effectively increasing Y's span. Similarly a region Y at a large level k could have little parallelism, wasting processor cycles and increasing completion time, since the scheduler is forced to allocate P_k processors to this sequential region.More intuitively, since a processor idle at the deepest level is still considered part of an active processor group at the next level, in terms of scheduling, any inefficiency in a leaf region might compound as we go up the hierarchy tree. These observations suggest that (1) all regions should have sufficient parallelism so that scheduling overheads don't dominate, and (2) the cache benefits we get from the locality regions need to dominate the loss due to increase in scheduling overheads.

5 CACHE-COMPLEXITY BOUNDS FOR *HWS*

In this section, we examine the cache complexity of computations executed using *HWS*. In Section 2, we mentioned briefly that if locality regions are specified using the space bound hint, such that each locality region at level k fits in a level k cache, then for certain types of applications, *HGS* and *HWS* can provide optimal cache complexity due to the results proven in [13]. In this section, we derive a more general worst-case bound on the number of cache misses that *HWS* incurs when executing nested-parallel race-free computations, assuming a hierarchical cache model with ideal replacement policies. As is common in the literature (e.g., in [1] and others), we consider the restricted class of ***nested-parallel computations***, since without this restriction, the theoretical worst-case bound on cache misses for a parallel execution can be significantly worse than the number of misses incurred by a sequential execution [10]. This bound illustrates more rigorously that for some computations, having locality regions can improve cache complexity because processors are able to exploit a shared cache; thus the theory matches the intuition that well-placed locality regions can improve program performance.

Jim: I really want to say something stronger here, but it might not be true... We said "cache misses dues to scheduling" but I don't remember why...

To analyze the effects of caching on hierarchical work-stealing, we consider hierarchical cache models from the literature, namely, hierarchical multi-level multicore (HM) model of Chowdhury *et al.* in [13] and the Parallel Tree-of-Caches (PToC) model of Blelloch *et al.* in [9]. We assume each processor group in the hierarchy tree at level k shares an cache of size C_k, which operates on blocks (lines) of size B_k. Each level-k cache is assumed to satisfy a regularity condition, $C_k \geq c \cdot p_k \cdot C_{k-1}$ for some constant $c > 1$, i.e., a level-k cache is at least a constant factor larger than the combined sizes of its level-$k-1$ caches and the caches are assumed to be inclusive. We also assume every cache miss at level k has latency of m_k time steps.

To state the bound, we require several definitions. For *HWS*, let $Q(X, Z, k)$ denote the cache complexity at level k executing a region X (using a group of $P(X)$ processors) in the HM / PToC model, assuming an ideal cache replacement policy, and assuming Z is the total size of all level-k caches. To analyze *HWS* for a particular cache hierarchy, we want to bound $Q(X, C_k P/P_k, k)$ for all levels k; however, we assume $Q(X, Z, k)$ is defined for any $Z \geq C_k P/P_k$. Note that for *HWS*, the cache hierarchy is implicit in the definition of $Q(X, Z, k)$; Z is always divided into P/P_k distinct caches, each shared between a group of P_k processors. We analyze $Q(X, Z, k)$ by comparing it to the number of cache misses incurred by a corresponding sequential execution of X. Let $M_p^v(X,C)$ be the cache complexity of *HWS* executing X using a group of $P(X)$ processors, assuming *each* processor has a private *simple cache* (as defined in [1]) of size C. Finally, define $\widetilde{M}_p^v(X,C)$ as the cache complexity of executing a region X on p processors, assuming every processor has a simple private cache of size C, but not counting any cache misses incurred by the execution of any regions Y nested inside X. By definition, we always have $\widetilde{M}_p^v(X,C) \le M_p^v(X,C)$.

Theorem 4 . *Consider a homogenous region X whose execution uses at most $\mathcal{S}(X)$ space. Suppose for all $Y \in N(X)$, we have $\texttt{level}(Y) = j$. For any level k, define $Q^r(X,Z,k)$ as*

$$Q^r(X,Z,k) = \sum_{n=1}^{|N(X)|} \left(Q\left(Y_i, \frac{ZP_j}{P(X)}, k\right) + \frac{ZP_j}{B_k P(X)} \right) + \widetilde{M}_1^v\left(X, \frac{Z}{P(X)}\right) + O\left(\left\lceil \frac{m_k}{s} \right\rceil \frac{Z}{B_k} \left(\widetilde{T}_\infty(X) + \ln(1/\varepsilon) \right) \right).$$

Then, we have

$$Q(X,Z,k) \le \begin{cases} Q^r(X,Z,k) & \text{if } \texttt{level}(X) > k \text{ or } \mathcal{S}(X) > C_k. \\ O(\mathcal{S}(X)/B_k) & \text{if } \texttt{level}(X) \le k \text{ and } \mathcal{S}(X) \le C_k \end{cases}.$$

PROOF SKETCH. Theorem 4 has two cases for $Q(X,Z,k)$, depending on whether the memory used by X fits into a single level-k cache that is shared or not. In the first case, either X does not fit ($\mathcal{S}(X) \ge C_k$) and/or the execution of X is distributed across processors that might not share a level-k cache ($\texttt{level}(X) > k$). Then, we can adapt and apply the cache-complexity bounds (roughly those given in [9]) for analyzing *HWS*; see Appendix C for the details. Otherwise, in the second case, X fits, and the execution of X only incurs misses to bring the blocks accessed by X into cache. □

The recursive structure of Theorem 4 highlights potential benefits of having locality regions. If a computation X has locality regions Y_i at with $j = \texttt{level}(Y_i)$, then the execution of Y_i can exploit the fact that all P_j processors share a cache for levels k satisfying $j \le k \le \texttt{level}(X)$ (the second case of Theorem 4). Without the locality region Y_i, one must consider interactions between the work from Y_i and other work in X, which makes understanding the cache behavior more difficult; in terms of the theory, the cache misses from the work from Y_i would be mixed into the analysis of the first term $Q^r(X,Z,k)$. Thus, having locality regions can help improve the cache complexity of a parallel computation and/or make it more predictable.

6 UNIFORM

We can now try to understand the effect of hierarchical scheduling on a subclass of applications in order to understand the tradeoffs between time and cache complexity. In order to do so, we apply the completion time bound for *HWS* to a special class of uniform divide-and-conquer computations. Our *HWS* completion time is stated recursively, but for this special class of applications, we can derive a closed-form expression for the running time. For uniform recursive computations, a problem of size n is divided into a recursive subproblems of equal size n/b, for some integer constant b. In this section, we see that if these applications are "sufficiently parallel", then *HWS* can provide good speedup as long as scheduling overheads are low. However, even for these "nice" applications, the constants due to scheduling overheads of realistic schedulers can hamper performance for deep hierarchies. In [13], the authors claim that for ideal hierarchical schedulers (schedulers with no overhead), some of these applications provide linear speedup and ideal cache complexity. We see in this section, however, that for schedulers with overheads, hierarchical scheduling provides tradeoffs between good cache performance and good speedup that must be carefully considered while creating regions.

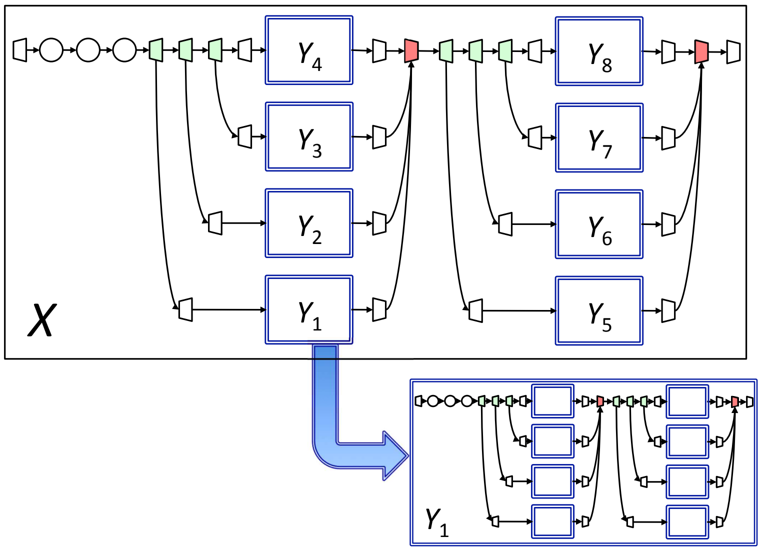

Figure 4: Computation DAG for recursive matrix multiply-add from Figure 3. This computation is a uniform recursive, with $a = 8$, $b = 2$, and $x = 2$, $f(n) = O(1)$, and $g(n) = O(1)$.

We say that the computation X with problem size n is a ***uniform recursive*** computation with parameters (a,b,x) if the computation dag for X can be expressed as a dag with a recursive subcomputations Y_j, where each recursive computation Y_j is a uniform recursive computation with problem size n/b, and x is the largest number of regions Y along any path through the dag for X. For any problem of size n, let $f(n)$ denote the work of X outside the nested recursive calls, and let $g(n)$ denote the longest path through the dag of X ignoring recursive calls. For these computations,

$$T_1(n) = aT_1(n/b) + f(n), \quad T_\infty(n) \le xT_\infty(n/b) + g(n),$$

with constant-size base cases, $T_1(1) = O(1)$ and $T_\infty(1) = O(1)$.[3] For example, Figure 4 shows a uniform-recursive computation, namely a matrix multiply-add of size n which spawns 4 matrix multiply-adds of size $n/2$ in parallel, synchronizes, and then spawns the 4 remaining multiply-adds. For this computation, we have $a = 8$, $b = 2$ and $x = 2$, $f(n) = O(1)$ and $g(n) = O(1)$.

For these applications, in order to get good cache complexity, say locality regions are added using the space bound hint from Section 2. Let $S(n)$ be the amount of space used by the computation for a problem of size n, and let C_i denote the size of the shared cache for processor groups at level i. If n_i represents the maximum-size subproblem that will be executed as a level-i locality region, then we choose $n_i = S^{-1}(C_i)$. For matrix multiplication example, we get $n_i = \sqrt{C_i/3}$. Let $d_i = \sqrt{C_i/C_{i-1}}$ be the depth of recursion that is unrolled in level-i region.

In order to prove that uniform recursive computations get speedup, we must make the assumption that they are sufficiently parallel, both in terms of ***flat parallelism*** and ***region parallelism***. ***Flat parallelism*** is

[3]For simplicity, we assume that a,b and x are all integers, with $a > x \ge 1$ and $b > 1$, and that $f(n)$ and $g(n)$ are "reasonable"functions, so that these recurrences are solvable by the Master method ([14], Section 4.3).

essentially the parallelism of the region without considering any nested regions. γ_1 is the measure of how much the flat parallelism exceeds the number of processors allotted to the region. If γ_1 is large, then no scheduler has any hope of providing speedup since it is forced to waste processor cycles due to lack of sufficient parallelism. ***Region parallelism*** is the ratio between the total number of nested regions at level $j-1$ and the number of nested regions along the span. γ_2 is essentially the measure of how much the region parallelism exceeds the p_j. If γ_2 is large, then the execution work $\widetilde{T}_1(n_i)$ is not large enough compared to $\widetilde{T}_\infty(n_i)$ to overcome the scheduling overheads. γ_2 can also be seen as a measure of scheduling overhead in that it increases with K_2. The exact definitions of γ_1 and γ_2 are given in Definition 2 in Appendix A.

Using these parameters, we can substitute the parameters, solve the recursion and state the closed form completion time result for uniform recursive computations.

Theorem 5 *If a uniform recursive computation which satisfies Definition 2,*

1. *If $T_1(n_i) = O(n_i^{\log_b a})$ and $T_\infty(n_i) = O(n_i^{\log_b x})$ (work and span fall in case 1 of Master Theorem), then $T(n_i) \le K_1(1+\gamma_1)(1+\gamma_2)^i \frac{T_1(n_i)}{P_i}$*

2. *If $T_1(n_i) = O(n_i^{\log_b a})$ or $T_1(n_i) = O(n_i^{\log_b a} \lg n_i)$ (Case 1 or 2 of Master Theorem) and $T_\infty(n_i) = O(n_i^{\log_b x} \lg n_i)$ (Case 2 of Master Theorem), then $T(n_i) \le K_1(1+\gamma_1)(1+\gamma_2)^i \frac{T_1(n_i) \lg i}{P_i}$*

3. *If $T_\infty(n_i) = O(g(n_i))$, then $T(n_i) \le K_1(1+\gamma_1)\left(\frac{(1+\gamma_2)^i - 1}{\gamma_2}\right)\frac{T_1(n_i)}{P_i}$*

In order to understand Theorem 5, let us consider the first case. As γ_1 and γ_2 approach 0, this result bounds the running time by $K_1 T_1(n)/P_i$, that is, we get a speedup of P_i at level i (implying linear speedup at the top level). However, for a given system and computations, as n_i, a, x and P_i remain constant, γ_2 increases with K_2 which is the overhead due to the scheduler. And for large γ_2, the running time increases with the depth of the hierarchy. This analysis suggests that the constant factor overhead K_2 on the span term has a significant impact on the runtime for systems with deep hierarchies. Therefore, for any parallel scheduler with overheads, using this scheduler in a hierarchical manner leads to decrease in performance (in terms of worst case theoretical runtime). In [13], the authors claim linear speedup for arbitrarily deep hierarchies. But they assumed an ideal scheduler with no overheads. Essentially, the $(1+\gamma_2)^i$ term is the theoretical impact of analyzing a scheduler with overheads rather than one without.

Another way of looking at this impact is to notice that it has an impact on the cache size. For example, for matrix multiplication, we have $C_i \ge (P_i K_2/(P_{i-1}\gamma_2)) \cdot C_{i-1}$.[4] Therefore, in order to get the same speedup, the larger the scheduling overhead K_2, larger the ratio C_i/C_{i-1} needs to be. For the ideal scheduler described in [13], the ratio C_i/C_{i-1} must be $2P_i/P_{i-1}$. For an non ideal scheduler with overheads, this ratio might need to be much larger. Thus, unless we can design schedulers with very small overheads, deep hierarchies may be detrimental to performance.

7 CONCLUSIONS

In this paper, we have described hierarchical scheduling for programs with locality regions. We see that hierarchical scheduling provides a tradeoff between time complexity and cache complexity. If correctly specified, locality regions improve cache complexity. But the time complexity may get worse for deep

[4]More generally, suppose a computation has space bound $S(n) = n^\alpha$. Then, for a space-bound scheduler, we need $d_i = \log_b(C_i/C_{i-1})^{1/\alpha} \ge \log_{a/x}(P_i K_2/(P_{i-1}\gamma_2))$, i.e., $C_i \ge C_{i-1} \cdot (P_i K_2/(P_{i-1}\gamma_2))^{\alpha \log_{a/x} b}$.

hierarchies since the overheads tend to multiply up the hierarchy tree. Therefore, the advantage from the decrease in cache misses must be enough to overcome these overheads.

As future work. it would be interesting to understand hierarchical scheduling for other schedulers or combinations of schedulers. For example, the parallel depth-first (PDF) scheduler [6], which is a form of greedy scheduler, has been shown to achieve better cache complexity than work-stealing (e.g., [8]) for some kinds of nested-parallel computations. For the current *HWS* design, it is straightforward to use different schedulers inside a leaf region; the analysis in Theorem 3 still applies, assuming one can bound $T(X)$ for the scheduler being used. It would be interesting to see if one could adapt *HWS* to accommodate PDF schedulers, or hybrids between PDF and work-stealing [22] for higher-level regions as well.

More generally, we think locality regions can be used for purposes other than just reducing the cache complexity. As P increases for future multicores and hierarchies get more complicated, we believe that the encapsulation provided by a locality region will be a useful abstraction. The ability to specialize schedulers for particular regions and analyze the time complexity of program execution in a recursive fashion provides a useful tool for understanding the behavior of programs on future multicore systems.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *SPAA*, pages 1–12, New York, NY, USA, 2000.

[2] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Helper locks for fork-join parallel programming. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Bangalore, India, January 2010.

[4] Nimar S. Arora, Robert. D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, Puerto Vallarta, Mexico, 1998.

[5] D. H. Bailey. FFTs in external or hierarchical memory. *Journal of Supercomputing*, 4(1):23–35, May 1990.

[6] Guy Blelloch, Phil Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

[7] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 501–510, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

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

[9] Guy E. Blelloch, Phillip B. Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In *SPAA '10: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 189–199, New York, NY, USA, 2010. ACM.

[10] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995.

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

[12] Rezaul Alam Chowdhury, Francesco Silvestri, Brandon Blakeley, and Vijaya Ramachandran. Oblivious algorithms for multicores and network of processors. Technical Report TR-09-19, UTCS, 2009.

[13] Rezaul Alam Chowdhury, Francesco Silvestri, Brandon Blakeley, and Vijaya Ramachandran. Oblivious algorithms for multicores and network of processors. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Atlanta, GA, USA, April 2010.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, second edition, 2001.

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[16] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, New York, October 17–19 1999.

[17] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. Slaw: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *International Parallel and Distributed Processing Symposium (IPDPS)*, New York, NY, USA, 2010. ACM.

[18] Jia-Wei Hong and H. T. Kung. I/O complexity: the red-blue pebbling game. pages 326–333, Milwaukee, 1981.

[19] Youngjoon Jo and Milind Kulkarni. Brief announcement: Locality-aware load balancing for speculatively-parallelized irregular applications. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 183–185, New York, NY, USA, 2010. ACM.

[20] Edya Ladan-Mozes and Charles E. Leiserson. A consistency architecture for hierarchical shared caches. In *Proceedings of the 20th ACM Symposium on Parallel Algorithms and Architectures*, pages 11–22, Munich, Germany, June 2008.

[21] Malcolm Yoke Hean Low, Weiguo Liu, and Bertil Schmidt. A parallel BSP algorithm for irregular dynamic programming. In *7th International Symposium on Advanced Parallel Processing Technologies*, pages 151–160. Springer, 2007.

[22] Girija J. Narlikar and Guy E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Transactions on Programming Languages and Systems*, 21(1):138–173, 1999.

[23] G.P. Pezzi, M.C. Cera, E. Mathias, and N. Maillard. On-line scheduling of mpi-2 programs with hierarchical work stealing. In *19th International Symposium on Computer Architecture and High Performance Computing, 2007.*, pages 247 –254, 24-27 2007.

[24] J. E. Savage. Extending the Hong-Kung model to memory hierarchies. In Ding-Zhu Du and Ming Li, editors, *Computing and Combinatorics*, volume 959 of *Lecture Notes in Computer Science*, pages 270–281. Springer Verlag, 1995.

[25] Leslie G. Valiant. A bridging model for multi-core computing. In *Proceedings of the 16th annual European symposium on Algorithms*, pages 13–28, Berlin, Heidelberg, 2008. Springer-Verlag.

A UNIFORM RECURSIVE COMPUTATIONS

Definition 2 *A computation satisfies* ***parallelism conditions*** *with parameters* γ_1 *and* γ_2 *if the following two conditions are satisfied:*

$$\gamma_1 K_1 \frac{T_1(n_i)}{P_i} > K_2 T_\infty(n_1) + K_3, \tag{1}$$

and

$$K_2 \leq \gamma_2 \frac{P_{j-1}}{P_j} \left(\frac{a}{x}\right)^{d_j} \textit{ for all } j \leq i. \tag{2}$$

Condition (1) basically says that if a computation of size n_i is declared a region at level i, then that computation (when all the inner regions are flattened) has sufficient parallelism to use P_i processors effectively. Condition 2 of Definition 2 translates into a minimum unrolling of the recursion at each level in the hierarchy, i.e., a minimum value for d_i, in order to make sure that the execution work $\widetilde{T}_1(n_i)$ is large enough compared to $\widetilde{T}_\infty(n_i)$ to overcome the scheduling overheads. One can rewrite this condition as $d_i \geq \log_{a/x}(K_2 P_j / \gamma_2 P_{j-1})$.

Solving for Time Complexity of Uniform Recursive Computations

In order to analyze the completion time of uniform recursive computations, we solve the recurrence in Theorem 1. We know the running time of a region at level i can be written as

$$T(X) \quad \leq \quad K_1 \frac{\tau_1(X)}{P(X)} + K_2 \widetilde{T}_\infty(X) + \sum_{Y \in N(X)} \left(\frac{P(Y)T(Y)}{P(X)}\right) + K_3$$

Using the recurrence, we can prove the following lemma.

Lemma 6 *Consider a uniform recursive computation* X *with problem size* n_i*. Define quantities* $\widetilde{a}_i$ *and* $\widetilde{b}_i$ *as*

$$\widetilde{a}_i = a^{d_i}, \quad \widetilde{x}_i = K_2 \frac{P_i}{P_{i-1}} x^{d_i}, \quad \widetilde{b}_i = K_1 \frac{\tau_1(n_i)}{P_i} + K_2 \tau_\infty(n_i) + K_3.$$

Then, we have

$$T(n_i) = \sum_{k=1}^{i} \left(\frac{\widetilde{b}_k P_k}{P_i} \prod_{j=k+1}^{i} (\widetilde{a}_j + \widetilde{x}_j) \right).$$

PROOF. One can bound the region work and span as

$$\tau_1(n_i) = \sum_{\ell=1}^{d_i - 1} a^\ell f(n_i / b^\ell), \quad \tau_\infty(n_i) \leq \sum_{\ell=1}^{d_i - 1} x^\ell g(n_i / b^\ell).$$

In this expression, we overload notation and let $\tau_1(n_i)$ to represent $\tau_1(X)$ where X is a problem of size n_i. A uniform recursive region X with problem size n_i, has $|N(X)| = a^{d_i}$, i.e., X has a^{d_i} nested child regions and the number of nested regions on the critical path are x^{d_i}. Therefore, we have

$$\widetilde{T}_\infty(n_i) \leq \tau_\infty(n_i) + x^{d_i} T(n_{i-1}), \tag{3}$$

By substituting Equation (3) into Theorem 1 and collecting terms, we can show that

$$T(n_i) \le \frac{(\widetilde{a}_i + \widetilde{x}_i)P_{i-1}}{P_i} T(n_{i-1}) + \widetilde{b}_i,$$

with base case $T(n_0) = 0$. □

Lemma 7 *For a uniform recursive computation that satisfies Condition 1 of Definition 2, we have* $\widetilde{b}_i < (1+\gamma_1)K_1 T_1(n_i)/P_i$.

PROOF. From the definitions in Lemma 6, and the fact that $\tau_1(n_i) < T_1(n_i)$ and $\tau_\infty(n_i) < T_\infty(n_i)$. □

Lemma 8 *For a uniform recursive computation that satisfies Condition 2 of Definition 2,*

$$\prod_{j=k+1}^{i} (\widetilde{a}_j + \widetilde{x}_j) \le (1+\gamma_2)^{i-k} (n_i/n_k)^{\log_b a}.$$

PROOF. Bounding K_2 in Lemma 6 using Equation (2), we get $\widetilde{x}_i \le \gamma_2 \widetilde{a}_i$. Then, we show the product of $\widetilde{a}_i$ terms is proportional to $(n_i/n_k)^{\log_b a}$ (the number of regions at level k contained within a level-i region). □

Finally, we prove the completion time result for uniform recursive computations stated in Theorem 5.
PROOF. From Lemmas 6, 7 and 8, we know that

$$\begin{aligned}
T(n_i) &= \sum_{k=1}^{i} \left[\frac{\widetilde{b}_k P_k}{P_i} \cdot \prod_{j=k+1}^{i} (\widetilde{a}_j + \widetilde{x}_j) \right] \\
&\le \sum_{k=1}^{i} \left[\left(K_1 (1+\gamma_1) \frac{T_1(n_k)}{P_i} \right) \cdot \left((1+\gamma_2)^{i-k} \left(\frac{n_i}{n_k} \right)^{\log_b a} \right) \right] \\
&= K_1 (1+\gamma_1)(1+\gamma_2)^i \left(\frac{n_i^{\log_b a}}{P_i} \right) \cdot \sum_{k=1}^{i} \left((1+\gamma_2)^{-k} \frac{T_1(n_k)}{n_k^{\log_b a}} \right)
\end{aligned}$$

Now consider the three cases for $T_1(n) = aT_1(n/b) + f(n)$. We know by the Master method, that there exists a constant c_m such that

$$T_1(n) \le \begin{cases} c_m n^{\log_b a} & \text{if } n^{\log_b a - \Delta} = \Omega(f(n)) \\ c_m n^{\log_b a} \log_b n & \text{if } n^{\log_b a} = \Theta(f(n)) \\ c_m f(n) & \text{if } n^{\log_b a + \Delta} = O(f(n)) \end{cases}$$

For the third case to hold, $f(n)$ must also satisfy a regularity condition, that $af(n/b) < \gamma_m f(n)$ for some constant $\gamma_m < 1$.[5]

[5]Technically, we assume slightly stronger conditions on $f(n)$ and $g(n)$, so that the solutions hold for all n, and not just for large n.

When $T_1(n_k)$ falls into Case 1, we get

$$\begin{aligned}
T(n_i) &\leq K_1(1+\gamma_1)(1+\gamma_2)^i \left(\frac{n_i^{\log_b a}}{P_i}\right) \sum_{k=1}^{i} (1+\gamma_2)^{-k} \frac{T_1(n_k)}{n_k^{\log_b a}} \\
&\leq K_1(1+\gamma_1)(1+\gamma_2)^i \left(\frac{n_i^{\log_b a}}{P_i}\right) \sum_{k=1}^{i} (1+\gamma_2)^{-k} \frac{c_m n_k^{\log_b a}}{n_k^{\log_b a}} \\
&\leq K_1(1+\gamma_1)(1+\gamma_2)^i \left(\frac{c_m n_i^{\log_b a}}{P_i}\right) \frac{1}{\gamma_2} \left(1 - \frac{1}{(1+\gamma_2)^i}\right) \\
&= K_1(1+\gamma_1) \left(\frac{(1+\gamma_2)^i - 1}{\gamma_2}\right) \frac{T_1(n_i)}{P_i}
\end{aligned}$$

For Case 2, the math is identical to Case 1 except for the extra $\log_b n_k$ factor. For Case 3, we invoke the regularity condition to show that $f(n_k) \leq (\gamma_m/a)^{D_k} f(n_i)$, where $D_k = \sum_{j=k+1}^{i} d_j$. Then, since $a^{D_k} \cdot n_k^{\log_b a} = n_i^{\log_b a}$, we have $f(n_k) \leq \gamma_m^{D_k} \frac{f(n_i)}{n_i^{\log_b a}}$. Then, substituting for $T_1(n_k)$ gives us the desired result. $\square$

B COMPLETION-TIME BOUNDS FOR *HWS*

This appendix provides a more detailed proof of the completion time bound for *HWS* in Theorem 3. To restate the bound, we have:

Theorem 3 *Suppose HWS executes a region X with* `level`$(X) = k$ *in time* $T(X)$. *At level k, let* s_k *be the number of time steps required for a steal or poll operation, and* $\delta(k)$ *be the polling interval at level k, with* $\delta(k) \geq 2hs_k$. *Then, for some constant* $c \leq 192$,

$$T(X) \quad \leq \quad \left(\frac{\tau_1(X)}{P(X)} + cs_k\widetilde{T}_\infty(X) + \frac{cs_k}{4}\ln\left(\frac{1}{\varepsilon}\right)\right)\left(1 + \frac{1}{\delta(k)}\right) + \sum_{Y \in N(X)} \left(\frac{P(Y)T(Y)}{P(X)}\right) + \delta(k) + s_k$$

with probability at least $1 - \varepsilon$.

Theorem 3 is analogous to Theorem 1 for greedy scheduling, except with additional terms to account for polling and work-stealing. The $(1 + 1/\delta(k))$ term accounts for the overhead of processors polling every $\delta(k)$ time steps to check for potential preemptions. The $\delta(k) + s_k$ term accounts for the time that X spends as `PENDING`, waiting for all $P(X)$ other processors to poll, preempt, and start executing in X. Increasing $\delta(k)$ decreases the overhead due to polling, but increases the possible delay when starting a region.

The term $cs_k \ln(1/\varepsilon)/4$ term accounts for the number of steals needed to achieve the bound with high-probability; the constant is dependent on the analysis of work-stealing. Although the high-probability bound in Theorem 3 applies only for one region X, if X contains a total of N^* regions (whether directly nested or not), one simple way to generate a high-probability bound at the top level is to apply Theorem 3 requiring a success probability of ε/N^*, and applying a union bound over all all regions. This approach changes the $cs_k \ln(1/\varepsilon)$ term to $cs\ln(N^*/\varepsilon)$.

Abstract Model Description

In order to prove Theorem 3, we first present an abstract model for the operation of *HWS*. On each time step, when executing a region X, each processor $i \in G(i, \texttt{level}(X))$ can either be executing a "normal" action, or a "poll" action.

Lets consider normal actions first. Since i is working on X, it has a deque in $\texttt{dqpool}(X)$. Each deque has a (possibly empty) set of ready nodes and an ***assigned node*** which is either ready, suspended or null. Then, on every step, processor i take actions based on their assigned node u.

Consider the deque q which processor i is currently assigned to. Let u be the assigned node for i, or null, and let X be the region that i is currently working in. The various cases for u are as follows:

1. u is a regular node.
   (a) u has a single child v or u is the last predecessor of v to finish. Then i replaces its assigned node with v.
   (b) u is a spawn node with left child v_1 and right child v_2. Then processor i sets its assigned node to v_1, and pushes v_2 onto the bottom of its deque.
   (c) u has a single child v that is a join node, and u is not the last parent of v to finish. Then processor i tries to pop the bottom node x from its deque and set x as its assigned node. Otherwise, i sets its assigned node to null.
2. u is a region node for a region Y and the status of $G(i, \texttt{level}(Y))$ is `INACTIVE`. Then u tries to atomically set the status to `PENDING` for Y and reset the waiting counter of $G(i, \texttt{level}(Y))$ to 1.
3. u is a region node for Y with $\texttt{level}(Y) = k$, and the status of $G(i,k)$ is `PENDING` for a region Y'.

   If $Y' = Y$, and the waiting counter of $G(i,k)$ is P_k, then u successfully starts Y by entering the deque pool $\texttt{dqpool}(Y)$, and sets its assigned node in that deque pool to $\texttt{root}(Y)$. The node u remains as the assigned node for i's deque in $\texttt{dqpool}(X)$.

   Otherwise, $Y' \neq Y$, and i suspends its region node u, increments the waiting counter of $G(i,k)$, and enters the deque pool for Y'.[6]
4. u is $\texttt{final}(X)$ (i.e., worker i is finishing region X). Then, i resets the status of $G(i, \texttt{level}(X))$ to `INACTIVE`.
5. u is null, and the status of $G(i, \texttt{level}(X))$ is `INACTIVE`. In this case, region X has been finished by one of the processors. The, processor i leaves $\texttt{dqpool}(X)$ and returns to X', the parent region of X.

   Let v be its assigned node in $\texttt{dqpool}(X)$. If i's deque in X' has a suspended assigned node v, then worker does the following: v is the region node for X, then it changes the assigned node to the successor of $\texttt{final}(X)$ in X'. If v is a regular node belonging to X', then it resumes v by making it active.
6. u is null, and the status of $G(i, \texttt{level}(X))$ is `ACTIVE`. Then, worker i is stealing in $\texttt{dqpool}(X)$.

   Worker i chooses a victim processor j uniformly at random in $\texttt{dqpool}(X)$, and checks j's deque. The last case can be divided into several subcases, depending on the state of processor j's deque q:
   (a) q is not empty. Then, i steals the top node x from q's deque and sets x as its assigned node as usual.
   (b) q is empty and has no assigned node or an active assigned node v (regular or region). Then the steal fails.
   (c) q is empty, and has an assigned node v which is a suspended regular node. Then i mugs v and executes it.

[6] For this case, it doesn't matter if the processor enters the deque pool of Y' early and tries to steal; there is no work anywhere in Y' until $\texttt{root}(Y')$ gets assigned.

(d) q is empty, and has an assigned node v which is a suspended region node for Y. Then i mugs v and sets it as its active assigned node. (In its next action, i will try to start Y).

To implement polling, every processor i working in X increments its poll counter $\texttt{pcount}(X,i)$ every time step *where i is executing in X*. For a homogenous region X which is currently executing nested regions Y_i at level $k = \texttt{level}(Y_i)$, processor i checks whether its poll counter satisfies $\texttt{pcount}(X,i) \geq \delta(k)$; if so, it resets the counter and polls. While polling for level-k, processor i checks whether the status of $G(i,k)$ is PENDING for region Y. If yes, then i suspends its currently assigned node and enters $\texttt{dqpool}(Y)$. Otherwise, i takes no additional action. We assume that for all k, $\delta(k) \geq \gamma hs$, where $\gamma \geq 2$ is some integer constant. The actions in this model could take more than a single time step. For our model, we assume action 1, when a processor works on a regular node, takes unit time; all other actions (which involve starting regions, polling, or stealing) we assume may take up to s units of time.

Jim: need to fix this part...

Completion Time Analysis

To analyze the completion time for *HWS* executing a region X, for each processor i, we classify each processor step by placing into one of 5 buckets: a poll bucket, a pending bucket, a region bucket, a work bucket, and a steal bucket.

To define the buckets, we require some notation. In any execution $\mathcal{E}$, let $t_p(X)$ denote the time step on which X is marked as PENDING, and let $t_a(X)$ denote the time step on which X is marked as ACTIVE. Let $\texttt{ag}(X)$ denote the processor group that executes X.

Consider a processor $i \in \texttt{ag}(X)$, and any time step t which satisfies $t \in [t_p(X), t_p(X) + T(X)]$. We place time t for processor i into buckets as follows:

- Poll bucket: processor i is executing a poll action for X.
- Pending bucket: we have $t \in [t_p(X), t_a(X)]$. In this case, we say t is a ***pending step*** for i in region X. For actions which span multiple time steps (e.g., if the steal cost is $s > 1$), we also count as pending step, any time spent on the action by i which starts before the specified interval, but which finishes in the interval.
- Region bucket: $t \in [t_a(X), t_p(X) + T(X)]$, and for some child region $Y \in N(X)$, we have $i \in \texttt{ag}(Y)$ and $t \in [t_p(Y), t_p(Y) + T(Y)]$. In this case, we say t is a ***region step*** for processor i in region X.
- Work bucket: on time step t, processor i is executing a regular node.
- Steal bucket: processor i is stealing on step t, but t is not a region step for i. We refer to steals which count towards the steal bucket as ***contributing steals***; other steal attempts may occur on region steps.
- Poll bucket: i polls at time t.

First, we bound the number of steps in the work, pending, and region buckets. One can show there are exactly $\tau_1(X)$ steps in the work bucket for X, $P(X)(\delta\texttt{level}(X) + s)$ steps in the pending bucket and at most $\sum_{Y \in N(X)} P(Y)(T(Y))$ steps in the region bucket. In addition, let L_i be the number of steps in the poll bucket due to processor i. The poll counter $\texttt{pcount}(X,i)$ does not increment on steps which for i fall into the region bucket or the pending bucket, it does increment on steps in work, and steal buckets. The most complicated part of the analysis is to bound the number of contributing steal attempts. In the remainder of this section, we show that the expected number of steps in the steal bucket is $csP(X)\widetilde{T}_\infty(X) + \ln(1/\varepsilon)/4)$ with probability at least $1 - \varepsilon$ for some constant c. Adding up all the steps in all the buckets gives us the bound in Theorem 3.

Contributing Steal Attempts

To bound the number of contributing steal attempts, we use the potential function approach originally described in [4] to analyze work-stealing. The original analysis in [4] has been adapted to include a cost for steals s (e.g., [1]), and also to deal with regions [3]). Our analysis is a variant which combines these elements.

Before defining the potential function, we require several auxiliary definitions. For every node u which belongs to X and is not a region node, define the ***depth*** $d(u)$ of u as the maximum path length for region X over all paths from $\texttt{root}(X)$ to u in $\mathcal{C}(X)$. Define the ***weight*** of a node u as $w(u) = \widetilde{T}_\infty(X) - d(u)$.

For a region node v, we define a time-dependent weight $w(v,t)$ function. Consider a region node v for a region Y, nested in X. Then, we define $w(v,t)$ as

$$w(v,t) = \begin{cases} \widetilde{T}_\infty(X) - d(v) & \text{if } 0 \le t < t_p(Y) \\ \widetilde{T}_\infty(X) - (d(v) + t - t_p(Y)) & \text{if } t_p(Y) \le t \le t_p(Y) + T(Y) \\ \widetilde{T}_\infty(X) - (d(v) + T(Y)) & \text{if } t > t_p(Y) + T(Y) \end{cases}$$

Conceptually, the weight of a region node v decreases by 1 for every time step that the region Y that v corresponds to spends as `PENDING` or `ACTIVE`.

Jim: I think it doesn't matter whether an assigned node is active or suspended.

We now define the potential for nodes and extend it to deques and regions.

Definition 3 *The* ***potential*** *of a node $u \in \mathcal{E}$ is $3^{2w(u)-1}$ if u is assigned, and $3^{2w(u)}$ otherwise.*

We extend the potential to deques as follows. Let q be a deque belonging to worker i, and if q is active, let u be i's assigned node. Define the potential of q as $\sum_{v\in q}\Phi(v)$ if q is inactive, or $\Phi(u) + \sum_{v\in q}\Phi(v)$ if q is active. Similarly, we extend the potential to a region X as $\Phi(X) = \sum_{q\in\texttt{dqpool}(X)}\Phi(q)$.

The following two lemmas are analogous to lemmas from [4]. The proof (omitted) involves induction on each step that a processor takes, checking all the cases in the formal model.

Lemma 9 *For any deque q in any deque pools, let $v_1, v_2, \ldots, v_k$ be the nodes in q ordered from the bottom of the deque to the top, and let v_0 be the assigned node for that deque, if one exists. Then, we have $w(v_0) \le w(v_1) < \cdots < w(v_k)$.*

Lemma 10 *During a computation $\mathcal{E}(X)$, the potential never increases.*

In order to bound the number of contributing steals, we divide the execution of a region X into ***phases***, with each phase j represented as a time interval $[t_j, t'_j]$. Let $t'_0 = t_a(X)$. Phase j begins at time $t_j = t'_{j-1} + 1$, and ends either (1) at a time $t'_j = \max(t_j + 2s, t')$, where t' is first time that satisfies the condition that at least $P(X)$ steal attempts have started and finished in the interval $[t_j, t']$, or (2) ends at the time when $\texttt{final}(X)$ is executed. Note that by this definition, in every phase but possibly the last, at least $P(X)$, and most $2P(X)$ steal attempts can completely fall in the interval $[t_j, t'_j]$. Also, at most $3P(X) - 1$ steal attempts can complete with this interval, since at most $P(X) - 1$ steal attempts could start in phase $j-1$ and end in phase j.

For each phase j, we also classify the deques in $\texttt{dqpool}(X)$ into two types. At the beginning of phase j for X, let $E_j(X)$ be the sum of potentials on the due to deques which are empty, but have an active assigned node v. Let $D_j(X)$ be the potential on all other deques, i.e., nonempty deques, and empty deques which have a suspended assigned node v. Therefore, the potential of X at the beginning of phase j is $\Phi_j(X) = E_j(X) + D_j(X)$.

Lemma 11 *For any phase j for X*

$$\Pr\left\{\Phi_{j+1}(X) - \Phi_j(X) \geq \Phi_j(X)/4\right\} \geq 1/4.$$

PROOF. First, we bound the potential decrease in $D_j(X)$. By definition, every phase (except possibly the last phase) has at least P steal attempts. Thus we can apply Lemma 8 from [4] directly to claim that $D_j(X)$ decreases by a factor of $1/4$ with probability $1/4$.

The analysis relies on the key property that whenever a processor i makes a contributing steal attempt and hits a deque q counted in $D_j(X)$, it is able to steal or mug the highest potential node v from q. If q was not empty, then v is the node on top of q, and assigning the node v reduces the potential as in [4]. If q is empty and v is an assigned node, however, then we must have that v is suspended; otherwise, q would be counted against $E_j(X)$. Also, any processor i which makes a contributing steal is able to mug v, and either execute v immediately if v is a regular node, or start the suspended nested region Y if v is a region node. Suppose for contradiction that processor i failed to start the region Y because of some other region Y'. We know Y' must be `PENDING`, not `ACTIVE` since i is stealing. But then, this steal attempt for i would be counted as a region step for Y', not a contributing steal.

We can also argue that $E_j(X)$ always reduces by a constant fraction. For any deque with an active assigned node v at the beginning of phase j, the potential decreases by more than 1/4 because during the phase, either (1) v is a regular node which gets executed, and replaced with a successor, or (2) v is an active region node (which has status `PENDING` or `ACTIVE`), and the time-dependent potential decreases the potential of v. Note that in the first case, because each phase lasts at least $2s$ steps, each processor is guaranteed to complete at least one non-polling action in the phase.

□

Using the previous lemma, one can prove the following.

Lemma 12 *The number of contributing steals when executing a region X is at most $csP(X)(\widetilde{T}_\infty(X) + \ln(1/\varepsilon)/4)$ with probability at least $1-\varepsilon$, for some sufficiently large constant c.*

C CACHE COMPLEXITY ANALYSIS FOR *HWS*

In this appendix, we give the details of the hierarchical cache model and the proof of Theorem 4. To analyze the effects of caching on hierarchical work-stealing, we consider the HM model from [13] PToC from in [9]. We assume each processor group in the hierarchy tree at level k shares an cache of size C_k, which operates on blocks (lines) of size B_k. Each level-k cache is assumed to satisfy a regularity condition, $C_k \geq c \cdot p_k \cdot C_{k-1}$ for some constant $c > 1$, i.e., a level-k cache is at least a constant factor larger than the combined sizes of its level-$k-1$ caches and the caches are assumed to be inclusive. We also assume every cache miss at level k has latency of m_k time steps.[7]

We are interested in the ***cache complexity*** of a region X at each level k, that is, the total number of block transfers in and out of all level-k caches incurred when executing X.[8] We consider caches with two kinds of replacement policies. First, we consider an ***ideal cache***, that is, one which uses an optimal (offline)

[7] In [9], the authors use C_k to denote the latency of a cache miss at level k. They also use Z_k and L_k to represent cache size and block size, respectively.

[8] For the HM model, this was referred to as total cache complexity. The parallel cache complexity, which could take into account that cache misses may be served in parallel, is more complicated to analyze.

replacement policy. Second, we consider a simple cache (as defined in [1]), which uses a replacement policy which is (1) deterministic, and (2) makes a decision to overwrite a line ℓ in cache only using information about accesses made after the last access to ℓ. We can consider the cache complexity in three models:

1. $Q(X,Z,k)$: The cache complexity at level k of *HWS* executing a region X (using a group of $P(X)$ processors) in the HM / PToC model, assuming the total size of level-k caches used is Z.
2. $M_p^v(X,C)$: The cache complexity of *HWS* executing a computation X using a group of $P(X)$ processors, *each* processor has a *simple* private cache of size C.

To analyze *HWS*, we want to bound $Q(X, C_kP/P_k, k)$ for all levels k; however, we assume $Q(X,Z,k)$ is defined for any $Z \geq C_kP/P_k$. For our analysis, we consider the restricted class of ***nested-parallel computations*** [1], that is, race-free series-parallel computations that exhibit no false sharing.

Jim: For these quantities, we are ignoring the cache misses incurred by the *HWS* runtime / scheduler itself. We should probably make sure that there aren't too many misses due to sharing in the hierarchy tree, since this part isn't race-free anymore.

Traditional Work-Stealing for Nested-Parallel Computations

In this section, we review prior work which analyzes cache misses for nested-parallel computations executed using work-stealing schedulers on cache hierarchies, both flat ([1]) and multilevel ([9]). *HWS* reduces to traditional work-stealing in the case for a computation X with no nested regions.

For nested parallel computations executing on a system where each processor has one private cache, [1] shows how to bound $M_p^v(X,C)$ (the cache complexity of a parallel execution on p processors) in terms of $M_1^v(X,C)$ (the cache complexity of a serial execution), plus an additional term which is proportional to the product of the cache size and the number of "drifted" nodes in the parallel execution. More precisely, for a computation X, let $\mathcal{E}_p$ be a p-processor execution of X, and let $\mathcal{E}_1$ be a single-processor execution. Let u, v be nodes of X, where u is the node immediately executed before v in $\mathcal{E}_1$. A node v is said to be ***drifted*** in $\mathcal{E}_p$ if u is not executed immediately before v by the same processor that executes v.

Lemma 13 *Consider an execution $\mathcal{E}$ of a nested-parallel computation X with $N(X) = \emptyset$ on p processors, each with a private simple cache of size C and block size B. Let D be the number of drifted nodes of X in $\mathcal{E}$. Then,*

$$M_p^v(X,C) \leq M_1^v(X,C) + CD/B,$$

where $D = O(\lceil m/s \rceil p(\tau_\infty(X) + \ln(1/\varepsilon)))$ with probability at least $1-\varepsilon$.

PROOF. Given in [1]. Intuitively, each drifted node may cause a cache to be refilled and the the number of drifted nodes is at most twice the number of successful steals. □

One can apply Lemma 13 to a hierarchical model by partitioning each shared cache of size C_k into P_k private caches of size C_k/P_k to prove the following Theorem. [9]

Theorem 14 *Consider a region X with $N(X) = \emptyset$. When HWS execute X on $p = P(X)$ processors, we have*

$$Q(X,Z,k) \leq M_1^v\left(X, \frac{Z}{p}\right) + O\left(\left\lceil \frac{m}{s} \right\rceil \frac{Z}{B_k}\left(\widetilde{T}_\infty(X) + \ln(1/\varepsilon)\right)\right)$$

with probability at least $1-\varepsilon$.

[9]This result is essentially equivalent to the construction Blelloch et al. use to convert from the PToC model to their PMDH model in [9]. The authors also demonstrate a lower bound computation for which this bound is asymptotically tight.

HWS ***for Nested-Parallel Computations***

Now we consider a homogenous region X, and generalize the bounds of Theorem 14 to regions with nesting. Theorem 14 does not directly translate into a bound for a region X with nested regions because the nested regions change the behavior of steals, and thus affects the counts of drifted nodes. As in Section 4, this cache complexity function is also recursive in terms of the cache complexity of the child computations.

To state the bound, we define $\widetilde{M}_p^v(X,C)$, which is the cache complexity of executing a region X on p processors, assuming every processor has a simple private cache of size C, but not counting any cache misses incurred by the execution of any regions Y nested inside X. By definition, we always have $\widetilde{M}_p^v(X,C) \le M_p^v(X,C)$.

Theorem 4 *Consider a homogenous region X whose execution uses at most $\mathcal{S}(X)$ space. Suppose for all $Y \in N(X)$, we have $\texttt{level}(Y) = j$. For any level k, define $Q^r(X,Z,k)$ as*

$$Q^r(X,Z,k) = \sum_{n=1}^{|N(X)|} \left(Q\left(Y_i, \frac{ZP_j}{P(X)}, k\right) + \frac{ZP_j}{B_k P(X)} \right) + \widetilde{M}_1^v\left(X, \frac{Z}{P(X)}\right) + O\left(\left\lceil \frac{m_k}{s} \right\rceil \frac{Z}{B_k} \left(\widetilde{T}_\infty(X) + \ln(1/\varepsilon) \right) \right).$$

Then, we have

$$Q(X,Z,k) \le \begin{cases} Q^r(X,Z,k) & \text{if } \texttt{level}(X) > k \text{ or } \mathcal{S}(X) > C_k. \\ O(\mathcal{S}(X)/B_k) & \text{if } \texttt{level}(X) \le k \text{ and } \mathcal{S}(X) \le C_k \end{cases}.$$

PROOF.

Theorem 4 has two cases for $Q(X,Z,k)$, depending on whether the memory used by X fits into a single level-k cache that is shared or not. The second case is the simpler case: X fits, and the execution of X only incurs misses to bring the blocks accessed by X into cache.

In the first case, either X does not fit ($\mathcal{S}(X) \ge C_k$) and/or the execution of X is distributed across processors that might not share a level-k cache ($\texttt{level}(X) > k$). Then, as in Theorem 14, we partition the total Z into $p = P(X)$ pieces, one for each processor. When a processor is executing region work of X (i.e., work not inside X's nested regions), it uses its share of the cache as a private cache. The term $\widetilde{M}_1^v\left(X, \frac{Z}{P(X)}\right)$ counts the cache misses due to this work. When P_j processors are working together on a nested region Y_i, however, they each pool together their portions of the cache to execute the region Y_i, using the same replacement policy as an ideal "level-j" cache of size $ZP_j/P(X)$. The sum over $Q\left(Y_i, \frac{ZP_j}{P(X)}, k\right)$ counts the cache misses due to these nested regions.

As in Lemma 13, each drifted node may cause a processor to potentially refill its cache (of size $Z/P(X)$). Drifted nodes occur either because of (1) steals, or (2) when processors resume work in X after entering a nested region Y. Thus, $D \le P_j|N(X)| + O\left(\lceil m_k/s \rceil P(X)(\widetilde{T}_\infty(X) + \ln(1/\varepsilon))\right)$.

$\square$