Oblivious Algorithms for Multicores and Networks of Processors

Rezaul Alam Chowdhury[a], Vijaya Ramachandran[b], Francesco Silvestri[c,*], Brandon Blakeley[d]

[a] *Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-4400, USA*
[b] *Department of Computer Science, University of Texas, Austin, TX 78712, USA*
[c] *Department of Information Engineering, University of Padova, Padova 35131, Italy*
[d] *Department of Computer Science & Engineering, University of Washington, Seattle, WA 98195-2350, USA*

Abstract

We address the design of algorithms for multicores that are oblivious to machine parameters. We propose HM, a multicore model consisting of a parallel shared-memory machine with hierarchical multi-level caching, and we introduce a multicore-oblivious approach to algorithms and schedulers for HM. A multicore-oblivious algorithm is specified with no mention of any machine parameters, such as the number of cores, number of cache levels, cache sizes and block lengths. However, it is equipped with a small set of instructions that can be used to provide hints to the run-time scheduler on how to schedule parallel tasks. We present efficient multicore-oblivious algorithms for several fundamental problems including matrix transposition, FFT, sorting, the Gaussian Elimination Paradigm, list ranking, and connected components. The notion of a multicore-oblivious algorithm is complementary to that of a network-oblivious algorithm, introduced by Bilardi et al. (2007) for parallel distributed-memory machines where processors communicate point-to-point. We show that several of our multicore-oblivious algorithms translate into efficient network-oblivious algorithms, adding to the body of known efficient network-oblivious algorithms.

Keywords: multicore, cache, network, oblivious algorithm, Gaussian elimination paradigm, list ranking

*Corresponding author: Department of Information Engineering, University of Padova, Via Gradenigo 6/B, 35131 Padova, Italy, (email) silvest1@dei.unipd.it, +39 049 8277954 (phone), +39 049 8277799 (fax).

Email addresses: rezaul@cs.stonybrook.edu (Rezaul Alam Chowdhury), vlr@cs.utexas.edu (Vijaya Ramachandran), silvest1@dei.unipd.it (Francesco Silvestri), blakeley@cs.washington.edu (Brandon Blakeley)

1. Introduction

The cache-oblivious framework [1] has provided a convenient and general-purpose approach to developing algorithms that perform efficiently on a microprocessor with a single core and a cache hierarchy (see [2, 3] and the references therein). A noteworthy feature of such algorithms is that they incorporate no machine parameters in their code, and yet are shown to perform efficiently at all levels of the cache hierarchy. The notion of *network-oblivious* algorithm was introduced in [4]: a network-oblivious algorithm is designed for parallel distributed-memory machines where processors communicate in a point-to-point fashion, and it runs efficiently even though it includes no machine parameters, such as the number of processors or the network topology, in its specification.

The oblivious approaches in [1, 4] are not suitable for modern multicore platforms as multicores represent a paradigm shift in general-purpose computing away from the von Neumann model to a collection of cores on a chip communicating through a cache hierarchy under a shared memory. But the oblivious approach is of particular relevance to multicores since multicores with a wide range of machine parameters have already become the default desktop configuration, resulting in the widespread need for efficient, *portable* code for them.

Efficient algorithms for multicores must address both *caching issues* and shared-memory *parallelism*, and in recent years, a number of models, algorithms and schedulers for multicores have been proposed. In its simplest form, a multicore is modeled as a collection of processing elements or *cores* sharing an arbitrarily large main memory containing all data and featuring one level of cache which could be either private (e.g., [5, 6, 7, 8]) or shared among all the cores (e.g., [9, 7]). Since multicores are evolving towards a hierarchy of caches, a 3-level model was introduced in [10], which consists of a collection of cores, each with a private L_1 cache, sharing an arbitrarily large main memory through a shared L_2 cache; in [11] a multi-level version of this model is briefly introduced. The Multi-BSP in [12] is a hierarchical shared-memory model that uses latency and gap parameters in a bulk synchronous manner. Most of the multicore algorithms in the literature are resource-aware, that is, they make use of some machine parameters in their specifications. However, as has already been demonstrated by cache- and network-oblivious approaches, algorithms that do not use knowledge of resource parameters offer advantages of simplicity and portability, which has motivated us to consider multicore-oblivious algorithms in this paper.

Since the conference version of this work [13], other papers have studied parallel algorithms oblivious to cache or processor organizations. The work [14] introduces a parallel version of the cache-oblivious framework in [1], named the Parallel Cache-Oblivious model, and describes a scheduler for oblivious irregular computations. On the other hand, the papers [15, 16, 17] study efficient algorithms that are designed to work with schedulers that are oblivious of the cache hierarchy; these papers derive *resource-oblivious* algorithms that are analyzed to run efficiently when scheduled by any scheduler that generates a small number of parallel tasks at caches at each level of the cache hierarchy during run-time.

1.1. Our Results

First, and of independent interest, we present a *hierarchical multi-level caching model* (*HM*) for multicores which was briefly described in [11] and extends the 3-level multicore caching model in [10]. The HM model consists of a collection of cores sharing an arbitrarily large main memory through a hierarchy of caches of finite but increasing sizes that are successively shared by larger groups of cores. Parallelism is specified by parallel **for** loops and forking and joining through recursive calls, and is asynchronous otherwise (in contrast to the bulk-synchronous nature of Multi-BSP [12]).

Next we introduce the notion of *multicore-oblivious* algorithms for the HM model, which are algorithms that make no mention of the number of cores, number of cache levels, cache size and block transfer length of each level in the multicore. However, for improved performance, a multicore-oblivious algorithm is allowed to provide advice or hints to the run-time scheduler through a small set of instructions on how to schedule the parallel tasks it spawns. We illustrate our framework by providing efficient/optimal multicore-oblivious algorithms for several fundamental problems, including matrix transposition, sorting, FFT, the Gaussian Elimination Paradigm (GEP) [7], list ranking, connected components, and other graph problems.

We observe that our notion of multicore-oblivious algorithms is complementary to that of network-oblivious algorithms [4]: the former is defined in a shared-memory model, while the latter is defined in a distributed-memory model with point-to-point communications. Nevertheless, in many cases, a problem can be solved by multicore-oblivious and network-oblivious algorithms through similar strategies, and in fact, our multicore-oblivious algorithms for matrix transposition and FFT are adapted from their network-oblivious counterparts in [4]. We further reinforce this connection by deriving efficient network-oblivious algorithms for GEP, list ranking, connected components, and other graph problems by adapting our multicore-oblivious algorithms. These results enrich the body of known efficient network-oblivious algorithms and raise hopes for a unified notion of obliviousness in parallel computation. A summary of our key results appears in Table 3 at the end of the paper.

1.2. Paper Organization

In Section 2 we present the HM model. In Section 3 we describe three types of scheduler hints to support multicore-obliviousness, and illustrate them with multicore-oblivious algorithms for matrix transposition, sparse matrix dense vector multiplication, sorting, and FFT. In Section 4 we review the network-oblivious framework. We present efficient multicore-oblivious and network-oblivious algorithms for important applications of GEP in Section 5, and for list ranking, connected components, and other graph problems in Section 6. Finally, in Section 7 we summarize our results and discuss some open problems.

h	number of cache level (including the main memory)
p	number of cores
q_i	number of caches at level-i
C_i	cache size at level-i
B_i	block length at level-i
p_i	number of level-$(i-1)$ caches that share the same level-i cache
p'_i	number of cores that share the same level-i cache

Table 1: Most important parameters used in the paper.

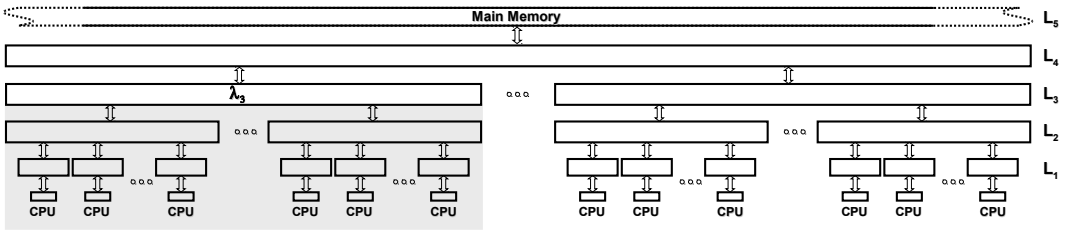

Figure 1: The HM model illustrated for $h = 5$. The dark area below the level-3 cache λ_3 covers the caches and cores in the shadow of λ_3.

2. HM Model

The *hierarchical multi-level caching multicore* (HM) model with h levels consists of a collection of p cores P_i, $1 \leq i \leq p$, with an arbitrarily large shared-memory and $h-1$ levels of caches of finite but increasing sizes. For $1 \leq i \leq h$, we denote the i-th level of the memory hierarchy with *level-i* or L_i: at level-h we have the shared-memory, while at level-i, for each $1 \leq i < h$, there are q_i caches, each one with size C_i and block length B_i and shared by $p'_i = p/q_i$ cores. With a slight abuse of notation, we will sometimes refer to the main memory also as the level-h cache and set $C_h = +\infty$ and $q_h = 1$ for convenience. For $1 < i \leq h$, the number of successive level-$(i-1)$ caches that share a given level-i cache is p_i and we set $p_0 = p'_0$. Clearly, we have $p'_i = p/q_i = \prod_{j=1}^{i+1} p_j$. An arbitrary cache at level-i is denoted by λ_i, and the *shadow of* λ_i is defined as the set including the p'_i cores that share λ_i as their level-i cache, and all level-j caches, for each $1 \leq j < i$, that lie between those cores and λ_i. The most important terms used in the paper are summarized in Table 1, while figure 1 shows the model for $h = 5$.

Similar to the sequential cache hierarchy, the memory hierarchy in the HM model satisfies the *inclusion property* meaning that for $1 < i \leq h$, each level-i cache stores all elements present in all level-$(i-1)$ caches under its shadow. Every cache uses an *optimal* cache replacement policy. The HM model allows *concurrent reads* and, when a memory word x in cache λ_{i+1} is read by two or more cores not sharing the same level-i cache, the block of size B_i containing x is replicated and stored in the respective level-i caches. Parallelism in an HM algorithm is specified by parallel **for** loops, denoted with **pfor**, and by forking and joining subtasks.

We now define the complexity measures we use for evaluating an HM algorithm $\mathcal{A}$, assum-

ing all cores run at the same rate. The *level-i cache complexity* of $\mathcal{A}$, with $1 \leq i < h$, is the maximum number of block transfers into and out of any single cache at level-i. The *parallel time complexity* of $\mathcal{A}$ is the maximum number of operations performed by a core ignoring delays due to memory accesses (i.e., by ignoring the memory hierarchy). The *critical path* of $\mathcal{A}$ is defined as the length of the longest sequence of dependent operations, and matches with the notion of critical path in the PRAM model.

In the model used in this paper we make the following assumptions which typically hold for real multicore machines. We set $p_1 = 1$, that is, each core has a private cache at the first level, and $p_h = 1$, that is, the topmost two levels $h-1$ and h represent a possible sequential cache hierarchy at the highest level. We also require that, for each $1 \leq i < h$, $C_i \geq B_i^2$ (i.e., each cache is *tall* [1]), $B_{i-1} \leq B_i \leq p_i B_{i-1}$ and $C_i \geq c_i \cdot p_i \cdot C_{i-1}$ for a suitable value $c_i \geq 1$ which, if not stated otherwise, is a constant. The constraints on relative cache sizes imply the following upper bound on the number of cores: $p = \prod_{i=1}^{h} p_i = \prod_{i=2}^{h-1} p_i \leq K \cdot C_{h-1}/C_1$, where $K = \prod_{i=2}^{h-1} (1/c_i)$.

We have chosen to stay with a simple definition of the HM model, leaving unspecified several parameters of a multicore's caching system that are not salient to the algorithms we present in this paper. For the algorithms we consider in this paper, we do not need cache coherence protocols since all parallel write operations performed by our algorithms are always on disjoint sets of data, and hence coherence is never invoked. Furthermore, for many of the results in this paper, the optimal cache replacement policy can be replaced with a simple generalization of LRU to multi-level caches, where the block evicted from a cache λ_i to make room for a new block is the one with entries that were least recently used by the cores in the shadow of λ_i, and that block is also removed from all caches under λ_i's shadow.

The multicore model in [10] is the HM model with $h = 3$. As noted there for the 3-level multicore model, there is an inherent tension between cache-efficient scheduling for private level-1 caches, and that for a single shared level-2 cache: for the former, a schedule that gives large independent tasks to the different cores is typically cache-efficient, while for the latter, a fine-grained schedule where all cores work on the portion of the data present in the shared level-1 cache is effective. This tension is magnified when the number of levels increases. Multicore algorithms for sorting were given in [5, 18, 17], and the algorithms in [18, 17] claim fairly good performance on a multi-level cache hierarchy. However, this claimed good performance is still a factor of p_i' worse than the best possible for each cache level-i, and is obtained by analyzing a simple assignment to each core of a proportionate slice of the level-i cache above it, for each i. In contrast, the multicore-oblivious algorithms we present fully exploit the higher level caches, rather than just a small fraction of the best possible.

3. Multicore-Oblivious Algorithms on the HM Model

For the effective use of multicores, we introduce the notion of efficient *multicore-oblivious* algorithms, that is, algorithms that do not use specific values for multicore parameters such

as number of cores and cache levels, cache sizes and block lengths, etc., and yet perform efficiently across a wide variety of multicores.

To address this challenge, we propose some simple enhancements to HM algorithms in the form of instructions or *hints* in the algorithm that are meant to be interpreted and used by the run-time scheduler to decide how to schedule parallel tasks generated during execution. Our initiative to introduce these special instructions for the run-time scheduler is in the spirit of the recent trend towards *multiresolution* languages [19], where the basic programming language is enhanced with constructs that can be used by the savvy programmer to enhance performance. It appears that some mechanisms of this type are needed in order to extract efficient performance of algorithms on multicores with multiple levels of caches.

The multicore-oblivious algorithms introduced in this paper use two main types of scheduling hints, namely, CGC (coarse-grained contiguous) and SB (space-bound), and a third one that combines these two (CGC⇒SB). Among these, CGC is useful for computations involving parallel **pfor** loops, while SB and CGC⇒SB are useful for algorithms that recursively spawn parallel tasks. These scheduling hints do not make use of the HM parameters, and thus are oblivious of those parameters.

The remaining part of the section is organized as follows: Section 3.1 introduces some notation; then Sections 3.2, 3.3 and 3.4 describe the CGC, SB and CGC⇒SB schedulings, respectively.

3.1. Preliminaries

The *space bound* $s(\tau)$ of a task τ is an upper bound on the space required by τ, and includes the space used by τ's subtasks. The *aggregate space bound* of two tasks τ_1 and τ_2 is $s(\tau_1)$ if τ_2 is a descendant subtask of τ_1, $s(\tau_2)$ if τ_1 is a descendant subtask of τ_2, and $s(\tau_1) + s(\tau_2)$ otherwise.

A task τ is said to be *anchored* to a level-i cache λ_i, with $1 \leq i \leq h$, if λ_i is large enough to meet the space requirement of τ (i.e., $s(\tau) \leq C_i$), and τ and all its subtasks are executed by cores in the shadow of λ_i. A task can be anchored to only one cache at any given time, though more than one cache can satisfy the space requirement. In contrast, many tasks can be anchored to a a given cache λ_i assuming that the aggregate space bound does not exceed C_i.

Each cache λ_i, for each $1 \leq i \leq h$, is associated with a queue $\mathcal{Q}(\lambda_i)$ containing tasks waiting to be anchored to λ_i. When a task anchored to λ_i ends or a new task is inserted into $\mathcal{Q}(\lambda_i)$, the run-time support dequeues tasks from $\mathcal{Q}(\lambda_i)$ and anchors them to λ_i until the aggregate space bound is satisfied or the queue is empty. (Note that an hint may apply different dequeue policies to the queue.) A task anchored to λ_i can be executed by any core in the shadow of λ_i unless specified otherwise by the scheduling hints.

3.2. Coarse-Grained Contiguous Scheduling

The *coarse-grained contiguous (CGC) scheduling* distributes an ordered collection of parallel fine-grained subtasks in contiguous chunks across a sequence of contiguous cores. In

MO-MT(A, n)
Input: an $n \times n$ matrix A in row-major order.
Output: the transpose A^T in row-major order.
1: **[CGC] pfor** $0 \leq z \leq n^2 - 1$ **do**
2: $(i, j) = \beta(z)$;
3: $A^T[j, i] := A[i, j]$;

Figure 2: MO-MT: a multicore-oblivious matrix transposition algorithm.

particular, CGC is used to decompose a parallel **pfor** loop acting on a contiguous chunk of data into segments, each of which is executed by a core in parallel to others.[1]

Suppose τ is a task anchored to a level-k cache λ_k, with $1 \leq k \leq h$, and forks a collection of m parallel subtasks, indexed by a variable t running from 1 to m. Then, CGC decomposes the values taken by t into p'_k contiguous segments of the same or almost the same length, and tasks in the j-th such segment are inserted into queue $\mathcal{Q}(\lambda_1^j)$, where λ_1^j denotes the level-1 cache of the j-th core in the shadow of λ_k, for each $1 \leq j \leq p'_k$. The scheduler ensures that each segment, but one, contains at least B_1 subtasks, in order to respect cache block boundaries and reduce cache complexity by avoiding ping-ponging of blocks among caches. In the next two sections, we exercise the CGC scheduling by deriving multicore-oblivious algorithms for matrix transposition and prefix sum.

3.2.1. Matrix Transposition

Figure 2 gives a multicore-oblivious algorithm for transposing an $n \times n$ matrix A based on the network-oblivious algorithm in [4] and scheduled with CGC. We denote with $A[i, j]$, $0 \leq i, j < n$, the entry of A in the i-th row and the j-th column. For the sake of simplicity, we assume n to be an even power of two. Intuitively, the algorithm scans entries in A according to the Z-Morton layout, and put them in their final positions in A^T. We denote by $\beta(z)$, $0 \leq z < n^2$, the ordered pair of integers (i, j) whose *bitwise interleaving* gives the binary representations of z, that is, the entry of A in the z-th position according to the Z-Morton layout. More specifically, let $(z)_2 = z_{2r-1}z_{2r-2} \cdots z_1 z_0$ be the binary representations of z, with $r = \log n$; then i and j are the integers whose binary representations are obtained by extracting from z the bits in even and odd locations, respectively, i.e., $(i)_2 = z_{2r-2} \cdots z_2 z_0$ and $(j)_2 = z_{2r-1} \cdots z_3 z_1$. We assume that $\beta(z)$ is computed in constant time by the hardware.

Theorem 1. MO-MT *correctly transposes the $n \times n$ matrix A in $O\left(n^2/p + B_1\right)$ parallel time and $O\left(n^2/(q_i B_i) + p'_i B_1/B_i\right)$ cache misses at each level-i cache, for all $1 \leq i \leq h-1$, assuming $c_i = \Theta\left((B_i/B_{i-1})^2\right)$. This result is optimal when $n = \Omega\left(\sqrt{B_1 p}\right)$.*

Proof. The algorithm is correct since each entry of A is touched and moved to the correct position in A^T. The parallel time of the algorithm follows since each core performs $O\left(n^2/p + B_1\right)$

[1] With a slight abuse of notation, in the description of CGC hint, we use the notions of anchored task and of queue, although CGC does not require task space bounds. Formal definitions of these terms which do not make use of space bounds are straightforward.

iterations of the loop. We will now describe a cache replacement strategy that guarantees $O\left(n^2/(q_i B_i) + p'_i B_1/B_i\right)$ cache misses at each level-i cache. Since the HM model uses an optimal cache replacement policy, it will never incur more cache misses than those incurred by our strategy. By hypothesis we have $c_{l+1} = \Theta\left(B_{l+1}/B_l\right)^2$, for any $1 \leq l < h-1$, and $C_1 = \Omega\left(B_1^2\right)$: then, it follows by induction on l that $C_l = \Omega\left(p'_l B_l^2\right)$. So, let us assume that in each λ_l our replacement policy reserves $\Theta\left(B_l\right)$ cache blocks for each core in the shadow of λ_l. We now bound the number of misses at level-l, for each $1 \leq l < h$, incurred by a core while performing b_l consecutive iterations of the loop, where b_l is the biggest even power of two not larger than B_l^2. In the b_l iterations the z value assumes $O\left(1\right)$ configurations in the $2\log n - \log b_l$ most significant bits, and then the algorithm reads b_l entries of A contained into $O\left(\sqrt{b_l}\right)$ blocks of size B_l. Similarly for A^T. Hence, a core performing b_l iterations incurs $O\left(B_l\right)$ misses at level-l by exploiting the $\Theta\left(B_l\right)$ reserved cache blocks. Since each core performs $\max\{n^2/p, B_1\}$ iterations and $p'_l = p/q_l$ cores share the same level-l cache, the cache complexity at level l is $O\left((n^2/p + B_1)p'_l/B_l\right) = O\left(n^2/(q_l B_l) + B_1 p'_l/B_l\right)$. Simple lower bounds for time and cache complexities are $\Omega\left(n^2/p\right)$ and $\Omega\left(n^2/(q_l B_l)\right)$, respectively, and hence the algorithm is optimal when $n^2 = \Omega\left(B_1 p\right)$. □

We observe that the assumptions on the cache hierarchy required in Theorem 1 (e.g., $p \leq n^2/B_1$ and $c_i = \Theta\left((B_i/B_{i-1})^2\right)$) are in general satisfied by common multicores [20]. In the following Section 3.3 we describe another multicore-oblivious algorithm for matrix transposition which uses the SB hint and is built on the classical cache-oblivious algorithm.

3.2.2. Prefix sums

We now describe an optimal multicore-oblivious algorithm for computing the prefix sums of n values. Consider a sequence $(x_0, x_1, \ldots, x_{n-1})$ of n elements taken from a set S with an associative binary operation $*$. We define the i-th partial sum s_i of such a sequence to be $s_i = x_0 * x_1 * \cdots * x_i$, for each $0 \leq i < n$. For simplicity, we assume n to be a power of two. We show that the recursive algorithm in [21] exhibits optimal performance using the CGC scheduling. The algorithm is described in Figure 3, where we suppose n to be a power of two for the sake of simplicity: at a high level, elements are added in pairs, the resulting problem is recursively solved, and then this solution is extended to the remaining summands.

Theorem 2. MO-PS *correctly computes the prefix sums of n entries in $O\left(n/p + B_1 \log p\right)$ parallel time and $O\left(n/(q_i B_i) + (p_i B_1/B_i)\log p\right)$ cache misses at each level-i cache, for all $1 \leq i \leq h-1$. This result is optimal when $n = \Omega\left(B_1 p \log p\right)$.*

Proof. The correctness of this algorithm can be proved by induction as shown in [22]. We now analyze its performance. The CGC scheduler assigns contiguous blocks of elements to contiguous processors and requires that at most one core is given less than B_1 units of data, even if this requires some processors to be idle. In Step 2 and Steps 4–7, we have $O\left(n\right)$ elementary operations to distribute across the $\min(p, \lfloor n/B_1 \rfloor)$ consecutive cores which the CGC scheduling assigns tasks to. As each recursive stage decreases the size of the input by

MO-PS($x_0, x_1, \ldots, x_{n-1}$)
Input: sequence $(x_0, x_1, \ldots, x_{n-1})$ of summands with associative binary operator $*$.
Output: sequence $(s_0, s_1, \ldots, s_{n-1})$ of partial sums.

1: **if** $n = 1$ **then** $s_0 := x_0$; **return**; (s_0) **endif**
2: **[CGC] pfor** $0 \leq i \leq (n/2 - 1)$ **do** $y_i := x_{2i} * x_{2i+1}$ **end pfor**
3: $(z_0, z_1, \ldots, z_{n/2-1}) :=$ MO-PS$(y_0, y_1, \ldots, y_{n/2-1})$
4: **[CGC] pfor** $0 \leq i \leq n - 1$
5: **if** i is even **then** $s_i := z_{i/2-1} * x_i$
6: **else** $s_i := z_{(i-1)/2}$ **endif**
7: **end pfor**
8: **return** $(s_0, s_1, \ldots, s_{n-1})$

Figure 3: MO-PS: a multicore-oblivious algorithm for prefix sums.

a factor of two, the number of processors the CGC scheduler assigns tasks to decreases by a factor of two as well whenever $n \leq pB_1$. Then, the running time $T(n, p)$ is upper bounded by the following recurrence relation:

$$T(n,p) = \begin{cases} T(n/2,p) + O(n/p) & n > pB_1 \\ T(n/2,p/2) + O(n/p) & B_1 < n \leq pB_1 \\ O(n) & n \leq B_1 \end{cases}$$

This recurrence solves to $O(n/p + B_1 \log p)$, which is optimal as soon as $n = \Omega(B_1 p \log p)$.

Next, we analyze the level-i cache complexity of this algorithm. Each recursive call incurs $O(n/(q_i B_i))$ misses at level-i when $n > pB_1$. When $p'_i B_1 < n \leq pB_1$, each level-i cache receives $O(p'_i B_1)$ entries and thus there are $O(p'_i B_1/B_i)$ cache misses at level-i. Finally, when $n \leq p'_i B_1$, the problem can be contained in a level-i cache and there are $O(p'_i B_1/B_i)$ misses. Then, the cache complexity $Q_i(n, p)$ at level-i is upper bounded by the following recurrence relation:

$$Q_i(n,p) = \begin{cases} Q_i(n/2,p) + O(n/(q_i B_i)) & n > pB_1 \\ Q_i(n/2,p/2) + O(p'_i B_1/B_i) & p'_i B_1 < n \leq pB_1 \\ O(p'_i B_1/B_i) & n \leq p'_i B_1 \end{cases}$$

This recurrence solves to $O(n/(q_i B_i) + (p'_i B_1/B_i) \log q_i)$, which is optimal when $n = \Omega(B_1 p \log p)$. □

We note that these complexity bounds also apply to every algorithm which recursively solves a problem by a scan of the data at each level of recursion and then solving a geometrically smaller subproblem all the way down to a constant size.

3.3. Space-Bound Scheduling

The *space-bound (SB) scheduling* requires an algorithm to supply an upper bound on the space used by each task that is forked during the computation. In particular, the SB scheduling is applied to recursively forking tasks where a constant number of tasks are generated at

MO-MT2(A, B, n)
Input: $n \times n$ matrices A and B in row-major order.
Output: the transpose A^T stored in row-major order stored in B.
Space Bound: $2n^2$.
1: **if** $n = 1$ **then** $B = A$; **return**; **endif**
2: Partition A (B, resp.) into four quadrants $A_{i,j}$ ($B_{i,j}$, resp.), $i, j \in \{1, 2\}$;
3: **[SB] pfor** $1 \leq i, j \leq 2$ **do** MO-MT2$(A_{i,j}, B_{j,i}, n/2)$;

Figure 4: MO-MT2: a multicore-oblivious matrix transposition algorithm using the SB hint.

each fork, each with a space bound that is a constant factor smaller than that of the forking task. Intuitively, if all subtasks generated by a task τ anchored to a level-i cache λ are anchored only to caches under the shadow of λ, then the total number of level-i cache misses incurred by τ can be upper bounded by those needed to read in the initial input to λ once and write out the final output from λ once.

As already mentioned in Section 3.1, each level-k cache λ_k maintains a queue $\mathcal{Q}(\lambda_k)$ for tasks with space bound in $(C_{k-1}, C_k]$ which are to be executed under the shadow of λ_k. When the current task assigned to λ_k completes, a task τ from $\mathcal{Q}(\lambda_k)$ is extracted and executed while anchored at λ_k. When τ forks a task τ' and $s(\tau') \leq C_{k-1}$, τ' is assigned to the least loaded cache under the shadow of λ_k at the smallest level $j < k$ such that $s(\tau') \leq C_j$, otherwise it is inserted into $\mathcal{Q}(\lambda_k)$.

In the subsequent section we apply the SB scheduler to the recursive cache-oblivious algorithm for matrix transposition given in [1] and to the aforementioned recursive algorithm for prefix sums. Furthermore, in Section 5, we show that the SB hint can be successfully applied to derive a multicore-oblivious algorithm for the GEP paradigm.

3.3.1. Matrix Transposition and Prefix Sums with SB

In Section 3.2.1 we showed how to implement matrix transposition using the CGC hint. For the sake of completeness we present a multicore-oblivious algorithm, named MO-MT2, which uses the SB scheduling hint and is based on the cache-oblivious algorithm in [1]. Figure 4 provides the pseudocode of the algorithm. We have the following theorem.[2]

Theorem 3. MO-MT2 *correctly transposes the $n \times n$ matrix A in $O\left(n^2/p + \log n\right)$ parallel time and $O\left(n^2/(q_i B_i)\right)$ cache misses at each level-i cache, for all $1 \leq i \leq h-1$, when $C_j \geq c \cdot p_j \cdot C_{j-1}$ for some constant $c > 1$ and $2 \leq j \leq h-1$, and $C_1 \geq \max_{j=1}^{h-1}\{p_j\}$.*

Proof. Correctness follows from [1]. Let r_i be the smallest integer such that a subproblem of size $n/2^{r_i} \times n/2^{r_i}$ fits into a level-i cache, for each $1 \leq i \leq h-1$. Since the space bound is $2n^2$ for an $n \times n$ input, we have $r_i = \lceil \frac{1}{2} \log(2n^2/C_i) \rceil$. There are $\Theta\left(4^{r_i}\right)$ subproblems of size $n/2^{r_i} \times n/2^{r_i}$ each, and an equal number of them will be anchored to each of the q_i

[2] In the paper we denote the quadrants of a matrix A as: A_{11} (top-left), A_{12} (top-right), A_{21} (bottom-left), A_{22} (bottom-right).

MO-PS2($x_0, x_1, \ldots, x_{n-1}$)
Input: sequence $(x_0, x_1, \ldots, x_{n-1})$ of summands with associative binary operator $*$.
Output: sequence $(s_0, s_1, \ldots, s_{n-1})$ of partial sums.
Space Bound: $2n$.

1: **if** $n = 1$ **then** $s_0 := x_0$; **return** (s_0) **endif**
2: **parallel spawn**
3: **[SB]** $(s_0, s_1, \ldots, s_{n/2-1})$:= MO–PS2$(x_0, x_1, \ldots, x_{n/2-1})$
4: **[SB]** $(s_{n/2}, s_{n/2+1}, \ldots, s_{n-1})$:= MO–PS2$(x_{n/2}, x_{n/2+1}, \ldots, x_{n-1})$
5: **end parallel spawn**
6: **[SB]** MO–D&C–ADD$((s_{n/2}, s_{n/2+1}, \ldots, s_{n-1}), s_{n/2-1})$[3]
7: **return** $(s_0, s_1, \ldots, s_{n-1})$

Figure 5: MO-PS2: a multicore-oblivious algorithm for prefix sums using SB hint.

level-i caches. When such a subproblem is anchored to a level-i ($i > 1$) cache it is decomposed into $\Theta\left(4^{r_{i-1}}/4^{r_i}\right)$ subproblems of size $n/2^{r_{i-1}} \times n/2^{r_{i-1}}$ each in time $\Theta\left(4^{r_{i-1}}/4^{r_i}\right)$. Hence, the total parallel time spent in decomposing level-i subproblems into level-$(i-1)$ subproblems is $\Theta\left(\left(4^{r_i}/q_i\right) \times \left(4^{r_{i-1}}/4^{r_i}\right)\right) = \Theta\left(4^{r_{i-1}}/q_i\right) = \Theta\left(n^2/\left(q_i C_{i-1}\right)\right) = O\left(n^2/\left(pc^{i-2}\right)\right)$ as $q_i C_{i-1} = \left(p/\prod_{j=2}^{j=i} p_j\right) C_{i-1} \geq \left(p/p_i\right) c^{i-2} C_1 \geq pc^{i-2}$. Finally, there are 4^{r_1} subproblems of size $n/2^{r_1} \times n/2^{r_1}$ each. Each such subproblem is solved by a single processing core in $\Theta\left(n^2/4^{r_1}\right)$ sequential time, and each core solves an equal number of them. Hence, adding the $\Theta(\log n)$ critical pathlength of MO-MT2 due to the size of the input, the overall parallel running time of MO-MT2 is $O\left(\sum_{i=2}^{h}\left(n^2/\left(pc^{i-2}\right)\right) + \left(4^{r_1}/p\right) \times \left(n^2/4^{r_1}\right) + \log n\right) = O\left(n^2/p + \log n\right)$.

We now upper bound the number of cache misses incurred by MO-MT2 at a level-i cache λ_i, for $1 \leq i \leq h-1$. In order to obtain this bound we only need to consider the cache misses incurred by the tasks anchored to that cache. Only subproblems of size $n/2^{r_i} \times n/2^{r_i}$ are anchored to λ_i, and each such subproblem incurs $O\left(C_i/B_i\right)$ cache misses at λ_i. Since $\Theta\left(4^{r^i}/q_i\right) = \Theta\left(n^2/\left(q_i C_i\right)\right)$ such subproblems are anchored to λ_i over the execution of the entire algorithm, the number of cache misses incurred by MO-MT2 at λ_i is $O\left(\left(n^2/\left(q_i C_i\right)\left(C_i/B_i\right)\right) = O\left(n^2/\left(q_i B_i\right)\right)\right.$. □

Section 3.2.2 showed an implementation of parallel prefix sums using the CGC hint. Figure 5 shows how to find prefix sums multicore-obliviously using a classic parallel divide-and-conquer algorithm with SB hint. The following theorem gives the performance bounds of the algorithm. The proof is somewhat similar to that of Theorem 3, and hence has been omitted.

Theorem 4. MO-PS2 *correctly computes the prefix sums of n entries in $O\left(n/p + \log n\right)$ parallel time and $O\left(n/(q_i B_i)\right)$ cache misses at each level-i cache, for all $1 \leq i \leq h-1$, when $C_j \geq c \cdot p_j \cdot C_{j-1}$ for some constant $c > 1$ and $2 \leq j \leq h-1$, and $C_1 \geq \max_{j=1}^{h-1}\{p_j\}$.*

[3]In the pseudocode MO–D&C–ADD$((v_0, \ldots v_n), v)$ is a multicore-oblivious algorithm that adds the term v to each term of the sequence $(v_0, \ldots v_n)$ by recursively subdividing the sequence. We omit its simple pseudocode.

3.4. CGC on SB Scheduling

The CGC⇒SB scheduling is useful in algorithms that recursively fork a large number of parallel tasks. Informally, under CGC⇒SB, an ordered collection of subtasks forked from a task are distributed in a CGC manner across caches at a suitable lower level where the cache size is large enough to accommodate each subtask's space bound and at the same time, the parallelism is fully exploited. We now specify the mechanism of this scheduler.

Let τ be a task anchored to a level-k cache λ_k, with $1 \leq k \leq h$, which recursively spawns a collection of m parallel subtasks, indexed by a variable t running from 1 to m. For simplicity of exposition let us assume that all generated subtasks have the same space bound σ. The CGC⇒SB scheduler finds the smallest level-i with $C_i \geq \sigma$. If $i = k$ then all tasks are anchored to λ_k. Suppose $i < k$. The hint then decomposes the values taken by t into $p_{k,i} = \Pi_{j=i+1}^{k} p_j$ contiguous segments such that each segment uses at least B_i space [4]. Then, tasks in the j-th such segment are inserted into the task queue $\mathcal{Q}(\lambda_i^j)$, where λ_i^j denotes the j-th level-i cache in the shadow of λ_k, for each $1 \leq j \leq p_{k,i}$.

We remark that SB is more concerned with load balancing while CGC⇒SB is more about retaining cache locality: indeed, CGC⇒SB assigns consecutive numbered subtasks, which are unlikely to access data stored in relatively remote locations in memory, to consecutive caches; on the other hand, the SB scheduling may assign consecutive subtasks to nonconsecutive caches, reducing cache locality. Furthermore, differently from CGC, CGC⇒SB may anchor subtasks to level-i caches with $i > 1$. We note that CGC and SB can be viewed as special cases of CGC⇒SB, and algorithms scheduled with CGC and SB can be executed under CGC⇒SB without degrading their asymptotic performance.

Below, we apply the CGC⇒SB scheduling for deriving multicore-algorithms for the Fast Fourier Transform, sorting and sparse-matrix dense-vector multiplication.

3.4.1. Fast Fourier Transform (FFT)

The *discrete Fourier transform* (DFT) of a vector X of n complex numbers is given by another complex vector Y of the same length, where $Y[i] = \sum_{0 \leq j < n} X[j] \cdot \omega_n^{-ij}$ for $0 \leq i < n$, and $\omega_n = e^{2\pi\sqrt{-1}/n}$. In Figure 6, we present MO-FFT, obtained by adapting the cache-oblivious FFT algorithm in [1] to the HM model. MO-FFT can also be viewed as an adaptation of the network-oblivious FFT algorithm given in [4]. We use two types of scheduling in MO-FFT: CGC and CGC⇒SB. The following theorem gives the performance bounds of MO-FFT in the HM model.

Theorem 5. MO-FFT *correctly computes the FFT of n values in $O((n/p)\log n)$ parallel time and $O\left((n/(q_i B_i))\log_{C_i} n\right)$ cache misses at each level-i cache, for all $1 \leq i \leq h-1$, when $n > C_{h-1}$, $C_i > 12{p'_{i+1}}^2$ and $c_{i+1} = \Theta\left((B_{i+1}/B_i)^2\right)$.*

[4] Since $B_j \leq p_j B_{j-1}$, if CGC⇒SB ensures at least B_i space usage at each level-i cache under the shadow of λ_k, then for $i < j < k$, at least B_j space usage is ensured at every level-j cache under λ_k's shadow.

MO-FFT(X, n)
Input: A vector X of length $n = 2^k$ for some integer $k \geq 0$.
Output: In-place FFT of X.
Space Bound: $S(n) = 3n$.

1: **if** n is a small constant **then** compute FFT using the direct formula and **return**.
2: Let $n_1 = 2^{\lceil \frac{k}{2} \rceil}$ and $n_2 = 2^{\lfloor \frac{k}{2} \rfloor}$ (observe that $n_1 \in \{n_2, 2n_2\}$), and let A be an $n_1 \times n_2$ matrix stored in row-major order.
3: **[CGC] pfor** $0 \leq i < n_1, 0 \leq j < n_2$ **do** $A[i, j] := X[i \cdot n_2 + j]$
4: **[CGC]** MO-MT(A, n_1). (When $n_1 = 2n_2$, the matrix is computed by splitting the matrix into two square matrices and then invoking MO-MT twice.)
5: **[CGC⇒SB] pfor** $0 \leq i < n_2$ **do** MO-FFT$(A[i, 0 \ldots (n_1 - 1)], n_1)$
6: **[CGC]** Multiply the n entries of A by appropriate twiddle factors
7: **[CGC]** MO-MT(A, n_1)
8: **[CGC⇒SB] pfor** $0 \leq i < n_1$ **do** MO-FFT$(A[i, 0 \ldots (n_2 - 1)], n_2)$
9: **[CGC]** MO-MT(A, n_1)
10: **[CGC]** Copy the n entries of A into X

Figure 6: MO-FFT: multicore-oblivious in-place FFT.

Proof. We first upper bound the parallel time. Consider a task MO-FFT(X', n') generated during the recursive evaluation of MO-FFT(X, n), where $n' = 2^{k'}$ for some integer $k' > 0$. Suppose this task is anchored to a level-i cache λ_i. Then $C_i \geq S(n') > C_{i-1} \Rightarrow 3n' > 12{p'_i}^2 \Rightarrow \min\{n'_1, n'_2\} > p'_i$, where $n'_1 = 2^{\lceil \frac{k'}{2} \rceil}$ and $n'_2 = 2^{\lfloor \frac{k'}{2} \rfloor}$. Hence, MO-FFT$(X', n')$ will generate enough subtasks to keep all cores under λ_i busy, and assuming h to be a constant the work performed by any two cores in the shadow of λ_i will be within a constant factor of each other[5]. Consequently, the parallel running time of MO-FFT(X, n) will be $O\left(T_1(n)/p + T_\infty(n)\right) = O\left((n/p)\log n\right)$, where $T_1(n) = O\left(n \log n\right)$ and $T_\infty(n) = O\left(\log n\right)$ denote the sequential time and the critical pathlength of MO-FFT, respectively. The parallel time is clearly optimal.

In order to compute the cache complexity of MO-FFT, consider any level-i cache λ_i. Cache-misses at λ_i are due to the CGC operations of tasks anchored to higher levels and to misses for reading and writing data of tasks anchored to a level-i cache. Starting with an input of size n, $\log_{C_i} n$ levels of recursion are needed until the input becomes small enough to fit into λ_i. At each of these levels $O\left(n/\left(q_i B_i\right)\right)$ cache-misses are incurred by the algorithm at λ_i, and no additional cache-misses are incurred once the data fits into the cache. Thus the total number of cache-misses at λ_i is $O\left(\left(n/\left(q_i B_i\right)\right)\log_{C_i} n\right)$. The optimality of this bound follows from a straight-forward extension of the cache-miss lower bound proved in [17] for the computation DAG of MO-FFT on a two-level multicore model to the multi-level model. □

3.4.2. Sorting

[5] More specifically, if $C_i > 12\alpha {p'_{i+1}}^2$ for some $\alpha \geq 1$, then no core will perform more than a factor of $\left(1 + \frac{1}{\sqrt{\alpha}}\right)^h$ more work than any other core.

MO-MS($\mathcal{L}, r, m, L$)
Input: A set of r sorted lists $\mathcal{L} = \{L_1, \cdots, L_r\}$,where $m = \sum_{i=1}^{r} |L_i|$, with $m \leq r^6$. For convenience we assume that all elements are distinct.
The sorting algorithm SPMS, on input $A[1..n]$, calls MO-MS($\mathcal{L}, n, n, L$), where $\mathcal{L} = \{A[1], \cdots, A[n]\}$ to compute the desired sorted sequence.
Output: The list L containing the elements in $\mathcal{L}$ in sorted order.
Space Bound: $S(m) = cm$, for a suitable constant c.

1: **if** m is a small constant **then** compute MO-MS using an $O(m \log m)$ sorting algorithm and **return**.
2: **[SB]** Partition the input merging problem $\mathcal{M} = (\mathcal{L}, r, m, L)$ into $k = O(m/r^2)$ disjoint merging problems $\mathcal{M}_1, \cdots, \mathcal{M}_k$ such that for each $\mathcal{M}_i = (\mathcal{L}_i, r_i, m_i, L_i)$: $m_i \leq r^3$, $r_i \leq r$, and all elements in $\mathcal{L}_i$ are smaller than any element in $\mathcal{M}_{i+1}$, for $1 \leq i \leq k-1$.
3: **[SB] pfor** $1 \leq i \leq k$ **do** Group the lists in $\mathcal{M}_i$ into disjoint sets of $O(\sqrt{r})$ lists. Let there be k_i such groups $\mathcal{G}_{i,j}$, $1 \leq j \leq k_i$, where the jth group has $r_{i,j}$ lists (with $r_{i,j} = O(\sqrt{r})$), and $m_{i,j}$ equals the number of elements in these r_j lists. Note that each group $\mathcal{G}_{i,j}$ is a valid input to MO-MS of size $O(\sqrt{m})$.
4: **[SB]** Perform the u-tree computation on the sizes of the sequence of groups $\mathcal{G}_{i,j}$, $1 \leq i < k$, $1 \leq j \leq k_i$, to obtain a balanced collection of MO-MS tasks $\mathcal{H}_j$, $j = 1, 2, \cdots$.
5: **[CGC⇒SB]** Call MO-MS on the sequence of tasks $\mathcal{G}_{i,j}$ by using the tasks $\mathcal{H}_j$ to achieve parallelism with the same space-bound for all subtasks. Let the resulting output sequence of sorted lists be $G_{i,j}$, $1 \leq i < k$, $1 \leq j \leq k_i$. Note that the group $\mathcal{G}_i$, consisting of the k_i sorted lists $G_{i,j}$, $1 \leq j \leq k_i$, is a valid input to MO-MS of size $O(r^3)$.
6: **[SB]** Perform the u-tree computation on the sizes of the sequence of groups $\mathcal{G}_i$, $1 \leq i \leq k$, to obtain a balanced collection of MO-MS tasks $\mathcal{H}'_j$, $j = 1, 2, \cdots$.
7: **[CGC⇒SB]** Perform MO-MS on the tasks $\mathcal{G}_i$ in parallel by using the tasks $\mathcal{H}'_j$ to achieve the same space-bound for all subtasks (similar to Step 5). Let the sequence of output sorted lists be G_i, $1 \leq i \leq k$.
8: **[CGC]** (or **[SB]**) Return the concatenation of G_1 followed by $G_2, \cdots, G_k$

Figure 7: MO-MS: multicore-oblivious multi-merge, used in the sorting algorithm SPMS.

Sample Partition Merge Sort (SPMS) is a resource-oblivious algorithm for sorting on a multicore with just private caches [17]. It is shown in [17] that on an input of length n, SPMS runs in $O(n \log n)$ sequential time with $O((n/B) \log_C n)$ cache misses, that it has critical pathlength $O(\log n \log \log n)$, a linear space bound for all tasks, and that it can be scheduled with optimal cache miss cost on a multicore with private caches. SPMS is resource-oblivious in that it does not contain machine parameters, and further, it can be scheduled efficiently (for private caches) by a variety of schedulers that also do not know the machine parameters, as long as the number of sequential task fragments scheduled across the cores is small. It is also shown in [17] that SPMS performs well even in the presence of false-sharing under an unknown block size B_1, but this is not needed for our results here.

Let the input array be $A[1..n]$, and let $\mathcal{L} = \{A[1], \cdots, A[n]\}$. SPMS sorts the array A with a call to MO-MS($\mathcal{L}, n, n, L$), where the algorithm for MO-MS is given in Figure 7. In general, MO-MS takes as input a set of r sorted lists $\mathcal{L} = \{L_1, \cdots, L_r\}$,where $m = \sum_{i=1}^{r} |L_i|$, with $m \leq r^6$. For convenience we assume that all elements are distinct.

While more complex than MO-FFT, MO-MS has the same structure, except that the CGC steps in MO-FFT that use MO-MT and other parallel computations with $O(1)$ critical

pathlength, are instead replaced by a constant number of applications of prefix sums and other *balanced parallel computations* ('BP' computations) [17]. In [17], these BP computations are formulated as binary forking computations that map naturally into SB scheduling (though many of them can also be re-formulated to be scheduled under CGC). We will follow the constructs in [17], and use the SB scheduling hint for all BP computations. The other new feature in MO-MS (relative to MO-FFT) is that the recursive sub-tasks are not guaranteed to have sizes that are within a constant factor of each other, though they are all of size $O(r^3)$. In order to achieve similar sizes for parallel tasks that solve these sub-problems, SPMS calls the 'u-tree' computation, which is a BP computation that packages the parallel recursive calls into approximately equal-sized sub-computations. We give a high-level description of the steps in MO-MS below. Further details on these steps can be found in [17].

In Step 2, the algorithm MO-MS deterministically samples the r sorted lists, sorts the samples, and uses the sorted list of sample points to partition the input set of lists into $O(m/r^2)$ disjoint and linearly-ordered merging problems. Since each of these newly created merging subproblems can contain many more lists in relation to the total size than the ideal size for MO-MS, the merging of these lists is done in two stages: in Step 5, the lists are grouped (arbitrarily) in groups of $O(\sqrt{r})$ lists, which is a suitable number of lists for the size of the problem, and each such subcollections of lists is combined into a single sorted list by a call to MO-MS. Then in Step 7, for each index i, the collection of output lists from Step 5 is combined into a sorted list by a call to MO-MS.

Step 2 consists of several calls (though a constant number) to computations with the same structure as space-bounded prefix sums and space-bounded matrix transposition. Thus the bounds derived for those algorithms apply to Step 2. Steps 3 and 8 are very simple steps that group consecutive elements together, and can be performed efficiently with either CGC or SB. As mentioned earlier, Steps 4 and 6 are needed because the recursive sub-tasks generated by MO-MS are not guaranteed to have sizes that are within a constant factor of each other, though they are all of size $O(r^3)$. In order to achieve similar sizes for parallel tasks that solve these sub-problems, the SPMS algorithm in [17] calls the 'u-tree' computation, which packages the parallel recursive calls into approximately equal-sized computations. By calling the same u-tree computation in MO-MS, we obtained tasks of similar sizes, which can then be scheduled in the same way as the CGC⇒SB steps in the FFT algorithm.

Due to the similarity in the structure of MO-MS and MO-FFT, the results for MO-FFT under our scheduler translate to SPMS. However, the critical pathlength increases from $O(B_1 \log n)$ to $O(B_1 \log n \log\log n)$ due to the BP computations that use prefix sums and related computations; these have $O(B_1 \log n)$ critical pathlength instead of the constant depth MO-MT used in MO-FFT. This gives us the following result.

Theorem 6. *The multicore-oblivious algorithm MO-MS terminates in* $O((n/p)\log n + B_1 \log n \log\log n)$ *parallel time, and incurs* $O\left((n/(q_i B_i))\log_{C_i} n\right)$ *cache misses at each level-i cache, for all* $1 \le i \le h-1$*, provided all caches are tall and* $n \ge C_{h-1}$.

The cache-miss bound stated in Theorem 6 is optimal, and this can be established by ex-

MO-SpM-DV($(A_v, A_0), x; y; k_1, k_2$)
Input: A row-major representation (A_v, A_0) of a sparse $n \times n$ matrix A, and a vector x of length n. In (A_v, A_0), A_v is a vector of all non-zero elements $A[i, j]$ of A sorted in lexicographically non-decreasing order of $\langle i, j\rangle$, and each element $A[i, j]$ is stored as an ordered pair $\langle j, A[i, j]\rangle$. Each entry $A_0[i]$ of vector A_0 contains the starting location of row i in A_v with $A_0[n + 1]$ containing $n + 1$.
Output: Computes $y[k_1 \ldots k_2]$, where y is a vector of length n containing the product Ax.
Space Bound: $S(m) = 4m$, where $m = k_2 - k_1 + 1$.

1: **if** $k_1 = k_2$ **then**
2: $y[k_1] := 0$
3: **for** $k := A_0[k_1]$ **to** $A_0[k_1 + 1] - 1$ **do**
$\langle j, a\rangle := A_v[k],\ \ y[k_1] := y[k_1] + a \times x[j]$
4: **else**
5: $k := \lfloor (k_1 + k_2)/2 \rfloor$
6: **[CGC⇒SB] parallel:** MO-SpM-DV($(A_v, A_0), x; y; k_1, k$),
MO-SpM-DV($(A_v, A_0), x; y; k + 1, k_2$)

Figure 8: MO-SpM-DV: multicore-oblivious sparse matrix and dense vector multiplication.

tending to multi-level caches, the cache-miss lower bound observed in [17] for any comparison-based sorting algorithm on a two-level multicore model.

Finally, we note that if SPMS is executed on an input of size $m \leq q_i C_i$, then the cache complexity at level-i is $O\left((m/(q_i B_i)) \log_{r_i} m\right)$, where $r_i = \min\{C_i, m/q_i\}$. This fact is used in the MO-LR algorithm in Section 6.

3.4.3. Sparse Matrix Dense Vector Multiplication (SpM-DV)

In this section we show that CGC⇒SB efficiently schedules the separator-based sparse matrix dense vector multiplication algorithm given in [10], provided the matrix has a support graph with good separators. A class of graphs closed under the subgraph relation is said to satisfy a $f(n)$-*edge separator theorem* if there exist constants $\alpha \in [1/2, 1)$ and $\beta > 0$ such that every n-node graph G from the class can be partitioned into two vertex-disjoint subgraphs containing at most αn vertices each and with no more than $\beta f(n)$ edges of G crossing the partition [23]. The *support graph* G_A of an $n \times n$ matrix A is defined to be the graph with vertex set $\{1, \ldots, n\}$ and edge set $\{(i, j)|A[i, j] \neq 0\}$. We say that A satisfies an $f(n)$-edge separator theorem if its support graph satisfies such a theorem. A separator tree of A, denoted by T_A, is constructed by applying the separator theorem to the whole support graph to get two components, and then recursively applying the theorem to each component until only a single node remains at each leaf of the tree. The pseudocode of MO-SpM-DV is provided in Figure 8.

As in [10], we assume that the rows and columns of the matrix A input to MO-SpM-DV are reordered based on the left to right ordering of leaves in its separator tree which leads to the following theorem proved in the full paper.

Theorem 7. *Any $n \times n$ sparse matrix A satisfying an n^ϵ-edge separator theorem with $\epsilon < 1$ can be reordered so that when executed on an h-level HM model with p cores MO-SpM-DV ter-*

minates in $O(n/p + B_1 + \log(n/B_1))$ *parallel steps, and incurs* $O\left((n/q_i)\left(1/B_i + 1/C_i^{1-\epsilon}\right)\right)$ *cache misses at each level-i cache, for all* $1 \leq i \leq h-1$, *provided* $n \geq C_{h-1}$.

Proof. It is not difficult to see that under the CGC⇒SB scheduler each task anchored at C_1 will have space bound $\Omega(B_1)$ (since $n \geq C_{h-1} \geq p \cdot C_1 \geq p \cdot B_1$), and that at each level i cache λ, $\Theta(n/(q_i C_i))$ tasks will be anchored whose parents have space bound too large for λ.

Once such a task τ is anchored to λ all its descendant subtasks will be executed completely under the shadow of λ. Hence, the total number of cache misses incurred at λ will be the sum of the cache misses incurred by these tasks at λ. Since the space bound of τ is $4m$, where m is the length of the segment of y computed by τ, clearly, $C_i/8 < m < C_i/4$. Let the starting and the ending index of y assigned to τ be k_1 and k_2, respectively. Now if we load a segment of x of length $2m$ centered at index $(k_1 + k_2)/2$, then for each index $j \in [k_1, k_2]$, the entire subtree T_j of T_A with leaves spanning indices $[j - m/2, j + m/2]$ will be in λ. When the algorithm is at row k consider a non-zero element $A[k, j]$ causing a read of $x[j]$ which corresponds to an edge (k, j) in G_A. If j is within T_j, then $x[j]$ is a cache hit, otherwise it may incur a cache miss. However, according to the edge separator theorem, only $O(m^\epsilon)$ such misses can occur. Observe that $O(m/B_i)$ additional cache misses will be incurred for loading y, A_v, A_o, and $x[(k_1 + k_2)/2 - m, \ldots, (k_1 + k_2)/2 + m]$ into λ. Hence, τ will incur $O(m/B_i + m^\epsilon) = O(C_i/B_i + {C_i}^\epsilon)$ cache misses. Therefore, $\mathcal{Q}_i(n) = O(n/(q_i C_i) \cdot (C_i/B_i + {C_i}^\epsilon)) = O\left((n/q_i) \cdot \left(1/B_i + 1/C_i^{1-\epsilon}\right)\right)$.

Since the scheduler distributes the tasks across cores evenly, each row of A has at most a constant number of non-zero entries, and the computation has a critical pathlength of $O(B_1 + \log(n/B_1))$, the $O(n/p + B_1 + \log(n/B_1))$ parallel running time of the algorithm follows immediately. Since the total work is $\Omega(n)$, the speed-up is optimal for $p \leq n/(B_1 + \log(n/B_1))$. □

4. Review of Network-Obliviousness

The notion of obliviousness in distributed-memory platforms is explored in [4], which introduces the following framework for the design and analysis of algorithms which perform efficiently across a wide class of parallel machines with differing computing and communication characteristics, without mentioning machine parameters in their specifications. A network-oblivious algorithm is designed in an abstract model, denoted with $\mathsf{M}(N)$, independent of machine parameters; then, its performance is analyzed in a model, named $\mathsf{M}(p, B)$, featuring two parameters characterizing parallelism and interconnection network. The above framework is interesting since optimality of the network-oblivious algorithm in $\mathsf{M}(p, B)$ for wide parameter ranges implies optimality on the Decomposable BSP (D-BSP) model [24, 25] which effectively describes the platforms on which we expect the network-oblivious algorithm to be actually executed [26]. An in-depth coverage of the framework is also provided in [27, 28, 29].

More in details, a network-oblivious algorithm $\mathcal{A}$ is an algorithm designed for the $\mathsf{M}(N)$ model, where N is a suitable function of the input size and represents the maximum number

of processors for which the computation is designed. An $\mathsf{M}(N)$ is a complete network of N *processing elements* (*PEs*) each consisting of a CPU and an unbounded local memory. $\mathcal{A}$ consists of a sequence of synchronous supersteps: in a superstep a PE performs operations on local data and sends messages to other PEs. The complexity of $\mathcal{A}$ is then evaluated by executing the algorithm on the $\mathsf{M}(p, B)$ model. The $\mathsf{M}(p, B)$, where $p \leq N$ and $B \geq 1$, is an $\mathsf{M}(p)$ whose PEs are called *processors* and where messages exchanged between two processors in a superstep can be envisioned as traveling within *blocks* of fixed size B. A network-oblivious algorithm can be naturally executed on $\mathsf{M}(p, B)$ for every $p \leq N$ and B by stipulating that each processor carries out the operations of N/p consecutive PEs. The *communication* (resp., *computation*) *complexity* of $\mathcal{A}$ is the sum over all supersteps of the maximum number of blocks sent/received (resp., operations performed) by a processor in each superstep.

Under some reasonable assumptions, network-oblivious algorithms with optimal communication complexity on any $\mathsf{M}(p, B)$ exhibit optimal communication time on a variant of the D-BSP model, denoted as $\mathsf{D\text{-}BSP}(P, \boldsymbol{g}, \boldsymbol{B})$, where $\boldsymbol{g} = (g_0, \ldots, g_{\log P-1})$ and $\boldsymbol{B} = (B_0, \ldots, B_{\log P-1})$. A D-BSP is essentially an $\mathsf{M}(P, \cdot)$ machine where processors are recursively partitioned into 2^i clusters of size $P/2^i$ for each $0 \leq i < \log P$. The *communication cost* of a superstep s, where each processor communicates with processors within its cluster of size 2^i, is defined to be $h_s g_i$, where h_s is the maximum number of blocks of size B_i sent/received by a processor during s on $\mathsf{M}(P, B_i)$.

Network-oblivious algorithms for matrix multiplication and transposition, FFT, and sorting are provided in [4]. The network and multicore-oblivious approaches have strong connections: indeed, other than sorting and SpM-DV, the multicore-oblivious algorithms in Section 3 are all adapted from network-oblivious algorithms presented in [4]. In the following sections we reinforce these connections by showing that network-oblivious algorithms for GEP and list ranking can be derived by exploiting multicore-oblivious solutions as well (see Sections 5.2 and 6.2, respectively).

5. Gaussian Elimination Paradigm

Let x be an $n \times n$ matrix with entries from an arbitrary domain $\mathcal{S}$, and let $f : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ be an arbitrary function. By *Gaussian Elimination Paradigm* [7] we refer to the computation in Figure 9, where the algorithm modifies x by applying a given set of *updates*, denoted by $\langle i, j, k \rangle$ for $i, j, k \in [0, n)$. We let Σ_f denote the set of updates the algorithm needs to perform. Many problems can be solved by GEP, including Floyd-Warshall's all-pairs shortest path, Gaussian Elimination and LU decomposition without pivoting, and matrix multiplication.

A cache-oblivious recursive implementation of GEP, called *I-GEP*, was presented in [30] and parallelized in [7] for multicore models with one level of cache. I-GEP consists of four recursive functions $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ which are reproduced in the appendix for convenience. The functions accept as input four matrices $X \equiv x[I, J]$, $U \equiv x[I, K]$, $V \equiv x[K, J]$ and $W \equiv x[K, K]$, where I, J, K denote suitable intervals in $[0, n)$, and they differ in the amount

Input: $n \times n$ matrix x, function $f : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \times \mathcal{S} \to \mathcal{S}$, set Σ_f of triplets $\langle i, j, k\rangle$, with $i, j, k \in [0, n)$.
Output: transformation of x defined by f and Σ_f.

```
1: for k ← 0 to n − 1 do
2:   for i ← 0 to n − 1 do
3:     for j ← 0 to n − 1 do
4:       if ⟨i, j, k⟩ ∈ Σ_f then
5:         x[i, j] ← f(x[i, j], x[i, k], x[k, j], x[k, k])
```

Figure 9: Gaussian Elimination Paradigm (GEP).

of overlap X, U, V and W have among them: $\mathcal{A}$ assumes completely overlapping matrices, $\mathcal{B}$ (resp., $\mathcal{C}$) expects that $X \equiv V$ and $U \equiv W$ (resp., $X \equiv U$ and $V \equiv W$), and $\mathcal{D}$ assumes completely non-overlapping matrices (other types of overlapping are not possible). Each function performs updates in $\Sigma_f \cap (I \times J \times K)$ (i.e., each update $\langle i, j, k\rangle$ such that $x[i, j]$, $x[i, k]$, $x[k, j]$ and $x[k, k]$ are contained in X, U, V, and W, respectively) by means of eight recursive calls to $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$, and using suitable quadrants of X, U, V, and W as inputs. The initial call is $\mathcal{A}(x, x, x, x)$. The four functions also differ in the amount of parallelism they offer: intuitively, the less the overlap among the input matrices the more flexibility the function has in ordering its recursive calls, and thus leading to better parallelism.

I-GEP produces the correct output under certain conditions which are met by all notable instances mentioned above, and incurs $O(n^3/B\sqrt{C})$ cache misses and terminates in $O(n^3/p + n\log^2 n)$ parallel time [7] when executed on p cores with one cache level of size C and block length B, for both shared and distributed caches. Also presented in [7] is *C-GEP* which extends I-GEP and implements correctly any instance of GEP with no asymptotic degradation in performance. Finally, tiled I-GEP [11] runs in $O(n^3/p+n)$ parallel time without increasing the cache complexity on HM [31] but is not multicore-oblivious.

In the next subsections we show that the parallel implementation of I-GEP can be transformed into an MO algorithm through the SB scheduler. Then, we describe the NO algorithm and introduce the notion of commutative GEP computation to prove its correctness.

5.1. Multicore-Oblivious Algorithm

Theorem 8 below states the performance of I-GEP under SB scheduler. The cache-miss bound is based on the observation that a total of $\Theta\left(n^3/(C_i\sqrt{C_i})\right)$ tasks are anchored at level-i caches incurring $O\left(C_i/B_i + \sqrt{C_i}\right)$ level-i cache-misses each. The parallel time follows from the observation that if C_i is larger than $p_i C_{i-1}$ by the factor stated in the theorem, the critical path of the I-GEP computation DAG has no asymptotic effect on its parallel time.

Theorem 8. *When executed under the SB scheduler, I-GEP on an $n \times n$ input, $n^2 \geq C_{h-1}$, incurs $O\left(n^3/(q_i B_i\sqrt{C_i})\right)$ cache misses at each level-i cache, for all $1 \leq i \leq h-1$, and terminates in $O\left(n^3/p\right)$ parallel time, provided all caches are tall, and $C_i \geq c_i \cdot p_i \cdot C_{i-1}$ with $c_i = 2\log^2\left(C_i/C_{i-1}\right)$ holds for $2 \leq i \leq h-1$. These bounds are optimal.*

	I-GEP's $\mathcal{D}$	N-GEP's $\mathcal{D}^*$
Round 1	$\mathcal{D}(X_{11}, U_{11}, V_{11}, W_{11}), \mathcal{D}(X_{12}, U_{11}, V_{12}, W_{11})$ $\mathcal{D}(X_{21}, U_{21}, V_{11}, W_{11}), \mathcal{D}(X_{22}, U_{21}, V_{12}, W_{11})$	$\mathcal{D}^*(X_{11}, U_{11}, V_{11}, W_{11}), \mathcal{D}^*(X_{12}, U_{12}, V_{22}, W_{22})$ $\mathcal{D}^*(X_{21}, U_{22}, V_{21}, W_{22}), \mathcal{D}^*(X_{22}, U_{21}, V_{12}, W_{11})$
Round 2	$\mathcal{D}(X_{11}, U_{12}, V_{21}, W_{22}), \mathcal{D}(X_{12}, U_{12}, V_{22}, W_{22})$ $\mathcal{D}(X_{21}, U_{22}, V_{21}, W_{22}), \mathcal{D}(X_{22}, U_{22}, V_{22}, W_{22})$	$\mathcal{D}^*(X_{11}, U_{12}, V_{21}, W_{22}), \mathcal{D}^*(X_{12}, U_{11}, V_{12}, W_{11})$ $\mathcal{D}^*(X_{21}, U_{21}, V_{11}, W_{11}), \mathcal{D}^*(X_{22}, U_{22}, V_{22}, W_{22})$

Table 2: Recursive calls performed by $\mathcal{D}$ and $\mathcal{D}^*$; calls in a round are performed in parallel.

Proof. Since I-GEP accesses data only inside recursive function calls with input size 1×1, it suffices to compute the misses incurred only by tasks with space bound $\leq C_{h-1}$. Let τ be a task anchored at a level-i cache λ. Observe that each subtask generated by τ has space bound $s(\tau)/4$, where $s(\tau)$ is the space bound of τ. Moreover, since $p_i \geq 2 \Rightarrow C_i \geq c_i \cdot p_i \cdot C_{i-1} > 4C_{i-1}$ for $i \in [2, h-1]$, each descendant of τ anchored at an L_{i-1} and an L_1 cache under $shadow(\lambda)$ will have space bound larger than $C_{i-1}/4$ and $\Omega\left(B_1^2\right)$, respectively. There are $\Theta\left(n/\sqrt{C_i}\right)$, $\Theta\left(n^2/C_i - n/\sqrt{C_i}\right)$ and $\Theta\left(n^3/(C_i\sqrt{C_i}) - 2 \cdot n^2/C_i + n/\sqrt{C_i}\right)$ tasks with space bound $\Theta\left(C_i\right)$ corresponding to I-GEP function $\mathcal{A}$, $\mathcal{B}/\mathcal{C}$ and $\mathcal{D}$, respectively. Observing that when executed entirely under λ any such task will incur $O\left(\sqrt{C_i} + C_i/B_i\right)$ misses in λ, the claimed cache-miss bound follows. Since I-GEP includes matrix multiplication, a cache-miss lower bound for I-GEP matching its upper bound can be proved by extending the lower bound result proved in [17] for matrix multiplication on a two-level multicore model to the multi-level model.

We will compute the parallel running time inductively. Let $\mathcal{T}_i(s)$ be an upper bound on the parallel running time of any I-GEP function with space bound s executed on any level-i cache. Clearly, when anchored at any L_1 cache and thus executed by a single core, for any task τ_1 with space bound $\Theta\left(C_1\right)$, $\mathcal{T}_1(C_1) = O\left(C_1^{3/2}\right) = O\left(C_1^{3/2}/p_1'\right)$ (since $p_1' = 1$). Hence as inductive hypothesis let us assume that $\mathcal{T}_{i-1}(C_{i-1}) = O\left(C_{i-1}^{3/2}/p_{i-1}'\right)$ holds for some $i-1 \geq 1$ (recall that p_{i-1}' is the number of cores subtended by any level-$(i-1)$ cache). Now consider any task τ with space bound $\Theta\left(C_i\right)$ anchored at any level-i cache λ. Following [7] one can verify that the critical pathlength of τ is $O\left(\sqrt{C_i/C_{i-1}}\log^2\sqrt{C_i/C_{i-1}} \cdot \mathcal{T}_{i-1}(C_{i-1})\right)$, and thus using Brent's principle, $\mathcal{T}_i(C_i) = O\left(((C_i/C_{i-1})^{3/2}/p_i + \sqrt{C_i/C_{i-1}}\log^2\sqrt{C_i/C_{i-1}}) \cdot \mathcal{T}_{i-1}(C_{i-1})\right)$. Since $C_i \geq c_i \cdot p_i \cdot C_{i-1}$ and $c_i = 2\log^2\left(C_i/C_{i-1}\right)$, we get $\sqrt{C_i/C_{i-1}}\log^2\sqrt{C_i/C_{i-1}} \leq \frac{1}{8}(C_i/C_{i-1})^{3/2}/p_i \Rightarrow \sqrt{C_i/C_{i-1}}\log^2\sqrt{C_i/C_{i-1}} \cdot \mathcal{T}_{i-1}(C_{i-1}) = O\left(C_i^{3/2}/p_i'\right)$. Also $C_i \geq c_i \cdot p_i \cdot C_{i-1}$ implies $C_i^{3/2}/p_i' \geq C_1\sqrt{C_i} > 1$ for all i, and thus $(C_i/C_{i-1})^{3/2}/p_i = \left(C_i^{3/2}/p_i'\right) / \left(C_{i-1}^{3/2}/p_{i-1}'\right) = O\left(C_i^{3/2}/p_i'\right)$. Hence, $\mathcal{T}_i(C_i) = O\left(C_i^{3/2}/p_i'\right)$. Extending the induction up to level-$(h-1)$, we obtain, $\mathcal{T}_{h-1}(C_{h-1}) = O\left(C_{h-1}^{3/2}/p_{h-1}'\right) = O\left(C_{h-1}^{3/2}/p\right)$, and since there are $O\left((n/\sqrt{C_{h-1}})^3\right)$ I-GEP functions with space bound $\Theta\left(C_{h-1}\right)$, we conclude that $\mathcal{T}(n) = O\left((n/\sqrt{C_{h-1}})^3 \cdot C_{h-1}^{3/2}/p\right) = O\left(n^3/p\right)$. Since I-GEP performs $\Omega\left(n^3\right)$ work, the parallel speed-up is optimal for $p = \prod_{i=1}^{h} p_i = \prod_{i=2}^{h-1} p_i \leq (C_{h-1}/C_1)\prod_{i=2}^{h-1}(1/c_i)$, which is the maximum number of cores allowed in the HM model due to the constraints on relative cache sizes. □

5.2. Network-Oblivious Algorithm

N-GEP is an optimal network-oblivious algorithm which performs correctly any commutative GEP computation which is solved by I-GEP. It exhibits space optimality, which is not yielded by a straightforward network-oblivious implementation of I-GEP, and is optimal on $\mathsf{M}(p, B)$ and on the D-BSP model for a wide range of their machine parameters. A GEP computation is *commutative* if its function f satisfies $f(f(y, u_1, v_1, w_1),\ u_2, v_2, w_2) = f(f(y, u_2, v_2, w_2), u_1, v_1, w_1)$ for each $y,\ u_1,\ v_1,\ w_1,\ u_2,\ v_2,\ w_2$ in $\mathcal{S}$. Not all GEP computations are commutative, however all of the instances of GEP for the aforementioned problems are commutative.

N-GEP is built on the parallel implementation of I-GEP in [7] (given in appendix for convenience) from which it inherits the recursive structure, and is designed for $\mathsf{M}(n^2/\log^2 n)$ (the number of PEs reflects the critical pathlength of I-GEP). N-GEP consists of four functions $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}^*$: the first three functions are simple adaptations of their counterparts in I-GEP to the $\mathsf{M}(n^2/\log^2 n)$ model; in contrast, $\mathcal{D}^*$ is based on I-GEP's $\mathcal{D}$ but solves the eight recursive calls in a different order and is equivalent to $\mathcal{D}$ for commutative GEP computations.

Recall that $\mathcal{D}$ consists of eight recursive calls to itself which are solved in two rounds of four parallel calls each and accept suitable quadrants[6] of X, U, V and W as input (see Table 2). We observe that, in each round, there are quadrants of U, V and W which are given as input to two concurrent calls. Since these quadrants are required in read-only mode, $\mathcal{D}$ can be efficiently executed on a CREW shared-memory, as the model where I-GEP was designed. A network-oblivious implementation of $\mathcal{D}$ increases the communication complexity of the algorithm notably: indeed, if k PEs read the same value stored in an unique PE, then the communication complexity is $\Theta(k)$ (while it is $O(1)$ in a CREW shared-memory). However, if quadrants are replicated, communication decreases at the cost of a non-constant memory blow-up. Hence, we adopt $\mathcal{D}^*$ where recursive calls are ordered in such a way no quadrants of U and V are required twice in a round (see Table 2). $\mathcal{D}^*$ exhibits a constant memory blow-up and is equivalent to $\mathcal{D}$ for commutative GEP computations, since subproblems in $\mathcal{D}$ can be performed in any order when a GEP computation is commutative. We note that W_{11} and W_{22} are still required twice in each round: however, since W_{12} and W_{21} are not used, W_{22} and W_{22} can be duplicated without a memory blow-up by setting $W_{12} = W_{11}$ and $W_{21} = W_{22}$.

The following theorem shows that N-GEP correctly computes any commutative GEP computation solved by I-GEP, and performs optimally on the $\mathsf{M}(p, B)$ and D-BSP models.

Theorem 9. *The network-oblivious algorithm N-GEP performs correctly any commutative GEP computation solved by I-GEP. When executed on $\mathsf{M}(p, B)$ for $p \le n^2/\log^2 n$ and input size n, N-GEP exhibits optimal $\Theta\left(n^3/p\right)$ computation complexity and $O\left(n^2/(\sqrt{p}B) + n\log^2 n\right)$ communication complexity, which is optimal when $p \le n^2/\log^4 n$ and $B \le n/(\sqrt{p}\log^2 n)$. Furthermore, N-GEP is optimal on a D-BSP$(P, \boldsymbol{g}, \boldsymbol{B})$ when $P \le n/\log n$ and $B_i = O\left(n2^{i/2}/(P\log n)\right)$,*

[6] We denote the top-left, top-right, bottom-left and bottom-right quadrants of X by X_{11}, X_{12}, X_{21} and X_{22}, respectively (similarly for U, V and W).

for each $0 \leq i < \log P$.

Proof. When a GEP computation is commutative, updates in I-GEP's $\mathcal{D}$ can be performed in any order since U, V and W are fixed in $\mathcal{D}$. Then, it can be proved by induction that N-GEP's $\mathcal{D}^*$ is equivalent to I-GEP's $\mathcal{D}$. As a consequence, $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are also equivalent to their respective implementations in I-GEP. The first part of the theorem follows.

When N-GEP is executed on $\mathsf{M}(p, B)$, a depth-i recursive call to $\mathcal{D}^*$ with input size $m = n/2^i$ is executed by q $\mathsf{M}(p, B)$-processors, where $p/4^i \leq q \leq p$ (the exact value depends on the kind of the i previous recursive calls). Since a call to $\mathcal{D}^*$ consists of two rounds and in each round four recursive calls to $\mathcal{D}^*$ of size $m/2$ are performed in parallel, its communication complexity $H_{\mathcal{D}^*}(m, q, B)$ is upper bounded by the following simple recurrence:

$$H_{\mathcal{D}^*}(m, q, B) \leq \begin{cases} O(1) & \text{if } q \leq 1 \text{ or } m \leq 1 \\ 2H_{\mathcal{D}^*}(\frac{m}{2}, \frac{q}{4}, B) + O\left(\frac{m}{qB} + 1\right) & \text{otherwise} \end{cases}$$

which solves to $O\left(m^2/(qB) + \min\{\sqrt{q}, m\}\right)$. Similarly, it can be proved that the communication complexity is $O\left(m^2/(qB) + m \log m\right)$ for functions $\mathcal{B}$ and $\mathcal{C}$, and $O\left(m^2/(qB) + m \log^2 m\right)$ for $\mathcal{A}$, respectively. Since N-GEP consists of a call to $\mathcal{A}$ with $m = n$ and $q = p$, the upper bound on the communication complexity of N-GEP follows. Similar recurrence systems show that the computation complexity of functions $\mathcal{A}, \mathcal{B}, \mathcal{C}$ and $\mathcal{D}^*$ is $O\left(n^2/p\right)$ as soon as $p \leq n^2/\log^2 n$. The space used by each $\mathsf{M}(p, B)$ processor while executing N-GEP is $\Theta\left(n^2/p\right)$ space: indeed, $\mathcal{D}^*$ does not require additional space since no new matrices are allocated, while the amount of additional space required by $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ decreases geometrically in each recursive level and is asymptotically negligible.

The optimality of the network-oblivious algorithm derives by observing that NO-GEP solves matrix multiplication using only semiring operations and $\Theta\left(n^2/p\right)$ space per processor. Known lower bounds for this problem are $\Omega\left(n^3/p\right)$ for computation and $\Omega\left(n^2/(\sqrt{p}B)\right)$ for communication [32]. Therefore, the optimality of N-GEP follows for the stated intervals of p and B.

The upper bound on the communication time is derived as for the communication complexity, however different B_i's and g_i's are used in each recursive level. Tedious, but simple, derivations shows that the communication time of N-GEP on $\mathsf{D\text{-}BSP}(P, \mathbf{g}, \mathbf{B})$, for $1 < P \leq n^2/\log^2 n$, is $O\left(\sum_{i=0}^{\log P - 1} \left(\frac{n^2 2^{i/2}}{B_i P} + n \log n\right) g_i\right)$. The time matches, for the ranges stated in the theorem, the lower bound in [27] for the communication time on the D-BSP of matrix multiplication algorithms using only semiring operations and $O\left(n^2/p\right)$ space per processor. □

Finally, we affirm that N-GEP can be extended to correctly implement any commutative GEP computation, without performance degradation, by adopting ideas from C-GEP.

6. List Ranking and Other Graph Algorithms

We now present multicore-oblivious and network-oblivious algorithms for list ranking and other graph problems. We represent a linked list of n nodes as an array where each position contains a node identifier and pointers to its successor and predecessor. The rank of a node is its distance from the end of the list, and the list ranking problem consists in determining the ranks of every node.

6.1. Multicore-Oblivious Algorithms

6.1.1. List Ranking

Our multicore-oblivious algorithm, named *MO-LR*, employs the list contraction technique described in [33]: this technique solves the list ranking problem by identifying an independent set S of the list[7] of size $\Theta(n)$, contracting the list by removing nodes in S, recursively solving the list ranking problem on the contracted list, and then extending the solution to the removed nodes. It is not difficult to see that list contraction and the extension of the solution to nodes in the independent set can be accomplished with $O(1)$ sorts and scans using the CGC⇒SB and CGC schedules, respectively. This algorithm is derived from those in [34, 35] by adopting our multicore-oblivious primitives for sorting and scanning and by recursively contracting the linked list down to a constant size, instead of a variable size based on architectural parameters. We use as sorting primitive the multicore-oblivious algorithm MO-MS given in Section 3.4.

The multicore-oblivious algorithm, named *MO-IS*, for computing an independent set S is given in Figure 10. We observe that each step of MO-IS is solved by $O(1)$ sorts and scans using the CGC⇒SB and CGC schedules, respectively. In Step 1 we identify a $\log\log n$ coloring of the nodes by applying twice the deterministic coin flipping algorithm in [33] which, given a k-coloring, constructs a $(1 + \log k)$-coloring: since the new color of a node is determined by only the initial colors of the node and of its successor, $O(1)$ sorts and scans suffices to accomplish the coloring. During the j-th iteration of Step 7 we create duplicates of successor and predecessor of each color j node v in the independent set S to indicate that v is in S. Hence, when a node is found replicated in Step 6 of later iterations, it is removed and not inserted in S because at least one of its neighbors is in S. Clearly, Steps 5-7 require $O(1)$ sorts and scans.

Theorem 10. *MO-LR with input size n terminates in $O((n/p)\log n + (B_1\log(pB_1) + \log n \log\log n)\cdot \log(n/B_1)\cdot\log\log n)$ parallel time, and incurs $O\left((n/(q_iB_i))\log_{C_i} n\right)$ cache misses at each level-i cache, for all $1 \le i \le h-1$, for $n/\log\log n = \Omega\left(C_i \log_{p'_iB_1} C_i\right)$ and $B_i = O(n/(q_i \log n \log\log n))$.*

Proof. Consider the execution of MO-IS and let n_j denote the number of list nodes of color j after the coloring. At the beginning of the j-th iteration of the **for** loop, with $1 \le j \le \log\log n$, there are at most $3n_j$ nodes of color j since at most $2n_j$ nodes of color j were added in earlier iterations. A loop iteration consists of $O(1)$ sorts and scans of $\Theta(n_j)$ nodes, and

[7] An independent set of a linked list is a subset S of its nodes such that no two nodes in S are adjacent.

MO-IS(L)
Input: linked list L of n nodes.
Output: an independent set.

1: **[CGC⇒SB]** Identify a $\log\log n$ coloring of the nodes.
2: **[CGC⇒SB]** Sort a copy of the nodes by successor (and predecessor) to associate with each node the color of its successor (and predecessor).
3: **[CGC⇒SB]** Group in consecutive memory positions nodes of the same color by sorting nodes by color.
4: **for** each color $1 \leq j \leq \log\log n$ **do**
5: **[CGC⇒SB]** Sort nodes of color j by identifier.
6: **[CGC]** Identify and remove duplicates by comparing identifiers of consecutive nodes. Add the remaining nodes to the independent set.
7: **[CGC]** Add a duplicate of the successor (and predecessor) of each remaining node and move it in the respective color group. This is an indication that successors/predecessors cannot be inserted into the independent set.
8: **return** all nodes added to the independent set.

Figure 10: MO-IS: multicore-oblivious algorithm for computing an independent set.

the j-th iteration requires $O((n_j/p)\log n_j + \log n_j \log\log n_j)$ parallel steps if $n_j > pB_1$, and $O(B_1 \log(pB_1) + \log n_j \log\log n_j)$ otherwise because the scheduler reduces the number of active cores so that at most one core has less than B_1 nodes. It follows that MO-IS requires $O((n/p)\log n + (B_1 \log(pB_1) + \log n \log\log n)\cdot\log\log n)$ parallel steps since $\sum_{j=0}^{\log\log n - 1} n_j = n$.

Similarly, we can show that the j-th iteration incurs $O(n_j/(q_iB_i)\log_{C_i} n_j + (C_i/B_i)\log_{p'_iB_1}(q_iC_i) + \log n_j \log\log n_j)$ misses, where the last term is due to the critical pathlength of sorting (applied $\log\log n$ times). Then, MO-IS incurs $O((n/(q_iB_i))\log_{C_i} n + (C_i/B_i)\log_{p'_iB_1}(q_iC_i)\log\log n + \log n(\log\log n)^2)$ misses. The upper bounds in the theorem follow by observing that the returned independent set has size $n/3$ [34], and thus the linked list shrinks to a size smaller than B_1 in $O(\log(n/B_1))$ applications of MO-IS. □

We note that the $\log(n/B_1)$ factor can be replaced by $O(\log\log n)$ by using a hybrid list ranking algorithm [22] that switches to the standard pointer jumping algorithm after $O(\log\log n)$ contractions of the linked list using MO-IS. At this stage the linked list will have $O(n/\log n)$ nodes and hence, even though the size of the input will not contract when we use the standard pointer jumping algorithm, the overall work will remain optimal. Each phase of the pointer jumping algorithm involves a sort step (executed using CGC⇒SB) to order the elements of the linked list in terms of the successor values, followed by a scan (executed using CGC) to replace each element's successor by the successor's successor, which is the pointer-jumping step. Also note that the $\log\log n$ factor common to the 2nd and the 3rd term in the parallel running time of MO-LR can be reduced to[8] $\log^{(k)} n$ for any integer constant $k > 2$ by repeating the coloring algorithm k times (instead of twice) in Step 1 of MO-IS.

[8] $\log^{(1)} n = \log n$, and for any positive integer $k > 1$, $\log^{(k)} n = \log\log^{(k-1)} n$.

6.1.2. Other Graph Problems

By using the CGC hint, it is straightforward to obtain as in [22, 34, 35] multicore-oblivious algorithms for Euler tour, and several tree problems such as rooting a tree, traversal numbering, vertex depth, subtree size and connected components of a forest. These algorithms have the same complexity as MO-LR.

Our multicore-oblivious algorithm for computing the connected components of a graph is based on the PRAM CREW algorithm in [36] adapted to adjacency lists, and uses MO-LR and tree computations to obtain the following result. Again, these algorithms are derived from the algorithms in [34, 35] by adopting our multicore-oblivious primitives for sorting and scanning and by recursively contracting the graphs down to a constant size, instead of variable sizes based on architectural parameters. More details are given in [31].

Theorem 11. *There exists a multicore-oblivious algorithm for computing the connected components of a graph of n nodes and m edges, which terminates in $O(N \log N \log(N/B_1)/p + (B_1 \log(pB_1)\ \log\log N + \log N(\log\log N)^2)(\log^2(N/B_1)))$ parallel time, and incurs $O(N/(q_i B_i) \log_{C_i} N \log(N/B_1) + ((C_i/B_i) \log_{p'_i B_1}\ (q_i C_i) \log\log n + \log N(\log\log n)^2) \log^2(N/B_1))$ cache misses at each level-i cache, for all $1 \leq i \leq h-1$, for $N = n + m$.*

6.2. Network-Oblivious Algorithms

6.2.1. List Ranking

Our network-oblivious algorithm, named NO-LR, is defined on $\mathsf{M}(n^{1-\epsilon})$, where $\epsilon \in (0, 1)$ is an arbitrary constant, and employs the list contraction technique described in the previous section. NO-LR relies on algorithm NO-IS which computes an independent set of a list. Both algorithms are simple adaptations to the network-oblivious framework of the respective multicore-oblivious implementations. However, while casting MO-IS to the network-oblivious framework, some attention is required for improving performance. Below, we describe the most relevant adjustments.

The sorting and scan primitives in NO-IS are carried out by the following network-oblivious algorithms. The scan operations are implemented through a casting to $\mathsf{M}(n^{1-\epsilon})$ of the algorithm MO-PS described in Section 3.2.2 for prefix sums; the so defined algorithm exhibits $O(\log p)$ communication and $O(n/p)$ computation complexities on $\mathsf{M}(p, B)$, for each $p \leq n^{1-\epsilon}$ and $B \geq 1$. For sorting, NO-IS uses the network-oblivious sorting algorithm in [4] based on column sort. However, differently than the original algorithm, which is specified on $\mathsf{M}(n)$, we assume the network-oblivious algorithm to be defined in $\mathsf{M}(n^{1-\epsilon})$. This improves the performance of the algorithm when executed on $M(p, B)$, for each $p \leq n^{1-\epsilon}$ and $B \geq 1$, yielding $O\left(n/(pB) + \sqrt{n/p}\right)$ communication and $\Theta(n \log n/p)$ computation complexities. Further details on the primitives may be found in [31]. We note that a network-oblivious version of SPMS (see Section 3.4.2) looks likely, though there are some technical issues to address to port it from shared memory to the network-oblivious framework: however, this is beyond the scope of this paper, so we instead present the column sort algorithm.

Algorithm NO-IS guarantees that, at the beginning of the j-th iteration of the **for** loop (see Step 4 of MO-IS in Figure 10), for each $1 \leq j \leq \log\log n$, the N_j nodes with the same color are evenly distributed among the $n^{1-\epsilon}$ PEs. Furthermore, since the network-oblivious algorithms for sort and scan are designed in $\mathsf{M}(n^{1-\epsilon})$, the sort and scan operations within the j-th iteration are performed by $\Theta\left(N_j\right)^{1-\epsilon}$ PEs evenly selected among all the PEs. The network-oblivious algorithm is similar to the multicore-oblivious algorithm in the other details, and we obtain the following performance bounds.

Theorem 12. *When executed on $\mathsf{M}(p, B)$ for $p \leq (n/\log\log n)^{1-\epsilon}$ and input size n, NO-LR exhibits optimal $\Theta\left((n/p)\log n\right)$ computation complexity and $O(n/(pB) + ((n/p)\log\log n)^{1/2} + p^{\epsilon}\log n\log\log n)$ communication complexity, which is optimal for $B = O(\tilde{B})$, where $\tilde{B} = \min\{(n/(p\log\log n))^{1/2}, n/(p^{1+\epsilon}\log n\log\log n)\}$. Furthermore, NO-LR is optimal on a D-BSP$(P, \boldsymbol{g}, \boldsymbol{B})$ for $P \leq (n/\log\log n)^{1-\epsilon}$ and $B_0 = O(\tilde{B})$.*

Proof. As noted in the proof of Theorem 10, at the beginning of the i-th iteration of the **for** loop, with $1 \leq j \leq \log\log n$, there are at most $N_i = 3n_i$ nodes of color i, and a loop iteration consists of $O\left(1\right)$ sorts and scans of $\Theta\left(n_i\right)$ nodes using $n_i^{1-\epsilon}$ PEs evenly distributed among the $n^{1-\epsilon}$ PEs. Hence, when executed on the $\mathsf{M}(p, B)$, sorts and scans are performed by $p_i = \min\{n_i^{1-\epsilon}, p\}$ processors yielding $O\left(n_i/(pB)) + n_i^{\epsilon}/B + \sqrt{n_i/p} + n_i^{\epsilon/2} + \log p\right)$ communication and $O\left(n_i\log n/p + n_i^{\epsilon}\log n\right)$ computation complexities. The cost for moving the $\Theta\left(n_i\right)$ nodes in the $n_i^{1-\epsilon}$ selected PEs, that is $\min\{p, n_i^{1-\epsilon}\}$ processors of $\mathsf{M}(p, B)$, is bounded by $O\left(n_i^{\epsilon}/B + p^{\epsilon}\right)$ for communication and for $O\left(n_i^{\epsilon}\right)$ computation. Finally we remark that at the end of the j-th iteration the algorithm creates at most n_j duplicates of different colors, which should be distributed among PEs in order to guarantee that nodes of same color are evenly distributed in following iterations. This can be accomplished through a sort and at most $\log\log n$ scans (one per color). Therefore, the cost of the j-th iteration is $O\left(n_i/(pB) + n_i^{\epsilon}/B + \sqrt{n_i/p} + n_i^{\epsilon/2} + p^{\epsilon} + \log p\log\log n\right)$ for communication and $O\left(n_i\log n/p + n_i^{\epsilon}\log n\right)$ for computation. Since $\sum_{j=0}^{\log\log n-1} n_j = n$, we have that the communication and computation complexities of NO-IS are $O\left(n/(pB) + \sqrt{(n/p)\log\log n} + p^{\epsilon}\log\log n\right)$ and $O\left(n\log n/p + n^{\epsilon}(\log\log n)^{1-\epsilon}\log n\right)$, respectively.

The upper bounds on NO-LR follow by observing that the returned independent set has size $n/3$, and then the input of recursive calls decrease geometrically. In a D-BSP with the specified parameters, the communication time of NO-LR is $O\left(ng_0/p + 1\right)$ which is clearly optimal. □

6.2.2. Other Graph Problems

As with multicore-oblivious algorithms, it is easy to derive network-oblivious algorithms with the same complexities as NO-LR for Euler tour and many tree problems. By using the NO-LR algorithm and the network-oblivious primitives for sorting and scan described in Section 6.2.1, we get the following result for computing connected components in a graph.

Theorem 13. *There exists an network-oblivious algorithm defined on $\mathsf{M}((n+m)^{1-\epsilon})$, where ϵ*

is an arbitrary constant in $(0, 1)$*, for computing the connected components of a graph of* n *nodes and* m *edges. On* $\mathsf{M}(p, B)$ *for* $p \leq (\tilde{N}/\Gamma)^{1-\epsilon}$*, it exhibits* $O(\tilde{N}/(pB) + (\tilde{N}\Gamma/p)^{1/2} + p^{\epsilon}\Gamma \log n)$ *communication and* $O((\tilde{N}/p) \log n)$ *computation complexities, where* $\tilde{N} = n + m \log n$ *and* $\Gamma = \log n \log \log n$*.*

7. Conclusion

In this paper we have addressed the design of multicore algorithms that are oblivious to machine parameters. To this end, we proposed the HM model, which models the multicore as a parallel shared-memory machines with hierarchical multi-level caching, extending the 3-level multicore caching model in [10]. Then, we introduced the notion of a multicore-oblivious algorithm, which is a parallel algorithm that makes no mention of multicore parameters, but is allowed to supply certain directives ('hints') to the run-time scheduler to enable it to schedule the algorithm to exploit efficiency in both parallelism and caching. We have presented multicore-oblivious algorithms for several problems, including matrix transposition, FFT, sorting, GEP, list ranking and connected components.

We introduced two major types of scheduler hints, CGC (coarse-grained contiguous) and SB (space-bound), and a third one that combines these two (CGC⇒SB). While these were sufficient to extract efficiency in the algorithms we have presented, it is conceivable that a general-purpose run-time scheduler could be built to schedule a wide range of multicore algorithms efficiently and in an oblivious manner by suitably enhancing this set of hints.

As mentioned earlier, the notion of a multicore-oblivious algorithm is complementary to that of a network-oblivious algorithm [4]. However there are some important differences: In going from multicore-oblivious to network-oblivious, we need to move from shared-memory to message passing, which involves moving from the fine-grained reads in shared-memory to a coarse-grained message-passing environment. Here we are helped by the fact that the local memory of a processor in the network-oblivious model is large enough to hold all of the data that is sent to it (in contrast to the limited sizes of the caches in the multicore-oblivious model). Also, going from multicore-oblivious to network-oblivious involves moving from the concurrent read environment used in multicores to exclusive read. In going from network-oblivious to multicore-oblivious, we need to move from message passing to shared-memory, and also need to develop methods to exploit locality at all levels of the cache hierarchy. Nevertheless, in this paper we have established several connections between efficient multicore-oblivious and network-oblivious algorithms: we derived network-oblivious algorithms for GEP and list ranking by suitably adapting our multicore-oblivious algorithms, and our multicore-oblivious algorithms for matrix transposition and FFT from network-oblivious algorithms in [4]. These results raise hopes for a unified notion of obliviousness in parallel computation, which may be of interest with the advent of networks of multicores, such as the Blue Waters system [37].

We conclude with a summary of our main results in Table 3, where some minor constraints relating to cache and network parameters are omitted. Recall that under multicore-oblivious,

Problem	**Time**	**MO cache complexity**	**NO communication complexity**	**Max value of p**
Matrix transposition	$\Theta(n^2/p)$	$\Theta(n^2/(q_iB_i))$	$\Theta(n^2/(Bp))$ [4]	MO: n^2/B_1 NO: n^2
Prefix sum	$\Theta(n/p)$	$\Theta(n/(q_iB_i))$	$\Theta(\log p)$ [31]	MO: $n/(B_1 \log n)$ NO: $n/\log n$
FFT	$\Theta(n \log n/p)$	$\Theta(n/(q_iB_i) \log_{C_i} n)$	$\Theta(n/(pB) \log_{n/p} n)$ [4]	MO: n/B_1 NO: n
Sorting	$\Theta(n \log n/p)$	$\Theta(n/(q_iB_i) \log_{C_i} n)$	$\Theta(n/(pB))$ [31]	MO: $n/(B_1 \log \log n)$ NO: $n^{1-\epsilon}$, $\epsilon \in (0,1)$
GEP	$\Theta(n^3/p)$	$\Theta(n^3/(q_iB_i\sqrt{C_i}))$	$\Theta(n^2/(B\sqrt{p}))$	MO: $\min\left\{n^2/\log^2 n, C_{h-1}/\left(C_1 \prod_{i=2}^{h-1}\left(2\log^2(C_i/C_{i-1})\right)\right)\right\}$ NO: $n^2/\log^2 n$
List ranking	$\Theta(n \log n/p)$	$O(n/(q_iB_i))$	$O(n/(Bp))$	MO: $n/(B_1 + \log \log n) \log n \log \log n$ NO: $(n/\log \log n)^{1-\epsilon}$, $\epsilon \in (0,1)$

Table 3: Summary of our results. (MO: multicore-oblivious, NO: network-oblivious

the cache complexity at the i-th level is defined as the maximum number of block transfers into and out of any single L_i cache by the p'_i cores which share L_i, while under network-oblivious, the communication complexity is the maximum number of blocks sent or received by a processor. For prefix sums (the first row in the table), the network-oblivious communication cost is much smaller than the multicore-oblivious cache complexity due to the fact that the network model assumes that the data is initially distributed across the processors in the desired configuration, while in the HM model, all data resides initially in the shared memory, and the loading of the data into caches factors into the cache complexity. For sorting, the network-oblivious algorithm we analyze is based on column sort, while the multicore-oblivious algorithm, SPMS [17], is a hybrid of merge-sort and sample-sort; these are two different algorithms with different complexities, giving rise to the differences in the table, both for sorting, and for list ranking, which uses sorting as a subroutine. For all other entries in the table, the network-oblivious communication complexity coincides with the average number of cache misses incurred by a core in a level-i cache of size N/p, where N is the input size (i.e., N is n^2 for matrix computations, and is n otherwise).

Acknowledgment

The authors thank the anonymous reviewers for their comments. F. Silvestri would also like to thank A. Pietracaprina and G. Pucci for useful discussions. V. Ramachandran was supported in part by NSF Grants NSF CCF-0830737 and CCF-0850775. F. Silvestri was supported in part by MIUR under PRIN national research project "AlgoDEEP", and by University of Padova under projects STPD08JA32 and CPDA121378.

References

[1] M. Frigo, C. E. Leiserson, H. Prokop, S. Ramachandran, Cache-Oblivious Algorithms, ACM Transactions on Algorithms 8 (1) (2012) 4:1–4:22.

[2] L. Arge, G. S. Brodal, R. Fagerberg, Cache-Oblivious Data Structures, in: D. P. Mehta, S. Sahni (Eds.), Handbook of Data Structures and Applications, chap. 34, Chapman and Hall/CRC 2004, 34.1–34.27, 2004.

[3] E. D. Demaine, Cache-Oblivious Algorithms and Data Structures, 2002.

[4] G. Bilardi, A. Pietracaprina, G. Pucci, F. Silvestri, Network-Oblivious Algorithms, in: Proc. 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE Computer Society, 1–10, 2007.

[5] L. Arge, M. Goodrich, M. Nelson, N. Sitchinava, Fundamental parallel algorithms for private-cache chip multiprocessors, in: Proc. 20th ACM Symposium on Parallelism in Algorithms and Architecture (SPAA), ACM, 197–206, 2008.

[6] M. A. Bender, J. T. Fineman, S. Gilbert, B. C. Kuszmaul, Concurrent cache-oblivious B-trees, in: Proc. 17th ACM Symposium on Parallelism in Algorithms and Architecture (SPAA), ACM, 228–237, 2005.

[7] R. A. Chowdhury, V. Ramachandran, The Cache-Oblivious Gaussian Elimination Paradigm: Theoretical Framework, Parallelization and Experimental Evaluation, Theory of Computing Systems 47 (4) (2010) 878–919.

[8] M. Frigo, V. Strumpen, The Cache Complexity of Multithreaded Cache Oblivious Algorithms, Theory of Computing Systems 45 (2) (2009) 203–233.

[9] G. Blelloch, P. Gibbons, Effectively Sharing a Cache Among Threads, in: Proc. 16th ACM Symposium on Parallelism in Algorithms and Architecture (SPAA), ACM, 235–244, 2004.

[10] G. Blelloch, R. A. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen, M. Kozuch, Provably Good Multicore Cache Performance for Divide-and-Conquer Algorithms, in: Proc. 19th ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, 501–510, 2008.

[11] R. A. Chowdhury, V. Ramachandran, Cache-efficient dynamic programming algorithms for multicores, in: Proc. 20th ACM Symposium on Parallelism in Algorithms and Architecture (SPAA), ACM, 207–216, 2008.

[12] L. G. Valiant, A bridging model for multi-core computing, Journal of Computer and System Sciences 77 (1) (2011) 154–166.

[13] R. A. Chowdhury, F. Silvestri, B. Blakeley, V. Ramachandran, Oblivious Algorithms for Multicores and Network of Processors, in: Proc. 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS), IEEE Computer Society, 1–12, 2010.

[14] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, H. V. Simhadri, Scheduling irregular parallel computations on hierarchical caches, in: Proc. 23rd ACM symposium on Parallelism in algorithms and architectures (SPAA), ACM, 355–366, 2011.

[15] R. Cole, V. Ramachandran, Revisiting the Cache Miss Analysis of Multithreaded Algorithms, in: Proc. 10th Latin American Theoretical Informatics (LATIN), vol. 7256 of *LNCS*, Springer-Verlag, 172–183, 2012.

[16] R. Cole, V. Ramachandran, Efficient resource oblivious algorithms for multicores with false sharing, in: Proc. 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS), IEEE Computer Society, 201–214, 2012.

[17] R. Cole, V. Ramachandran, Resource oblivious sorting on multicores, in: Proc. 37th International Colloquium on Automata, Languages and Programming (ICALP), vol. 6198 of *LNCS*, Springer-Verlag, 226–237, 2010.

[18] G. E. Blelloch, P. B. Gibbons, H. V. Simhadri, Low depth cache-oblivious algorithms, in: Proc. 22nd ACM symposium on Parallelism in algorithms and architectures (SPAA), ACM, 189–199, 2010.

[19] B. Chamberlain, D. Callahan, H. Zima, Parallel Programmability and the Chapel Language, International Journal of High Performance Computing Applications 21 (3) (2007) 291–312, ISSN 1094-3420.

[20] J. Hennessy, D. Patterson, A. Arpaci-Dusseau, Computer architecture: a quantitative approach, The Morgan Kaufmann Series in Computer Architecture and Design, Morgan Kaufmann, 2007.

[21] R. E. Ladner, M. J. Fischer, Parallel Prefix Computation, Journal of the ACM 27 (4) (1980) 831–838, ISSN 0004-5411.

[22] J. JáJá, An Introduction to Parallel Algorithms, Addison-Wesley, 1992.

[23] R. J. Lipton, R. E. Tarjan, A separator theorem for planar graphs, SIAM Journal on Applied Mathematics 36 (1979) 177–189.

[24] P. d. l. Torre, C. P. Kruskal, Submachine Locality in the Bulk Synchronous Setting (Extended Abstract), in: Proc. 2nd European Conference on Parallel Processing (Euro-Par), Springer-Verlag, 352–358, 1996.

[25] G. Bilardi, A. Pietracaprina, G. Pucci, Decomposable BSP: A Bandwidth-Latency Model for Parallel and Hierarchical Computation, in: J. Reif, S. Rajasekaran (Eds.), Handbook of Parallel Computing: Models, Algorithms and Applications, CRC Press, 277–315, 2007.

[26] G. Bilardi, C. Fantozzi, A. Pietracaprina, G. Pucci, On the Effectiveness of D-BSP as a Bridging Model of Parallel Computation, in: International Conference on Computational Science (ICCS), vol. 2074 of *LNCS*, Springer-Verlag, 579–588, 2001.

[27] F. Silvestri, Oblivious Computations on Memory and Network Hierarchies, Ph.D. thesis, Department of Information Engineering, University of Padova, `http://paduaresearch.cab.unipd.it/1595`, 2009.

[28] K. T. Herley, Network Obliviousness, in: D. A. Padua (Ed.), Encyclopedia of Parallel Computing, Springer, 1298–1303, 2011.

[29] G. Bilardi, A. Pietracaprina, G. Pucci, M. Scquizzato, F. Silvestri, Network-Oblivious Algorithms, manuscript, 2013.

[30] R. A. Chowdhury, V. Ramachandran, Cache-Oblivious Dynamic Programming, in: Proc. 17th ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, 591–600, 2006.

[31] R. A. Chowdhury, F. Silvestri, B. Blakeley, V. Ramachandran, Oblivious Algorithms for Multicores and Network of Processors, Tech. Rep. 09-19, Univ. of Texas at Austin, `ftp://ftp.cs.utexas.edu/pub/techreports/tr09-19.pdf`, 2009.

[32] D. Irony, S. Toledo, A. Tiskin, Communication lower bounds for distributed-memory matrix multiplication, Journal of Parallel and Distributed Computing 64 (9) (2004) 1017–1026.

[33] R. Cole, U. Vishkin, Deterministic coin tossing with applications to optimal parallel list ranking, Information and Control 70 (1) (1986) 32–53, ISSN 0019-9958.

[34] L. Arge, M. Goodrich, N. Sitchinava, Parallel External Memory Graph Algorithms, in: Proc. 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE Computer Society, 1–17, 2010.

[35] Y. J. Chiang, M. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, J. S. Vitter, External-memory graph algorithms, in: Proc. 6th ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, 139–149, 1995.

[36] F. Y. Chin, J. Lam, I. N. Chen, Efficient parallel algorithms for some graph problems, Communications of the ACM 25 (9) (1982) 659–665.

[37] Blue Waters project, `http://www.ncsa.uiuc.edu/BlueWaters`, 2012.

Appendix

For convenience, we reproduce from [7] the parallel cache-oblivious algorithm I-GEP defined by the function f and the set Σ_f. I-GEP consists of four functions $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ and the initial call is $\mathcal{A}(x, x, x, x)$, where x denotes the input $n \times n$ matrix (n is assumed to be a power of 2). In the pseudocode, we denote the top-left, top-right, bottom-left and bottom-right quadrants of X by X_{11}, X_{12}, X_{21} and X_{22}, respectively (similarly for U, V and W). We use the **parallel** construct to denote recursive calls that are invoked in parallel.

$\mathcal{A}(X, U, V, W)$
Input: $X \equiv U \equiv V \equiv W \equiv x[I, I]$, where I is an interval in $[0, n)$.
Output: execution of all updates in $\Sigma_f \cap T$, where $T = I \times I \times I$.
Space Bound: $S(m) = m^2$, where $m = |I|$.

1: **if** $T \cap \Sigma_f = \emptyset$ **then return**
2: **if** X is a 1×1 matrix **then** $X \leftarrow f(X, U, V, W)$; **return**
3: $\mathcal{A}(X_{11}, U_{11}, V_{11}, W_{11})$
4: **parallel** : $\mathcal{B}(X_{12}, U_{11}, V_{12}, W_{11})$, $\mathcal{C}(X_{21}, U_{21}, V_{11}, W_{11})$
5: $\mathcal{D}(X_{22}, U_{21}, V_{12}, W_{11})$
6: $\mathcal{A}(X_{22}, U_{22}, V_{22}, W_{22})$
7: **parallel** : $\mathcal{B}(X_{21}, U_{22}, V_{21}, W_{22})$, $\mathcal{C}(X_{12}, U_{12}, V_{22}, W_{22})$
8: $\mathcal{D}(X_{11}, U_{12}, V_{21}, W_{22})$

$\mathcal{B}(X, U, V, W)$
Input: $X \equiv V \equiv x[I, J]$ and $U \equiv W \equiv x[I, I]$, where I and J denote disjoint intervals (of the same length) in $[0, n)$.
Output: execution of all updates in $\Sigma_f \cap T$, where $T = I \times J \times I$.
Space Bound: $S(m) = 2m^2$, where $m = |I|$.

1: **if** $T \cap \Sigma_f = \emptyset$ **then return**
2: **if** X is a 1×1 matrix **then** $X \leftarrow f(X, U, V, W)$; **return**
3: **parallel** : $\mathcal{B}(X_{11}, U_{11}, V_{11}, W_{11})$, $\mathcal{B}(X_{12}, U_{11}, V_{12}, W_{11})$
4: **parallel** : $\mathcal{D}(X_{21}, U_{21}, V_{11}, W_{11})$, $\mathcal{D}(X_{22}, U_{21}, V_{12}, W_{11})$
5: **parallel** : $\mathcal{B}(X_{21}, U_{22}, V_{21}, W_{22})$, $\mathcal{B}(X_{22}, U_{22}, V_{22}, W_{22})$
6: **parallel** : $\mathcal{D}(X_{11}, U_{12}, V_{21}, W_{22})$, $\mathcal{D}(X_{12}, U_{12}, V_{22}, W_{22})$

$\mathcal{C}(X, U, V, W)$

Input: $X \equiv U \equiv x[I, J]$ and $V \equiv W \equiv x[J, J]$, where I and J denote disjoint intervals (of the same length) in $[0, n)$.

Output: execution of all updates in $\Sigma_f \cap T$, where $T = I \times J \times J$.

Space Bound: $S(m) = 2m^2$, where $m = |I|$.

1: **if** $T \cap \Sigma_f = \emptyset$ **then return**

2: **if** X is a 1×1 matrix **then** $X \leftarrow f(X, U, V, W)$; **return**

3: **parallel** : $\mathcal{C}(X_{11}, U_{11}, V_{11}, W_{11})$, $\mathcal{C}(X_{21}, U_{21}, V_{11}, W_{11})$

4: **parallel** : $\mathcal{D}(X_{12}, U_{11}, V_{12}, W_{11})$, $\mathcal{D}(X_{22}, U_{21}, V_{12}, W_{11})$

5: **parallel** : $\mathcal{C}(X_{12}, U_{12}, V_{22}, W_{22})$, $\mathcal{C}(X_{22}, U_{22}, V_{22}, W_{22})$

6: **parallel** : $\mathcal{D}(X_{11}, U_{12}, V_{21}, W_{22})$, $\mathcal{D}(X_{21}, U_{22}, V_{21}, W_{22})$

$\mathcal{D}(X, U, V, W)$

Input: $X \equiv x[I, J]$, $U \equiv x[I, K]$, $V \equiv x[K, J]$ and $W \equiv x[K, K]$, where I, J, K denote intervals (of the same length) in $[0, n)$, and $I \cap K = \emptyset$, and $J \cap K = \emptyset$.

Output: execution of all updates in $\Sigma_f \cap T$, with $T = I \times J \times K$.

Space Bound: $S(m) = 4m^2$, where $m = |I|$.

1: **if** $T \cap \Sigma_f = \emptyset$ **then return**

2: **if** X is a 1×1 matrix **then** $X \leftarrow f(X, U, V, W)$; **return**

3: **parallel** : $\mathcal{D}(X_{11}, U_{11}, V_{11}, W_{11})$, $\mathcal{D}(X_{12}, U_{11}, V_{12}, W_{11})$,
$\mathcal{D}(X_{21}, U_{21}, V_{11}, W_{11})$, $\mathcal{D}(X_{22}, U_{21}, V_{12}, W_{11})$

4: **parallel** : $\mathcal{D}(X_{11}, U_{12}, V_{21}, W_{22})$, $\mathcal{D}(X_{12}, U_{12}, V_{22}, W_{22})$,
$\mathcal{D}(X_{21}, U_{21}, V_{21}, W_{22})$, $\mathcal{D}(X_{22}, U_{22}, V_{22}, W_{22})$