Revisiting the Cache Miss Analysis of Multithreaded Algorithms

Richard Cole * Vijaya Ramachandran †

September 19, 2012

Abstract

This paper concerns the cache miss analysis of algorithms when scheduled in work-stealing environments. We focus on the effect of task migration on cache miss costs, and in particular, the costs of accessing "hidden" data typically stored on execution stacks (such as the return location for a recursive call). We assume a setting where no delay due to false-sharing occurs during the computation. We obtain bounds that are a function of the number of steals, and thus they apply to any scheduler given bounds on the number of steals it induces.

Prior analyses focussed on the randomized work-stealing environment, and with the exception of [1], did not account for the just-mentioned hidden costs, and it is not clear how to extend them to account for these costs. By means of a new analysis, we show that for a variety of basic algorithms these task migration costs are no larger than the costs for the remainder of the computation, and thereby recover existing bounds. We also analyze a number of algorithms implicitly analyzed by [1], namely Scans (including Prefix Sums and Matrix Transposition), Matrix Multiply (the depth n in-place algorithm, the standard 8-way divide and conquer algorithm, and Strassen's algorithm), I-GEP, finding a longest common subsequence, FFT, the SPMS sorting algorithm, list ranking and graph connected components; we obtain sharper bounds in many cases.

*Computer Science Dept., Courant Institute of Mathematical Sciences, NYU, New York, NY 10012. Email: cole@cs.nyu.edu. This work was supported in part by NSF Grant CCF-0830516.

†Dept. of Computer Science, University of Texas, Austin, TX 78712. Email: vlr@cs.utexas.edu. This work was supported in part by NSF Grants CCF-0850775 and CCF-0830737.

1 Introduction

Work-stealing is a longstanding technique for distributing work among a collection of processors [4, 16, 2]. Work-stealing operates by organizing the computation in a collection of tasks with each processor managing its currently assigned tasks. Whenever a processor p becomes idle, it selects another processor q and is given (it *steals*) some of q's available tasks. A natural way of selecting q is for p to choose it uniformly at random from among the other processors; we call this randomized work stealing, RWS for short. RWS has been widely implemented, including in Cilk [3], Intel TBB [17] and KAAPI [15]. RWS is also an oblivious scheduler, in that it does not use system parameters such as block and cache size.

RWS scheduling has been analyzed and shown to provide provably good parallel speed-up for a fairly general class of algorithms [2]. Its cache overhead for private caches was considered in [1], which gave some general bounds on this overhead; these bounds were improved in [14] for a class of computations whose cache complexity function can be bounded by a concave function of the operation count. However, in a parallel execution, a processor executing a stolen task may incur additional cache misses while accessing data on the execution stack of the original task. We will refer to such accesses as "hidden data" accesses. Including these accesses in the cache miss analysis can cause cache miss costs of different tasks performing the same amount of work to vary widely, and this can lead to significant overestimates of the total cache miss cost when using a single bounding concave function.

Overview of Our New Analysis. As in prior work [1, 14], we assume a collection of parallel processors, each with a private cache, and we present an improved analysis of the cache miss overhead incurred when a work-stealing scheduler is used on a class of multithreaded computations that includes efficient algorithms for several problems, including matrix computations, FFT, sorting and list ranking. These algorithms are all instances of cache-regular HBP algorithms, notions we explain shortly. Our analysis is expressed as a function of the number of steals induced by the work-stealing scheduler, and thus gives bounds for any such scheduler once one obtains a bound on the number of steals it induces.

Cache-regularity is specified by identifying two desirable properties of a parallel algorithm, each related to additional cache miss costs incurred due to steals. The first type of cost arises because a processor p executing a task stolen from processor q may need to access data already in q's cache, thereby inducing additional cache misses. An upper bound on the number of additional cache misses depends on how tightly this data is clustered within blocks; we will make this precise through the notion of *cache friendliness*. The second cost can arise if there is a change in the alignment with respect to block boundaries of data generated during the computation of the task being executed by p. The *polynomial cache dependence property* will limit the increase in cache misses due to this cost by at most a constant factor. We now describe these two properties; they are defined more precisely in Section 3.

We say that a collection of r words is $f(r)$-cache friendly if it is contained in $r/B + f(r)$ blocks. A task τ is said to be f-cache friendly if its input and output, of total size r, is $f(r)$-cache friendly. $X(\tau) = \min\{M/B, r/B + f(r)\}$ is called the *task reload cost* for τ; it is an upper bound on the additional cache misses incurred by a stolen task τ over the cost of executing it within the sequential computation. An algorithm $\mathcal{A}$ is f-cache friendly if every task within $\mathcal{A}$ is f-cache friendly. We will restrict our attention to non-decreasing functions f such that $f(cr) \geq c \cdot f(r)$ for $c \geq 1$, which we call *normal* f; we also call a collection of tasks τ cache-normal if the function f for the corresponding

$X(\tau)$ is normal.

We say that a cache miss bound has polynomial cache dependence if a constant factor change in the size of the cache changes the cache miss bound by at most a constant factor (this is called the *regularity* property in [13]). We note that this is a property that is implicit in virtually all, if not all, of the known cache-oblivious sequential algorithms, where the analysis of these algorithms uses the "ideal cache assumption." This assumption asserts that the cache evictions that occur result in a minimum number of cache misses at that cache. Fortunately, the widely-used LRU policy results in at most twice as many cache misses when using a cache of twice the size [18]. One would like the impact on the cache miss cost to remain bounded by a constant even if the cache size were unchanged. As shown in [13], this is guaranteed if the algorithm has the property that if the cache size is halved, then the cache misses incurred increase by at most a constant factor. As we will see, this polynomial cache dependence property will be used in additional ways in the analysis of the class of parallel algorithms we consider.

Finally, we will say an algorithm is *cache-regular* if it is both cache-normal and has polynomial cache dependence.

The analysis due to Acar et al. [1] used the fact that each steal caused at most an additional M/B cache misses. In fact, this may overlook additional costs due to alignment issues when local variables are declared (though it may be that this was simply a case of implicitly invoking an assumption of polynomial cache dependence). Frigo and Strumpen [14] noted that M/B may be a pessimistic bound when the number of processors is large; they obtained improved bounds for a class of recursive algorithms whose only procedure calls are the recursive ones. Again, this analysis does not include some costs. In the current paper we present a fuller analysis of the cost of the cache miss overhead under work-stealing and in addition, we obtain improved bounds over those that can be obtained using [1] and [14] for several problems. As in the case in [1, 14], we assume that no false sharing is present (see [11] for results on bounding false sharing costs in algorithms).

Our analysis bounds the cache miss costs of cache-regular HBP algorithms: HBP algorithms were introduced in [9] and precisely defined in [11], and for completeness, we repeat the definition in Section 2. Our new analysis accounts for all cache miss costs in an HBP algorithm by identifying a subset of well-behaved disjoint tasks, which we term *natural* tasks; we define this in Section 2. Improved bounds will result from bounding the task reload cost for these disjoint natural tasks more tightly than the $O(M/B)$ bound used in [1][1].

We first present the results for *standard* HBP algorithms. An HBP algorithm is standard if it uses linear space and it is reasonably parallel in the sense that every recursive call occurs in a collection of two or more recursive calls that can be executed in parallel. All but two of the algorithms we analyze are standard.

Theorem 1.1. *Let $\mathcal{A}$ be a standard cache-regular HBP algorithm, and let $\mathcal{A}$ incur at most Q cache misses in a sequential execution. Now consider an execution of $\mathcal{A}$ which incurs S steals. Then there is a collection of $s = O(S)$ disjoint natural subtasks $\mathcal{C} = \{\nu_1, \nu_2, \cdots, \nu_s\}$ such that the cache miss cost of $\mathcal{A}$ is bounded by $O(Q + S + \sum_i X(\nu_i))$, where $X(\nu_i)$ is the task reload cost for ν_i.*

In a nutshell, the term $\sum_i X(\nu_i)$ replaces the term $S \cdot M/B$ in the analysis due to Acar et al. [1], at least matching the previous bound, and often improving on it.

[1] [1] does not identify natural tasks; rather it simply uses a bound of $O(M/B)$ on the reload costs due to each steal.

We analyze two non-standard (though linear space bounded) HBP algorithms, for LCS and I-GEP, and they are covered by the following theorem which handles a more general collection of HBP algorithms, algorithms that allow for some recursive calls to be to a single subproblem of smaller size, and are restricted to perform work that is at most an exponential in their input plus output size (We could analyze algorithms that perform even more work, but as the costs would be higher, and we are primarily interested in polynomial work, this does not seem important.)

Theorem 1.2 uses an extended variant of the task reload cost, denoted by $X'(\tau)$, where the term $f(\tau)$ in $X(\tau)$ is replaced by a more general term that includes a summation of such terms on tasks with geometrically decreasing sizes.

Theorem 1.2. *Let $\mathcal{A}$ be a cache-regular HBP algorithm whose tasks are all exponential-time bounded. Suppose that in a sequential execution $\mathcal{A}$ incurs at most Q cache misses. Now consider an execution of $\mathcal{A}$ which incurs S steals. Then there is a collection of $s = O(S)$ disjoint natural subtasks $\mathcal{C} = \{\nu_1, \nu_2, \cdots, \nu_s\}$ such that the cache miss cost of $\mathcal{A}$ is bounded by $O(Q + S \log B + \sum_i X'(\nu_i))$, where B is the block size, and $X'(\nu_i)$ is the extended task reload cost for ν_i. If $\mathcal{A}$ uses linear space, then the $O(S \log B)$ term improves to $O(S)$.*

With these theorems in hand, the cache miss analysis of a multithreaded algorithm under work-stealing reduces to an analysis of the sequential cache miss cost of the algorithm plus determining its worst case decomposition into disjoint natural tasks and bounding their task reload costs. This tends to be relatively straightforward. Thus the algorithm design task has two goals: first, to obtain an efficient sequential cache-miss performance, and second to maximize the potential parallelism, while bounding the increase in cache-miss cost due to steals by at most a constant factor. Since the number of steals is an increasing function of the number of processors, the latter is achieved by minimizing the overall task reload cost for a given number of steals. In turn, this amounts to maximizing the size of a worst-case collection $\mathcal{C}$ of tasks whose combined task reload cost, $\sum_{i \in \mathcal{C}} X(\nu_i)$ (or $X'(\nu_i)$), has cost bounded by the cache miss cost of a sequential execution. We note that this is similar to standard parallel algorithm design issues.

For a given multithreaded algorithm, let Q denote an upper bound on its sequential cache complexity, and let $C(S)$ be an upper bound on the number of cache misses incurred in a parallel execution with S steals. Table 1 gives our bounds on $C(S)$ for several algorithms. Previous work in [14] obtained the bounds for $C(S)$ shown under Our Results in Table 1 for the Depth-n Matrix Multiply [13], I-GEP [6] and computing the length of an LCS [5, 7] (and some stencil computations); as already noted, these bounds overlooked some costs, now included. We determine new bounds on $C(S)$ for several other algorithms (FFT, SPMS sort, List Ranking, Connected Components, and others). [1] had obtained a bound of $O(Q + S \cdot M/B)$ for every algorithm.

1.1 Computation Model

We model the computation induced by a program using a directed acyclic graph, or dag, D (good overviews can be found in [12, 2]). D is restricted to being a series-parallel graph, where each node in the graph corresponds to a size $O(1)$ computation. Recall that a directed series-parallel graph has start and terminal nodes. It is either a single node, or it is created from two series-parallel graphs, G_1 and G_2, by one of:

i. Sequencing: the terminal node of G_1 is connected to the start node of G_2.

Algorithm	Q	**Cache miss upper bound with S steals, $C(S)$**	
		In [1]	**Our Results ($O(Q+(M/B)\cdot S)$ or better)**
Scans, Matrix Transp. (MT)	$\frac{n}{B}$	$Q+(M/B)\cdot S$	$Q+S$
Depth-n-MM, 8-way MM, Strassen	$n^3/(B\sqrt{M})$	$Q+(M/B)\cdot S$	$Q+S^{\frac{1}{3}}\frac{n^2}{B}+S$ (in BI) [2] $Q+S^{\frac{1}{3}}\frac{n^2}{B}+S\cdot\min\{M/B,B\}$ (in RM)
I-GEP	$n^3/(B\sqrt{M})$	$Q+(M/B)\cdot S$	$Q+S^{\frac{1}{3}}\frac{n^2}{B}+S\cdot\min\{M/B,\log B\}$ (in BI) [2] $Q+S^{\frac{1}{3}}\frac{n^2}{B}+S\cdot\min\{M/B,B\}$ (in RM)
Finding LCS sequence	$n^2/(BM)$	$Q+(M/B)\cdot S$	$Q+\frac{n}{B}\sqrt{S}+S\cdot\min\{M/B,B\}$ [2]
FFT, SPMS Sort	$\frac{n}{B}\log_M n$	$Q+(M/B)\cdot S$	$Q+S\cdot B+\frac{n}{B}\frac{\log n}{\log[(n\log n)/S]}$
List Ranking	$\frac{n}{B}\log_M n$	$Q+(M/B)\cdot S$	$Q+S\cdot B+\frac{n}{B}\frac{\log n}{\log[(n\log n)/S]}$
Graph Conn. Comp.	$\frac{n}{B}\log_M n$	$Q+(M/B)\cdot S$	$Q+S\cdot B+\frac{n}{B}\frac{\log^2 n}{\log[(n\log n)/S]}$

Table 1: Bounds for cache miss overhead, $C(S)$, under RWS in [1] (column 3) and our results (column 4) for some HBP algorithms; $O(\ \cdot\ )$ omitted on every term. The sequential cache complexity is Q (a term $f(r)$, specified below in Definition 3.2, is omitted from Q). Always, the new bound matches or improves the bound in [1].

ii. A parallel construct (binary forking): it has a new start node s and a new terminal node t, where s is connected to the start nodes for G_1 and G_2, and their terminal nodes are connected to t.

One way of viewing this is that the computational task represented by graph G decomposes into either a sequence of two subtasks (corresponding to G_1 and G_2 in (i)) or decomposes into two independent subtasks which could be executed in parallel (corresponding to G_1 and G_2 in (ii)). The parallelism is instantiated by enabling two threads to continue from node s in (ii) above; these threads then recombine into a single thread at the corresponding node t. This multithreading corresponds to a fork-join in a parallel programming language.

We will be considering algorithms expressed in terms of tasks, a simple task being a size $O(1)$ computation, and more complex tasks being built either by sequencing, or by forking, often expressed as recursive subproblems that can be executed in parallel. Such algorithms map to series-parallel computation dags, also known as nested-parallel computation dags.

Processing Environment We consider a computing environment which comprises p processors, each equipped with a local memory or cache of size M. There is also a shared memory of unbounded size. Data is transferred between the shared and local memories in size B blocks (or cache lines). The term *cache miss* denotes a read of a block from shared-memory into processor C's cache, when a needed data item is not currently in cache, either because the block was never read by processor C, or because it had been evicted from C's cache to make room for new data.

Each processor maintains a work queue, on which it stores tasks that can be stolen. When a processor C generates a new stealable task it adds it to the bottom of its queue. If C completes its current task, it retrieves the task τ from the bottom of its queue, and begins executing τ. The steals, however, are taken from the top of the queue. An idle processor C' picks a processor C'' uniformly at random and independently of other idle processors, and attempts to take a task (to

steal) from the top of C'''s task queue. If the steal fails (either because the task queue is empty, or because some other processor was attempting the same steal, and succeeded) then processor C' continues trying to steal, continuing until it succeeds.

Definition 1.1. *A task kernel is the portion of a task computation dag that remains after the computation dags for all its stolen subtasks are removed.*

There is another cost that could arise, namely cache misses due to false sharing. As in [1, 14], we assume that there is no false sharing, perhaps as a result of using the Backer protocol, as implemented in Cilk [3]. Even when false sharing is present [10, 11], the cache miss costs as analyzed here remain relevant, since false sharing can only further increase the costs.

Execution Stacks. The original task in the algorithm and each stolen subtask will have a separate computation thread. The work performed by a computation thread for a task τ is to execute the task kernel τ^K for task τ. We now explain where the variables generated during the computation are stored, including the variables needed for synchronization at joins and for managing procedure calls. In a single processor algorithm a standard solution is to use an execution stack. We proceed in the same way, with one stack per thread. Each computation thread will keep an execution stack on which it stores the variables it creates: variables are added to the top of the stack when a subtask begins and are released when it ends.

Usurpations. Let C be the processor executing task kernel τ^K. As τ^K's execution proceeds, the processor executing it may change. This change will occur at a join node v at which a stolen subtask τ' ends, and it occurs if the processor C' that was executing the stolen task τ' reaches the join node later than C. Then, C' continues the execution of τ^K going forward from node v. C' is said to *usurp* the computation of τ^K; we also say that C' usurps C. In turn, C' may be usurped by another processor C''. Indeed, if there are k steals of tasks from τ, then there could be up to k usurpations during the computation of τ^K.

A usurpation may cause cache misses to occur due to hidden data. By hidden data we mean undeclared variables stored on a task's execution stack such as the variables used to control task initiation and termination. If in cache, this data can be accessed for free by a processor C, but a usurper C' incurs cache misses in accessing the same data.

1.2 Prior Work

In both [1] and [14] the computation is viewed as being split into subtasks both at each fork at which there is a steal and at the corresponding join node, and then the costs of these subtasks are bounded by making a worst case assumption that each subtask is executed beginning with an empty cache. Further, for most analyses in [14], blocks are assumed to have size $O(1)$. In [1], the following simple observation is used: whether or not it is stolen, a subtask accessing $2M$ or more words would incur the same number of cache misses, up to constant factors. Thus the cache miss overhead due to the steals is bounded by $O(\frac{M}{B} \cdot S)$, where S is the number of stolen tasks. In [14], improved bounds are obtained in the case the cache misses incurred by any task in the computation can be bounded by a concave function C_f of the work the task performs: if W is the total work, the cache miss cost is bounded by $S \cdot C_f(W/S)$, which can yield better bounds when the average stolen task size is less than M.

In this paper, we work with the class of cache-regular Hierarchical Balanced Parallel (HBP) algorithms [9, 11], which includes the algorithms considered in [14] and many others. We develop

a cache miss analysis for HBP algorithms that includes the cost of accesses to hidden data, and we obtain cache miss bounds that match or improve on those obtained using [1]. We also match bounds obtained in [14] even after including the cache miss overhead of accesses to hidden data.

Road-map: In Section 2 we review the definition of HBP algorithms. In Section 3 we bound the cache miss costs for BP algorithms, a subset of HBP algorithms. The analysis for HBP algorithms is given in Section 4. Also, in Section 4.1, we illustrate the applicability of our results by obtaining the cache miss bounds for FFT; the remaining cache miss bounds are given in Section 5. Finally, in Section 6, we determine lower bounds on the problem sizes for which optimal speed-up is achieved, as a function of p, M, and B.

2 HBP Algorithms

We review the definition of HBP algorithms [9, 11]. Here we define the size of a task τ, denoted $|\tau|$, to be the number of already declared distinct variables it accesses over the course of its execution (this does not includes variables τ declares during its computation). Also, we will repeatedly use the following notation: τ_w will denote the steal-free task that begins at a fork node w and ends at the corresponding join node.

Definition 2.1. *A* BP computation π *is an algorithm that is formed from the down-pass of a binary forking computation tree T followed by its up-pass, and satisfies the following properties.*

i. In the down-pass, a task that is not a leaf performs only $O(1)$ computation before it forks its two children. Likewise, in the up-pass each task performs only $O(1)$ computation after the completion of its forked subtasks. Finally, each leaf node performs $O(1)$ computation.

ii. Each node declares at most $O(1)$ variables, called local variables*; π may also use size $O(|T|)$ arrays for its input and output, called* global variables*.*

iii. Balance Condition. *Let w be a node in the down-pass tree and let v be a child of w. There is a constant $0 < \alpha < 1$ such that $|\tau_v| \leq \alpha|\tau_w|$.*

Although a BP computation can involve sharing of data between the tasks at the two sibling nodes in the down-pass tree, it is not difficult to see that the number of nodes, k, in a BP computation is polynomial in the size n of the task at the root of the BP computation (recall that the size of this task is the number of already declared variables that it accesses during the computation). As it happens, for all the BP algorithms we consider, $k = \Theta(n)$. A simple BP example is the natural balanced-tree procedure to compute the sum of n integers. This BP computation has $n = \Theta(k)$, and there is no sharing of data between tasks initiated at sibling nodes in the down-pass tree.

Definition 2.2. *A* Hierarchical Balanced Parallel (HBP) Computation *is one of the following:*

1. A Type 0 Algorithm, a sequential computation of constant size.

2. A Type 1 Algorithm, a BP computation.

[2]These bounds were obtained for Depth-n-MM and I-GEP by [14], but with hidden costs overlooked, which adds a term $S \cdot \min\{M/B, \log B\}$ to the cost of I-GEP when using the BI format. For LCS, [14] bounded the cost of finding the length of the sequence, but not the sequence itself; also, again, there was the same additional charge due to hidden costs.

3. Sequencing: A sequenced HBP algorithm of Type t results when $O(1)$ HBP algorithms are called in sequence, where these algorithms are created by Rules 1, 2, or 4, and where t is the maximum type of any HBP algorithm in the sequence.

4. Recursion: A Type $t+1$ recursive *HBP algorithm, for $t \geq 1$, results if, for a size n task, it calls, in succession, a sequence of $c = O(1)$ ordered collections of $v(n) \geq 1$ parallel recursive subproblems, where each subproblem has size $\Theta(r(n))$, where $r(n)$ is bounded by $\alpha \cdot n$ for some constant $0 < \alpha < 1$.*

Each of the c collections can be preceded and/or followed by a sequenced HBP algorithm of type $t' \leq t$, where at least one of these calls is of type exactly t. If there are no such calls, then the algorithm is of Type 2 if $c \geq 2$, and is Type 1 (BP) if $c = 1$.

Each collection of parallel recursive subproblems is organized in a BP-like tree T_f, whose root represents all of the $v(n)$ recursive subproblems, with each leaf containing one of the $v(n)$ recursive subproblems. In addition, we require the same balance condition as for BP computations for nodes in the fork tree.

We also require the balance condition to extend to the procedures called by the recursive tasks, in the following sense. Let τ be a recursive type t task and let τ' be one of its recursive calls. Let ν be one of the subtasks forming τ (either one of the up to c collections in (4), or one of the tasks in the up to $c+1$ sequenced tasks in (4)) and let ν' be the corresponding subtask in τ'. In addition to having $|\tau'| \leq \alpha|\tau|$, we require that $|\nu'| \leq \alpha|\nu|$, for every such ν.

Lemma 2.1. *Let u and v be the children of a fork node. Then $|\tau_u| = \Theta(|\tau_v|)$.*

Proof. Let w denote the fork node. $|\tau_w| \leq O(1) + |\tau_u| + |\tau_v|$, since only $O(1)$ variables are accessed by the computation at node w. As $|\tau_v| \leq \alpha|\tau_w|$, $|\tau_u| \geq (1-\alpha)|\tau_w| - O(1)$, and hence $|\tau_u| \geq \frac{1-\alpha}{\alpha}|\tau_v| - O(1) = \Theta(|\tau_v|)$. ■

Matrix Multiply (MM) with 8-way recursion is an example of a Type 2 HBP algorithm. The algorithm, given as input two $n \times n$ matrices to multiply, makes 8 recursive calls in parallel to subproblems with size $n/2 \times n/2$ matrices. This recursive computation is followed by 4 matrix additions, which are BP computations. Here $c = 1$, $v(n^2) = 8$, and $r(n^2) = n^2/4$. Depth-n-MM [13, 8] is another Type 2 HBP algorithm for MM with $c = 2$, $v(n^2) = 4$, and $r(n^2) = n^2/4$.

Linear Space Bound and Standard HBP. We obtain stronger bounds for certain types of HBP computations, including *linear space bounded* HBP computations and *standard* HBP computations. Linear space boundedness simply means that the computation uses space that is linear (or smaller) in the size of its input and output data, with the additional caveat that in an HBP computation, the linear space bound also applies to all recursive tasks.

An HBP computation is *standard* if is linear space bounded, and also, for every subproblem of size $n = \omega(1)$, for each of its collections, $v(n) > 1$, i.e., each recursive call is to a collection of at least two parallel subproblems.

All the HBP algorithms analyzed in this paper are linear space bounded, and all but two are standard HBP algorithms.

With the HBP definition in hand, we can now define natural tasks.

Definition 2.3. *A* Natural Task *is one of the following:*

1. A task built by one of Rules 1–4 in Definition 2.2.

2. A task that could be stolen: a task τ_w beginning at a node w in a fork tree and ending at the corresponding node of the corresponding join tree and including all the intermediate computation.

Work Stealing Detail. At a fork node, it is always the right child task that is made available for stealing by being placed on the bottom of the task queue. This ensures that in a BP computation, a task kernel τ^K always comprises a contiguous portion of the leaves in τ as executed in sequential order thereby minimizing additional cache misses over the sequential cache complexity. For an HBP computation, an analogous requirement applies to each ordered collection of parallel recursive tasks. As mentioned earlier, in work stealing the task queue is a double-ended queue, where the steals are performed in queue order, and the local computation extracts tasks from the task queue in stack order.

3 Bounding the Cache Misses

3.1 Preliminaries

We begin with some natural assumptions regarding the block allocation. Specifically, we assume that space is allocated in block-sized units, that blocks allocated to different cores are distinct, and are distinct from blocks allocated for global variables.

We now address two issues that are also relevant in the sequential cache oblivious setting [13]: the robustness of the cache miss bound to a constant factor change in the cache size, and the effect of non-contiguous data access patterns on the cache miss bound.

Dependence of Cache Miss Bound on Size of Cache. The cache oblivious model [13] assumes that an ideal offline cache replacement policy FIF is used to evict blocks. This is often justified by the known result that LRU is a provably good online approximation to FIF in that it incurs no more than twice the number of cache misses when used on a cache with double the size of the cache used by FIF [18] on any sequence of memory accesses. However, it is not difficult to construct examples of data accesses where even increasing the size of the cache by 1 block can result in an unbounded improvement in the cache miss bound on sufficiently large sequences.

We now define the notion of *polynomial cache dependence*, which is a property that guarantees that LRU will gives bounds within a constant factor of those guaranteed by FIF. (This property is also noted in [13].)

Definition 3.1. *Let $Q(\mathcal{A}, I, M)$ be the number of cache misses incurred when executing algorithm $\mathcal{A}$ sequentially on input I using a processor with a cache of size M. $\mathcal{A}$ is said to have* Polynomial Cache Dependence *if for all inputs I, and any constant $c > 1$, $Q(\mathcal{A}, I, M/c) = O(Q(\mathcal{A}, I, M))$.*

Virtually all of the known cache miss bounds for sequential algorithms satisfy this property, and thus LRU gives a performance that is provably within a constant factor of the offline optimal FIF with a cache of the same size. In addition, in our analysis we will invoke the polynomial cache dependence property to handle the *block alignment* issue.

Dependence of Cache Miss Bound on Data Layout. Some computations, such as those on matrices in row major format, incur extra cache miss costs when they need to access a collection of data that are not packed into contiguous locations. The analysis of such accesses has been studied widely in sequential cache efficient algorithms, and the notion of a 'tall cache' has often been used in this context [13].

For example, consider the cache miss bound $Q(n, M, B)$ for Depth-n-MM when the $n \times n$ matrix is stored in row-major format. We have $Q(n, M, B) = \frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + n$. The second term $\frac{n^2}{B}$ represented the cost of sequentially reading (in row-major order) the matrix into cache. It dominates the leading term $\frac{n^3}{B\sqrt{M}}$ when the input size n^2 is smaller than the cache size, and is otherwise dominated by the leading term. The third term occurs because the algorithm reads in submatrices of varying sizes during the computation, and the row major order of storing the data implies that an $r \times r$ submatrix is stored in $r \cdot \lceil r/B \rceil \geq r$ blocks. In the literature, this 3-term bound for $Q(n, M, B)$ is often abbreviated as $Q(n, M, B) = O(\frac{n^3}{B\sqrt{M}})$; this assumes that the input size n^2 is larger than the size of the cache, and it also assumes a *tall cache* $M \geq B^2$, which would cause the third term to be dominated by the second term when the input size is larger than M. Similar issues arise with the cache complexity of other computational problems, such as sorting.

In work-stealing this issue is more challenging, since there is less control over organizing the sequencing of tasks to minimize the cache miss costs. To help bound the cache miss costs of non-constant sized blocks, we formalize the notion of data locality with the following definition.

Definition 3.2. *A collection of r words of data is f-cache friendly if the r words are contained in $r/B + f(r)$ blocks. A task τ is f-cache friendly if the collection of data it accesses is f-cache friendly. Finally, an algorithm $\mathcal{A}$ is f-cache friendly if all its natural tasks are f-cache friendly.*

As r words occupy at most r blocks it suffices to consider functions f with $f(r) \leq r$. In addition, we limit our attention to the following well-behaved functions f, which we call normal f.

Definition 3.3. *A cache-friendliness function f is normal if it is non-decreasing and for all $c \geq 1$, $f(cr) = O(c \cdot f(r))$. For short, if a task or algorithm has a normal cache friendliness function, we will say it is normal.*

In the example of Depth-n-MM when the matrices are stored in row-major format, we have $f(r) = \sqrt{r}$, which is a normal cache-friendliness function, and the data accessed by a task τ multiplying $r \times r$ matrices occupies at most $r^2/B + f(r)$ blocks. The tasks τ in all algorithms we consider use only normal cache-friendliness functions since they are all either $O(1)$- or $O(\sqrt{|\tau|})$-friendly.

All of the algorithms we consider have both polynomial cache dependence and a normal cache-friendliness function. Accordingly, we will use the following terminology.

Definition 3.4. *An algorithm $\mathcal{A}$ is* cache-regular *if it has both polynomial cache dependence and a normal cache-friendliness function.*

Next, we wish to consider the additional costs for executing a natural task τ (Definition 2.3) in *stand-alone* mode. Consider the the following two scenarios.

(i) τ is not a stolen task.
Immediately prior to starting the computation on τ, the execution stack being used for this computation is flushed. Let P be the processor on which τ is being executed.

(ii) τ is a stolen task.
Let P be the processor from which τ was stolen.

We define τ's *reload cost*, denoted by $X(\tau)$, as the number of blocks in P's cache, before the flush or the steal, that might be accessed during the execution of τ in either of these scenarios. Sometimes it is convenient to write $X(r)$ where $r = |\tau|$.

Lemma 3.1. *Let τ be a natural task in HBP algorithm $\mathcal{A}$ and suppose it has cache friendliness function f. τ's reload cost is at most $X(\tau) = \min\{M/B, |\tau|/B + f(|\tau|)\}$.*

Proof. Let time t be the instant before the flushing in Case 1, or the moment before the steal in Case 2. There are at most M/B blocks in the cache of processor P at time t, which therefore upper bounds the reload cost. Also, the only blocks currently in P's cache at time t that could be accessed for τ's computation are blocks storing τ's input or output variables, and there are at most $|\tau|/B + f(|\tau|)$ of these. ■

Let us denote the second term, $|\tau|/B + f(|\tau|)$, in $X(\tau)$ by $\rho(\tau)$ (or $\rho(|\tau|)$), and call it the *single access cost* for τ, since it is the number of distinct blocks accessed by the computation during the execution of τ. Then, τ's reload cost becomes $X(\tau) = \min\{M/B, \rho(\tau)\}$, i.e., the reload cost is the minimum of the single access cost or M/B.

We can now express the cache complexity of Depth-n-MM as $Q(n, M, B) = O\left(\frac{n^3}{B\sqrt{M}} + \rho(n)\right)$, where $\rho(r) = r^2/B + r$ bounds the single access cost of any task of size r in the Depth-n-MM computation. More generally, the cache complexity of a (sequential) algorithm can be written as the sum of two terms, the first being the asymptotic complexity when the accessed data overflows the cache, and the second being the single access cost, a bound on the number of blocks used to store the data accessed by the algorithm, or equivalently, the cost of first accessing these blocks. This notation represents an alternate (to tall cache and other assumptions) and uniform method of coping with the various terms in the cache complexity expression. Additionally, the reload cost $X(\tau)$ is relevant in the parallel context when a task is moved from one processor to another, and the goal is to account for the *additional* cache miss cost incurred by the second processor when it needs to read in the task's data once again. Here, $X(\tau)$ limits this excess cache miss cost by at most M/B since any block read beyond the first M/B distinct blocks would also be read by the original processor (had the task not moved from it), and hence does not contribute to the excess cache miss cost. In this paper, we will deal with the task reload cost rather than the single access cost, since we are interested in bounding the excess cache miss cost incurred due to steals, and not the intrinsic cache complexity of a sequential execution.

3.2 Our Approach

We start by reviewing the primary problem analyzed in [14] to show why the approach using concave functions need not yield tight bounds.

Depth-n matrix multiply. Possibly there is a steal of a leaf node task τ_v in a final recursive call; if the processor P' executing τ_v usurps the remainder of the computation, then P' is going to carry out the processing on the path to the root of the up-pass tree, and P' could end up executing $\log n$ nodes in the up-pass. All that has to be done at these nodes is to synchronize and possibly terminate a recursive call, but these both require accessing a variable stored on the execution stack E_τ for the parent task τ (the one from which the steal occurred). The variables P' accesses will be consecutive on E_τ. This results in $\Theta([\log n]/B)$ cache misses. In general, a task that performs $x \geq \log n$ work may incur $\Theta(x/[B\sqrt{M}] + [\log n]/B)$ cache misses which yields a

bound of $\Theta(S^{\frac{1}{3}}\frac{n^2}{B} + [S \log n]/B)$ cache misses rather than $O(S^{\frac{1}{3}}\frac{n^2}{B} + S)$ as in [14]; the former is a larger bound when $B = o(\log n)$ and $S \geq n^3/\log^{3/2} n$.

We provide two more examples showing that the approach using concave functions need not yield tight bounds.

Prefix Sums. We consider the standard 2-pass tree-based algorithm, where the leaves correspond to the input data in left to right order. In the first pass, each internal tree node seeks to compute the sum of values at the leaves of its subtree. The BP algorithm will compute this value for a node v as the sum of the values at its children, and this value is computed on the up-pass of the BP computation. Suppose these values are stored in an array in infix order, as is standard (as they are needed for the second pass, they have to be saved).

We analyze the complexity of tasks involved in this first pass. Possibly there is a steal of a leaf node task τ_v; if the processor P' executing τ_v usurps the remainder of the computation, then P' is going to carry out the processing on the path to the root of the up-pass tree. So possibly it performs $\Theta(\log n)$ work, while incurring $\Theta(\log n - \log B)$ cache misses (because there is a write to the output array at each node in the up-pass tree, and above level $\log B$ in the tree, the write locations are going to be more than distance B apart, and hence each one results in a cache miss). In general, a task that performs x work can incur $\Theta(\log n - \log B + x/B)$ cache misses. While this is a concave function, if there are $S = n/\log n$ stolen subtasks, this analysis will yield a bound of $O(n)$ cache misses, but nothing tighter, while for $B = o(\log n)$, the correct bound is $O(n/B)$.

It turns out that if the output is stored in postfix order, the cost for a task performing x work can be bounded by $O(x/B)$ (that is, for tasks as defined in the analysis in [14]), and thus in this case there is no need for a new analysis.

FFT. The FFT algorithm includes calls to Matrix Transpose: in these calls subtasks that perform x work will incur $O(x/B)$ cache misses and this bound is tight in the worst case, while recursive FFT calls that perform x operations will incur $O(\frac{x}{B \log M})$ cache misses. Using the bound for the matrix transpose procedure would overestimate the costs of the whole algorithm in the concave function analysis.

Our Method. Our method determines bounds on the worst case number of natural tasks of a given size, and shows that the cache miss cost of the given algorithm is bounded by $O(Q)$, where Q is an upper bound on the cache misses incurred in a sequential execution of the algorithm, plus $O(S)$, plus the task reload cost for $O(S)$ disjoint natural tasks. We prove our bound in three parts:

1. A bound assuming there are no usurpations. (Recall that usurpations are discussed at the end of Section 1.1.)

2. A bound on the cost of the up-passes following the usurpations.

3. A bound on the costs of re-accessing data following a usurpation, aside the costs in (2).

For BP computations, which we analyze in the next section, (3) does not arise; (1) is handled by Lemma 3.4 below, and (2) by Lemma 3.5. For HBP computations the argument is more involved; it is given in Section 4. The analysis in [14] bounds (1) and (3) accurately, assuming the cache miss function is a tight concave function of the work, but it does not bound (2).

3.3 Analysis of BP Computations

We begin, in Section 3.3.1, with a technical issue concerning block alignments, and then, in Section 3.3.2, we present the analysis of BP computations.

3.3.1 Block Alignments and Cache Sizes

Consider a BP algorithm $\mathcal{A}$. A sequential execution of $\mathcal{A}$ has a single execution stack S. In a parallel execution, a stolen task τ' starts a new execution stack S', and if the first element e added to S' was in the middle of a block in stack S in a sequential execution, then all data placed on S' will have their block boundaries shifted backward by the offset of e with respect to its block boundary in S. In particular, this means that the contents of a single block in S in a sequential computation could be spread across two blocks in the execution stack of a stolen task in a parallel execution. We refer to this as a block misalignment. If τ'' is a task stolen from τ' then there could be a similar block misalignment of the execution stack of τ'' with respect to S'. The following lemma bounds the effect of block misalignment in a parallel execution of a BP computation.

Lemma 3.2. *Let τ^K be the kernel of a stolen task τ in a parallel execution Π of a BP computation $\mathcal{A}$, and let β be a block used to store variables accessed during the computation of τ^K in the sequential execution of $\mathcal{A}$. During the computation of τ^K in the parallel execution Π, the variables in β accessed by τ^K are stored in at most 3 blocks.*

Proof. If β stores only global variables, then the same block will be accessed by Π. Otherwise β contains variables that were placed on the execution stack in the sequential execution of $\mathcal{A}$. In a parallel execution π of this BP computation, the variables accessed by task kernel τ^K can be stored in at most two execution stacks: the execution stack E for τ^K, and the execution stack E_p, where the parent p of the start node of τ stored its local variables. This implies that the contents of β could be spread across at most 4 blocks in a parallel execution Π due to block misalignment. However, the bound can be reduced to 3 blocks by observing that if β is spread across both E_p and E, then it must include the first element in E, and hence could include at most one block in E, for a total of at most 3 blocks. ■

The consequence of this lemma is that if the processors performing execution Π have cache sizes of $3M$ or larger, then all the variables stored in cache during the sequential execution, assuming a cache of size M, can be held in cache during execution Π. Furthermore, the data brought in cache by one cache miss in the sequential execution is brought in cache following at most 3 cache misses in execution Π.

In fact, the execution Π will be on processors with cache size M. Thus we are interested in algorithms for which a constant factor reduction in the size of the cache increases the cache misses incurred in their sequential execution by just a constant factor, which is ensured by the polynomial cache dependence property.

3.3.2 The Analysis

There are two ways in which additional cache misses can arise due to a processor q stealing a subtask τ from a processor p. The first is that q might access τ's input and output variables,

and these variables might already have been in p's cache, thereby causing additional cache misses compared to a computation in which this steal did not occur.

The second arises due to usurpations. Again, a processor q usurping a processor p may need to access variables already in p's cache, causing additional misses compared to the usurpation-free execution.

We are now ready to analyze the cache misses incurred in executing a task kernel τ^K as part of an S-steal execution Π of $\mathcal{A}$. As in [11], we specify the *steal path* P, the path in the fork tree separating τ^K from the subtasks stolen from τ. The path comprises alternating segments descending to the left and to the right, denoted $L_1, R_1, L_2, R_2, \cdots, L_k, R_k$, where the first segment may be empty (see Figure 1.) In the following, we will denote by τ_x the natural task that begins at a node x in the fork tree and ends at the complementary node in the join tree.

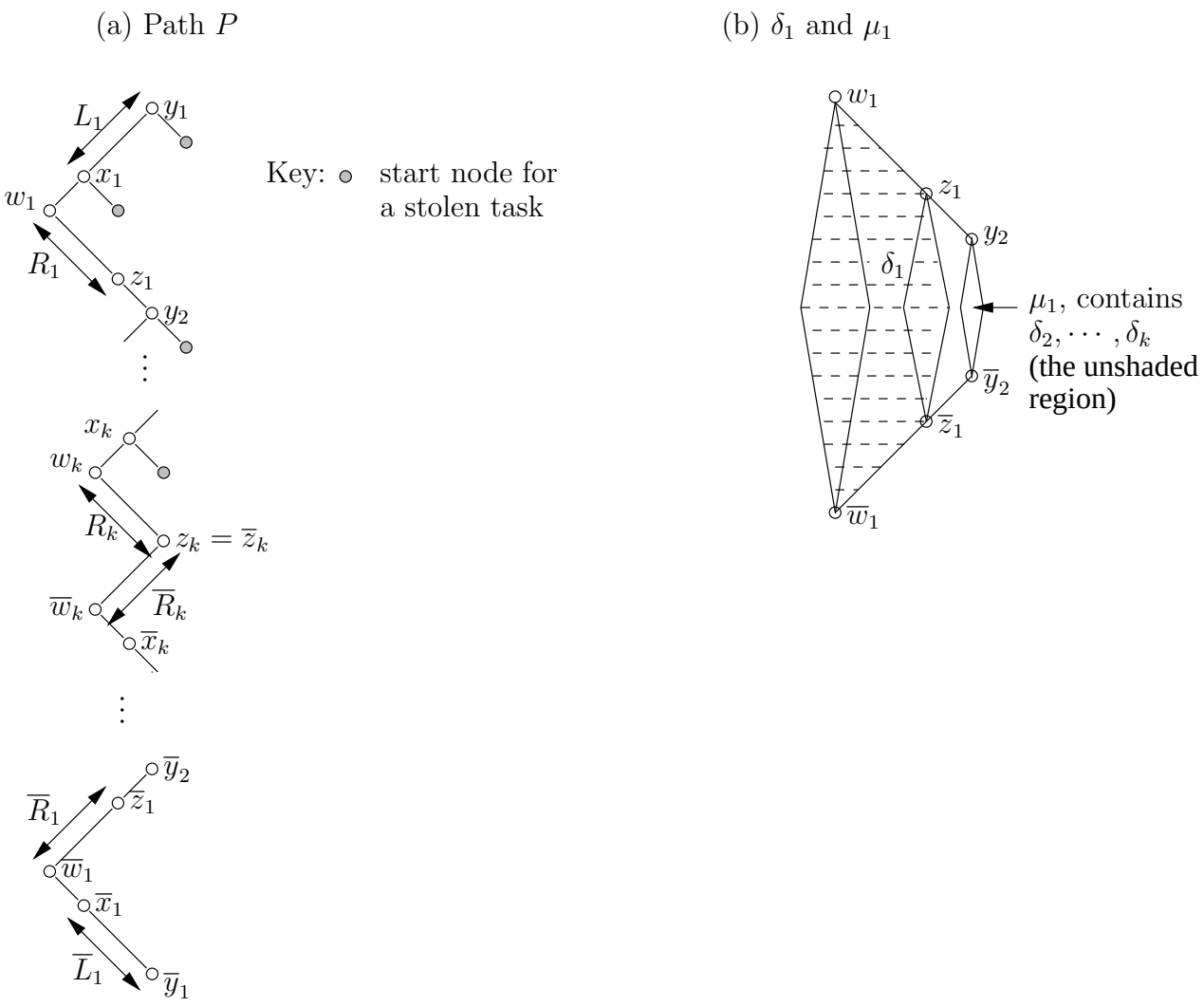

Figure 1: The path P.

The right children of the nodes on L_i, for $1 \leq i \leq k$ are the roots of the subtasks stolen from τ

in forming τ^K. Let path L_i begin at node y_i and end at node x_i, and let path R_i begin at node w_i, the left child of x_i, and end at node z_i, where for $i < k$, z_i has right child y_{i+1}. The node z_k is a leaf. Also, let $\overline{L}_i$ and $\overline{R}_i$, for $1 \leq i \leq k$, denote the paths in the join tree complementary to L_i and R_i, respectively. Likewise, for any node v in the fork tree, let $\overline{v}$ denote the complementary node in the join tree. For $1 \leq i < k$, let μ_i be the sub-task of τ^K that begins at node y_{i+1} and ends at its complementary node; μ_k denotes an empty task. Finally, for $1 \leq i \leq k$, let u_i be the left child of w_i, except that we let $u_k = w_k$ if w_k is a leaf, and let v_i be the sibling of u_i (v_k does not exist if w_k is a leaf). We will not need the v_i in this section, but they will be used in the next section when we address HBP algorithms.

For the purposes of the analysis, we will be identifying the following subunits of τ^K, which we call *task differences*.

Definition 3.5. *The portion of τ_{w_i} that remains once μ_i is removed is called the ith* task difference*; δ_i denotes this task difference; it is also denoted by $\tau_{w_i} - \mu_i$.*

The following lemma is immediate (recall that all computation occurs at the nodes in the computation dag).

Lemma 3.3. *$\tau^K = \cup_i[\delta_i \cup L_i \cup \overline{L}_i]$, where L_i and $\overline{L}_i$ are interpreted as denoting the vertices on these paths.*

In addition, we note that $\sum_i |L_i| = \sum_i |\overline{L}_i| = O(S)$. As the execution of any one node incurs $O(1)$ cache misses, we can immediately bound the cache misses incurred in executing $\cup_i[L_i \cup \overline{L}_i]$ by $O(S)$. Thus all that remains is to bound the cache misses incurred in executing δ_i, which we do first for usurpation-free computations and then allowing usurpations.

Definition 3.6. *An execution of task kernel τ^K is* usurpation-free *if at every join ending a steal, the processor C currently executing τ^K continues to execute τ^K following the join.*

The following notation will be helpful.

Definition 3.7. *Let W be a subset of the nodes in the computation dag for $\mathcal{A}$. With a slight overload of notation, we let $Q(W, M, I)$ denote the cache misses incurred in executing the nodes in W as part of the sequential execution of algorithm $\mathcal{A}$ on input I when the cache has size $\Theta(M)$. We will often omit the parameters M and I when they are clear from the context.*

Lemma 3.4. *Let $\mathcal{A}$ be a cache-regular BP algorithm Let Π be an S-steal computation of $\mathcal{A}$. If the computation of τ^K in Π is usurpation-free, the number of cache misses incurred in computation Π when executing δ_i is bounded by $O(Q(\delta_i) + X(\tau_{u_i}))$, where u_i is the left child of w_i, and $X(\tau_{u_i})$ is the reload cost of τ_{u_i}.*

Proof. We will be comparing the sequential execution of $\mathcal{A}$ using a cache of size M with the parallel execution Π using a cache of size $3M$. By Lemma 3.2, the data held in each block β used in executing δ_i during the sequential execution of $\mathcal{A}$ is stored in at most 3 blocks in execution Π. Thus the execution Π using a cache of size $3M$ can store all the variables in cache in the sequential execution which uses a cache of size M. Then in executing δ_i as part of computation Π there will be at most three cache misses for each cache miss incurred in the sequential execution, plus potentially cache misses for the initial accesses to the blocks holding the input and output variables for τ_{w_i}. It follows that in Π there will be at most $Q(\delta_i) + X(\tau_{w_i})$ cache misses when executing δ_i.

We complete the proof by showing that $X(\tau_{w_i}) = O(X(\tau_{u_i}))$. $X(\tau_{w_i}) = \min\{M/B, |\tau_{w_i}|/B + f(|\tau_{w_i}|)\}$. Let v_i denote w_i's right child. Now $|\tau_{w_i}| \leq O(1) + |\tau_{u_i}| + |\tau_{v_i}|$. By Lemma 2.1, $|\tau_{v_i}| = O(|\tau_{u_i}|)$, and so $|\tau_{w_i}| = O(|\tau_{u_i}|)$. As f is normal, it follows that $f(|\tau_{w_i}|) = O(f(|\tau_{u_i}|))$. Thus $X(\tau_{w_i}) = O(\min\{M/B, |\tau_{u_i}|/B + f(|\tau_{u_i}|)\}) = O(X(\tau_{u_i}))$. ■

Next, we bound the additional costs due to usurpations and in particular the costs of executing the nodes on a path $\overline{R}_i$, the path in the join tree complementary to R_i.

Lemma 3.5. *Let $\mathcal{A}$ be a cache-regular BP algorithm. The cache misses incurred in executing the nodes on path $\overline{R}_i$, when including the costs of usurpations, are bounded by $O(Q(\delta_i) + X(\tau_{u_i}))$, where u_i is the left child of w_i.*

Proof. The variables accessed in executing these nodes are either (i) global variables, or (ii) variables generated by ancestors of node w_i, or (iii) variables generated by τ_{w_i}; the only variables of type (iii) that remain present when $\overline{R}_i$ is being computed are the variables generated by nodes on R_i. There are a total of $O(|R_i|)$ such variables and they are stored consecutively on the execution stack being used by τ^K. The accesses to the variables of types (i) and (ii) cost no more than the accesses analyzed in Lemma 3.4; the variables of type (iii) cause at most an additional $\lceil |R_i|/B) \rceil$ cache misses for the first accesses to the blocks storing these variables, and the costs of subsequent accesses are again bounded by the analysis in Lemma 3.4. Thus the total cache misses incurred are bounded by $O(Q(\delta_i) + X(\tau_{u_i}) + |R_i|/B)$.

We now bound $|R_i|$. By Lemma 2.1, the height of the subtree of the fork tree rooted at w_i is $O(\log |\tau_{w_i}|)$. Clearly, path R_i is no longer than the height of the fork tree rooted at w_i. Thus $|R_i| = O(\log |\tau_{w_i}|) = O(1 + \log |\tau_{u_i}|) = O(|\tau_{u_i}|)$. We conclude that the total cache misses incurred are bounded by $O(Q(\delta_i) + X(\tau_{u_i}) + |\tau_{u_i}|/B) = O(Q(\delta_i) + X(\tau_{u_i}))$. ■

Lemma 3.6. *Let $\mathcal{A}$ be a cache-regular BP algorithm, and let τ be a task that undergoes S steals to form its kernel τ^K. Then τ^K incurs $O(Q(\tau^K) + S + \sum_i X(\tau_{u_i}))$ cache misses, including the costs of usurpations.*

Proof. The computation of τ^K comprises the execution of the task differences δ_i, for $1 \leq i \leq k$, plus the execution of the nodes on paths L_i and $\overline{L}_i$. There are a total of $2S$ nodes on the paths L_i and $\overline{L}_i$, and as each node performs $O(1)$ operations, their execution incurs $O(S)$ cache misses. By Lemma 3.5, the cost of executing the task differences is bounded by $O(Q(\tau^K) + \sum_i X(\tau_{u_i}))$. The result now follows. ■

Theorem 3.1. *Let $\mathcal{A}$ be a cache-regular BP algorithm and let Q be the number of cache misses incurred when executing $\mathcal{A}$ sequentially. Consider an execution of $\mathcal{A}$ which incurs $S \geq 1$ steals. Then there is a collection $\mathcal{C} = \{\nu_1, \nu_2, \cdots \nu_s\}$ of $s \leq 2S$ disjoint natural tasks such that the execution of $\mathcal{A}$ incurs at most $O(Q + S + \sum_i X(\nu_i))$ cache misses.*

Proof. Let $\tau_1, \tau_2, \cdots, \tau_{S+1}$ denote the original task and the S stolen tasks in the execution of $\mathcal{A}$. A collection $\mathcal{C}$ is constructed by applying Lemma 3.6 to each of the $k \leq S$ tasks τ_i that incur a steal, and adding tasks $\tau_{u_{i_j}}$, $1 \leq j \leq \sigma_i$ to the collection $\mathcal{C}$, where σ_i is the number of steals incurred by τ_i. In addition, for the remaining $S + 1 - k$ tasks τ_i that do not incur a steal, τ_i itself is added to $\mathcal{C}$. Since $k \geq 1$ and $\sum_i \sigma_i \leq S$, we obtain $|\mathcal{C}| \leq 2S$. ■

We can now apply the above theorem to obtain a nontrivial upper bound on the cache miss overhead in a parallel BP computation in which S steals occur, by bounding the maximum reload cost incurred by any collection of $O(S)$ disjoint natural tasks in $\mathcal{A}$.

4 HBP Analysis

As we will see shortly, the above analysis can be extended to type $t \geq 1$ HBP algorithms, leading to Theorem 1.1 and Theorem 1.2. Before proceeding to the proofs of these theorems, we illustrate how to apply Theorem 1.1 by analyzing the FFT algorithm described in [13, 8].

4.1 Analysis of FFT

The FFT algorithm views the input as a square matrix, which it transposes, then performs a sequence of two recursive FFT computations on independent parallel subproblems of size $\sqrt{n}$, and finally performs a matrix transpose (MT) on the result. This algorithm performs $W = O(n \log n)$ work,

has sequential cache complexity $Q = O(\frac{n}{B} \log_M n)$ [13]. and has critical path-length $O(\log n \cdot \log \log n)$.

The Type 2 HBP algorithm FFT, when called on an input of length n, makes a sequence of $c = 2$ calls to FFT on $v(n) = \sqrt{n}$ subproblems of size $r(n) = \sqrt{n}$ with a constant number of BP computations of MT performed before and after each collection of recursive calls. It has $f(r) = \sqrt{r}$.

Lemma 4.1. *The FFT algorithm incurs $O(\frac{n}{B} \log_M n + S \cdot \min\{M/B, B\} + \frac{n}{B} \frac{\log n}{\log[(n \log n)/S]})$ cache misses when it undergoes S steals.*

Proof. We apply Theorem 1.1. The sequential execution incurs $O((n/B) \log_M n)$ cache misses. $X(\nu_i) = \min\{M/B, |\nu_i|/B + f(|\nu_i|)\}$, and for FFT, $f(r) = O(\sqrt{r})$; thus if $|\nu_i| < B^2$ then $f(|\nu_i|) = O(B)$, and if $|\nu_i| \geq B^2$, then $f(|\nu_i|) = O(|\nu_i|/B)$; so $\sum_i f(|\nu_i|) = O(S \cdot \min\{M/B, B\} + \sum_i |\nu_i|/B)$.

It remains to bound the term $\sum_{\nu_i \in \mathcal{C}} |\nu_i|/B$. 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 $S = \Theta(\frac{n}{r} \log_r n)$, implies that $r \log r = \Theta(n \log n/S)$, so $\log r = \Theta(\log([n \log n/S])$. Thus $\max_{\mathcal{C}} \sum_{\nu_i \in \mathcal{C}} \frac{|\nu_i|}{B} = O(\frac{n}{B} \log_r n) = O(\frac{n}{B} \frac{\log n}{\log[(n \log n)/S]})$. ■

As shown in Section 6, this yields linear (optimal) speedup for $n \geq p \log \log n \cdot (M^\epsilon + \min\{M, B^2\} \log M)$ for any fixed $\epsilon > 0$. If $M > B^2$, this improves on the bound of $n \geq p(M/B)T_\infty / \log_M n$ in [1], which requires $n \geq p \log \log n \cdot M \log M$ for optimal speed-up.

4.2 Preliminaries

Before demonstrating Theorem 1.1 it will be helpful to refine the local and global variable terminology. We say that variable v is local w.r.t. task τ if v is declared by τ or by a subtask of τ; otherwise, if it is an already declared variable that is accessed during the computation of τ, variable v is global w.r.t. τ.

We begin with a lemma very similar to Lemma 3.2.

Lemma 4.2. *Let β be a block used to store variables accessed during the computation of τ^K as part of the sequential execution of HBP algorithm $\mathcal{A}$. During the computation of τ^K as part of execution Π, the variables in β accessed by τ^K are stored in at most 4 blocks.*

Proof. The proof is similar to that of Lemma 3.2. The difference is that we show a 1:4 block ratio rather than a 1:3 ratio. The reason for the changed ratio is that τ^K's global variables may be the local variables in a recursive call containing τ^K. Consequently, the variables accessed by τ^K and stored in a single block β used in the sequential execution may be contained in as many as 3 execution stacks in execution Π, one for the global variables, one for the variables defined by the parent of τ's start node, and one for the variables generated by τ^K. As shown in the proof of Lemma 3.2, if the contents of β are spread over k different execution stacks, then they are spread over at most $k+1$ blocks: the earliest generated stack that contains contents of β could have these contents spread across two differents block due to block misalignment, but for each of the remaining stacks, only an initial portion of the first block will contain contents from β. Thus, in an HBP algorithm the contents of β are spread across at most 4 blocks in a parallel execution. ■

Next, we specify the steal path, mentioned in [11], that separates a task kernel τ^K from the remainder of τ in an HBP algorithm.

Definition 4.1. *Let D be the series-parallel computation dag for a task τ. Let $\tau_1, \tau_2, \cdots, \tau_k$ be the sequence of successive subtasks stolen from τ in forming τ's kernel τ^K; let v_i be the start node for task τ_i and let w_i be the parent of v_i in D, for $1 \leq i \leq k$. Note that w_{i+1} is a proper descendant of w_i.*

The steal path *P_τ is defined to be the rightmost path in D starting at the root of D and connecting the nodes w_i, for $1 \leq i \leq k$.*

Observation 4.1. *Let $S_\tau = \{\tau_1, \tau_2, \cdots, \tau_k\}$ be the set of subtasks stolen from a task τ in forming task kernel τ^K, and consider the steal path P_τ as defined above. Then, S_τ consists of the subtasks that begin at those nodes of D that are the right child of a node on P_τ but are not themselves on P_τ.*

Proof. The steal path P_τ is build as follows. First the start node for τ is added to P_τ. The rest of the path is built as follows. Let v be the node currently at the top of τ's task queue. It will be the case that w, the parent of v, is the node most recently added to P_τ. If v is stolen then u, the sibling of v, is added to the steal path. Note that u is w's left child. While if v is not stolen then v is added to the steal path. Note that v is w's right child. It follows that P_τ is the rightmost path connecting the nodes w_i, the parents of the stolen nodes v_i. And clearly, the nodes at which stolen subtasks begin are exactly those right children of nodes on P_τ which are not themselves on P_τ. ■

4.2.1 Fork-Join Tasks

We start by defining a fork-join task.

Definition 4.2. *A* fork-join task *is a task as defined in Part 2 of a natural task: It is a natural task that begins with a fork tree and ends with the corresponding join tree.*

A fork-join task generalizes a BP computation since is a recursive HBP computation that is, in general, a nontrivial task instead of an $O(1)$ computation in a BP fork-tree.

Now we state lemmas very similar to Lemmas 3.4–3.6. These will concern fork-join tasks μ that incur steals in their fork trees. We use the same notation w_i, u_i, v_i, L_i, R_i, μ_i, and the task difference $\delta_i = \tau_{w_i} - \mu_i$, as for the BP computations (see beginning of Section 3.3.2 and Figure 1). There is one small change to the relevant definitions: if a steal occurs in a task corresponding to a leaf node of a fork tree, for the purposes of these definitions we will view this leaf node as being stolen even if it is a subtask within the leaf node that is stolen rather than the leaf itself.

The following lemmas deal with the effect of steals from the initial fork tree in a fork-join task. The general case, when steals can also occur within sub-tasks of a leaf task, is addressed in the next section, in Lemmas 4.7 and 4.8.

Lemmas 4.3 and 4.4 below are proved in the same way as Lemma 3.4 (except that now we use Lemma 4.2), and Lemma 3.5, respectively.

Lemma 4.3. *Let $\mathcal{A}$ be a cache-regular HBP algorithm, and let μ be a fork-join task in $\mathcal{A}$. Let Π be a computation of $\mathcal{A}$ which incurs $S \geq 1$ steals in the fork tree for μ. If the computation of μ^K in Π is usurpation-free, the number of cache misses incurred in computation Π when executing δ_i is bounded by $O(Q(\delta_i) + X(\tau_{u_i}))$, where u_i is the left child of w_i, and $X(\tau_{u_i})$ is the reload cost of τ_{u_i}.*

Lemma 4.4. *Let $\mathcal{A}$ be a cache-regular HBP algorithm. The cache misses incurred in executing the nodes on path $\overline{R}_i$ are bounded by $O(Q(\delta_i) + X(\tau_{u_i}))$, where u_i is the left child of w_i.*

For our final lemma, we need to define the notion of a *reduced* μ^K, $\mu^{\text{K-red}}$ for fork-join task μ. If all the steals forming μ^K occur in the fork tree for μ, $\mu^{\text{K-red}}$ is defined to be μ^K itself. Otherwise, (exactly) one of the steals forming μ^K must occur inside a leaf node of the fork tree (recall that the leaf nodes correspond to recursive computations). Let z be this leaf node. Then $\mu^{\text{K-red}}$ is defined to be $\mu^K - z$.

Lemma 4.5. *Let $\mathcal{A}$ be a cache-regular HBP algorithm, and let μ be a fork-join task in $\mathcal{A}$. Then $\mu^{K\text{-}red}$ incurs $O(Q(\mu^{K\text{-}red}) + S + \sum_i X(\tau_{u_i}))$ cache misses, including the costs of any usurpation.*

This is proved in the same way as Lemma 3.6, where the term $\sum_i X(\tau_{u_i})$ pays for the cache miss cost of re-accessing variables after a usurpation, and the $O(S)$ term pays for the cost of accessing join nodes in the fork tree after usurpations.

We also need to identify a particular *max-clean* subtask of steal-incurring fork-join tasks (see Figure 2.) Informally, the max-clean subtask is the same subtask as the task τ_{u_1} defined in Lemma 3.6 for a BP computation, except in the case that u_1 is a leaf node of the fork tree (necessarily the leftmost leaf). In the HBP setting, the task corresponding to leaf node u may be more than a single node computation, and so may incur steals, and consequently in this case we let the max-clean subtask be the empty subtask.

Definition 4.3. *Let μ be a fork-join task. Suppose that μ incurs one or more steals. Let F denote the fork tree in μ. Let u be the highest node in F, if any, on the leftmost path descending from F's root such that the following subtask μ_u is steal-free: the subtask beginning at node u and ending at the corresponding node in the join tree. If there is no such node u, we let μ_u denote an empty subtask. μ_u is called μ's* max-clean *subtask, and u is the* root of μ's max-clean task.

Informally, the root u of the max-clean task μ_u is the first node visited while executing μ that is the root of a steal-free task. By the structure of an HBP computation, this means that all steals

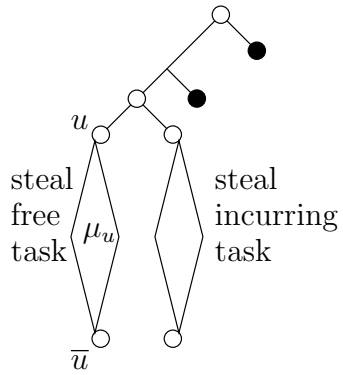

Figure 2: The max-clean subtask μ_u.

in this computation occur in the task μ_v rooted at the node v, which is the sibling of u in the fork-tree. As with task τ_{u_1} in the BP analysis, the re-load cost of μ_u will be used in our HBP analysis to account for the overhead of re-accessing variables after usurpations.

In general, there may be several steals in μ_v, the sibling task to the max-clean task μ_u, so there may a need to identify additional steal-free tasks to account for the cost of accesses after usurpations. Further, a fork-join task is only one component in an HBP computation. In the next sub-section we introduce the notions of constituent tasks and fragmentation to identify key subtasks that can be used to account for the cost of re-accessing variables after usurpations.

4.2.2 Constituent Tasks and Fragmentation

We recall the following constants relating to an HBP computation τ.

- d denotes the maximum number of HBP tasks that are sequenced by applying Rule 3 of Definition 2.2 in forming τ; note that $d = O(1)$.
- c denotes the maximum number of ordered collections of parallel recursive subproblems created by applying Rule 4 of Definition 2.2 in forming τ; recall that $c = O(1)$.

In the next three definitions we introduce the terms *constituent tasks* and *fragmentation*, the latter for both recursive HBP and for fork-join tasks.

Definition 4.4. *The constituent tasks in an HBP task τ consist of the following tasks:*
Case i. *τ is a sequenced HBP task (i.e. specified by Rule 3 in Definition 2.2).*
Recall that a sequenced type t HBP task τ is formed by sequencing at most d (non-sequenced) HBP tasks of type t or lower. These up to d tasks comprise the constituent *subtasks of τ.*
Case ii. *τ is a recursive HBP task (i.e. specified by Rule 4 in Definition 2.2).*
Recall that a recursive type t task τ is formed by sequencing up to $2c + 1$ subtasks. There are two sorts of subtasks: up to $c + 1$ lower type sequenced HBP computations, which we call lower-type *subtasks; and up to c collections of recursive type t computations, with each collection being implemented as a fork-join task. In the case that a parallel collection contains a single recursive task, the fork and join trees are trivial 1-node trees.*

The constituent *subtasks of this HBP task consist of the up to c type t fork-join subtasks, one for each collection of recursive calls, and for each of the up to $c+1$ sequenced HBP computations μ, the up to d subtasks being sequenced to form μ. In all, a recursive HBP task can have up to $c + (c+1) \cdot d$ constituent subtasks.*

Note that each constituent subtask is either a sequential computation of constant size (type 0), a single BP computation (Rule 2 in Definition 2.1) or a collection of recursive HBP computations (Rule 4 in Definition 2.2). It is not a sequenced computation, since each individual HBP task that occurs in any of the up to $c+1$ sequenced HBP tasks in a recursive HBP task τ is identified as a separate constituent task.

Definition 4.5. *Let τ be a sequenced or recursive HBP task which contains at least one steal. The process of* fragmenting *τ is to form the collection of constituent subtasks of τ, and to identify within this collection the steal-free constituent subtasks in τ as the* fragments of *τ.*

Recall that every steal occurs at a right child in a fork tree (where we view a binary fork as a special case of a fork tree). Hence, we also use the fragmentation terminology w.r.t. a fork tree in which steals occur. This will specify essentially the same tasks as in Lemma 4.5.

Definition 4.6. *Let μ be a fork-join task and let F be its fork tree. Suppose that one or more steals occur at nodes of F. As for BP computations, we trace the steal path separating the steal-free portion of the fork tree. (see Figure 3.)*

Let w_i be the nodes at which the path switches from descending to the left to descending to the right. Specifically, let $w_1, w_2, \cdots, w_k$ be the sequence of nodes in F for which w_i has a steal-free left subtree rooted at u_i and a steal-incurring right subtree rooted at v_i, for $1 \leq i \leq k$ (and w_{i+1} is a node in this right subtree, for $1 \leq i < k$). Then the fragmentation of μ comprises the tasks $\mu_i = \tau_{u_i}$ that begin at node u_i and ends at the corresponding node in the join tree, for $1 \leq i \leq k$, except that u_k is not included if it is a leaf node of the fork tree and this leaf node corresponds to a multiple-node computation dag, and this dag incurs one or more steals; in this case, the fragmentation comprises the tasks μ_i for $1 \leq i < k$.

It is not difficult to see that we will create $O(S)$ fragments when we fragment a computation with S steals. (This is shown in Lemma 4.8 in the next section.)

4.2.3 Overview of HBP Analysis

In the next two sections, we bound the cache miss overhead for HBP computations. The analysis for HBP computation is performed inductively, using the structure of HBP computations as defined in Definition 2.2. Additionally, we need to account for the fact that in the up-pass that follows a join from a steal, the bound of Lemma 3.5 for BP computations no longer applies. Rather than simply accessing $O(\log |\tau|) = O(|\tau|)$ local variables in the up-pass, τ may create additional local variables (that are placed on its execution stack), and following the join ending the steal of a subtask, it may seek to re-access these variables. If these variables fit in cache, then if there were no steal, no cache misses would be incurred in accessing these variables. However, in the event of a usurpation, the re-accesses could cause additional cache misses that need to be included when computing the cache miss overhead.

In Section 4.3 we establish a cache miss bound that is essentially identical to the BP case for standard HBP. The key property we establish in a standard HBP is that the cache miss overhead

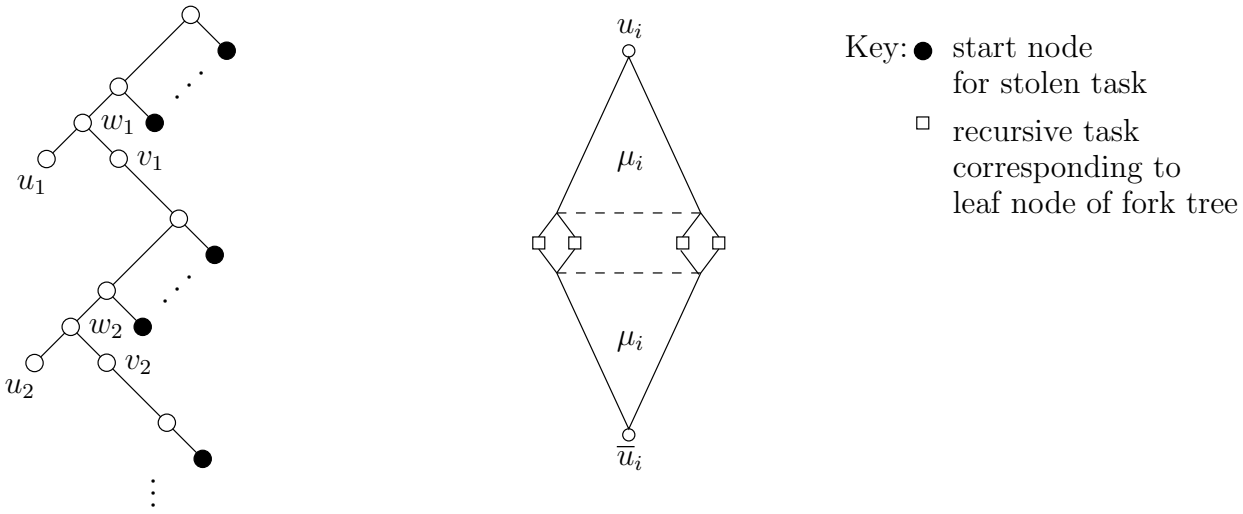

Figure 3: The path P.

incurred in the up-pass in the task kernel, following the join performed by a stolen task, including the cost of accessing local variables on a remote execution stack following a usurpation, is bounded by the reload cost of a max-clean task in the task kernel. We are thus able to recover the bound we established for BP computations (to within a constant factor). With the exception of LCS and I-GEP, all of the specific HBP algorithms we consider in Section 5 are standard. In Section 4.4 we bound the cache miss overhead for HBP that use super-linear space, but have an exponential bound on the number of recursive calls they make, and we establish that the overhead in this case incurs only a modest increase in the $O(S)$ term to $O(S \log B)$. For technical reasons, we also defer the analysis of linear space HBP that have collections of size 1 (i.e., $v(n) = 1$), as in I-GEP and LCS, to Section 4.4; here the only difference in the final result is that we need to replace the reload cost $X(\tau) = |\tau|/B + f(|\tau|)$ by $X'(\tau) = |\tau|/B + \sum_{i \geq 0} f(\alpha^i |\tau|)$, where α is the geometrically decreasing factor in the sizes of tasks in the fork tree.

4.3 Standard HBP

Our analysis will deal primarily with a *recursive* type t HBP. It is then a simple matter to extend the result to a general type t HBP, which is simply as sequence of $O(1)$ HBP of type at most t. In this subsection we consider standard HBP.

We begin with a structural lemma concerning the distribution of steals in a recursive type t HBP algorithm.

Lemma 4.6. *Let μ be a type t recursive task in an HBP algorithm. Suppose that every steal μ incurs occurs in a type t fork-join subtask μ_R of μ, and suppose that there is no smaller type t fork-join subtask for which this property holds. Further suppose that there are no steals at internal nodes of μ_R's fork tree. Then the steals occur only in a recursive type t computation $\overline{\nu}$ corresponding to the rightmost leaf of μ_R's fork tree. In addition, either the steals occur in at least two of $\overline{\nu}$'s constituent subtasks, or all the steals occur in a single lower type constituent subtask of $\overline{\nu}$.*

Proof. The fact that all the steals occur in the rightmost leaf of μ_R's fork tree follows immediately from Observation 4.1. If the steals all occurred in a single type t constituent subtask ν of μ_R, then ν would be a smaller type t fork-join task containing all the steals, contradicting the definition of μ_R. Thus the steals must either lie in two or more of $\overline{\nu}$'s constituent subtasks, or be contained in a single lower type constituent subtask. ■

Next, we give a lemma that identifies a set of natural subtasks that cover the cache miss costs of the execution of task kernel τ^K for the main type of task we consider: a recursive task in a standard HBP algorithm. Recall the $C(\mu)$ is a bound on the number of cache misses incurred by a task fragment μ in a parallel execution.

Lemma 4.7. *Let $\mathcal{A}$ be a standard cache-regular HBP algorithm. Let τ be a type t recursive task in $\mathcal{A}$ with $v(|\tau|) > 1$, and let μ be one of its constituent fork-join subtasks.*

Let μ_R be a type t subtask of μ (perhaps $\mu_R = \mu$) such that every steal incurred by μ occurs in μ_R, and there is no smaller type t subtask for which this property holds.

Then $C(\mu^K)$, the cache misses incurred in executing μ^K, including all costs due to usurpations, is bounded as follows.

1. *Suppose that every steal incurred by μ occurs in the fork tree F_R for μ_R. Then $C(\mu^K)$ is bounded by a constant times:*
   i. $Q(\mu^K)$, plus
   ii. $X(\mu_u)$, where μ_u is μ's max-clean subtask, plus
   iii. $O(S)$, where S is the number of steals μ_R incurs in forming μ_R^K, plus
   iv. $\sum_i X(\mu_i)$, where the μ_i are the tasks created by fragmenting μ_R.

2. *Suppose the steals in μ occur in both the fork tree for μ_R and in a recursive type t computation ν corresponding to a leaf of the fork tree. Then $C(\mu^K)$ is bounded by a constant times:*
   i. $Q(\mu^K)$, plus
   ii. $X(\mu_u)$, where μ_u is μ's max-clean subtask, plus
   iii. $C(\nu^K)$, plus
   iv. $O(S')$, where S' is the number of steals μ_R incurs in forming μ_R^K, plus
   v. $\sum_i X(\mu_i)$, where the μ_i are the tasks created by fragmenting μ_R.

3. *Suppose that every steal μ incurs occurs only in a recursive type t computation $\overline{\nu}$ corresponding to a leaf of the fork tree μ_R. If the steals occur in at least two of $\overline{\nu}$'s constituent subtasks, let $\nu = \overline{\nu}$. Otherwise, if all the steals occur in a lower type constituent subtask of $\overline{\nu}$, let ν denote this constituent subtask. Then $C(\mu^K)$ is bounded by a constant times:*
   i. $Q(\mu^K)$, plus
   ii. $X(\mu_u)$, where μ_u is μ's max-clean subtask, plus
   iii. $C(\nu^K)$.

The above cases cover all the ways in which the steals μ incurs can be distributed.

Proof. We begin by proving (1). The arguments for (2) and (3) are very similar.

We will analyze for the case when every possible usurpation of μ^K occurs, since this will maximize $C(\mu^K)$.

By Lemma 4.5, the cache misses incurred in executing μ_R^{K-red} are bounded by $O(Q(\mu_R^{K-red}) + \sum_i X(\mu_i) + S)$. If $\mu \neq \mu_R$, there may be additional cache misses in the computation of $\mu^K - \mu_R^{K-red}$.

We bound these as follows. Before μ_R's execution, the following units are executed: the root node w of μ, μ's max-clean subtask μ_u which begins at w's left child u, and an initial portion μ_v^{Init} of μ_v, the natural task beginning at node u's sibling v. As in the proof of Lemma 3.4, this computation, being steal-free, incurs $O(Q(w \cup \mu_u \cup \mu_v^{\text{Init}}) + 1 + X(\mu_u) + X(\mu_v))$ cache misses, which is bounded by $O(Q(w \cup \mu_u \cup \mu_v^{\text{Init}}) + X(\mu_u))$.

Following μ_R's execution, the following units are executed: the portion μ_v^{Final} of μ_v that follows the completion of μ_R plus the node $\overline{w}$ which is the complement of w. Since we assume that every possible usurpation occurs, executing $\overline{w}$ incurs $O(1)$ additional cache misses. Executing μ_v^{Final} incurs $O(Q(\mu_v^{\text{Final}}) + X(\mu_v))$ cache misses plus the cache misses for the first accesses to the blocks storing already declared local variables w.r.t. μ_w on the execution stack for τ (if these blocks are not in cache due to usurpation). By the linear space bound, these variables use space $O(|\mu_w|)$ and further they are contiguous on the execution stack used by μ^K. Thus the first accesses to the blocks storing the already declared local variables incur $O(|\mu_u|/B + |\mu_v|/B + 1) = O(X(\mu_u))$ cache misses. In all, these units incur $O(1 + Q(\mu_v^{\text{Final}}) + X(\mu_v) + X(\mu_u)) = O(Q(\mu_v^{\text{Final}}) + X(\mu_u))$ cache misses.

This is a total of $O(Q(\mu^{K-red}) + Q(w \cup \mu_u \cup \mu_v^{\text{Init}}) + Q(\mu_v^{\text{Final}}) + X(\mu_u) + \sum_i X(\mu_i) + S) = O(Q(\mu^K) + X(\mu_u) + \sum_i X(\mu_i) + S)$ cache misses, which proves (1).

The only change in proving (2) is to note that the cache miss cost for computing ν^K is separated out, and S', the number of steals in the fork tree, replaces S. (3) is similar, except that here $S' = 0$.

Finally, to see that this covers all the possibilities, we argue as follows. If there are steals in μ_R's fork tree this is covered by Cases (1) or (2). Otherwise, Lemma 4.6 applies, from which it follows that the options specified in Case (3) suffice. ■

Now we can bound the number of tasks that need to be added to $\mathcal{C}$. In the following, we let $\rho = cd + c + d$, the maximum number of subtasks generated in the fragmentation of a recursive task.

Lemma 4.8. *Let $\mathcal{A}$ be a standard cache-regular HBP algorithm. Let τ be a type t task in $\mathcal{A}$ which incurs $S \geq 1$ steals. Then, there is a collection $\mathcal{C}$ of at most $3 \cdot \rho \cdot t \cdot S = O(S)$ disjoint natural tasks in τ^K such that executing τ^K starting with an empty stack incurs $O(Q + S + \sum_{\tau_i \in \mathcal{C}} X(\tau_i))$ cache misses.*

Proof. We prove the result by induction on t.

Base case. $t = 1$. Then, τ is a sequence of at most d BP computations, Then, by Lemma 4.5, there is a collection of at most $d - 1 + S \leq 3\rho S$ disjoint natural tasks that satisfy the requirements of the lemma, where the $d - 1$ term bounds the steal-free BP computations in τ, which are added to $\mathcal{C}$, and S accounts for the tasks added to $\mathcal{C}$ as in the proof of Lemma 3.5.

Induction Step. $t \geq 2$. Let S_1 of the S steals be of type t tasks in type t fork trees in τ, and so the remaining $S_2 = S - S_1$ steals are of tasks of type smaller than t.

By Lemma 4.7, the following collection of tasks suffice to include all tasks that need to be in $\mathcal{C}$:

Rule 1 Fragment τ and add the resulting steal-free subtasks to $\mathcal{C}$.

Rule 2 For each steal-incurring constituent subtask μ of τ, proceed as follows:

a If μ is a fork-join task of the same type as τ, add the task μ_u and the tasks μ_i specified by Lemma 4.7 to $\mathcal{C}$, and handle the task ν (if it exists) recursively (by treating ν as τ).

b If μ is a lower-type task, handle it recursively.

We now count the number of tasks this procedure adds to $\mathcal{C}$.

Let T be a fork tree initiating a type t fork-join task, and suppose that T incurs $s \geq 1$ steals of its nodes (so this does not count steals that occur within a task at a leaf node of tree T). Rule 1 adds at most $\rho - 1$ steal-free tasks to $\mathcal{C}$ in the fragmentation that isolates the task containing tree T; Rule 2 then adds one max-clean task, plus one task per steal obtained through the fragmentation of T. This adds up to a total of at most $\rho + s \leq 2s\rho$ tasks. Across all S_1 steals from such trees T in fork-join tasks this is a total of at most $2\rho S_1$ tasks.

Additionally, we need to account for steals that occur within leaf tasks in type t fork-join trees. Here, as noted in Case 3 of Lemma 4.7, either:

i. There are two or more steal-incurring constituent subtasks of leaf task $\overline{\nu}$. This can occur at most $S - 1$ times (for these $S - 1$ occurrences would generate S distinct steal-incurring tasks). Note that each occurrence is associated with a distinct task being fragmented by Rule 1, namely either τ itself or a task being handled recursively. Each occurrence adds one more task to $\mathcal{C}$, namely the max-clean task of μ, and together with the Rule 1 charge, yields a total charge of at most $\rho \cdot (S - 1)$ tasks.

ii. $\overline{\nu}$ has a single steal-incurring constituent subtask ν, which is of lower type than τ. If ν incurs s' steals, then by the inductive assumption, the number of tasks that need to be added to $\mathcal{C}$ is no more than $3\rho(t-1)s'$. So, across all lower type tasks incurring steals we add no more than $3\rho(t-1)S_2$ tasks to $\mathcal{C}$, and together with the Rule 1 charge this yields a total charge of at most $\rho - 1 + 3\rho(t-1)S_2$ tasks to $\mathcal{C}$.

Hence $|\mathcal{C}| \leq 2\rho S_1 + \rho \cdot (S-1) + (\rho - 1) + 3\rho(t-1)S_2 \leq 3\rho S + 3\rho(t-1)S_2 \leq 3\rho t S$, since $t \geq 2$ and $S = S_1 + S_2$. ■

We are now ready to prove Theorem 1.1.

Proof. (of Theorem 1.1.) For each τ which is either a stolen task or the original task, apply Lemma 4.8 to add tasks to $\mathcal{C}$; in addition, for each steal-free stolen task τ, add τ to $\mathcal{C}$. This is a total of $O(S)$ tasks added to $\mathcal{C}$. Again, by applying Lemma 4.8, we see that executing $\mathcal{A}$ incurs $O(Q + S + \sum_i X(\nu_i))$ cache misses. ■

4.4 Extensions: Superlinear Space and Size One Collections

When the linear space bound no longer applies, we need to revisit the bound on the cost of reaccessing local variables in the event of a usurpation. The reason is that for a task τ fragmented as in Lemma 4.7, the cost of accessing local variables defined by the nodes in the fork trees may grow. Recall that the execution of a type t task τ following a usurpation will include the completion of a sequence of one or more join trees, and in each join tree this computation entails executing a path of nodes. It turns out that the cost in cache misses for these accesses is bounded by the number of these join trees, plus $1/B$ times the length of the paths in these join trees. We will establish that with the modest restriction that there are at most an exponential number of recursive calls (a restriction clearly satisfied by any polynomial- or exponential-time algorithm), the first term is bounded by $O(\log |\tau|) = O(|\tau|/B + \log B)$, and the second term is bounded by $O(|\tau|/B)$. Since the $|\tau|/B$ terms are absorbed by the $Q(\tau) + X(\tau)$ terms that appear in the cache miss bound, an

additional $\log B$ cost arises once per application of the new Lemma 4.12, analogous to Lemma 4.7, which is at most once per steal. The net effect is to increase the bound on the number of cache misses by just an additive $O(S \cdot \log B)$, so long as every fork-join task runs in at most exponential time.

The accesses to local variables following a usurpation also impose additional costs for accessing variables declared by recursive tasks. Consider the case that $v(n) > 1$. Let $\tau_1, \tau_2, \cdots, \tau_k$ be the sequence of recursive type t tasks called on the path from μ to μ_R, i.e. for $1 \leq i < k$, τ_i calls τ_{i+1}, and τ_k contains μ_R as a constituent subtask. Each of the τ_i may declare a superlinear number of variables and we need to bound the number of cache misses due to accesses to these variables during the computation following the completion of μ^R. Previously, with the linear space bound, we could assert that the total space used by the variables declared by all the τ_i was $O(|\tau_u|)$, where τ_u is the max-clean task in μ, which lead to the bound $O(Q(\tau^K)+X(\mu_u))$. While $O(Q(\tau^K)+X(\mu_u))$ bounds the number of blocks storing the variables declared by τ which are accessed by μ_u, for these are accesses that are global w.r.t. μ_u, in general, it does not provide a bound on the number of blocks storing variables declared by $\tau_1, \tau_2, \cdots, \tau_k$ and which are accessed by μ_u, for these accesses are local w.r.t. μ_u. When $v(n) = 1$, we may have no suitable max-clean task, thus even the linear-space case needs a new analysis. It turns out that the above approach using the τ_i can be used to handle $v(n) = 1$ tasks, both for linear- and super-linear space bounds.

This alternate analysis to bound the cost of accesses to local variables again increases the costs; the algorithms we consider that are covered by this case are those that contain tasks with $v(n) = 1$ (LCS, I-GEP) and this increases the bound on the number of extra cache misses by an additive $O(S \cdot \log B)$ term as for the case of accesses to nodes on fork-join trees following usurpations, though through a different analysis.

For this analysis, we need to associate a new worst case cost $X'(|\nu|)$ with natural tasks ν, which we call the *extended reload cost*. It is defined as follows: $X'(|\nu|) = \min\{\frac{M}{B},\ \frac{|\nu|}{B} + \sum_{i \geq 0} f(\alpha^i |\nu|)\}$. Recall that α is the factor by which the sizes of recursive problems decrease — see Definition 2.2.

Comment. For $f(r) = \Theta(\sqrt{r})$, $X'(|\nu|) = \min\{\frac{M}{B}, |\nu|/B + f(|\nu|))\} = O(X(\nu))$, and for $f(r) = O(1)$, $X'(\nu) = \min\{\frac{M}{B}, \frac{|\nu|}{B} + \log|\nu|) = \min\{\frac{M}{B}, O(\frac{|\nu|}{B} + \log B))\} = \min\{\frac{M}{B}, O(X(\nu) + \log B)\}$. These are the cases that arise in our algorithms. But for other functions f, other bounds on X' may apply. For example, if $f(r) = \log^2 r$, then (ignoring the first term M/B), we have $X'(\nu) = O(|\nu|/B + \log^2|\nu|) = O(|\nu|/B + \log^2 B)$.

In our analysis, we use the extended reload cost of suitable tasks to bound the cost of accessing local variables after usurpations.

We need one more modest constraint on the algorithms we consider, namely that any fork-join tree has linear height, or equivalently, the algorithm makes at most *exponentially* many recursive calls.

Definition 4.7. *Let $\mathcal{A}$ be an HBP algorithm. $\mathcal{A}$ is said to be* exponentially bounded *if every task in $\mathcal{A}$ runs in time at most exponential in its size.*

The following lemma is immediate.

Lemma 4.9. *If $\mathcal{A}$ is exponentially bounded then for every recursive task τ the number of recursive calls $v(|\tau|)$ that it makes satisfies* $\log v(|\tau|) = O(|\tau|)$.

Proof. τ must run in time exponential in its size and hence $v(|\tau|) = 2^{O(|\tau|)}$; it follows that $\log v(|\tau|) = O(|\tau|)$. ■

Lemmas 4.3–4.5 continue to hold; however, Lemma 4.4 needs a new justification. Previously, we had observed (in the proof of Lemma 3.5) that $|R_i| = O(1 + \log|\tau_{u_i}|) = O(|\tau_{u_i}|)$. Here, we observe that $|R_i|$ is bounded by the height of the fork tree which begins at τ_{u_i}'s start node, and as $\mathcal{A}$ is exponentially bounded, this height is $O((|\tau_{u_i}|)$, so here too $|R_i| = O(|\tau_{u_i}|)$.

The next two lemmas identify what changes from Lemma 4.7 with superlinear space and with size one collections. Before proceeding with the lemmas we state a useful observation used in the proof of Lemma 4.10.

Observation 4.2. *For any $x, B \geq 1$, $\log x = O(x/B + \log B)$.*

Proof. To see this note that if $x \geq B \log B$ then $x/\log x \geq B$, hence $\log x \leq x/B$, while if $x \leq B \log B$, $\log x = O(\log B)$. ■

Lemma 4.10. *Let $\mathcal{A}$ be a cache-regular HBP algorithm that is exponentially bounded. Let τ be a type t recursive task in $\mathcal{A}$ and let μ be one of its constituent fork-join subtasks containing type t recursive sub-tasks.*

Let μ_R be a type t fork-join subtask of μ such that every steal incurred by μ occurs in μ_R, and there is no smaller type t fork-join subtask for which this property holds.

If $\mu \neq \mu_R$, let τ_1 be the type t recursive call corresponding to the rightmost leaf in μ's fork tree (i.e. the recursive call on the path to μ_R). Then, in the execution of $\mu^K - \mu_R^K$, the cache misses incurred in accessing local variables w.r.t. μ, including the costs of usurpations, are bounded by

$$O(Q(\mu^K - \mu_R^K) + X'(\mu_u) + \sum_i X'(\kappa_{1j}) + \min\{M/B, \log B\})$$

where μ_u is μ's max-clean subtask, and the κ_{1j} are the tasks created by fragmenting τ_1.

Proof. We begin with some useful notation.

Let E be the execution stack used in τ's execution. E is either τ's execution stack if τ was stolen or is the original task in the algorithm; if not, E is the execution stack for an ancestor of τ.

Let $\tau_1, \tau_2, \cdots, \tau_k$ be the sequence of successive recursive type t calls, such that, (i) τ_1 corresponds to the rightmost leaf of μ's fork tree, (ii) for $1 \leq i < k$, τ_i calls τ_{i+1}, and (iii) τ_K calls μ_R. These are the open (i.e. started but not completed) recursive calls inside of μ at the time μ_R^K completes. Note that the sizes of the τ_i decrease geometrically as i increases (by the HBP definition). Let $\mu = \mu_1, \mu_2, \cdots, \mu_k$ be the sequence of type t fork-join tasks such that τ_i corresponds to the rightmost leaf node in the fork tree for μ_i. For $1 \leq i < k$, let κ_{ij}, $1 \leq j \leq l$, be the constituent subtasks of τ_i, excluding fork-join task μ_{i+1}. Finally, let P_i be the rightmost path of nodes in the fork tree for μ_i, starting at the root and ending at the parent of the rightmost leaf.

We turn to the analysis.

If there are no usurpations, the cost of the cache misses is bounded by $O(Q(\mu^K - \mu_R^K)$. So henceforth, we focus on the case that there is a usurpation; this will affect only the cost of the computation following the completion of μ_R^K.

Additional cache misses, over and above those accounted for by the term $O(Q(\mu^K - \mu_R^K))$, occur due to accesses to variables already on the execution stack when the computation of μ_R completes. These variables are either the variables declared by the join nodes corresponding to the fork nodes on the steal path from μ to μ_R or the variables declared by the recursive type t procedures $\tau_1, \tau_2, \cdots, \tau_k$. We will bound the additional costs under usurpations due to both categories of variables. Recall that this additional cost is always bounded by $O(M/B)$.

The difficulty we face in bounding the accesses to the variables declared by the recursive type t subtasks is that they are to some or all of the variables in possibly superlinear sized sets (relative to the sizes of the tasks declaring these variables). However, note that any access to such a variable that causes a cache miss whose cost is not covered by the Q term, is an access by some subtask ν to a variable that is *global* w.r.t. ν. Thus these additional costs can be bounded by a sum of terms $X(\nu)$ for suitable ν. The core of the analysis lies in identifying the ν for which a term $X(\nu)$ needs to be included in the overall cost.

Following the completion of μ_R^K, the variables declared by $\tau_1, \tau_2, \cdots, \tau_{k-1}$ can be accessed only by the tasks μ_i and κ_{ij}. Thus the cost of these accesses is bounded by

$$O\left(Q(\mu^K) + \min\left\{M/B, \sum_i \left[\lceil |\mu_i|/B \rceil + f(|\mu_i|)\right] + \sum_{i,j} \left[\lceil |\kappa_{ij}|/B \rceil + f(|\kappa_{ij}|)\right]\right\}\right).$$

Since the sizes of the successive τ_i shrink geometrically (as i increases), so do the sizes of the successive μ_i and of the successive κ_{ij}, for each j (by the definition of HBP tasks, Definition 2.2). Thus $\sum_{1 \le i < k} \lceil |\kappa_{ij}|/B \rceil = O(|\kappa_{1j}|/B + \log |\kappa_{1j}|)$, and $\sum_{1 \le i \le k} \lceil |\mu_i|/B \rceil = O(|\mu_1|/B + \log |\mu_1|)$. Applying Observation 4.2 yields bounds of $O(|\kappa_{1j}|/B + \log B)$ and $O(|\mu_1|/B + \log B)$, respectively. This yields an overall bound of

$$\begin{aligned} & O\left(Q(\mu^K) + \min\{M/B, \log B\} + |\mu_1|/B + \sum_i f(|\mu_i|) + \sum_j \left[|\kappa_{1j}|/B + \sum_i [f(|\kappa_{ij}|)\right]\right) \\ = \; & O\left(Q(\mu^K) + \min\{M/B, \log B\} + X'(\mu_1) + \sum_j X'(\kappa_{1j})\right). \end{aligned}$$

Finally, $X'(\mu_1) = O(1 + X'(\mu_u) + X'(\mu_v)) = O(X'(\mu_u))$, so the overall bound becomes $O(Q(\mu^K) + \min\{M/B, \log B\} + X'(\mu_u) + \sum_j X'(\kappa_{1j}))$.

As $\mathcal{A}$ is exponentially bounded, there are $O(|\mu_i|)$ nodes on the path portion P_i. Each node on P_i declares $O(1)$ variables, and further the variables declared by these nodes are contiguous on E. Thus they occupy $O(\lceil |\mu_i|/B \rceil)$ blocks. Clearly, the cost of these accesses is covered by the bound in the previous paragraph. ■

We now extend the result in Lemma 4.10 (which bounded the cost of accesses to local variables in $\mu^K - \mu_R^K$) to establish the same bound for all cache misses incurred in executing $\mu^K - \mu_R^K$.

Lemma 4.11. *Let $\mathcal{A}$ be a cache-regular HBP algorithm that is exponentially bounded, and let τ, μ, μ_R, τ_1, μ_u and the κ_{1j} be as in Lemma 4.10. Then $C(\mu^K - \mu_R^K)$, the cache misses incurred in executing $\mu^K - \mu_R^K$, including all costs due to usurpations, is bounded by*

$$Q(\mu^K - \mu_R^K) + X'(\mu_u) + \sum_i X'(\kappa_{1j}) + O(\min\{M/B, \log B\})$$

Proof. The computation of $\mu^K - \mu_R^K$ consists of the initial steal-free portion until the start of μ_R, and then the computation following μ_R^K.

The computation until the start of μ_R incurs no more cache misses than $Q(\mu^K - \mu_R^K)$.

The computation following μ_R^K comprises a portion of μ_v^K plus the final node of μ. The final node of μ incurs $O(1)$ cache misses. The computation of the portion of μ_v^K following μ_R^K performs two types of accesses: accesses to global variables and to local variables w.r.t. μ. To bound the global accesses, note that they are all to global variables of μ_v, where v is the sibling of u, the root of the max-clean task, and hence the reload cost they induce is $O(Q(\mu^K - \mu_R^K) + X(\mu_v)) = O(Q(\mu^K - \mu_R^K) + X(\mu_u)) = O(Q(\mu^K - \mu_R^K) + X'(\mu_u))$. Finally, Lemma 4.10 bounds the costs of the local accesses by the same bound we need to establish.

Since each of the costs is no more than the bound we need to establish, the result follows. ■

We now state a modified version of Lemma 4.7 which applies without the linear-space and $v(n) > 1$ restrictions.

Lemma 4.12. *Let $\mathcal{A}$ be a cache-regular HBP algorithm that is exponentially bounded and let τ, μ, μ_R, τ_1, μ_u and the κ_{1j} be as in Lemma 4.10. Then $C(\mu^K)$, the cache misses incurred in executing μ^K, including all costs due to usurpations, is bounded as follows.*

1. *Suppose that every steal incurred by μ occurs in the fork tree F_R for μ_R. Then $C(\mu^K)$, the cache misses incurred in executing μ^K, is bounded by a constant times:*
   i. $Q(\mu^K)$, plus
   ii. $X'(\mu_u)$, where μ_u is μ's max-clean subtask, plus
   iii. $O(S)$, where S is the number of steals μ_R incurs in forming μ_R^K, plus
   iv. $\sum_i X'(\mu_i)$, where the μ_i are the tasks created by fragmenting μ_R, plus
   vi. $O(\min\{M/B, \log B\})$.

2. *Suppose the steals in μ occur in both the fork tree for μ_R and in a recursive type t computation ν corresponding to a leaf of the fork tree. Then $C(\mu^K)$ is bounded by a constant times:*
   i. $Q(\mu^K)$, plus
   ii. $X'(\mu_u)$, where μ_u is μ's max-clean subtask, plus
   iii. $C(\nu^K)$, plus
   iv. $O(S')$, where S' is the number of steals μ_R incurs in forming μ_R^K, plus
   v. $\sum_i X'(\mu_i)$, where the μ_i are the tasks created by fragmenting μ_R, plus
   vi. $\sum_i X'(\kappa_{1j})$, where the κ_{1j} are the tasks created by fragmenting τ_1, plus
   vii. $O(\min\{M/B, \log B\})$.

3. *Suppose that every steal μ incurs occurs only in a recursive type t computation $\overline{\nu}$ corresponding to a leaf of the fork tree μ_R. If the steals occur in at least two of $\overline{\nu}$'s constituent subtasks, let $\nu = \overline{\nu}$. Otherwise, if all the steals occur in a lower type constituent subtask of $\overline{\nu}$, let ν denote this constituent subtask. Then $C(\mu^K)$ is bounded by a constant times:*
   i. $Q(\mu^K)$, plus
   ii. $X'(\mu_u)$, where μ_u is μ's max-clean subtask, plus
   iii. $C(\nu^K)$, plus
   iv. $\sum_i X'(\kappa_{1j})$, where the κ_{1j} are the tasks created by fragmenting τ_1, plus
   v. $O(\min\{M/B, \log B\})$.

The above cases cover all the ways in which the steals μ incurs can be distributed.

Proof. The costs can be partitioned into the costs of executing $\mu^K - \mu_R^K$, which are bounded by Lemma 4.11, and the costs of executing μ_R^K which is bounded in the same way as in Lemma 4.7, except that we replace X by X' in inductive terms. The above bounds follow immediately. ■

Lemma 4.13. *Suppose the assumptions in Lemma 4.12 apply. Suppose that both τ and τ_1 are fragmented. Then for each pair of corresponding constituent subtasks κ_i and κ_{1i} in τ and τ_1 respectively, if κ_i is added to $\mathcal{C}$ there is no need to add κ_{1i}.*

Proof. $X(\kappa_{1i}) = O(X(\kappa_i))$, hence $X(\kappa_{1i}) + X(\kappa_i) = O(X(\kappa_i))$. So the cache miss costs accounted for by $X(\kappa_{1i})$ can be charged to the term $O(X(\kappa_i))$. ■

The next lemma follows by essentially the same proof as Lemma 4.8.

Lemma 4.14. *Let $\mathcal{A}$ be a cache-regular HBP algorithm that is exponentially bounded. Let τ be a type t HBP task in $\mathcal{A}$ which incurs $S \geq 1$ steals. Then, there is a collection $\mathcal{C}$ of $O(c{\cdot}d{\cdot}t{\cdot}S) = O(S{\cdot}t)$ natural tasks in τ^K such that executing τ^K incurs $O(Q + S \cdot \log B + \sum_{\tau_i \in \mathcal{C}} X'(\tau_i))$ cache misses, plus the cost of local accesses w.r.t. P by the nodes on each usurped path P.*

Proof. There are three changes to the proof of Lemma 4.8. First, we apply Lemma 4.12 instead of Lemma 4.7, which causes the terms $X(\tau_i))$ to be replaced by $X'(\tau_i))$. Second, we have to account for the fragmentations of τ_1 that occur when Lemma 4.12 is applied. However, by Lemma 4.13, each fragmentation of a τ_1 together with its parent τ can be controlled so as to add at most $\rho - 1$ tasks to $\mathcal{C}$, the bound that was being used in Lemma 4.8 for the number of tasks added to $\mathcal{C}$ by the fragmentation of τ. So this does not affect the overall bound. Third, each application of Lemma 4.12 adds an additional cost of $\log B$, and this can occur $O(St)$ times, which increases the term $O(S)$ in Lemma 4.8 to $O(S \log B)$ here. ■

We can now establish Theorem 1.2.

Proof. (of Theorem 1.2.) The argument is the same as for Theorem 1.1 except that now we apply Lemma 4.14 instead of Lemma 4.8. ■

5 Analysis of Specific Algorithms Under RWS Scheduling

Recall that, as shown in [1, 10], when an algorithm is scheduled on p processors using RWS, $S = O(p \cdot T_\infty)$ w.h.p. in n, where T_∞ is the depth of the computation dag for the given algorithm. Henceforth, we will omit the phrase w.h.p. in n. Also recall from Lemma 3.1 in Section 3.1 that the reload cost of a task τ is is $X(\tau) = O(\min\{M/B, |\tau|/B + f(|\tau|)\})$. (We will deal with X' later when we consider I-GEP and CS.)

Finally, we recall the definition of the *bit interleaved (BI) layout*: it recursively places the elements in the top-left quadrant, followed by recursively placing the top-right, bottom-left, and bottom-right quadrants. The advantage of the BI layout is that it results in BP tasks that have an $O(1)$ cache friendliness function.

For completeness we also describe BP algorithms that convert between the BI and RM formats. For the three MM algorithms, I-GEP and FFT, if their input is in the RM format, it is advantageous

to first convert to BI format and at the end convert back to RM format. We will address this explicitly for the depth n MM algorithm.

5.1 Scans, Matrix Transpose (MT) in BI format

For all BP algorithms, including Scans and Matrix Transpose, $T_\infty = O(\log n)$, hence $S = O(p \cdot \log n)$.

We apply the bound from Theorem 1.1. For Scans and Matrix Transpose (MT) algorithms in BI format, $f(|\tau|) = O(1)$. So with S steals, for scans the worst case cost for $\sum_i X(\nu_i) = O(\frac{n}{B} + S)$, as $Q = O(n/B)$ and $\sum_i |\nu_i| = O(n)$ since the ν_i access disjoint sets of data. This is $O(\frac{n}{B} + p \cdot \log n)$ cache misses. Similarly, for MT, which has an input of size n^2, there are $O(\frac{n^2}{B} + S) = O(\frac{n^2}{B} + p \cdot \log n)$ cache misses. When in RM format, $f(|\tau|) = O(\sqrt{|\tau|})$ and then, since $\frac{n^2}{B} \geq n$ when $n \geq B$, there are $O(\frac{n^2}{B} + S \cdot \min\{M/B, B\}) = O(\frac{n^2}{B} + p \cdot \min\{M/B, B\} \cdot \log n)$ cache misses.

5.2 RM to BI and BI to RM algorithms

These are both BP algorithms in which all the computation (reads and writes) occurs at the leaves, which are ordered according to the BI format. Thus to covert from RM to BI, the reads will have a cache friendliness function $f(r^2) = r$ (for accessing a size $r \times r$ array in RM format causes $r^2/B + r$ cache misses), while the writes, to an array in BI format, will have a cache friendliness function $f(r^2) = O(1)$. The situation is reversed for the BI to RM computation. In both cases there are $O(\frac{n^2}{B} + S \cdot \min\{M/B, B\}) = O(\frac{n^2}{B} + p \cdot \min\{M/B, B\} \cdot \log n)$ cache misses.

5.3 Matrix Multiply

Depth n MM. The algorithm we consider, from [13], recursively multiplies an initial four pairs of $n/2 \times n/2$ matrices, recording the results, and then multiplies a second collection of four pairs, adding the second set of results to the results from the initial multiplications. So $c = 2$ and $r(n^2) = n^2/4$. Also this algorithm has $W = O(n^3)$, sequential cache miss cost $Q = O(n^3/(B\sqrt{M}))$, and $T_\infty = O(n)$, so $S = O(p \cdot n)$.

We now demonstrate the same cache miss bound for depth-n-MM as obtained by [14], even after taking account of accesses to hidden variables.

Lemma 5.1. *When it undergoes S steals, the depth n MM algorithm incurs $O(n^3/(BM^{1/2}) + S^{1/3}\frac{n^2}{B}) = O(n^3/(BM^{1/2}) + (p \cdot n)^{1/3}\frac{n^2}{B}$ cache misses plus $O(S) = O(p \cdot n)$ when using BI format, and plus $O(S \cdot \min\{M/B, B\}) = O(p \cdot \min\{M/B, B\} \cdot n)$ when using RM format.*

Proof. We apply the bound from Theorem 1.1. We need to consider both MM and MA (Matrix Addition) subtasks. If BI format is being used, $f(r) = O(1)$ and if RM format is used, $f(r) = O(\sqrt{r})$. Thus the term $\max_{\mathcal{C}} \sum_{\nu_i \in \mathcal{C}} f(|\nu_i|)$ where $|\mathcal{C}| = O(S)$ yields $O(S)$ cache misses for the BI format and $O(S \cdot B)$ for the RM format.

Next, we bound the term $\max_{\mathcal{C}} \sum_{\nu_i \in \mathcal{C}} \frac{|\nu_i|}{B}$. We start by bounding the number of available MM subtasks of each size. For each integer $i \geq 0$, there are $O\left(8^i \left(\frac{n}{\sqrt{M}}\right)^3\right)$ subtasks of size $M/4^i$. They cause $m_i = O\left(\left(2^i \frac{M}{B} + 8^i\right) \cdot \left(\frac{n}{\sqrt{M}}\right)^3\right)$ cache misses. Summing from $i = 0$ to some fixed j we get a total of $O\left(8^j \left(\frac{n}{\sqrt{M}}\right)^3\right)$ cache misses, i.e., the total cost is dominated by cache misses for tasks of

size $M/4^j$. Now we determine the smallest value of j for which there can be $|\mathcal{C}|$ tasks. As $|\mathcal{C}| \leq c' \cdot S$ for some constant $c' \geq 1$, we need j to be the smallest index that yields $c'S = O\left(8^j \left(\frac{n}{\sqrt{M}}\right)^3\right)$. Substituting $i = j$, this gives a total of $m_j = O(S^{1/3}\frac{n^2}{B} + S)$ cache misses.

We bound the number of MA tasks similarly. There are $\sum_{0 \leq i \leq \log n/r} \frac{2^i \cdot n^2}{r^2} \leq 2\frac{n^3}{r^3}$ MA tasks of size r^2; equivalently, there are $O\left(8^i \left(\frac{n}{\sqrt{M}}\right)^3\right)$ tasks of size $M/4^i$, so the same bounds apply as for the MM subtasks.

The bounds in terms of p and n follow by substituting $S = O(p \cdot n)$. ■

If the matrix is in RM format, but we use the RM to BI and BI to RM algorithms to provide BI format input to the depth n MM algorithm, the cache miss cost becomes: $O(n^3/(BM^{1/2}) + (p \cdot n)^{1/3}\frac{n^2}{B} + pn)$ for the MM algorithm itself and $O(\frac{n^2}{B} + p \cdot B \cdot \log n)$ for the conversion. So long as $n^2 \geq M$ (i.e. the input does not fit in one cache) this is $O(n^3/(BM^{1/2}) + (p \cdot n)^{1/3}\frac{n^2}{B} + pn + p \cdot B \cdot \log n)$ cache misses.

Depth $\log^2 n$ MM and Strassen's MM. The depth $\log^2 n$ algorithm multiplies two $n \times n$ matrices by recursively multiplying eight $n/2 \times n/2$ matrices, followed by a tree (BP) computation to add pairs of these recursively multiplied matrices. This algorithm has $c = 1$, $r(n^2) = n^2/4$, $T_\infty = O(\log^2 n)$, $W = O(n^3)$, and $Q = O(n^3/(B\sqrt{M}))$ cache misses in a sequential computation.

Strassen's algorithm recursively multiplies seven $n/2 \times n/2$ matrices, followed by a BP computation to combine these results. This algorithm has $c = 1$, $r(n^2) = n^2/4$, $T_\infty = O(\log^2 n)$, $W = O(n^\lambda)$ with $\lambda = \log_2 7$, and $Q = O(n^3/(B\sqrt{M}))$.

The exact same analysis as for depth-n MM applies to both algorithms, yielding the same bound as in Lemma 5.1 (for Strassen's algorithm, n^λ replaces n^3). Note that the values of S are much smaller for these algorithms, namely $S = O(p \log^2 n)$, so in fact these are improved bounds (the term $(pn)^{1/3}n^2/B$ is replaced by $(p \log^2 n)^{1/3}n^2/B)$). However, note that the depth $\log^2 n$ MM algorithm uses space $O(p^{1/3}n^2)$, which is larger than the $O(n^2)$ space usage of the depth n MM algorithm.

5.4 SPMS Sort, List and Graph Algorithms

The SPMS sorting algorithm has the same structure as the FFT algorithm analyzed in Section 4.1 and has the same performance bounds ($W = O(n \log n)$ work, $Q = O(\frac{n}{B} \log_M n)$ sequential cache complexity, and $T_\infty = O(\log n \cdot \log \log n)$). For details on SPMS, see [9].

However, to obtain this bound, we need to broaden the definition of HBP algorithms, for the recursive subproblems may vary significantly in size. In fact, it suffices, in the BP and HBP definitions, that sizes decrease by a constant factor in $O(1)$ levels rather than a single level as in Definitions 2.1 and 2.2. Further it is not necessary that the sizes of the subproblems be equal up to constant factors. We showed in [9] how to construct a fork tree for arbitrary sized tasks on the fly in time $O(\log v(n))$ while achieving the constant factor size decrease in $O(1)$ levels. The sorting algorithm also employs virtual sizes, which are upper bounds on the actual sizes, to achieve virtual, rather than actual, balance; but this too suffices to carry out the analysis.

The list ranking algorithm (LR) described in [11] has $Q = O(\frac{n}{B} \log_M n)$ sequential cache complexity, $W = O(n \log n)$ (which is optimal for the cache miss bound), and $T_\infty = O(\log^2 n \cdot \log \log n)$. It begins with $\log \log n$ phases which reduce the list length by an $O(\log n)$ factor. In the

ith phase there are $k \geq 2$ sorts of size $O(n/2^i)$ followed by a sequence of $\log^{(k)}$ sorts of combined size $O(n/2^i)$, where $k \geq 2$ is a constant. So the number of these sorts is $O(\log\log n \cdot \log^{(k)} n) = O((\log\log n)^2)$. The remaining phase, on a list of size $O(n/\log n)$, uses the standard parallel algorithm. Each step of this algorithm requires a sort of size $O(n/\log n)$. Putting these together, we see that the LR algorithm comprises a sequence of $O(\log n)$ sorts of combined size $O(n)$. The algorithm also uses $O(1)$ BP computations for each sort, but each BP computation is less expensive than the corresponding sort, so it suffices to bound the cost of the sorts.

If the ith sorting problem has size n_i and incurs S_i faults, we conclude that the bound on cache misses is $O(Q + S \cdot B + \sum_i \frac{n_i}{B} \frac{\log n_i}{\log[((n_i \log n_i)/S_i]}) = O(Q + S \cdot B + \sum_i \frac{S_i}{B} \frac{(n_i \log n_i)/S_i}{\log[((n_i \log n_i)/S_i]})$ and as $x/\log x$ is a concave function, we conclude that this sum is maximized when $n_i \log n_i / S_i = n'/S = O(n/S)$ where $S = \sum_i S_i$ and $n' = \sum_i n_i$. This immediately yields a bound of $O(Q + S \cdot B + \frac{n}{B} \frac{\log n}{\log[((n \log n)/S]})$ cache misses. This bound is the same as that for SPMS sort, but $S = O(p \cdot \log^2 n \cdot \log\log n)$, which is a factor of $\log n$ larger than the bound for SPMS sort.

Likewise, the most expensive part of the connected components algorithm is $O(\log n)$ iterations of LR [11], which yields a bound of $O(Q + S \cdot B + \frac{n+m}{B} \frac{\log^2 n}{\log[((n+m)\log n)/S]}$ cache misses, where n in the number of vertices and m the number of edges in the graph.

5.5 I-GEP and LCS

Before entering into the algorithm specific analyses, recall that the extended task reload cost for a task τ is $X'(|\nu|) = |\nu|/B + \sum_{i \geq 0} f(\alpha^i |\nu|)$, where α is the factor by which the sizes of recursive problems decrease.

If the cache friendliness function $f(r) = O(1)$, $\sum_i f(|\nu_i|) = O(S)$, and if $f(r) = O(\sqrt{r})$, $\sum_i f(|\nu_i|) = O(S \cdot B)$. The later bound arises because when $|\nu_i| \geq B^2$, $|\nu_i|/B \geq f((|\nu_i|)$, and so we only have to account for the f term for steals of size less than B^2.

I-GEP. I-GEP [5] is a recursive cache-oblivious implementation of GEP, the *Gaussian Elimination Paradigm*, and a multithreaded parallel implementation is presented and analyzed in [6]. I-GEP is a template that solves several important problems including Gaussian elimination without pivoting and Floyd-Warshall's all-pairs shortest paths, and a degenerate form of I-GEP reduces to depth-n-MM. In the sequential context I-GEP has the same complexity as depth-n-MM, with $O(n^3)$ running time and $O(n^3/(B\sqrt{M})$ sequential cache complexity. Its multithreaded critical path length is slightly larger, it is $\Theta(n \log^2 n)$ (though this can be reduced to $\Theta(n)$ if the scheduler is allowed to tailor its steals to tasks of size n^2/p, which is not an option under RWS [6]). When the input matrix is in BI format, $f(r) = O(1)$, and in RM format, $f(r) = \sqrt{r}$.

When given as input a square matrix of size n^2, I-GEP calls itself recursively $c = 2$ times on one subproblem of size $n^2/4$. In between the two recursive calls is a call to a parallel collection of two Type 3 HBP, followed by a sequence of two calls to parallel collections of two Type 2 HBP (depth-n-MM), and then another call in sequence to a parallel collection of two Type 3 HBP, with each call in the parallel collections being to an input of size $n^2/4$. Thus I-GEP is a Type 4 HBP, since the sequence of 4 calls to HBP of Types 2 and 3 satisfy the requirement of having $O(1)$ calls to lower type HBP.

As shown in [5], there are $\Theta(8^i)$ tasks of size $n^2/4^i$ generated in an I-GEP computation, which is the same as in depth-n-MM, to within a constant factor. Hence, essentially the same analysis as for the three MM algorithms applies for bounding the cache misses as a function of the steals, except that we are applying Theorem 1.2 here. We obtain the following bound on the number of

cache misses incurred due to steals: $O(S^{1/3}\frac{n^2}{B})$ plus $O(S \cdot \min\{M/B, B\})$ if using RM and plus $O(S \log B)$ if using BI. For I-GEP, the number of steals is $S = O(p \cdot n \log^2 n)$, hence as a function of p and n, the number of cache misses is $O(pn \log^2 n \cdot \min\{M/B, B\} + (pn \log^2 n)^{1/3} \cdot \frac{n^2}{B})$ (RM case), and $O(pn \log^2 n \log B + (pn \log^2 n)^{1/3} \cdot \frac{n^2}{B})$ (BI case). These are similar to the bound in [14] (but [14] had some cost omissions).

Longest Common Subsequence (LCS). The sequential cache oblivious algorithm for computing the LCS of two input strings of length n using an $O(n^2)$ time dynamic programming algorithm [5] runs in $O(n)$ space with $O(\frac{n^2}{MB})$ cache misses. The algorithm consists of a forward pass that makes a sequence of $c = 3$ calls to collections of recursive subproblems of size $n/2$ (the first and third calls are to one subproblem each, while the second call is to a collection of two subproblems). Thus the forward pass is a Type 2 HBP with critical pathlength $T_{1,\infty}(n) = O(n^{\log_2 3})$. This computes the length of an LCS. To compute an actual LCS on a size n input, the overall algorithm calls a sequence of 4 calls to the Type 2 forward pass on inputs of size $n/2$, followed by one recursive call to a collection of three parallel subproblems of size $n/2$ [7]. Thus the overall algorithm to compute an LCS is a Type 3 HBP which continues to have $T_\infty(n) = O(n^{\log_2 3})$. Again, we will apply Theorem 1.2. There are $\Theta(4^i)$ subproblems of size $n/2^i$, and since the cache friendliness function is $f(r) = O(1)$, it follows that the total extended reload cost is $O\left(4^j \cdot \min\left\{M/B, \lceil \frac{n}{2^j \cdot B} \rceil + \log B\right\}\right)$ cache misses overall, where j is defined by $\sum_{i=0}^{j} 4^i = \Theta(S)$. Hence the total extended reload cost is $O(\min\{S \cdot M/B, \frac{n}{B}\sqrt{S} + S \cdot \log B\})$.

The number of steals $S = O(p \cdot n^{\log_2 3})$ w.h.p. under RWS. Thus the cache misses incurred due to steals under RWS are bounded by $O(\min\{M/B \cdot p \cdot n^{\log_2 3}, \frac{1}{B}\sqrt{p} \cdot n^{1+\frac{1}{2}\log_2 3} + p \cdot n^{\log_2 3} \log B\})$; modulo a $\log B$ factor, this is the same as the result in [14], (which had some cost omissions). We note that the result in [14] is obtained only for the computation of the LCS length, and not for the computation of the LCS itself.

6 Optimal Speed Up Under RWS Scheduling

We finish by determining for each algorithm, for a given number p of processors, what size of n is required to achieve optimal (linear) speedup compared to the cost of a sequential execution of the algorithm. These bounds are shown in Table 2. A very strong result would be a constraint $n \geq pM$, namely that the problem size n just needs to exceed the space available in the p caches. As can be seen by inspecting the bounds in Table 2, our bounds are just a little larger: depending on the particular algorithm, larger by factors of $\log n$, $\sqrt{M}$, $\log \log n \cdot \log M$, and $\sqrt{M} \log^2 n$.

In comparision to earlier results, we improve on the bounds obtained with [1] for BP computations (the first two rows of the table), and for FFT, SPMS Sort, List Ranking and Graph Connected Components (the last 3 rows of the table). For the remaining algorithms, which are the three matrix multiplication algorithms, I-GEP and LCS, we match the bounds obtained with [1]. These algorithms, with the exception of MM and Strassen, are also analyzed in [14], and we obtain exactly the same dependence on M as $M \to \infty$ as in [14].

6.1 Scans

The cache miss delay in a sequential execution is $O(n/B)$. Linear speedup is achieved when $p \cdot \log n = O(n/B)$, i.e. $n \geq p \cdot B \cdot \log n$.

HBP Algorithm	**Constraint Derived from Analysis in [1]**	**New Constraints for Optimal Performance**
Scans	$n \geq p \cdot M \log n$	$\mathbf{n \geq p \cdot B \log n}$
MT: BI format RM format	$n \geq p \cdot M \log n$	$\mathbf{n^2 \geq p \cdot B \log n}$ $\mathbf{n^2 \geq p \cdot \min\{M, B^2\} \log n}$
Depth-n-MM: BI or RM	$n^2 \geq p(M)^{3/2}$	$n^2 \geq p(M)^{3/2}$
MM: BI or RM format	$n^3 \geq pM^{3/2} \log^2 n$	$n^3 \geq pM^{3/2} \log^2 n$
Strassen: BI or RM ($\lambda = \log_2 7$)	$n^\lambda \geq pM^{3/2} \log^2 n$	$n^\lambda \geq pM^{3/2} \log^2 n$
I-GEP: BI or RM	$n^2 \geq pM^{3/2} \log^2 n$	$n^2 \geq pM^{3/2} \log^2 n$
LCS	$n^{1-\frac{1}{2}\log_2 3} \geq p^{\frac{1}{2}} M$	$n^{1-\frac{1}{2}\log_2 3} \geq p^{\frac{1}{2}} M$
FFT, SPMS Sort	$n \geq p \cdot M \log\log n \log M$ $= n_{\text{sort}-[1]}$	$\mathbf{n \geq p \log\log n[M^\epsilon + \min\{M, B^2\} \log M]}$ $\mathbf{= n_{sort-new}}$
List Ranking (LR)	$n \geq n_{\text{sort}-[1]} \cdot \log n$	$\mathbf{n \geq n_{sort-new} \cdot \log n}$
Conn. Cpts. (CC) $n = \|V\| + \|E\|$	$n \geq n_{\text{sort}-[1]} \cdot \log^2 n$	$\mathbf{n \geq n_{sort-new} \cdot \log^2 n}$

Table 2: Input size n (n^2 for matrix computations) needed for optimal cache misses under RWS. An entry is in bold in the last column when our result improves on the result in column 2.

6.2 MT

In the case that the matrices are in BI format, The analysis is the same as for scans except that the problem size is n^2 rather than n. This yields the constraint $n^2 \geq p \cdot B \cdot \log n$. For RM format, the constraint becomes $n^2 \geq p \cdot B^2 \cdot \log n$.

6.3 Depth n MM

The sequential execution incurs $n^3/(B\sqrt{M})$ cache misses. Thus, for RM format, the constraint on n is:

$$S^{\frac{1}{3}} \frac{n^2}{B} + S \cdot \min\left\{\frac{M}{B}, B\right\} \leq \frac{n^3}{B\sqrt{M}}.$$

This simplifies to $S \cdot M^{3/2} \leq n^3$, which yields the constraint $n^2 \geq p \cdot M^{3/2}$, as $S = O(p \cdot n)$. In BI format, the constraint on S is

$$S^{\frac{1}{3}} \frac{n^2}{B} + S \leq \frac{n^3}{B\sqrt{M}},$$

which yields the same bound on n. We achieve the same improved trade-offs as mentioned in [14] since our bound for the cache miss overhead is the same.

6.4 8-way MM

As for depth n MM, for RM or BI format, we obtain $S \cdot M^{3/2} \leq n^3$. This yields $n^3 \geq pM^{3/2} \log^2 n$ as $S = O(p \cdot \log^2 n)$.

6.5 Strassen

The analysis is as for 8-way MM, except that n^λ replaces n^3. In fact, a slightly better bound is possible; we leave this to the interested reader.

6.6 Sort, FFT

For optimality we need that $\log[(n \log n)/S] = \Omega(\log M)$, i.e. that $S = O([n \log n]/M^\epsilon)$ for some constant $\epsilon > 0$. We also need $S \cdot \min\{M/B, B\} \leq \frac{n}{B} \log_M n$. Now $S = O(p \cdot \log n \log\log n)$. This yields the constraint $p \cdot \log n \log\log n(M^\epsilon + \min\{M, B^2\} \log M) \leq n \log n$, i.e. $n \geq p \cdot \log\log n(M^\epsilon + \min\{M, B^2\} \log M)$.

6.7 List Ranking (LR)

As the cache miss costs increase by at most a $\log n$ factor compared to sorting, whereas they are unchanged in a sequential execution, the bounds on optimality increase by a $\log n$ factor compared to the sorting algorithm.

6.8 Connected Components (CC)

As all the costs increase by a $\log n$ factor over those for LR, CC is optimal for the same range of n as LR.

6.9 I-GEP

In RM format, the bounds on S are the same as for depth n MM. The one difference is that $S = O(p \cdot n \log^2 n)$ rather than $O(n)$, yielding the constraint $n^2 \geq p \log^2 n \cdot M^{3/2}$; as for depth n MM, the same bound holds for the BI format.

6.10 LCS

The sequential execution incurs $n^2/(MB)$ cache misses. Thus the constraint on n is: $\min\{M/B \cdot p \cdot n^{\log_2 3}, \frac{1}{B}\sqrt{p} \cdot n^{1+\frac{1}{2}\log_2 3} + p \cdot n^{\log_2 3} \log B\} \leq n^2/(MB)$, which simplifies to $n^{1-\frac{1}{2}\log_2 3} \geq p^{\frac{1}{2}} M$.

Acknowledgements. We thank the reviewers of our conference submission for their thoughtful suggestions.

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002. Springer.

[2] R. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, pages 720–748, 1999.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuzmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, 1995.

[4] F. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 187–194, 1981.

[5] R. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proc. of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '06, pages 591–600, 2006.

[6] R. Chowdhury and V. Ramachandran. The cache-oblivious Gaussian Elimination Paradigm: Theoretical framework, parallelization and experimental evaluation. *Theory of Comput. Syst.*, 47(1):878–919, 2010.

[7] R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proc. of the Twentieth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 207–216, 2008.

[8] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *Proc. 2010 IEEE International Symposium on Parallel & Distributed Processing*, IPDPS '10, pages 1–12, 2010.

[9] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. In *Proc. of the Thirty Seventh International Colloquium on Automata, Languages and Programming*, ICALP'10, pages 226–237. Springer-Verlag, 2010.

[10] R. Cole and V. Ramachandran. Analysis of randomized work stealing with false sharing. *CoRR*, abs/1103.4142, 2011.

[11] R. Cole and V. Ramachandran. Efficient resource oblivious algorithms for multicores with false sharing. In *Proc. IEEE IPDPS*, 2012.

[12] T. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. MIT Press, 2009.

[13] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1):4:1–4:22, Jan. 2012.

[14] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. *Theory Comput Syst*, 45:203–233, 2009.

[15] T. Gautier, X. Besseron, and L. Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proc. International Workshop on Parallel Symbolic Computation*, PASCO '07, pages 15–23, 2007.

[16] R. H. J. Halstead. Implementation of Multilistp: Lisp on a multiprocessor. In *Proc. ACM Symposium on LISP and Functional Programming*, pages 9–17, 1984.

[17] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in tbb. In *Proc. IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '08, pages 1–8, 2008.

[18] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2), 1985.