Efficient Resource Oblivious Algorithms for Multicores 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*—We consider algorithms for a multicore environment in which each core has its own private cache and false sharing can occur. False sharing happens when two or more processors access the same block (i.e., cache-line) in parallel, and at least one processor writes into a location in the block. False sharing causes different processors to have inconsistent views of the data in the block, and many of the methods currently used to resolve these inconsistencies can cause large delays.**

We analyze the cost of false sharing both for variables stored on the execution stacks of the parallel tasks and for output variables. Our main technical contribution is to establish a low cost for this overhead for the class of multithreaded block-resilient HBP (Hierarchical Balanced Parallel) computations. Using this and other techniques, we develop block-resilient HBP algorithms with low false sharing costs for several fundamental problems including scans, matrix multiplication, FFT, sorting, and hybrid block-resilient HBP algorithms for list ranking and graph connected components. Most of these algorithms are derived from known multicore algorithms, but are further refined to achieve a low false sharing overhead.

Our algorithms make no mention of machine parameters, and our analysis of the false sharing overhead is mostly in terms of the the number of tasks generated in parallel during the computation, and thus applies to a variety of schedulers.

***Keywords*-false-sharing; cache-efficiency; multicores**

I. INTRODUCTION

We consider the design of efficient multithreaded algorithms (see, e.g., [15], chapter 27) with low false sharing overhead for a multicore computing environment. We model a multicore as consisting of p cores (or processors) with an arbitrarily large shared memory, where each core has a private cache of size M. Data is organized in blocks of size B, and the initial input of size n is in the main memory, in n/B blocks.

There has been considerable work recently on developing efficient algorithms for multicores [20], [9], [4], [10], [11], [5], [23], [2], [17]; many of these algorithms are multithreaded. An efficient multicore algorithm attempts to obtain both work-efficient speed-up and cache-efficiency. However, none of these prior results has addressed false sharing costs when considering cache-efficiency, with the exception of our sorting algorithm in [12]; but even there, only some of the possible instances of false sharing were considered.

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

Cache Misses and False Sharing. When a core C needs a data item x that is not in its private cache, it reads in the block (i.e., cache-line) β that contains x from main memory at the cost of one cache miss. This new block replaces an existing block in the private cache. If another core C' modifies an entry in block β, then C and C' will have different local views of the block β, and this will need to be resolved in some manner the next time core C needs to access data in β. A commonly employed mechanism in current multicores is to use a *cache coherence protocol* in which the hardware invalidates the copy of β in C's cache when C' modifies β, and the next time core C needs to access data in β, an updated copy of β is brought into C's cache. In the absence of cache coherence, some method of assigning ownership to a block is needed so that the updates to items in a block are correctly performed. For instance, the Backer protocol [8] is an alternate method to handle this.

The situation described above in which two or more cores access portions of the same block during a time interval when at least one core writes into the block is called *false-sharing*. The delay caused by false sharing can be quite significant, and this is a caching delay that is present only in the parallel context. For instance, consider a parallel execution in which $q \geq 2$ cores between them perform multiple accesses to a block β, which include $x \geq 1$ writes. These accesses could cause $\Omega(b \cdot x)$ delay at $\Omega(q)$ cores accessing β, where b is the delay due to a single cache miss. These costs might arise if two cores are sharing a block (which occurs for example if data partitioning does not match block boundaries) or if many cores access a single block (which could occur if the cores are all executing very small tasks). Further, x can be arbitrarily large unless care is taken in the algorithm design. We will refer to any access of a block that is not in cache due to false sharing as a *false sharing miss*, or an *fs miss* for short.

The cost of false sharing is drawing renewed attention with the advent of multicore computing. However, a similar communication cost is present in any parallel computation. For instance, many parallel models such as the BSP [22]

Table I
FS MISS BOUNDS FOR SOME OF OUR BLOCK-RESILIENT HBP ALGORITHMS; $O(\ \cdot\ )$ OMITTED ON EVERY TERM.
$L(r)$ IS THE BLOCK-SHARING FUNCTION AND S_C IS A CENTRALIZED SCHEDULER THAT DOES NOT KNOW THE VALUE OF B.
THE INPUT SIZE IS n, EXCEPT FOR MATRIX COMPUTATIONS, WHERE THE INPUT SIZE IS n^2; λ IS 3 FOR MM AND $\log_2 7$ FOR STRASSEN.
ALL CACHE AND FS MISS COSTS ARE IN UNIT OF b, THE COST OF ONE CACHE MISS.

Block Resilient HBP Algorithm	**New Results**		**Known Results**			
	$\boldsymbol{L(r)}$	**Fs Misses with Z Parallel Tasks**	**Value of Z for Scheduler S_C**	**Sequential Cache Misses Q**	**Cache Misses w/ Z Parallel Tasks [14]**	**Critical path-length T_∞**
Scans (PS, MT)	1	$B \cdot Z$	p	n/B	$Q + Z$ [19], [14]	$\log n$
Depth-n-MM	1	$B \cdot Z$	$p^{3/2}$	$n^3/(B\sqrt{M})$	$Q + Z^{\frac{1}{3}} \frac{n^2}{B} + Z$ [19], [14]	n
MM, Strassen	1	$B \cdot Z$	$p \log p$	$n^\lambda/(B \cdot M^{\frac{\lambda}{2}-1})$	$Q + Z^{\frac{1}{3}} \frac{n^2}{B} + Z$	$\log^2 n$
RM to BI	1	$B \cdot Z$	p	n^2/B	$Q + Z \cdot B$	$\log n$
Direct BI to RM	$\sqrt{r}$	$\frac{n}{\sqrt{p}} B \cdot Z$	p	n^2/B	$Q + Z \cdot B$	$\log n$
BI-RM (gap RM)	gap	$\min\{\frac{n}{\sqrt{p}}, B \log^2 B\} B \cdot Z$	p	n^2/B	$Q + Z \cdot B$	$\log n$
BI-RM for FFT	1	$B \cdot Z$	$p \cdot \log \frac{\log n}{\log(n^2/p)}$	$\frac{n^2}{B} \log \log_M n$	$Q + ZB + \frac{n^2}{B} \log \log_B n$	$\log n$
FFT, Sort [12]	1	$B \cdot Z$	$p \cdot \frac{\log n}{\log(n/p)}$	$\frac{n}{B} \log_M n$	$Q + Z \cdot B$	$\log n \cdot \log \log n$
LR (Hybrid HBP)	1	$B \cdot Z$	$p \cdot \frac{\log^2 n}{\log(n/p)}$	$\frac{n}{B} \log_M n$	$Q \log n + Z \cdot B$	$\log^2 n \cdot \log \log n$

consider the cost of sending packets between processors. But there is also an implicit cost associated with assembling and unpacking these packets; this can be viewed as being part of the cost in the 'o' term in the LogP [16] model. Thus, in any parallel environment, one either pays for packing and unpacking data in packets that are moved between processors, or one has to deal with false sharing misses if the data is prepackaged in blocks. Distributed memory architectures incur the former type of costs, and these costs are typically masked by the large cost of transporting a packet across the communication network. The architectures for current multicores incur the latter type of costs, and further, data is accessed at a fine-grained level. Hence we have the current need to develop algorithmic strategies to minimize the cost of fs misses. However, we note, that in one way or another, this cost is present in any parallel environment. The algorithmic techniques we present in this paper for reducing the cost of fs misses will also help to reduce the packing and unpacking costs in other parallel environments, since we minimize the number of data boundaries at which different tasks interact.

Resource Obliviousness. The notion of a *cache oblivious* algorithm was introduced in [18], where a sequential algorithm is not allowed the use the cache parameters, namely the cache size M and the cache line or block size B, although these parameters are used in the analysis of the caching performance of the algorithm.

In a multicore that supports a multithreaded parallel environment, we are interested in both parallelism and cache efficiency. The notion of a *multicore oblivious* algorithm was introduced in [11]. Such an algorithm uses no multicore parameters but is allowed to include some simple hints to the run-time scheduler. It was shown in [11] that these hints could be used by a suitable scheduler (that knows the multicore parameters) to efficiently schedule the algorithm on multicores with a multilevel cache hierarchy and any given number of cores. The *PCO model* [6] is a variant of the multicore oblivious framework in [11], again for a multilevel cache hierarchy. The results in [11], [6] consider cache misses, but as with most prior work on multicore algorithms, assume that no false sharing occurs.

An algorithm that is efficient while being oblivious to machine parameters has the benefit of maintaining its efficiency across a range of machines, resulting in portable efficient code. In a multicore environment we also have the issue that a multicore algorithm could be scheduled in different ways, leading to different performance bounds. For instance, the multicore oblivious algorithms in [11] were shown to achieve efficiency under a specific scheduler.

In this paper, we confine our attention to multicores with private caches, and we present multicore algorithms whose efficiency we analyze in terms of the number of parallel tasks generated during the computation (and in some cases, also as a function of the task sizes). Thus, our algorithms can be analyzed across a wide range of schedulers. We demonstrate the efficiency of our algorithms with a simple centralized scheduler. We have also established efficiency under the Priority Work Stealing (PWS) scheduler [12], and in recent work [13] we establish bounds for our algorithms under the well-known randomized work-stealing (RWS) scheduler. We use the term *resource oblivious* to denote that our algorithms are independent of machine parameters, and yet are analyzed to run efficiently under different schedulers.

The focus of this paper is on reducing the cost of false sharing; in a companion paper [14] we show that the same class of algorithms has good cache miss bounds, again as a function of the number of parallel tasks generated during the computation. Thus, combining the results in the current paper with those in [14], we have a collection of resource oblivious multithreaded algorithms for several fundamental problems that are efficient with respect to both cache miss overhead and false sharing costs when scheduled by a run-time scheduler that is economical in the number of parallel tasks that it generates.

Our Contributions. Our main contribution is to set up a framework to analyze – and optimize for – the caching overhead of fs misses while also allowing for high parallelism. We start with a basic primitive, the *Balanced Parallel (BP)* computation; *Hierarchical Balanced Parallel (HBP)* computations are obtained through sequencing and parallel recursion [12]. We further introduce the notion of a *block-resilient* computation, for which we establish that any shared block is accessed $O(B)$ times and hence incurs $O(B)$ fs misses. The main challenge here is in bounding the accesses to blocks stored on the execution stack of recursive procedures that have parallelism. However, even outside of the execution stack, a shared block can have an unbounded number of accesses in the fairly asynchronous environment in which we schedule our algorithms, if care is not taken with the algorithm design. We present additional techniques for reducing the cost of fs misses, namely *gapping*, and minimizing the *block sharing function* $L(r)$; in most of our algorithms $L(r) = O(1)$, and when that is not achieved, we resort to gapping to reduce the cost of fs misses.

We present and analyze block resilient HBP algorithms for scans, matrix transposition (MT), matrix multiplication, converting a matrix between row major (RM) and bit interleaved (BI) layouts, and FFT, and hybrid HBP algorithms for list ranking (LR) and graph connected components (CC) (a hybrid HBP algorithm is obtained by sequencing $\omega(1)$ HBP computations). Most of these are known algorithms, but some are new; also many of the known algorithms are modified to make them block-resilient. These algorithms will achieve relatively low fs miss cost under most schedulers; we illustrate their performance by analyzing their fs miss cost when scheduled by the simple centralized scheduler, which we call S_C, that is used in [9] (S_C does not know the block size). A summary of our results is in Table I, where, for each algorithm, $\Delta(n, p, B)$ gives an upper bound on the worst case cost of fs misses across all processors (measured in units of cache miss cost), when scheduled by S_C on p processors, where B is the block size, which is unknown to S_C.

Roadmap. In Section II we describe the multithreaded computation model, and we discuss our set-up for fs miss costs and introduce the block-sharing function $L(r)$. In Section III we define the classes of BP and HBP algorithms, and in Section IV we describe the data layout used in our algorithms. In Section V we define the notion of block-resilience, and we establish the important technical result that in a block-resilient HBP computation, any shared block is transferred at most $O(B)$ times, where B is the number of words in a block. In Section VI we present block-resilient HBP algorithms for several problems, and in Section VII we illustrate our results by analyzing the fs miss overhead of these algorithms using the centralized scheduler S_C. We conclude in Section VIII.

II. Computation Model

We model a computation as a directed acyclic graph (dag) D (good overviews can be found in [15], chapter 27, [7]). 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:

1. Sequencing, where the terminal node of G_1 is connected to the start node of G_2.
2. 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 task represented by graph G decomposes into either a sequence of two subtasks (corresponding to G_1 and G_2 in (i)) or 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, which is 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.

Parallel tasks can be scheduled on cores using either a centralized scheduler or a distributed work stealing scheduler. In either case, each processor will execute a sequence of tasks during the computation, where a task is a fragment of a sequential execution of the multithreaded computation. Following the work-stealing terminology, we will often refer to any such task, other than the initial task that starts the execution, as a *stolen task*. We will also use the term *parallel task* to denote either the initial task or a stolen task.

Execution Stack. We now describe 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. Before elaborating we define task kernels.

Definition II.1. *For a task τ, its* task kernel *τ^K is the portion of τ's computation dag that remains after the computation dags for all its stolen subtasks are removed.*

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. 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. For each procedure the thread initiates, it creates a *segment* to hold the variables declared by the procedure. The segment is placed at the top of S_τ. When the procedure completes, the space used by the segment is released.

Usurpations. As the execution of a task kernel τ^K proceeds, the processor executing it may change. This change can occur at a join node v at which a stolen subtask τ' ends, and it occurs if the processor C' that was executing τ' reaches the join node after the processor C that had been executing τ^K. 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. 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. *Cache Misses*

The parallel execution of a multithreaded algorithm causes it to incur additional cache misses over those incurred in a sequential execution, and it also introduces fs misses. In this paper we consider the overhead due to fs misses, but we first outline our treatment of cache misses in [14]. Following the cache-oblivious model [18], an optimal cache replacement policy is assumed at the private cache of each processor.

The dependence of cache miss costs on the data layout has been widely addressed in the extensive work on cache efficient algorithms, in both the sequential and parallel setting. In [14] we set up a systematic method for analyzing the cache miss costs in a computation by defining the following cache-friendliness function $f(r)$.

Definition II.2. *A task τ that accesses r words of data is* f-cache friendly *if these r words of data are contained in $O(r/B+f(r))$ blocks. A computation has* cache friendliness function $f(r)$ *if every task in it is f-cache friendly.*

For instance, $f(r) = 1$ if τ accesses an array stored in contiguous locations; if τ access a $\sqrt{r} \times \sqrt{r}$ submatrix of a matrix stored in RM (i.e., row major) order, then $f(r) = \sqrt{r}$.

Since we consider the overhead of fs misses in this paper, our analysis does not use the cache-friendliness function $f(r)$; instead it uses the block sharing function $L(r)$ defined in the next section. Nonetheless, the algorithms we present in this paper all continue to achieve the cache miss bounds established in [14] while additionally achieving the upper bounds on the cost of fs misses that we establish here.

B. *False Sharing Misses*

An fs miss occurs if two or more processors access values stored in a block β within the same time interval, and at least one of the accesses is a write. We refer to such a block β as a *writable block* in this time interval. We assume that fs misses are handled under a cache coherence protocol whereby a write into a location in a shared block β by core C invalidates the copy of β in every other cache that holds β at the time of the write. This is done to maintain data consistency across all copies of a block in caches at all times. There are other ways of dealing with fs misses (see, e.g., [21]), but we believe that the fs miss cost with our invalidation rule 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.

We do not make any assumptions about the manner in which a shared block is transferred to processors that need it and have invalidated their local copy. The only assumption we make about fs misses is that a processor cannot unduly delay transferring a requested block. For specificity, we assume that the delay for a single block transfer is of the same magnitude as the cache miss delay. Thus, the cost of an fs miss is at least that of one cache miss, but it could be much larger, depending on the number of cores that share a block β and write into it; in fact, the cost of an fs miss could be unbounded in a scenario where several cores repeatedly write into locations in the block, if the system mechanism for transferring access to the block does not ensure fairness. Our upper bounds apply even to such situations, hence the bounds we obtain are truly worst-case.

Definition II.3. *Suppose that block β is moved m times from one cache to another due to fs misses during a time interval $T = [t_1, t_2]$. Then m is defined to be the* block delay *incurred by β during T.*

The block wait cost *incurred by a task τ on a block β is the delay incurred during the execution of τ due to fs misses when accessing β, measured in units of cache misses.*

Note that the block wait cost incurred by a task τ on a block β is the delay incurred due to fs misses as measured in units of cache misses. Clearly, the block delay of a block β during a time interval T is an upper bound on the block wait cost incurred by any task on block β during T, though the latter could be smaller for some or all tasks. In this paper we will usually use the block delay incurred by β to upper bound the block wait cost at each task that accesses β. (In our recent work on false sharing costs under randomized work stealing [13], which builds on the current paper, we perform a more refined analysis to bound the block wait costs instead of simply using the block delay.)

For blocks storing data on the execution stack, we view a block β as existing for a time interval T, which begins when a variable is first added to β and ends when that variable is released. If β is subsequently reused to store another portion of the execution stack, those accesses will not contribute to the block wait cost for accesses during T, hence for the purposes of analyzing the overall delay τ incurs, we can view this as a new block.

Definition II.4. *A task τ of size r is* $L(r)$-block sharing, *if there are at most $O(L(r))$ writable blocks that τ can share with all other tasks that could be scheduled in parallel with τ and that could access a location in the block. A computation has block sharing function L if every task in it is L-block sharing.*

Definition II.5. *An algorithm is* limited-access *if each of its writable variables is accessed $O(1)$ times.*

The two main algorithmic techniques that we use to reduce the cost of fs misses are to enforce limited access and to try to obtain $O(1)$-block sharing. In some algorithms, we also use a *gapping* technique to reduce the fs miss cost. These techniques reduce the number of blocks in a task that are subject to fs misses as well as the number of accesses that give rise to fs misses in a given block, and are thus helpful in reducing the block delay. However, there is inevitable contention at a block storing a portion of the execution stack of a task, and this contention can give rise to significant block delays especially in recursive algorithms. We address this cost in Section V, where we obtain manageable bounds on the block delay for the class of block-resilient hierarchical balanced parallel (HBP) computations. Algorithms for many common problems fall into this class, as we show in Section VI.

III. HBP ALGORITHMS

We present the definition of HBP computations, introduced in [12]. In the following, 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).

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 balanced-tree procedure to compute the sum of n integers.

Definition III.2. *A* Hierarchical Balanced Parallel (HBP) Computation *is one of the following:*
1. A Type 0 Algorithm, a size $O(1)$ sequential computation.
2. A Type 1 Algorithm, a BP computation.
3. Sequencing. A Type t sequenced 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 of the sequenced HBP algorithms.
4. Recursion. A Type $t+1$ recursive *HBP algorithm, for $t \geq 1$, results if, on an input of size n, 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 $r(n) \leq \alpha n$ for a constant $0 \leq \alpha < 1$. Often, we will let $b = 1/\alpha$.*

Each of the c collections can be preceded and/or followed by a sequenced HBP algorithm of type at most t, and 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.

In the rest of the paper we will use $t,\ c,\ v,\ r,\ \alpha$ and b as specified in the above definition.

Matrix Multiply (MM) with 8-way recursion is an example of a Type 2 HBP algorithm. Given as input two $n \times n$ matrices to multiply, it 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 [18], [11] is another Type 2 HBP algorithm for MM with $c = 2$, $v(n^2) = 4$, and $r(n^2) = n^2/4$. For both algorithms $f(r) = \sqrt{r}$ if the matrices are in row major (RM), and $f(r) = O(1)$ if they are in bit-interleaved (BI) format.

The HBP class is closely related to the *Hierarchical Divide and Conquer (HD&C)* class in [4] (after the parallelism is exposed in the HD&C algorithms). The main differences are that we allow sequencing of HBP computations even at the top level, and that we allow a non-constant number of subproblems to be called recursively.

An HBP algorithm is *block-resilient* if (i) it is limited-access, (ii) it satisfies certain requirements on the data layout, and (iii) it is *top dominant*. We specify the data layout requirements in Section IV and we define top dominance in

Section V. Block-resilience will be key to reducing the cost of fs misses on the execution stacks of tasks.

IV. Data Layout

The layout of data plays an important role in the number of cache and fs misses incurred by a computation. In this section we describe the data layout that our HBP algorithms will use.

We start by defining local and global variables.

Definition IV.1. *A variable x declared in a procedure P is called a* local *variable of P. A variable y accessed by P and declared in a procedure Q calling P or used for the inputs or outputs of the algorithm $\mathcal{A}$ containing P is said to be* global *with respect to P. Note that y would be a local variable of Q if declared in Q.*

An algorithm's input and output variables are always 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.

A. *Execution Stack*

Let τ be a task of size n in a Type 2 HBP computation. The execution stack S_τ for τ consists of an initial segment of size $S(n)$ for τ's local variables and then $\log v(n)$ segments each of $O(1)$ length to keep track of parallel recursive calls; each recursive call will then create similar entries following the initial entries for τ. The topmost segment on S_τ will either be one of the above types of segments or, if τ is currently executing a BP computation τ' within the HBP computation, at the top there will be $O(\log|\tau'|)$ segments of length $O(1)$ for this BP computation. An analogous organization applies to type $t > 2$ HBP computations.

The execution stack S_τ is created by the processor C that starts τ's execution. The variables on S_τ may be accessed by stolen subtasks, which can cause fs misses. Also, if another processor C' takes over τ's execution through usurpation (i.e., by being the second, and hence last, processor to finish the work preceding a join, see Section II), then C' will continue using S_τ for the remainder of τ's computation (at least until yet another processor takes over τ's computation). The analysis of fs miss costs needs to handle the possible resulting moves of blocks storing the execution stack due to the usurpation.

In addition to the variables stored on the execution stacks, an HBP algorithm $\mathcal{A}$ needs variables in which to store its output. These 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. Also, note that in an HBP algorithm $\mathcal{A}$, the only variables guaranteed to be non-writable are those for the input, which again is not stored on any execution stack.

B. *Data Layout in a BP Computation*

Recall that a BP computation comprises a tree T_d of forking nodes, called the *down-pass* tree, followed by leaf nodes, followed by a complementary tree T_u of join nodes, called the *up-pass* tree. Also, recall that each node performs $O(1)$ operations. We now specify some data layout requirements on both global and local variables. These are not required for the results in this paper, but they are needed for our results on cache and fs misses under randomized work stealing [14], [13], and all of the algorithms we present in this paper satify these requirements.

Global Variables: We assume that the writes to the global variables (typically arrays of size n) obey the following *well-buffered rule*.

Well Buffered Rule. Let v be a node in the down-pass tree 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. An analogous definition applies to nodes in the up-pass tree.

Note that prefix-sums can be implemented as a sequence of two well-buffered BP computations, where the input data is viewed as being at the leaves of a balanced binary tree. The first BP computation computes the sums of values in each subtree, storing them in infix order; these are the inputs to the second computation which finds the prefix sums.

Local Variables: The local variables are used to store the data needed by individual computation nodes; by the BP definition, there are at most $e = O(1)$ such (one-word) variables per node. When the computation of a task τ begins, its local variables are added to its thread's execution stack, and when it ends, this space is released.

We limit the BP algorithms we consider to obey the following constraint (achieved by a natural scoping of variables and the use of return value variables).

Local Constraint. Writes to local variables by the task for a node v are to v's local variables, and in the up-pass possibly to the local variables at $parent(v)$ in T_d.

C. *Data layout in an HBP computation*

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 constraint:

HBP Data Accesses. In order to enforce limited access writes, a recursive call performs its writes to the array of variables of its calling program by means of a BP computation that occurs at the end of the recursive call. The collection of these BP computations terminating the recursive calls obeys the well buffered rule for a BP collection when accessing an array A.

V. Bounding the Transfers of a Single Block

The limited access property states that each writable variable is accessed $O(1)$ times, and this would appear to bound the block delay of any block by $O(B)$. However, due to procedure calls, including recursive ones, over time more than B variables could all be stored in a block β on an execution stack, and so the limited access property does not suffice to yield the $O(B)$ bound that we seek on the number of accesses to β.

Consider, for instance, the execution stack S_τ of a BP computation τ. S_τ has height $O(\log n)$, where n is the size of τ. However, a block on S_τ stores the data for a subtree of τ of depth that can be up to $d = \Theta(\min\{B, \log n\})$, and hence up to $2^{\Theta(d)}$ different values can be stored in β during this computation, even though it is a limited-access computation.

In this section we establish that for a block β on the execution stack S_τ of any task τ in the class of *block-resilient* HBP algorithms, the block delay $Y(\tau, B)$ is in fact bounded by $O(B)$; block-resilience is defined in Definition V.6. Since a block delay of $\Theta(B)$ is inevitable in an asynchronous setting even if every location in the block is accessed only once, this is an optimal upper bound on the block delay for a block on the execution stack.

In Section V-A we present some preliminary results. In Sections V-B and V-C, respectively, we bound $Y(\tau, B)$ by $O(B)$ for BP and block-resilient HBP computations.

A. *Preliminaries*

We will assume the following property holds for space allocation by the runtime system.

Property V.1. (Space Allocation Property.) *Whenever a processor requests space it is allocated in block sized units; naturally, the allocations to different processors are disjoint and entail no block sharing.*

In accordance with Property V.1 we will assume that each new instantiation of a block on an execution stack represents a new block. This is consistent with our definition of block delay and block wait cost (Definition II.3) since two different accesses to the same block in different instantiations on an execution stack will not contribute to each other's block wait costs. This assumption will play an important role in the proof of Lemma V.7.

To simplify the discussion, we assume that each stolen subtask begins at the right child of the fork node from which it was stolen. Incidentally, this rule enables constant factor reduction in the worst-case bound on cache misses compared to having the task at either child being stolen.

The next observation follows from the fact that the kernel τ^K of a task τ corresponds to the portion of τ that remains after its stolen subtasks are removed, together with the fact that τ^K is executed in a modified depth first search order in which a (join) vertex is explored only when all of its in-edges have been traversed.

Observation V.2. *Let D be the series-parallel computation dag for a task τ, and let τ^K be its kernel. Consider the start nodes for all of τ^K's stolen subtasks, and let v be the deepest such node in D. Let P_τ be the path in D from the root of D to the parent of v. Then, the set of tasks stolen from τ^K consists of the tasks corresponding to those nodes of D that are the right child of a node on P_τ but are not themselves on P_τ.*

We will refer to the path P_τ in the above definition as the *steal path* of τ. Note that if a task stolen from τ accesses τ's execution stack, then the segment it accesses on τ must correspond to that for a node on P_τ.

Let β be a block on the execution stack of a task τ. The following lemma establishes that the block delay incurred by β can be bounded as a function of the number of accesses to β by stolen subtasks of τ, independent of the number of accesses to β by the processor executing the kernel of τ.

Lemma V.3. *Let β be a block on the execution stack S_τ of a task τ, and let T' be any sub-interval of time during which τ is executed. Suppose that processors $C_1, \cdots, C_k$ are the only processors executing stolen subtasks of τ during T'. Further suppose that they access block β a total of x times during T'. Then β incurs a block delay of at most $2x + u$ during T', where u is the number of β-accessing usurpations of τ that occur during T'.*

Proof: By Definition II.3 we need to bound the number of times β is moved between caches. Processors $C_1, \cdots, C_k$ cause at most x moves of block β to their caches as a result of their x accesses. Thus processor C needs at most x moves of block β to its cache to handle all its accesses to β, regardless of their number. If the execution of τ shifts from C to C' due to a usurpation, then C will not access S_τ any further, since it has completed its execution of τ^K. Thus the usurpation, if it accesses β, causes just one move of the block to C', and hence there are only u additional moves of β due to usurpations. This proves the lemma. ■

B. *Block Delay in a BP Computation*

In this section we bound the block delay $Y(\tau, B)$ of any block on τ's execution stack by $O(\min\{B, \log|\tau|\})$ when τ is a task in a BP computation.

This bound is shown by demonstrating that the only segments read by stolen subtasks of τ are those for nodes on P_τ, that these segments are all present on S_τ at one moment in common and hence occupy disjoint locations, and as their variables are limited access, a bound of $O(B)$ on $Y(\tau, B)$ ensues. The bound of $O(\log|\tau|)$ follows because there are $O(\log|\tau|)$ such segments and they each have size $O(1)$.

Lemma V.4. *Let $\mathcal{A}$ be a limited-access BP algorithm and let τ be a parallel task in the execution of $\mathcal{A}$. Let β be a block used for τ's execution stack S_τ. Then β incurs a block delay of $O(\min\{B, \log(|\tau|)\})$ during τ's execution.*

Proof: By Observation V.2, there is a single path P_τ, starting at the root node of τ, such that stolen subtasks of τ correspond to off-path right children of P_τ. Each node v of P_τ stores $O(1)$ local variables contiguously on S_τ in its segment; we will denote the segment for v by σ_v. As noted earlier, the only segments that can be accessed by the stolen subtasks of P_τ are the segments for nodes on P_τ. In addition, these segments for nodes on P_τ occupy disjoint portions of S_τ, so there is no re-use of β for different variables. As each of the variables stored on S_τ is a limited access variable, it follows that β can be accessed $O(\min\{B, \log(|\tau|)\})$ times by the stolen subtasks, since $|P_\tau| \leq \log(|\tau|)$.

Finally, usurpations can only occur on the up-pass of the computation, when the segments on S_τ are all segments for nodes on P_τ, and thus only these segments can be accessed by a usurped portion of the computation. But $O(\min\{B, \log(|\tau|)\})$ bounds the total length of the portion of segments of P_τ on β, and hence the number of accesses to β, and hence the number of usurpations accessing β.

The result now follows from Lemma V.3. ■

Note that a BP computation of size r may produce an output of length r. This output is not part of the local variables of the BP computation, and hence is not considered in Lemma V.4; by limited access, each block in the output is accessed $O(B)$ times. This output will need to be considered if the BP computation is called from within an HBP computation; this will be addressed in the next section.

C. *Block Delay in an HBP Computation*

We now bound $Y(\tau, B)$ for a task in a block-resilient HBP algorithm $\mathcal{A}$ with computation dag D. For this, we first define top-dominance and then block-resilience. Although these definitions are quite technical, we describe in the next section block-resilient HBP algorithms for several fundamental algorithms, many of which are known multicore algorithms, or small variants of such algorithms.

Before entering into the first definition, we make one observation regarding the size of a non-recursive procedure $\mathcal{B}$ called from a procedure $\mathcal{A}$. The size of $\mathcal{B}$ could be larger than the size of $\mathcal{A}$. This could occur, for example, if $\mathcal{A}$ has a large collection of local variables, and many or all of these variables form the input to B. Accordingly, we use n to denote the size of an invocation of $\mathcal{A}$ and m to denote the size of the corresponding invocation of $\mathcal{B}$.

Definition V.5. *Let $\mathcal{A}$ be a Type $t \geq 2$ HBP algorithm with computation dag D. $\mathcal{A}$ is $S(n)$*-top dominant, top dominant *for short, if $S(n)$ is non-decreasing, and if, for each size n invocation of $\mathcal{A}$,*
(i) The size, $X(n)$, of $\mathcal{A}$'s segment on the execution stack satisfies $X(n) = O(S(n))$.
(ii) The total length of the segments for the nodes along any computation path P in D is $O(X(n))$.
(iii) Every Type $t' \geq 2$ procedure $\mathcal{B}$ called by $\mathcal{A}$ is $S_B((m)$-top-dominant, and further if the call to $\mathcal{B}$ results in an invocation of size m, then $S_B(m) = O(X(n))$.

We will refer to $S(n)$ as the *space bound* of the top dominant algorithm $\mathcal{A}$ since the space used on the execution stacks in the computation of $\mathcal{A}$ is bounded by $S(n)$. Note that by this definition, a computation cannot be top dominant if it uses only constant space for its local variables.

Later in this section we will show that algorithms in a fairly natural class of 'fast shrinking' algorithms that generate parallel recursive tasks of geometrically decreasing sizes are all top dominant. In all of our algorithms, $X(n) = \Theta(S(n))$ for all inputs of length n.

We are now ready to define block-resilience.

Definition V.6. *An HBP algorithm is* block-resilient *if it is limited access, top dominant, and its accesses also observe the well-buffered rule and the local constraint.*

We now bound $Y(D, B)$ for block-resilient HBP tasks. As for BP computations, the main idea is to note that the only segments accessed by stolen tasks are those for nodes on P_τ. Here the difficulty is that some of these segments could occupy the same locations on β, albeit at different times. Despite this, we show that the total length of the portions of the segments overlapping β for nodes on P_τ is $Y(\tau, B) = O(\min\{S(|\tau|), B)\})$. Together with the limited access property this implies there are only $O(\min\{S(|\tau|), B)\})$ accesses to these segments, which includes all accesses by stolen subtasks. We also need to bound the number of usurpations, since in an HBP computation, usurped tasks may access segments stored in β which are for nodes not on P_τ. Here too we show that there are only $O(\min\{S(|\tau|), B)\})$ such accesses.

Lemma V.7. *Let $\mathcal{A}$ be a block-resilient HBP algorithm with space bound $S(n)$. Let τ be a parallel task in the execution of $\mathcal{A}$, and let β be a block used for τ's execution stack S_τ. Then the block delay incurred by β during the execution of τ is bounded by $Y(\tau, B) = O(\min\{S(|\tau|), B)\})$.*

Proof: We partition the accesses to β into two categories: 1. those accessing segments for nodes on P_τ, the P_τ-segments, and 2. those accessing segments for nodes not on P_τ. The P_τ-segments could be accessed by stolen tasks, by the computation of usurped portions of the task kernel, and by the computation of the initial non-usurped portion. We show that the number of accesses to the P_τ-segments is bounded by $O(\min\{S(|\tau|), B)\})$. Then, by Lemma V.3, we can conclude that the Category 1 accesses contribute $O(\min\{S(|\tau|), B)\})$ to the block delay. The segments in Category 2 are accessed only by the processors executing τ^K, since stolen subtasks can only access segments for nodes on P_τ. Thus by Lemma V.3, the number of block transfers induced by the Category 2 accesses is bounded by the number of distinct usurpations which execute one or more of these accesses, and it suffices to bound this number.

Category 1: Accesses to segments on P_τ.
Let σ' be the segment on the bottom of β (i.e. all other segments present at the same time as σ' were added to the execution stack S_τ later than σ'), and let τ' be the corresponding task. The block β will be deallocated when τ' completes its computation, and we need to bound the number of accesses to β until this event occurs.

In the cases below, we will either directly show a bound of $O(\min\{S(|\tau|), B)\})$ on the number of accesses, or we will show that the sum of the lengths of the segment portions overlapping β for nodes on P_τ is bounded by $O(\min\{S(|\tau|), B)\})$. We call this the *length bound*. By the limited access property, this implies the same bound on the number of Category 1 accesses. (In most of the cases below, we show a bound of $O(\min\{S(|\tau'|), B)\}) = O(\min\{S(|\tau|), B)\})$.)

Case 1. σ' is the segment for a node in a BP computation. Then β is deallocated once this BP computation completes, and hence the bound from Lemma V.4 applies, a bound of $O(\min\{\log(\tau''), B\}$, where τ'' is the BP task at hand; by the top dominance of τ, this is bounded by $O(\min\{S(|\tau|), B)\})$.

Case 2. σ' or a portion of σ' fills β.
During τ''s lifetime all the accesses to β are to σ' and hence by limited access there are only $O(B) = O(\min\{S(\tau'), B\})$ such accesses.

Case 3. τ' is a task of type $t' > 1$ and $X(|\tau'|) \leq B$.
Then the other segments that appear in β are all part of τ''s computation and by top dominance the length of the subset of the P_τ-segments is $O(X(\tau')) = O(\min\{S(\tau'), B\})$.

Case 4. τ' is a task of type $t' > 1$ and $X(|\tau'|) \geq B$.
For each of its up to c collections of recursive calls, τ' will generate on P_τ a sequence of $\log(v(|\tau'|)$ size $O(1)$ segments for nodes along a path in the BP-like forking tree that generates the $v(n)$ recursive tasks. As in Lemma V.4, for each collection the length bound is $O(\min\{\text{ht}(\tau'), B\})$; but $\text{ht}(\tau') = O(\log v(|\tau'|) = O(S(|\tau'|)$, the last bound following by top dominance; thus the length bound per collection is $O(\min\{S(\tau'), B\})$. Because $c = O(1)$, the overall length bound is also $O(\min\{S(\tau'), B\})$.

To complete the Case 4 bound, we need to consider the space in β used by the segments σ'' for the procedures, recursive or non-recursive, called by τ'. Let τ'' be one such task. If $S(|\tau''|) \leq B$ then, as in Case 3 the space in β used by τ'''s segment plus the segments its computation generates that have nodes on P_τ is bounded by $O(\min\{S(\tau), B\})$. Otherwise, the segments generated by τ'''s computation all lie outside β as $\sigma_{\tau''}$ fills the remaining space in β and thus τ'' uses space $O(\min\{S(|\tau''|), B\}) = O(\min\{S(\tau), B\})$ in β. There are at most $2c+1$ such recursive and non-recursive procedure calls, and as $c = O(1)$, this is a total of $O(\min\{S(\tau), B\})$ space used in β.

Case 5. σ' is the segment for a node in a tree forking recursive calls.
A length bound of $O(\min\{S(\tau), B\})$ is obtained by an analysis largely identical to that for Case 4.

It follows that there are at most $O(\min\{S(\tau), B\})$ Category 1 accesses, and hence at most $O(\min\{S(\tau), B\})$ transfers of β due to these accesses.

Category 2: Accesses to segments for nodes not on P_τ.
As already noted, the Category 2 accesses we need to bound are all due to usurpations, and occur in the following situation. Following a usurpation by a processor C', C' continues the execution of τ^K. This includes the execution of as yet unexecuted nodes that are descendants of nodes on P_τ and to the left of this path (recall that off-path right children of nodes on P_τ are the nodes where stolen subtasks begin). If C''s first access to β is to a segment for such a node then this causes an additional transfer of β. So we need to bound the number of usurpations causing this type of access. Recall that, by Lemma V.3, each usurpation contributes at most $O(1)$ to the block delay of β.

We obtain the desired bound by associating each usurping processor with the segment for a distinct node on P_τ as follows. Let v be the node at which the stolen subtask causing the usurpation started. We will associate this usurpation with the segment for the parent of node v; note that this associated segment is for a node on P_τ. Some of these associated segments may overlap β (Category 1 segments) while others may be more recently added and may lie above β on S_τ (Category 2 segments). The number in Category 1 is readily bounded: There are $O(\min\{S(|\tau|), B)\})$ such segments overlapping β, since, by the argument for Category 1 accesses, between them, these segments incur $O(\min\{S(|\tau|), B)\})$ accesses to β. So the usurping processors associated with Category 1 segments cause $O(\min\{S(|\tau|), B)\})$ additional accesses.

The only usurping processors C remaining to be considered are those whose associated segments do not overlap β. Such segments are all higher up on S_τ than β (i.e. more recently added). Let C be such a usurping processor, let σ be the associated segment, and let σ' be the topmost segment on S_τ overlapping β during σ's lifetime. As σ is a segment for a node on P_τ, so is σ'. Since we are now considering accesses to segments on β for nodes not on P_τ, the access by C which causes the additional transfer can occur only after σ' ceases to be on S_τ. Consequently, there is a distinct segment σ' overlapping β and on P_τ for each such usurping processor, and thus there are at most $O(\min\{S(|\tau|), B\})$ such usurping processors, as there are $O(\min\{S(|\tau|), B\})$ segments σ' for nodes on P_τ overlapping β.

This shows that there are at most $O(\min\{S(|\tau|), B\})$ transfers of β due to usurpations of τ^K's computation. ■

Next, we define the fairly natural class of 'fast shrinking' HBP algorithms, and we show that algorithms in this class are top dominant.

Definition V.8. *Let $\mathcal{A}$ be a Type t HBP algorithm whose work is polynomially bounded. Then, $\mathcal{A}$ is* fast shrinking *if there is a non-decreasing function $S(n) = \Omega(\log n)$ satisfying the following three conditions:*
(i) A size n invocation of $\mathcal{A}$ has an initial segment of size $\Theta(S(n))$.
(ii) There is a constant $\nu < 1$ such that $c \cdot S(r(n)) \leq \nu S(n)$.
(iii) Any Type $2 \leq t' < t$ procedure $\mathcal{B}$ called by $\mathcal{A}$ is fast shrinking, and if their invocations have sizes m and n respectively, then the space bound $S_B(m)$ for $\mathcal{B}$ satisfies $S_B(m) = O(S(n))$.

All of the HBP algorithms we present are fast shrinking. The polynomially bounded work is required to ensure $\log v(n) = O(\log n)$ for all HBP subcomputations; we can remove this restriction by requiring $S(n) = \Omega(\log(n \cdot v(n)))$ (instead of $\Omega(\log n)$) in each HBP procedure in the algorithm.

Lemma V.9. *If Type t $\mathcal{A}$ is fast shrinking then it is top dominant.*

Proof: We use induction on t.

Base case: $t = 2$.
Recall that $\mathcal{A}$ makes $O(1)$ calls to BP procedures, and that it makes $c = O(1)$ collections of recursive calls to subproblems of sizes bounded by $r(n)$, and that $v(n)$ upper bounds the number of recursive calls in each collection of recursive calls (see Definition III.2). We note that as $\mathcal{A}$'s work is polynomially bounded, $v(n)$ must be polynomially bounded too, and so $\log v(n) = O(\log n)$.

Next, we observe that it suffices to prove property (ii) in Definition V.5 (for properties (i) and (iii) follow immediately from the corresponding properties in Definition V.8). To this end, let P be a path in D. We identify the sequence of segments that are created for the nodes on P. First, for each BP task τ called by $\mathcal{A}$, each of the $O(\log n)$ nodes of τ that are on P will have a segment of length $O(1)$. Second, for each collection of recursive calls made by $\mathcal{A}$, there will be a forking tree to instantiate the recursive calls. Each such tree will have $\log v(n) = O(\log n)$ nodes on P, and each node will have a segment of length $O(1)$. Third, in each collection of recursive calls made by $\mathcal{A}$, there will be one recursive task whose computation nodes intersect with P. Each such task τ has a segment of length at most $S(r(n)) = O(S(n))$. The total length of the segments in the above three categories is $O(\log n + S(n)) = O(S(n))$.

In addition, each of the c successive recursive calls whose nodes overlap with P will generate segments for the BP tasks it calls and for any further recursive calls it may make. So the total length of the segments for all the nodes on P is bounded by $O(S(n) + cS(r(n)) + \cdots + c^i S(r^{(i)}(n)) + \cdots) = O(S(n) + \nu S(n) + \cdots + \nu^i S(n) + \cdots) = O(S(n))$.

Inductive step: $t > 2$.
The one change to the argument in the base case is that we need to account for the length of the segments of the non-recursive procedures called by $\mathcal{A}$. But, by induction, as they are fast shrinking each such procedure B will generate segments of total length $O(S_B(n)) = O(S(n))$ for the nodes along any path P. Hence the total length of the segments for non-recursive procedures is $O(S(n))$ as in the base case, and the overall bound follows as before. ■

In summary, in this section we have established that in any block-resilient HBP, the block delay incurred by a single shared block in a task τ is bounded by $Y(\tau, B) = O(\min\{S(\tau), B\})$ for an HBP computation, and $Y(\tau, B) = O(\min\{\log|\tau|, B\}$ for a BP computation. Since both $S(\tau)$ and $\log \tau$ depend only on the size of τ, we will henceforth refer to Y as $Y(r, B)$, where r is the size of the task τ.

VI. HBP Algorithms

Table 1 (in Section I) lists the block-resilient HBP and hybrid HBP algorithms that we present and analyze in this paper. We now describe these algorithms, starting with known algorithms that are inherently block-resilient HBP.

Scans and **MA** (Matrix Addition) [4] can be implemented as a single BP computation. *Prefix sums* (**PS**) can be implemented as a sequence of two BP computations, where the first BP computation computes sums of disjoint subarrays of size 2^i, for $i \leq \log n$ and the second BP computation computes the final output. These are type 1 HBP computations with $f(r) = O(1)$, $L(r) = O(1)$, and $Y(r, B) = O(\min\{\log r, B\})$ by Lemma V.4.

Matrix Computations. For matrix computations, we assume that the matrix is in the *bit interleaved (BI) layout*, which 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 are $O(1)$-friendly, and have $O(1)$-block sharing, which allows us to obtain good cache and fs miss bounds. We describe several methods to convert between the standard *row major (RM)* layout and BI; these methods can be used in conjunction with our algorithms for BI if the input and output matrices are to be in RM.

MT is matrix transposition when the $n \times n$ matrix is given in the BI layout. When we expose the parallelism in the recursive algorithm in [18] we obtain a BP computation with $f(r) = O(1)$, $L(r) = O(1)$, and $Y(r, B) = O(\min\{\log r, B\})$.

MM. This algorithm multiplies two $n \times n$ matrices by recursively multiplying eight $n/2 \times n/2$ matrices, and performs the matrix additions for the divide and combine steps using MA. This results in a Type 2 HBP computation with $c = 1$ collection of $v = 8$ subproblems of size $s(m) = m/4$, where $m = n^2$ is the size of the matrix. This algorithm computes the 8 recursive submatrices in new subarrays. These matrices are then combined with pairwise matrix additions, performed using MA, and the final four submatrices are written back to the four quadrants in the parent matrix. Thus each variable in this algorithm is written only a constant number of times, and the algorithm is inherently limited access. When the matrices are in the BI layout, this computation has $f(r) = O(1)$ and $L(r) = O(1)$, The space used for local variables when multiplying $k \times k$ submatrices is $S(k^2) = k^2$. Since this is a fast shrinking computation it is top dominant by Lemma V.9, hence $Y(r, B) = O(\min\{r, B\})$ by Lemma V.7. The sequential cache complexity is $\Theta(\frac{n^3}{B\sqrt{M}})$, and $T_\infty = O(\log^2 n)$.

Strassen. This algorithm has the same HBP structure and parameters as MM, except that $v = 7$ instead of 8. Hence this is is a top dominant computation with $L(r) = O(1)$, $Y(r, B) = O(\min\{r, B\})$, and sequential cache complexity $\Theta(\frac{n^\lambda}{BM^\gamma})$, where $\lambda = \log_2 7$ and $\gamma = (\lambda/2) - 1$ and T_∞ remains $O(\log^2 n)$.

Since we have assumed in the above algorithms that matrices are in the BI layout, we need methods to convert between the traditional RM (row major) layout and the BI layout. It turns out that RM to BI is easy to execute with $O(1)$ block-sharing, while BI to RM requires more effort.

RM to BI. We use a simple BP computation that recursively converts each quadrant in parallel, with all writes in BI order. The writes are thereby arranged so that tasks share $L(r) = O(1)$ blocks for writing. Reading, however, is only $f(r) = \sqrt{r}$-friendly. (This is an example of an algorithm where $f(r)$ is larger than $L(r)$.) This is a type 1 HBP computation with $Y(r, B) = O(\min\{\log r, B\})$.

Direct BI to RM. This uses the same recursion as RM to BI. Thus $Y(r, B)$ remains $O(\min\{B, \log r\})$. However, since the writes are to an output matrix in RM, both $L(r)$ and $f(r)$ are $\sqrt{r}$, and so this computation is $\sqrt{r}$-block sharing. We describe improved methods for BI to RM later.

FFT. We expose the parallelism in the cache-oblivious FFT algorithm in [18]. As noted in [11], this is also a low-depth multicore algorithm. The 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 $\Theta(\sqrt{n})$, and finally performs MT on the result. The sequential time is $O(n \log n)$, the sequential cache complexity is $O(\frac{n}{B} \cdot \log_M n)$ [18], and the parallel depth is readily seen to be $O(\log n \cdot \log \log n)$.

We keep the matrices in the BI representation. Thus, the 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 $s(n) = \sqrt{n}$ with a constant number of MT computations performed before and after each recursive call. We have $f(r) = O(1)$ and $L(r) = O(1)$, outside of the cost to convert between BI and RM formats. At the end, to convert to the RM format we use either the Direct BI to RM described above, or BI-RM for FFT (described below), which will give an improved bound for block delay. This is a fast shrinking Type 2 HBP with $S(r) = \Theta(r)$, hence $Y(r, B) = O(\min\{B, r\})$.

We now turn to HBP algorithms which need some modifications in order to achieve block-resilience (primarily the limited access property).

Depth-n-MM. The standard version this algorithm is CO-MM [4], [20], which is obtained by exposing the parallelism in the cache-oblivious matrix multiplication algorithm [18]. It has $c = 2$ sequenced collections of recursive calls each to $v = 4$ parallel subproblems of size $s(n^2) = n^2/4$. It performs $O(n^3)$ work, has $O(n^3/(B\sqrt{M}))$ sequential cache complexity, and $O(n)$ critical path length.

CO-MM is not a limited access algorithm, since each entry in the matrix is written n times. Hence, we convert it into a limited access algorithm as follows. Each MM subproblem creates 4 subarrays to store the results of the 4 recursive calls that it makes. Each array entry is then written and read only twice (once for each of the two recursive calls made in sequence on that subarray). After the computation of a submatrix is completed, we use a BP computation to add it back to the appropriate quadrant in the parent matrix. We note that this causes the algorithm to use $O(n^2 \cdot \log p)$ extra space, but this is still a better bound than the excess space of $\Theta(n^2 \cdot p^{1/3})$ incurred by MM.

This algorithm is a type 2 HBP with $f(r) = O(1)$ and $L(r) = O(1)$ when the matrices are in the BI layout. Since this is a fast shrinking computation it is top dominant, hence $Y(r, B) = O(\min\{r, B\})$.

BI-RM (gap RM). We have the same algorithm as as Direct BI to RM, but to mitigate the fs miss cost, we use a *gapping* technique. The destination array representing the RM matrix will be given gaps as follows: between $r \times r$ subarrays (for values of r corresponding to recursive subproblems) the rows will be given a length $r/\log^2 r$ gap. Now, tasks of size r^2

for $r = \Omega(B \log^2 B)$ share zero blocks for their writing. This gives a cost of $O(Br)$ for the fs misses for a size r^2 task, for $r = O(B \log^2 B)$. So $L(r^2) = O(r)$, but only for $r \leq B \log^2 B$.

The justification for this choice of size is that it increases the size of the array by only a constant multiplicative factor (since $\sum_{r=2^i} \frac{1}{\log^2 r} = O(1)$). Indeed a gap of $r/[\log r (\log\log r)^2]$, or any analogous sequence of iterates, also works, reducing the fs miss cost correspondingly.

Having written to an array with gaps one needs to compress the array using a standard scan. This is a BP computation which has $f(r) = O(1)$ and $L(r) = O(1)$. This is a sequence of two BP computations so we have $Y(r, B) = O(\min\{\log r, B\})$.

We now turn to BI-RM for FFT, which is a new algorithm for converting from BI to RM. Although the number of operations is now $\omega(n^2)$, this algorithm achieves a lower block delay cost than BI-RM (gap RM), and is a better choice for FFT, and for all three MM algorithms.

BI-RM for FFT. This is an $O(n^2 \log\log n)$ operation algorithm with $T_\infty = O(\log n)$. The algorithm divides the input BI array of length n^2 into n subproblems, each of which it recursively converts to the RM order. Then, using a BP computation, it copies the n subarrays into one subarray, accessing data according to the RM order in the target output. This is a type 2 HBP computation that calls $c = 1$ collection of $v(n^2) = n$ subproblems of size $s(n^2) = n$. The BP computation for the copying is organized so that the writes are in RM order, and hence $L(r) = O(1)$.

This is a fast shrinking Type 2 HBP with $S(r) = \Theta(r)$, hence $Y(r, B) = O(\min\{B, r\})$. It can be shown that $f(r) = \sqrt{r}$; we omit the details here.

Finally, we discuss SPMS sort [12].

SPMS Sort. ([12]) The SPMS sort has the same structure as FFT, though its forking trees have the more relaxed property that the subproblem sizes decrease by at least constant every $O(1)$ levels rather than every level. As shown in [12], it has $f(r) = \sqrt{r}$ and $L(r) = O(1)$. Its space usage for local variables is $S(r) = \Theta(r)$, and it is fast shrinking, similar to FFT. Hence $Y(r, B) = O(\min\{B, r\})$.

We use SPMS Sort in the following two hybrid HBP algorithms, LR and CC.

List Ranking (LR). Efficient multicore algorithms for LR based on eliminating large independent sets are given in [3], [11], [5]. As in [5] we adapt the PRAM algorithm whose first stage performs $O(\log\log n)$ stages of eliminating a constant fraction of the elements in the linked list. To find a large independent set, we use the method MO-IS in [11] that constructs an $O(\log^{(k)} r)$-size coloring of the linked list, and then extracts an independent set of size at least $r/3$ (r is the current length of the linked list) 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 on inputs whose combined length is r, and as shown in [11], 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 $O(\log^2 n \log\log n)$ parallel time. As the LR algorithm is essentially a sequencing of iterations of SPMS sort, $Y(r, B) = O(\min\{B, r\})$.

In [13] we introduce gapping into this LR algorithm to further reduce the cost of fs misses under the RWS scheduler.

CC. We use the connected components algorithm in [11]. The dominant cost is $\log n$ stages of list ranking, so each of the work, parallel time, cache complexity, and fs miss cost increases by a factor of $\log n$.

Other multithreaded algorithms, such as I-GEP [9] and LCS [10] can be analyzed similarly. We omit the details here.

VII. Scheduling Bounds

In this section we bound the block delay of the algorithms we have presented when scheduled by S_C, the simple centralized multicore-oblivious scheduler used in [9]. For this analysis we will bound $Y(r, B)$ by $O(B)$, ignoring the dependence on r. We will also explain below, by means of an example, why in the absence of the HBP properties, the false sharing costs could be considerably larger than those achieve with block-resilient HBP algorithms. The example illustrating this will be the depth n MM algorithm.

We have established in Section V that each block in a linear space block-resilient algorithm incurs at most $O(B)$ cache miss cost due to fs misses. Thus, if the algorithm is $O(1)$ block-sharing and we have an upper bound of S on the number of steals in the computation when scheduled by a given scheduler, the total number of parallel tasks, including the initial task, is $S + 1$, and hence the total overhead due to fs misses is bounded by $O(B \cdot (S + 1))$ cache misses. If we employ gapping, we may get better bounds by also considering the sizes of the parallel tasks.

A Centralized Scheduler S_C. Let us consider the simple centralized scheduling algorithm in [9], which we will denote by S_C. We assume that S_C does not know the block size B, so it schedules obliviously with respect to block size B. Given p processors, the scheduler S_C expands the binary forking in BP and HBP computations, in a BFS manner until there are at least p tasks, all of approximately the same depth, and it schedules these tasks on the p processes. Given the balanced nature of HBP computations, typically the entire computation is decomposed into a sequence of groups of p roughly equal-sized parallel tasks by this scheduler. Thus we can readily bound the number of parallel tasks.

In the following, we use $\Delta(n, p, B)$ to denote the block delay cost in the computation when scheduled by S_C (recall this is in units of cache misses). We let $Q(n, M, B)$ be the sequential cache complexity of the algorithm. If $\Delta(n, p, B) = O(Q(n, M, B))$, then the overhead of the cost due to fs misses is absorbed by the sequential cache complexity of the algorithm, and hence will not dominate the running time of the computation. (For most of the algorithms we consider it is well-known that the parallel cache complexity with private caches is at least that of the sequential cache complexity on a single cache of the same size.) In many cases, we achieve $\Delta(n, p, B) = O(Q(n, M, B))$ for reasonable sized inputs.

Scans. The scheduler S_C will unfold the BP tree for $\log p$ levels and will schedule the p tasks, each of size n/p, on the p processors. This results in p parallel tasks, hence the block delay cost is $\Delta(n, p, B) = O(p \cdot B)$ for Scans and Prefix Sums. Since $Q(n, M, B) = O(n/B)$, we will have $\Delta(n, p, B) = O(Q(n, M, B))$ when $n = \Omega(p \cdot B^2)$. For MA, MT and RM to BI, the input size is n^2, and the same result holds with n replaced by n^2.

Depth-n-MM. Here we have a sequence of $\sqrt{p}$ parallel executions of matrix multiplications on $(n/\sqrt{p}) \times (n/\sqrt{p})$ matrices. Since we assume the BI format, we have both $f(r)$ and $L(r)$ being $O(1)$. The number of parallel tasks is $p^{3/2}$ hence the block delay cost is $\Delta(n, p, B) = O(B \cdot p^{3/2})$, and $\Delta(n, p, B) = O(Q(n, M, B))$ when $n^2 = \Omega(p \cdot B^{4/3} M^{1/3})$.

By contrast, consider the original Depth n MM algorithm in [4], [20]. This algorithm fails to be block-resilient because it is not limited-access; in fact, each location in the output matrix is accessed n times in this algorithm. Consider a single parallel computation on the p cores in the sequence of $\sqrt{p}$ parallel computations scheduled by S_C. Each core updates a disjoint output matrix of size n^2/p, and due to the BI format, it shares just two blocks with other cores. But a core 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 core that shares this block. Across the p cores, 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 bound of $n \cdot B \cdot p$ block transfers due to false sharing. By contrast, since the maximum number of processors that can be used work-efficiently is $O(n^2)$, the block-resilient depth-n-MM bound of $O(B \cdot p^{3/2})$ is only $O(\sqrt{n} \cdot B \cdot p)$ even at the highest level of parallelism, and is smaller for lower levels of parallelism.

MM and Strassen. In both of these algorithms, S_C unfolds $O(\log p)$ levels of recursion in order to generate p recursive tasks of maximum size ($\log p^{1/3}$ levels for MM and $\log p^{1/\log_2 7}$ levels for Strassen). After computing on these p parallel tasks, $O(\log p)$ levels of matrix additions remain to complete the computation, each of which is performed with p parallel tasks. Hence the number of parallel tasks is $O(p \log p)$ for both algorithms, and the block delay cost under the centralized scheduler is $\Delta(n, p, B) = O(B \cdot p \log p)$, and $\Delta(n, p, B) = O(Q(n, M, B))$ if $n^2 = \Omega(B^{4/\lambda} M^{1/\lambda} \cdot (p \log p)^{2/\lambda})$, where $\lambda = \log_2 7$ for Strassen, and 3 for MM.

BI to RM for FFT. We have one recursive call to $\sqrt{n}$ subproblems. The recursion needs to be unraveled $i = O(\log \frac{\log n}{\log(n^2/p)})$ time in order to generate p parallel tasks that can be scheduled on the p processors. Thus, the number of parallel tasks is $O(p \cdot \log \frac{\log n}{\log(n^2/p)})$, and the block delay cost is $O(B \cdot p \log \frac{\log n}{\log(n^2/p)})$; We claim $\Delta(n, p, B) = O(Q(n, M, B))$ if $n^2 = \Omega(p \cdot (B^2 + M))$. To see this, observe that it suffices to have $B^2 \cdot p \cdot \log M \cdot \log \frac{\log n^2}{\log(n^2/p)} = O(n^2 \log n^2)$, and for this $B^2 \cdot p \cdot \log M \frac{\log n^2}{\log(n^2/p)} = O(n^2 \log n^2)$, or equivalently $B^2 \cdot p \cdot \log M = O(n^2 \cdot \log(n^2/p))$ suffices. If $Mp = O(n^2)$, then $\log M = O(\log(n^2/p))$, and $B^2 \cdot p = O(n^2)$ suffices, yielding that the condition $p(B^2 + M) = O(n^2)$ suffices.

FFT and SPMS Sort. In FFT we have $c = 2$ recursive calls to $\sqrt{n}$ subproblems interleaved with the BP computation MT. As in BI to RM for FFT, the recursion needs to be unraveled $i = O(\log \frac{\log n}{\log(n/p)})$ time in order to generate p parallel tasks that can be scheduled on the p processors. Since $c = 2$, there are $2^i = O(\log n / \log(n/p))$ calls to MT, each of which is scheduled using p parallel tasks. Hence the computation is scheduled with $O(p \cdot (\log n)/\log(n/p))$ parallel tasks, leading to a block delay cost of $O(B \cdot (p \log n)/\log(n/p))$ cache misses. Similar to BI-RM for FFT, $\Delta(n, p, B) = O(Q(n, M, B))$ if $n = \Omega(p \cdot (B^2 + M))$.

We now turn to the algorithms that use gapping.

BI to RM (gap RM). We have a sequence of two BP computations where the second is simply a scan with $L(r) = O(1)$, thus its block delay cost is $O(pB)$. For the first BP computation we have $L(r) = \sqrt{r}$ if $r = O(B^2 \log^4 B)$ and is zero otherwise. Thus, if $n^2/p > B^2 \log^4 B$ there is no fs miss cost here; for smaller values of n^2/p, the block delay cost is $O(\sqrt{n^2/p} \cdot Bp) = O(Bn\sqrt{p})$, and hence $\Delta(n, p, B) = O(Q(n, M, B))$ if $n^2 = \Omega(pB^2 \log^4 B)$. (In contrast, if we stay with Direct BI to RM, we have $L(r) = \sqrt{r}$, so $\Delta(n, p, B) = \Theta(Bn\sqrt{p})$ for all n, and then $\Delta(n, p, B) = O(Q(n, M, B))$ only if $n^2 = \Omega(pB^4)$).

List Ranking and CC. The total number of parallel tasks under S_C is dominated by the $O(\log n)$ sorting steps in the pointer jumping phase, leading to block delay cost of $O(B \cdot \frac{p \cdot \log^2 n}{\log(n/p)})$ cache misses. For CC, this cost is multiplied by a factor of $\log n$.

Other Schedulers. We have bounded the fs miss cost cost incurred by our block-resilient HBP algorithms when scheduled by the natural centralized scheduler S_C. In principle we can apply the bounds we have obtained for $L(r)$ and $Y(r, B)$ in Section VI (together with the sizes of the tasks scheduled on the processors as needed), to determine the block delay overhead for these algorithms under any scheduler.

In recent research [13] that builds on the results in this paper, we analyze the performance of *randomized work stealing* [7], [1] when fs misses are considered, and we present improved bounds for series-parallel dags, and further improved bounds for block-resilient HBP computations.

VIII. Discussion

In this paper we have presented a collection of algorithmic techniques to mitigate the overhead of false sharing. In particular, we have established that any block-resilient HBP algorithm with block-sharing function $L(r) = O(1)$ incurs a cost of no more than $O(B)$ cache misses due to false sharing, for each parallel task generated in the computation. We have shown that many of the known highly parallel multithreaded algorithms can be adapted to satisfy these requirements. We also used an alternate gapping technique to reduce the false sharing overhead in an algorithm that does not achieve $O(1)$ block sharing. Our results are general and apply to any scheduling algorithm, and we illustrated our results using the simple centralized scheduler used in [9].

Our results are for multicores with private caches. However, they also hold at higher levels of a cache hierarchy, where the bound is in terms of the number of parallel tasks scheduled at caches at that level. Centralized schedulers, such as those in [4], [11], [6] reduce the number of parallel tasks at higher level caches in an attempt to minimize cache misses. By the results we have presented in this paper, this strategy also reduces the false sharing costs, provided the algorithms are block-resilient, with $O(1)$-block sharing or suitable gapping.

References

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

[2] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proc. ACM SPAA*, pages 197–206, 2008.

[3] L. Arge, M. T. Goodrich, and N. Sitchinava. Parallel external-memory graph algorithms. Proc. IPDPS, 2010.

[4] G. Blelloch, R. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proc. SODA*, pages 501–510, 2008.

[5] G. Blelloch, P. Gibbons, and H. Simhadri. Low depth cache-oblivious algorithms. In *Proc. ACM SPAA*, pages 189–199, 2010.

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

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

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuzmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. ACM PPoPP*, pages 207–216, 1995.

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

[10] R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proc. ACM SPAA*, pages 207–216, 2008.

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

[12] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. In *Proc. ICALP, Track A*, 2010.

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

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

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

[16] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, S. E., R. Subramonian, and T. von Eicken. Logp: Toward a realistic model of parallel computation. In *Proc. ACM PPoPP*, pages 1–12, 1993.

[17] R. Dorrigiv, A. Lopez-Ortiz, and A. Salinger. Brief announcement: Optimal speedup on a low-degree multi-core parallel architecute (LoPRAM). In *ACM SPAA*, pages 185–187, 2008.

[18] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. FOCS*, pages 285–297, 1999.

[19] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *Proc. of the 18th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 271–280, New York, NY, USA, 2006. ACM.

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

[21] J. L. Hennessy and D. A. Patterson. *Computer Architecture, A Quantitative Approach, 4th Edition*. Elsevier, 2007.

[22] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[23] L. G. Valiant. A bridging model for multi-core computing. In *Proc. ESA*, volume 5193 of *LNCS*, pages 13–28, 2008.