Resource Oblivious Sorting on Multicores

Richard Cole [*] Vijaya Ramachandran [†]

May 17, 2010

Technical Report **UTCS TR-10-13**
Department. of Computer Science
University of Texas at Austin

Abstract

We present a new deterministic sorting algorithm that interleaves the partitioning of a sample sort with merging. Sequentially, it sorts n elements in $O(n \log n)$ time cache-obliviously with an optimal number of cache misses. The parallel complexity (or critical path length) of the algorithm is $O(\log n \log\log n)$, which improves on previous bounds for deterministic sample sort. Given a multicore computing environment with a global shared memory and p cores, each having a cache of size M organized in blocks of size B, our algorithm can be scheduled effectively on these p cores in a cache-oblivious manner.

We improve on the above cache-oblivious processor-aware parallel implementation by using the Priority Work Stealing Scheduler (PWS) that we presented recently in a companion paper [11]. The PWS scheduler is both processor- and cache-oblivious (i.e., resource oblivious), and it tolerates asynchrony among the cores. Using PWS, we obtain a resource oblivious scheduling of our sorting algorithm that matches the performance of the processor-aware version. Our analysis includes the delay incurred by false-sharing. We also establish good bounds for our algorithm with the randomized work stealing scheduler.

1 Introduction

We present a new parallel sorting algorithm, which we call *Sample, Partition, and Merge Sort (SPMS)*. It has a critical path length of $O(\log n \log\log n)$ and performs optimal $O(n \log n)$ operations with optimal sequential cache misses. More importantly, using the PWS scheduler for multicores developed and analyzed in [11], and new algorithmic techniques given in this paper, we can schedule it resource-obliviously on a multicore while maintaining these performance bounds. We present background information on multicores, cache-efficiency and resource-obliviousness in Section 2.

The core of the sorting algorithm is a recursive multi-way merging procedure. A notable and novel aspect of this procedure is that it creates its recursive subproblems using a sample sort

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

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

methodology. We view the sorting algorithm as interleaving a merge sort with a sample sort in a natural way.

Previous Work. Sorting is a fundamental algorithmic problem, and has been studied extensively. For our purposes, the most relevant results are sequential cache-oblivious sorting, for which provably optimal algorithms are known [12], optimal sorting algorithms addressing pure parallelism [3, 10], and recent work on multicore sorting [5, 4, 6, 13].

The existing multicore algorithms take two main approaches. The first is merge sort [4, 6, 5], either simple or the pipelined method from [10]. The second is deterministic sampling [13]: this approach splits the input into subsets, sorts the subsets, samples the sorted subsets, sort the sample, partitions about a subsample, and recursively sorts the resulting sets. Our algorithm can be viewed as applying this approach to the problem of merging a *suitable number* of sorted sets, which eliminates the need for the first two steps, resulting in significant speed-up.

More specifically, the algorithm in [6] is a simple multicore mergesort; it has polylog parallel time, and good, though not optimal cache efficiency; it is cache-oblivious for private caches (the model we consider in this paper). The algorithm in [4] achieves the optimal caching bound on an input of length n, with $O(\log n)$ parallel time (modulo dependence on cache parameters), but it is both cache-aware and core-aware; this algorithm is based on [10]. The algorithm in [5] is cache oblivious with $O(\log^2 n)$ parallel time, but due to an additive term the cache performance is not optimal on a multicore. The algorithm in [13] is designed for a BSP-style version of a cache aware, multi-level multicore. It uses a different collection of parameters, and so it is difficult to compare with it directly.

Roadmap. In Section 2 we present some background on multicores, and then state our main sorting result. In Section 3 we give a high level description of our parallel sorting algorithm, omitting the details needed to have a resource oblivious implementation. In Section 4, we review the computation model, the work stealing scheduler PWS, and the class of BP algorithms, as developed in [11].

In Section 5, we return to the sorting algorithm, describing the details needed for a resource oblivious implementation. We then begin the analysis by considering the processor aware case, and give a matching lower bound for the number of cache misses in the multicore setting. In the following sections we analyze the fully oblivious complexity, in Section 6 with respect to the PWS scheduler, and in Section 7 with respect to the randomized work stealing scheduler. Finally, in Section 8, we discuss the results and the extent to which they are optimal.

2 Statement of our Results

Before stating our main result, we give some background, as developed in [11].

Multicore with Private Caches. We model a multicore as consisting of p *cores* (or processors) with an arbitrarily large main memory, which serves as a shared memory. Additionally, each core has a private cache of size M. Data in the main memory is organized in blocks of size B, and the initial input of size n is in main memory, in n/B blocks. When a core C needs a data item x that is not in its private cache, it reads in the block β that contains x from main memory. This new block replaces an existing block in the private cache, which is evicted using an optimal cache replacement policy (LRU suffices for our algorithms). If another core C' modifies an entry in β, then β is *invalidated* in C's cache, and the next time core C needs to access data in block β, an updated copy of β is brought into C's cache.

Cache and Block Misses. We distinguish between two types of cache-related costs incurred in a parallel execution.

The term *cache miss* denotes a read of a block from shared-memory into core C's cache, when a needed data item is not currently in the cache, either because the block was never read by core 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 occurs, and is accounted for, in sequential cache complexity analysis.

The term *block miss* denotes an update by a core $C' \neq C$ to an entry in a block β that is in core C's cache; this entails core C' acquiring block β; if C has a subsequent write, it needs to reacquire the block. This type of 'cache miss' does not occur in a sequential computation, and is a problematic situation that can occur quite often, especially in the resource oblivious setting that we seek.

Resource Obliviousness. We have claimed that our multicore algorithms are resource oblivious: we mean that the algorithm is specified without any mention of the multicore parameters (p, M and B) and further, the PWS scheduler we use schedules tasks on available idle cores, without reference to the multicore parameters. Since multicores with a wide range of parameters are expected to appear on most desktops, such a resource oblivious feature in multicore algorithms appears to be helpful in supporting the portability of program codes. The PWS scheduler uses *work-stealing* [8, 7], where load balance is achieved by cores stealing tasks from other cores as needed.

Our main result, the SPMS sorting algorithm and its resource-oblivious performance, has the bounds stated below in Theorems 2.1 and 2.2. Our analysis uses the following parameters. We suppose that each core performs a single operation in $O(1)$ time, a cache miss takes at most b time, a steal request takes at most s time (whether successful or not). We consider a multicore with p cores, each having a private cache of size M organized in blocks of size B, with all caches sharing an arbitrarily large global memory. The input, of size $n \geq Mp$ (this restriction ensures that both cores and caches can be fully utilized), is in the shared memory at the start of the computation.

PWS schedules the algorithm in rounds, where a round, roughly speaking, corresponds to a parallel step. The scheduler has additional work at the start of each round, work which takes at most S time per round. Then:

Theorem 2.1. *When scheduled under PWS, on an input of length n, assuming $M \geq B^2$ (the 'tall cache'), for $p \leq \frac{n}{\max\{\log\log n,\ M\}}$, the sorting algorithm SPMS takes parallel time*

$$O\left(\frac{1}{p}\left(n\log n + b\cdot\frac{n\log n}{B\log M}\right)\ +\ (b+s+S)\log n\log\log n\ +\ b\beta(n,p,B)\right).$$

The fourth term, $\beta(n,p,B) = O(B\log n + \frac{n}{pB}\frac{\log n}{\log M})$ is the block miss cost, and is bounded by the optimal sequential cache complexity provided $p \leq \frac{n}{B^2\log M}$ (i.e., with a slightly 'taller' cache — $M \geq B^2\log B$ suffices). This cost may also be reduced to match the optimal sequential cache complexity without this additional restriction on p if system support is provided for the locking of a block during writes, and limiting the minimum task size to be at least B.

If we ignore the block miss cost for the moment, this bound represents the optimal work bound, plus the optimal cache miss bound, plus the critical path length times the cost of one cache miss plus one steal plus one scheduling event. Further, we note that there is no need for a global clock, or tight synchronization on the part of the cores, though the scheduler certainly imposes a significant degree of synchronization. The computation is entirely resource-oblivious in that the algorithm

makes no mention of p, M or B, and PWS services idle cores without any reference to the number available or their cache parameters.

Extending the analysis of the randomized work stealer in [7, 1], we can obtain:

Theorem 2.2. *On an input of length n, assuming $M \geq B^2$, for $p \leq \frac{n}{\max\{\log\log n,\ M\}}$, the sorting algorithm SPMS when scheduled by the randomized work stealer, and taking into account both cache and block misses, takes expected parallel time*

$$O\left(\frac{1}{p}\left(n\log n + b\cdot\frac{n\log n}{B\log M}\right) \ + \ \left(s+\frac{M}{B}\cdot b\right)\left(\frac{bB}{s}\frac{\log n}{\log B}+\frac{(b+s)}{s}\log n\log\log n\right)\right).$$

Discussion. Our sorting algorithm is optimal in all respects except for the critical pathlength. The sorting algorithm for PEM in [4] achieves optimal $O(\log n)$ parallel steps, but is both cache- and core-aware. Achieving the same bound in a resource-oblivious manner appears considerably more challenging, and it is not clear if it is possible. We leave this as a topic for further research.

Another challenging topic is to extend our results to resource-oblivious scheduling on a multi-level caching hierarchy. Given the conflicting requirements of private and shared caches noted in [5, 9], it appears that some mechanism of supplying scheduler hints within the algorithm, and having a scheduler that uses the machine parameters effectively is needed. One such approach is used in [9]; however, that scheduler is *not* resource-oblivious, in contrast to our results.

In comparing our PWS scheduling to the PEM and Multi-BSP models, we note that these models both compute in a bulk-synchronous manner. We can easily adapt our results to work on either PEM or multi-BSP with the same performance as achieved with the PWS scheduler. However, our PWS framework adapts much more gracefully to differences in speeds among the cores than these bulk-synchronous models. Thus, if we have a few cores that execute faster than others (perhaps because they have smaller cache miss cost due to the cache layout), then PWS would enable the faster cores to take over (i.e, steal) work from the slower cores, balancing the work across the cores more effectively.

3 SPMS, A New Deterministic Sample, Partition, and Merge Sort

The heart of the algorithm is a procedure for computing a merging subproblem MS, whose input comprises r sorted lists $L_1, L_2, \cdots, L_r$, of total length m, with $m \leq r^c$, where $c \geq 6$ is a constant.

The sorting algorithm simply calls the merging procedure with $r = m = n$.

The merging algorithm performs two successive collections of recursive $\sqrt{r}$-way merges, each merge being on lists of total length at most $r^{c/2}$. To enable this, suitable samples of the input lists will be sorted by a logarithmic time procedure, which then allows the original problem to be partitioned into smaller subproblems that are merged recursively. More precisely:

Step 1. Partition the original problem MS into $k = O(m/r^{\frac{c}{2}-1})$ disjoint merging subproblems, $M_1, M_2, \cdots, M_k$, each comprising r sorted lists, with each subproblem having at most $r^{\frac{c}{2}}$ items in its r sorted lists. In addition, the items in M_i precede those in M_{i+1}, for $1 \leq i < k$.

Step 2. For each subproblem M_i, group its lists into disjoint subsets of $\sqrt{r}$ lists, and then in parallel merge the lists in each group. As M_i contains at most $r^{\frac{c}{2}}$ items, this bound applies to each of the individual groups too. Thus the $\sqrt{r}$-way merge in each group can be performed recursively. The output, for each subproblem M_i, is a collection of $\sqrt{r}$ sorted lists of total length at most $r^{\frac{c}{2}}$.

Step 3. For each subproblem M_i, recursively merge the $\sqrt{r}$ sorted lists computed in Step 2.

Step 1 details. The basic idea is to take a deterministic sample S of the input set comprising every $r^{\frac{c}{2}}$-th item in each list, to sort S, and to partition the r input lists about the items in S thereby forming smaller r-way merging subproblems. Some of these subproblems may have size as large as $r^{\frac{c}{2}+1}$, rather than the desired $r^{\frac{c}{2}}$. Any such subproblems are partitioned further, as needed, via samples S' of size $m'/r^{\frac{c}{2}-1}$ for each subproblem of size $m' \geq r^{\frac{c}{2}}$. The samples S and S' are sorted by performing all pairwise comparisons. More precisely:

Step 1.1. Let S comprise every $r^{\frac{c}{2}}$-th item in each of the input lists. Extract S from the input lists and then sort S, using a simple logarithmic time, quadratic work algorithm.

Step 1.2. Partition the r input lists $L_1, L_2, \cdots, L_r$ about S, creating subproblems $M'_1, M'_2, \cdots, M'_{k'}$, where $k' = |S| + 1$, and M'_i contains r sublists holding the items between the $(i-1)$th and ith items in S.

Step 1.3. Further partition any subproblem M'_i of size more than $r^{\frac{c}{2}}$, creating an overall collection of merging subproblems $M_1, M_2, \cdots, M_k$, each of size at most $r^{\frac{c}{2}}$, with the further property that the items in M_i precede those in M_{i+1}, for $1 \leq i < k$. This is done using a sample comprising every $r^{\frac{c}{2}-1}$-th item in M'_i.

Lemma 3.1. *The merging algorithm, on an input of r sorted lists of total length $m \leq r^c$, uses $O(m \log r)$ operations and $O(\log r \log\log r)$ parallel time, if $c \geq 6$.*

Proof. The parallel run time $T(r, m)$ is given by: $T(r, m) \leq \log r + 2T(\sqrt{r}, r^{c/2}) = O(\log r \log\log r)$.

Clearly, Steps 1.1 and 1.2 take $O(m)$ operations. To see the same bound applies to Step 1.3, we argue as follows. Each subproblem M'_i of size m' generates a sorting task of size $m'/r^{\frac{c}{2}-1} \leq r^{\frac{c}{2}+1}/r^{\frac{c}{2}-1} = r^2$. Performing all these sorting tasks requires at most $r^2 \cdot \sum m'/r^{\frac{c}{2}-1} \leq r^2 \cdot m/r^{\frac{c}{2}-1} \leq m$ operations, if $c \geq 6$.

Let $W(r, m)$ be the operation count for a collection of merging problems of total size m, where each comprises the merge of r lists of combined size at most r^c. Then we have: $W(r, m) \leq m + 2W(r^{1/2}, m) = O(m \log r)$. ■

Corollary 3.1. *The sorting algorithm, given an input of size n, performs $O(n \log n)$ operations and has parallel time complexity $O(\log n \log\log n)$, if $c \geq 6$.*

4 The Computation Model and PWS Scheduling

Before giving the resource oblivious implementation, we need to review the computation model and the PWS scheduling environment, mainly as developed in [11], although we make some changes here to address some new algorithmic features in SPMS.

The building blocks for our algorithms are computations on balanced binary trees such as for prefix sums. Such a computation is carried out by tasks: initially there is one task at the root of the tree; it forks two subtasks for each of its subtrees, and when they are done, it resumes and concludes the computation at the root. We will also use a tree of forking tasks to initiate a collection of parallel recursive calls, as needed in the merging and sorting algorithms.

Initially the root task for such a tree is given to a single core. Subtasks are acquired by other cores via task stealing. To this end, each core C has a task queue. It adds forked tasks to the bottom of the queue, while tasks are stolen from the top of the queue. So in particular, when C, on executing τ, generates forked tasks τ_1 and τ_2, it places the larger of τ_1 and τ_2 on its queue, τ_2 say, and continues with the execution of τ_1. This is a small generalization from [11], where the two

forked tasks were assumed to be of the same size. When C completes τ_1, if τ_2 is still on its queue, it resumes the execution of τ_2, and otherwise there is nothing on its queue so it seeks to steal a new task. Except for one routine, our algorithm will be constructed from BP trees [11], which are trees of equal-sized forking nodes with an $O(1)$ operation computation at each node, and with the leaf nodes having either an $O(1)$ operation task or a recursive computation as their task. There is a mirror image tree for the joins which also performs $O(1)$ operations at each node. We will often talk of a subtree of the BP tree, when we really intend a subtree plus the mirror image subtree.

Let τ be a task associated with such a subtree. As in [11], by the size of τ, $|\tau|$, we mean the amount of data τ accesses in its computation. In contrast to [11], sometimes we will use the *virtual size* of τ, $\text{vs}(\tau)$; always $\text{vs}(\tau) \geq |\tau|$. Efficiency is ensured by the following BP tree property: if τ' is forked by τ, then $\text{vs}(\tau') \leq \frac{1}{2}\text{vs}(\tau)$.

To help with the scheduling, each node in a BP tree receives the integer priority $\log \text{vs}(\tau)$. These are strictly decreasing from parent to child. We will use the Priority Work-Stealing Scheduler (PWS) [11], which only allocates tasks of highest priority in response to steal requests. As noted in [11], task priorities are strictly decreasing on each task queue, and thus there will be at most one steal of a task of priority d from each core, and so at most p steals of tasks of priority d, for any d. This is key to bounding the overhead of the PWS scheduler.

As noted in Section 2, the I/O cost of an individual task τ is measured in *cache misses*, which we upper bound by how many blocks the core executing τ has to read into its cache, assuming none are present at the start of τ's execution, and *block misses*, which capture the cost of multiple cores writing to the same block.

As we shall see, each BP task τ in the sort algorithm incurs $O(\text{vs}(\tau)/B + \sqrt{\text{vs}(\tau)})$ cache misses when executed sequentially. Each task incurs only $O(B)$ block miss delay: for most of the tasks this follows from [11] because they are *write-once* computations that engage in consecutive writes to a linear array, where a write-once computation is one that writes into any given location at most once. We will establish the same $O(B)$ block miss delay for the other class of tasks that arise in the sorting algorithm. We will use the following Fact from [11] about block miss delay in write-once computations.

Fact 4.1. *Let τ be a write-once computation executing on a core C. Each shared block in τ can cause C to incur a delay due to block misses that is no greater than that due to $B-1$ cache misses. If b is an upper bound on the cost of a cache miss, then each shared block in τ causes C to incur a delay of at most $b \cdot (B-1)$.*

We will use the following bounds derived in [11] for a collection of parallel BP computations of total size n and sequential cache complexity Q, when scheduled under PWS. Here, the maximum (virtual) size of any root task in the BP collection is x, and any task of size s incurs $O(s/B + \sqrt{s})$ cache misses and shares $O(1)$ blocks with other tasks:

Fact 4.2. *For the I/O cost for a computation of the type stated above:*

1. *The cache miss bound is $O(Q + p \cdot (\frac{\min\{M,x\}}{B} + \log \min\{x, B\} + \sqrt{x}))$.*
2. *The block miss bound is $O(p \cdot \min\{B, x\} \cdot (1 + \log \min\{x, \frac{n}{p}\})$.*

4.1 Ideal PWS Costing

The merging algorithm MS is built by combining (collections of parallel) BP computations, first by sequencing, and second by allowing the leaves of a BP tree to be recursive calls to the merging

algorithm. This generalizes the above tree computation to a dag which is a series-parallel graph. The formal definition of such an 'HBP' computation is given in [11]. While we do not need the details of a general HBP computation here, we do need to define priorities carefully in view of the possible differences in the sizes of the recursive subproblems generated by a call to MS. We define these priorities in a natural way so that they are strictly decreasing along any path in the MS computation dag, and all tasks with the same priority have roughly the same size, as detailed in Section 6.1.

The recursive subproblems generated in Step 2 of MS need not be of the same size, so this portion of the algorithm does not exactly fit the BP frame-work of [11]. To handle this, we will first determine the cache-miss overhead for the natural parallel implementation of the algorithm, which we call the *ideal PWS costing*, and then add in the additional cache-miss cost for the PWS schedule. (The cost of block misses is discussed later.)

Definition 4.1. *The* ideal costing *assumes that a BP computation uses at most* $2n/M$ *cores, one for each distinct subtree of size* M *and one for each node ancestral to these subtrees.*

The cache miss cost in the ideal costing is $O(M/B)$ per subtree, plus $O(1)$ for each ancestral node, for a total of $O(n/B)$ cache misses. The block miss cost in the ideal costing is $O(B)$ per subtree, which is $O(M/B)$ if $M \geq B^2$.

We generalize this BP analysis to MS and SPMS by analyzing the algorithm in terms of parallel collection of tasks, each task of virtual size M. The cost of each task collection is bounded in turn: each task is costed as if it was allocated to a distinct core. As we will see, each such collection has total virtual size $O(n)$, and hence incurs $O((n/B) + \frac{n}{M}\sqrt{M})$ cache misses, which is $O(n/B)$ if $M \geq B^2$.

To analyze the cache and block miss cost of the PWS scheduling of MS, we separate the cost of steals of small tasks τ (those with $\text{vs}(\tau) \leq M$), which we bound later, and consider the impact of steals of large tasks. To bound this cost for a large stolen task τ, we overestimate by supposing that no small tasks are stolen from τ. Then (the possibly overestimated) τ executes one or more of the size M subtrees that are executed as distinct tasks in the ideal PWS costing, plus zero or more nodes ancestral to these subtrees. Clearly, τ's cache and block miss cost is at most the sum of the cache and block miss costs of the corresponding distinct tasks in the ideal PWS costing. An analogous claim holds for the root task. Summing over the costs of the root task and of the large stolen tasks, yields that their total cost is bounded by the ideal PWS costing. We bound the cost of steals of small tasks using Fact 4.2, and results we derive in this paper.

A final point concerns the management of local variables in recursive calls. We assume that if a task τ stolen by a core C has local variables, then the space allocated by the memory manager for these variables does not share any blocks with space allocated to other cores. Further, if the data resides in cache till the end of C's execution of τ, then the now unneeded local variables are not written back to the shared memory. This assumption reduces the cost of block misses for stolen tasks corresponding to small recursive subproblems, as we shall see.

5 The Cache-Oblivious Parallel Implementation of SPMS

5.1 Additional Features in the Sorting Algorithm

To achieve efficient oblivious performance, the merging algorithm MS needs to be implemented using tasks achieving the optimal ideal costing, as defined above. Many of the steps in MS are

standard BP computations; their ideal costing is $O(n/B)$ and their PWS overhead can be bounded directly using Fact 4.2. However, here we address three types of computations in MS that do not fall within the framework in [11].

1. The recursion may well form very unequal sized subproblems. However, to achieve a small cache miss cost, the PWS scheduling requires forking into roughly equal sized subtasks. Accordingly we present the method of *grouping unequal sized tasks*, which groups subproblems in a task tree so as to achieve the balanced forking needed to obtain good cache-miss efficiency.

2. Balancing I/O for reads and writes in what amount to generalized transposes, which we call *transposing copies*. This is needed in the partitioning in Steps 1.2 and 1.3. An issue that arises here is the need to address the delay due to block misses in a multicore implementation.

3. One collection of tasks for sorting the samples in Step 1 uses non-contiguous writes. Fortunately, they use relatively few writes. We develop the *sparse writing* technique to cope.

Grouping Unequal Sized Tasks. We are given k ordered tasks $\tau_1, \tau_2, \cdots, \tau_k$, where each τ_i accesses $O(|\tau_i|/B)$ blocks in its computation (they are all recursive merges). We require that $t_i \leq t_{ave}^2$ for all tasks, where t_{ave} is the average size of the tasks.

The tasks need to be grouped in a height $O(\log k)$ binary tree, called the *u-tree*, with leaves holding the tasks in their input order. The u-tree is used for the forking needed to schedule the tasks. The u-tree will use virtual sizes for scheduling its subtasks and has the bounds given below. The proof of the following Lemma is given in Section 5.2.

Lemma 5.1. *The ideal PWS costing for scheduling the u-tree plus the cost of executing tasks τ_i of size M or less is $O(\sum_{i=1}^{k} t_i/B)$, where $t_i = |\tau_i|$.*

The Transposing Copy. The problem, given a vector A consisting of the sequence $A_{11}, \cdots, A_{1k}$, $\cdots, A_{h1}, \cdots A_{hk}$ of subvectors, is to output the transposed sequence $A_{11}, \cdots, A_{h1}, \cdots, A_{1k}, \cdots, A_{hk}$, where we are given that the average sequence length $l = |A|/hk \geq h$.

This is done by creating $\left\lceil \frac{|A_{ij}|}{l} \right\rceil$ tasks of virtual size l to carry out the copying of A_{ij}. The tasks are combined in column major order, i.e. in the order corresponding to destination locations. The result is that each task accesses $O(1)$ shared blocks in its writing, and gives an acceptable block miss cost of $O(B)$ cache misses for a task. We now bound the cache miss cost.

Lemma 5.2. *Let τ be a task copying lists of combined size s in the transposing copy. Then τ incurs $O(s/B + \sqrt{s})$ cache misses.*

Proof. Suppose that τ copies g lists. Then τ incurs $O(s/B + g)$ cache misses. Note that $s \geq g \cdot l$ as each list has virtual size at least l.
Case 1. $g \leq l$.
Then $s \geq g \cdot l \geq g^2$. So in this case the cache miss bound is $O(s/B + \sqrt{s})$.
Case 2. $g > l$.
Case 2.1. $l \geq B$.
As $s \geq g \cdot l$, $s/B \geq g$, thus τ incurs $O(s/B)$ cache misses in this case.
Case 2.2. $l < B$.
The core, if using an optimal cache replacement policy, would keep in cache the most recently read block from each row, for which it has sufficient space, assuming that $M \geq B^2$, as $h \leq l \leq B$. As a result, the task uses at most $2h$ partially read blocks, giving an overall bound of $O(s/B + h)$ cache misses. And $O(s/B + h) = O(s/B + l) = O(s/B + \sqrt{g \cdot l}) = O(s/B + \sqrt{s})$.

It is straightforward to argue that an LRU policy for cache block replacement will also yield this bound if $l \leq B/2$, which still results in the $O(s/B + \sqrt{s})$ cache miss bound. ■

Sparse Writing. Let A be an $s \times s$ array in which each of locations $c \cdot s$, $1 \leq c \leq s$ is written exactly once, but not in any particular order. A sequential execution incurs at most s cache misses.

Now consider a BP execution of this computation in which each leaf is responsible for one write. We claim that the I/O cost for all writes to A is $O(s^2/B + B)$ regardless of the ordering of the writes. We establish this bound as follows.

If $s \geq B$, each write into A incurs one cache miss, for a total cost of $O(s) \leq O(s^2/B)$ cache misses. There are no block misses in this case.

If $s < B$, there are only s accesses, but these can incur block misses. Let i be the integer satisfying $s \cdot i \leq B < s \cdot (i+1)$. Then, at most i writes occur within a single block. Each such write to a block may incur a block wait cost equal to that of $\Theta(i)$ cache misses. Hence the overall delay in this case is at most that of $O(s \cdot i) = O(B)$ cache misses.

5.2 The u-Tree Construction

Suppose k ordered tasks $\tau_1, \tau_2, \cdots, \tau_k$ are given, where each τ_i accesses $O(|\tau_i|/B)$ blocks in its computation (they are all recursive merges). The tasks need to be grouped in a height $O(\log k)$ binary tree, called the *u-tree*, with leaves holding the tasks in their input order.

The u-tree will satisfy the following properties: (i) the virtual sizes at least double from child to parent; (ii) the virtual size of a node is at least the actual size of the single task it holds if it is a leaf node; (iii) the virtual size of the root task is $O(\sum_{i=1}^{k} t_i)$, where t_i denotes $|\tau_i|$.

Let $t_{\max}$ be the size of the largest task and t_{av} be the average size of a task. We show how to group the tasks so as to achieve a balanced grouping. This will increase the work by at most an additive $O(\sum_{i=1}^{k} t_i)$ factor, so long as $t_{\max} \leq t_{av}^2$.

First, the size of each task is rounded up to be a multiple of t_{av}. The sizes are further increased by additive factors of t_{av} (at most $2t_{av}$ per task) so that the total of the tasks sizes is of the form $2^k t_{av}$ for some integer k (this is readily done by means of a prefix sum-style computation). For each task τ_i, the resulting size provides its *enlarged size* and is denoted by t'_i.

Let $t''_i = t'_i/t_{av}$ and $t''_{\max} = \max_i t''$. The tasks are then spread out, as follows, into an initially all zero array T of length $s'' = \sum_{i=1}^{k} t''_i$, with the ith task at location $s''_i = \sum_{h=1}^{i-1} t''_h + \lceil t''_i/2 \rceil$ (obtained via a prefix sums computation), so the ith task is in the middle of an interval, called I_i, of length t''_i.

1. Write (τ_i, s''_i) to $M[i \cdot t''_{\max}]$, using a size $t''_{\max}$ task.
2. Create $k \cdot t''_{\max}$ size one tasks. The jth task reads $M[j]$ and if the entry is nonzero, (τ_i, s''_i) say, copies τ_i to location $T[s''_i]$.

Note that as $t_{\max} \leq t_{av}^2$, $k \cdot t''_{\max} \leq k \cdot t_{av} = O(\sum_{i=1}^{k} t_i)$.

Let $l_i = \sum_{h=1}^{i} t''_h$; interval $I_i = (l_{i-1}, l_i]$ is associated with task τ_i.

Now a standard binary tree B is placed over the s'' cells of array T; its nodes are stored in inorder. We define the interval $I(u)$ of a node u to be the interval of leaves spanned by the subtree rooted at u. The virtual size of u, $vs(u)$, is defined to be $4 \cdot t_{av} \cdot I(u)$.

Tree B is then trimmed, removing task-free portions, and compressing single child paths, keeping the bottommost node on paths ending at an internal node, and the topmost node on paths ending at

a leaf. Thus the leaf node for task τ_i in the trimmed tree is the following node from the untrimmed tree: the highest ancestor u of location s_i'' whose interval $I(u)$ is wholly contained in I_i.

Next, we show that the just defined virtual size has the properties we seek.

Lemma 5.3. *i. If u is the parent of v in the trimmed B, then $vs(u) \geq 2 \cdot vs(v)$.*
ii. Let r be the root of B. Then $vs(r) \leq 4 \sum_i t_i' = O(\sum_i t_i)$.
iii. Let u_i be the leaf node in the trimmed tree corresponding to task τ_i. Then $vs(u_i) > t_i'$.

Proof. (i) follows immediately from the definition of $I(u)$. (ii) follows because $vs(r) = 4 \cdot t_{av} \cdot |I(r)| \leq 4 \cdot t_{av} \cdot \sum_i t_i'' = 4 \cdot \sum_i t_i'$.

To see (iii) we argue as follows. Since interval $I(u_i)$ is the longest such interval wholly contained in I_i, $4 \cdot |I(u_i)| > |I_i|$, and thus $vs(u_i) = 4 \cdot t_{av} \cdot |I(u_i)| > t_{av} \cdot |I_i| = t_{av} \cdot t_i'' = t_i'$. ■

The computation of the final tree structure is done via an up-pass over tree B. Clearly the tree building is a BP computation and thus incurs $O(s''/B)$ cache-misses (this depends on storing B's nodes in inorder).

Lemma 5.1. *The ideal PWS costing of the u-tree including the cost of executing tasks τ_i of size M or less is $O(\sum_{i=1}^{k} t_i/B)$.*

Proof. There are $O(\sum_{i=1}^{k} t_i/M)$ internal nodes in the u-tree having virtual size M or more. The fringe nodes (the nodes having virtual size in the range $(M/2, M]$ plus smaller second children of larger nodes, if any), each incur $O(M/B)$ cache misses. Since virtual sizes at least double from child to parent, there are $O(\sum_{i=1}^{k} t_i/M)$ such nodes. Thus executing the tasks associated with these nodes incurs a total of $O(\sum_{i=1}^{k} t_i/M \cdot M/B) = O(\sum_{i=1}^{k} t_i/B)$ cache misses. ■

5.3 Details of Step 1 in SPMS

Now, we describe the algorithm in detail.

Each substep (except one) uses a BP computation or a collection BP computations running in parallel. We characterize the complexity of each size x (collection of) computations. Clearly it will have depth $O(\log x)$, and unless otherwise specified will incur $O(x/B)$ cache misses. The only other cache miss cost that occurs is $O(x/B + \sqrt{x})$, and this arises only in the transposing copy.

Our design approach for achieving these cache miss bounds is to constrain each ordered collection of parallel tasks, except for those used in a transposing copy, as follows. We ensure that any sequence of tasks, having combined virtual size x, incurs $O(x/B)$ cache misses; this implies that a single core executing such a sequence of tasks also incurs at most this cost. The most common pattern achieving this is for such a collection of tasks to read one contiguous length $O(x)$ array segment in left to right order. Another pattern allows one or more subsegments σ to be read several times, with the contribution to the virtual size of the tasks being $O([\text{number of times } \sigma \text{ is read}] \cdot |\sigma|)$.

Some additional notation is helpful. Let $L_1, L_2, \cdots, L_r$ be the r sorted input lists of total length $m \leq r^c$. The r lists are stored in sequential order. Let $S = \{e_1, e_2, \cdots, e_s\}$ comprise every $r^{\frac{c}{2}}$th item in the sequence of sorted lists; recall that S is sorted in Step 1.1, and then used to partition the r lists in Step 1.2.

Step 1.1. Sort S.

1.1.1. Construct arrays $S_1, S_2, \cdots, S_s$; each S_i is an array of length s which contains a copy of the sequence of elements in S.

(a) Compact the s samples within the list sequence $L_1, \cdots, L_r$, using prefix sums for the compaction. The result is an array $S_1[1..s]$ containing the s samples. This uses a sequence of 2 BP computations of size m.
(b) Form arrays S_i, $2 \le i \le s$, where each S_i is a copy of S_1. This is a BP computation of size $s^2 \le m$.

1.1.2. In parallel for each i, compute rank of e_i in S_i.
First, for each S_i, compare e_i to each element in S_i. Then count the number of $e_j \ge e_i$, the desired rank of e_i in S_i. This uses two BP computations, and over all i, $1 \le i \le s$, they have combined size $O(s^2)$.

1.1.3. Create the sorted array $S[1..s]$ where $S[i]$ contains the element e_j with rank $\rho_j = i$.
The simple way to implement this step is for each element e_i to index itself into location $S[\rho_i]$. This will incur s cache misses, which can be argued is acceptable with a tall cache, since $s^2 = O(m)$. But this implementation could incur $s \cdot B$ block misses, which is excessive. To reduce the block miss cost, we split this step into two substeps:
(a) Initialize an all-zero auxiliary array $A'[1..m]$ and write each e_i into location $A'[\rho_i \cdot r^{c/2}]$.
This is the sparse writing setting analyzed earlier, and results in $O(s^2/B + B) = O(m/B + B)$ cache and block misses in a depth $\log s$ computation.
(b) Compact array A' into $S[1..s]$, which gives the desired sorted array of the samples. This is a prefix sums computation, a BP computation of size $O(m)$.

Step 1.2. Partition $L_1, L_2, \cdots, L_r$ about S.
Further notation: Let $\text{rank}(e, U)$ denote the rank of e in sorted set U if $e \in U$ and of the predecessor of e in U otherwise. Let SL_i denote $S \cap L_i$.

1.2.1. Find the rank of each $e \in S$ in each list L_i.
This step uses a distinct copy of the sorted S for each list L_i.
For each pair of successive items from SL_i, e_j and e_k say, it forms the subproblem of partitioning the $r^{\frac{c}{2}}$ items of L_i lying between e_j and e_k (readily identified from $\text{rank}(e_j, SL_i)$) w.r.t. those items of S in the range (e_j, e_k). This is anywhere from 0 to $r^{\frac{c}{2}}$ items in S. For e in this range, define $i\text{-rank}(e) = \text{rank}(e, SL_i)$).
For each item $e \in S$, compare it to those items in L_i with rank in the range $[r^{\frac{c}{2}} \times i\text{-rank}(e_j) + 1, r^{\frac{c}{2}} \times i\text{-rank}(e_j) + (r^{\frac{c}{2}} - 1)]$. As this entails too many comparisons, the comparisons are done in two stages:
First, e compares itself to every $r^{\frac{c}{2}-1}$-th item in the relevant range, that is to r items. e thereby identifies the two items f' and f'' straddling it; then e compares itself to the $r^{\frac{c}{2}-1}$ items between f' and f''.

Obtaining the sets of r and $r^{\frac{c}{2}-1}$ items that e wants to compare to itself is nontrivial. We proceed as follows.

a. Every $r^{\frac{c}{2}-1}$-th item is extracted from L_i, forming a list L_i'. Each item $e \in S$ needs to compare itself to r of these items.

The extraction is a compaction, done as in Step 1.1.1a. It uses a size m BP computation. Then, for each list L_i and each $e \in S$, the relevant r items are copied into a subarray. For each L_i, the copying subtasks (of size r) for each $e \in S$, are nodes in a BP tree, with the subtasks in the same order as the sorted items in S.

b. For each $e \in S$, the relevant collection of all $r^{\frac{c}{2}-1}$ items needs to be extracted from L_i, as follows.

b1. Each item e determines if it is the leftmost item seeking this interval of $r^{\frac{c}{2}-1}$ items. If so, it marks the corresponding left boundary item in L'_i.

For each i, this is a BP computation of virtual size rs. Each leaf node is given virtual size r, and the virtual size of an internal node is just the sum of the virtual sizes of its children. This ensures that the length of the interval accessed in the writes is always O(task size).

b2. A prefix sum of the marked items is calculated. Each item in L'_i creates a task of virtual size $r^{\frac{c}{2}-1}$. The task for an unmarked item does nothing. The task for a marked item copies the corresponding interval of $r^{\frac{c}{2}-1}$ items to the location indicated by the prefix sum (multiplied by $r^{\frac{c}{2}-1}$).

A task for x consecutive items in L'_i, marked or unmarked, accesses a subarray of length at most $x \cdot r^{\frac{c}{2}-1}$, and so the BP computation for L'_i has virtual size $|L'_i| \cdot r^{\frac{c}{2}-1}$. Over all i, this gives a combined size of $s \cdot r \cdot r^{\frac{c}{2}-1} = O(m)$.

c. The leftmost items from (b1) read the location where their size $r^{\frac{c}{2}-1}$ data is written, and disseminate the location by means of a scan to the other items in S wanting to read the same data. For each i, the BP computation for L'_i has size $|L'_i| \cdot r^{\frac{c}{2}-1}$.

d. Each item in S copies the $r^{\frac{c}{2}-1}$ data items it needs. For each i, this is a BP computation of size $s \cdot r^{\frac{c}{2}-1}$.

1.2.2. Perform the partitioning by means of a transposing copy.

Step 1.3. For each subproblem M'_i with $|M'_i| > r^{\frac{d}{2}}$ create a task to further partition M'_i. It is analogous to Steps 1.1 and 1.2 except that it uses a sample S' of size $m'_i/r^{\frac{d}{2}-1}$, where $m'_i = |M'_i|$.

5.4 Ideal PWS Costing for the Merge

Summarizing the above discussion of the cache-miss costs for the merge (MS) gives the following bound for the number of cache misses in the ideal PWS costing.

Lemma 5.4. *In the ideal PWS costing, the merging algorithm MS, in performing a collection of merging tasks of total size $n \geq Mp$, in which each task comprises the merge of r lists of combined length at most r^c, incurs $O(\lceil \frac{n}{B} \rceil \left\lceil \frac{\log r}{\log M} \right\rceil)$ cache-misses, if $c \geq 6$ and $M \geq B^2$.*

Proof. As argued in the description of the algorithm, for each merging problem of size $m = \Omega(M)$, Substep 1 incurs $O(m/B + \sqrt{m}) = O(m/B)$ cache-misses, as $M \geq B^2$; smaller subproblems fit in cache and so incur $O(\lceil m/B \rceil)$ cache-misses.

Now let $C(r, n)$ be the cache-miss count for performing such a collection of merges for problems of merging r lists each of combined size at most r^c. Then, as the algorithm uses $O(n)$ space, we have, for a suitable constant $\gamma > 1$: for $n \leq \gamma M$: $C(r, n) = \lceil n/B \rceil$, and for $n \geq \gamma M$: $C(r, n) \leq \frac{n}{B} + 2C(r^{1/2}, n)$. ■

5.5 Analysis of Processor-Aware Cache-Oblivious SPMS

For a processor-aware, cache-oblivious implementation of SPMS, there is only a constant number of block misses per task executed by a core, costing $O(bB)$ per task by Fact 4.1, and the number of tasks in a p-core processor-aware implementation is $O(p \cdot \frac{\log n}{\log(n/p)})$. Thus, the block miss cost is

dominated by the cache miss cost under our assumption that $n \geq Mp$ and $M \geq B^2$. Hence, with the above analysis and the parallel time bounds for the basic SPMS algorithm, as well as for the BP computations in the implementations given in this section, we obtain the result that SPMS can be scheduled on p cores, for $p \leq \frac{n}{\max\{M, \log\log n\}}$, to obtain optimal speed-up and cache-oblivious cache-efficiency, including the block miss cost. Note that in such a processor-aware schedule, there is no need for steals, and hence there is no further overhead beyond the cache-miss, block miss, and depth bounds that we have established for the computation.

Lower Bound for Cache Misses. We can adapt the I/O lower bound for sorting given in [2] to the multicore setting. It is not immediately evident their bound applies to our setting, for the bound in [2] concerns a machine with a single cache of size M, whereas the multicore setting has p caches each of size M. Conceivably, these p caches could have the same power as a single size pM cache in the original setting. In fact, they do not as we show.

Recall that the lower bound in [2] is obtained by counting the reduction in the number of permutations consistent with the new comparisons enabled by a single read into a cache of size M. We can similarly analyze the effect in the multicore setting by serializing the I/O, and noting that each memory access is performed by some core with a cache of size M.

For completeness, we summarize the result obtained in [2]: a bound of $\Omega(\frac{n}{B}\frac{\log n/B}{\log M/B})$ cache misses for sorting in the comparison model. As we assume that $n \geq M \geq B^2$, this is also a bound of $\Omega(\frac{n}{B}\frac{\log n}{\log M})$ cache misses. When an input block is first brought into a local memory, the number of permutations is reduced by $B!$ simply due to the permutation of its B elements. This can happen n/B times. Additionally, each time a block is loaded into a local memory it reduces the number of possible permutations by a factor of at most $\left(\begin{array}{c} M \\ B \end{array} \right)$, the number of ways of inserting B sorted items into $M - B$ sorted items. Thus if C is the number of cache misses the algorithm incurs, then

$$(B!)^{n/B} \left(\begin{array}{c} M \\ B \end{array} \right)^C \geq n!$$

yielding $C = \Omega(\frac{n}{B}\frac{\log n/B}{\log M/B})$.

In the multicore setting, there is, in addition, a choice of p memories (i.e., caches), one per core. However, each read of a block causes that block to be brought into some specific cache, and this causes the same decrease in the number of permutations as in the sequential case. Hence the lower bound remains the same in the multicore setting.

Finally, we note that a parameter P is used in [2] . This denotes the number of parallel disks (i.e., external memories), and is unrelated to p, the number of parallel caches (i.e., internal memories).

6 Resource Oblivious Scheduling under PWS

6.1 Assigning Priorities

The goal for the scheduling priorities is to ensure that computations which are meant to occur in parallel have the same priority. In our companion paper [11], the priority of a task was simply its depth in the computation dag. However, in the merging algorithm, leaves of u-trees may have different sizes and as a result be at different depths. But we need to ensure the start of the leaf computations form an implicit synchronization point, by giving all of them equal priority.

To this end, for each recursive invocation I of the merge procedure, we keep two counters. One counter records the number of non-recursive procedure calls made directly from I; let s denote this counter.

The second counter, named r, records the maximum number of successive recursive calls of the merge procedure on any path from the start of the computation to I. Both r and s are local to invocation I. The counter s is initialized to 0. Whenever a new BP procedure or a new u-tree computation is invoked by I, s is incremented. The counter r receives its initial value as part of the input associated with I's invocation. The first call in the whole computation has $r = 1$. Immediately prior to its invocation of a recursive call, I increments r and this new value is the input to the recursive call being made by I. On return from the recursive call, I inherits the increased value of r. To ensure consistency on return from a collection of parallel recursive calls, I computes the maximum of these returned values. This is readily done during the up-pass of the u-tree.

The priority of a task τ is given by the triple (r, s, depth), ordered lexicographically, where the depth refers to the depth in a BP computation, if τ is a subtask of a BP computation, or to $\log \text{vs}(\cdot)$ if τ is a subtask of a u-tree computation. (Note that the first two parameters are ordered by decreasing value, that is larger values have lower priority, and the third field is ordered by decreasing depth/increasing size.)

6.2 The Analysis of the PWS overhead

In addition to the results in Fact 4.2, the companion paper [11] shows that:

1. The cost of each up-pass is bounded by that of the corresponding downpass in BP and HBP algorithms.

2. The idle work (the time spent by a core when it is not computing nor writing on a cache or block miss), in a (parallel collection of) BP computations, aside the waiting already accounted for in the up-pass, is bounded by $O(p \cdot ((s+S+b) \log x + b \min\{x, B\}))$ where x is the size of the largest task in the collection, s bounds the time for a steal, S bounds the time to initiate a scheduling round, and b bounds the time for a cache miss.

3. Additional cache miss costs due to small tasks taking over the work of large tasks on an up-pass are bounded by the cache miss costs in the downpass.

And, as already noted in this paper, for the present algorithm:

4. The delay due to block misses for a stolen task τ is bounded by the time for $O(B)$ cache misses. This follows from results in Fact 4.1, for block misses, and from our method for Step 1.1.3, described earlier.

Lemma 6.1. *Assuming that $M \geq B^2$, the additional cache misses, $C_M(n, r^c)$, due to stolen tasks of size M or less in the merging algorithm for a collection of merging problems each of size at most r^c, and of total size $n \geq Mp$ is bounded by:*

$$\begin{cases} p\dfrac{M}{B}\dfrac{\log r^c}{\log M} & \text{if } r^c \geq M \\ p(\dfrac{r^c}{B} + r^{c/2}) & \text{if } B \leq r^c < M \\ pr^{c/2} & \text{if } r^c < B. \end{cases}$$

Proof. Using Fact 4.2 (which states the bounds in [11]), the top level BP computation causes the

following number, $\mathrm{CT}(n, r^c)$, of cache misses:

$$\begin{cases} p\dfrac{M}{B} & \text{if } r^c \geq M \\ p\left(\dfrac{r^c}{B} + \log B + r^{c/2}\right) & \text{if } B \leq r^c < M \\ p(r^{c/2}) & \text{if } r^c < B. \end{cases}$$

And C_M is given by the following recurrence equation:

$$C_M(n, r^c) \leq \mathrm{CT}(n, r^c) + 2C_M(n, r^{c/2}).$$

Induction confirms the claimed bound. ■

We now bound the block miss delay in the merging algorithm.

Lemma 6.2. *Let $M \geq B^2$. The delay $\mathrm{BM}_M(n, r^c)$ due to block misses in the merging algorithm for a collection of merging problems each of size at most r^c, and of total size $n \geq Mp$, is bounded by: $pB \log r^c(\log\log \frac{n}{p} - \log\log B)$ if $r^c \geq B$ and $B \leq \frac{n}{p} < r^c$, $pB \log r^c(\log\log r^c - \log\log B)$ if $r^c \geq B$ and $\frac{n}{p} \geq r^c$, and by $pr^c \log r^c$ if $n/p, r^c < B$.*

Proof. Using the bounds in Fact 4.2 for block misses, and since $M \geq B^2$ the top level BP computation causes the following number, $\mathrm{BMT}(n, r^c)$, of block misses: $pB \log \frac{n}{p}$ if $\frac{n}{p} \leq r^c$ and $r^c \geq B$, $pB \log r^c$ if $\frac{n}{p} > r^c$ and $r^c \geq B$, and $pr^c \log r^c$ if $r^c < B$.

Since $\mathrm{BM}_M(n, r^c) \leq \mathrm{BMT}(n, r^c) + 2\mathrm{BM}_M(n, r^{c/2})$, induction confirms the claimed bound. ■

Corollary 6.1. *Let $M \geq B^2$. The delay $\mathrm{BM}_M(n, r^c)$ due to block misses in the merging algorithm for a collection of merging problems each of size at most r^c, and of total size $n \geq Mp$, is bounded by: $pB \log n + \frac{n}{B}\frac{\log n}{\log M}$.*

Proof. By Lemma 6.2, the cost is bounded by $pB \log n(\log\log \frac{n}{p} - \log\log B)$. We now bound this expression.

Case 1. $n/p \geq M^3$.
Then

$$\begin{aligned} Bp \log n(\log\log n/p - \log\log B) &\leq Bp\frac{(n/p)/(\log n/p \log\log n/p)}{M^3/(\log M^3 \log\log M^3)} \log n \log\log n/p \\ &\leq \frac{n}{B}\frac{\log n/\log n/p}{M^3/(B^2 \log M^3 \log\log M^3)} \\ &\leq \frac{n}{B}(\log n/\log M). \end{aligned}$$

Case 2. $M^3 \geq n/p \geq (B \log M)^3$.

Then

$$\begin{aligned} Bp\log n(\log\log n/p - \log\log B) &\leq Bp\frac{(n/p)/(\log\log n/p)}{(B\log M)^3/(\log\log(B\log M)^3)}\log n\log\log n/p \\ &\leq \frac{n}{B}\frac{\log n/\log M}{(B\log M)/(\log\log(B\log M)^3)} \\ &\leq \frac{n}{B}\frac{\log n}{\log M}. \end{aligned}$$

Case 3. $n/p \leq (B\log M)^3$ and $\log M \geq B$.
Then

$$\begin{aligned} Bp\log n(\log\log n/p - \log\log B) \leq Bp\log n\log\log(\log M)^6 &\leq \frac{Mp}{B}\frac{\log n}{\log M}\frac{B^2\log M\log\log(\log M)^6}{M} \\ &\leq \frac{n}{B}\frac{\log n}{\log M}. \end{aligned}$$

Case 4. $n/p \leq (B\log M)^3$ and $\log M \leq B$.
Then $Bp\log n(\log\log n/p - \log\log B) = O(Bp\log n)$. ■

Finally, we bound the idle time in each recursive call to a collection of merging problems.

Lemma 6.3. *The idle time, $I_M(n, r^c)$, in the merging algorithm for a collection of merging problems each of size at most r^c, and of total size n is bounded by: $O(p\log r^c\log\log r^c(S+s+b)+pbB\frac{\log r^c}{\log B})$.*

Proof. The top level BP computation causes the following idle time, $\mathrm{IT}(n, r^c)$: $O(p\log r^c(S+s+b)+pb\min\{B, r^c\})$. And I_M is given by the following recurrence equation:

$$I_M(n, r^c) \leq \mathrm{IT}(n, r^c) + 2I_M(n, r^{c/2}).$$

Induction confirms the claimed bound. ■

Corollary 6.2. *The costs $C_S(n, r^c)$, $\mathrm{BM}_S(n, r^c)$, $I_S(n, r^c)$ for respectively, the cache misses, the block misses and the idle time of the sorting algorithm when run on a collection of sorting problems each of size at most r^c, and of total size $n \geq \max\{Mp, r^{c-1}\}$ have the same asymptotic bounds as the merging algorithm.*

Adding the costs given by Lemmas 5.4–6.3 and Corollary 6.1 yields our main result:

Theorem 2.1. *When run on $p \leq \min\{\frac{n}{\log\log n}, \frac{n}{M}\}$ cores using the PWS scheduler on an input of size $n \geq Mp$, where $M \geq B^2$, the merging algorithm runs in time*

$$O\left(\frac{n\log n}{p} + \frac{bn}{Bp}\frac{\log n}{\log M} + \log n\log\log n(s+S+b) + bB\log n\right).$$

The same bound applies to the sorting algorithm.

7 Analysis for the Randomized Work Stealing Scheduler

In our companion paper [11] we present an analysis of the expected running time, including the effect of block misses, when the randomized work stealing scheduler is used in computations that expose parallelism with binary forking of tasks. The following theorem is obtained by applying the result in [11] to the sorting algorithm.

Theorem 2.2 *On an input of length n, assuming $M \geq B^2$, for $p \leq \frac{n}{\max\{\log\log n,\ M\}}$, the sorting algorithm SPMS executed with the randomized work stealer, takes expected parallel time*

$$O\left(\frac{1}{p}\left(n\log n + b\cdot\frac{n\log n}{B\log M}\right) \ + \ \left(s+\frac{M}{B}\cdot b\right)\left(\frac{bB}{s}\frac{\log n}{\log B}+\frac{(b+s)}{s}\log n\log\log n\right)\right).$$

Proof. Let σ be the expected number of attempted steals performed by the randomized work stealer when SPMS is executed on an input of length n. Then, by our earlier analysis, the expected overhead in the number of cache misses due to attempted steals beyond the sequential cache complexity is $O\left(\sigma\cdot b\cdot\frac{M}{B}\right)$. The cost of the attempted steals is $O(s\sigma)$, as a single attempted steal costs $O(s)$.

In [11] it is shown that $\sigma = O(p\cdot\lceil\frac{T'_\infty}{s}\rceil)$, where s is the cost of an attempted steal. Here, T'_∞ is the *augmented critical path length* of the computation, which is bounded by b times the standard critical pathlength T_∞, plus a bound γ on the total cost of the block misses on any path in the computation dag, plus $2s$ times a bound ϕ on the number of forks on any path.

For SPMS, we have $T_\infty = O(\log n\cdot\log\log n)$, and ϕ is bounded by the same bound. For γ, we observe that any path in the SPMS computation has a sequence of $O(\log n)$ BP computations. Of these BP computations, $\Theta\left(\frac{\log n}{\log B}\right)$ have size B or larger, and each incurs $O(1)$ block misses on the given path, each causing a delay of $O(bB)$. Further, $\Theta\left(\frac{2^i\log n}{\log B}\right)$ of the BP computations on a path have size $B^{1/2^i}$, each incurring $O(1)$ block misses, and each causing a delay of $O(b\cdot B^{1/2^i})$, for this is the amount of data in the recursive subproblem to which the BP computation belongs. Summing over all i yields $\gamma = O(bB\frac{\log n}{\log B})$. Thus $T'_\infty = O\left(bB\frac{\log n}{\log B}+(b+s)\log n\log\log n\right)$. This also establishes the second term in the bound in the theorem as the bound for the expected cache-miss overhead for the steals plus the overhead for the attempted steals.

The overhead for block misses in a stolen task is dominated by that for cache misses, since there are only $O(1)$ block misses per task, giving rise to a cost of $O(b\cdot B)$, which is $O(b\cdot(M/B))$ with a tall cache. Thus the cost of the overhead due to attempted steals under randomized work-stealing is

$$O\left(\left(1+\frac{M}{B}\cdot\frac{b}{s}\right)\left(bB\frac{\log n}{\log B}+(b+s)\log n\log\log n\right)\right).$$

Adding in the remaining cost of $(1/p)$ times (work + processor-aware cache-oblivious cache and block miss cost) gives the result stated in the theorem. ■

Comment. In the randomized work stealing environment, task priorities are not used. Also, the u-tree construction for grouping unequal sized tasks is not needed; instead, the straightforward pairwise grouping suffices.

8 Discussion

In this paper we have presented a new sorting algorithm which combines elements from merge-sort and sample-sort. The algorithm has optimal sequential running time, optimal sequential cache complexity, and optimal parallel speed-up up to critical pathlength $O(\log n \log\log n)$.

We analyzed the parallel cache complexity, including the additional overhead due to block misses (or 'false-sharing'), and we matched the sequential cache miss bound of $\Theta\left(\frac{n}{B}\log_M n\right)$, when each core has a private cache of size M, with the sequential requirement of $M \geq B^2$ (a tall cache). With p cores, these bounds hold when the input size $n \geq Mp$, since with a smaller input size, either the cores or the cache capacity will not be used fully effectively. We also showed that our cache miss bound continues to be optimal in the multicore setting.

Finally, we analyzed the performance of the sorting algorithm when executed resource-obliviously using work-stealing. When executed using the deterministic Priority Work Stealing (PWS) scheduler described in our companion paper [11], our sorting algorithm runs resource obliviously while matching the bounds of the processor-aware version as long as the cache is slightly taller with $M \geq B^2 \log B$. These bounds again include the effect of block misses.

We have also analyzed the resource-oblivious performance of our algorithm when executed using the randomized work-stealer, and when including the cost of block misses.

When implemented with PWS our algorithm is essentially optimal in all respects except for the critical pathlength, which is $\Theta(\log n \log\log n)$ for our algorithm. It appears to be quite challenging to achieve $O(\log n)$ critical pathlength while retaining cache-optimality and resource obliviousness. We have not considered a multi-level cache hierarchy, and here a *multicore-oblivious* implementation as in [9] is achievable, but a resource-oblivious version does not appear to be likely.

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] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *CACM*, 31:1116–1127, 1988.

[3] M. Ajtai, J. Komlos, and E. Szemeredi. An O(n log n) sorting network. *Combinatorica*, 3:1–19, 1983.

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

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

[6] G. Blelloch, P. Gibbons, and H. Simhadri. Brief announcement: Low depth cache-oblivious sorting. In *ACM SPAA*. ACM, 2009.

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

[8] F. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *ACM FPLCA*, pages 187–194, 1981.

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

[10] R. Cole. Parallel merge sort. *SIAM J Comput*, 17(4), 1988.

[11] R. Cole and V. Ramachandran. Efficient resource oblivious scheduling of multicore algorithms. Manuscript, 2010.

[12] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *IEEE FOCS*, 1999.

[13] L. G. Valiant. A bridging model for multi-core computing. In *Proc. of the 16th Annual ESA*, volume 5193, pages 13–28, 2008.