Bounding Cache Miss Costs of Multithreaded Computations Under General Schedulers

Richard Cole*
Courant Institute, New York University
New York, NY 10012
cole@cs.nyu.edu

Vijaya Ramachandran†
Department of Computer Science, UT-Austin
Austin, TX 78712
vlr@cs.utexas.edu

ABSTRACT

We analyze the caching overhead incurred by a class of multithreaded algorithms when scheduled by an arbitrary scheduler. We obtain bounds that match or improve upon the well-known $O(Q + S \cdot (M/B))$ caching cost for the randomized work stealing (RWS) scheduler, where S is the number of steals, Q is the sequential caching cost, and M and B are the cache size and block (or cache line) size respectively.

CCS CONCEPTS

•**Theory of computation → Shared memory algorithms;**

KEYWORDS

Fork-Join algorithms; cache oblivious; schedulers

ACM Reference format:
Richard Cole and Vijaya Ramachandran. 2017. Bounding Cache Miss Costs of Multithreaded Computations Under General Schedulers. In *Proceedings of SPAA '17, July 24-26, 2017, Washington DC, USA, ,* 12 pages.
DOI: 10.1145/3087556.3087572

1 INTRODUCTION

The design and analysis of multithreaded cache-efficient parallel algorithms has been widely studied in recent years [6, 7, 11, 12, 17, 19]. Many of these algorithms are based on parallel divide and conquer (called variously *hierarchical divide and conquer* [6], *hierarchical balanced parallel (HBP) computations* [13, 14], etc.). The performance of these algorithms is usually analyzed for a specific scheduler, especially with regard to caching costs.

In this paper, we present general bounds on the cache miss cost for several algorithms, when scheduled using an arbitrary scheduler. Our bounds match the best bounds known for work stealing schedulers. The class of algorithms we consider includes efficient multithreaded algorithms for several fundamental problems such as matrix multiplication [19], the Gaussian Elimination Paradigm (GEP) [11], longest common subsequence (LCS) and related dynamic programming problems [9, 11], FFT [19], SPMS sorting [17], list ranking [12], and graph connectivity [12]. These are all well-known multithreaded algorithms that use parallel recursive divide and conquer. Our contribution here is to analyze their caching performance with a general scheduler, as a function of the number of parallel tasks scheduled across the processors (or *steals*), and to obtain bounds that match the current best bounds known only for work stealing schedulers.

We only consider multithreaded algorithms in this paper. As such, we do not directly deal with related work on parallel, cache-efficient algorithms designed for specific models such as the Multi-BSP, Parallel External Memory model, etc. [2–4, 22, 23], though all of the algorithms we consider can be scheduled and analyzed on these models.

1.1 Related Work

In a parallel execution of a multithreaded algorithm, the computation starts on one processor, and the scheduler moves parallel tasks to idle processors as needed. Each move of a parallel task from one processor to another is called a steal.

Let Q be the sequential caching cost of a multithreaded computation, and let $C(S)$ be the caching cost incurred in a parallel execution with S steals. Acar et al. [1] observed that an execution of a computation that incurs S steals when scheduled under randomized work-stealing (RWS) can be partitioned into $O(S)$ fragments, where each fragment runs on a single processor in this parallel execution, and represents a contiguous portion of the sequential execution of the computation. They then observed that the computation regains the state of the sequential execution after reading at most M/B distinct blocks, and therafter inherits the sequential cache complexity. Thus, $C(S)$ is bounded by $O(Q + S \cdot M/B)$.

Frigo and Strumpen [20] considered the above set-up for computations where any fragment of size r that occurs in a parallel execution incurs $O(f(r))$ cache misses, for some concave function f. They then showed that some of the known cache-efficient multithreaded algorithms have good concave functions f satisfying the above property and used this to refine the bound in [1]. If a multithreaded algorithm makes calls to different subroutines with different cache complexities, then the concave function will be at most as good as the least efficient of the caching bounds. Thus, the results in [20] are most effective for cache-efficient algorithms that recursively call only themselves, such as the matrix multiplication

*This work was supported in part by NSF grants CCF-1217989 and CCF-1527568.
†This work was supported in part by NSF Grant CCF-1320675.

SPAA '17, July 24-26, 2017, Washington DC, USA

DOI: 10.1145/3087556.3087572

HBP Algorithm	Seq. Cache Bound Q	Cache miss bound with S steals, $C(S) = \min\{A, B\}$	
		Bound A (Thm. 1)	Bound B (Thm. 2)
Scan, Prefix Sums	n/B	$Q + (M/B) \cdot S$	$Q + S$
Matrix Transpose	n/B	$Q + (M/B) \cdot S$	$Q + S \cdot B$
n^3 Matrix Multiply, GEP	$n^3/(B\sqrt{M})$	$Q + (M/B) \cdot S$	$Q + (n^2/B) \cdot S^{\frac{1}{3}} + S \cdot B$
Strassen Matrix Multiply	$n^\lambda/(BM^\gamma)$	$Q + (M/B) \cdot S$	$Q + (n^2/B) \cdot S^{1-\frac{2}{\lambda}} + S \cdot B$
FFT, SPMS, List Ranking	$\frac{n}{B} \cdot \log_M n$	$Q + (M/B) \cdot S$	$Q + \frac{n}{B} \cdot \frac{\log n}{\log[(n \log n)/S]} + S \cdot B$
Graph Connectivity	$\frac{n+m}{B} \cdot \log_M n$	$Q + (M/B) \cdot S$	$Q + \frac{n+m}{B} \cdot \frac{\log^2 n}{\log[((n+m) \log n)/S]} + S \cdot B$
Finding LCS sequence	$n^2/(BM)$	$Q + (M/B) \cdot S$	$Q + (n/B) \cdot \sqrt{S} + S \cdot B$

Table 1: Our upper bound for cache miss cost, $C(S)$, with S steals, for a general scheduler; $O(\cdot)$ is omitted. The sequential cache miss bound is Q, and a tall cache is assumed. Always, the new bound for a general scheduler matches or improves the bound in [1, 20] and matches the bound in [15]; all of these prior bounds held only for work stealing. For Strassen, $\lambda = \log_2 7$ and $\gamma = (\lambda/2) - 1$ [13, 19].

algorithm with depth n (Depth-n-MM), the Gaussian Elimination paradigm (IGEP) [11], and stencil computations. Further, even for these algorithms, the results in [20] apply only if parent stealing is used (i.e., if the node forking the two parallel tasks is placed on the task queue, as is the case in Intel CilkPlus). In [15] an example is given where the result in [20] for Depth-n-MM does not hold under child stealing (where the right child of the forking node is placed on the task queue, as in Intel TBB and Microsoft PPL).

The bounds in [20] were matched and also extended to a more general class of HBP computations for RWS under child stealing in [15]. The methodology in [15] is to charge the cost of the cache miss overhead to $O(S)$ disjoint tasks in the sequential computation, where each task is an HBP sub-computation, and then to bound the cost of the worst-case configuration of such a collection of $O(S)$ disjoint tasks.

These prior results were reported only for RWS, but the analysis holds for any work stealing scheduler. Work stealing is a natural and effective method for scheduling multithreaded algorithms, and is implemented in CilkPlus, TBB and PPL, as noted above. A key feature of work-stealing is that the task that an idle processor steals (i.e., moves) from another processor is the one at the head of the other processor's task queue. In other words, tasks are stolen from the task queue at any given processor in FIFO order. However, a multithreaded algorithm may be scheduled in environments where a work stealing scheduler is not available. In such a case, the system scheduler will be used to schedule the parallel tasks and this scheduler may not necessarily schedule tasks in FIFO order. For instance, SJF (Shortest Job First) is a commonly used scheduling policy, and this policy need not be FIFO at each processor. The Linux scheduler uses the Completely Fair Scheduler, and it is not clear if that scheduler uses the FIFO needed for work stealing.

Another reason for considering a general scheduler is to obtain 'oblivious' results as in sequential cache-oblivious algorithms [19], network-oblivious algorithms for distributed memory [5], and multicore-oblivious [12] and resource-oblivious [14, 17] algorithms for shared memory multicores. In all of these cases the desire is to have algorithms analyzed in a machine-independent manner so that bounds hold across diverse platforms. In that spirit our results give scheduler-independent results that extend across all types of schedulers as long as there is no preemption, duplication of tasks, or failures.

Further, one could consider future scenarios where new criteria such as power consumption may dictate the need for new types of schedulers. Our results show that there is not much degradation in the caching performance as a function of the number of parallel tasks scheduled even if such schedulers do not steal from the top of the dequeue.

If the scheduler does not steal in FIFO order, then the analysis used to derive the earlier results for caching overhead when using RWS is not valid. Thus, new techniques need to be developed in order to analyze caching costs with a general scheduler. This is the topic of this paper.

In this paper, we show that for a general class of multithreaded algorithms, including all those with series-parallel fork-join calls, the cache miss excess remains bounded by $O(Q + S \cdot M/B)$, and that for a class of well-structured HBP algorithms (including those listed in Table 1), the cache miss excess is bounded by the best bound currently known for work stealing schedulers.

We are able to achieve good bounds even when considering the worst-case effects of 'false-sharing' (fs misses) as long as we use the algorithms with the small modifications given in [14]; we omit discussing this in this extended abstract.

1.2 Overview of Our Results

We assume a tall cache ($M \geq B^2$), and we assume that a sequential execution that accesses r data items accesses $O((r/B) + \sqrt{r})$ blocks (see Sections 6.3, 6.4). Our main results are Theorems 2.1 and 2.5 in Section 2, and Table 1 lists the bounds we obtain for some well-known algorithms by applying these two theorems. All of these algorithms are well-known parallel multithreaded algorithms, and all have excellent sequential cache-oblivious caching bounds.

Consider a computation whose parallel execution incurs S steals. Previous analyses for the cache miss overhead all took the following approach: the sequential execution was partitioned into $O(S)$ consecutive pieces or fragments, which we call *task kernels*, with the property that in the parallel execution each task kernel was executed on a single processor. Then the analyses amounted to bounding the amount of data a task kernel uses that was used by

an earlier task kernel in the sequential execution and which could have been available in the cache; this upper bounds the additional reloads due to steals. However, with a general scheduler, a partitioning with these properties in not possible in general, as we show in Example 4.3 in Section 4.2. Nonetheless, we are able to recover the simple $O(Q + S \cdot M/B)$ bound on $C(S)$, the number of cache misses with S steals. Further, with a more sophisticated analysis we achieve the results in Bound B in Table 1, bounds that match the earlier results in [15] which hold only for RWS, and which can be a strict improvement (depending on the value of S) over the $O(Q + S \cdot M/B)$ bound, as shown in Section 2.1 for FFT and SPMS sorting.

At a high level, our approach to establishing our bounds is similar to the one used in [15] for work stealing schedulers. It bounds the caching overhead for an HBP computation incurring S steals as being no more than the cost of reloading the cache for $O(S)$ HBP tasks in the computation. The final bound is obtained by considering the worst-case cost for a collection of $O(S)$ HBP tasks in the computation. However, within this high level approach, our current method differs from the one in [15], as described below.

In [15], the $O(S)$ tasks were required to be disjoint tasks (as was the case in [1, 20] as well), and this resulted in several different case analyses for different types of HBP computations. It also required rather strong balance conditions for the sizes of sibling recursive tasks because costs were being allocated from a steal-incurring subtask to a steal-free sibling. In our current analysis, we allow these $O(S)$ distinct HBP tasks to overlap, and we allocate the costs to the steal-incurring task itself. This allows us to unify the analysis for all HBP computations into a single argument.

Organization of the Paper. In Section 2 we state our two main theorems, and we describe the concrete results we obtain from our second theorem for specific algorithms. Section 3 gives basic background on work stealing and scheduling parallel tasks, and Section 4 describes our set-up for general schedulers. In Section 5 we define *task kernels* and give a proof of our first main theorem (Theorem 2.1). Finally, in Section 6 we present our refined analysis for BP and HBP computations, and establish our second main theorem. Some of the details and proofs are deferred to the full paper [16].

2 OUR MAIN THEOREMS

We consider a shared memory parallel environment comprising p processors, each with a private cache of size M. The p processors communicate through an arbitrarily large shared memory. Data is organized in blocks (or 'cache lines') of size B.

We will express parallelism through paired fork and join operations. A fork spawns two tasks that can execute in parallel. Its corresponding join is a synchronization point: both of the spawned tasks must complete before the computation can proceed beyond this join. For an overview of this model, see Chapter 27 in [18].

Our first theorem applies to the cache miss overhead under the scheduling of any series-parallel computation dag by a general scheduler, and it generalizes an earlier result in [1] that held only for RWS.

Theorem 2.1. *Let $\mathcal{A}$ be a series-parallel algorithm and suppose it incurs S steals in a parallel execution using a general scheduler. Then the cache miss cost of this execution is $C(S) = O(Q + S \cdot M/B)$, where Q is the number of cache misses incurred by $\mathcal{A}$ in a sequential execution.*

Our second theorem improves on the above theorem for the following class of HBP algorithms, based on [13, 15]. Here, given a task τ, its *size*, $|\tau|$, is the number of distinct data items read or written by τ; this excludes any local variables declared by τ. A *balanced fork-join computation* consists of a fork tree followed by a join tree on a common set of leaves, where the sizes of the tasks decrease geometrically from parent to child in the fork tree.

Definition 2.2. **(HBP task)** A BP algorithm (or task) is a balanced binary fork-join computation on n leaves, where each fork, join and leaf node performs $O(1)$ computation.

A Type 1 HBP task comprises a sequence of $O(1)$ BP tasks.

A Type k HBP task, for $k \geq 2$, comprises a sequence of $O(1)$ *constituent* tasks. Each constituent task is either a BP task, a Type $h < k$ HBP task, or a *recursive constituent*, which is an ordered collection of one or more recursive instances of the Type k task. Each such ordered collection is initiated by a binary fork tree and ended by a complementary join tree.

In addition, certain requirements apply to data layout and data accesses as described in Section 6.2.

In order to bound the additional cache misses incurred due to steals, we now define $x(\tau)$, the *extended size* of τ, as follows.

Definition 2.3. **(Extended size)** Let τ be a task that calls $\tau_1, \tau_2, \ldots, \tau_l$, where each τ_i is a BP constituent task or an individual recursive task forked by a recursive constituent of τ. Then, τ's extended size $x(\tau)$ is given by $x(\tau) = |\tau| + \sum_{i=1}^{l} |\tau_i|$,

The extended size of a task τ incorporates τ's size, $|\tau|$, together with the sizes of individual tasks in its constituent tasks. The additional term over $|\tau|$ is the sum of the sizes of the tasks called by τ. This is done in order to account for the fact that a stolen sub-task of τ may need to read again some of this data, and τ can have several stolen sub-tasks. Also, there may be overlap in the data accessed by different tasks called recursively by τ. In general, in the extended size of τ the individual sizes of the tasks called by τ are added to the size of τ, possibly resulting in a value much larger than $|\tau|$. However, for all the algorithms we consider (see Table 1), the value of $x(\tau)$ remains $O(|\tau|)$.

We now state the constraints we impose on the algorithms we consider. To achieve the strongest cache miss bounds we need the algorithm to be cache-compliant, as defined next.

Definition 2.4. **(Cache-compliant task)** An HBP task $\mathcal{A}$ is *cache-compliant* if for each recursive task τ in $\mathcal{A}$ and for each recursive call τ' made by τ, there is a constant $\alpha < 1$ such that

- $|\tau'| \leq \alpha|\tau|$,
- $x(\tau') \leq \alpha \cdot x(\tau)$, and
- τ makes $O(|\tau|)$ recursive calls.

All the algorithms we consider are cache-compliant.

Our second theorem, given below, provides a refined bound for $C(S)$ for cache-compliant HBP algorithms (Table 1 gives a tighter

result for Scan and Prefix Sums that does not follow directly from Theorem 2.5; that result is shown in Section 6.3).

Theorem 2.5. *Suppose a cache-compliant Type k HBP algorithm $\mathcal{A}$ incurs S steals when executed using a general scheduler, and suppose that in a sequential execution $\mathcal{A}$ incurs Q cache misses.*
(i) If $k = 1$ then $C(S) = O(Q + S \cdot B)$.
(ii) If $k \geq 2$ then there is a collection $\tau_1, \tau_2, \ldots, \tau_l$ of distinct recursive tasks, with $l = O(S)$, where each of the τ_i is an h-HBP task for some $2 \leq h \leq k$, including possibly the whole computation, such that the cost, $C(S)$, of the cache misses incurred by this execution of $\mathcal{A}$ is bounded by

$$C(S) = O\left(Q + (\sum_{i=1}^{l} x(\tau_i)/B) + S \cdot B\right).$$

With the above bound in hand, it will suffice to bound $\sum_i x(\tau_i)/B$, where the sum is over all l tasks specified in Theorem 2.5. The result is a bound on $C(S)$ that is never worse than the earlier bound of $O(Q+S\cdot M/B)$, and in some cases improves on it, and which applies not only to work stealing schedulers but also to general schedulers.

As in [1, 8], we can incorporate the above bound for the overall cache miss cost $C(S)$ for any scheduler that steals S tasks into a bound on the overall time for the parallel execution as follows. Let b be the cost of a cache miss, and s the cost of a steal, i.e., s is the time taken by the scheduler to transfer a parallel task from its original processor to another processor that will execute it in parallel. Let T_1 be the sequential execution time for the computation, let T_∞ be the span (or critical path length) of the parallel computation, and let I be the total time spent by processors idling while not computing, stealing, or waiting on a cache miss. Then the time taken by this parallel execution is given by:

$$T_p = \frac{1}{p}\left(T_1 + b \cdot C(S) + s \cdot S + I\right) + b \cdot T_\infty.$$

In the above equation, S and I depend on the scheduler: A well-designed scheduler would steal as few tasks as it can while keeping all processors engaged in computation. Our contribution in this paper is to obtain a good bound for cache-compliant HBP computations for the term $C(S)$ in the above equation.

At a high level, our analysis proceeds as follows. It identifies $O(S)$ 'special' recursive tasks, some of which may be nested one in another, and assigns to these special tasks all the cache-miss costs apart from the sequential execution cost. In addition, each steal will be assigned to a special task (this task will be said to *own* the steal). Let τ_i be one of these special tasks and suppose it owns S_i steals; then the costs assigned to τ_i will be bounded by $O(x(\tau_i)/B + S_i \cdot B)$ as we will see later.

2.1 Analysis of Specific Algorithms

We apply Theorem 2.5 to several well-known algorithms, to obtain the results for bound B in Table 1. The GEP and LCS algorithms are presented in [10, 17], while the others are described in [14] (where their false sharing costs are analyzed). Here we obtain bound B for a couple of entries listed in Table 1. For the remaining entries in the table see [16].

$\log^2 n$-MM. This is a Type 2 HBP that has one recursive constituent that makes 8 recursive calls to $n/2 \times n/2$ matrices, and a BP task that adds up the outputs of the recursive calls in pairs. Its sequential cache complexity is $O(n^3/(B\sqrt{M}))$. Applying Theorem 2.5 we see that the l largest HBP tasks are obtained by including all recursive tasks up to $j = (1/3)\log l$ levels of recursion. The sum of the sizes of these tasks is $O((n^2/4^j) \cdot 8^{(1/3)\log l}) = O(n^2 \cdot l^{1/3})$. Since $l = O(S)$, we obtain the overall cache miss cost with S steals as $O(Q + (n^2/B) \cdot S^{1/3} + S \cdot B)$.

FFT, SPMS Sort. The algorithms for both FFT [19] and SPMS sort [17] have the same structure, being Type 2 HBP algorithms that recursively call two collections of $O(\sqrt{n})$ parallel tasks of size $O(\sqrt{n})$, together with a constant number of calls to BP computations.

To bound $\sum_{i=1}^{O(S)} |v_i|/B$ for FFT, we observe that the total size of tasks of size r or larger is $O(n \log_r n)$, and there are $\Theta(\frac{n}{r} \log_r n)$ such tasks. Choosing r so that $S = \Theta(\frac{n}{r} \log_r n)$ we obtain $r \log r = \Theta(n \log n/S)$, so $\log r = \Theta(\log([n \log n/S])$. Thus $\max_C \sum_{v_i \in C} \frac{|v_i|}{B} = O(\frac{n}{B} \log_r n) = O(\frac{n}{B} \frac{\log n}{\log[(n \log n)/S]})$.

The analysis for SPMS is very similar, except that we need to handle two BP computations with somewhat irregular access patterns.

Observation. For FFT and SPMS, our refined bound is strictly better than the $O(Q + S \cdot M/B)$ bound since our overhead remains $O(Q)$ when $M^\epsilon = O((n \log n)/S$ for any constant $\epsilon > 0$, while the simple bound needs $M \log M = O((n \log n)/S)$ for Q to dominate $S \cdot M/B$.

3 SCHEDULING PARALLEL TASKS

The *computation dag* for a computation on a given input is the acyclic graph that results when we have a vertex (or node) for each unit (or constant) time computation, and a directed edge from a vertex u to a vertex v if vertex v can begin its computation immediately after u and v's other predecessors complete their computations, but not before. Since we consider multithreaded algorithms with binary forking, where the fork-joins are nested, the computation dag is a series-parallel graph.

During the execution of a computation dag on a given input, a parallel task is created each time a fork step f is executed. At this point the main computation proceeds with the left child of f, as in a standard sequential dfs computation, while the task τ at the right child r of the fork node is made available to be scheduled in parallel with the main computation. (This is *child stealing*, or the *help-first policy* [21]; one could also use the work-first policy where the main computation proceeds with the task spawned at the right child.) The parallel task τ consists of all of the computation starting at r and ending at the step before the join corresponding to the fork step f. A run-time scheduler determines if a forked task is to be moved to another processor for execution in parallel.

3.1 Caching Overhead Under Work-stealing

An important class of schedulers is the work-stealing scheduler. Each processor maintains a task queue on which it enqueues the parallel tasks it generates. When a processor is idle it attempts to steal, i.e., obtain a parallel task from the head of the task queue of another processor. The exact method for identifying the processor from which to obtain an available parallel task determines the type

of work-stealing scheduler being used; however the stolen task is always the task at the head of the task queue of the chosen processor. The most popular type is randomized work-stealing (RWS, see e.g., [8]), where a processor picks a random processor and steals the task at the head of its task queue, if there is one. Otherwise, it continues to pick random processors and tries to find an available parallel task until it succeeds, or the computation completes. RWS has been widely analyzed and used, notably in Cilk. The following is a well-known fact about work stealing schedulers (see Figure 1).

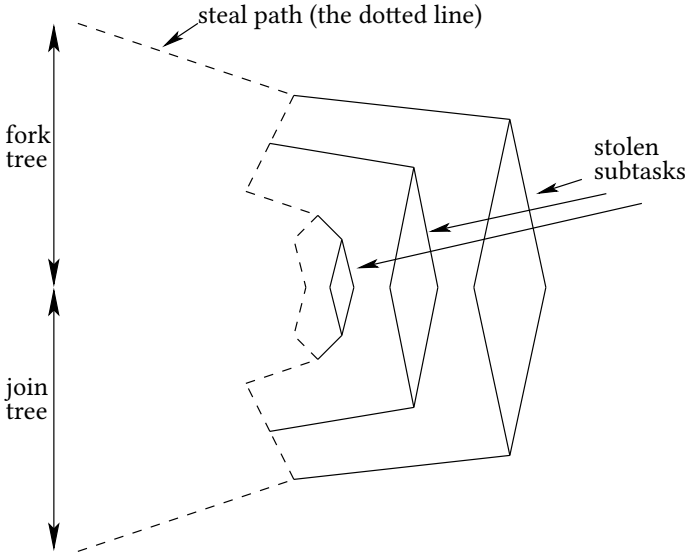

Figure 1: A steal path for a work-stealing scheduler with three stolen subtasks of a BP task.

Fact 1. **The Steal Path for Work-stealing Schedulers).** *Let τ be either the original task or a stolen subtask. Suppose that τ incurs steals of subtasks $\tau_1, \cdots, \tau_k$. Then there exists a path P_τ in τ's computation dag from its root to its final node such that the parent of every stolen task τ_i lies on P_τ, and every off-path right child of a fork node on P is the start node for a stolen subtask.*

We now review a well-known bound for RWS in Acar et al. [1] on $R(S)$, the caching overhead (over and above the sequential caching cost) in a parallel execution that incurs S steals. We will assume an optimal offline cache replacement policy (as in the sequential case). Let σ be the sequence of steps executed in a sequential execution. Now consider a parallel execution that incurs S steals. Partition σ into contiguous portions so that in the parallel execution each portion is executed in its sequential order on a single processor. Then, each processor can regain the state of the cache in the sequential execution once it has accessed M/B distinct blocks during its execution. Thus if there are K portions, then there will be $R(S) = O(K \cdot M/B)$ additional cache misses. (Actually, the justification of this claim requires the parallel analogue of the regularity assumption formulated in [19] to make it complete, see [17].)

It is shown in [1] that $K \leq 2S + 1$ for a work-stealing scheduler. This is readily seen from Fact 1. A steal creates three fragments within the sequential computation — (1) the sequential computation up to when the stolen task would start its computation, (2) the computation of the stolen task, which will occur on a different processor, and (3) the sequential computation following the stolen task. Each of these three fragments is computed sequentially. Each successive steal creates two additional fragments leading to the bound of $K \leq 2S + 1$ sequential fragments for a work stealing scheduler, and a bound of $R(S) = O(K \cdot M/B)$ additional cache misses. This implies $C(S) = O(Q + S \cdot (M/B))$.

4 GENERAL SCHEDULERS

In this paper, we consider the cache miss overhead for a general scheduler that is not necessarily work stealing. We will assume that there is no redundancy in the computation, and that each node in the computation dag is executed at exactly one processor. We will view the general scheduler as being similar to a work stealing scheduler, except that the task stolen from the chosen processor can be an arbitrary parallel task available for computation, not necessarily the task at the head of its task queue.

When a task other than the topmost task on a task queue is stolen, we call this a *deep* steal. More precisely, we have the following definition.

Definition 4.1. **(Deep steal)** Let σ be a task stolen from the task queue, Π, of τ, and suppose that σ is not the first task placed on Π. Let σ' be the task placed on Π immediately before σ. Then, this steal of σ is a *deep steal* if σ' is not stolen from Π.

In order to essentially maintain Fact 1 (the Steal Path Fact) we will treat all the tasks ahead of σ on the task queue Π as if they were 'pseudo-stolen' as per the following definition.

Definition 4.2. **(Pseudo-stolen task)** Consider the task queue Π for a task τ, and let σ be stolen from Π as a deep steal. Any task that was placed on Π before σ and which remains unstolen is a *pseudo-stolen* task.

We observe that in a computation that incurs deep steals, the steal path will contain not only the parent of every stolen task but also the parent of every pseudo-stolen task.

4.1 Execution Stacks

In order to obtain a tighter bound on the additional costs due to steals, we now take a closer look at how one stores variables that are generated during the execution of the algorithm. It is natural for the original task and each stolen task to each have an execution stack on which they store the variables declared by their residual task. But as we will see, in some circumstances, we may need more execution stacks. E_τ will denote the execution stack for a task τ if it has one. As is standard, each procedure and each fork node stores the variables it declares in a segment on the current top of the execution stack for the task to which it belongs, following the usual mode for a procedural language.

Execution Stack and Task Queue. The parallel tasks for a processor P are enqueued on its task queue in the order in which the segments for their parent fork nodes are created on P's execution stack. The task queue is a double-end queue, and P will remove an enqueued task σ from its task queue when it begins computing on σ's segment.

As noted in Section 3, work stealing is a popular scheduling strategy where a task that is stolen (i.e., transferred to another processor) is always the one that is at the head of the task queue in the processor from which it is stolen. However, with a general scheduler, an arbitrary task on the task queue can be stolen.

Cache Misses when Accessing an Execution Stack. Suppose that a subtask τ' is stolen from a task τ. Consider the join node v immediately following the node at which the computation of τ' terminates. Let P be the processor executing $\tau - \tau'$ when it reaches node v and let P' be the processor executing τ' at this point. To avoid unnecessary waiting, whichever processor (of P and P') reaches v second is the one that continues executing the remainder of τ. If this processor is P', we say that P' has *usurped* the computation of τ. The effect, in terms of cache misses, is that in order to access variables on E_τ, P' will incur cache misses that P might not. Even if P' does not usurp τ, P may have to read additional data when continuing the execution of τ beyond τ' (due to its having been first read by the stolen subtask). Our analysis of cache miss overhead in a parallel execution will use a single method to cover the costs in both cases. This analysis assumes that no data is in cache at the start of the execution of τ beyond τ', whether P or P' is performing this execution, and hence can only overestimate the necessary reloads of data.

Execution Stacks for a General Scheduler. We observe that when using a general scheduler, additional execution stacks may be needed. For suppose that processor P is executing a task τ from which a deep steal of subtask σ occurs. Let P' be the processor executing σ. Suppose that P is the first of P and P' to reach the join node at which the steal ends. Then P will leave the continuation of the execution of τ to P'. But P still needs to execute the parallel tasks remaining on its task queue, performing them in dfs order (i.e. from the rear of the queue). However, P cannot use the execution stack for τ to store the variables for these subtasks as (a) this would violate the standard practice in which the current variable order on the execution stack corresponds to the current path of open procedure calls, and (b) additional space on this stack may be needed for the execution by P' of the portion of τ following the join node. In these circumstances P will create a new execution stack for each such pseudo-stolen task on its task queue as and when it starts its execution. This is exactly what would happen were the task to be stolen. The difference is that this will not count as a steal. In fact, whether P reaches the join node first or not, it will need to execute the tasks remaining on its task queue, and it will use a new execution stack for each such pseudo-stolen task. P will continue to have a single task queue, however.

4.2 Caching Overhead for General Schedulers

We now give an example of an execution where a task could be fragmented into a sequence of several non-contiguous fragments of execution due to a single steal by a general scheduler, and hence the analysis in [1] for the cache miss excess bound of $R(S) = O(S \cdot M/B)$, which we saw earlier for work-stealing schedulers, does not immediately hold.

Example 4.3. See Figure 2. Let τ be a balanced fork-join task with n leaves, with unit-cost computation at each node. Suppose τ incurs one steal of a subtask τ_1, where the start of τ_1 is reached by traversing a path $\mathcal{P}$ of $k = (\frac{1}{2}\log n) - 1$ left child links followed by one right child link. Let $\mu_1, \mu_2, \ldots, \mu_k$ be the right subtasks of path $\mathcal{P}$, from top to bottom, preceding τ_1. Note that each of the tasks $\mu_1, \mu_2, \ldots, \mu_k$ is a pseudo-stolen task. Let $\overline{\mathcal{P}}$ be the path in the join tree complementary to $\mathcal{P}$ and suppose it comprises nodes $v_{k+1}, v_k, v_{k-1}, \ldots, v_1$ from bottom to top (v_1 is the root of the join tree and v_{k+1} is the join node that is the parent of the final node in the stolen subtask).

Let P be the processor executing τ initially and let P_1 be the processor executing stolen task τ_1. Suppose the timing is such that P_1 executes all the nodes on $\overline{\mathcal{P}}$. Then, P executes $k = \Theta(\log n)$ non-contiguous fragments of sequential computation, one for each μ_i. Likewise P_1 executes each of the v_i in turn, and these are also non-contiguous. Since $\Theta(\log n)$ fragments are created by one steal, the simple argument providing the $O(M/B)$ cache miss overhead per steal will not apply to general schedulers.

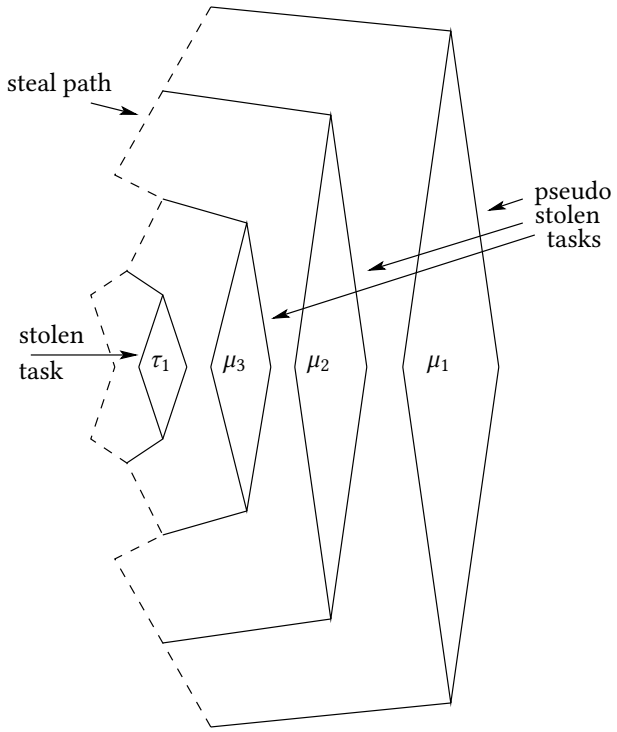

Figure 2: Illustrating Example 4.3. Here there are four stealable tasks to the right of the steal path, $\mu_1, \mu_2, \mu_3, \tau_1$. With a general scheduler, if the lowest such subtask, τ_1, were the only one that was stolen, this would be a deep steal inducing a pseudo task kernel consisting of the three other stealable tasks but not the portions of the steal path connecting them.

In the next section we recover the $R(S) = O(S \cdot M/B)$ bound for a general scheduler (for all series parallel computation dags) using a different analysis, and in Section 6 we present a further refined analysis for HBP algorithms.

5 THE NEW CACHE MISS ANALYSIS

We begin by specifying the partitioning of the computation into *task kernels* in Section 5.1. We follow this by demonstrating in Section 5.2 the simple $O(Q + S \cdot M/B)$ bound on the cache miss cost. We then outline our more sophisticated bound, analyzing in turn BP computations in Section 6.3 and HBP computations in Section 6.4.

5.1 Tasks and Task Kernels

Let τ be a task that incurs steals. Informally, a task kernel of τ is a maximal contiguous(or essentially contiguous) fragment of the computation that lies entirely within the unstolen portion of τ or entirely within a stolen task.

We now give the definition of the task kernels induced by the steals in a computation. This definition applies to any computation dag in which any given pair of forks and joins is either nested or disjoint, as for example in series-parallel dags, and this includes all HBP computations. Figure 3 gives examples of task kernels in a BP computation.

Definition 5.1. **(Task kernels.)** Consider a parallel execution of a computation C under a general scheduler, and suppose it incurs S steals, $\sigma_1, \sigma_2, \cdots, \sigma_S$, numbered in the order in which the stolen tasks are generated (i.e., the order in which the parent fork nodes are executed) in a sequential execution. In turn, we partition the computation dag into task kernels with respect to the sequence $\Sigma_i = \langle \sigma_1, \sigma_2, \ldots, \sigma_i \rangle$ to create the collection C_i. We let Σ_0 be the empty sequence and its associated partition C_0 be the single task kernel containing the entire computation dag. For each $i \geq 1$ the partition C_{i+1} is obtained from C_i as follows. Let τ be the task kernel in C_i that contains the fork node v_f at which steal σ_{i+1} is performed, and let v_j be the corresponding join node. Then, τ is partitioned into the following task kernels, each categorized as being of type *starting, finishing,* or *pseudo.* The initial task kernel in C_0 is given type starting.

1. τ_1, the stolen subtask created by σ_{i+1}. It is called a *starting task kernel.*

2. τ_2, the portion of τ preceding the stolen subtask in the sequential execution (this includes the portion of the computation descending from the left child of v_f that precedes v_j.) It is given the same type as τ.

3. If σ_{i+1} is a deep steal, let $\nu = \langle \mu_1, \mu_2, \ldots, \mu_k \rangle$ be the sequence of pseudo-stolen tasks forked from τ that are on the task queue for the processor executing τ at the time of this steal, with μ_k immediately preceding the subtask stolen by σ_{i+1}. Suppose that $\mu_{i_1}, \mu_{i_2}, \ldots, \mu_{i_j}$ (for $i_1 < i_2 < \cdots$) incur steals (note that any such steal would occur after σ_{i+1} in our ordering of steals). Then each of the collections $(\mu_1, \mu_2, \ldots, \mu_{i_1-1})$, $(\mu_{i_1}, \mu_{i_1+1}, \ldots, \mu_{i_2-1})$, ..., $(\mu_{i_j}, \mu_{i_j+1}, \ldots, \mu_k)$ forms a *pseudo task kernel.*

4. τ_3, the portion of τ starting at v_j in the sequential execution but excluding the collection ν in part 3 above. This includes the join nodes following the pseudo-stolen tasks in ν. Then, $C_{i+1} = (C_i - \{\tau\}) \cup \{\tau_1, \tau_2, \tau_3\} \cup \{\text{pseudo task kernels formed in part 3, if any}\}$.

The final collection C_S is the collection of task kernels for this parallel execution of C.

In part 3 above, each pseudo task kernel comprises a maximal sequence of pseudo-stolen task kernels, which in execution order ends with a pseudo-stolen kernel that incurs a steal, with all the other pseudo-stolen task kernels in the sequence being steal-free. In part 4, the finishing task kernel μ_3 comprises the nodes descendant from v_j in the computation dag, including v_j itself. Note that in the sequential execution, both the finishing task kernel μ_3 and the pseudo task kernels in ν are executed after the stolen subtask, and they interleave in their execution. Some implications of this interleaving were explored in Example 4.3.

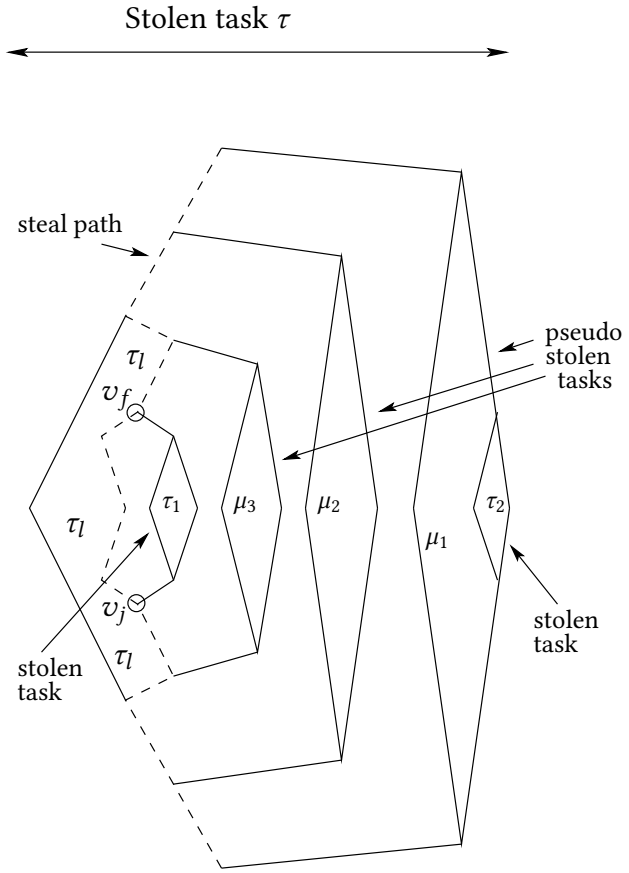

Figure 3: Task kernel types in a BP computation. Suppose a task τ undergoes a deep steal of subtask τ_1 at fork node v_f. Then τ_1 forms a starting task kernel. Also, τ_l, the portion of τ to the left of the steal path, plus the steal path up to but not including v_j, forms another starting task kernel. μ_1, μ_2, μ_3 are all pseudo stolen tasks; if μ_1 undergoes a steal but μ_2 and μ_3 do not, then $\mu_2 \cup \mu_3$ form a pseudo task kernel. Finally, the portion of the path of join nodes descending from v_j, and including v_j forms a finishing task kernel. (A finishing task kernel can incur a steal only in a Type k HBP, for $k > 1$.)

Lemma 5.2. *A series parallel computation with S steals has at most $4S + 1$ task kernels, of which at most $S + 1$ are starting kernels, S are finishing, and $2S$ are pseudo.*

Proof. In the absence of a deep steal, the number of task kernels is exactly $2S + 1$, since there is initially one task kernel, and each

successive steal replaces a current task kernel with three new ones, according to parts 1, 2 and 4 in Definition 5.1, one finishing, one starting, and one of the previous type. This yields at most $S + 1$ starting and S finishing task kernels.

Now, let us consider the effect of part 3 in Definition 5.1, which creates $i_j + 1$ pseudo task kernels when i_j of the pseudo stolen tasks in v incur steals. We claim that we can bound the number of pseudo task kernels by $2S$ by charging at most two of them to each steal as follows. To each deep steal σ we assign the last pseudo task kernel $(\mu_{i_j+1}, \mu_{i_j+2}, \ldots, \mu_k)$ in its collection v (as defined in part 3 of Definition 5.1). We assign each of the remaining pseudo task kernels for σ to the earliest steal σ' (in our ordering) in the steal-incurring pseudo-stolen task kernel. Now consider σ'. It may be assigned another pseudo task kernel if it is itself a deep steal (in a different state of the execution stack). So σ' could be assigned two different pseudo task kernels. But it cannot be assigned a third one, since for any pseudo stolen task that contains v, the earliest steal in it is either σ or a steal earlier than σ. Thus, there are at most $2S$ additional task kernels created due to the pseudo task kernels, and this adds up to a total of at most $4S + 1$ task kernels. □

5.2 Proof of Theorem 1

Recall Example 4.3. Observe that the sequence $\langle \mu_k, v_k, \mu_{k-1}, v_{k-1}, \ldots, \mu_1, v_1 \rangle$ is contiguous. Thus each of P_1 and P executes a portion of the same contiguous sequence, and between them they execute all of it. Therefore their combined caching overhead is at most twice the sequential cost plus an additional M/B term for each of them. The proof of Theorem 2.1 will build on this insight to establish that in fact $C(S) = O(Q + S \cdot M/B)$ under a general scheduler for the entire class of series-parallel dags.

Proof of Theorem 2.1. For the purposes of this proof, we further refine the partitioning into task kernels as follows. Let μ be a pseudo task kernel that ends in a steal-incurring pseudo-stolen task. Let v_t be the terminal node in μ, i.e., the final node in μ in a sequential execution. Let v_j be the node following v_t in a sequential execution; then v_j is a join node and it lies in a finishing task kernel, which we will denote by v^μ. We split v^μ in two, where the initial portion v_1^μ contains the portion of v^μ up to, but not including v_j, and the latter portion v_2^μ contains the portion of v^μ starting at v_j. We then merge μ with v_1^μ to form a *super-finishing task kernel* μ-v (and we discard μ and v_1^μ). We repeatedly perform this split and merge into super-finishing task kernels at each steal-incurring pseudo task kernel.

The above process partitions the computation into at most $4S + 1$ task kernels, some of which may be super-finishing task kernels. Each of these task kernels has the useful property that it is executed contiguously in a sequential execution. Furthermore, at most two processors are used to execute each super-finishing task kernel, namely the processor starting the corresponding finishing kernel and the processor completing the corresponding pseudo task kernel. The other task kernels are all executed by a single processor.

Thus, as in the work-stealing case, for each of these $4S + 1$ kernels there is a cost of $O(M/B)$ cache misses to restore the state that existed in the sequential execution, and as the execution of each of the at most S super-finishing kernels is shared among two processors, this at most doubles the cache miss cost of these portions of the computation, leading to a bound of at most $Q + (5S + 1) \cdot M/B$ additional cache misses due to the steals. This establishes the desired bound. □

6 THE HBP ANALYSIS

In this section we present an improved bound on the caching overhead of HBP algorithms under a general scheduler. Our approach to improving the bound in Theorem 2.1 is to carefully examine the features of HBP algorithms and tailor our analysis to algorithms in this class.

6.1 Reload Cost

We now define the notion of the reload cost of a sequence of steps executed within a sequential execution of a task τ.

Definition 6.1. **(Reload Cost)** Let μ be a sequence of steps within a task τ that are executed contiguously in a sequential execution of τ. The *reload cost of* μ is the number of distinct blocks accessed by μ during its execution, excluding blocks that contain variables declared during μ's computation.

In our analysis we will use the reload cost in place of the simple upper bound of M/B for the additional cache miss cost in executing a stolen task, or any task kernel that consists of the steps executed contiguously in a sequential execution (typically starting task kernels). We will use the following lemma in our analysis.

Lemma 6.2. *Let μ be a sequence of step within a task τ that are executed contiguously in a sequential execution of τ. Let Q be an upper bound on the number of cache misses incurred by τ during its execution of μ in a sequential execution. If μ is executed as a separate computation, then its cache miss cost is $O(Q + R)$, where R is its reload cost.*

Proof. Let us consider the additional cache miss cost in a separate execution of μ for reading in data that may have already resided in cache in an execution of μ within an execution of τ. Let us refer to the variables accessed by the execution of μ excluding variables declared during μ's computation as *new variables*, and the R blocks in memory in which they reside as *new blocks*. The only difference between a separate execution of μ and the execution of μ within a sequential execution of τ is that some of the R new blocks may already be in cache at the start of the latter execution, and hence the cost of reading these blocks is not included in Q. Now consider a separate execution of μ. If μ does not evict any of the R new blocks during its separate execution, then its cache miss cost is bounded by $O(Q + R)$ since the two executions only differ in the initial presence of these R blocks. On the other hand, any new block evicted by μ in its separate execution must also be evicted by τ in its execution of μ since both perform the same computation and both are assumed to use a given optimal cache replacement policy. Hence the number of cache misses in a separate execution of μ is bounded by $O(Q + R)$. □

The above proof does not address the 'block misalignment' cost [17] that arises from the fact that the block boundaries for data on an execution stack may be different in a parallel execution of a task from what they would be in a sequential execution. However, it

is shown in [17] that its effect is bounded by a constant factor for HBP computations, if the cache miss bound is polynomial in M and B[1], so the bound in the above lemma holds even when accounting for block misalignment costs. In fact, this observation extends to the full analysis of the HPB algorithms in this paper.

6.2 Data Layout and Caching Costs

The caching cost of a computation, even in the sequential context, is highly dependent on the data layout and the pattern of accesses to the data during the computation. Since we are bounding the caching overhead for a class of algorithms, rather than for a specific algorithm, we now specify the type of data layouts that we allow in the algorithms we analyze, and the data access patterns. We focus mainly on BP computations, since that is where the most of variation in data layout occurs. An HBP computation may declare shared arrays which are accessed by BP computations within its recursive computations.

A BP computation will access access variables placed on the execution stack by its fork and join nodes as well as the shared data structures declared at the start of the computation. Recall that each BP node performs $O(1)$ computation. Here are the types of accesses we allow in the algorithms for which we bound the cache miss overhead.

Accesses to the Execution Stack. A BP or HBP node can access its $O(1)$ data, and it can also access data declared by its parent node.

Accesses to Shared Data. Consider a shared array where each data item is associated with a single node in the BP tree. Depending on the algorithm, a node may access just its associated data, or its data plus the data for some or all of its neighbors (children and parent). We will assume that the data in this array is laid out contiguously according to an *inorder* traversal of the BP fork tree. Our results will go through if preordering or postordering is used instead of the inorder traversal we assume. We choose to use inorder traversal because it aids in obtaining our false sharing results (false sharing is mentioned in the introduction, but is not included here).

Recall that we refer to the computation dag as a task, and any parallel task spawned by a fork node is also a task. We now define the notion of an extended task. Here we use the convention that a fork tree includes the non-fork leaf nodes that lie between it and the complementary join tree.

Definition 6.3. **(Extended task)** An *extended task* in a BP computation is any sequence of $r > 1$ consecutive nodes in the inorder traversal of its fork tree together with the complementary join nodes.

Any task is clearly also an extended task. From the definition of a starting task kernel, we can see that it is basically an extended task, except that some of the nodes on its steal path may not lie within this extended task kernel. Our cache miss analysis will separately analyze the costs due to the portion of the starting task kernel that forms an extended task, and the portion outside of this extended task.

We now define the *data dispersal* function $f(r)$ (previously called the cache friendliness function in [15]), which parameterizes the cost of accesses to the shared data structures.

[1]The polynomial dependence on M and B is implicit in the earlier work.

Definition 6.4. **(Data dispersal function $f(r)$ for BP computations)** A collection of r words is $f(r)$-dispersed if it is contained in $(r/B) + f(r)$ blocks. An extended task is $f(r)$-dispersed if the data its nodes access when executed is contained in $(r/B) + f(r)$ blocks. A BP computation is $f(r)$-dispersed if every extended task in it is $f(r)$-dispersed.

Notions similar to our use of $f(r)$ have been used in sequential caching analyses, though our set-up is more general, and this generality is needed to obtain bounds for the general class of BP and HBP computations, as opposed to analyzing a single algorithm.

Examples of data dispersal functions for algorithms for scans, prefix sums and matrix transpose are given in the full paper [16]. The algorithms in Table 1 (except for two procedures in SPMS sorting) are all $O(\sqrt{r})$-dispersed, and the ones for scans and prefix sums can be made $O(1)$-dispersed (for which we give an improved bound in Section 6.3). As a result they satisfy our assumption in Section 1.2 that a task that accesses r words will access $O((r/B) + \sqrt{r})$ blocks. We present this more general analysis here using $f(r)$ since it allows one to fully analyze the SPMS algorithm and other algorithms with complex data access patterns.

6.3 BP and Type 1 HBP Computations

Consider a BP computation τ. We begin with a high-level description of the structure of task kernels in a BP computation, which are also illustrated in Figure 3.

Starting task kernel. We first observe that in a BP computation, a starting task kernel will consist of a zig-zag path in the fork tree (the steal path) with subtasks comprising its off-path left subtrees, together with the complementary subtrees in the join tree, but not the complementary zig-zag path in the join tree, for it forms a finishing task kernel. Each left-going segment in the fork tree zig-zag path contains the parents of stolen or pseudo-stolen tasks, and the left subtrees in each right-going segment are part of the starting task kernel, as implicitly specified in Definition 5.1.

Pseudo task kernel. A pseudo task kernel comprises a sequence of one of more pseudo-stolen tasks, where each pseudo-stolen task is itself a (smaller) BP computation which returns to a parent node on the join path of the task from which it was "stolen." This join node is not part of the pseudo-stolen task or the pseudo task kernel. The topmost pseudo-stolen task in the pseudo task kernel may have incurred steals and as a result may have the same form as for a starting task kernel. The remaining pseudo-stolen tasks, if any, are steal-free.

Finishing task kernel. In a BP computation, a finishing task kernel is simply a path in the join tree that ends at a parent of a returning stolen task, or at the root.

Accessing the Shared Data Structures. We start by analyzing the cost of accesses to the shared data structures, and we first analyze this cost for finishing kernels. As noted above, in a BP computation, each finishing task kernel is simply a path in the join tree, and these paths are disjoint. As the computation at a node may also access the data for its parent, the accesses by a finishing task kernel may overlap those by other task kernels, but only at their end nodes.

We will bound the overall cache miss cost for all finishing task kernels, by separately analyzing the cost of accesses to nodes in the topmost $\log|\tau| - \log B$ levels in the join tree, and then to the nodes in the bottom $\log B$ levels. There are only $|\tau|/B$ nodes in total in the topmost $\log|\tau| - \log B$ levels, and each causes $O(1)$ cache misses and so collectively they have a cost of no more than $O(|\tau|/B)$ cache misses; this cost could be smaller if the $O(S)$ finishing task kernel join paths traverse fewer than $\Theta(|\tau|/B)$ nodes in this portion of the join tree. There are $O(\log B)$ accesses by each finishing task kernel to nodes in the bottom $\log B$ levels of the join tree; we simply charge these $O(\log B)$ accesses to each of these task kernels, which adds up to $O(S \cdot \log B)$ cache misses across all finishing task kernels. Hence the total cost of cache misses for finishing task kernels is $O(|\tau|/B + S \cdot \log B)$.

In the case that $f(r) = O(1)$ we can improve the $O(S \cdot \log B)$ term to $O(S)$. The reason is that the nodes in the subtree of height $\log B$ comprise $O(B)$ contiguous nodes in the inorder traversal and hence the data these B nodes access is stored in $O(1 + f(1)) = O(1)$ blocks. Clearly this bound also applies to the subset of $\log B$ nodes on the path in question. Consequently the execution of all these nodes, for each path, will incur $O(1)$ cache misses, or $O(S)$ cache misses when summed over all the finishing task kernels. This yields a total cost of $O(|\tau|/B + S)$ cache misses.

For a starting task kernel, the nodes on the zig-zag path in the fork tree which have off-path left children, plus their off-path subtrees, together with the complementary subtrees, are contiguous in inorder, and hence form an extended task. However, the accesses by the nodes on the remainder of the fork tree zig-zag path, namely the nodes with off-path right subtrees, will be non-contiguous. To bound this cost, we observe that the cost of the shared data structure accesses by these nodes is no larger than the bound for the finishing task kernels, as the complement of each fork tree zig-zag path is the union of one or more finishing task kernels. Hence we add in the charged cost to the complementary finishing task kernels. Clearly, each finishing task kernel is charged at most once.

In a pseudo task kernel μ, the data accesses are to the data for the nodes in the kernel plus the data for the parents of pseudo-stolen tasks forming μ, and this is a contiguous collection of nodes in the inorder traversal, aside any additional discontinuities caused by steals from μ, if any; any steals from the pseudo task kernel μ cause the same sort of discontinuities as the steals from a starting kernel, as discussed in the previous paragraph, and are bounded by a similar charging scheme.

Aside the accesses to the data for the portions of the fork tree zig-zag path nodes specified two paragraphs above, we see that every starting and pseudo task kernel is accessing a disjoint contiguous interval of nodes in inorder, hence by Definitions 6.1 and 6.4 and Lemma 6.2, the cache miss cost for these is bounded by $O(\sum_{i=1}^{k} r_i/B + f(r_i))$, where r_i is the number of nodes in the fork tree that are also in the i-th starting or pseudo kernel, $\sum_{i=1}^{k} r_i = n$, and there are $k = O(S)$ of these kernels in total. This sum totals $O(n/B + \sum_{i=1}^{k} f(r_i))$.

We now give tight upper bounds on this term for $f(r) = O(1)$ and $f(r) = O(\sqrt{r})$. Clearly, when $f(r) = O(1)$ this term is just $O(n/B + S)$. For $f(r) = O(\sqrt{r})$, the sum $\sum_{i=1}^{k} f(r_i)$ is maximized when the r_i are all equal, and contributes the term $O(S\sqrt{n/S}) = O(\sqrt{Sn})$. When $S \le n/B^2$ this is $O(n/B)$ and when $S > n/B^2$, this is $O(S \cdot B)$. Thus it is always bounded by $O(n/B + S \cdot B)$.

Accessing the Execution Stacks. The overhead at execution stacks occurs when a stolen or pseudo-stolen task accesses the execution stack for its parent task when it returns from its computation, and when the subsequent finishing task kernel (in the case of a stolen task) has to reload the segments on the execution stack that it needs to access. This entails $O(1)$ accesses by each stolen task, $O(1)$ accesses by each pseudo-stolen task to the data segment for a distinct node in a finishing task kernel since each pseudo-stolen task is accessing its parent in the join tree, and $O(1)$ accesses by each node in each finishing task kernel, since each finishing task kernel accesses $O(1)$ data on the execution stack at each node on the join path that comprises the finishing task kernel. Since the segments for the nodes in each finishing task are consecutive on its execution stack, the cost for accessing a path of l nodes is $O(\lceil l/B \rceil)$ cache misses. Furthermore, each pseudo task kernel will be accessing a portion of a zig-zag path in the join tree, and it will be the only pseudo task kernel to access each of these nodes, aside from one finishing task kernel. Thus the cost of the accesses to the parent nodes in the join tree by each pseudo task kernel is no more that the cost for the corresponding finishing task kernel.

The total length of the paths for the finishing tasks is $O(|\tau|)$. Thus the cost of the accesses to the segments on the execution stack is bounded by $O(|\tau|/B + S)$; The $+S$ term is due to the rounding up of the term for each of the $O(S)$ paths, one path for each of the $O(S)$ finishing task kernels. Finally, the accesses by the stolen tasks add another $O(S)$ accesses to the total, which therefore sums to $O(|\tau|/B + S)$ cache misses. This bound for accesses to the execution stacks, together with the earlier bound for accesses by the task kernels, leads to the following lemma.

Lemma 6.5. *Consider a BP computation of size n. When scheduled with a general scheduler, it will incur $O((n/B) + S \cdot B)$ cache misses if $f(r) = O(\sqrt{r})$ and $O(n/B + S)$ cache misses if $f(r) = O(1)$, where S is the number of steals. In general, the number of cache misses will be bounded by $O(n/B + \max \sum_{i=1}^{3S+1} f(r_i))$ where $\sum_{i=1}^{3S+1} r_i = n$.*

Proof. The expression for the general bound arises because by Lemma 5.2, there are at most $3S + 1$ starting and pseudo kernels, each of which contributes to at most one of the $f(r_i)$ terms. □

This establishes part (i) of Theorem 2.5.

Note that it is only the accesses to the shared data structure that depend on the function $f(r)$. The bound for the accesses to the execution stacks is always $O(|\tau|/B + S)$. Other more irregular patterns of access to the shared data structure can arise and these would need to be analyzed separately.

Type 1 HBP Computations The one extra feature in a Type 1 HBP computation is that if one constituent μ of the computation incurs a steal then the finishing task kernel that emerges from μ may need to access variables previously accessed in the computation of μ (or earlier constituents, if any). But this entails $O(|\tau|/B)$ cache misses, and occurs at most once for each constituent (except the first), which is a total additional cost of $O(|\tau|/B)$ cache misses, as each Type 1 algorithm has $O(1)$ constituents by Definition 2.2 (and this is

the reason for this restriction). This leads to the bound in Table 1 for Prefix Sums. Also, it turns out that a more careful analysis shows that the second prefix sums algorithm described earlier, which had $f(r) = O(\log(r/B))$, achieves the same bound as the algorithm with $f(r) = O(1)$. We omit the details here.

6.4 Outline of the HBP Analysis

We give here an overview of the HBP analysis. The full analysis is available in [16].

We start by defining the notions of fork tree ownership and local and remote steals.

Definition 6.6. An HBP task τ owns the fork trees for its constituent tasks. A steal in τ that occurs in a fork tree it owns is a *local steal* and is owned by τ. A steal in τ that occurs in a recursive task and not in a fork tree it owns is called a *remote steal*.

Our HBP analysis will proceed as follows.

1. Local Steals. We bound the costs of local steals by a straightforward extension of our previous analysis for BP computations. For this, we first extend the definition of $f(r)$-data dispersal to HBP tasks, and then we bound the cache miss costs for starting task kernels induced by local steals owned by an HBP task τ, to obtain the following lemma (See [16] for the proof.)

Lemma 6.7. *Let τ be a recursive task owning h steals. Then the cache miss cost for executing τ is bounded by the sequential cost plus $(x(\tau)/B + h \cdot B)$, excluding the costs induced by remote steals.*

We can also bound the costs of the finishing and pseudo task kernels which are formed by local steals owned by τ and which are fully contained in τ. More precisely, a finishing or pseudo task kernel is analyzed under local steals only if both the steal at which the task kernel starts and the steal at which it ends are both local steals. Otherwise, the task kernel is considered to be formed from a remote steal, and its cache miss overhead is captured under the analysis for remote steals. The cost of a task kernel created by a remote steal will be assigned to τ only if the corresponding task kernel is completely contained in τ and is not contained in any task that is a proper descendant of τ.

2. Remote Steals. Consider a remote steal in an HBP task τ. This will be a local steal in some recursive task τ' within τ, but it may create a finishing task kernel and possibly a pseudo task kernel that contains a portion of $\tau - \tau'$. We bound the cost of remote steals in Section 6.4.1. There, we re-assign the cache miss cost of each finishing (and pseudo) task kernel ν to one or two HBP tasks, resulting in $O(S)$ different HBP tasks being charged under this scheme. For a finishing task kernel, we will see that, in contrast to BP computations, in the Type k HBP for $k \geq 2$ it can be more complex than simply a join path, and can itself incur steals.

3. Overall Analysis. In Section 6.4.2, we bound the overall cache miss costs of all local and remote steals by performing a second level of reassignment of cache miss costs:. We define certain *special* HBP tasks and we show that we can re-assign the costs that were assigned to $O(S)$ HBP tasks in Section 6.4.1 to at most $4S-1$ special tasks. We show that this leads to part (ii) of Theorem 2.5.

6.4.1 Analyzing Remote Steals. We now give an overview of our analysis for remote steals. Suppose an HBP task $\tau = \tau_1$ incurs a remote steal in one of its recursive constituents, μ. Suppose the steal occurs in subtask τ_k, where τ_i calls τ_{i+1} recursively, for $1 \leq i < k$, and $\tau_2 - \tau_3, \ldots, \tau_{k-1} - \tau_k$ are all steal-free. This sequence of τ_i starts at τ_1, either because it is the root task for the whole computation, or because $\tau_1 - \tau_2$ is steal-incurring, and we will say that τ_1 *adopts* this remote steal and the finishing task kernel ν created by this steal.

We consider the task kernels for which additional cache misses may occur due to the presence of this remote steal. As noted above, any starting kernel would be created by a steal in a fork-join tree owned by task τ_1, and hence would be created by a local steal and would be handled by the analysis for local steals. Thus the only task kernels that may incur new costs are pseudo task kernels and finishing task kernels. Our analysis will focus on the cost of finishing task kernels. The analysis of pseudo task kernels is similar.

The finishing task kernel, ν, that emerges from τ_k will execute portions of $\tau_{k-1} - \tau_k, \tau_{k-2} - \tau_{k-1}, \ldots, \tau_2 - \tau_3, \tau_1 - \tau_2$, in turn. In each $\tau_i - \tau_{i+1}$, for $i > 1$, the execution of ν starts with the traversal of a path in a join tree (the join tree complementary to the fork tree from which τ_{i+1} was forked) followed by the execution of the remaining constituent tasks in τ_i, if any. In $\tau_1 - \tau_2$, ν will traverse a path in the join tree for μ (recall that μ is the recursive constituent of τ that contains ν), and may execute other constituent tasks that follow μ in τ_1, depending on where the next steal in the computation occurs. We analyze separately the additional cache miss costs of ν's access to the join paths, and ν's execution of remaining constituent tasks in each τ_i.

In the analysis that follows, we apply a *charging* scheme. All of the cache miss costs incurred by the finishing task kernel ν that emerges from a remote steal incurred by $\tau = \tau_1$ will be distributed as charges to τ_1 and τ_2, by exploiting the features of cache-compliant HBP algorithms, including the geometric decrease in the extended sizes of successive recursive tasks.

The charging scheme for finishing task kernels that end in τ. First, let us consider the finishing task kernel ν described above, and let τ, μ, and the τ_i be as described above. We distribute the cache miss costs incurred by ν as follows.

- **C1** The cache miss costs incurred by ν in $\tau_1 - \tau_2$ are charged to τ_1.
- **C2** All of the remaining cache miss costs (i.e., the costs for the portions of ν in $\tau_i - \tau_{i+1}$, for $i > 1$) are charged to τ_2.

We perform the above distribution for all finishing task kernels that start in μ and end in τ, and we obtain the following two bounds on these charges. The proofs of these two Lemmas are in [16].

Lemma 6.8. *Let τ be an HBP task that incurs remote steals. Let the i-th constituent task of τ incur h_i steals in its fork tree, and let it incur remote steals in c_i of its collection of recursive tasks. Let $c = \sum_i c_i$ and $h = \sum_i h_i$. Across all steals, the C1 charge to τ is bounded by $O(x(\tau)/B + B + h + c)$ to τ.*

Lemma 6.9. *Let $\tau_1, \cdots, \tau_k$ be HBP tasks as defined at the start of Section 6.4.1. There is a C2 charge of $O(x(\tau_2)/B + B)$ to τ_2 if $\tau_2 - \tau_3$ is steal free (otherwise $\tau_2 = \tau_k$, and there is no C2 charge to τ_2). Across all steals, this is the only C2 charge made to τ_2.*

6.4.2 Overall HBP Analysis. In Lemmas 6.7–6.9 we bounded the additional cache miss cost of local and remote steals by charging this cost to suitable recursive tasks (or the task that starts the computation). It remains to determine how many different (possibly overlapping) recursive tasks can be charged, and the amount charged to these tasks, as a function of their extended sizes. Once we have obtained good bounds for these, we will readily obtain the desired bound in part (*ii*) of Theorem 2.5.

There are at most S recursive tasks that incur a local steal, i.e., a steal in a fork tree they own. We call these *Type 1 special tasks*. It is convenient to make the root task, the task which starts the computation, Type 1 also. There are at most a further $S-1$ tasks which have steals in two or more of the recursive subtasks they call, while having no steals in their fork trees. We call these *Type 2 special tasks*. The Type 1 and Type 2 tasks correspond to the τ_1's in the analysis in Section 6.4.1; also, local steals occur only in Type 1 tasks. The remaining charged tasks, corresponding to the τ_2 when $k > 2$, must all be a child of a Type 1 or Type 2 task and further must have a descendant of Type 1 or 2 (corresponding to the τ_k). We call these *Type 3 special tasks*. Thus there are at most $2S-1$ of these, yielding a total of $4S-1$ special tasks. Note that only the special tasks have been charged.

We now perform one more round of redistribution of costs in order to remove the c term from Lemma 6.8. This will result in a charge to each special task that depends only on its extended size and the number of local steals.

Lemma 6.10. *The charges to the special tasks for all steals can be redistributed so that a Type 1 task τ_1 that owns h steals receives a charge of $O(x(\tau_1)/B + B + h \cdot B)$, a Type 2 task τ_1' receives a charge of $O(x(\tau_1')/B + B)$ and a Type 3 task τ_2 receives a charge of $O(x(\tau_2)/B + B)$.*

Summing over all charged tasks, and noting that the number of special tasks is at most $4S-1$ (as shown above) yields a total charge of $O(\sum_{1 \le i \le O(S)} x(\tau_i) + S \cdot B)$, where the τ_i are distinct recursive or BP tasks.

Together with the analysis of the costs due to pseudo task kernels, this proves the second claim in Theorem 2.5.

Acknowledgement. We thank Charles Leiserson for extensive discussions and helpful comments. We also thank Simon Peters and Emmett Witchel for their comments on schedulers used in practice.

REFERENCES

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. 2002. The Data Locality of Work Stealing. *Theory of Computing Systems* 35, 3 (2002). Springer.
[2] D. Ajwani, N. Sitchinava, and N. Zeh. 2010. Geometric algorithms for private-cache chip multiprocessors. In *Proc. Eur. Symp. Alg. (ESA)*. 75–86.
[3] L. Arge, M. T. Goodrich, N. Nelson, and N. Sitchinava. 2008. Fundamental parallel algorithms for private-cache chip multiprocessors. In *ACM SPAA*. 197–206.
[4] L. Arge, M. T. Goodrich, and N. Sitchinava. 2010. Parallel external memory graph algorithms. In *IEEE IPDPS*.
[5] Gianfranco Bilardi, Andrea Pietracaprina, Geppino Pucci, Michele Scquizzato, and Francesco Silvestri. 2016. Network-Oblivious Algorithms. *JACM* 63 (2016), Article 3.
[6] G. Blelloch, R. A. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. 2008. Provably Good Multicore Cache Performance for Divide-and-Conquer Algorithms. In *Proc. ACM-SIAM SODA*. 501–510.
[7] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. 2010. Low depth cache-oblivious algorithms. In *Proc. ACM SPAA*. 189–199.
[8] R. Blumofe and C. E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *JACM* (1999), 720–748.
[9] R. Chowdhury, H. Le, and V. Ramachandran. 2010. Cache-oblivious dynamic programming for multicores. *IEEE/ACM Trans. Computational Biology (TCBB)* 7 (2010), 495–510.
[10] R. A. Chowdhury and V. Ramachandran. 2010. The Cache-oblivious Gaussian Elimination Paradigm: Theoretical Framework, Parallelization and Experimental Evaluation. *Theory of Computing Systems* 47, 1 (2010), 878–919.
[11] Rezaul Alam Chowdhury and Vijaya Ramachandran. 2008. Cache-efficient dynamic programming algorithms for multicores. In *Proc. ACM SPAA*. 207–216.
[12] Rezaul Alam Chowdhury, Vijaya Ramachandran, Francesco Silvestri, and Brandon Blakeley. 2013. Oblivious algorithms for multicores and network of processors. *Jour. Parallel and Distr. Computing* 23 (2013), 911–925.
[13] R. Cole and V. Ramachandran. 2011. Efficient Resource Oblivious Algorithms for Multicores. *CoRR* arXiv:1103.4071 [cs.DC] (2011).
[14] R. Cole and V. Ramachandran. 2012. Efficient Resource Oblivious Algorithms for Multicores with False Sharing. In *Proc. IEEE IPDPS*.
[15] R. Cole and V. Ramachandran. 2012. Revisiting the Cache Miss Analysis of Multithreaded Algorithms. In *Proc. LATIN'12*.
[16] R. Cole and V. Ramachandran. 2017. Bounding Cache Miss Costs of Multithreaded Computations Under General Schedulers. *CoRR* arXiv:1705.08350 [cs.DC] (2017).
[17] Richard Cole and Vijaya Ramachandran. 2017. Resource Oblivious Sorting on Multicores. *ACM Trans. on Parallel Computing (TOPC)* 3 (2017). Article 23.
[18] T. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms, Third Edition.* MIT Press.
[19] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. 2012. Cache-Oblivious Algorithms. *ACM Trans. Algor.* 4 (2012), 285–297.
[20] M. Frigo and V. Strumpen. 2009. The Cache Complexity of Multithreaded Cache Oblivious Algorithms. *Theory of Computing Systems* 45 (2009), 203–233.
[21] Y. Gao, I. Zhao, R. Barik, R. Raman, and V. Sarkar. 2009. Work-first and help-first scheduling policies for async-finish task parallelism. In *IEEE IPDPS*.
[22] N. Sitchinava and N. Zeh. 2012. A parallel buffer heap. In *ACM SPAA*. 214–223.
[23] Leslie G. Valiant. 2008. A Bridging Model for Multi-core Computing. In *Proc. Eur. Symp. Alg. (ESA)*. 13–28.