Analysis of Randomized Work Stealing with False Sharing

Richard Cole
Computer Science Dept.
Courant Institute of Mathematical Sciences, NYU
New York, NY 10012, USA
Email: cole@cs.nyu.edu

Vijaya Ramachandran
Dept. of Computer Science
University of Texas at Austin
Austin, TX 78712, USA
Email: vlr@cs.utexas.edu

***Abstract*—This paper analyzes the overhead due to false sharing when parallel tasks are scheduled using randomized work stealing (RWS). We obtain high-probability bounds on the cache miss overhead, including the overhead due to false sharing, for several parallel cache-efficient algorithms when scheduled using RWS. These include algorithms for fundamental problems, such as matrix computations, FFT, sorting, basic dynamic programming, list ranking and graph connected components. Our main technical contribution, from which these results follow, is the derivation of nontrivial high-probability bounds on the number of steals incurred by these algorithms in the presence of false sharing, when using RWS.**

***Keywords*-Randomized work stealing; false sharing; performance analysis;**

I. Introduction

Work-stealing is a longstanding technique for distributing work among a collection of processors [1], [2], [3]. Work-stealing operates by organizing work in tasks, with each processor managing its currently assigned tasks. Whenever a processor C becomes idle, it selects another processor C' and is given (it *steals*) some of C''s available tasks. A natural way of selecting C' is for C to choose it uniformly at random from among the other processors. To emphasize the inherent randomization in this process, we call it randomized work stealing, RWS for short. RWS is a simple, lightweight, distributed scheduler; to our knowledge there is no similarly effective deterministic scheduler. It has been implemented multiple times, including in Cilk [4], Intel TBB [5] and KAAPI [6].

RWS scheduling has been analyzed and shown to provide provably good parallel speed-up for a fairly general class of algorithms [3]. Its cache miss overhead for private caches was considered in [7], which gave some general bounds on this overhead; these bounds were improved in [8] for a class of computations whose cache complexity function can be bounded by a concave function of the operation count, and further improved for a larger class of algorithms in [9].

These analyses assume that there is no false sharing. False sharing refers to the risk of creating an apparent inconsistency when two different processors seek to access distinct locations in the same block: when one or both performs a write, this may create unnecessary overhead due to methods in place for preventing or managing potential inconsistencies. There are various methods, such as cache coherence, that are used in practice to avoid memory inconsistencies and that also prevent false sharing; in general, these methods can reduce the possible parallelism and potentially increase algorithm runtime.

In this paper, we analyze the efficiency of algorithms when scheduled using RWS, taking account of delays due to false sharing. Past work has assumed that false sharing is prevented at the level of the operating system, e.g. by means of the Backer protocol, as implemented in some versions of Cilk [4]. Another approach is to engineer the memory layout to avoid false sharing. While potentially reasonable for high-performance computing, this is less appealing for general purpose programming.

Instead of assuming that false sharing is prevented through other means, in this paper we obtain bounds on the overhead due to false sharing as a function of the worst-case delay due to a single fs miss. We leverage an algorithmic approach for controlling the costs of fs misses, encapsulated in the class of *block-resilient Hierarchical Balanced Parallel (HBP)* algorithms [10]. These algorithms are highly cache-efficient with good parallelism, and also have a relatively low overhead for false sharing. This class includes Sample, Partition, and Merge Sort [11], SPMS for short, standard divide and conquer algorithms for matrix multiply and FFT, basic dynamic programming such as LCS, and some list and graph algorithms (other than SPMS, these algorithms are at most modest modifications of already known algorithms). It is shown in [10] that these algorithms have a false sharing cost of only $O(B \cdot S)$ cache misses, where B is the size of a block and S is the number of steals. However, there still remains the task of bounding the number of steals under RWS when false sharing could be present, and this is the task we address in this paper.

Our main contribution is the derivation of nontrivial high probability upper bounds on the number of steals in the presence of false sharing, when using RWS. As mentioned above, the cache and fs miss overhead for a variety of algorithms was bounded by a function of the number of steals in [10]. By combining our new bounds on the number of steals under RWS with the results in [10], we obtain

nontrivial bounds on the cache and fs miss overhead of these algorithms when using RWS. It should be noted that the RWS scheduler remains unchanged; our contribution lies in the analysis that bounds the number of steals in the presence of false sharing, when using the standard RWS scheduler.

A. Computation Model

We use a directed acyclic graph, or dag, D, to model the computation induced by a program. (Good overviews of this approach can be found in [12], [3].) 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, where the terminal node of G_1 is connected to the start node of G_2.
ii. A parallel construct (binary forking), with 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.

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).

In RWS, 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.

Cache and False Sharing Misses: We distinguish between two types of cache-related costs, as discussed in [10].

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 was evicted from C's cache to make room for new data. This is the standard type of cache miss that is accounted for in sequential cache complexity analysis and in earlier work on RWS [7], [8].

False sharing of a block β occurs when two or more processors access different locations in β, with at least one of them performing a write. More specifically, suppose a processor C' updates an entry in block that is in processor C's cache. If cache coherency is supported [13], this results in block β being invalidated, and results in processor C needing to read in block β the next time it accesses data in this block. This is done so that data consistency is maintained within the elements of a block across all copies in caches at all times. This type of 'cache miss' does not occur in a sequential computation, and we refer to it as an *fs miss*. There are other ways of dealing with fs misses, but we believe that the fs miss cost with our invalidation rule based on cache coherence is likely as high as (or higher than) that incurred by other mechanisms. Thus, our upper bounds should hold for most of the coping mechanisms known for handling fs misses.

A series of fs misses could occur when two processors share a block and the block repeatedly ping-pongs between their caches as they alternate accesses. It could also arise when multiple processors each executing small tasks all share the same block, and the block then rotates from one processor's cache to another. We do not make any assumptions about the protocol used to transfer a block from one processor to another when several processors need to access data within that block. Our one assumption is that a processor cannot unduly delay transferring a requested block. For specificity, we assume that this delay is of the same magnitude as the cache miss delay. Thus an fs miss incurs a delay of at least one cache miss, but without further assumptions, as shown in [10], it can also be unboundedly expensive. In this paper, we assume that the computation scheduled by RWS has an upper bound on the worst case cost for an fs miss, achieved, for instance, by using the algorithmic techniques developed in [10], [11], and we measure the cost of fs misses in units of cache miss cost.

B. Our Results

An Example: We illustrate the potential impact of fs misses with an example. Consider the standard Depth-n matrix multiply (Depth-n-MM) algorithm (e.g., [14], [8], [15], [10]). This recursive algorithm, given two $n \times n$ matrices to multiply, makes, in succession, two collections of parallel calls each to 4 subproblems of size $n/2 \times n/2$, with the results being accumulated in a single output $n \times n$ array; the sets of subproblems are chosen so that their writes are disjoint. We will suppose the algorithm uses the Bit

Interleaved (BI) format (as opposed to Row Major) as this minimizes fs misses[1].

Consider a parallel execution of this algorithm on p processors with an "ideal" scheduler, i.e. with each processor executing a sequence of size $(n/\sqrt{p})\times(n/\sqrt{p})$ subproblems. Each processor updates a disjoint output matrix of size n^2/p, and due to the BI format, it shares just at most blocks with other processors. But a processor will update each entry in the portion of the output matrix assigned to it $n/\sqrt{p}$ times as it performs its computation, and under an adverse asynchronous execution, each of its updates to the shared block may entail false sharing with the other processor that shares this block. Across the p processors, this could result in $n \cdot B \cdot \sqrt{p}$ transfers of blocks due to false sharing. The entire computation performs a sequence of $\sqrt{p}$ parallel executions of this type, resulting in a worst-case fs miss bound of $n \cdot B \cdot p$ cache misses.

But with an RWS scheduler it is unclear if even this bound can be achieved, as the sizes of the allocated tasks are now likely to vary and so the above analysis no longer suffices. In fact, under adversarial stealing and an adversarial cache coherence protocol, one could create a single false sharing miss which incurs a delay of $\Theta(n)$ cache misses.

The approach used in prior work for RWS (that considered only the cost of cache misses and no false sharing) was to to bound the longest time duration t of any path in the computation dag D: this bound can then be used to bound the expected number of successful steals by $O(pt/s)$, where p is the number of processors and s is the time taken by a successful steal [3], [7]. To bound t in the presence of false sharing requires a bound on the possible delay due to fs misses faced by a single computation step. But given the high worst-case cost of $\Omega(n)$ cache misses for a single false sharing miss in the standard Depth-n-MM algorithm, this approach gives an unacceptably high bound for t and does not lead to reasonable bounds for the number of steals under RWS.

In contrast to the above setting, it is shown in [10] that a small modification to the standard Depth-n-MM algorithm that enforces 'limited writes' results in a *block-resilient* version that can reduce the overhead of false-sharing to the cost of $O(B)$ cache misses *per task*, and hence also to at most a cost of $O(B)$ cache misses per fs miss. As shown in [10] this results in a worst-case fs miss cost for the block-resilient Depth-n-MM of only the cost of $O(p^{3/2} \cdot B)$ cache misses when using an ideal scheduler, which is a significant improvement over the $n \cdot B \cdot p$ bound for the standard Depth-n-MM.

We do not expect to match the bounds achieved with an ideal scheduler when using RWS, given the distributed and randomized nature of RWS. On the other hand, the results presented in this paper allow us to obtain nontrivial upper bounds on the overhead due to false sharing under RWS for any series-parallel computation dag, and improved bounds for block-resilient computations. In particular, for block-resilient Depth-n-MM, we are able to obtain, with high probability in n, a bound of $S = O(n \cdot p \cdot \sqrt{B})$ on the number of steals under RWS. This in turn implies, a high probability bound of $O(n \cdot p \cdot B^{3/2})$ cache miss cost for fs misses in the block-resilient version of Depth-n-MM. Further, in the absence of a bound on the number of steals in the presence of false sharing, even the earlier bounds for cache miss overhead are not valid, since they bounded the largest delay in a computation path without taking into account the delays due to false sharing. With our high probability bound on S, and using the results in [9], we are also able to obtain a high probability bound of $O(S^{1/3} \cdot (n^2/B) + S)$ cache miss overhead (outside of the cost due to false sharing) due to steals.

End of example

We assume a standard memory allocation in which blocks are allocated and deallocated as the computation proceeds. Let $\Gamma(D, B)$, Γ for short, be an upper bound on the number of transfers of any single block β from one cache to another during a single allocation of β in the execution of computation dag D. We use Γ and the following parameters to specify our results: We suppose that an operation on in-cache data takes $O(1)$ time units, that the cost of a cache miss is $O(b)$ time units, that the cost for a successful steal of a task is $\Theta(s)$ time units, and the cost for an unsuccessful steal is $O(s)$ time units[2]. We will assume that $s \geq b$, since a successful steal would incur a delay of at least one cache miss. Finally, let T_∞ be the maximum number of nodes on any path descending D, and let n be the size of the input plus output.

We prove two bounds on the number of steals that occur in our algorithms when scheduled under RWS. We begin with a result that bounds the number of steals in a general series-parallel dag, the setting analyzed in [7], but when false sharing is present with a bound of Γ on the number of transfers of any single block from one cache to another.

Theorem I.1. *Let D be a series-parallel computation dag scheduled by RWS. Then, with high probability in n (i.e. for any fixed $c > 0$, with probability $1 - 1/n^c$), the computation of D incurs at most the following number S of successful steals:*

$$S = O\left(p \cdot T_\infty \cdot (1 + \frac{b}{s}\Gamma(D, B))\right).$$

In addition, with high probability in n, the time spent on all steals, successful and unsuccessful, is $O(s \cdot S)$.

The bound in Theorem I.1 can be obtained by extending

[1]In BI format, an $n \times n$ matrix $A = (A_{ij})$ for $1 \leq i, j \leq 2$, is stored as the sequence $A_{11}, A_{12}, A_{21}, A_{22}$, where the A_{ij} are themselves stored recursively.

[2]This generalizes prior work, which required unsuccessful steals to also take $\Theta(s)$ time units.

the analysis in [7] using the parameter $\Gamma(D, B)$, but only when the cost of an unsuccessful steal is required to be $\Theta(s)$ time units. We provide a new proof for this theorem, which generalizes the earlier proof to allow for two different types of phases in the analysis (the analysis in [7] has only one type of phase). We need the two different types of phases for our second, and main, theorem, and by using these two types of phases in the proof of Theorem I.1 we are also able to handle the more general $O(s)$ bound on the cost of an unsuccessful steal.

In general, $\Gamma(D, B)$ can be unbounded, but as mentioned above, $\Gamma(D, B) = O(B)$ for the class of algorithms we analyze, called *block-resilient* HBP algorithms [10]. In a block-resilient algorithm, each block, during a single allocation, will store $O(B)$ variables; while noting that this property is somewhat challenging to guarantee on execution stacks due to natural space reuse, [10] nevertheless gives simple algorithmic constraints ensuring this property. In addition, each writable variable can be accessed only $O(1)$ times. Together, these ensure that $\Gamma = O(B)$.

Theorem I.1 implies that for block-resilient algorithms the presence of false sharing increases the number of steals by at most an $O(B)$ factor (strictly, an $O(1 + \frac{b}{s}B)$ factor) over the case analyzed in [7] when no false sharing is present. Theorem I.2 will show a smaller factor increase than B. The reason the bound in Theorem I.1 can be unduly pessimistic is that it does not make use of the fact that, at each occurrence of an fs miss, one processor does, in fact, succeed in accessing a block (at the cost of at most a cache miss). By exploiting this fact, we are able to establish sharper bounds on S for block-resilient HBP algorithms. We introduce a new parameter, $\bar{l}(D)$, which captures the contribution of fs misses incurred during accesses to 'global data', and bounds the number of steals more precisely. $\bar{l}(D)$ is defined precisely in Section IV; to a first approximation, it is the maximum number of distinct BP computations — tree-like HBP computations — along any computation path in D. We will prove the following theorem.

Theorem I.2. *Let D be the computation dag of a block-resilient HBP algorithm. Then, when scheduled using RWS, with high probability in n, the computation of D incurs at most the following number S of successful steals:*

$$S = O\left(p \cdot T_\infty \cdot \left(1 + \frac{b}{s} \cdot B \cdot \frac{\bar{l}(D)}{T_\infty}\right)\right)$$

In addition, with high probability in n, the time spent on all steals, successful and unsuccessful, is $O(s \cdot S)$.

For block-resilient HBP algorithms, as $\Gamma(D, B) = O(B)$, Theorem I.2 replaces the term $p \cdot \frac{b}{s} \cdot B \cdot T_\infty$ in Theorem I.1 by the term $p \cdot \frac{b}{s} \cdot B \cdot \bar{l}(D)$. A gain occurs when $\frac{b}{s} \cdot B = \omega(1)$ because, as we will establish, $\bar{l}(D) = o(T_\infty)$. For example, for BP algorithms we have $T_\infty = \Theta(\log n)$, and we will establish that $\bar{l}(D) = O(1)$. Thus, while Theorem I.1 bounds S for BP computations by $S = O(p \cdot \log n \cdot (1 + \frac{b}{s}B))$, Theorem I.2 improves the bound to $S = O(p \cdot \log n(1 + \frac{b}{s}\frac{B}{\log n}))$.

HBP algorithms are built hierarchically from BP algorithms; this leads to analogous gains in the bounds on S for these algorithms. The full collection of bounds for the algorithms we analyze is shown in Table I, where the results for the expected number of steals are obtained by using Theorem I.2 together with Theorem V.2 in Section IV, which establishes bounds on $\bar{l}(D)$ for specific classes of HBP algorithms.

Using Theorem I.2 we can derive w.h.p. in n bounds on the parallel time taken in an execution of any algorithm in Table I, when scheduled under RWS, by observing that at every point during an execution under RWS, each processor is performing one of the following: a local computation, or waiting on a cache miss or an fs miss, or performing a steal, successful or unsuccessful. Let $W(n)$ and $Q(n, M, B)$ be worst-case upper bounds on the sequential time (i.e., the work) and sequential cache complexity of the algorithm; these are well-known quantities, and $Q(n, M, B)$ is listed in the second column of Table II for the algorithms we consider. Let $C(n, M, B)$ be an upper bound on cache miss excess under S steals, and $F(n, M, B)$ be the excess due to fs misses under S steals. Upper bounds for C and F were established in [9] and [10] respectively, and are listed in columns 3 and 4 of Table II (for C earlier bounds may be found in [7], [8]). The bounds for Q, C, and F are all in units of b, the cost of a cache miss. Finally, Theorem I.2 gives w.h.p. bounds on S, the number of successful steals under RWS, which can be now used to bound C and F. Recall that s is the time spent on a successful steal, and Theorem I.2 establishes that the time spent on all steals, successful and unsuccessful, is bounded by $O(s \cdot S)$, even if an unsuccessful steal could take less time than a successful one. With these values in hand, we can now bound, w.h.p. in n, the time taken by an execution on p processors under RWS by

$$T_p = O\left(\frac{W + b \cdot (Q + C + F) + s \cdot S}{p}\right)$$

Considering only the three components that are measured in units of b (the cost of a cache miss), the last column of Table II gives bounds on p for which $C + F = O(Q)$, i.e., when the cache miss and fs miss overhead due to steals under RWS is dominated w.h.p. in n by the sequential cache complexity of the algorithm. We refer to this as a bound for optimal performance.

For matrix algorithms, it turns out that the bit-interleaved (BI) format is more effective in controlling fs miss costs than the Row Major (RM) format; consequently, we analyze the

[3]These are variants of the standard algorithms which ensure that each writable variable is written $O(1)$ times.

Block Resilient HBP Algorithm	T_∞	$\overline{l}$ (Thm V.2)	Expected # Steals, S with FS Misses	
			Theorem I.1	Theorem I.2
Scans, Matrix Transpose (MT)	$\log n$	1	$p \cdot \log n \cdot (1 + \frac{b}{s}B)$	$p \cdot \log n \cdot (1 + \frac{b}{s}B/\log n)$
Row Major (RM) to Bit Interleaved (BI) format	$\log n$	1	$p \cdot \log n \cdot (1 + \frac{b}{s}B)$	$p \cdot \log n \cdot (1 + \frac{b}{s}B/\log n)$
Matrix Multiply (MM), Strassen	$\log^2 n$	$\log n$	$p \cdot \log^2 n \cdot (1 + \frac{b}{s}B)$	$p \cdot \log^2 n \cdot (1 + \frac{b}{s}B/\log n)$
Depth-n-MM [3]	n	$\frac{n}{\sqrt{B}}$	$p \cdot n \cdot (1 + \frac{b}{s}B)$	$p \cdot n \cdot (1 + \frac{b}{s}\sqrt{B})$
I-GEP (Gaussian Elimination) [3]	$n \log^2 n$	$\frac{T_\infty}{\sqrt{B}}$	$p \cdot n \cdot \log^2 n \cdot (1 + \frac{b}{s}B)$	$p \cdot n \cdot \log^2 n \cdot (1 + \frac{b}{s}\sqrt{B}/\log^2 n)$
Longest Common Subsequence (LCS) [3]	$n^{\log_2 3}$	$\frac{T_\infty}{B}$	$p \cdot n^{\log_2 3} \cdot (1 + \frac{b}{s}B)$	$p \cdot n^{\log_2 3} \cdot (1 + \frac{b}{s})$
BI to RM for MM and FFT	$\log n$	1	$p \cdot \log n \cdot (1 + \frac{b}{s}B)$	$p \cdot \log n(1 + \frac{b}{s}B/\log n)$
FFT, sort	$\log n \cdot \log\log n$	$\frac{\log n}{\log B}$	$p \cdot \log n \cdot \log\log n \cdot (1 + \frac{b}{s}B)$	$p \cdot \log n \cdot \log\log n \cdot$ $(1 + \frac{b}{s}B/(\log B \log\log n))$
List Ranking	$\log^2 n \cdot \log\log n$	$\frac{T_\infty}{\log n}$	$p \cdot \log^2 n \cdot \log\log n \cdot (1 + \frac{b}{s}B)$	$p \cdot \log^2 n \cdot \log\log n \cdot 1 + \frac{b}{s}B/\log n)$
Connected Components	$\log^3 n \cdot \log\log n$	$\frac{T_\infty}{\log n}$	$p \cdot \log^3 n \cdot \log\log n \cdot (1 + \frac{b}{s}B)$	$p \cdot \log^3 n \cdot \log\log n \cdot (1 + \frac{b}{s}B/\log n)$

Table I
BOUNDS ON THE NUMBER OF STEALS; $O(\ \cdot\ )$ OMITTED ON EVERY TERM. THE BOUNDS IN THEOREM I.1 WITH $B = O(1)$ ARE THE SAME AS THE BOUNDS WITH NO FS MISSES STEMMING FROM THE ANALYSIS IN [7]. NOTE THAT $\Gamma = O(B)$ FOR ALL THESE ALGORITHMS.

Block-resilient Algorithm	Cache Misses Seq. Execution, Q	Cache Miss Excess with S Steals, C [9]	FS Misses F [10]	Constraints for Optimal RWS Performance with False Sharing
Scans	$\frac{n}{B}$	S [8], [9]	$S \cdot B$	$n \geq pB^2(\log n + B)$
MT, RM to BI	$\frac{n^2}{B}$	$S \cdot \min\{M/B, B\}$	$S \cdot B$	$n^2 \geq pB^2(\log n + B)$
MM	$n^3/(B\sqrt{M})$	$S^{\frac{1}{3}} \cdot \frac{n^2}{B} + S$	$S \cdot B$	$n^3 \geq pM^{1/2}(M + B^2)\log n(B + \log n)$
Strassen ($\lambda = \log_2 7$)	$n^\lambda/(M^{\lambda/2-1}B)$	$S^{1/\lambda}\frac{M}{B}\left(\frac{n}{\sqrt{M}}\right)^{\lambda-1} + S$	$S \cdot B$	$n^\lambda \geq pM^{\lambda/2-1}(M + B^2)\log n(B + \log n)$
Depth-n-MM	$n^3/(B\sqrt{M})$	$S^{\frac{1}{3}}\frac{n^2}{B} + S$ [8], [9]	$S \cdot B$	$n^2 \geq p(MB)^{1/2}(M + B^2)$
I-GEP	$n^3/(B\sqrt{M})$	$S^{\frac{1}{3}}\frac{n^2}{B} + S\log B$ [8], [9]	$S \cdot B$	$n^2 \geq p(\log^2 + \sqrt{B})\sqrt{M}(M + B^2)$
LCS	$n^2/(MB)$	$n\sqrt{S}/B + S\log B$ [8], [9]	$S \cdot B$	$n^{2-\log_2 3} \geq p \cdot B \cdot M \cdot (B^2 + M)$
BI to RM for MM and FFT	$\frac{n^2}{B}\log\log_M n$	$S \cdot \min\{M/B, B\}$ $+\frac{n^2}{B}\log\log_M n$	$S \cdot B$	Dominated by MM and FFT
FFT, sort	$S_{\text{sort}} = \frac{n}{B}\log_M n$	$C_{\text{sort}} = O(S \cdot \min\{M/B, B\}$ $+\frac{n}{B}\frac{\log n}{\log[(n\log n)/S]})$	$S \cdot B$	$n \geq p \cdot (\log\log n + B/\log B) \cdot (M^\epsilon + B^2 \log M)$
List Ranking	S_{sort}	$C_{\text{sort}} \cdot \log n$	$S \cdot B$	$n \geq p \cdot \log\log n \cdot (\log n + B) \cdot (M^\epsilon + B^2 \log M)$
Connected Cmpts.	$S_{\text{sort}} \cdot \log n$	$C_{\text{sort}} \cdot \log^2 n$	$S \cdot B$	$n \geq p \cdot \log\log n \cdot (\log n + B) \cdot (M^\epsilon + B^2 \log M)$

Table II
CONSTRAINT ON INPUT SIZE FOR OPTIMAL MULTICORE PERFORMANCE UNDER RWS WHEN BOTH FS MISS OVERHEAD [10] AND CACHE MISS OVERHEAD [8], [9] ARE CONSIDERED; $O(\ \cdot\ )$ OMITTED ON EVERY TERM. Q, C, F ARE IN UNITS OF b, THE CACHE MISS COST.

matrix multiply (MM) algorithms assuming the BI format. To enable input or output in the RM format, we also analyze algorithms for converting between these two formats. We note that the algorithm for BI to RM conversion performs superlinear work, which is also the reason for its higher cache miss cost; as explained in [10], this is done in order to reduce the fs misses.

Road-map: In Section II we outline the proof of Theorem I.1. In Section III we review the definition of HBP algorithms and in Section IV we outline the proof of Theorem I.2. Finally, in Section V we describe and analyze the individual algorithms. For lack of space, some proofs are only sketched.

II. BOUNDING THE STEALS IN A SERIES PARALLEL DAG COMPUTATION

Proof: (of Theorem I.1). Broad-brush, our analysis follows the approach taken in [7]. As in [7], we bound the number of steals using a potential function ϕ.

We begin by defining ϕ. We assign a cost to each node in the execution dag D for a given computation. To this end, let e' be an upper bound on the number of operations (reads, writes and computations) performed in the execution of any one node. By assumption, $e' = O(1)$. Each node is given a cost of $b \cdot e' \cdot \Gamma(D, B) = b \cdot e' \cdot \Gamma$. We view each task corresponding to a node as performing up to $b \cdot e' \cdot \Gamma$ "work units" when it is executed, each work unit corresponding to one unit of time being expended on its execution. In addition,

to cover the cost of steals, any node performing a fork or join is given an additional cost of $2s$ (the factor of 2 simplifies the analysis). The cost of a path in D is simply the sum of the costs of the nodes on the path.

We define the the *max-path-cost* $c(u)$ of a vertex u in D to be $1/s$ times the maximum cost of all the paths descending from u. Note that $c(u) \leq \frac{1}{s}[b \cdot e' \cdot \Gamma \cdot (h(u)+1)+2s \cdot (h(u))]$, where $h(u)$ denotes the height of u in D (i.e. the maximum path length starting at u). And if t denotes the topmost node in D, it follows that $c(t) \leq \frac{1}{s}[b \cdot e' \cdot \Gamma \cdot T_\infty + 2s \cdot (T_\infty - 1)]$.

Definition II.1. *If the task τ_u associated with vertex u is on a task queue, u's associated potential $\phi(u) = 2^{1+c(u)}$; if τ_u is currently being executed by a processor, with x of its work units already having been performed, u has potential $\phi(u) = 2^{c(u)-(x/s)}$; otherwise, u's potential is zero. $\phi = \sum_u \phi(u)$.*

To show progress, we analyze the algorithm in periods called *phases*. We identify two types of phases, *steal* and *computation* phases. At the start of a new phase, if at least half the potential ϕ is associated with vertices u whose associated tasks τ_u are on task queues, this is a steal phase. Otherwise, it is a computation phase. A steal phase lasts until $2p$ attempted steals complete, successfully or not, while a computation phase lasts for b time units. This is in contrast to [7], where every phase lasts until p attempted steals complete. By defining the length of the computation phase in terms of the work done and not the number of attempted steals, we are able to bound the work done on steals even when an unsuccessful steal could take less time than a successful one.

Lemma II.2. *i. In a steal phase, the expected value of ϕ reduces to at most $\frac{7}{8}$ of its starting value; further, with probability at least $\frac{1}{15}$, ϕ reduces to at most $\frac{15}{16}$ of its starting value.*
ii. In a computation phase, ϕ reduces to at most $\left(1 - \frac{b}{8s}\right)$ of its starting value.

Proof: i. We first show the reduction in the expected value. Potential of at least $\frac{\phi}{3}$ is associated with tasks at the heads of queues, since, on any task queue, the heights of successive tasks decrease by an additive factor of at least 2, and hence at least $\frac{2}{3}$ of the potential associated with tasks on task queues is for tasks at the heads of these queues. Let τ_u be a task at the head of a task queue. The probability that τ_u is not stolen in one attempted steal is $1 - 1/p$. Hence over the at least $2p$ attempted steals, it is not stolen with probability $(1 - 1/p)^{2p} \leq 1/e^2$, and hence is stolen with probability more than $\frac{3}{4}$. If τ_u is stolen, the potential $\phi(u)$ decreases by a factor of 2. Consequently, the expected value of ϕ is reduced to at most $\frac{2}{3}\phi + \frac{1}{4} \cdot \frac{\phi}{3} + \frac{3}{4} \cdot \frac{1}{2} \cdot \frac{\phi}{3} = \frac{7}{8}\phi$.

Let ϕ' denote the value of ϕ after the steal phase. We have shown that $E[\phi'] \leq (7/8) \cdot \phi$. Since ϕ' is a non-negative random variable, we apply Markov's inequality to obtain that the probability that $\phi' \geq \frac{15}{14}E[\phi]$ is at most $14/15$. We have shown that $E[\phi'] \leq (7/8) \cdot \phi$, hence $\phi' \leq \frac{15}{16}\phi$ with probability at least $1/15$.

ii. Now we analyze the reduction to ϕ in a computation phase.
Suppose processor C is currently executing task τ_u corresponding to vertex u. Then in the current phase, C can do one of three things.
a. It could complete the computation of its current node. This could have one of three results: 1. A join occurs and C starts the computation of node w, the successor of u in D; as $\phi(w) \leq \phi(u)/2$ (recall that we are comparing the initial value of $\phi(w)$ to the current value of $\phi(u)$), the reduction in potential is at least $\phi(u)/2$. 2. C takes a task from its task queue. Then the reduction is potential is by more than $\phi(u)$. 3. C has no more work at hand and begins attempting steals. The reduction in potential is by $\phi(u)$.
b. C could perform a fork. We show that this reduces the associated potential $\phi(u)$ to at most $\frac{3}{4}\phi(u)$. When a processor executing task τ_u forks, it creates tasks τ_v and τ_w, placing τ_w on its task queue. Recall that each forking node is assigned an additional cost of $2s$. Hence, the forked task v that is placed on the task queue has potential $\phi(u)/2$, and the forked task w that continues the execution has potential $\phi(u)/4$. Thus, the potential is reduced from $\phi(u)$ to $\phi(v) + \phi(w) \leq \phi(u)/4 + \phi(u)/2 = \frac{3}{4}\phi(u)$.
c. If (a) and (b) do not hold, then processor C executes its task throughout the phase without forking. Hence processor C performs a sequence of at least b work units. This reduces the starting potential $\phi(u)$ to at most $\phi(u)2^{-b/s} = \phi(u)(1+1)^{-b/s} \leq \phi(u)(1 - \frac{b}{2s})$ if $b \leq s$.

At the start of a computation phase, at least half the potential is associated with nodes that are being computed. For each such node u, its potential reduces by at least one of the following: Case a: $\phi(u)/2$; Case b: $\phi(u)/4$; Case c: $\frac{b}{2s}\phi(u)$; this is at least $\min\{\frac{1}{4}, \frac{b}{2s}\}\phi(u) \geq \frac{b}{4s}\phi(u)$ (using the assumption that $b \leq s$). Hence in one computation phase the potential is reduced to at most $\frac{\phi}{2} + \left(1 - \frac{b}{4s}\right\}) \frac{\phi}{2} \leq \left(1 - \frac{b}{8s}\right)\phi$. ∎

We continue the proof of Theorem I.1.

Say that a steal phase is successful if ϕ reduces to at most $\frac{15}{16}$ of its value at the start of the phase, and that it is unsuccessful otherwise. By Lemma II.2(i), a steal phase is successful with probability at least $\frac{1}{16}$.

Suppose that there are x successful steal phases, y unsuccessful ones, and z computation phases during the computation. Recall that in a computation phase the potential reduces to at most $(1 - \frac{b}{8s})\phi$, and in a successful steal phase to at most $\frac{15}{16}\phi$. Initially $\phi = 2^{c(t)}$, and when ϕ has reduced to 1 all the successful steals have completed. Hence both $\left(\frac{15}{16}\right)^x$ and $\left(1 - \frac{b}{8s}\right)^z$ are at most $O(2^{c(t)})$. Thus, $x + \frac{b}{s}z = O(c(t))$. Also, since the probability of an unsuccessful phase is at most $15/16$ and steal phases are mutually independent, by a Chernoff bound we have $y = O(x)$ w.h.p. in n since

$c(t) = \Omega(T_\infty)$ (and $T_\infty = \Omega(\log n)$ since we have binary forking and n is the amount of input and output data accessed.)

Now each computation phase takes b time units and hence uses $O(pb)$ time over all p processors. So the z computation phases use $O(pbz)$ time units over all p processors. As a successful steal takes at least $\Theta(s)$ time units, there can be only $O(pbz/s)$ successful steals that start and finish in a contiguous sequence of computation phases. Any other successful steal either starts or ends during a steal phase; there can be at most $2p$ of these per steal phase, $O(p(x+y))$ in total.

Hence w.h.p. in n, the total number of successful steals is at most $2p \cdot (x+y) + \frac{pbz}{s} = O(p \cdot (x + \frac{b}{s}z)) = O(p \cdot c(t)) = O(p \cdot T_\infty \cdot (1 + \frac{b}{s}\Gamma))$.

This establishes the first part of the Theorem. Further, the total time spent on steals, both successful and unsuccessful is $O(s \cdot p \cdot (x+y) + pbz)$, and by the above bound, this total time is $O(s \cdot p \cdot T_\infty \cdot (1 + \frac{b}{s}\Gamma)) = O(s \cdot S)$ w.h.p. in n. This establishes the second part of the Theorem. ■

III. HBP Algorithms

We review the definition of HBP algorithms [10], [11]. Here we define the size of a task τ, denoted $|\tau|$, to be the number of distinct, already declared variables it accesses over the course of its execution (this does not includes variables τ declares during its computation).

Definition III.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 $|\tau_v| \leq \alpha|\tau_w|$.*

A simple BP example is the natural top-down balanced-tree procedure to sum n integers.

Definition III.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.
3. Sequencing. A sequenced Type t HBP algorithm 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. Recursion. A Type $t+1$ recursive *HBP algorithm, for $t \geq 1$, results if, for a size n problem, 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 $\Theta(r(n))$, and $r(n)$ is bounded by αn for some constant $0 \leq \alpha < 1$.*

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.

An algorithm is *limited-access* if any writable variable is accessed $O(1)$ times during its execution [10], [11]. An HBP algorithm is *block-resilient* if (i) it is limited-access, (ii) it is 'top-dominant', and (iii) it satisfies certain requirements on the data layout that we describe in Section IV for BP computations. Since the only property of top-dominance that we need in this paper is that it enables the property $\Gamma(D, B) = O(B)$ we refer the reader to [10] for its definition. Also, [10] defines a *block-sharing function* $L(r)$ but since all of the algorithms we consider have $L(r) = O(1)$, we will not discuss it here.

Matrix Multiply (MM) with 8-way recursion is an example of a block-resilient 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 $r(n^2) = n^2/4$.

IV. Bounding the Steals in HBP Algorithms

In turn, we analyze BP and then HBP computations. But before entering into this analysis it will be helpful to look more closely at how variables are stored.

An algorithm's input and output variables are called *global variables*. All other variables will be local to some procedure within the algorithm's execution. The local variables are all stored on execution stacks as we explain next.

Execution Stack. When a new thread is started, either because it is executing the original task τ for the computation, or a stolen subtask τ, the thread creates an execution stack E_τ to keep track of the procedure calls and variables in the work it performs on τ. For each node the thread executes, it creates a *segment* to hold the variables declared by the node. In a BP computation, this will be $O(1)$ variables. The

segment is placed at the top of E_τ. When the procedure completes, the space used by the segment is released.

Finally, we note that the input and output variables, which may be arrays, are stored in memory locations separate from those used for the execution stacks, and share no blocks with the execution stacks.

The Potential Function: Our approach is to replace $c(u)$ in the definition of ϕ with a tighter bound on the cost of executing any path descending from u. We denote this bound by $\ell(u)$, the *hbp-length* of u. We will express $\ell(u)$ as $\ell_1(u) + \frac{b}{s} \cdot (\ell_2(u) + \ell_3(u) + \ell_4(u))$. Here, $\ell_1(u)$ serves the same purpose as $2 \cdot h(u)$ in the proof of Theorem I.1, and $\ell_2(u)$ and $\ell_3(u)$ address false sharing costs due to 'global' and 'local' variables, respectively, as described below. The parameter $\ell_4(u)$ is not needed for BP computations, and in an HBP computation, it addresses fs costs due to accesses to a block from different recursive calls. We will show that $\ell_1(u) + \ell_3(u) = O(T_\infty)$, while $\ell_2(u)$ and $\ell_4(u)$ depend on B.

The parameter $\overline{\ell}(D)$ in Theorem I.2 is defined as $\frac{1}{B}$ times the initial value of $(\ell_2(t) + \ell_4(t))$.

The hbp-length $\ell(u)$ will change dynamically as the algorithm execution proceeds. The challenge in designing ℓ is that for an edge (u, v), the difference $\ell(u) - \ell(v)$ needs to be at least the time taken to perform the operations at node u, which could include the delay due to fs misses. However, we want this difference to be large only if there really are fs misses. Further, the nodes at which fs misses occur depend on the order of execution, and our analysis needs to account for every possible order. We enable sufficiently large differences when needed by *dynamically reducing* the value of $\ell(w)$ for nodes w that could access block β when β incurs an fs miss; we also reduce the ℓ value for selected descendants of such nodes w.

A. BP Computations

In a BP computation, we do not use $\ell_4(u)$, hence $\ell(u) = \ell_1(u) + \frac{b}{s} \cdot (\ell_2(u) + \ell_3(u))$ here, and $\overline{\ell}(D) = \frac{1}{B}$ times the initial value of $\ell_2(t)$.

Recall that a BP computation comprises a down-pass tree, which we name T_d, followed by an up-pass tree, named T_u, with the leaf nodes being common to the two trees; and Theorem I.1 gives $S = O(p \cdot (\log n + \frac{b}{s} \cdot B \log n))$.

We distinguish between data accesses to global and to local variables. Global variables are the inputs and outputs of the BP computation. Due to the recursive fork-join nature of the computation, two nodes that have an ancestor-descendent relation cannot execute concurrently. Hence, fs misses can only occur between nodes in sibling subtrees. Further, by definition, block-resilient BP computations [10] satisfy the natural property that any node in the forking tree can access only a constant number of locations in the input and output, centered around the position of that node in an in-order ordering. Thus, nodes at levels $\Theta(\log B)$ and higher above the leaves of the down-pass and up-pass trees will have *no* false sharing cost. We set up a component $\ell_2(u)$ in $\ell(u)$ (recall $\ell(u)$ replaces the $c(u)$ in Theorem I.1) so that it incorporates the benefit of this property, and for global variables, this replaces the $B \log n$ term in Theorem I.1 by just B.

BP computations also access local variables on the execution stack. Here, there is indeed the possibility of fs misses at any level in the BP dag. However, the false sharing interactions on an execution stack occur only among the task that created the execution stack and the tasks stolen from it. Further all of the stolen tasks are right siblings of nodes that lie on a single path (the *steal path* [10], [9]) in the down-pass (fork) tree. We set up the third component $\ell_3(u)$ in $\ell(u)$ to account for this special type of interaction on the execution stacks. This allows us to argue that for local variables, the $B \log n$ term in Theorem I.1 reduces to $O(\log n)$. The contributions of ℓ_2 and ℓ_3 together allow us to derive the improved bound in Theorem I.2 for fs misses.

The BP Analysis. Let D be the dag of a BP computation with root t. Note that the down-pass tree T_d and the up-pass tree T_u both have height $H = \Theta(\log n)$. For a node u in D, let $h(u)$ be the length of a longest path descending from u; then $h(t) = O(\log n)$.

For the BP computation dag D, the max-path cost $c(u)$ of a node u from Lemma II.2 is $c(u) = 2h(u) + \frac{b}{s} \cdot e' \cdot \Gamma \cdot [h(u) + 1])$, with $\Gamma = O(B)$. Here, we will replace $c(u)$ by $\ell(u) = l_1(u) + \frac{b}{s} \cdot (\ell_2(u) + \ell_3(u))$, and we will define $\phi(u)$ as in Definition II.1 with $c(u) - x/s$ replaced by ℓ. Also, $l_1 = O(\log n)$ will serve the same purpose as the term $2 \cdot h(u)$ in $c(u)$ and we will not discuss it further.

We now define $\ell_2(u)$ and $\ell_3(u)$. Both of these values can change during the computation, and will be set up so that they satisfy the two necessary properties of always being non-negative, and of always having $\ell_i(v) \geq \ell_i(w)$, $i = 2, 3$, for each edge (v, w) in D (the *edge rule*). This ensures that, if we traverse any path down the dag D, $\phi(u)$ decreases by a factor of at least 4 at each node along the path.

Global Variables and ℓ_2. Writes to the global variables (typically arrays of size n) in a block-resilient computation [10] obey the following *well-buffered rule*. Let v be a node in T_d, and let T be the subtree in the down-pass tree rooted at v. Suppose that T's nodes can access an array A. Then all the accesses by T occur in an interval I of length $\Theta(|T|)$ and the only nodes that can access I are those in T and in the complementary tree in the up-pass tree. Furthermore, v can only access the middle of I: there are left and right portions of I of length $\Theta(|T|)$ that v cannot access, and which can be accessed only by v's left and right subtrees, respectively. A similar property applies to T_u. (Note that prefix-sums can be implemented as a sequence of two well-buffered BP computations.)

Conflicting accesses cannot occur between a node and its

proper ancestor, hence a key property of this access pattern is that there is no fs miss for accesses to global variables at nodes that are the roots of subtrees of size at least $e''B$ in either T_d or T_u, for a suitable constant $e'' > 1$. In other words, any conflicting accesses can occur only in the following *conflict* subtrees: subtrees of size at most $e''B-1$ at the bottom of either T_d or T_u. We define the initial values of ℓ_2 as follows. For nodes in the up-pass tree below the conflict subtrees (when looking top-down in D) $\ell_2(u) = 0$ throughout the computation; for all other nodes $\ell_2(u) = e \cdot B$ initially, where $e = 3 \cdot e' \cdot e''$.

When a process accesses a block storing a global variable that some node u in conflict subtree T needs to access, $\ell_2(w)$ is decremented for *every node* in T and in its complementary conflict tree T'. Since any node in a conflict tree T has $O(1)$ accesses, and since the blocks it can access can be accessed only by nodes in T, in T's left and right neighbors, and in their complementary trees, and since each conflict tree has size $O(B)$, ℓ_2 can be decremented at most $O(B) \leq 3 \cdot e \cdot B$ times. Also since nodes outside the conflict trees have no decrements in ℓ_2, ℓ_2 remains non-negative at every node and satisfies the edge rule.

Local Variables and ℓ_3. The local variables are used to store data needed by the individual computation nodes; there are at most e' such (one-word) variables per node. Each computation thread will have an execution stack on which it stores the local variables it generates. When the computation of a node v begins, its local variables are added to its thread's execution stack, and when the computation of node v ends, this space is released.

Block resilient BP algorithms obey the following *local constraint* [10] (achieved by a natural scoping of variables and the use of return value variables). Writes to local variables by the task for a node v are to v's local variables, and in the up-pass to the local variables declared by the complementary downpass node v' plus possibly to the local variables at $parent(v')$ in T_d.

Consider the execution stack S_τ for a task τ in D executed by a processor C. The tasks stolen from τ and executing at other processors correspond to the right subtrees of a subset of the nodes whose local variables are stored on τ's execution stack. With ℓ_3 we will bound the cost of fs misses due to accesses to blocks on the execution stack of τ by C and these other processors. This is the most nontrivial part of our analysis.

Let β be a block storing a portion of S_τ. The only nodes that can access β are some of those in the non-stolen portion of τ, and in tasks stolen from τ. We define ℓ_3 so as to ensure a sufficient reduction in the overall potential when such accesses cause one or more processors to incur an fs miss. Our definition of ℓ_3 is set up so that in some cases there is no reduction to the ℓ_3 values of nodes that can be delayed by an access to β (this is done in order to maintain the non-negativity of ℓ_3 and the edge rule); however, this happens only when the contribution to the potential by these nodes is small in comparison to that by another node whose potential will decrease.

Initial values of ℓ_3. For a node v in T_u, $\ell_3(v) = 2e'(h(v) + 1)$.

Let l_3^* be the maximum value of ℓ_3 among all nodes in T_u, (this maximum value occurs at the leaves of T_u). For a non-leaf node v in T_d, we set $\ell_3(v) = l_3^* + e'(h(v) - 1)$.

We will discuss updates to ℓ_3 within the proof of Lemma IV.1, which we are now ready to prove. The change from Lemma II.2 is that the initial value of $\ell(t)$ is significantly smaller than the initial (and unchanging) value of $c(t)$ at the root t of D, and hence the initial value of ϕ is correspondingly smaller. In turn, this enables an improved bound on the number of steals in Theorem I.2, the analog of Theorem I.1, but the proof is more challenging.

Lemma IV.1. *i. In a steal phase, the expected value of ϕ reduces to at most $\frac{7}{8}$ of its starting value; further, with probability at least $\frac{1}{16}$, ϕ reduces to at most $\frac{15}{16}$ of its starting value.*
ii. In a computation phase, ϕ reduces to at most $\left(1 - \Theta(\frac{b}{s})\right)$ of its starting value.

Proof: We define steal and computation phases as in Lemma II.2, except that a steal phase will now last for $2b$ steps to ensure that a processor C with enough work will either complete at least b steps of work, or if it waits for a block β, some other processor competing with C will access β in that phase.

The proofs of (i) and of (ii) for the case when there is no fs miss are similar to those in Lemma II.2, and are omitted. We now account for the delay, in a computation phase, due to fs misses at each node u with $\phi(u) > 0$. First, as we will see, at all times $\ell_2(u)$ and $\ell_3(u)$ remain non-negative and $\ell_i(u) - \ell_i(w) \geq 0$ for each edge (u, w) in D; this ensures that $\phi \geq 1$ while the computation is ongoing. Then, we analyze each u that is delayed by an fs miss in this computation phase, in part (a) if u was accessing a global variable, and in part (b) if u was accessing a local variable, as follows.

(a) Global access: We have set up ℓ_2 so that when a block β that *could be* accessed by a node in a conflict tree T is touched during a computation phase, ℓ_2 is decreased by 1 for every node in T. As $\phi(u) = 2^{\ell(u)}$ for a node u currently being executed, and as $\ell(u) = \ell_1(u) + \frac{b}{s}(\ell_2(u) + \ell_3(u))$, if there is an access to a global variable in a block that u could access, then whether or not u incurs an fs miss due to this access, $\phi(u)$ decreases by a $(1 - \Theta(b/s))$ factor.

(b) Local access: There are four subcases in this part of the analysis. As we will see, decrements to ℓ_3 occur only in Cases 1 and 3.

Case 1. Node u in the up-pass tree succeeds in an access or is blocked by a node v that is the sibling of u or of one of

u's descendants in D.
Here, $\ell_3(u)$ is decremented by 1 and $\phi(u)$ decreases by an $e^{-b/s} \geq 1 - \frac{1}{2}\frac{b}{s}$ factor (as $s \geq b$ by assumption). As $\ell_3(u)$ is decremented at most $e'(h(u)+1)$ times, $\ell_3(u) \geq 0$ always. Further, $\ell_3(u)$ is decremented at most $2e'$ times when its immediate descendant in D is unchanged (due to accesses by u and u's sibling); thus, for any edge (u, v) in D, $\ell_3(u) \geq \ell_3(v)$ always.

Case 2. Node u in either tree attempts to access a local variable in a block β on the execution stack of task τ, when a task is present on τ's task queue.

Then $\ell_3(u)$ is unchanged. We will show below that the total potential of the nodes covered by this case is $\frac{1}{3}\phi$, where ϕ is the total potential at the start of the current computation phase. This will suffice, for then it follows that the nodes covered by the other cases have combined potential at least $(\frac{1}{2}-\frac{1}{3})\phi = \frac{1}{6}\phi$ at the start of the phase, and hence the overall potential reduction is by a factor of at least $1 - \frac{1}{6}\cdot\frac{3}{8}\frac{b}{s}$, since the potential of the nodes handled by the other cases reduces by a factor of at least $1 - \frac{3}{8}\frac{b}{s}$.

Let v be the root node for the task on τ's task queue.

Consider the nodes accessing β. By the Local Constraint, the only nodes which can be accessing β are a node u_1 in the non-stolen task rooted at v's sibling and nodes $u_2, \cdots, u_k$ that are the terminal nodes for subtasks stolen from ancestors of v. Note that u_1 is either the sibling of v or a descendant of that sibling. Since v is on the task queue, it follows that $\phi(v) \geq 2\phi(u_1)$. Further, since ℓ_1 reduces by 2 in each successive level of the up-tree, the potential for all the nodes accessing β is bounded by $\phi(u_1)[1 + \frac{1}{4} + \frac{1}{16} + \cdots] \leq \frac{4}{3}\cdot\phi(u_1) = \frac{2}{3}\phi(v)$. Since this is a computation phase, the total potential for all nodes on the task queue was at most half the potential at the start of the phase, and hence, as claimed, the potential at the start of the current computation phase associated with the nodes u is at most $\frac{1}{3}\phi$.

Case 3. Node u in the down-pass tree attempts to access a local variable and there is no task on the task queue as defined in Case 2.
In this case $\ell_3(u)$ is decremented; in addition, if its access was blocked by the subtask stolen from one of its ancestors v, then $\ell_3(w)$ is decremented for *all non-leaf descendants* w of u in the down-pass tree. Note that leaf nodes are already covered by Case 1. Note that as in Case 1, $\phi(u)$ decreases by at least a $1 - \frac{1}{2}\frac{b}{s}$ factor.

There are up to e' such decrements at the root of the downpass tree, and for a node u in the down-pass tree $\ell_3(u)$ can be decremented at most e' times without also decrementing $\ell_3(w)$ for its descendants w. This ensures the edge rule is satisfied for every pair of internal nodes in the down-pass tree. The maximum number of decrements of $\ell_3(u)$ at a node u which is the parent of leaves of the down-pass tree in bounded by $e'(h(u) - 1$; this ensures the edge rule is obeyed by these nodes u also. As the ℓ_3 values will remain non-negative at the leaves of the down-pass tree, it follows that they remain non-negative at the internal nodes of the down-pass tree.

Case 4. Node u in the up-pass tree is blocked by a node v and node u is not handled in Cases 1–3.
Here we leave $\ell_3(u)$ unchanged. In this case, v is higher up the dag than u, and since the component of the potential due to ℓ_1 decreases by a factor of 4 from one level to the next down the dag, the contribution to the potential due to all nodes like u plus node v is at most $\frac{4}{3}\phi(v)$. Node v is handled by one of Cases 1 or 3. Consequently the total potential of these nodes decreases by a factor of at least $1 - \frac{3}{8}\frac{b}{s}$. ■

Theorem I.2 for BP algorithms follows from Lemma IV.1 in the same way that Theorem I.1 follows from Lemma II.2.

We have $\bar{l}(D) = \frac{1}{B}\cdot(\ell_2(D) + \ell_4(D))$. For a BP computation $\ell_2(D) = O(B)$ and $\ell_4(D) = 0$, hence $\bar{l}(D) = O(1)$. We can now apply Theorem I.2 to bound the number of successful steals: $S = O(p\cdot(\log n + \frac{b}{s}))$ with high probability in n in a BP computation of size n. The algorithms for Scans, Matrix Transpose (MT), and RM to BI are all BP computations, and this leads to the bounds given in the last column of the first two rows of Table I.

B. HBP computations

We begin by generalizing the local/global variable terminology to HBP algorithms.

Definition IV.2. *A variable x declared in a procedure Q is called a* local *variable of Q. A variable y accessed by a procedure P is* global *with respect to P if y is declared in a procedure Q calling P or is used for the inputs or outputs of the algorithm $\mathcal{A}$ containing P.*

Also, we comment in more detail on how the local variables are arranged on the execution stacks. First, we note that a node of the computation dag that corresponds to the start of an HBP task could be declaring many variables, perhaps in the form of an array, and its segment would be correspondingly large. Second, we comment on the sequencing of segments. in particular, let τ be a task of size n in a Type 2 HBP computation. As the execution of τ advances, (a prefix of) the following sequence of segments will be on its execution stack E_τ: an initial segment for its local variables, and then up to $\log v(n)$ segments of length $O(1)$ to keep track of parallel recursive calls; the current recursive call will then create similar entries following these initial entries on E_τ. The topmost segment on E_τ will either be one of the above types of segments or, if τ is currently executing a BP computation within the HBP computation, at the top there will be $O(\log n)$ segments of length $O(1)$ for this BP computation. An analogous description applies to higher type HBP computations.

Finally, it should be noted that in an HBP algorithm $\mathcal{A}$, the only non-writable variables are those for the input, which

are global variables to all procedures, and are not stored on any execution stack.

The rules restricting the writes in BP computations apply equally to the down-pass and up-pass trees used to instantiate recursive calls in HBP algorithms. The subgraphs corresponding to the recursive computations are analogous to the leaves of a BP computation. This permits us to perform an analysis of the HBP computations which is similar to that for the BP computations.

To enable such an analysis, we require that the writes by the recursive computations to the local variables (arrays) of their calling procedures obey an analog of the well-buffered rule for BP global variable access, namely that the left-to-right sequence of recursive computations write to successive disjoint portions of the parent's arrays. Further, we require that within each recursive procedure, its writes to these arrays be similarly constrained. A simple way of ensuring this is to impose the following constraints:

HBP Write Constraints. i. A recursive call performs its writes to such arrays by means of a BP computation that occurs at the end of the recursive call.
ii. The collection of these BP computations terminating the recursive calls obeys the *Well Buffered Rule for a BP Collection* when accessing an array A, namely: each leaf accesses a disjoint interval in A; further, the inorder listing of the leaves (i.e. in left to right order) matches the left to right ordering of the corresponding intervals in A.

An HBP algorithm can always be modified to have this structure, by accumulating such writes in an array local to the recursive call which is then copied at the end of the recursive call; these writes can be restricted so that they all occur at the leaf level of the BP copying task.

We use the ℓ_i functions, modulo some small changes, as for the BP computations. In fact, the definition of ℓ_1 in unchanged: $\ell_1(u)$ is 2 times the height of u in the dag D.

To define ℓ_2 and ℓ_3, we view the computation dag of an HBP computation as comprising a collection of paired down-pass and up-pass trees as used in BP computations and in forking and joining recursive computations, plus singleton nodes corresponding to type 0 tasks. We want to use essentially the potentials as defined for BP trees, but in addition, we need to ensure that each terminal node w in a BP tree (a leaf in a down-pass tree, the root in an up-pass tree) always has at least as large an ℓ_i value as its successor node x in the computation dag. But this is easily achieved: if w is a root node of up-pass tree T_u, and $\ell_i^*(x)$ is the initial value of $\ell_i(x)$, for $i = 1, 2$, we simply add $\ell_i^*(x)$ to the BP-like potential for each of the nodes in T_u. A similar rule is used for a down-pass tree T_d, but now we take the maximum value $\ell_i^*(v)$, maximizing over the successor nodes of the leaves v in T_d.

For a node v in an up-pass tree, the BP-like potentials $\ell_2^{BP}(v)$ and $\ell_3^{BP}(v)$ are defined exactly as for the BP-computation. For a node v in a down-pass tree, $\ell_2^{BP'}(v) = e \cdot B$ and $\ell_3^{BP'}(v) = e' \cdot ht'(v) + 1$, where $ht'(v)$ is the height of v in its down-pass tree. Finally for the nodes corresponding to type 0 tasks, we define both $\ell_2^{BP}(v)$ and $\ell_3^{BP}(v)$ to be zero.

It is straightforward to show that ℓ_i, for $i = 2, 3$ continues to obey the edge rule and is always non-negative.

In fact, we use a smaller initial value of $\ell_2(v)$ for nodes v in a task τ when $|\tau|$ is small enough. For τ can access at most $|\tau|$ global variables, by the definition of size. Let e_2 be the bound in the limited access assumption, i.e. each writable variable is accessed at most e_2 times. Then, for $e_2 \cdot |\tau| < e \cdot B$, we redefine the initial value of $\ell_2^{BP}(v)$ to be $e_2 \cdot |\tau|$, where before it had initial value $e \cdot B$.

We need to introduce a fourth function ℓ_4. As usual, let τ be a task having an execution stack E_τ. The function ℓ_4 will handle competing accesses to the one block β that may be shared between the variables declared by τ and variables that are subsequently added to E_τ during the course of τ's execution.

We now define $\ell(v)$ to be $\ell(u) = \ell_1(u) + \frac{b}{s}[\ell_2(u) + \ell_3(u) + \ell_4(u)]$.

The definition of ℓ_4 will rely on a stronger bound on the block delay, as proved in [10], which takes account of the size of the task τ "owning" the block β being accessed; by this we mean that β is one of the blocks storing the execution stack E_τ for τ. The observation is that if τ uses space $S(|\tau|) < B$, then the number of cache transfers of β will be bounded by a tighter $O(S(|\tau|))$ (rather than $O(B)$); we denote this bound by $\Gamma(\tau, B) = O(\min\{S(|\tau|), B)\})$. This tighter bound is used in Lemmas V.1 and V.3 below. There is a small overloading of notation, as we use both of the terms $\Gamma = \Gamma(D, B)$ and $\Gamma(\tau, B)$, but this should not cause any confusion.

In all the HBP algorithms we analyze, the space $S(\tau) = \Theta(|\tau|)$. Thus the total delay due to fs misses incurred by any one task τ in performing all its accesses to β is bounded by $O(b \cdot \min\{S(\tau), B\}) = O(b \cdot \min\{|\tau|, B\})$.

Let e_1 be the constant such that $\Gamma(\tau, B) \leq e_1 \min\{S(|\tau|), B)\}$. The initial value of $\ell_4(v)$ is defined procedurally as follows. We begin with $\ell_4(v) = 0$. Then, for each type $t \geq 2$ task τ in the algorithm, $e_1 \min\{S(|\tau|), B)\}$ is added to the $\ell_4(v)$ value for every node v in τ's computation dag. The contribution associated with τ is used to pay for accesses to the at most one block β_τ shared between τ's local variables and the other variables stored on E_τ, i.e. the local variables for the non-stolen procedures called by τ.

Whenever β_τ is transferred, $\ell_4(v)$ is decremented for all nodes in τ's computation dag. Note that for every b time units a processor is delayed when computing node v and trying to access β_τ, there will be a block transfer of β_τ, and a corresponding decrement to $\ell_4(v)$. As there at at most $e_1 \min\{S(|\tau|), B\})$ block transfers of β_τ, it follows that ℓ_4

obeys the edge rule and is always non-negative.

The bound of Lemma IV.1 extends unchanged to the HBP computations, and on setting $\overline{l}(D)$ to be the initial value of $\ell_2(t)+\ell_4(t)$, where t is the root of D, we obtain Theorem I.2 for the HBP algorithms.

V. Steals Under RWS for Individual Algorithms

By Theorem I.2, which we proved in the previous section, we can obtain expected and high probability bounds on the number of steals under RWS for a given HBP algorithm if we can compute $\overline{\ell}$. Since $\overline{\ell}$ depends on ℓ_2 and ℓ_4, we need to bound ℓ_2 and ℓ_4 for specific HBP algorithms, which is done for Type 2 HBP in the next lemma.

Lemma V.1. *Let $\mathcal{A}$ be a block-resilient, Type 2 HBP algorithm which uses space $S(n)$. Suppose that each recursive call $\mathcal{A}$ makes has size at most $r(n) \leq n/b$, for some constant $b > 1$. Let $c \geq 1$ denote the number of collections of recursive calls made by $\mathcal{A}$ and let $\mathcal{A}$ have size n. Then for every node v in its computation dag D,*

$$\ell_2(v), \ell_4(v) = O(\ B \sum_{i<r^*(n,B)} c^i + \sum_{i\geq r^*(n,B)} c^i \cdot S(r^{(i)}(n))\ ),$$

where $r^*(n,B)$ where $i = r^*(n,B)$ is the number of applications of r needed to reduce n to at most $S^{-1}(B)$, i.e. such that $S(r^{(i)}(n)) \leq B$.

Proof: This is immediate from the recursive structure of the HBP computation. Let τ be a task corresponding to either a BP computation or to a recursive computation in $\mathcal{A}$. τ adds $O(\min\{S(|\tau|), B\})$ to ℓ_2 and ℓ_4. There are c^i tasks of size $r^{(i)}(n)$. When $i < r^*(n,B)$, the size $r^{(i)}(n)$ is larger than B, so we add $O(B)$ to $\ell_2(v)$ and $\ell_4(v)$. This is the first summation. When $i \geq r^*(n,B)$, the size becomes smaller than B, so in the second summation we add the space bound of the task, $S(r^{(i)}(n))$ instead. ■

Recall that $\overline{\ell}(D) = (1/B)\cdot(l_2(t)+l_4(t))$. We bound $\overline{\ell}(D)$ in terms of the recursive structure of the HBP algorithms in the following theorem. Once we have $\overline{\ell}(D)$, we can readily apply Theorem I.2 to obtain the bounds on the number of steals under RWS given in the last column in Table I.

Theorem V.2. *Let $\mathcal{A}$ be a block-resilient Type 2 HBP algorithm that uses linear space. Recall that $c \geq 1$ denotes the number of collections of recursive calls made by $\mathcal{A}$, and that $r(n)$ is a bound on the size of the recursive subproblems called by $\mathcal{A}$. Then, $\overline{\ell}(D)$ is bounded as follows.*

(i) $c = 1$: $\overline{\ell}(D) = O(r^*(n,B))$, where $r^*(n,B)$ is the number of applications of r needed to reduce n to at most B.

(ii) $c = 2$ and $r(n) = \sqrt{n}$: $\overline{\ell}(D) = O(\frac{\log n}{\log B})$.

(iii) $c = 2$ and $r(n) = n/4$: $\overline{\ell}(D) = O(\sqrt{n/B})$.

(iv) $c = 3$ and $r(n) = n/2$: $\overline{\ell}(D) = O(n^{\log_2 3}/B)$.

Proof: (i) follows immediately from Lemma V.1. For (ii), we have $r^*(n,B) = \log(\frac{\log n}{\log B})$; on substituting in the bound from Lemma V.1, the result is immediate; similarly, for (iii) $r^*(n,B) = \log\sqrt{n/B}$, and for (iv), the dominant term is the final term in the second sum. ■

These choices of c and $r(n)$ are the ones that occur in our Type 2 HBP algorithms [10].

- MM, Strassen, and BI to RM for FFT are covered by Case (i),
- FFT and sort by Case (ii),
- Depth n MM by Case (iii), and
- the basic LCS (i.e. the algorithm computing the optimal cost, but not an optimal sequence) by Case (iv).

The full LCS algorithm that finds an optimal sequence, in addition to its optimal length, is a Type 3 algorithm and I-GEP is a Type 4 algorithm; to analyze these, we need the generalization of Lemma V.1, given below.

The list ranking (LR) and graph connected components (CC) algorithms employ the gapping technique, explained below, which leads to yet another bound. (The remaining algorithms we analyze are BP algorithms, and the bound for steals for BP algorithms was derived at the end of Section IV-A.)

As an example, MM and Strassen are Case (i) with $r(n^2) = n^2/4$, $c = 1$, and $T_\infty = O(\log^2 n)$. Hence we obtain $r^*(n,B) = O(\log(n/B)) = O(\log n)$ and so, $\overline{\ell}(D) = O(\log n)$. Using Theorem I.2, this gives the bound on steals reported in Table I.

Next, we provide the lemma needed to bound ℓ_2 and ℓ_4 for Type $t > 2$ algorithms.

Lemma V.3. *Let $\mathcal{A}$ be a block-resilient, Type $t > 2$ HBP algorithm which uses space $S(n)$. Suppose that each recursive call $\mathcal{A}$ makes has size at most $r(n) \leq n/b$, for some constant $b > 1$. Let $c \geq 1$ denote the number of collections of recursive calls made by $\mathcal{A}$ and let $\mathcal{A}$ have size n. Suppose that for the non-recursive calls made by $\mathcal{A}$, their ℓ_2 and ℓ_4 parameters are bounded by $x(n,B)$. Then for every node v in $\mathcal{A}$'s computation dag D,*

$$\ell_2(v), \ell_4(v) = O(\ \textstyle\sum_{i<r^*(n,B)}(B + x(n,B)) \cdot c^i + \sum_{i\geq r^*(n,B)} c^i \cdot [S(r^{(i)}(n)) + (x(r^{(i)}(n), B)]\ ),$$

where $i = r^*(n,B)$ is the number of applications of r needed to reduce n to at most $S^{-1}(B)$, i.e. such that $S(r^{(i)}(n)) \leq B$.

Proof: This is again immediate from the recursive structure of the HBP computation. ■

Corollary V.4. *i. For LCS, $\overline{\ell}(D) = O(n^{\log_2 3}/B)$.*
ii. For I-GEP, $\overline{\ell}(D) = O(n/\sqrt{B})$.

Proof: We start with i. LCS comprises an initial computation that amounts to the Type 2 basic LCS algorithm (with some intermediate information being saved); it is followed by a single collection ($c = 1$) of recursive calls on problems

of size $n/2$. Substituting in Lemma V.3 gives the bound $O(\sum_{i\geq 0}(n/2^i)^{\log_2 3}) = O(n^{\log_2 3})$ on ℓ_2 plus ℓ_4.

For ii, we recall from [15] that the Type 4 I-GEP has $c = 2$ with $r(n^2) = n^2/4$; in addition it calls Type 2 and 3 procedures. The Type 2 procedures are Depth-n-MM. The Type 3 procedure also has $c = 2$ with $r(n^2) = n^2/4$; in addition it also calls MM.

By Theorem V.2, ℓ_2 and ℓ_4 for Depth-n-MM are bounded by $O(n\sqrt{B})$. Substituting in Lemma V.3 gives the bound $O(\sum_{i\geq 0}(n/4^i)\cdot 2^i\sqrt{B}) = O(n\sqrt{B})$. The same substitution yields the bound $O(n\sqrt{B})$ on ℓ_2 plus ℓ_4 for the Type 4 I-GEP algorithm. ■

Finally, we consider the List Ranking (LR) and connected components (CC) algorithms.

The LR Algorithm: We modify the LR algorithm in [10] by introducing gaps in the recursive subproblems as described below. The LR algorithm in [10] has two stages. The first stage performs $O(\log\log n)$ stages of eliminating a constant fraction of the elements in the linked list and, when the size of the linked list falls below $n/\log n$, switches to the second stage, which is the basic pointer jumping algorithm. To find a large independent set in a linked list of size r, the algorithm constructs an $O(\log^{(k)} r)$-size coloring of the linked list, and then extracts an independent set of size at least $r/3$ by examining elements of each color class in turn. A phase on a list of length r performs $O(\log^{(k)} r)$ calls to SPMS sort on inputs whose combined length is r, and incurs $O(\frac{r}{B}\log_M r)$ cache misses in parallel time $O(\log r\cdot\log\log r\cdot\log^{(k)} r)$. The algorithm switches to pointer jumping when the list has length $O(n/\log n)$, and its overall cost is $O(n\log n)$ work, $O((n/B)\log_M n)$ cache misses, and parallel time $O(\log^2 n\log\log n)$.

By modifying the data layout in the recursive calls, we are able to reduce the initial values of ℓ_2 and ℓ_4 for the corresponding tasks. To this end, we introduce gaps between the elements of the contracted linked list as follows: When the list has size n/x^2, it is written in space n/x, using every xth location only. Thus, when the list has size n/B^2 or less, for each subtask τ, ℓ_2^{BP} and $\ell_4(v)$ can be reduced to 0. Thus there are $O(\log^{(k)} n\cdot\min\{\log B,\log\log n\}+\log B)$ iterations of the sorting algorithm that each contribute $O(B\log_B n)$ to the initial values of $\ell_2(t)$ and $\ell_4(t)$; the remaining $O(\log n)$ iterations add nothing further. This yields a bound of $O(B\log n\log^{(k)} n) = O(B\log n\cdot\log\log n)$ on the initial values of $\ell_2(t)$ and $\ell_4(t)$ (the second bound follows by choosing $k = 2$). As $\bar{l}(D)$ is $1/B$ times the initial value of $\ell_2(t)+\ell_4(t)$, on substituting in Theorem I.2, we obtain the bound in Table I. (It can be verified that this modification of the list ranking algorithm does not affect the cache miss cost beyond a constant factor.)

The CC algorithm: This has $O(\log n)$ iterations of the LR algorithm and thus all bounds increase by an $O(\log n)$ factor.

VI. Optimal Speed-Up Bounds for Individual Algorithms

In this section, we determine for each algorithm we have considered, the input size above which we can guarantee, with high probability in the input size, that the worst case overhead in cache miss and fs miss costs due to steals under RWS is dominated by the sequential cache miss cost for the algorithm. For these input sizes, our analysis has established that the cache and fs miss overhead incurred due to the use of RWS is within a constant factor or less of the inherent cache miss cost of the computation, even in the sequential setting. We call this bound on the input size the *optimal speed-up bound* for the given algorithm.

The optimal speed-up bounds are shown in Table II. We now derive these bounds for the individual algorithms. Some of these algorithms are described in the previous section, and the descriptions of the remaining algorithms can be found in [10]. In the analysis that follows we factor out the parameter b as everything is being measured in the cache miss cost b.

Scans: The algorithm incurs $C = O(S)$ cache miss excess and a delay $F = O(S\cdot B)$ due to fs misses. As $S = O(p\cdot(\log n+B))$ (using the assumption that $b\leq s$), the combined cache and block miss delay is $O(p\cdot B\cdot(\log n+B))$. The sequential cache miss delay is $Q = O(n/B)$. For optimal speedup we therefore need $p\cdot B\cdot(\log n+B) = O(n/B)$, i.e. $n\geq p\cdot B^2\cdot(\log n+B)$.

MT, RM to BI: The analysis is the same as for scans except that the problem size is n^2 rather than n.

MM: Here the condition becomes $S^{\frac{1}{3}}\frac{n^2}{B}+S\cdot B\leq\frac{n^3}{B\sqrt{M}}$, where $S = O(p\cdot(\log^2 n+B\log n)$. This simplifies to $S\sqrt{M}\cdot(B^2+M)\leq n^3$. This yields $n^3\geq p\sqrt{M}\log n(B+\log n)(B^2+M)$.

Strassen: The analysis is as for MM, except that n^λ replaces n^3 and $S^{1/\lambda}\cdot\frac{M}{B}(\frac{n}{\sqrt{M}})^{\lambda-1}+S\cdot B\leq\frac{n^\lambda}{M^{\lambda/2-1}B}$ yielding the constraint $n^\lambda\geq pM^{\lambda/2-1}(M+B^2)\log n(B+\log n)$.

Depth n MM: As for MM, we have the constraint $S\sqrt{M}\cdot(B^2+M)\leq n^3$, but here $S = O(p\cdot n\sqrt{B})$, which yields the constraint $p\sqrt{BM}\cdot(B^2+M)\leq n^2$.

I-GEP: As for MM, we have the constraint $S\sqrt{M}\cdot(B^2+M)\leq n^3$, but here $S = O(p\cdot n\cdot(\log^2 n+\sqrt{B}))$, which yields the constraint $p(\log^2 n+\sqrt{B})\sqrt{M}\cdot(B^2+M)\leq n^2$.

LCS: The sequential computation incurs $n^2/(MB)$ cache misses. By Lemma V.1 and Theorem V.2, there are $O(p\cdot n^{\log_2 3})$ steals (here $r(n) = n/2$ and $c = 3$). As the cache miss excess is $C = \frac{n\sqrt{S}}{B}+S\cdot\log B$ this yields the constraint $\frac{n\sqrt{S}}{B}+S\cdot B\leq\frac{n^2}{BM}$ or $S\leq n^2(\frac{1}{B^2M}+\frac{1}{M^2})$, and on substituting for S yields the constraint $p\cdot B\cdot M\cdot n^{\log_2 3}(B^2+M)\leq n^2$.

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 $S \cdot B \leq \frac{n}{B} \log_M n$. Now $S = O(p \cdot (\log n \log\log n + B \log_B n)$. This yields the constraint $p \cdot (\log n \log\log n + B \log_B n)(M^\epsilon + B^2 \log M) \leq n \log n$, i.e. $n \geq p \cdot (\log\log n + B/\log B)(M^\epsilon + B^2 \log M)$.

BI to RM for MM and FFT: We are interested in this algorithm only as a front end to MM and FFT. Thus it suffices to note that its costs are always dominated by those for MM and FFT. Thus whenever the latter achieve optimal speedup this is unaffected by using BI to RM for MM and FFT as a front end, i.e. using an RM rather than BI format for the inputs and outputs.

List Ranking, LR: The analysis is as for sorting except that the bound on S changes. For list ranking this yields the constraint $p \cdot (\log^2 n \log\log n + B \log n \log\log n)(M^\epsilon + B^2 \log M) \leq n \log n$, i.e. $n \geq p \cdot \log\log n \cdot (\log n + B)(M^\epsilon + B^2 \log M)$.

Connected Components, CC: All costs increase by a $\log n$ factor from LR, giving the same optimality bound as for LR.

ACKNOWLEDGMENT

This work was supported in part by NSF Grants CCF-0830516 and CCF-1217989 (Richard Cole) and CCF-0830737 (Vijaya Ramachandran).

REFERENCES

[1] F. Burton and M. R. Sleep, "Executing functional programs on a virtual tree of processors," in *Proc. ACM Conf. on Func Prog Languages and Comp Arch*, 1981, pp. 187–194.

[2] R. H. J. Halstead, "Implementation of Multilistp: Lisp on a multiprocessor," in *Conf. ACM Symp. on LISP and Functional Programming*, 1984, pp. 9–17.

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

[4] 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.*, vol. 30, pp. 207–216, 1995.

[5] A. Robison, M. Voss, and A. Kukanov, "Optimization via reflection on work stealing in tbb." in *IPDPS*. IEEE, 2008, pp. 1–8. [Online]. Available: http://dblp.uni-trier.de/db/conf/ipps/ipdps2008.html#RobisonVK08

[6] T. Gautier, X. Besseron, and L. Pigeon, "Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors," in *Proceedings of the 2007 international workshop on Parallel symbolic computation*, ser. PASCO '07, 2007, pp. 15–23. [Online]. Available: http://doi.acm.org/10.1145/1278177.1278182

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

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

[9] R. Cole and V. Ramachandran, "Revisiting the cache miss analysis of multithreaded algorithms," in *Proceedings of the Tenth Latin American Theoretical Informatics Symposium*, ser. LATIN '12, 2012.

[10] ——, "Efficient resource oblivious algorithms for multicores with false sharing," in *Proc. IEEE IPDPS*, 2012.

[11] ——, "Resource oblivious sorting on multicores," in *Proc. ICALP Track A*, 2010.

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

[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 2006.

[14] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proc. FOCS*, 1999, pp. 285–297.

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