Oblivious Algorithms for Multicores and Network of Processors [⋆]

Rezaul Alam Chowdhury[1], Francesco Silvestri[2], Brandon Blakeley[1], and Vijaya Ramachandran[1]

[1] Department of Computer Sciences, University of Texas, Austin, TX 78712, USA,
{shaikat,blakeley,vlr}@cs.utexas.edu
[2] Department of Information Engineering, University of Padova, Padova, Italy,
silvest1@dei.unipd.it

UTCS Technical Report TR-09-19

July 22, 2009

Abstract. We address the design of parallel algorithms that are oblivious to machine parameters for two dominant machine configurations: the chip multiprocessor (or multicore) and the network of processors. First, and of independent interest, we propose HM, a hierarchical multi-level caching model for multicores, and we propose a multicore-oblivious approach to algorithms and schedulers for HM. We instantiate this approach with provably efficient multicore-oblivious algorithms for matrix and prefix sum computations, FFT, the Gaussian Elimination paradigm (which represents an important class of computations including Floyd-Warshall's all-pairs shortest paths, Gaussian Elimination and LU decomposition without pivoting), sorting, list ranking, Euler tours and connected components.
We then use the network oblivious framework proposed earlier as an oblivious framework for a network of processors, and we present provably efficient network-oblivious algorithms for sorting, the Gaussian Elimination paradigm, list ranking, Euler tours and connected components. Many of these network-oblivious algorithms perform efficiently also when executed on the Decomposable-BSP.

1 Introduction

The cache-oblivious framework [20] has provided a convenient and general-purpose approach to developing algorithms that perform efficiently in a microprocessor with a single core and a cache hierarchy. 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 (assuming a good cache replacement policy is used). In concept, oblivious algorithms can be defined on other models, in particular on those that represent parallel computing platforms. The first attempt in this direction appears in [7], where the authors propose the *network-oblivious framework* (*NOF*) for parallel distributed-memory computers interconnected by a fixed network.

We are now moving into a new parallel platform as we enter the multicore era: *chip multicores* (*CMP*) are the current default for microprocessors with 2 and 4 cores already on most desktops, and the number of cores is expected to increase dramatically for the foreseeable future. Multicores represent a paradigm shift in general-purpose computing, away from the von Neumann model that has served as a simple and effective model for general-purpose algorithm design for the past several decades. In contrast to the von Neumann model of a single processor that executes unit-cost steps with unit-cost access to data in memory, a typical multicore is a collection of processors/cores on a chip communicating through a cache hierarchy under a shared memory. Thus efficient algorithms for multicores must address both *caching issues* and shared-memory *parallelism*.

With multicore becoming the default desktop configuration, there is a pressing need to develop efficient and portable code for this computing environment. Building on a 3-level multicore caching

[⋆] This work was supported in part by NSF Grant CCF-0514876 and NSF CCF-0850775; the third author was also supported by EU/IST Project "AEOLUS", and by MIUR of Italy under project "MAINSTREAM". Part of this work was done while the third author was visiting the Department of Computer Sciences, University of Texas, Austin.

model proposed in [8], in this paper we propose a *hierarchical multi-level caching model* (*HM*) for multicores and a *multicore-oblivious framework* (*MOF*) for HM. Multicore-oblivious algorithms make no mention of the number of cores, number of levels in the cache hierarchy, or the cache size or block size at any level in the multicore. However, the algorithm is provided with a small set of instructions that it can use to provide advice to the run-time scheduler on how to schedule the parallel tasks. Using this framework we present several multicore-oblivious algorithms for fundamental problems, including prefix sums, matrix computations, Fast Fourier Transform (FFT), sorting, list ranking (LR), some graph problems, and I-GEP, which solves important applications of the Gaussian Elimination Paradigm [13].

Building on the multicore-oblivious algorithms we also develop efficient network-oblivious algorithms for list ranking, some graph problems and I-GEP. In order to achieve optimal communication, network-oblivious algorithms are somewhat modified from their multicore counterparts due to the different models in which algorithms are specified: a shared-memory model for multicore-oblivious algorithms, a distributed-memory model for the network-oblivious ones. Furthermore, we provide an improvement of the network-oblivious sorting algorithm based on ColumnSort given in [7].

1.1 Organization of the Paper

In Section 2 we present HM, a hierarchical caching multicore model, generalizing the 3-level model in [8]. We propose the notion of a multicore-oblivious algorithm, in which the algorithm is specified at the highest level of parallelism and makes no mention of machine parameters, but is allowed to include certain simple 'hints' to the run-time scheduler. We present provably efficient multicore-oblivious algorithms for MT, MM, and FFT.

In Section 4 we quickly review the network-oblivious framework, describe a network-oblivious algorithm for prefix sums and present an improved version of the network-oblivious algorithm for sorting based on ColumnSort given in [7].

In Section 5, we provide some background on GEP for completeness and present a provably efficient MO and NO algorithms for the important applications of GEP solved by I-GEP: Gaussian elimination and LU-decomposition without pivoting and Floyd-Warshall's all-pairs shortest paths in edge-weighted graphs. In order to achieve optimal communication in the network model, the network-oblivious algorithm, named *N-GEP*, is modified from I-GEP. This modified algorithm however, maps back as an HM algorithm with the same performance bounds as I-GEP.

In Section 7, we present efficient MO and NO algorithms for the list ranking problem and some graph problems, including Euler path and connected components.

Finally, in Section 8 we give some final remarks and present a summarizing table which reports the complexities of MO and NO algorithms described in this paper.

2 HM Model and Tiling Algorithms

In its simplest form, a multicore is modeled as a collection of processing elements or *cores*, each with a finite cache (a *private L_1 cache*), and with an arbitrarily large main memory that contains all of the data. Several parallel, cache-efficient algorithms on this type of multicore model have been presented, e.g., for B-trees in [4], for matrix multiplication and other problems in [21], for the Gaussian elimination paradigm (I-GEP and C-GEP) [13], and for sorting in [2, 10, 15]. A complementary model in which there are no private caches but instead a single shared cache of finite size below an arbitrarily large shared memory is considered in [9] and for I-GEP and C-GEP in [13].

In this section we present a hierarchical caching model for multicores (the *HM model*) and we present optimal HM implementations of three general classes of dynamic programming algorithms

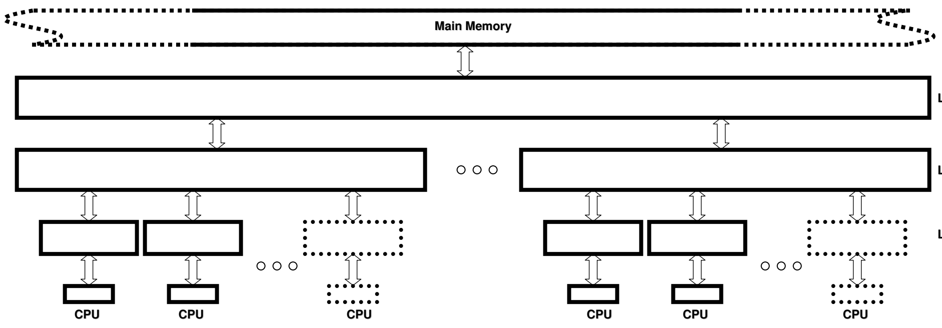

Fig. 1. The HM model illustrated for $h = 4$.

using *tiling sequences* [14]. These algorithms, however, need to specify the HM parameters in their tiling sequences in order to exploit good performance and hence they are not oblivious to the machine parameters. In Section 3 we introduce the notion of *multicore-obliviousness* and the rest of the paper presents efficient multicore-oblivious algorithms for a wide variety of problems.

2.1 The HM Model

Since multicores are evolving towards a hierarchy of caches, a 3-level multicore model was introduced in [8], and algorithms for this model are given in [8, 14]. In this section, we describe the *HM model*, a natural hierarchical generalization of the 3-level model to any number of cache levels $h \geq 3$. (This generalized model is briefly mentioned in [14].) A somewhat different hierarchical model for multicore computing is given in [32], using gap and latency parameters instead of cache misses as the cost measure. More recently, a model similar to ours is mentioned in [10].

The hierarchical multi-level multicore model (or HM model) with h levels consists of a collection of cores P_i, $1 \leq i \leq p$, with a a hierarchy of caches of finite but increasing sizes at levels 1 up to $h-1$, that are successively shared by larger groups of cores. At level h we have a shared memory that is arbitrarily large. For $1 \leq i \leq h-1$, a level i is denoted by L_i, and there are q_i of them. The size of each L_i is C_i, the block-size at level i is B_i, and the number of successive level-$(i-1)$ caches that share a given level-i cache is p_i. By p'_i we denote the number of cores subtended by any L_i, i.e., $p'_i = p/q_i = \prod_{j=1}^{i} p_j$. In the default model we assume that $p_h = 1$, i.e., there is a single cache of finite size C_{h-1} at level $h-1$ so that the top two levels $h-1$ and h represent a possible sequential cache hierarchy at the highest level (see, e.g., [20] and references therein). Also, in the default model, we assume that $p_1 = 1$, i.e., each core has a private cache at the first level. For levels $2 \leq i \leq h-1$ we assume $p_i \geq 2$. Clearly, $\prod_{i=1}^{h} p_i = p = q_1$.

We assume that an ith level cache is at least a constant factor larger than the combined sizes of level $(i-1)$ caches, i.e., $C_i \geq c \cdot p_i \cdot C_{i-1}$ for all i, for a suitable constant $c > 1$. Similar to the sequential cache hierarchy we assume the inclusion property, namely that all elements in each cache at level i are also present in the cache it shares at level $i+1$. We assume an optimal cache replacement policy is used at every cache. For many of the results in this paper a simple generalization of LRU to multi-level caches will suffice, namely the block evicted at a cache λ_i to make room for a new block is the one with entries that were least recently used among those blocks not contained in lower level caches that share λ_i.

The performance of an HM algorithm is determined by its *parallel complexity*, which is the number of parallel steps executed (assuming all cores run at the same rate), and the *cache complexity*, which is specified by the maximum number of block transfers into and out of any single cache at each level i, for $1 \leq i \leq h-1$. We will sometime use the *total* cache complexity, which is the total number of block transfers into and out of all caches at each given level i; this measure is useful when comparing to the sequential cache complexity.

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 the updates that are performed in parallel in our algorithms are always on disjoint sets of data, and hence coherence is never invoked. We specify parallelism by parallel *for* loops and forking and joining through recursive calls.

The multicore model in [8] is the HM model with $h = 3$: it has p cores, p caches of size C_1 at level 1, one cache of size C_2 at level 2, and $p_2 = p$. As required in the default HM model, it has $p_1 = 1$, and at the highest level 3 it has $p_3 = 1$ and size of that memory $C_3 = \infty$.

As noted in [8] for the 3-level multicore model, there is an inherent tension between cache-efficient scheduling for private L_1 caches, and that for a single shared L_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 very fine-grained schedule where all cores work on the portion of the data present in the shared L_2 cache is effective.

Recently, multicore algorithms for sorting are given in [2, 10, 15]. All three are for the simple multicore with private caches, but the latter two algorithms (in [10, 15]) claim fairly good performance for a multi-level cache hierarchy as well. 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 simply assigning to each core, a proportionate slice of the level-i cache above it, for each i. With this mechanism, one can invoke properties and results of the sequential cache-oblivious model to extend the results for the L_1 cache to all levels of the cache hierarchy. In contrast to this, our focus is in algorithms that fully exploit the higher level caches as was achieved for the 3-level case in [8, 14], rather than just a small fraction of the best possible.

2.2 Tiled HM Algorithms

In this section we present some provably optimal implementations of DP algorithms in the HM model by generalizing the results in [14] (which were for the 3-level model) for the following three large classes of algorithms:

1. *LDDP (Local Dependency DP) problems* that include Longest Common Subsequence (or LCS), and several sequence alignment problems of importance in bioinformatics and string matching.
2. *GEP (Gaussian Elimination Paradigm)*, which includes the Floyd-Warshall DP for all-pairs shortest paths in graphs as well as several other important problems such as LU-decomposition and Gaussian elimination without pivoting and matrix multiplication.
3. *Parenthesis problem*, which includes DPs for RNA secondary structure prediction, optimal matrix chain multiplication, construction of optimal binary search trees, and optimal polynomial triangulation.

In [14], *tiled parallel algorithms* for each of these three problem classes were presented, which gave excellent parallel and cache-efficient results for the 3-level multicore model (as well as its predecessors, which we named as D-CMP and S-CMP). We now extend this tiling strategy to the HM model.

In each of the tiled parallel algorithms presented in [14], there is a recursive algorithm where the *tiling parameter* is specified at each level of recursion. The tiling parameter is a non-negative integer (that could depend on input size, size of caches and number of cores), and is associated with a parallel (recursive) execution of each subproblem at that level of recursion. In [14], after specifying a general tiled parallel algorithm, it is shown that simple settings of the tiling parameters give the optimal performance for 3-level multicore as well as for D-CMP, S-CMP and sequential executions. In recursion levels where no parallelism is exposed, the tiling parameter is set to 2, and the execution is sequential. The reader is referred to [14] for further details on these algorithms.

For the HM model, we can start with the same parallel tiled algorithm but we need to specify the tiling recursion level as well as the tiling parameter for each level of the memory hierarchy. So, rather than just 1 ($= h-2$) nontrivial tiling parameter, the algorithm will have $h-2$ nontrivial recursion levels with their associated tiling values. These levels are determined as follows: For each level-i memory (of size C_i) we use the recursion level r_i where the size of each subproblem is $p_i \cdot C_{i-1}$, and we use tiling parameter p_i.

For example, for LCS, the first algorithm considered in [14], the smallest level of recursion $r_{h-1} = \log \frac{n}{p_{h-1} \cdot C_{h-2}}$, and the tiling parameter is p_{h-1}. For $h-1 > i \geq 2$, $r_i = r_{i+1} + \log \frac{C_i}{p_i \cdot C_{i-1}}$, and the tiling parameter is p_i. At recursion level r_2 we fork out the parallel computation on the p_2 base (core) processors in the HM model, and the remaining levels of recursion are executed sequentially on the assigned core. Since all of these cores are computing in parallel according to the tiled parallel algorithm, the parallel time bound remains $O((n^2/p) + n)$, where p is the total number of processors ($p = \Pi_{i=2}^{h-1} p_i$). Also, the size of each subproblem assigned to a memory at level i is of size C_i and hence the sequential cache complexity is achieved at every level of the memory hierarchy.

The tiling parameters for HM for the 3 classes of algorithms considered in [14] are given in Table 1. The associated HM algorithms have total cache complexity at every cache level that match the sequential cache complexity, and also achieve optimal speed-up up to the critical pathlength of the computation. These results can be established in a fairly straightforward way by generalizing the proofs in [14].

Problem	I/O	Previous I_∞	Our I_∞	Tiling Parameters for Our HM Algorithms
LCS & PA	$\mathcal{O}\left(\frac{n^2}{MB}\right)$	n^2 (seq.)	$\mathcal{O}(n)$	$t[r_i] = p_i,\ h-1 \geq i \geq 2$, $r_i =$ recursion level where subproblem size is $p_i \cdot C_{i-1}$ & $t[d] = 2$ with no parallelism if $d \neq$ any r_i
Median	$\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$	n^3 (seq.)	$\mathcal{O}(n)$	$t[r_i] = \sqrt{p_i}, h-1 \geq i \geq 2$, $r_i =$ recursion level where subproblem size is $p_i \cdot C_{i-1}$ & $t[d] = 2$ with no parallelism if $d \neq$ any r_i
Parenthesis		*		
GEP		$\mathcal{O}\left(n \log^2 n\right)$ [13, 8]		
RNA-SP	$\mathcal{O}\left(\frac{n^4}{B\sqrt{M}}\right)$	n^4 (seq.)	$O(n)$	–

Table 1. Table of our dynamic programming results for the HM model. Here the 'I/O' column lists the sequential cache-oblivious bound, and the I_∞ column lists the number of parallel steps in a work-optimal parallel algorithm whose *total cache complexity matches the sequential bound at every level of the cache hierarchy.* The tiling parameters are explained briefly in the text of the paper, and in more detail in [14].

The results summarized in Table 1 represent tiled multi-level algorithms that make very effective use of both parallelism and the cache hierarchy. However, these algorithms use the number of cores and the sizes of the caches to set the tiling parameters. In the next section we introduce our notion of *multicore-obliviousness*, and we present multicore-oblivious algorithms with excellent parallelism and cache efficiency for a wide variety of problems. Some of these algorithms can be viewed as extensions of results in [8] for the 3-level multicore model, while several others are new results. Among the multicore-oblivious algorithms we present is an algorithm for GEP in Section 5, for which Table 1 lists tiling parameters for a tiled HM algorithm.

3 Multicore-Oblivious Algorithms on the HM Model

For the effective use of multicores we need to have not only algorithms that are capable of running efficiently on them, but also run-time schedulers that can effectively extract this efficient performance. Also, for portability reasons, it is desirable to have a *multicore-oblivious (MO)* algorithm, i.e, an algorithm that does not use specific values for multicore parameters such as number of cores, number of levels of caches or their sizes, block sizes, etc., yet performs efficiently across a wide variety of multicores. This is in contrast to the tiled algorithms described in the previous section.

To address this challenge, we propose some simple enhancements to HM algorithms. These enhancements are 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. These hints can be either in the form of information known at compile time, or the result of a computation at run-time that is tagged as being meant for the run-time scheduler. In a multicore-oblivious algorithm, these hints must be independent of multicore parameters such as number of cores, number of cache levels, cache or block sizes.

Our initiative to introduce these special instructions for the run-time scheduler is in the spirit of the recent trend towards 'multiresolution' languages [11], 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. Our design can be viewed as one approach to extracting good performance on multi-level multicores from certain types of algorithms through a small and controlled set of such enhancements.

The multicore-oblivious algorithms we present use two main types of scheduler hints, and a third one that combines these two. Of these CGC is useful on scheduling computations involving parallel for loops, and SB and CGC$\Rightarrow$SB are useful for algorithms that recursively spawn parallel tasks. In order to simplify the exposition of these hints we now introduce some terms and notations. Let us define the *shadow* of a level i cache λ_i as those p'_i cores that share λ_i as their level i cache. We say that a task τ is *anchored* at a cache λ_i (or is *assigned* to λ_i) provided it is at the smallest level i such that C_i is large enough to meet the space requirement of τ, and all subtasks generated by τ are executed by cores under the shadow of λ_i.
A technical point: In order to avoid the cache misses caused by ping-ponging of cache blocks between private caches, the scheduler respects block boundaries whenever feasible during the decomposition and distribution of tasks. Ping-ponging arises when writes to a block by any given core are interleaved by reads/writes by other cores: each write removes the block from every other core's L_1 cache if it already contains the block, and any such core must re-fetch the block if it wants to perform subsequent read/write operations on the block. This results in an excessive number of block transfers. We also assume this hardware block invalidation mechanism is supported at higher level caches at size B_1.

3.1 Coarse-Grained Contiguous (CGC) Scheduling

The *coarse-grained contiguous (CGC) scheduling* is a very simple scheduler hint that could be considered a default option for the HM scheduler. This is an effective schedule for HM algorithms whose primary memory access pattern is simple scans of the input data. It simply distributes an ordered collection of parallel tasks in contiguous chunks among a sequence of contiguous cores. Suppose τ is a task anchored at some cache λ, and has a parallel loop with the index variable k running from 1 to t. If there are $p(\lambda)$ cores under the shadow of λ, under CGC scheduling the values taken by k are decomposed into $\lfloor t/p(\lambda) \rfloor$ contiguous segments, and the j-th such segment is assigned to the j-th core from the left under the shadow of λ. It is ensured that the length of input data scanned by each segment is at least B_1 even if that requires keeping some of the $p(\lambda)$ cores idle.

We now give two important examples of algorithms scheduled with CGC.

Prefix Sums. We now describe an optimal multicore oblivious algorithm solving prefix sums. 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 ith partial sum s_i of such a sequence to be $s_i = x_0 * x_1 * \cdots * x_{i-1}$. Then the prefix sums of such a sequence, originally described by Ladner and Fischer (cite), is the n partial sums of the sequence.

We show that the recursive algorithm due to Ladner and Fischer [26] originally posed in the parallel context has optimal cache efficiency when the coarse grained contiguous scheduler is used. 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. We now present the algorithm formally and then analyze its complexity.

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

Fig. 2. Multicore Oblivious Prefix Sums

Theorem 1. *When executed on a h-level HM model with p cores, MO–PS on an input of size n terminates in $\mathcal{O}(n/p + B_1 \log p)$ parallel steps, and incurs $\mathcal{Q}_i(n) = \mathcal{O}(n/(q_i B_i) + \log q_i)$ caches misses at each of the q_i caches in level i of the hierarchy.*

Proof. The correctness of this algorithm can be proved by induction on k, where the size of the input $n = 2^k$ (see [25] for the details). We now analyze the complexity. The coarse grained contiguous scheduler assigns contiguous blocks of elements to contiguous processors and requires that at each level i of the cache hierarchy, at most one nonempty level i cache is given less than B_i units of data, even if this requires some processors to idle. In lines 2, 6, and 7, we have $\mathcal{O}(n)$ summations to distribute across the $\min(p, \lceil \frac{n}{B_1} \rceil)$ processors which the CGC scheduler assigns tasks to. As each recursive stage decreases the size of the input by a factor of two, the number of processors the CGC scheduler assigns tasks to decreases by a factor of two as well whenever $n \leq pB_1$. We thus derive the following time complexity:

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

This recurrence solves to $\mathcal{O}(n/p + B_1 \log p)$. Next, we analyze the level i cache complexity of this algorithm. Similarly, the number of level i caches across which the input is distributed decreases in proportion to n. We thus derive the following recurrence relation:

$$\mathcal{Q}_i(n, q_i) = \begin{cases} \mathcal{Q}_i(n/2, q_i) + \mathcal{O}(\frac{n}{q_iB_i}) & q_iB_i < n \\ \mathcal{Q}_i(n/2, q_i/2) + \mathcal{O}(\frac{n}{q_iB_i}) & B_i < n \leq q_iB_i \\ \mathcal{O}(1) & n \leq B_i \end{cases}$$

This recurrence solves to $\mathcal{O}(\frac{n}{q_iB_i} + \log q_i)$. Thus, the cache and time complexity of this algorithm match the bounds stated. □

Furthermore, 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.

Matrix Transposition (MT). Given an ordered pair of $\log n$-bit indices (i, j), we define $\beta(i, j)$ as the ordered pair of integers (i', j') obtained by *bitwise interleaving* of the binary representations of i and j. In other words, if $i = i_{r-1}i_{r-2} \cdots i_0$ and $j = j_{r-1}j_{r-2} \cdots j_0$, then $i' = i_{r-1}j_{r-1}i_{r-2}j_{r-2} \cdots i_{r/2}j_{r/2}$ and $j' = i_{(r-2)/2}j_{(r-2)/2} \cdots i_0j_0$ (here $r = \log n$ and we have assumed for convenience that r is even; the case when r is odd is handled in a natural way). We let β^{-1} be the inverse of β.

A multicore-oblivious HM algorithm for matrix transposition is given in Fig. 3 which uses the CGC scheduler hint. This algorithm is based on the matrix transposition algorithm for the network-oblivious model in [7]. We assume that $\beta(i, j)$ and $\beta^{-1}(i, j)$ can be computed in constant time by the hardware.

```
MO-MT($A$, $n$)
**Input:** the input is an $n \times n$ matrix $A$ in row-major;
**Output:** the output is the transpose $A^T$, again in row-major.
**Comment.** $\beta(i, j)$ is defined in the text.
 1: **[CGC] pfor** $1 \leq i \leq n,\ 1 \leq j \leq n$ **do**  $I[i, j] := A[\beta^{-1}(i, j)]$ **end pfor**;
 2: **[CGC] pfor** $1 \leq i \leq n,\ 1 \leq j \leq n$ **do**  $A^T[i, j] := I[\beta(j, i)]$ **end pfor**;
```

Fig. 3. MO-MT: a multicore-oblivious algorithm for matrix transposition.

Note that the parallelization of the recursive optimal cache-oblivious algorithm for matrix transposition in [20] would take $\Theta(\log n)$ parallel steps and hence would not achieve optimal *constant* critical pathlength of our algorithm.

Theorem 2. MO-MT *correctly transposes the matrix A in an n^2 core HM model in $O(n^2/p + B_1)$ parallel steps with $O\left(n^2/(q_iB_i) + B_i\right)$ cache misses at each of the q_i caches in the ith level of the cache hierarchy, for all i, $1 \leq i \leq h - 1$, assuming that all caches are tall, and all $B_i \leq n$.*

Proof. Consider a level-l cache, and let $B_l = B$. Let S be a contiguous sequence of B elements written into I by a core q, starting with position (i_1, j_1) and ending in position (i_2, j_2). Since $B < n$, the bit representation of all $(i, j) \in S$ will have at most two different bit patterns in the most significant $2 \log n - \log B$ positions (and only one bit pattern in these positions if the indices don't wrap around to two rows.)

Consider the block S_A that contains $A[\beta^{-1}(i_1, j_1)]$. Here again, the bit representation of all $(i, j) \in S_A$ will have at most two different bit patterns in the most significant $2 \log n - \log B$ positions, and their bit-interleaved addresses will have at most 2 different bit patterns in the most significant $2 \log n - 2 \log B$ bit positions, and will differ in only the lowest $2 \log B$ bit positions otherwise. Hence the elements in block S_A of array A will lie in within distance B^2 of one another in each of at most two sequences of positions in I.

Since $I[i,j]$ is filled in row-major sequence, starting with position (i_1, j_1), the blocks containing the corresponding elements in array A are brought into cache. Although up to B different blocks may be brought in to fill the first block of elements in $I[i,j]$, starting at $\beta(i_1, j_1)$, all of the elements in each of these B blocks are meant to be located in positions in I that are in at most two different groups of elements that are within distance B^2 of one another in row major order in I. Since the cache is assumed to have size $\Omega(B^2)$ these elements will be available in the cache when they need to be written into I. Hence the write into I has the same cache complexity, to within a constant factor, as a scan, except that up to B_l^2 elements remain unused at the end in a level-l cache, for each l. Hence the result follows for step 1. The analysis for step 2 is similar.

Finally, if $n^2 < p \cdot B_1$, the scheduler will assign a block of data to $\lceil n^2/p \rceil)$ cores in order to respect the L_1 block boundaries. Hence in this situation, the parallel time is $O(B_1)$. Thus the overall parallel time is bounded by $O(n^2/p \ + \ B_1)$. □

3.2 Space-Bound (SB) Scheduling

In *space-bound (SB) scheduling* the algorithm supplies an upper bound on the space used by each task that is forked during the computation. To see why such a scheduler hint is helpful, consider a core P that is executing a task τ whose computation has a space upper bound of $s(\tau)$, and let i be the smallest level in the cache hierarchy for which $s(\tau) \leq C_i$. If $i \leq h-1$, and λ_i is the level i cache above core P, then as long as all tasks forked during execution of task τ are assigned to cores that also share cache λ_i (i.e., that lie in λ_i's shadow), the only cache misses incurred at level i during execution of τ are those needed to read in the initial input and write out the final output.

For each level i cache λ, the SB scheduler maintains a queue $Q(\lambda)$ for tasks with space bound not exceeding C_i which are to be executed under the shadow of λ. When the current task assigned to λ completes, the first task τ in Q is dequeued and executed while anchored to λ. When τ forks a task τ', this task is assigned to the least loaded cache under λ's shadow at level $i-1$ if $s(\tau') \leq C_{i-1}$, and it is enqueued in Q otherwise.

In order to keep the total number of cache misses under control, the SB scheduler makes sure that if any task τ with $s(\tau) \leq B_1$ is assigned to some core P for execution then all its descendant subtasks are also assigned to P (i.e., anchored at the same level i cache as τ). The SB scheduler should also use some strategy to minimize the ping-ponging of shared blocks between caches. One approach could be as follows which avoids ping-ponging by tasks with output arrays larger than $2B_1$. When any task τ anchored at some level $i > 1$ generates a set of subtasks migrating to level $i-1$, each of them makes a local copy (from its parent cache at level i to its own cache at level $i-1$) of each array it needs to modify or generate, and works on them. When each of these subtasks completes execution, it first copies the first half of each array it has updated/generated from its local cache to the parent cache. Then it waits until all other subtasks of the set do the same. After this synchronization point, all subtasks in the set copy the second half of the arrays to the parent cache. Observe that provided the length of each output array copied back by the subtasks is at least $2B_1$, this approach ensures that no two subtasks write to the same block at the same time.

We apply the SB scheduler to recursively forking tasks where a *constant* number of tasks (typically 2) are generated at each fork, each with a space bound that is a constant factor smaller than that of the forking task. We expect that the general space-bounded strategy is likely to have wide applicability in multicore scheduling, and can be configured in ways other than the SB and CGC⇒SB schedulers we use.

The following algorithm for matrix multiplication (MM) is a simple example of SB scheduling. Another application of the SB scheduler, I-GEP, is presented in Section 5.

Matrix Multiplication (MM). In Fig. 4 we provide an algorithm (MO-MM) for matrix multiplication in the HM model based on the SB scheduling strategy. The algorithm [3] is slightly modified from the original cache-oblivious matrix multiplication algorithm in [20]. While both algorithm correctly compute the result (for the case when addition is associative and commutative) we choose to use this version since a similar algorithm has optimal communication in the network setting [7].

```
MO-MM($A, B;\ C; n$)
Input: two $n \times n$ matrices $A$ and $B$;
Output: the product matrix $C$, initially all zeros.
Space Bound: $S(n) = 3n^2$.
 1: if $n = 1$ then return $C = C + A \cdot B$;
 2: [SB] in parallel: MO-MM($A_{00}, B_{00};\ C_{00}; n/2$), MO-MM($A_{01}, B_{11};\ C_{01},\ n/2$)
                      MO-MM($A_{10}, B_{01};\ C_{11};\ n/2$), MO-MM($A_{11}, B_{10};\ C_{10},\ n/2$)
 3: [SB] in parallel: MO-MM($A_{01}, B_{10};\ C_{00},\ n/2$), MO-MM($A_{00}, B_{01};\ C_{01},\ n/2$),
                      MO-MM($A_{11}, B_{11};\ C_{11},\ n/2$), MO-MM($A_{10}, B_{00};\ C_{10},\ n/2$);
```

Fig. 4. MO-MM: a multicore-oblivious algorithm for matrix multiplication.

The following theorem provides the parallel running time and cache complexity of MO-MM under the space-bound scheduler described above.

Theorem 3. *Consider the h-level HM model with p cores, where all caches in the hierarchy are tall (i.e., $C_i = \Omega(B_i^2)$, $1 \le i \le h-1$), and $C_i > 2 \cdot p_i \cdot C_{i-1}$ for $i \in [2, h-1]$. When executed under the space-bound scheduler,* MO-MM *terminates in $\mathcal{T}(n) = \mathcal{O}\left(n^3/p + nB_1 + B_1^{3/2}\right)$ parallel steps, while incurring $\mathcal{Q}_i(n) = \mathcal{O}\left(n^3/\left(q_i B_i \sqrt{C_i}\right) + \left(n^2\sqrt{C_{h-1}}\right)/\left(q_i B_i \sqrt{C_i}\right) + C_i/B_i + \sqrt{B_1}\right)$ cache misses at each of the q_i caches in level i of the hierarchy. For $n^2 \ge C_{h-1}$, the overall parallel running time and the number of cache misses at each level i cache reduce to $\mathcal{O}\left(n^3/p\right)$ and $\mathcal{O}\left(n^3/\left(q_i B_i \sqrt{C_i}\right)\right)$, respectively.*

Proof. (Sketch) Recall that if a task τ has space bound $s(\tau) \le B_1$, the SB scheduler will assign τ and all its descendant subtasks to the same core. Hence, if $n^2 = \mathcal{O}(B_1)$, MO-MM will take $\mathcal{O}\left(B_1^{3/2}\right)$ time to terminate. Otherwise, we can view the execution of MO-MM on $n \times n$ input matrices as consisting of $n/\sqrt{B_1}$ phases to be executed in sequence with each phase containing n^2/B_1 parallel tasks with space bound $\Theta(B_1)$ each. Each such task will be executed exclusively on a single core in $\mathcal{O}\left(B_1^{3/2}\right)$ time. Hence, the number of parallel steps is $\mathcal{O}\left(\left(n/\sqrt{B_1}\right) \cdot B_1^{3/2} \cdot \left(1 + \left(n^2/B_1\right)/p\right)\right) = \mathcal{O}\left(n^3/p + nB_1\right)$. The claimed bound follows from the two bounds derived above.

In order to derive the cache complexity of MO-MM, we will consider the following three cases based on the space bound σ of each migrated task τ anchored at each level i cache λ: (i) $\sigma = C_i$, (ii) $C_i > \sigma > B_1$, and (iii) $\sigma \le B_1$. For case (i), observe that $\mathcal{O}\left(1 + \left(n/\sqrt{C_i}\right)^3/q_i\right)$ tasks with space bound C_i will be anchored at λ, and each of them will incur $\mathcal{O}\left(\sqrt{C_i} + C_i/B_i\right)$ cache misses at λ. Thus $\mathcal{Q}_i(n) = \mathcal{O}\left(\left(1 + \left(n/\sqrt{C_i}\right)^3/q_i\right) \cdot \left(\sqrt{C_i} + C_i/B_i\right)\right) = \mathcal{O}\left(n^3/\left(q_i B_i \sqrt{C_i}\right) + C_i/B_i\right)$ in this case. In case (ii), $\mathcal{O}\left(1 + 8^{h-i-1}/q_i\right)$ tasks each with space bound $n^2/4^{h-i-1} < C_i$ will be anchored at λ, which leads to $\mathcal{Q}_i(n) = \mathcal{O}\left(n/2^{h-i-1} + \left(n^2 B_i\right)/4^{h-i-1} + (nC_{h-1})/(q_i C_i) + \left(n^2\sqrt{C_{h-1}}\right)/\left(q_i B_i \sqrt{C_i}\right)\right)$ $= \mathcal{O}\left(\left(n^2\sqrt{C_{h-1}}\right)/\left(q_i B_i \sqrt{C_i}\right) + C_i/B_i\right)$ since $C_i > 2 \cdot p_i \cdot C_{i-1} \ge 4 \cdot C_{i-1}$ for $i \in [2, h-1]$. Similarly, in case (iii), $\mathcal{Q}_i(n) = \mathcal{O}\left(\sqrt{B_1} + B_1/B_i + (nC_{h-1})/(q_i C_i) + \left(n^2\sqrt{C_{h-1}}\right)/\left(q_i B_i \sqrt{C_i}\right)\right) = \mathcal{O}\left(\left(n^2\sqrt{C_{h-1}}\right)/\left(q_i B_i \sqrt{C_i}\right) + B_1/B_i + \sqrt{B_1}\right)$. The bound claimed in the theorem follows by combining the three bounds derived above. □

[3] In the paper we denote the quadrants of a matrix A as: A_{00} (top-left), A_{01} (top-right), A_{10} (bottom-left), A_{11} (bottom-right).

3.3 CGC on SB (CGC⇒SB) Scheduling

This scheduler is useful in algorithms that fork parallel tasks recursively when there is a large number of parallel tasks created at forks. It can also be used when there is need to generate a sufficient number of tasks through forking at a task τ anchored at a given cache λ before assigning them to caches at the next lower level in λ's shadow in order to exploit the parallelism fully. Informally under CGC⇒SB, a collection of subtasks forked from τ are distributed evenly across caches at a suitable lower level where the cache size is sufficiently large 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 at a level k cache λ that recursively spawns parallel tasks, and consider the first level of recursion when $m \geq p_k$ subtasks are generated. We assume that all generated subtasks have the same space bound σ, to within a constant factor. The CGC⇒SB scheduler finds the smallest level i with $C_i \geq \sigma$, and the smallest level j such that there are no more than m level j caches under the shadow of λ. Then it uses CGC to distribute the subtasks evenly across the caches in level $\max(i, j)$ under the shadow of λ.

Multiple tasks can be anchored at λ for simultaneous execution provided the total space needed by all such tasks is upper bounded by C_i. When this happens, each is given a proportionate number of cores. In our applications, all such (active) tasks are of the same size, to within a constant factor.

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 $Y[i] = \sum_{0 \leq j < n} X[j+1] \cdot \omega_n^{-ij}$ for $1 \leq i \leq n$, and $\omega_n = e^{2\pi\sqrt{-1}/n}$ [19]. In Figure 5, we present MO-FFT, the well-known six-step variant [3, 33] of the Cooley-Tookey FFT algorithm [18] modified for efficient multicore-oblivious execution, and obtained by exposing the parallelism in the cache-oblivious FFT algorithm in [20]. For simplicity of exposition we assume that n is a power of 2. The algorithm is recursive, and uses the multicore-oblivious matrix transposition algorithm MO-MT as a subroutine. For some factorization $n_1 \times n_2$ of n, the six-step algorithm first computes n_2 transforms of size n_1 recursively, then multiplies the results by *twiddle factors* [19], followed by the recursive computation of n_1 transforms of size n_2 each. Matrix transposition is used to put the inputs to the recursive calls in correct contiguous locations. In order to use MO-MT which transposes square matrices, the $n_1 \times n_2$ matrix initially created from the input vector is transformed into a square matrix by padding it with dummy values. We use two types of scheduling in MO-FFT: CGC and CGC⇒SB.

```
MO-FFT($X, n$)
Input: A vector $X$ of length $n = 2^k$ for some integer $k$.
Output: In-place FFT of $X$ in $X$.
Space Bound: $S(n) = 3n$.
1: if $n$ is a small constant then compute FFT using the direct formula and return end if
2: Let $n_1 = 2^{\lceil \frac{k}{2} \rceil}$ and $n_2 = 2^{\lfloor \frac{k}{2} \rfloor}$ (observe that $n_2 \leq n_1 \leq 2n_2$).
   In the following, let $A$ be an $n_1 \times n_1$ matrix stored in row-major order.
   [CGC] pfor $1 \leq i \leq n_1, 1 \leq j \leq n_2$ do $A[i,j] := X[(i-1) \cdot n_2 + j]$ end pfor
3: [CGC] MO-MT($A, n_1$)
4: [CGC⇒SB] pfor $1 \leq i \leq n_2$ do MO-FFT($A[i, 1 \ldots n_1], n_1$) end pfor
5: [CGC] Multiply each of the first $n$ entries of $A$ by the appropriate twiddle factor [19]
6: [CGC] MO-MT($A, n_1$)
7: [CGC⇒SB] pfor $1 \leq i \leq n_1$ do MO-FFT($A[i, 1 \ldots n_2], n_2$) end pfor
8: [CGC] MO-MT($A, n_1$)
9: [CGC] Copy the first $n$ entries of $A$ into $X$
```

Fig. 5. MO-FFT: multicore-oblivious in-place FFT.

Theorem 4. *When executed on a h-level HM model with p cores,* MO-FFT *on an input of size $n \geq C_{h-1}$ terminates in $\mathcal{T}_p(n) = \mathcal{O}\left((1 + n/p) \log n\right)$ parallel steps, and incurs $\mathcal{Q}_i(n) = \mathcal{O}\left(\left(n/\left(q_i B_i\right)\right) \log_{C_i} n\right)$ cache misses at each of the q_i caches in level i of the hierarchy, provided $C_i = \Omega\left(B_i^2\right)$ (i.e., tall cache).*

Proof. Since the CGC computations in lines 2, 3, 5, 6, 8 and 9 all have constant critical path length, the critical pathlength of MO-FFT is $\mathcal{T}_\infty(n) = \mathcal{T}_\infty(n_1) + \mathcal{T}_\infty(n_2) + \Theta(1) = \mathcal{O}(\log n)$. Hence, the number of parallel steps using p processors $\mathcal{T}_p(n) = \frac{\mathcal{T}_1(n)}{p} + \mathcal{T}_\infty(n) = \mathcal{O}\left((1 + n/p) \log n\right)$ provided a core is not left idling when parallel tasks are available. Since CGC⇒SB executes all taskes anchored at the caches even when there are multiple tasks anchored at a given cache, the number of parallel steps in this computation remains $\mathcal{T}_p(n)$ provided $C_i \geq p_i' B_1$, for each cache level i, since this condition would ensure that each core gets at least one block in the CGC computations at each recursive step on the tasks anchored at caches.

We need to analyze the effect of this scheduling on the cache complexity of at every cache level. For this, consider a level i cache λ_i. Observe that cache-misses according to CGC are incurred in steps 2, 3, 5, 6, 8 and 9, while steps 4 and 7 call MO-FFT recursively with smaller inputs. 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 $\mathcal{O}\left(\left(n/\left(q_i B_i\right)\right)\right)$ 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 $\mathcal{Q}_i(n) = \mathcal{O}\left(\left(n/\left(q_i B_i\right)\right) \log_{C_i} n\right)$. ☐

Sorting. A multicore oblivious algorithm for sorting on the simple multicore model with just private caches is presented in [15]. This algorithm, *Sample Partition Merge Sort (SPMS)*, runs in $O(n \log n)$ sequential time and $O((n/B) \log_C n)$ sequential cache misses (assuming a tall cache of size C with block size B) on an input of length n. It is also shown in [15] that the algorithm has critical pathlength $O(\log n \log\log n)$ and can be scheduled optimally on private caches. The space bound is linear in the size of the input, and of each generated task.

The algorithm SPMS has exactly the same structure as the FFT algorithm given above, except that the CGC steps, which in our FFT algorithm use MO-MT and other constant parallel time computations, instead consist of a constant number of applications of prefix sums and other balanced parallel computations ('BP' computations) that can be scheduled under CGC. Thus a CGC step on a task of size n in the sorting algorithm has $O(\log n)$ critical pathlength. The algorithm SPMS uses these BP computations to decompose an original problem of size n into collection of independent subproblems, each of size at most $\sqrt{n}$. The overall problem of size n is solved by a sequence of two recursive calls to subproblems of size at most $\sqrt{n}$. Althougth individual subproblems may vary in size, there is a balanced task scheduler that ensures the generation of parallel tasks that are within a constant factor of one another in size. Thus the structure of this computation is exactly the same as that for FFT, hence the results for FFT under our scheduler translate to this sorting algorithm with the same scheduler. The parallel time increases from $O(\log n)$ to $O(\log n \log\log n)$ due to the use of prefix sums CGC computations (which has $\log n$ critical pathlength) instead of the constant depth MO-MT used in FFT. This gives us the following result.

Theorem 5. *Consider the SPMS algorithm executed on an an input of size $n \geq C_{h-1}$ on an h-level HM model with p cores, using CGC scheduler hints for prefix sums and BP computations, and CGC⇒SB hints as in FFT. Then, SPMS terminates in $\mathcal{T}_p(n) = \mathcal{O}\left((1 + n/p \log\log n) \log n \log\log n\right)$ parallel steps, and incurs $\mathcal{Q}_i(n) = \mathcal{O}\left(\left(n/\left(q_i B_i\right)\right) \log_{C_i} n\right)$ cache misses at each of the q_i caches in level i of the hierarchy, provided $C_i = \Omega\left(B_i^2\right)$ (i.e., tall cache).*

Finally, we note that if the sorting algorithm is executed on a smaller input size $m \leq q_i \cdot C_i$, then the cache complexity at a level i cache is $\mathcal{O}\left(\left(m/\left(q_i B_i\right)\right) \log_{r_i} m\right)$, where $r_i = \min\{C_i, m/q_i\}$. This fact will be used in the MO list ranking algorithm in Section 7.

Sparse Matrix Dense Vector Multiplication (SpM-DV). We show that when executed under the CGC⇒SB scheduler on the HM model, the separator-based sparse matrix dense vector multiplication algorithm given in [8] (see Figure 6) shows good cache performance, provided the matrix has a support graph ([8], and also see below) with good separators. Let S be a class of graphs closed under the subgraph relation. Then S is said to satisfy a $f(n)$-*edge separator theorem* if there exist constants $\alpha \in [\frac{1}{2}, 1)$ and $\beta > 0$ such that every graph $G = (V, E) \in S$ with $|V| = n$ can be partitioned into two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with $|V_1|, |V_2| \le \alpha n$, $V_1 \cap V_2 = \emptyset$, and $|\{(u, v) \in E | (u \in V_1 \wedge v \in V_2) \vee (u \in V_2 \wedge v \in V_1)\}| \le \beta f(n)$ [28]. The *support graph* of an $n \times n$ matrix A is defined to be the graph $G_A = (V, E)$ with $V = \{1, \ldots, n\}$ and $E = \{(i, j) | A[i, j] \ne 0\}$. We say that A satisfies an $f(n)$-edge separator theorem if G_A satisfies such a theorem.

As in [8], we assume that the rows and columns of matrix A are reordered based on the left to right ordering of leaves in the separator tree T_A of G_A. The separator tree is constructed by applying the separator theorem to the whole 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.

MO-SpM-DV$((A_v, A_0), x; y; k_1, k_2)$

Input: The inputs are 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: A vector y of length n containing the product Ax.

Scheduler Hint: space-bound scheduling.

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
4:         $\langle j, a\rangle := A_v[k]$
5:         $y[k_1] := y[k_1] + a \times x[j]$
6: else
7:     [CGC⇒SB] in parallel: MO-SpM-DV$((A_v, A_0), x; y; k_1, \lfloor \frac{k_1+k_2}{2} \rfloor)$,
8:                           MO-SpM-DV$((A_v, A_0), x; y; \lfloor \frac{k_1+k_2}{2} \rfloor + 1, k_2)$
```

Fig. 6. MO-SpM-DV: multicore-oblivious sparse matrix and dense vector multiplication.

Theorem 6. *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 *terminates in $\mathcal{T}(n) = \mathcal{O}(n/p)$ parallel steps, and incurs $\mathcal{Q}_i(n) = \mathcal{O}\left((n/q_i)\left(1/B_i + 1/C_i^{1-\epsilon}\right)\right)$ cache misses at each of the q_i caches in level i of the hierarchy, provided $n > C_{h-1}$.*

Proof. Sketch 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 $n > C_{h-1} \ge p \cdot C_1 \ge p \cdot B_1$), and that $\Theta(n/(q_i C_i))$ migrated tasks will be anchored at each level i cache λ. Recall that λ will meet the space requirement of each such migrated task τ, but will be too small for the parent of τ. Once τ is anchored at λ all its descendant subtasks will be executed completey 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 migrated tasks at λ. Since the space bound of τ is $s(\tau) = 4m$, 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 incure a cache miss. However, according to the edge seperator theorem, only $\mathcal{O}(m^\epsilon)$ such misses can occur. Observe that $\mathcal{O}(m/B_i)$ 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 $\mathcal{O}(m/B_i + m^\epsilon) = \mathcal{O}(C_i/B_i + C_i{}^\epsilon)$ cache misses. Therefore, $\mathcal{Q}_i(n) = \mathcal{O}(n/(q_iC_i)\cdot(C_i/B_i + C_i{}^\epsilon)) = \mathcal{O}\left((n/q_i)\cdot\left(1/B_i + 1/C_i^{1-\epsilon}\right)\right)$.

Since the scheduler distributes the tasks across cores evenly, and each row of A has at most a constant number of non-zero entries, the $\mathcal{O}(n/p)$ parallel running time of the algorithm follows immediately. □

4 Review of the Network-Obliviousness

A network-oblivious algorithm [7] is an algorithm designed for a network of processing elements (PEs) that refers to no parameters of the machine such as number of processors or the interconnection network parameters. A NO algorithm $\mathcal{A}$ is specified 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 PEs $PE_0, \cdots PE_{N-1}$, each consisting of a CPU and an unbounded local memory. $\mathcal{A}$ consists of a sequence of labeled supersteps: in an i-superstep, for $0 \le i < \log N$, a PE_j can perform operations on locally held data and send words of data to PEs whose numbers share the most significant i bits with j. A superstep ends with a global synchronization.

The communication complexity of $\mathcal{A}$ is defined on the $\mathsf{M}(p,B)$, for $1 \le p \le N$. The $\mathsf{M}(p,B)$ 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 (in words). The *block-degree* of a superstep is defined as the maximum number of blocks sent/received by a processor. For an $\mathsf{M}(p,B)$ algorithm, we define the *communication complexity* as the sum of the block-degrees of its supersteps, and the *computation complexity* as the sum over all supersteps of the maximum number of operations performed by a processor in each superstep. A NO algorithm can be naturally executed on $\mathsf{M}(p,B)$ for every $1 \le p \le N$ and B by stipulating that the j-th processor carries out the operations of $PE_{(N/p)j}, \cdots, PE_{(N/p)(j+1)-1}$: supersteps with label $i < \log p$ on $\mathsf{M}(N)$ become supersteps with the same label on $\mathsf{M}(p,B)$, local computation otherwise.

In [7] it is shown that in many cases, network-oblivious algorithms which have optimal communication complexities on $\mathsf{M}(p,B)$ exhibit optimal communication time (see below) on a variant of the *Decomposable-BSP* model [5, 31], denoted as $\mathsf{D\text{-}BSP}(P,\boldsymbol{g},\boldsymbol{B})$ where $\boldsymbol{g} = (g_0, \cdots g_{\log P-1})$ and $\boldsymbol{B} = (B_0, \cdots B_{\log P-1})$. A $\mathsf{D\text{-}BSP}(P,\boldsymbol{g},\boldsymbol{B})$ is essentially an $\mathsf{M}(P,\cdot)$ machine where the *communication time* of an i-superstep is defined to be $h(P,B_i)g_i$, where $h(P,B_i)$ denotes its block-degree on $\mathsf{M}(P,B_i)$. The communication time of a D-BSP algorithm is the sum of the communication times of its supersteps.

4.1 Some Network-Oblivious Algorithms

Prefix Sum Computation A network-oblivious algorithm, named *NO-PS*, for $\mathsf{M}(n)$ which computes the prefix sums of a sequence of n elements $(x_0, \ldots x_{n-1})$ on which it is defined a binary associative operation $*$ is described in [6]. We report it for reader convenience. For simplicity, we suppose the existence of the identity 1 of operation $*$: the algorithm can be easily generalized by using a special symbol to the case identity does not exist. The algorithm is a simple adaptation of the PRAM algorithm in [25] to the network-oblivious framework which exploits superstep labels. NO-PS uses two support structures $b(i,j)$ and $c(i,j)$, where $0 \le i \le \log n$ and $0 \le j < n/2^i$. At the end of the algorithm, the j-th prefix sum $x_0 * \ldots * x_j$ will be in $c(0,j)$.

Data are evenly distributed among PEs as follows: x_j, $b(0,j)$ and $c(0,j)$, for each $0 \leq j < n$, are in PE_j, while $b(i,j)$ and $c(i,j)$, for $0 < i \leq \log n$ and $0 \leq j < n/2^i$, in $PE_{\Lambda(i,j)}$, where $\Lambda(i,j) = j2^i + 2^{i-1} - 1$. It is not difficult to see that each PE contains only one entry of $b(i,j)$ and $c(i,j)$ with $i > 0$. Indeed, suppose a PE contains $b(i,j)$ and $b(\tilde{i},\tilde{j})$ (i.e., $\Lambda(i,j) = \Lambda(\tilde{i},\tilde{j})$), with $i, \tilde{i} > 0$ and $(i,j) \neq (\tilde{i},\tilde{j})$: if $i = \tilde{i}$, it follows that $j = \tilde{j}$; on the other hand, if $i < \tilde{i}$ (the case $i > \tilde{i}$ is symmetric), the i-th less significant bits of $\Lambda(i,j) + 1$ and $\tilde{\Lambda}(\tilde{i},\tilde{j}) + 1$ differ unless $i = \tilde{i}$, which is contradiction. In the same way it can be proved that each PE contains only one entry of $c(i,j)$ with $i > 0$. The pseudocode of NO-PS is given in Fig. 7.

```
NO-PS($x_0, \dots x_{n-1}$)
**Input:** $n$ elements $x_0, \dots x_{n-1}$
**Output:** The prefix sums of the elements.
 1: Each $PE_j$ sets $b(0,j)$ to $x_j$, for $0 \leq j < \log n$.
 2: for $i = 1$ to $\log n$ do
 3:     $PE_{\Lambda(i,j)}$ requires $b(i-1,2j)$ $b(i-1,2j+1)$ and sets $b(i,j) \leftarrow b(i-1,2j) * b(i-1,2j+1)$, for $0 \leq j < n/2^i$.
 4: $PE_{\Lambda(0,\log n)}$ set $c(0,\log n)$ to 1.
 5: for $i = \log n - 1$ down to 0 do
 6:     $PE_{\Lambda(i,j)}$ requires $c(i+1,\lfloor j/2 \rfloor)$ and $b(i,j-1)$ and computes, for $0 \leq j < n/2^i$:
            $c(i,j) = \begin{cases} c(i+1,j/2) & \text{if } j \text{ even} \\ c(i+1,(j-1)/2) * b(i,j-1) & \text{if } j \text{ odd} \end{cases}$
 7: Each $PE_j$ sets $c(0,j)$ to $c(0,j) * x_j$, for $0 \leq j < \log n$.
```

Fig. 7. Network-oblivious algorithm for computing the prefix sums of n elements.

The correctness of the algorithm follows by proving inductively that $b(i,j)$ and $c(i,j)$ contain $x_{2^i j} * \dots * x_{2^i(j+1)-1}$ and $x_0 * \dots * x_{2^i j-1}$ (or the identity if $j = 0$), respectively. We observe that, in the i-th iteration, for $1 \leq i \leq \log n$, of the first **for** loop, $PE_{j2^i+2^{i-1}-1}$, which contains $b(i,j)$, receives $b(i-1,2j)$ from $PE_{j2^i+2^{i-2}-1}$ (from PE_{2j} if $i = 1$) and $b(i-1,2j+1)$ from $PE_{j2^i+2^{i-1}+2^{i-2}-1}$ (from PE_{2j+1} if $i = 1$): hence, PEs perform a $(\log n - i)$-superstep since PEs' indexes differ on the first i less significant bits. In the same fashion, it can be proved that in the i-th iteration of the second **for** loop, PEs perform a $(\log n - i - 1)$-superstep.

Theorem 7. *The network-oblivious algorithm NO-PS with input size n exhibits the following communication $H_{\text{NO-PS}}(n,p,B)$ and computation $T_{\text{NO-PS}}(n,p)$ complexities*[4] *when executed on $\mathsf{M}(p,B)$, for $p \leq n$:*

$$H_{\text{NO-PS}}(n,p,B) = \Theta(\log p), \qquad T_{\text{NO-PS}}(n,p) = \mathcal{O}\left(\frac{n}{p}\log n\right). \tag{1}$$

The communication complexity is optimal.

Proof. There are 2 i-supersteps for each $0 \leq i < \log n$, and in each one 2^i PEs (call them *active* PEs), evenly distributed among the PEs, sends/receives $\mathcal{O}(1)$ messages and performs $\mathcal{O}(1)$ operations. On $\mathsf{M}(p,B)$, supersteps whose labels are bigger than or equal to $\log p$ become local computation. On the other hand, in supersteps whose label is smaller than $\log p$, each processor sends/receives $\mathcal{O}(1)$ messages since it simulates $\mathcal{O}(1)$ active PEs. Therefore, the communication complexity is $\Theta(\log p)$, which is optimal for [22]. Since in each superstep (even those whose labels are bigger than or equal to $\log p$) a processor simulates the n/p assigned PEs, the computation complexity is $\mathcal{O}\left(\frac{n}{p}\log n\right)$. $\square$

[4] Observe that the computation complexity of a network-oblivious algorithm is independent of the block communication size B, while this is not the case in general of the communication complexity.

NO-PS can be easily extended to compute prefix sums of n elements on $f(n)$ PEs, where $f(n) \leq n$. Each PE, which contains $n/f(n)$ adjacent elements, computes the sums of local data reducing the problem to a prefix sum computation of $f(n)$ elements; then the previous algorithm is adopted. Clearly, such an algorithm can be performed on $\mathsf{M}(p, B)$ for $p \leq f(n)$. In the following sections we refer to this extension, whose complexities are stated in the following corollary.

Corollary 1. *The computation of prefix sums of n elements on $M(f(n))$ PEs exhibits the following communication $H_{\text{NO-PS}}(n, p, B)$ and computation $T_{\text{NO-PS}}(n, p)$ complexities when executed on $M(p, B)$, for $p \leq f(n)$:*

$$H_{\text{NO-PS}}(n, p, B) = \Theta\left(\log p\right), \qquad T_{\text{NO-PS}}(n, p) = \mathcal{O}\left(\frac{n}{p} + \frac{f(n)}{p}\log n\right). \tag{2}$$

The algorithm exhibits optimal communication and computation complexities when $f(n) \leq n/\log n$.

The communication time of NO-PS on a D-BSP machine is provided by the subsequent theorem.

Lemma 1. *The network-oblivious algorithm NO-PS with input size n exhibits the following communication time $D_{\text{NO-PS}}(n, P, \boldsymbol{g}, \boldsymbol{B})$ when executed on a D-BSP$(P, \boldsymbol{g}, \boldsymbol{B})$ machine, for $P \leq n$:*

$$D_{\text{NO-PS}}(n, P, \boldsymbol{g}, \boldsymbol{B}) = \mathcal{O}\left(\sum_{i=0}^{\log P - 1} g_i\right). \tag{3}$$

The communication time is optimal when $g_i = \Theta\left(g_0/c^i\right)$, where c is an arbitrary constant and $c > 1$.

Proof. The proof is equivalent to the previous one: since there are 2 i-supersteps for each $0 \leq i < \log P$ and in each one a processor of the D-BSP machine sends/receives $\mathcal{O}(1)$ messages, the theorem follows. □

Sorting The network-oblivious algorithm for sorting n elements described in [7] is based on Column-Sort [27] and defined for an $\mathsf{M}(n)$ machine. However, its communication complexity is optimal on an $\mathsf{M}(p, B)$ with $B \leq \sqrt{n/p}$ and $p \leq n^{1-\epsilon}$ for any constant $0 < \epsilon < 1$, and its computation complexity suboptimal by a factor $\mathcal{O}\left((\log n)^{\log_{3/2} 6}\right)$.

By reducing the maximum parallelism of the algorithm, that is, designing the algorithm on an $\mathsf{M}(n^{1-(2/3)^\kappa})$ machine for an arbitrary constant $\kappa \geq 1$, we yield communication and computation optimality on $M(p, B)$ for any $p \leq n^{1-(2/3)^\kappa}$ and suitable values of B. This network-oblivious algorithm, named *NO-CS*, is defined as in [7] but, when the subproblem size is $n^{(2/3)^\kappa}$, the recursion ends and each subproblem is solved sequentially by an optimal work algorithm (e.g., mergesort).

Theorem 8. *The network-oblivious sorting algorithm NO-CS with input size n exhibits the following communication $H_{\text{NO-CS}}(n, p, B)$ and computation $T_{\text{NO-CS}}(n, p)$ complexities*[5] *when executed on $M(p, B)$ with $p \leq n^{1-(2/3)^\kappa}$:*

$$H_{\text{NO-CS}}(n, p, B) = \mathcal{O}\left(4^\kappa\left(\frac{n}{Bp} + \sqrt{\frac{n}{p}}\right)\right), \qquad T_{\text{NO-CS}}(n, p) = \Theta\left(6^\kappa \frac{n}{p}\log n\right).$$

Since κ is a constant, both complexities are optimal when $B = \mathcal{O}\left(\sqrt{n/p}\right)$.

[5] With abuse of notation we keep the constant κ in the asymptotical notation.

Proof. The algorithm consists of four rounds in which $n^{1/3}$ subproblems of size $n^{2/3}$ are solved in parallel. In each recursive call the algorithm performs a transposition which can be performed through an adaptation of the network-oblivious algorithm proposed in [7] which requires $\mathcal{O}\left(n/(Bp)+\sqrt{n/p}\right)$ communications and $\mathcal{O}(n/p)$ operations. The communication complexity is given by the following recursion:

$$H_{\text{NO-CS}}(n,p,B) \leq \begin{cases} 4H\left(n^{2/3}, p/n^{1/3}, B\right) + \mathcal{O}\left(n/(Bp)+\sqrt{n/p}\right) & \text{if } p>1 \\ 0 & \text{if } p \leq 1 \end{cases}$$

which gives $H_{\text{NO-CS}}(n,p,B) = \mathcal{O}\left(\left(n/(Bp)+\sqrt{n/p}\right)(\log n/\log(1+n/p))^{\log_{3/2} 4}\right)$. Since $p \leq n^{1-(2/3)^\kappa}$ the first equation follows. When we consider the computation complexity, the recurrence terminates when the input size is $n_0 = n^{(2/3)^\kappa}$ (here n denotes the initial input size) and the complexity is upper bounded by the following recurrence:

$$T(n,p) \leq \begin{cases} 4T\left(n^{2/3}, p/n^{1/3}\right) + \mathcal{O}(n/p) & \text{if } p>1 \\ 4n^{1/3}T\left(n^{2/3}, 1\right) + \mathcal{O}(n) & \text{if } n>n_0 \text{ and } p>1 \\ \mathcal{O}(n\log n) & \text{if } n \leq n_0 \text{ and } p \leq 1 \end{cases}$$

from which we get $T(n,p) = \mathcal{O}\left((n/p)\log n\left(\log n/\log(1+n^{(2/3)^\kappa})\right)^{\log_{3/2} 6}\right)$ from which follows the equation in the statement. □

We observe that in the previous theorem communication and computation complexities depend on κ (which is negligible in the asymptotical notation), but the bound on the communication block size B is independent of κ. The following lemma shows that NO-CS yields optimal performance also in a D-BSP machine.

Lemma 2. *The network-oblivious sorting algorithm NO-CS with input size n exhibits optimal communication time $D_{\text{NO-CS}}(n,P,\boldsymbol{g},\boldsymbol{B})$ on a D-BSP$(P,\boldsymbol{g},\boldsymbol{B})$, for $P \leq n^{1-(2/3)^\kappa}$ and $B_0 = \mathcal{O}\left(\sqrt{n/P}\right)$, and*

$$D_{\text{NO-CS}}(n,P,\boldsymbol{g},\boldsymbol{B}) = \Theta\left(\frac{n}{B_0 P} g_0\right).$$

Proof. The communication time can be straightforwardly derived. Since in the worst case $\Theta(n/P)$ elements must be exchanged between the first and the second halves of the P processors, the optimality follows. □

Since NO-CS is parametric in the constant κ, we have the following corollary. In the following sections, we then suppose NO-CS to be defined on $\mathsf{M}(n^{1-\epsilon})$, for any constant $\epsilon \in (0,1)$.

Corollary 2. *There exists an optimal sorting algorithm defined on $\mathsf{M}(n^{1-\epsilon})$, where ϵ is an arbitrary constant in $(0,1)$ which exhibits $\mathcal{O}\left(n/(Bp)+\sqrt{n/p}\right)$ communication complexity and $\mathcal{O}((n/p)\log n)$ computation complexity on any $\mathsf{M}(p,B)$ for $p \leq n^{1-\epsilon}$; the communication complexity is optimal when $B \leq \sqrt{n/p}$.*

Furthermore, the algorithm yields optimal $\mathcal{O}(ng_0/B_0P)$ communication time on any D-BSP$(P,\boldsymbol{g},\boldsymbol{B})$, for $P \leq n^{1-\epsilon}$ and $B_0 = \mathcal{O}\left(\sqrt{n/P}\right)$.

Proof. The corollary follows by setting $\kappa = \lfloor \log_{2/3} \epsilon \rfloor$. □

```
Input: $n \times n$ matrix $x$, function $f : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \times \mathcal{S} \to \mathcal{S}$, set $\Sigma_f$ of triplets $\langle i,j,k \rangle$, with $i,j,k \in [0,n)$.
Output: transformation of $x$ defined by $f$ and $\Sigma_f$.
1: for $k \leftarrow 0$ to $n-1$ do
2:   for $i \leftarrow 0$ to $n-1$ do
3:     for $j \leftarrow 0$ to $n-1$ do
4:       if $\langle i,j,k \rangle \in \Sigma_f$ then $x[i,j] \leftarrow f(x[i,j], x[i,k], x[k,j], x[k,k])$;
```

Fig. 8. Gaussian Elimination Paradigm (GEP).

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. For simplicity, we assume n to be a power of two. By *Gaussian Elimination Paradigm* (*GEP*), introduced in [12], we refer to the computation in Fig. 8, where the algorithm modifies x by applying a given set of *updates*, denoted by $\langle i,j,k \rangle$ for $i,j,k \in [0,n)$, of the form $x[i,j] \leftarrow f(x[i,j], x[i,k], x[k,j], x[k,k])$. We let Σ_f denote the set of such updates that the algorithm needs to perform, and suppose that the inclusion check in Line 5 and function f are computed in constant time. Many problems can be solved using a GEP computation, including Floyd-Warshall's all-pairs shortest paths, Gaussian Elimination and LU decomposition without pivoting, matrix multiplication.

A cache-oblivious recursive implementation of the GEP paradigm, called *I-GEP*, was presented in [12] and parallelized in [13] for multicore models with one level of caches. Specifically, I-GEP consists of four recursive functions $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$: each one accepts 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 performs updates in $\Sigma_f \cap (I \times J \times K)$ through eight recursive calls to $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$. The functions differ in the amount of overlap X, U, V and W have among them: $\mathcal{A}$ is invoked when $I \equiv J \equiv K$, $\mathcal{B}$ when $I \cap J = \emptyset$ and $I \equiv K$, $\mathcal{C}$ when $I \cap J = \emptyset$ and $J \equiv K$, $\mathcal{D}$ when I, J and K do not overlap. I-GEP incurs $\mathcal{O}\left(n^3/\left(B\sqrt{M}\right)\right)$ misses and requires $\mathcal{O}\left(n^3/p + n\log^2 n\right)$ parallel time. In [12], it is proved that I-GEP produces the correct output under certain conditions which are met by all notable instances mentioned above. Also presented in [13] is *C-GEP* which extends I-GEP and implements correctly any instance of GEP with no degradation in performance. Tiled I-GEP [14] runs in $\mathcal{O}(n^3/p + n)$ time without increasing the cache complexity on HM (Section 2) but is not multicore-oblivious.

In the following sections we describe multicore and network-oblivious algorithms for GEP which are based on I-GEP. The network-oblivious algorithm, named N-GEP, is somewhat modified from its multicore counterpart due to the different models in which algorithms are specified: a shared-memory model for the multicore-oblivious algorithm, a distributed-memory model for the network-oblivious one. To prove the correctness of N-GEP we will later introduce the notion of commutative GEP computation.

6 Multicore-Oblivious I-GEP

In this section we analyze a multicore-oblivious version of I-GEP based on our parallel I-GEP implementation given in [13]. As already mentioned in Section 8, I-GEP is an in-place cache-oblivious implementation of GEP, and consists of four recursive functions $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$. The initial call is made to function $\mathcal{A}$. Each function makes eight recursive function calls on inputs of size $\frac{n}{2} \times \frac{n}{2}$, where $n \times n$ is the size of the input matrix passed to it. The functions differ in the order the eight recursive calls are made, and also in the amount of parallelism they expose. Function $\mathcal{A}$ exposes the least amount of parallelism while $\mathcal{D}$ exposes the most.

Theorem 9. *When executed under the SB scheduler, I-GEP on an $n \times n$ input, $n^2 \geq C_{h-1}$, incurs $\mathcal{Q}_i(n) = \mathcal{O}\left(n^3/\left(q_i B_i \sqrt{C_i}\right)\right)$ cache misses at each level i cache, and terminates in $\mathcal{T}(n) = \mathcal{O}\left(n^3/p\right)$ parallel time, provided $C_i = \Omega\left(B_i^2\right)$ for all $i \in [1, h-1]$, and $C_i > c_i \cdot p_i \cdot C_{i-1}$ with $c_i = 2\log^2\left(C_i/C_{i-1}\right)$ holds for $i \in [2, h-1]$.*

Proof. (Sketch) Observe that I-GEP does not access any data item outside recursive function calls corresponding to inputs of size 1×1. Therefore, it suffices to compute the cache-misses incurred only by tasks with space bound smaller than the largest cache size, i.e., C_{h-1}. The SB scheduler ensures that once a task τ fits into an L_i cache λ and starts executing, all subtasks generated by τ and its descendants will be executed entirely by the cores under *shadow*(λ) without any interference from tasks generated outside *shadow*(λ). Observe that each subtask generated by τ has space bound $s(\tau)/4$, and since $p_i \geq 2 \Rightarrow C_i > c_i \cdot p_i \cdot C_{i-1} > 4C_{i-1}$ for $i \in [2, h-1]$, each migrated task anchored at C_{i-1} will have space bound larger than $C_{i-1}/4$, and each migrated task anchored at C_1 will have space bound $\Omega\left(B_1^2\right)$. There are only $\Theta\left(n/\sqrt{C_i}\right)$, $\Theta\left(n^2/C_i - n/\sqrt{C_i}\right)$ and $\Theta\left(n^3/\left(C_i\sqrt{C_i}\right) - 2 \cdot n^2/C_i + n/\sqrt{C_i}\right)$ migrated 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 $\mathcal{O}\left(\sqrt{C_i} + C_i/B_i\right)$ cache-misses in λ (for reading the input into λ and writing the output to the next higher level cache), the claimed bound follows.

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 L_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 $\Theta\left(C_1\right)$, $\mathcal{T}_1(C_1) = \mathcal{O}\left(C_1^{3/2}\right) = \mathcal{O}\left(C_1^{3/2}/p'_1\right)$ (since $p'_1 = 1$). Hence as inductive hypothesis let us assume that $\mathcal{T}_{i-1}(C_{i-1}) = \mathcal{O}\left(C_{i-1}^{3/2}/p'_{i-1}\right)$ 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_i with space bound $\Theta\left(C_i\right)$ anchored at any level i cache λ_i. Following the recurrence relations given in [13] for computing the critical path lengths of I-GEP functions in its computational DAG, one can verify that the critical path length of τ_i is $\mathcal{O}\left(\sqrt{C_i/C_{i-1}}\log^2\sqrt{C_i/C_{i-1}} \cdot \mathcal{T}_{i-1}(C_{i-1})\right)$, and thus using Brent's principle, $\mathcal{T}_i(C_i) = \mathcal{O}\left(\left((C_i/C_{i-1})^{3/2}/p_i + \sqrt{C_i/C_{i-1}}\log^2\sqrt{C_i/C_{i-1}}\right) \cdot \mathcal{T}_{i-1}(C_{i-1})\right)$. Since $C_i > c_i \cdot p_i \cdot C_{i-1}$ and $c_i = 2\log^2\left(C_i/C_{i-1}\right)$, the first term in $\mathcal{T}_i(C_i)$ dominates the second term, and thus $\mathcal{T}_i(C_i) = \mathcal{O}\left(C_i^{3/2}/\left(p_i p'_{i-1}\right)\right) = \mathcal{O}\left(C_i^{3/2}/p'_i\right)$. Hence, extending the induction up to level $h-1$, we obtain, $\mathcal{T}_{h-1}(C_{h-1}) = \mathcal{O}\left(C_{h-1}^{3/2}/p'_{h-1}\right) = \mathcal{O}\left(C_{h-1}^{3/2}/p\right)$, and since there are $\mathcal{O}\left(\left(n/\sqrt{C_{h-1}}\right)^3\right)$ I-GEP functions with space bound $\Theta\left(C_{h-1}\right)$, we conclude that $\mathcal{T}(n) = \mathcal{O}\left(\left(n/\sqrt{C_{h-1}}\right)^3 \cdot C_{h-1}^{3/2}/p\right) = \mathcal{O}\left(n^3/p\right)$. □

6.1 N-GEP: a Network-Oblivious Algorithm for GEP

We propose *N-GEP*, an optimal NO algorithm which performs correctly any commutative GEP computation which is correctly solved by I-GEP. It exhibits space optimality, which is not yielded by a straightforward NO implementation of I-GEP, and is optimal also on the D-BSP for a wide range of machine parameters.

We now formally define a commutative GEP computation. A GEP computation is *commutative* if its function f satisfies the following property for each y, u_1, v_1, w_1, u_2, v_2, w_2 in $\mathcal{S}$: $f(f(y, u_1, v_1, w_1), u_2, v_2, w_2) = f(f(y, u_2, v_2, w_2), u_1, v_1, w_1)$. Not all GEP computations are commutative, however all of the instances of GEP for the aforementioned notable problems can be easily seen to be commutative. We will use this notion to prove the correctness of N-GEP, the NO algorithm we present for these applications of GEP.

N-GEP is built on the parallel implementation of I-GEP in [13], 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, whose pseudocode is given in Appendix 1, are suitable adaptations of their counterparts in I-GEP; in contrast, $\mathcal{D}^*$, described in Fig. 9, is based on I-GEP's $\mathcal{D}$ but solves subproblems in a different order and is equivalent to $\mathcal{D}$ for commutative GEP computations. The differences between N-GEP and I-GEP are a consequence of the different models where algorithms are designed: I-GEP is built on a CREW shared-memory model, while N-GEP is defined in a distributed-memory model where PEs communicate in a point-to-point fashion.

```
$\mathcal{D}^*(X, U, V, W, m, \mathcal{P})$
**Input:** $m \times m$ matrices $X \equiv x[I, J]$, $U \equiv x[I, K]$, $V \equiv x[K, J]$ and $W \equiv x[K, K]$, with $I, J, K$ intervals in $[0, n)$; set $\mathcal{P}$ of consecutive numbered PEs assigned to the current instance of $\mathcal{D}^*$.
**Output:** execution of all updates $\langle i, j, k\rangle \in T$, with $T = \Sigma_f \cap (I \times J \times K)$.
1: if $T = \emptyset$ then return; if $m = 1$ or $|\mathcal{P}| = 1$ then solve the problem sequentially with I-GEP and return;
2: Let $\mathcal{P}_{i,j}$, with $0 \le i, j \le 1$, be a partition of $\mathcal{P}$ where each set contains $|\mathcal{P}|/4$ consecutive numbered PEs;
3: $W_{0,1} \leftarrow W_{1,1}$, $W_{1,0} \leftarrow W_{0,0}$; sync; {The assignment is achieved by a suitable communication among PEs}
4: Permute entries of $X$, $U$, $V$ and $W$ in such a way each quadrant will be allocated in row-major in a $\mathcal{P}_{i,j}$,
   according with the following recursive calls; sync;
5: in parallel: $\mathcal{D}^*(X_{0,0}, U_{0,0}, V_{0,0}, W_{0,0}, m/2, \mathcal{P}_{0,0})$, $\underline{\mathcal{D}^*(X_{0,1}, U_{0,1}, V_{1,1}, W_{0,1}, m/2, \mathcal{P}_{0,1})}$,
                $\underline{\mathcal{D}^*(X_{1,0}, U_{1,1}, V_{1,0}, W_{1,1}, m/2, \mathcal{P}_{1,0})}$, $\mathcal{D}^*(X_{1,1}, U_{1,0}, V_{0,1}, W_{1,0}, m/2, \mathcal{P}_{1,1})$;
6: As Line 6.1.
7: in parallel: $\mathcal{D}^*(X_{0,0}, U_{0,1}, V_{1,0}, W_{1,1}, m/2, \mathcal{P}_{0,0})$, $\underline{\mathcal{D}^*(X_{0,1}, U_{0,0}, V_{0,1}, W_{1,0}, m/2, \mathcal{P}_{0,1})}$,
                $\underline{\mathcal{D}^*(X_{1,0}, U_{1,0}, V_{0,0}, W_{0,0}, m/2, \mathcal{P}_{1,0})}$, $\mathcal{D}^*(X_{1,1}, U_{1,1}, V_{1,1}, W_{0,1}, m/2, \mathcal{P}_{1,1})$;
8: Re-establish the initial layout; sync;
```

Fig. 9. N-GEP's function $\mathcal{D}^*$. The underlined recursive calls are inverted in I-GEP's $\mathcal{D}$. The construct **sync** indicates the synchronization at the end of a superstep (superstep labels are not reported for simplicity).

We denote by n and m the initial input size and the input size of a generic recursive level, respectively. Each function receives as inputs at most four $m \times m$ matrices (i.e., X, U, V and W)[6] and the set $\mathcal{P}$ containing the $q = |\mathcal{P}|$ consecutive numbered PEs assigned to the function. We assume each of the four input matrices to be distributed according with a row-major layout among $\min\{q, m^2\}$ PEs, evenly chosen from $\mathcal{P}$. The initial call is $\mathcal{A}(x, n, \mathcal{P})$, where x is the $n \times n$ input matrix and $\mathcal{P} = \{PE_0, \dots PE_{n^2/\log^2 n-1}\}$.

When $m = 1$ or $q = 1$, each function solves the problem sequentially through I-GEP. When $m > 1$ and $q > 1$, each input matrix is split into four $m/2 \times m/2$ quadrants as in I-GEP, and then eight subproblems are solved recursively through calls to $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}^*$: subproblems in $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are solved in the same order as in I-GEP; in contrast, the order in which subproblems are solved in $\mathcal{D}^*$ differs from the one used in I-GEP's $\mathcal{D}$. In general, $\mathcal{D}^*$ is not equivalent to I-GEP's $\mathcal{D}$, but it guarantees a constant memory blow-up, which would not be obtained by a simple adaptation of $\mathcal{D}$ to the network-oblivious framework. Each subproblem is solved recursively by q/k consecutive numbered PEs of $\mathcal{P}$, where k, with $k \in \{1, 2, 4\}$, denotes the number of subproblems which are solved in parallel. Note that the way PEs are assigned does not guarantee the number q of assigned PEs to be small than the number m^2 of entries in a matrix. In functions $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, inputs to the eight subproblems are stored in new matrices of size $m/2 \times m/2$ suitably allocated among PEs in $\mathcal{P}$: hence, $\mathcal{O}\left(\lceil m^2/q \rceil\right)$ new space per PE is allocated in each invocation. In $\mathcal{D}^*$ no new space per PE is required.

[6] Differently from I-GEP, function $\mathcal{A}$ receives only X since U, V and W completely overlap with it, and function $\mathcal{B}$ (resp., $\mathcal{C}$) receives X and U (resp., X and V) since V and W (resp., U and W) overlap with them, respectively.

Theorem 10. *The NO algorithm N-GEP performs correctly any commutative GEP computation which is correctly solved by I-GEP, and each PE exhibits a constant memory blow-up.*

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. Function $\mathcal{D}^*$ does not require additional space since no new matrices are allocated, and the amount of additional space required by $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ decreases geometrically in each recursive level and is asymptotically negligible. □

We remark that if function $\mathcal{D}^*$ is replaced by a NO implementation of I-GEP's $\mathcal{D}$, then N-GEP would perform correctly any GEP computation which is correctly solved by I-GEP, however, each PE would exhibit a $\mathcal{O}(\log n)$ memory blow-up. N-GEP can be extended to correctly implement any commutative GEP computation, without performance degradation, by adopting the ideas in C-GEP.

The following theorem shows that N-GEP performs optimally on the $\mathsf{M}(p, B)$ for wide ranges of the p and B parameters.

Theorem 11. *A commutative GEP computation on an $n \times n$ matrix can be performed by N-GEP on an $\mathsf{M}(p, B)$, for $1 < p \leq n^2/\log^2 n$ and $B \geq 1$, with communication $H_{\mathrm{N-GEP}}(n, p, B)$ and optimal computation $T_{\mathrm{N-GEP}}(n, p)$ complexities, where:*

$$H_{\mathrm{N-GEP}}(n,p,B) = \mathcal{O}\left(n^2/(B\sqrt{p}) + n\log^2 n\right), \qquad T_{\mathrm{N-GEP}}(n,p) = \Theta\left(n^3/p\right).$$

The communication complexity is optimal when $1 < p \leq n^2/\log^4 n$ and $B = \mathcal{O}\left(n/(\sqrt{p}\log^2 n)\right)$.

Proof. Consider the execution of $\mathcal{D}^*$ with input size m and q assigned PEs. We denote with r, where $r \leq q$, the number of consecutive $\mathsf{M}(p, B)$ processors that simulate the q PEs, and with $H_{\mathcal{D}^*}(m, r, B)$ the communication complexity of $\mathcal{D}^*$. $H_{\mathcal{B}}(m, r, B)$ and $H_{\mathcal{A}}(m, r, B)$ are similarly defined. We refer to [29] for more details on the mathematical derivations of the following bounds.

When $m \leq 1$ and $r \leq 1$, that is, when $\mathcal{D}^*$ is in a base case or is assigned to PEs simulated by the same processor, there is no communication. Otherwise, each processor performs a constant number of supersteps with block-degree $\mathcal{O}\left(\lceil m^2/(Br)\rceil\right)$. Hence, we have (observe that m^2 can be either smaller or bigger than r):

$$H_{\mathcal{D}^*}(m,r,B) = \begin{cases} 2H_{\mathcal{D}^*}\left(\frac{m}{2}, \frac{r}{4}, B\right) + \mathcal{O}\left(\left\lceil \frac{m^2}{Br}\right\rceil\right) & \text{if } m > 1 \text{ and } r > 1 \\ 0 & \text{if } m \leq 1 \text{ or } r \leq 1 \end{cases}$$

which yields $H_{\mathcal{D}^*}(m, r, B) = \mathcal{O}\left(\lceil m^2/(Br)\rceil \min\{\sqrt{r}, m\}\right)$. In the same fashion, we obtain the communication complexity of $\mathcal{B}$:

$$H_{\mathcal{B}}(m,r,B) = \begin{cases} 2H_{\mathcal{B}}\left(\frac{m}{2}, \frac{r}{2}, B\right) + 2H_{\mathcal{D}^*}\left(\frac{m}{2}, \frac{r}{2}, B\right) + \mathcal{O}\left(\left\lceil \frac{m^2}{Br}\right\rceil\right) & \text{if } m > 1 \text{ and } r > 1 \\ 0 & \text{if } m \leq 1 \text{ or } r \leq 1 \end{cases}$$

which yields after some mathematical derivations $H_{\mathcal{B}}(m, r, B) = \mathcal{O}\left(m^2/(B\sqrt{r}) + m\log m\right)$. The above upper bound applies to function $\mathcal{C}$ as well. Finally, we observe that the communication complexity of $\mathcal{A}$ is dominated by the following recurrence:

$$H_{\mathcal{A}}(m,r,B) = \begin{cases} 2H_{\mathcal{A}}\left(\frac{m}{2}, r, B\right) + 2H_{\mathcal{B}}\left(\frac{m}{2}, \frac{r}{2}, B\right) + 2H_{\mathcal{D}^*}\left(\frac{m}{2}, \frac{r}{2}, B\right) + \mathcal{O}\left(\left\lceil \frac{m^2}{Br}\right\rceil\right) & \text{if } m > 1 \\ 0 & \text{if } m \leq 1 \end{cases}$$

from which we get $H_{\mathcal{A}}(m,r,B) = \mathcal{O}\left(m^2/(B\sqrt{r}) + m\log^2 m\right)$. Since $H_{\mathrm{N-GEP}}(n,p,B) = H_{\mathcal{A}}(n,p,B)$, the first equation in the theorem follows.

Let us denote with $T_{\mathcal{D}^*}(m,q,r)$ the computation complexity of function $\mathcal{D}^*$. $T_{\mathcal{B}}(m,q,r)$ and $T_{\mathcal{A}}(m,q,r)$ are similarly defined. We remind that q is the number of PEs assigned to function $\mathcal{D}^*$ and r is the number of $M(p,B)$ processors that simulate the q PEs. Note that each PE performs $\mathcal{O}\left(\lceil m^2/q\rceil\right)$ operations on local data in each superstep, and therefore each $\mathsf{M}(p,B)$ processor performs $\mathcal{O}\left(\lceil m^2/q\rceil (q/r)\right)$ operations per superstep. The computation complexity of $\mathcal{D}^*$ is upper bounded as follows:

$$T_{\mathcal{D}^*}(m,q,r) = \begin{cases} 2T_{\mathcal{D}^*}\left(\frac{m}{2},\frac{q}{4},\frac{r}{4}\right) + \mathcal{O}\left(\left\lceil\frac{m^2}{q}\right\rceil\frac{q}{r}\right) & \text{if } m>1 \text{ and } r>1\\ 8T_{\mathcal{D}^*}\left(\frac{m}{2},\frac{q}{4},1\right) + \mathcal{O}\left(\left\lceil\frac{m^2}{q}\right\rceil q\right) & \text{if } m>1,\ q>1 \text{ and } r\le 1\\ m^3 & \text{if } m\le 1 \text{ or } q\le 1\end{cases}$$

which solves to $T_{\mathcal{D}^*}(m,q,r) = \mathcal{O}\left((m^3+mq)/r\right)$. The computation complexity of $\mathcal{B}$ is given by the following relation:

$$T_{\mathcal{B}}(m,q,r) = \begin{cases} 2T_{\mathcal{B}}\left(\frac{m}{2},\frac{q}{2},\frac{r}{2}\right) + 2T_{\mathcal{D}^*}\left(\frac{m}{2},\frac{q}{2},\frac{r}{2}\right) + \mathcal{O}\left(\left\lceil\frac{m^2}{q}\right\rceil\frac{q}{r}\right) & \text{if } m>1 \text{ and } r>1\\ 4T_{\mathcal{B}}\left(\frac{m}{2},\frac{q}{2},1\right) + 2T_{\mathcal{D}^*}\left(\frac{m}{2},\frac{q}{2},1\right) + \mathcal{O}\left(\left\lceil\frac{m^2}{q}\right\rceil q\right) & \text{if } m>1,\ q>1 \text{ and } r\le 1\\ m^3 & \text{if } m\le 1 \text{ or } q\le 1\end{cases}$$

Hence, $T_{\mathcal{B}}(m,q,r) = \mathcal{O}\left((m^3 + mq\log m)/r\right)$. Finally, the computation complexity of $\mathcal{A}$ is given by the following relation:

$$T_{\mathcal{A}}(m,q,r) = \begin{cases} 2T_{\mathcal{A}}\left(\frac{m}{2},q,r\right) + 2T_{\mathcal{B}}\left(\frac{m}{2},\frac{q}{2},\frac{r}{2}\right) + 2T_{\mathcal{D}^*}\left(\frac{m}{2},q,r\right) + \mathcal{O}\left(\left\lceil\frac{m^2}{q}\right\rceil\frac{q}{r}\right) & \text{if } m>1\\ \mathcal{O}(1) & \text{if } m\le 1\end{cases}$$

which gives $T_{\mathcal{A}}(m,q,r) = \mathcal{O}\left((m^3 + mq\log^2 m)/r\right)$. Since $T_{\mathrm{N-GEP}}(n,p) = T_{\mathcal{A}}(n, n^2/\log^2 n, p)$, the theorem follows.

We now prove the optimality of N-GEP. Matrix multiplication with only semiring operations can be computed by a commutative GEP computation. Hence, lower bounds on its communication and computation complexities translate into worst-case lower bounds for an algorithm which performs any commutative GEP computation. An algorithm for solving matrix multiplication on $\mathsf{M}(p,B)$ requires $\Omega\left(n^3/p\right)$ operations and $\Omega\left(n^2/(B\sqrt{p})\right)$ communications per processor if each one uses $\Theta\left(n^2/p\right)$ words [24]. It follows that N-GEP is optimal when $p \le n^2/\log^4 n$ and $B \le n/(\sqrt{p}\log^2 n)$, since each PE uses $\mathcal{O}\left(\log^2 n\right)$ space and each processor simulates $n^2/(p\log^2 n)$ PEs. □

The following lemma states that, under certain circumstances, N-GEP performs optimally also on a D-BSP model. Note that N-GEP does not satisfy the assumptions required by Theorem 1 in [7] for proving that an optimal NO algorithm translates into an optimal D-BSP algorithm, but its optimality can be establish through a separate proof.

Corollary 3. *A commutative GEP computation on an $n\times n$ matrix can be performed by N-GEP on a D-BSP$(P,\boldsymbol{g},\boldsymbol{B})$, for $1 < P \le n^2/\log^2 n$, with communication time*

$$D_{\mathrm{N-GEP}}(n,P,\boldsymbol{g},\boldsymbol{B}) = \mathcal{O}\left(\sum_{i=0}^{\log P-1}\left(n^2 2^{\frac{i}{2}}/(B_iP) + n\log n\right)g_i\right),$$

which is optimal when $P \le n/\log n$ and $B_i = \mathcal{O}\left(n2^{i/2}/(P\log n)\right)$ for each $0 \le i < \log P$.

Proof. The upper bound can be derived by recursive relations similar to ones in the proof of Theorem 11, but different B_i's and g_i's are used in each recursive level. Optimality descends from the optimality in the D-BSP of the NO algorithm for matrix multiplication in [7], whose communication time is $\mathcal{O}\left(\sum_{i=0}^{\log P-1} n^2 2^{\frac{i}{2}} g_i/(B_i P)\right)$. Again, we refer to [29] for more details on the mathematical derivations. □

7 List Ranking and Graph Algorithms

In this section, we present multicore and network-oblivious algorithms for list ranking, Euler tours and some tree and graph problems.

We represent a linked list of n nodes as an array L where each position contains the identifier (ID) of a node and pointers to its successor and predecessor, denoted by $S(i)$ and $P(i)$ respectively. A linked list node does not typically contain pointers to predecessors, however they can be easily derived by suitably sorting pointers to successors. We define the rank of a node to be its distance from the end of the list. The list ranking problem consists in determining the ranks of every node in a list. Additionally, there is a weighted version of this problem where each edge has an associated weight and the rank of a node is defined to be the sum of the weights of all edges along the path from that node to the last node.

Given a tree T rooted in r, we define an *Euler tour* of T to be a traversal of T which starts and ends at r and visits every edge exactly twice, once in each direction. We assume a tree (and a graph as well) to be represented by adjacency lists.

7.1 Multicore-Oblivious Algorithms

In this section, we present multicore-oblivious algorithms for a variety of graph problems. These algorithms are based on a long series of earlier algorithms designed in the parallel, cache efficient, and multicore contexts. Each section begins with a formal definition of the problem and a summary of relevant prior work before detailing our novel results. In the analysis of all algorithms in this section, every parallel step corresponds to either a sort of the data or a linear scan of disjoint units of data bounded by the size of the initial input. The coarse grained contiguous scheduler described in Section 3 ensures that operations on disjoint units of data obtains the cache complexity of a linear scan at each cache level. So the cache complexity at each level is dominated by the complexity of some number of sorts. We will use the recently presented multicore-oblivious sorting algorithm of Ramachandran et al. [15], which has time complexity $\mathcal{O}((n/p)\log n + \log n \log\log n)$ and level i cache complexity $\mathcal{O}\left(\frac{n}{q_i B_i}\log n/\log(\min(C_i, n/q_i))\right)$.

List Ranking. Consider a linked list L of n nodes. Wyllie [34], using the pointer jumping technique, presented the first algorithm solving list ranking in parallel. However, this algorithm requires $\mathcal{O}(n\log n)$ work and so is not work-time optimal in light of the linear time sequential algorithm. Later developments eventually culminated in a $\mathcal{O}(\log n)$ time, linear work algorithm [17].

Later, these techniques were applied in the cache efficient context to develop algorithms for list ranking with complexity matching that of the sorting lower bound, and eventually to the multicore context. We now present our multicore oblivious algorithm for list ranking, called MO–LR, in Fig. 10. Our algorithm employs the list contraction technique described in [17]. This technique solves the list ranking problem by identifying an independent set of nodes, contracting these nodes out, recursively solving this subproblem, and then extending the solution to the contracted nodes. To find an independent set, we identify a ruling set using an algorithm presented by Arge et al [1]. Whenever the size of the problem is at most some small constant, we solve the problem using the sequential algorithm. We now present our algorithm more formally.

```
MO–LR($L$)
Input: linked list $L$ of $n$ nodes specified by a successor array $S$, predecessor array $P$, and arc weights $w$.
Output: ranks $R$ of each of the $n$ nodes.
 1: if $n$ is a small constant then solve using the sequential algorithm and return endif
 2: $I :=$ MO–IS($L$)
 3: [CGC⇒SB] sort a copy of $S$ by successor index and then construct $S^2$ (the successor of the successor)
 4: [CGC] pfor $0 \le i \le n-1$ (contract out nodes of $I$:)
 5:    if $S(i) \in I$ then $S'(i) := S^2(i)$, $w'(i) := w(i) + w(S(i))$
 6:    else if $i \notin I$ $S'(i) := S(i)$, $w'(i) := w(i)$
 7: end pfor
 8: Let $R' :=$ MO–LR($L' = (S', P', w')$)
 9: [CGC] pfor $0 \le i \le n-1$ (extend the solution to nodes of $I$:)
10:    if $i \in I$ $R(i) = R'(S(i)) - w(i)$
11:    else $R(i) := R'(i)$
12: [CGC] end pfor
13: return $R$
```

Fig. 10. Multicore-Oblivious List Ranking

We next present in Fig. 11 a multicore-oblivious algorithm, called MO–IS, which identifies an independent set of nodes in a linked list. This algorithm is an adaptation of a recent algorithm presented by Arge et al. [1].

```
MO–IS($L$)
Input: linked list $L$ of $n$ nodes specified by a successor array $S$, predecessor array $P$, and arc weights $w$.
Output: an independent set $I$ of at least $n/3$ nodes.
 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) index to associate with each node the
    color of its successor (and predecessor).
 3: [CGC⇒SB] Sort the nodes by color.
 4: for each color $0 \le j \le \log\log n - 1$ do
 5:    [CGC⇒SB] sort nodes of color $j$ by index
 6:    [CGC] identify duplicates by comparing indices of consecutive nodes
 7:    [CGC] remove duplicates and add remaining nodes to the independent set
 8:    [CGC] add a duplicate of the successor (and predecessor) of each remaining node to the node of its color.
 9:    end pfor
10: end for
11: return all nodes added to the independent set.
```

Fig. 11. Multicore-Oblivious Independent Set.

After step 2, for each group of nodes of the same color, we allot three times as much space for duplicates to be inserted. We can compute the sizes of the groups with a prefix sums computation. After step 4, nodes of the same ID will be consecutive in memory. We insert duplicates into the previously allotted space by computing the number of duplicates with a prefix sums and writing at an offset equal to the current group size.

Lemma 3. *On an input of size n, MO–IS returns an independent set of at least $n/3$ nodes.*

Proof. In each iteration, we add only nodes of the same color and so by the definition of a coloring, none of these nodes are adjacent. Across each iteration, we explicitly exclude the successor and predecessor of nodes already added to the set. Thus, the set returned is an independent set. Next, note that this algorithm excludes a vertex precisely when one of its neighbors has been added to the independent set, and so at least one node in every path of 3 nodes has been added to the set. Thus, the independent set contains at least $n/3$ nodes. □

We now analyze the complexity of this algorithm and obtain the following result.

Lemma 4. *When executed on a h-level HM model with p cores, MO–IS on an input of size n terminates in* $T_{\mathrm{IS}}(n,p) = \mathcal{O}((n/p)\log n + B_1 \log(pB_1)\log\log n + \log n(\log\log n)^2)$ *parallel steps, and incurs* $\mathcal{O}\left(\frac{n}{q_iB_i}\log_{C_i} n + \frac{C_i}{B_i}\log_{B_1}(q_iC_i)\log\log n + (\log\log n)^2\right)$ *cache misses at each of the* q_i *caches in level* i *of the hierarchy.*

Proof. First, we claim that steps 1 through 4 can be completed with a constant number of sorts and scans of the data. In each step, we are comparing each node with some property of its successor. To do this without incurring a cache miss per time step, we sort a copy of the input by successor index and compare entries of the same index in each array. To identify a $\log\log n$ coloring of the nodes in step 1 of MO–IS, we apply twice the deterministic coin flipping algorithm due to Cole and Vishkin [16] which, given a $|c|$-coloring, constructs a coloring with $1+\log|c|$ colors. Because this new color is a function of only the initial colors of itself and its successor, a constant number of sorts and scans using the CGC scheduler suffices to accomplish this recoloring, and so the complexity of steps 1 through 4 is dominated by that of sorting.

Next, let n_j denote the number of nodes of color i between steps 2 and 3. In the following loop, there will be at most $3n_j$ nodes of any particular color due to the possibility of 2 duplicates per node being added in earlier iterations. We now analyze the complexity of each iteration of step 5. The scheduler reduces the number of processors used so that at most one processor used has less than B_1 units of data, so sorting these n_j units of data requires time $\mathcal{O}\left(\frac{n_j \log n_j}{\min(p,\lceil n_j/B_1\rceil)}\right) = \mathcal{O}(n_j\log n_j/p + B_1\log(pB_1) + \log n_j \log\log n_j)$. This sums across all $\log\log n$ iterations to $\mathcal{O}((n/p)\log n + B_1\log(pB_1)\log\log n + \log n(\log\log n)^2)$, using the fact that $\Sigma_j n_j \le 3n$. As we had previously associated each node with its successor and predecessor index and color, the remaining steps in the loop correspond to scans of the data.

Next, step 6 consists of a sort of the data and so incurs $\mathcal{O}\left(\frac{n_j}{q_iB_i}\log n_j/(\log\min(C_i, n_j/q_i))\right)$ cache misses. Because the scheduler ensures that $n_j/p + 1 > B_1$ (where p is the number of active processors), whenever $n_j \le q_iC_i$, we still maintain that $\min(C_i, n_j/q_i) \ge B_1$. So this bound simplifies to $\mathcal{O}\left(\frac{n_j}{q_iB_i}\log_{C_i} n_j + \frac{C_i}{B_i}\log_{B_1}(q_iC_i)\right)$. Steps 7, 8, and 9 consist of a scan of the data; However, in step 9, a level i cache could be tasked with writing a node of each color, thus incurring $\mathcal{O}(\log\log n)$ cache misses. Thus, summed across all $\log\log n$ iterations, this loop incurs at most $\mathcal{O}\left(\frac{n}{q_iB_i}\log_{C_i} n + \frac{C_i}{B_i}\log_{B_1}(q_iC_i)\log\log n + (\log\log n)^2\right)$ cache misses. □

We now show that the complexity of each recursive level of MO–LR is dominated by that of MO–IS and so analyze the total complexity of this algorithm.

Theorem 12. *When executed on a h-level HM model with p cores, MO–LR on an input of size n terminates in* $T_{\mathrm{LR}}(n,p) = \mathcal{O}\left((n/p)\log n + (B_1\log(pB_1)\log\log n + \log n(\log\log n)^2)(\log\frac{n}{B_1})\right)$ *parallel steps, and incurs* $\mathcal{Q}_{\mathrm{LR}}(n,q_i) = \mathcal{O}(\frac{n}{q_iB_i}\log_{C_i} n + (\frac{C_i}{B_i}\log_{B_i}(q_iC_i)\log\log n + (\log\log n)^2)\log\frac{n}{B_1})$ *cache misses at each of the* q_i *level caches in level* i *of the hierarchy.*

Proof. First, note that the operations performed in the loops at lines 3 and 5 consist of only sorts and scans of the input. As the complexity of each of these operations is accounted for in the complexity of MO–IS, the complexity of each recursive stage of this algorithm is dominated by that of MO–IS.

Because the scheduler does not split blocks across private caches, the number of processors (and also the number of level i caches) decreases in proportion to the problem size. Specifically, we derive the following recurrences:

$$T_{\mathrm{LR}}(n,p) = \begin{cases} T_{\mathrm{LR}}(n/2,p) + T_{\mathrm{IS}}(n,p) & pB < n \\ T_{\mathrm{LR}}(n/2,p/2) + T_{\mathrm{IS}}(n,p) & B < n \leq pB \\ \mathcal{O}(B_1 \log B_1) & n \leq B \end{cases}$$

$$\mathcal{Q}_{\mathrm{LR}}(n,q_i) = \begin{cases} \mathcal{Q}_{\mathrm{LR}}(n/2,q_i) + \mathcal{Q}_{\mathrm{IS}}(n,q_i) & pB_1 < n \\ \mathcal{Q}_{\mathrm{LR}}(n/2,q_i/2) + \mathcal{Q}_{\mathrm{IS}}(n,q_i) & B_1 < n \leq pB_1 \\ \mathcal{O}(1) & n \leq B_1 \end{cases}$$

By solving these recurrences, we conclude that the time complexity of this algorithm is $T_{\mathrm{LR}}(n,p) = \mathcal{O}((n/p)\log n + (B_1 \log(pB_1) \log\log n + \log n(\log\log n)^2)(\log \frac{n}{B_1}))$ and similarly that the level i cache complexity is $\mathcal{Q}_{\mathrm{LR}}(n,q_i) = \mathcal{O}(\frac{n}{q_i B_i} \log_{C_i} n + (\frac{C_i}{B_i} \log_{B_i}(q_i C_i) \log\log n + (\log\log n)^2) \log \frac{n}{B_1})$. $\square$

Euler Tours and Tree Problems. In this section, we present multicore-oblivious algorithms solving various tree problems with the well known Euler tour technique [30], using list ranking as a subroutine.

We specify an Euler tour of $T = (V,E)$ by defining a successor function S mapping each arc to the next arc along the circuit. The standard way to do this, described by Tarjan and Vishkin [30], is to fix a cyclic ordering on V and the successor of each arc (u,v) is (v,w) where w is the node following u in the cyclic ordering of the adjacency of v. The proof that this defines an Euler tour (instead of a set of arc-disjoint cycles) can be found in [30].

Algorithmically constructing this function is straightforward. For each node v, we assume that the ordering on the set of nodes adjacent to v is simply the order in which these nodes appear in the adjacency list of v. Then for each edge (u_i, v), we can identify the successor $(v, u_{i'})$ as $u_{i'}$ follows u_i in the adjacency list, except when u_i is the last node in the adjacency list of v. We can fix this by making the adjacency list circular; that is, by appending the first node in the adjacency list of v to the end of the adjacency list of v. Thus, for each node u_i in a given adjacency list of v, we can define the successor $(v, u_{i'})$ of (u,v) in constant time.

For completeness, we include a formal presentation of this algorithm, called MO–ET, in Fig. 12.

```
MO–ET($T$)
Input: tree $T = (V,E)$ given as an adjacency list.
Output: Euler tour of $T$ specified by an linked list $L$ as previously specified.
1: [CGC] pfor each $v \in V$ make the adjacency list of $v$ circular end pfor
2: [CGC] pfor each pair of edges $(u,v)$, $(w,v)$ in the adjacency list of each node
3:     set $S(u,v) := (v,w)$ (possibly with some problem specific edge weight $w(u,v)$)
4: end pfor
5: return the Euler tour $S$ with weights $w$.
```

Fig. 12. Multicore-Oblivious Euler Tour

To make the adjacency lists circular, we must append the first node in each adjacency list to the end of the respective adjacency list. Specifically, we scan in parallel across the data and write separately the first edge pair in each adjacency list (step 1). By using the CGC scheduler, this requires $\mathcal{O}(n/p + \log p + B_1)$ parallel time and incurs $\mathcal{O}\left(\frac{n}{q_i B_i} + 1\right)$ cache misses at each level i cache. The next loop (step 2) consists of writing $\mathcal{O}(n)$ disjoint units of data, and so under the CGC scheduler, this step has the same cache- and parallel time complexity as step 1.

We now briefly describe how the Euler tour technique is used to solve various tree problems. Note that the descriptions of how to solve these specific problems require only a constant number of additional sorts and parallel scans of the data, and so the complexity of each of these algorithms is dominated by that of list ranking. We will conclude with a formal statement of this claim.

Rooting a Tree. Given a tree $T = (V, E)$ and a designated vertex $r \in V$, a function $p : V \to V$ *roots* a tree at r if for each node $v \neq r$, $p(v)$ is the next node on the unique path from v to r. Using the Euler tour technique, we can root a tree at r as follows. We construct an Euler tour S starting at r, but breaking the edge from the last node u on the adjacency list of r and setting $S((u, r)) = (u, r)$. We then compute the list ranking of the list of arcs defined by S. Finally, for each arc (x, y), we set $p(y) = x$ whenever the ranking of (x, y) is smaller than that of (y, x). Correctness of this algorithm follows from the observation that the Euler tour starting at r follows a depth first search of T; thus, an arc along the path towards the root will have smaller rankings than those along paths away from the root.

Traversal Numbering. Given a tree T rooted at r, the *preorder* of a node is the order in which that node is visited in a depth first search of T. We compute the preorder traversal d in which each node appears as follows. First, we construct an Euler tour S of T' as previously described. We then assign the weights $w((p(v), v)) = 1$ and $w((v, p(v))) = 0$ to these arcs. Then the preordering d of a vertex v equals the ranking of $(v, p(v))$, and $p(r) = 0$. With small variations, we can also compute the inorder traversal and the postorder traversal of T.

Vertex Depth. Given a tree $T = (V, E)$, the *depth* of a vertex v is the distance between v and r. We can compute the depth d of each vertex using the same approach as before, except by defining $w((p(v), v)) = 1$ and $w((v, p(v))) = -1$. Then the depth d of a node v equals the rank of $(p(v), v)$.

Subtree Size. Given a tree $T = (V, E)$ rooted at $r \in V$ by $p : V \to V$, the *subtree size* at v is the number of nodes below v in T; that is, the number of nodes whose path to r contains v. Using a similar approach, but with $w((p(v), v)) = 0$ and $w((v, p(v))) = 1$, the size of the subtrees rooted at each vertex v equals the difference between the ranking of $(v, p(v))$ and $(p(v), v)$ since each arc between these two arcs is in the subtree rooted at v.

Connected Components in Forests. Given a graph $F = (V, E)$ comprising of a set of t trees (that is, a forest) and a set of $R \subseteq V$ roots, we can construct a partition $D : V \to \{1, \ldots, t\}$ defining the connected components of F as follows. We construct a node s, called the superroot, and connect it to all other roots. We then define $w((s, r)) = 1$ for each $r \in R$ and $w((u, v)) = 0$ for all other arcs (u, v). We make s the first node in the linked list by breaking any one arc to s. Then the two arcs incident to any node except the superroot will have the same ranking, and so the connected component that the node $v \neq s$ belongs to is the ranking of either of its incident arcs.

Theorem 13. *When executed on a h-level HM model with p cores, MO–ET on an input of size n terminates in* $T_{\mathrm{ET}}(n, p) = \mathcal{O}\left((n/p)\log n + (B_1 \log(pB_1)\log\log n + \log n(\log\log n)^2)(\log \frac{n}{B_1})\right)$ *parallel steps, and incurs* $\mathcal{Q}_{\mathrm{ET}}(n, q_i) = \mathcal{O}\left(\frac{n}{q_i B_i}\log_{C_i} n + (\frac{C_i}{B_i}\log_{B_i}(q_i C_i)\log\log n + (\log\log n)^2)\log\frac{n}{B_1}\right)$ *cache misses at each of the q_i level caches in level i of the hierarchy. Furthermore, the problems of rooting a tree, computing a traversal numbering, the depth of vertices and the subtree sizes in a rooted tree can all be completed within these resource bounds.*

Graph Connectivity. In this section, we describe an algorithm due to Hirschberg, Chandra, and Sarwate [23] for connected components (and easily adapted for minimum spanning tree). We recursively solve the problem as follows. For each node, we select the edge incident to the vertex of smallest index. This set of edges induces a forest on the graph, and in each tree, some edge appears twice. We arbitrarily select one of the endpoints as the root of each tree, and then solve the connected components in forests problem as described under Euler Tours and Tree Problems. We then replace all edges (u, v) with edges $(D(u), D(v))$ where $D(u)$ is the connected component in the induced forest to which u belongs, eliminate duplicates, and recursively identify the connected components of this subproblem. We now formally present in Fig. 13 this algorithm, called MO–CC, describe the details of its operation, and analyze its complexity.

```
MO–CC($G$)
Input: graph $G = (V, E)$ of $n$ nodes and $m$ edges given in adjacency list format.
Output: partition $D : V \to V$ such that $D(u) = D(v)$ iff $u$ is connected to $v$.
 1: [CGC] $V_F = V \bigcup \{s\}$; $E_F = \emptyset$; $V' = E' = \emptyset$
 2: [CGC] pfor each $v \in V$ $D(v) = v$ end pfor
 3: if $m = 0$ then return $D$
 4: [CGC] pfor $v \in V$ add to $E_F$ the edge incident to the vertex of smallest index end pfor
 5: [CGC⇒SB] sort the nodes of $E_F$
 6: [CGC] pfor each edge $(u, v)$ added to $E_F$ twice, add $(u, s)$ to $E_F$ end pfor
 7: [CGC] eliminate duplicates from $E_F$.
 8: compute the connected components $D$ of the induced forest using MO–LR and MO–ET.
 9: [CGC] pfor each $u \in V$, add $D(u)$ to $V'$
10: [CGC] pfor each $(u, v) \in E$, add $(D(u), D(v))$ to $E'$.
11: [CGC⇒SB] sort the nodes in $V'$ and $E'$
12: [CGC] eliminate duplicates in $V'$ and $E'$
13: set $D' :=$ MO–CC($G' = (V', E')$)
14: [CGC] pfor $v \in V$, set $D'(v)$ to $D'(D(v))$ end pfor
15: return $D'$
```

Fig. 13. Multicore-Oblivious Connected Components

Theorem 14. *When executed on a h-level HM model with p cores, MO–CC on an input of size* $N = n + m$ *terminates in time* $T_{\mathrm{CC}}(n, m, p) = \mathcal{O}\left((N \log N \log \frac{N}{B_1})/p + (B_1 \log(pB_1) \log\log N + \log N \cdot (\log\log N)^2)(\log^2 \frac{N}{B_1})\right)$ *parallel steps, and incurs* $\mathcal{Q}_{\mathrm{CC}} = \mathcal{O}\left(\frac{N}{q_i B_i} \log_{C_i} N \log \frac{N}{B_1} + (\frac{C_i}{B_i} \log_{B_1}(q_i C_i) \log\log n + (\log\log n)^2) \log^2 \frac{N}{B_1}\right)$ *cache misses at each of the* q_i *caches in level* i *of the hierarchy.*

Proof. Correctness of this algorithm can be found in [23]. Note that in each connected component, there is some node of least index, and so the edge between this node and the node incident to it of least index will be added to E_F for both of these nodes. We identify these edges by sorting E_F in step 5 and comparing consecutive entries in steps 6 and 7. In steps 9, 10, and 14, we apply in parallel the function D or D' to $\mathcal{O}(n)$ nodes and $\mathcal{O}(m)$ edges. Note that each nonrecursive step of the algorithm other than that of list ranking corresponds to either a sort or a scan of the data. So the complexity of each recursive stage of this algorithm is dominated by that of list ranking. Furthermore, note that the number of nodes in each connected component of at least 2 nodes decreases by a factor of at least 2 at each level of recursion; thus, after $\mathcal{O}(\log n)$ levels of recursion, the number of nodes equals the number of connected components and there are no edges between the nodes. Thus, the level i cache complexity of this algorithm equals the level i cache complexity of list ranking times $\mathcal{O}\left(\log \frac{N}{B_1}\right)$ (because when the problem size reduces to less than B_1, the problem fits within a single private cache and the scheduler ensures that no more cache misses are incurred because only a single processor is active). □

7.2 Network-Oblivious Algorithms

We now propose network-oblivious algorithms for list ranking, Euler tours and connected components. Algorithms are built on their multicore-oblivious counterparts, however some modifications are required for dealing with the different specification models: indeed a network-oblivious algorithm is defined on a distributed-memory model, while a multicore-oblivious algorithm is defined on a CREW shared-memory.

List Ranking. In this section we describe a network-oblivious algorithm, named *NO-LR*, for computing the ranks of an n-node linked list L. The algorithm assumes the existence of a network-oblivious sorting algorithm *NO-SORT*$(n, f(n))$ which sorts $\Theta(n)$ entries using $f(n)$ PEs, where $f(n)$ is a suitable nondecreasing function of the input size such that $f(n) \leq n$ and $n/f(n)$ is a nondecreasing function. We present upper bounds on the communication and computation complexities of NO-LR which do not depend on the particular implementation of the sorting algorithm. Then, we consider the case NO-SORT is implemented through the network-oblivious algorithm NO-CS given in Section 4.1.

NO-LR is defined on $\mathsf{M}(f(n))$. We assume L to be uniformly distributed among the $f(n)$ PEs: the j-th PE contains nodes in positions $jn/f(n), \ldots (j+1)n/f(n) - 1$ of L, for $0 \leq j < f(n)$. At the high level, the algorithm works as the multicore-oblivious implementation in Fig. 10, however there are substantial changes due to the different specification models (i.e., distributed vs share memory). List contraction and expansion can be performed by means of $\mathcal{O}(1)$ calls to NO-SORT$(n, f(n))$. The procedure for the identification of an independent set, called *NO-IS*, is given in Fig. 14 (the pseudocode does not report superstep synchronizations and labels for clearness) and described in more details below. NO-IS requires some prefix sum computations that are performed through the network-oblivious algorithm NO-PS in Section 4.1. Each line consists of a constant number of 0-supersteps in addition to those required by NO-SORT and NO-PS.

```
NO-IS($L$)
Input: Linked list $L$ of $n$ nodes uniformly distributed among $f(n)$ PEs.
Output: An independent set.
 1: Identify a $\log\log n$ coloring of $L$. Let $G_i$ be the set of nodes of color $i$ and let $n_i = |G_i|$.
 2: Use NO-SORT to associate with each node of $L$ the color of its successor and predecessor.
 3: Redistribute all the nodes of $L$ among PEs in such a way each PE contains $\mathcal{O}(\lceil n_i/f(n) \rceil)$ nodes of $G_i$.
 4: for each color $i$, with $0 \leq i < \log\log n$ do
 5:     Redistribute nodes of $G_i$ among PEs in such a way each PE contains $\mathcal{O}(\lceil n_i/f(n) \rceil)$ nodes of $G_i$.
 6:     For each $0 \leq j < f(n)/f(n_i)$, PEs in the $j$-th segment of $f(n)/f(n_i)$ consecutive PEs move all nodes of $G_i$ into
        the first PE of the segment, denoted as selected.
 7:     Sort by ID nodes of $G_i$ with NO-SORT$(\Theta(n_i), f(n_i))$ using the $f(n_i)$ selected PEs, and then remove duplicates.
        Mark the remaining nodes as independent.
 8:     For each node of $G_i$ marked as independent create duplicates of its successor and predecessor.
 9:     Redistribute the new duplicates among the selected PEs is such a way each PE contains $\mathcal{O}(\lceil n_i/f(n_i) \rceil)$ duplicates
        and in particular $\mathcal{O}\left(\lceil n_i^j/f(n_i) \rceil\right)$ duplicates of color $i$, where $n_i^j$ is the number of duplicates of color $j$ (and hence
        in $G_j$) generated in Line 7.2 during the current iteration.
10:     Redistribute duplicates among the $f(n)$ PEs in such a way each PE contains $\mathcal{O}(\lceil n_i/f(n) \rceil)$ duplicates and in
        particular $\mathcal{O}\left(\lceil n_i^j/f(n) \rceil\right)$ duplicates of color $j$.
11: Collect nodes marked as independent in an array $I$ of size $\Theta(n)$ distributed among the $f(n)$ PEs.
```

Fig. 14. Network-Oblivious Independent Set.

To find the $\log\log n$ coloring in Line 7.2 we make use of two rounds of the deterministic coin tossing of Cole et al. [16], which can be accomplished with $\mathcal{O}(1)$ sorts. We denote by G_i the set of nodes of color i, and let $n_i = |G_i|$.

Line 7.2 is the most important step of the algorithm. To improve the parallelism in the **for** loop, each PE should contain at most $\mathcal{O}(\lceil n_i/f(n)\rceil)$ nodes of G_i: however, if nodes of the same color are stored in consecutive positions of the list as in the multicore-oblivious implementation, a PE will contain $\mathcal{O}(\min\{n_i, n/f(n)\})$ nodes of G_i, which is bigger than $\lceil n_i/f(n)\rceil$. We propose an alternative layout where nodes of G_i are stored every $\Theta(n/n_i)$ positions of L and hence each PE contains $\mathcal{O}(\lceil n_i/f(n)\rceil)$ nodes of G_i. We require each n_i to be a power of two and $n_i \geq n_{i+1}$ for each $0 \leq i < \log\log n$: the first requirement increases the size n by at most a factor two, while the second one is obtained by renaming colors (renaming can be accomplished with $\mathcal{O}(1)$ sortings and prefix sums computations). Initially, nodes are sorted by color and then the $\log\log n$ colors are partitioned into segments of consecutive colors in such a way that for each segment S the number of L's nodes colored with colors in S is n_0 (such a partition is well defined since the n_i's are powers of two and nonincreasing). Then, nodes in the k-th segment, for $0 \leq k < n/n_0$, are redistributed in L recursively every n/n_0 positions, starting from the k-th one. We now provide a formal description.

Each node of color i whose relative position in G_i is[7] j, for $0 \leq i < \log\log n$ and $0 \leq j < n_i$, (relative positions can be computed through a prefix sum computation) is assigned the key KEY(i,j) =OFF$_i$ + jn/n_i, where OFF$_i$ is an offset given by the recursive and sequential algorithm COMPUTE_OFF$(i,0,n)$ in Fig. 15. Then, nodes are sorted according with the new key: Lemma 5 shows the property of the new configuration of nodes in L. Note that, since each PE contains nodes colored by at most $\log\log n$ colors, each PE may require in the worst case the $\log\log n$ values $n_0, \ldots n_{\log\log n-1}$ for computing the key of each node: these values are obtained and spread among PEs through $\mathcal{O}(\log\log n)$ prefix sum computations.

```
COMPUTE_OFF$(i,k,m)$
Input: Values $i,k$ and $m$ such that $k \leq i$ and $\sum_{h=k}^{i} n_h \leq m$. The $n_k$'s, for $0 \leq k < \log\log n$, are global variables.
Output: The value OFF$_i$.
1: if $k = i$ return 0.
2: Find the smallest $k'$, for $k < k' \leq i$ such that $\left\lfloor \frac{\sum_{h=k}^{k'-1} n_h}{n_k} \right\rfloor = \left\lfloor \frac{\sum_{h=k}^{i-1} n_h}{n_k} \right\rfloor$.
3: return $\left\lfloor \frac{\sum_{h=k}^{i-1} n_h}{n_k} \right\rfloor + \frac{m}{n_k}\cdot$COMPUTE_OFF$(i,k',n_k)$.
```

Fig. 15. Algorithm for computing OFF$_i$.

Lemma 5. *After sorting nodes by KEY(i,j), each PE contains $\mathcal{O}(\lceil n_i/f(n)\rceil)$ nodes of G_i, for each $0 \leq i < \log\log n$.*

Proof. We first prove a property of procedure COMPUTE_OFF and then show that values $KEY(i,j)$ generates a permutation of values in $[0,n)$. Then, we show the stated result.

Observe that COMPUTE_OFF(i,k,m) terminates after at most $i+1$ recursive calls and

$$OFF_i = \left\lfloor \frac{\sum_{h=k_0}^{i-1} n_h}{n_{k_0}} \right\rfloor + \frac{n}{n_{k_0}} \left\lfloor \frac{\sum_{h=k_1}^{i-1} n_h}{n_{k_1}} \right\rfloor + \ldots + \frac{n}{n_{k_{s-1}}} \left\lfloor \frac{\sum_{h=k_s}^{i-1} n_h}{n_{k_s}} \right\rfloor \tag{4}$$

where $k_0, \ldots k_s$ is a non-empty subsequence of $0, \ldots i$, $k_0 = 0$, and, if $i > 0$, $k_s < i$. The j-th recursive call, for $0 \leq j < s$, contributes to the j-th term of the equation and we have $n_{k_j} > n_{k_{j+1}}$. Indeed, $n_{k_j} = n_{k_{j+1}}$ for a suitable $0 \leq j < s$ implies that, in the j-th recursive call, $n_{k'} = n_{k_j}$ and $k' = i$. It

[7] With abuse of notation, we suppose that nodes of G_i, for each $0 \leq i < \log\log n$, are ordered by ID.

follows that the $j+1$-st call is a base case (i.e., COMPUTE_OFF(i, i, n_{k_j})) and then we get $j = s$, which is a contradiction.

We now show that there cannot exist two nodes with the same key. Consider two nodes with keys KEY(i,j) and KEY$(\tilde{i},\tilde{j})$ and let $i \neq \tilde{i}$ or $j \neq \tilde{j}$. If $i = \tilde{i}$, then it is simple to see that KEY$(i,j) \neq KEY(\tilde{i},\tilde{j})$. Suppose $i \neq \tilde{i}$ and set without loss of generality $i < \tilde{i}$. Let OFF$_i$ defined as in Equation 4 and let OFF$_{\tilde{i}}$ be defined similarly and denote by $\tilde{k}_0, \dots \tilde{k}_{\tilde{s}}$ the correspondent subsequence of $0, \dots \tilde{i}$, where $\tilde{k}_0 = 0$ and $\tilde{k}_{\tilde{s}} < \tilde{i}$. Denote by s' the biggest value such that sequences $k_0, \dots k_{s'}$ and $\tilde{k}_0, \dots \tilde{k}_{s'}$ coincide. Clearly, $0 \leq s' < i$ and by construction we evince that

$$\left\lfloor \frac{\sum_{h=k_{h'}}^{i-1} n_h}{n_{k_{h'}}} \right\rfloor = \left\lfloor \frac{\sum_{h=\tilde{k}_{h'}}^{i-1} n_h}{n_{\tilde{k}_{h'}}} \right\rfloor \text{ for each } 0 \leq h' < s' \text{ and } \left\lfloor \frac{\sum_{h=k_{s'}}^{i-1} n_h}{n_{k_{s'}}} \right\rfloor \neq \left\lfloor \frac{\sum_{h=\tilde{k}_{s'}}^{i-1} n_h}{n_{\tilde{k}_{s'}}} \right\rfloor \tag{5}$$

For the sake of contradiction let KEY$(i,j) = KEY(\tilde{i},\tilde{j})$; then we have:

$$j\frac{n}{n_i} - \tilde{j}\frac{n}{n_{\tilde{i}}} = (OFF_{\tilde{i}} - OFF_i). \tag{6}$$

If $s' = 0$, we replace the equality in Equation 6 with the module n/n_0 relation. Since n/n_h is divisible by n/n_0's for each $0 \leq h < \log\log n$ we have:

$$0 \equiv \left\lfloor \frac{\sum_{h=0}^{\tilde{i}-1} n_h}{n_0} \right\rfloor - \left\lfloor \frac{\sum_{h=0}^{i-1} n_h}{n_0} \right\rfloor \mod \frac{n}{n_0}.$$

For Equation 5, the two floor terms are different and smaller than n/n_0, and then the right side is not in relation with 0: a contradiction arises and we conclude that the two keys are different. If $s' > 0$, we divide both terms in Equation 6 by $n/n_{k_{s'-1}}$ and replace the equality with the module $n_{k_{s'-1}}/n_{k_{s'}}$ relation. Since $n_{k_{s'-1}}/n_h$ is divisible by $n_{k_{s'-1}}/n_{k_{s'}}$ for each $k_{s'} \leq h < \log\log n$, it follows that

$$0 \equiv \left\lfloor \frac{\sum_{h=k_{s'}}^{\tilde{i}-1} n_h}{n_{k_{s'}}} \right\rfloor - \left\lfloor \frac{\sum_{h=k_{s'}}^{i-1} n_h}{n_{k_{s'}}} \right\rfloor \mod \frac{n_{k_{s'-1}}}{n_{k_{s'}}}. \tag{7}$$

However, the right term is not in relation with 0 since the two terms are different for Equation 5 and smaller than $n_{k_{s'-1}}/n_{k_{s'}}$: this is a contradiction and we conclude the two nodes do not have the same keys.

We now show that KEY$(i,j) < n$. If $n_i = n_0$, clearly KEY$(i,j) < n$. Suppose that $n_0 > n_i$. It is easy to see that for each $0 \leq h' \leq s$, we get:

$$\left\lfloor \frac{\sum_{h=k_{h'}}^{i-1} n_h}{n_{k_{h'}}} \right\rfloor \leq \frac{n - n_{k_{h'}}}{n_{k_{h'}}}.$$

Equation 4 becomes

$$OFF_i \leq \frac{n - n_{k_0}}{n_{k_0}} + \frac{n}{n_{k_0}}\frac{n_{k_0} - n_{k_1}}{n_{k_1}} + \dots \frac{n}{n_{k_{s-1}}}\frac{n_{k_{s-1} - n_{k_s}}}{n_{k_s}} = \frac{n}{n_{k_s}} - 1 < \frac{n}{n_i}, \tag{8}$$

Then, it follows that KEY$(i,j) = OFF_i + jn/n_i < n$.

Therefore, KEY(i,j) generates a permutation of integers in $[0,n)$ and can be used for permuting nodes in L. After sorting nodes according with $KEY(i,j)$, nodes of G_i are located every n/n_i positions: since each PE contains $n/f(n)$ consecutive positions of L, at most $\mathcal{O}\left(\lceil n_i/f(n) \rceil\right)$ nodes of G_i are contained in each PE. □

Let us consider now the i-th iteration of the **for** loop. In Line 7.2, nodes in G_i are evenly distributed among the $f(n)$ PEs: this operation is useless when $i = 0$ since the sorting in Line 7.2. However if $i > 0$, each PE contains $\mathcal{O}\left(\lceil n_i/f(n)\rceil + \log\log n\right)$ nodes of color i at the beginning of the **for** loop: the $\mathcal{O}\left(\lceil n_i/f(n)\rceil\right)$ nodes of color i actually in L and $\mathcal{O}\left(n_i/f(n) + \log\log n\right)$ duplicates of color i added in Line 7.2 of previous iterations (see below for more details). In rebalancing, each PE performs a prefix sum computation for computing the number of nodes of G_i stored in preceding PEs, and sends $\mathcal{O}\left(n_i/f(n) + \log\log n\right)$ messages to at most $\mathcal{O}\left(\log\log n\right)$ PEs.

In Line 7.2, $\Theta\left(n_i\right)$ nodes are sorted with NO-SORT for removing duplicates: however, NO-SORT requires $f(n_i)$ PEs for sorting $\Theta\left(n_i\right)$ values. For this reason, the $\Theta\left(n_i\right)$ nodes of G_i (included duplicates) are moved in Line 7.2 into $f(n_i)$ PEs (called *selected* PEs) evenly chosen among the $f(n)$ PEs: each selected PE receives $\mathcal{O}\left(\lceil n_i/f(n_i)\rceil\right)$ messages from $f(n)/f(n_i)$ consecutive numbered PEs. Nodes that are not removed are marked *independent*.

In Lines 7.2 and 7.2, duplicates of successors and predecessors of the remaining nodes of G_i are created. Each duplicate of a successor (resp., predecessor) is inserted in G_j where j is the color of the successor (resp., predecessor). Then, duplicates are distributed among the $f(n_i)$ selected PEs using a procedure similar to the one adopted in Line 7.2, in such a way each PE contains $\mathcal{O}\left(\lceil n_i/f(n_i)\rceil\right)$ duplicates, and in particular $\mathcal{O}\left(\lceil n_i^j/f(n_i)\rceil\right)$ duplicates of color j, where n_i^j denotes the number of duplicates of color j created at the current iteration.

In Line 7.2 each selected PE distributes duplicates created in Line 7.2 among the $f(n)/f(n_i)$ PEs from which it received nodes of G_i in Line 7.2: each PE receives no more duplicates than the number of nodes it sent in Line 7.2 and $\mathcal{O}\left(\lceil n_i^j/f(n)\rceil\right)$ duplicates of color j. Observe that since each PE receives $\mathcal{O}\left(\lceil n_i^j/f(n)\rceil\right)$ duplicates of color j in each iteration, a PE contains $\mathcal{O}\left(\sum_{i=0}^{j-1}\lceil n_i^j/f(n)\rceil\right) = \mathcal{O}\left(n_j/f(n) + \log\log n\right)$ duplicates of color j just before the j-th iteration. For these reason, nodes are redistributed in Line 7.2.

Finally in Line 7.2, the independent set is stored in an array of size $\Theta\left(n\right)$ distributed among PEs using a prefix sum computation.

It is easy to see that NO-IS and NO-LR are equivalent to their multicore implementations and that NO-IS returns an independent set of size at least $n/3$ (Lemma 3).

We now analyze the communication and computation complexities of NO-LR. We denote the communication and computation complexities of the sorting algorithm by $H_S(n,p,B)$ and $T_S(n,p)$ and set without loss of generality:

$$H_S(n,p,B) = \mathcal{O}\left(\frac{n}{Bp}\frac{\log n}{\log(1+n/p)} + K(n,p,B)\right), \qquad T_S(n,p) = \mathcal{O}\left(\frac{n}{p}\log n + D(n,p)\right) \tag{9}$$

where $K(n,p,B)$ and $D(n,p)$ are suitable functions increasing in n. We remind that NO-SORT can be simulated on p processors where $p \leq f(n)$ and when $p > f(n)$ only $f(n)$ processors are used: then $H_S(n,p,B) = \Omega\left(H_S(n,f(n),B)\right)$ and $T_S(n,p) = \Omega\left(T_S(n,f(n))\right)$ for $p > f(n)$. Without loss of generality, we assume $K(n,p,B) = \Omega\left(K(n,f(n),B)\right)$, $D(n,p) = \Omega\left(D(n,f(n))\right)$, $H_S(n,p,B) = \mathcal{O}\left(K(n,p,B)\right)$ and $T_S(n,p) = \mathcal{O}\left(D(n,p)\right)$ when $p \geq f(n)$ (indeed, $n\log n/(Bp\log(n/p))$ and $(n/p)\log n$ are loose lower bounds when $p = \Omega\left(f(n)\right)$).

Theorem 15. *The network-oblivious algorithm NO-LR with input size n exhibits the following communication $H_{\text{NO-LR}}(n,p,B)$ and computation $T_{\text{NO-LR}}(n,p)$ complexities when executed on $\mathsf{M}(p,B)$, for $p \leq f(n)$:*

$$H_{\text{NO-LR}}(n,p,B) = \mathcal{O}\left(\frac{n}{Bp}\frac{\log n}{\log(1+n/p)} + (K(n,p,B) + \log p \log\log n + p/f(n)) \log n \log\log n\right) \tag{10}$$

$$T_{\text{NO-LR}}(n,p) = \mathcal{O}\left((n/p)\log n + (D(n,p) + (f(n)/p)\log f(n) \log\log n) \log n \log\log n\right). \tag{11}$$

Proof. NO-IS uses the network-oblivious algorithm NO-PS for computing a prefix sum of n values on $f(n)$ PEs given in Section 4.1: we remind its communication and computation complexities on $\mathsf{M}(p,B)$, for $p \leq f(n)$, are $\mathcal{O}(\log p)$ and $\mathcal{O}(n/p + (f(n)/p)\log n)$, respectively.

We remind that when NO-LR is executed on $\mathsf{M}(p,B)$ each processor simulates $f(n)/p$ consecutive numbered PEs, and contains $\mathcal{O}(n/p)$ nodes and in particular $\mathcal{O}(\lceil n_i/p \rceil)$ nodes of color i after Line 7.2 (this can be proved as in Lemma 5).

Lines 7.2,7.2, 7.2, 7.2 require communication and computation complexities $\mathcal{O}(H_S(n,p,B) + \log p \log\log n)$ and $\mathcal{O}(T_S(n,p) + (f(n)/p)\log f(n) \log\log n)$, respectively. The first term is due to sorting, while the remains are a consequence of key computation. Indeed, a call to COMPUTE_OFF$(i,0,n)$ for a given color i, for $0 \leq i < \log\log n$, requires $\mathcal{O}(i+1)$ operations and values $n_0, \ldots n_i$: since each processor contains $\mathcal{O}(n/p)$ nodes, key computation requires $\mathcal{O}((n/p)\log\log n)$ operations, which is negligible compared with sorting. Values $n_0, \ldots n_{\log\log n - 1}$ are computed and spread among PEs through $\mathcal{O}(\log\log n)$ prefix sum computations which in total require $\mathcal{O}(\log p \log\log n)$ blocks and $\mathcal{O}((f(n)/p)\log f(n) \log\log n)$ operations on $\mathsf{M}(p,B)$.

Line 7.2 requires $\mathcal{O}(n_i/(Bp) + \log\log n + \log p)$ communications and $\mathcal{O}(n_i/p + \log\log n + (f(n)/p)$ $\log f(n))$ operations: a PE contains $\mathcal{O}(n_i/p + \log\log n)$ nodes of G_i that are distributed to $\mathcal{O}(\log\log n)$ PEs (which may be simulated by $\mathcal{O}(\log\log n)$ distinct $\mathsf{M}(p,B)$ processors), after a prefix sum computation.

In Line 7.2, the $f(n)$ PEs are partitioned into segments of $f(n)/f(n_i)$ consecutive numbered PEs and then PEs in each segment send their G_i's nodes to the first PE in the segment: if $f(n_i) \geq p$, the communication complexity is $\mathcal{O}(n_i/(Bf(n_i)) + 1)$ since each segment is contained into $\mathcal{O}(1)$ processors; if $n_i \geq p$ and $f(n_i) < p$, the communication is $\mathcal{O}(n_i/(Bf(n_i)) + p/f(n_i))$ since a segment is contained into $\mathcal{O}(p/f(n_i))$ processors; if $n_i < p$ (hence, $f(n_i) < p$) the communication is $\mathcal{O}(n_i/(Bf(n_i)) + p/f(p))$ since the n_i nodes are contained into n_i processors (since they are evenly distributed among PEs/processors), and only $n_i/f(n_i) \leq p/f(p)$ PEs (which are simulated by at most $p/f(p)$ processors) within a segment sends messages to the their respective selected PE. We evince Line 7.2 exhibits $\mathcal{O}(n_i/(Bf(n_i)) + p/f(p))$ communication complexity and $\mathcal{O}(n_i/f(n_i))$ computation complexity.

Line 7.2 requires $\mathcal{O}(H_S(n_i, \min\{p, f(n_i)\}, B))$ messages and $\mathcal{O}(T_S(n_i, \min\{p, f(n_i)\}))$ operations: a call to NO-SORT$(n_i, f(n_i))$ requires $f(n_i)$ PEs and it is executed by $\min\{p, f(n_i)\}$ processors of $\mathsf{M}(p,B)$.

The communication and computation complexities of Lines 7.2 and 7.2 are $\mathcal{O}(H_S(n_i, \min\{p, f(n_i), B\})$ $+ \log p \log\log n)$ and $\mathcal{O}(T_S(n_i, \min\{p, f(n_i)\}) +$ $(f(n)/p)\log f(n) \log\log n)$, respectively: the second term is due to the computation of n_i^j, i.e. the number of duplicates of color j generated at the i-th iteration, for each $0 \leq j < \log\log n$, through $\mathcal{O}(\log\log n)$ prefix sum computations.

Finally, the cost of Line 7.2 is equivalent to the one of Line 7.2.

Remember that $H_S(n,p,B) = \Omega(H_S(n, f(n), B))$ if $p \geq f(n)$, and thus $\min\{p, f(n_i)\}$ can be replaced with p in the above bounds. Since $H_S(n_i, p, B) = \Omega(n_i/(Bf(n_i)))$ and $T_S(n_i, f(n_i)) =$

$\Omega\left(n_i/f(n_i)\right)$, we evince that the communication and computation complexities of NO-IS are:

$$H_{\text{NO-IS}}(n,p,B) = \mathcal{O}\left(H_S(n,p,B) + (\log p \log\log n + p/f(p))\log\log n + \sum_{i=0}^{\log\log n - 1} H_S(n_i,p,B)\right), \tag{12}$$

$$T_{\text{NO-IS}}(n,p) = \mathcal{O}\left(T_S(n,p) + (f(n)/p)\log f(n)(\log\log n)^2 + \sum_{i=0}^{\log\log n - 1} T_S(n_i,p)\right). \tag{13}$$

By plugging Equations 9 into previous ones, we get:

$$H_{\text{NO-IS}}(n,p,B) = \mathcal{O}\left(\frac{n}{Bp}\frac{\log n}{\log(1+n/p)} + (K(n,p,B) + \log p \log\log n + p/f(n))\log\log n\right)$$
$$T_{\text{NO-IS}}(n,p) = \mathcal{O}\left((n/p)\log n + (D(n,p) + (f(n)/p)\log f(n)\log\log n)\log\log n\right).$$

NO-LR consists of $\mathcal{O}(\log n)$ recursive calls of size n/c^i for a suitable constant $c > 1$. The communication complexity is given by the following recurrence:

$$H_{\text{NO-LR}}(n,p,B) = \begin{cases} H_{\text{NO-LR}}\left(n/c, \min\{p, f(n/c)\}, B\right) + \mathcal{O}\left(H_{\text{NO-IS}}(n,p,B)\right) & \text{if } n > \mathcal{O}(1) \\ 0 & \text{otherwise} \end{cases}$$

When $p \geq f(n)$, $K(n,p,B) = \Omega\left(K(n,f(n),B)\right)$ and $H_S(n,p,B) = \mathcal{O}\left(K(n,p,B)\right)$ (i.e., $K(n,p,B) = \Omega\left(n\log n/(Bf(n)\log(n/f(n)))\right)$). Then, it is not difficult to see that the above recurrence solves to Equation 10. A similar proof gives Equation 11 as well. □

The above theorem provides a general result which applies to each $K(n,p,B)$ and $D(n,p)$. In many cases (as we will see in Lemma 7), the stated upper bounds can be improved using properties of $K(n,p,B)$ and $D(n,p)$.

Lemma 6. *The execution of NO-LR on $\mathsf{M}(p,B)$ requires $\mathcal{O}\left(n/p + (f(n)/p)\log\log n\right)$ words of memory, which is optimal when $f(n) \leq n/\log\log n$.*

Proof. Each PE requires $\mathcal{O}(n/f(n))$ words for storing list nodes and duplicates, and $\mathcal{O}(\log\log n)$ words for storing the $\log\log n$ values n_i required by COMPUTE_OFF. Since each $\mathsf{M}(p,B)$ processor simulates $f(n)/p$ PEs, the statement follows. □

Suppose NO-LR adopts as sorting algorithm NO-CS, which is described in Section 4.1. In this case, upper bounds on the communication and computation complexities of NO-LR are given by specializing results in Theorem 15. However, better upper bounds are given by an ad-hoc proof, provided in the following lemma.

Lemma 7. *Suppose NO-LR uses the network-oblivious algorithm NO-CS as sorting primitive. Then, the upper bounds on NO-LR given in Theorem 15 become:*

$$H_{\text{NO-LR}}(n,p,B) = \mathcal{O}\left(\frac{n}{Bp} + \left(\sqrt{\frac{n}{p}} + \sqrt{\frac{n^\epsilon}{B}} + p^\epsilon\right)\log n \log\log n\right), \tag{14}$$

$$T_{\text{NO-LR}}(n,p) = \Theta\left(\frac{n}{p}\log n + n^\epsilon \log^2 n \log\log n\right) \tag{15}$$

for $p \leq n^{1-\epsilon}$. When $p \leq n^{1-\epsilon}/\log n \log\log n$ and $B = \mathcal{O}\left(\min\{\sqrt{n/p}, n/p^{1+\epsilon}\}/(\log n \log\log n)\right)$, the algorithm has optimal communication and computation complexities.

An ad-hoc analysis gives the following improved upper bounds for NO-LR:

$$H_{\text{NO-LR}}(n,p,B) = \mathcal{O}\left(\frac{n}{Bp} + \sqrt{\frac{n}{p}\log\log n} + \frac{n^\epsilon}{B}(\log\log n)^{1-\epsilon} + n^{\frac{\epsilon}{2}}(\log\log n)^{1-\frac{\epsilon}{2}} + p^\epsilon \log n \log\log n\right) \tag{16}$$

$$T_{\text{NO-LR}}(n,p) = \mathcal{O}\left(\frac{n}{p}\log n + n^\epsilon \log n (\log\log n)^{1-\epsilon}\right). \tag{17}$$

for $p \leq n^{1-\epsilon}$. *When* $p \leq (n/\log\log n)^{1-\epsilon}$ *and* $B = \mathcal{O}\left(\min\{\sqrt{(n\log\log n)/p}, n/(p^{1+\epsilon}\log n)\}/\log\log n\right)$, *the algorithm has optimal communication and computation complexities.*

Proof. Remember from Corollary 2, that NO-CS is defined on $f(n) = n^{1-\epsilon}$ PEs for any arbitrary constant $\epsilon \in (0,1)$. In order to use Theorem 8, we have to rewrite upper bounds on communication and computation complexities of NO-CS in such a way $H_S(n,p,B) = \Omega(H_S(n,f(n),B))$ and $T_S(n,p,B) = \Omega(T_S(n,f(n),B))$ for $p > f(n)$. Then, we get:

$$H_{\text{NO-CS}}(n,p,B) = \mathcal{O}\left(\frac{n}{Bp} + \sqrt{\frac{n}{p}} + \frac{n^\epsilon}{B} + n^{\epsilon/2}\right), \qquad T_{\text{NO-CS}}(n,p) = \mathcal{O}\left(\frac{n}{p}\log n + n^\epsilon \log n\right).$$

We evince that $K(n,p,B) = \mathcal{O}\left(\sqrt{n/p} + n^\epsilon/B + n^{\epsilon/2}\right)$ and $D(n,p) = \mathcal{O}(n^\epsilon \log n)$. Equations 14 and 15 follows from Theorem 15.

Better upper bounds follow observing that both $K(n,p,B)$ and $D(n,p)$ are concave and then Equation 12 in the proof of Theorem 15 is maximized when $n_i = n/\log\log n$ and we get:

$$H_{\text{NO-IS}}(n,p,B) = \mathcal{O}\left(\frac{n}{Bp} + \sqrt{\frac{n}{p}\log\log n} + \frac{n^\epsilon}{B}(\log\log n)^{1-\epsilon} + n^{\epsilon/2}(\log\log n)^{1-\epsilon/2} + p^\epsilon \log\log n\right),$$

$$T_{\text{NO-IS}}(n,p) = \mathcal{O}\left(\frac{n}{p}\log n + n^\epsilon \log n (\log\log n)^{1-\epsilon}\right).$$

for $p \leq n^{1-\epsilon}$. Since each term, but the last, decreases geometrically in n, Equations 16 and 17 follow. □

Corollary 4. *Suppose NO-LR uses the network-oblivious algorithm NO-CS as sorting primitive. Then NO-LR when executed on a D-BSP($P, \boldsymbol{g}, \boldsymbol{B}$) with input size n exhibits optimal communication time,*

$$D(n,P,\boldsymbol{g},\boldsymbol{B}) = \Theta\left(\frac{ng_0}{PB_0}\right)$$

for $P \leq (n/\log\log n)^{1-\epsilon}$ *and* $B_0 = \mathcal{O}\left(\min\{\sqrt{(n\log\log n)/p}, n/(p^{1+\epsilon}\log n)\}/\log\log n\right)$.

Proof. The upper bound is straightforward, and optimality can be proved as is Lemma 2. □

7.3 Euler Tours and Graph Connectivity

As seen in Section 7.1, a number of tree problems are solved by computing an Euler tour and then performing a list ranking on the tour with appropriate weights. A network-oblivious algorithm for Euler tours can be derived from the multicore-oblivious one. More in details the algorithm, named NO-ET, is defined on $\mathsf{M}(f(n))$, where $f(n)$ is defined as in Section 7.2. (The NO-ET is defined for $f(n)$ PEs since is often followed by NO-LR in many applications as we will see). The n adjacency lists of the input tree T are distributed as follows: the lists are concatenated and then envisioned as an unique array which is distributed evenly among PEs as usual: it is easy to see that the list of a node i is distributed among $\mathcal{O}\left(\lceil d_i f(n)/n \rceil\right)$ consecutive numbered PEs, where d_i is the degree of i. We suppose that, whenever a list is distributed among many PEs, these PEs know in which PE the list head is stored and to which node the list belongs (this assumption may be obtained by an adaptation of the network-oblivious prefix sum algorithm). A list can be made circular without communications if it is contained in one PE, or through $\mathcal{O}(1)$ messages exchanged between the PEs containing the head and the tail. Since the successor of arc (u, v) is (v, u'), where u' is the successor of u in the adjacency list of v, the computation of the tour requires $\mathcal{O}(1)$ messages exchanged between adjacent PEs: indeed, at most one successor of nodes stored in a PE is stored in the adjacent PE.

Theorem 16. *The network-oblivious NO-ET with input size n exhibits $\mathcal{O}(1)$ communication complexity and $\mathcal{O}(n/p)$ computation complexity when executed on $\mathsf{M}(p, B)$ with $p \leq f(n)$.*

Proof. Making the list circular requires $\mathcal{O}(1)$ messages in $\mathsf{M}(p, B)$ since each processor of $\mathsf{M}(p, B)$ contains at most two incomplete adjacency lists. The actual computation of the tour requires $\mathcal{O}(1)$ messages because communication on $\mathsf{M}(f(n))$ is between adjacent PEs. □

In the remaining part of the section, we suppose that NO-CS is used as sorting primitive in NO-LR and then $f(n) = n^{1-\epsilon}$, where ϵ is an arbitrary constant in $(0, 1)$.

Corollary 5. *There exist network-oblivious algorithms defined on $\mathsf{M}(n^{1-\epsilon})$, where ϵ is an arbitrary constant in $(0, 1)$, for rooting a tree, traversal numbering, vertex depth, subtree size and connected components in a forest, whose communication and computation complexities on an $\mathsf{M}(p, B)$ are*

$$H(n, p, B) = \mathcal{O}\left(\frac{n}{Bp} + \sqrt{\frac{n}{p}\log\log n} + \frac{n^{\epsilon}}{B}(\log\log n)^{1-\epsilon} + n^{\epsilon/2}(\log\log n)^{1-\epsilon/2} + p^{\epsilon}\log n \log\log n\right),$$

$$T(n, p) = \mathcal{O}\left(\frac{n}{p}\log n + n^{\epsilon}\log n(\log\log n)^{1-\epsilon}\right)$$

for $p \leq n^{1-\epsilon}$. When $p \leq (n/\log\log n)^{1-\epsilon}$ and $B = \mathcal{O}\left(\min\{\sqrt{(n\log\log n)/p}, n/(p^{1+\epsilon}\log n)\}/\log\log n\right)$, the algorithm has optimal communication and computation complexities.

Proof. As seen in a previous section the cited problems can be solved through the computation of an Euler tour and of its list ranking with suitable weights. The stated results follow by using NO-CS as sorting primitive in NO-LR and upper bounds in Lemma 7 and Theorem 16 □

We observe that the adjacency lists of a tree can be constructed from the set of undirected arcs by replacing each undirected arch (u, v) with two direct arcs (u, v) and (v, u) and then sorting them lexicographically.

A network-oblivious algorithm for computing connected components of a graph is given by a straightforward adaptation of the multicore-oblivious algorithm given in Section 13 which uses the network-oblivious algorithms for sorting, list ranking and connected components in a tree.

Corollary 6. *There exists a 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, whose communication and computation complexities on an $\mathsf{M}(p, B)$ are*

$$H(n, p, B) = \mathcal{O}\left(\frac{N(n,m)}{Bp} + \sqrt{\frac{N(n,m)}{p}\varGamma(n)} + \frac{N(n,m)^{\epsilon}}{B}\varGamma(n)^{1-\epsilon} + N(n,m)^{\epsilon/2}\varGamma(n)^{1-\epsilon/2} + p^{\epsilon}\log n\varGamma(n)\right) \tag{18}$$

$$T(n, p) = \mathcal{O}\left(\frac{N(n,m)}{p}\log n + N(n,m)^{\epsilon}\varGamma(n)^{1-\epsilon}\log n\right), \tag{19}$$

where $N(n,m) = n + m\log n$, $\Gamma(n) = \log n \log\log n$ *and* $p \le (n+m)^{1-\epsilon}$.

Proof. The network-oblivious algorithm consists of $\log n$ recursive calls of size $\mathcal{O}\left(n/2^i + m\right)$ and each recursive call is dominated by the computation of the list ranking. Then by using NO-CS as sorting primitive in NO-LR, we get that the communication complexity of the algorithm is asymptotically upper bounded by the following summation:

$$\sum_{i=0}^{\log n-1} \frac{\tilde{n}_i + m}{Bp} + \sqrt{\frac{\tilde{n}_i+m}{p}\log\log n} + \frac{(\tilde{n}_i+m)^{\epsilon}}{B}(\log\log n)^{1-\epsilon} + (\tilde{n}_i+m)^{\epsilon/2}(\log\log n)^{1-\epsilon/2} + \\ + p^{\epsilon}\log n\log\log n$$

where $n_i = n/2^i$ and $\sum_{i=0}^{\log n-1}\tilde{n}_i < 2n$. Since the second, third and fourth terms are concave, Equation 18 follows by setting $n_i = n/\log n$ in the concave terms. Equation 19 can be proved similarly. □

8 Conclusion

In this paper we have addressed the design of parallel algorithms that are oblivious to machine parameters for two dominant machine configurations: the chip multicores and the network of processors.

Initially, we have focused on multicores and described the Hierarchical Memory (HM) model, which generalizes the 3-memory-level multicore model in [8] to any number of memory levels $h \ge 3$. Then, we have introduced the notion of multicore-oblivious algorithm, that is, an algorithm that does not use specific values for multicore parameters such as number of cores, number of levels of caches or their sizes, block sizes, etc., yet performs efficiently across a wide variety of multicores. To address this challenge, we have proposes some simple enhancements to HM algorithms, which consist in 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. The multicore-oblivious algorithms we have presented use two main types of scheduler hints: the Coarse-Grained Contiguous (CGC) scheduling and the Space-Bound (SB) scheduling. We also have proposed a third hint (CGC on SB) that combines these two. By using these hints, the algorithm suggests the run-time scheduler how it should distribute computation among cores and caches, while it remains oblivious of machine parameters since the actual distribution is performed by the scheduler.

We have instantiated this approach with provably efficient multicore-oblivious algorithms for a number of fundamental problems: matrix computations (matrix multiplication and transposition), prefix sum computation, FFT, Gaussian Elimination paradigm, sorting, list ranking, Euler tours, a number of tree problems based on Euler tours, and connected components.

Building on some of the above multicore-oblivious algorithms, we have developed novel and efficient network-oblivious algorithms. The concept of network-oblivious algorithm was introduced in [7] and refers to an algorithm for a distributed-memory model which does not depend on the characteristic of the machine, e.g. processor number, bandwidth and latency parameters. We have proposed efficient network-oblivious algorithms for sorting, Gaussian Elimination paradigm, list ranking, Euler tours, a number of tree problems based on Euler tours, and connected components. These algorithms are appealing because in many cases they exhibit optimal performance on the Decomposable-BSP model [5], which effectively describes a wide and significant class of parallel platforms.

It worth to be noticed that in many cases the multicore and network-oblivious algorithms for a given problem are based on the same ideas, which work also for cache-oblivious algorithms. Nevertheless, multicore and network-oblivious algorithms have substantial differences due to the different model in which they are specified. An example is the network-oblivious algorithm for GEP, named N-GEP which, in order to achieve optimal communication, is somewhat modified from I-GEP.

In conclusion we propose a summarizing table containing the main results of the paper. The second column gives the time/computation complexity of multicore and network-oblivious algorithms which coincide under some assumptions; the third and fifth columns provide the cache and communication complexities of multicore and network-oblivious algorithms respectively; the fourth and sixth columns provide the parameter ranges for which the entries in table are verified (note that the are proposed in a simplified form for readability). We remind that network-oblivious algorithms for matrix multiplication, transposition, FFT and prefix sum computation were proposed in [7, 6].

Problem	**MO and NO time**	**MO Cache**	**MO Range**[7]	**NO Communication**	**NO Range**
Prefix sum	$\Theta\left(\frac{n}{p}\right)$	$\Theta\left(\frac{n}{q_i B_i}\right)$	$p \leq n/\log n$	$\Theta(\log p)$	$p \leq n/\log n$, $B \geq 1$
Matrix transposition	$\Theta\left(\frac{n^2}{p}\right)$	$\mathcal{O}\left(\frac{n^2}{q_i B_i} + B_i\right)$	$p \leq n^2$, $B_i^2 \leq C_i$	$\mathcal{O}\left(\frac{n^2}{Bp} + \sqrt{\frac{n}{p}}\right)$	$p \leq n^2$, $B \geq 1$
Matrix multiplication	$\Theta\left(\frac{n^3}{p}\right)$	$\Theta\left(\frac{n^3}{q_i B_i \sqrt{C_i}}\right)$	$p \leq n^2$, $B_i^2 \leq C_i$	$\Theta\left(\frac{n^2}{B\sqrt{p}}\right)$	$p \leq n^2$, $B \leq n^2/p$
GEP	$\Theta\left(\frac{n^3}{p}\right)$	$\Theta\left(\frac{n^3}{q_i B_i \sqrt{C_i}}\right)$	$p \leq n^2/\log^2 n$, $B_i^2 \leq C_i$; $C_i = \Omega\left(p_i C_{i-1} \log^2 \frac{C_i}{C_{i-1}}\right)$	$\mathcal{O}\left(\frac{n^2}{B\sqrt{p}} + n \log^2 n\right)$	$p \leq n^2/\log^2 n$, $B \geq 1$
FFT	$\Theta\left(\frac{n \log n}{p}\right)$	$\Theta\left(\frac{n}{q_i B_i} \log_{C_i} n\right)$	$p \leq n$, $B_i^2 \leq C_i$	$\Theta\left(\frac{n}{pB} \log_{(1+n/p)} n\right)$	$p \leq n$, $B \leq \sqrt{n/p}$
Sorting	$\Theta\left(\frac{n}{p} \log n\right)$	$\Theta\left(\frac{n}{q_i B_i} \log_{C_i} n\right)$	$p \leq n/\log\log n$, $B_i^2 \leq C_i$	$\mathcal{O}\left(\frac{n}{pB} + \sqrt{\frac{n}{p}}\right)$	$p \leq n^{1-\epsilon}$, ϵ arbitrary constant in $(0,1)$, $B \geq 1$
List ranking	$\Theta\left(\frac{n}{p} \log n\right)$	$\mathcal{O}\left(\frac{n}{q_i B_i} \log_{C_i} n + (\log\log n)^2 \log \frac{n}{B_i}\right)$	$p \leq \frac{n}{(\log\log n)^2 \log n}$, $B_i^2 \leq C_i$	$\mathcal{O}\left(\frac{n}{Bp} + \sqrt{\frac{n}{p} \log\log n} + p^\epsilon \log n \log\log n\right)$	$p \leq (n/\log\log n)^{1-\epsilon}$, ϵ arbitrary constant in $(0,1)$, $B \geq 1$

Acknowledgments. The authors would like to thank Andrea Pietracaprina and Keshav Pingali for useful discussions.

References

1. L. Arge, M. Goodrich, and N. Sitchinava. Parallel external memory graph algorithms. 2009.
2. L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proc. of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 197–206, New York, NY, USA, 2008. ACM.
3. D. Bailey. FFTs in external or hierarchical memory. *Journal of Supercomputing*, 4:23–35, 1990.
4. M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proc. of the 17th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 228–237, New York, NY, USA, 2005. ACM.
5. G. Bilardi, A. Pietracaprina, and G. Pucci. Decomposable BSP: A bandwidth-latency model for parallel and hierarchical computation. In J. Reif and S. Rajasekaran, editors, *Handbook of Parallel Computing: Models, Algorithms and Applications*, pages 277–315. CRC Press, 2007.
6. G. Bilardi, A. Pietracaprina, G. Pucci, and F. Silvestri. manuscript in preparation.
7. G. Bilardi, A. Pietracaprina, G. Pucci, and F. Silvestri. Network-oblivious algorithms. In *Proc. of the 21st IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, 2007.
8. G. Blelloch, R. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proc. of the 19th ACM-SIAM Symposium on Discrete Algorithms*, pages 501–510, Philadelphia, PA, USA, 2008. SIAM.
9. G. Blelloch and P. Gibbons. Effectively sharing a cache among threads. In *Proc. of the 16th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 235–244, New York, NY, USA, 2004. ACM.
10. G. Blelloch, P. Gibbons, and H. Simhadri. Brief announcement: Low depth cache-oblivious sorting. In *Proc. of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, New York, NY, USA, 2009. ACM.

[7] Some (minor) constraints relating to cache parameters are omitted.

11. B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.
12. R. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proc. of the 17th ACM-SIAM Symposium on Discrete Algorithms*, pages 591–600, New York, NY, USA, 2006. ACM.
13. R. Chowdhury and V. Ramachandran. The cache-oblivious Gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. In *Proc. of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 71–80, New York, NY, USA, 2007. ACM.
14. R. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proc. of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 207–216, New York, NY, USA, 2008. ACM.
15. R. Cole and V. Ramachandran. Resource oblivious multicore sorting, 2009.
16. R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Inf. Control*, 70(1):32–53, 1986.
17. R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Inf. Comput.*, 81(3):334–352, 1989.
18. J. Cooley and J. Tukey. An algorithm for the machine computation of the complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.
19. P. Duhamel and M. Vetterli. Fast Fourier transforms: a tutorial review and a state of the art. *Signal Process.*, 19(4):259–299, 1990.
20. M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. of the 40th IEEE Symposium on Foundations of Computer Science*, pages 285–297, Washington, DC, USA, 1999. IEEE.
21. M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *Proc. of the 18th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 271–280, New York, NY, USA, 2006. ACM.
22. M. T. Goodrich. Communication-efficient parallel sorting. *SIAM Journal on Computing*, 29(2):416–432, 1999.
23. D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Commun. ACM*, 22(8):461–464, 1979.
24. D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.
25. J. Jaja. *An Introduction to Parallel Algorithms.* Addison-Wesley Professional, March 1992.
26. R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, 1980.
27. F. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1991.
28. R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. Applied Mathematics*, 36:177–189, 1979.
29. F. Silvestri. *Oblivious Computations on Memory and Network Hierarchies.* PhD thesis, Department of Information Engineering, University of Padova, 2009.
30. R. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Computing*, 14:862–874, 1984.
31. P. d. l. Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting (extended abstract). In *Proc. of the 2nd International Euro-Par Conference on Parallel Processing-Volume II*, volume 1124 of *LNCS*, pages 352–358, London, UK, 1996. Springer-Verlag.
32. L. G. Valiant. A bridging model for multi-core computing. In *Proc. of the 16th Annual European Symposium*, volume 5193 of *LNCS*, pages 13–28, Berlin, Heidelberg, 2008. Springer-Verlag.
33. J. Vitter and M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12:148–169, 1994.
34. J. C. Wyllie. *The Complexity of Parallel Computations.* PhD thesis, Cornell University, Ithaca, NY, USA, 1979.

Appendix 1: N-GEP's functions $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$

For convenience, we give the pseudocodes of N-GEP's functions $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$. The construct **sync** indicates the synchronization at the end of a superstep (superstep labels are not reported for simplicity) and the assignment $L_2 \leftarrow L_1$, with L_1 and L_2 matrices of equal dimension, involves the copy of each entry of L_1 into the corresponding entry of L_2 and is achieved by means of a suitable communication among the PEs.

```
$\mathcal{A}(X, m, \mathcal{P})$
Input: $m \times m$ matrix $X \equiv x[I, I]$ with $I$ interval in $[0, n)$; set $\mathcal{P}$ of consecutive numbered PEs assigned to $\mathcal{A}$.
Output: execution of all updates $\langle i, j, k \rangle \in T$, with $T = \Sigma_f \cap (I \times I \times I)$.
 1: if $T = \emptyset$ then return; if $m = 1$ then $X[0,0] \leftarrow f(X[0,0], X[0,0], X[0,0], X[0,0])$ and return;
 2: Let $\mathcal{P}_0$ and $\mathcal{P}_1$ be the partition of $\mathcal{P}$ where each set contains $|\mathcal{P}|/2$ consecutive numbered PEs;
 3: Allocate space for eight $m/2 \times m/2$ matrices $\tilde{X}$, $\tilde{X}_0$, $\tilde{X}_1$, $U$, $U'$, $V$, $V'$ and $W$, distributed as follows: $U'$ and $\tilde{X}_0$
    (resp., $V'$ and $\tilde{X}_1$) are evenly distributed in row-major among $\min\{|\mathcal{P}|/2, m^2/4\}$ PEs of $\mathcal{P}_0$ (resp., $\mathcal{P}_1$); $\tilde{X}$, $U$, $V$
    and $W$ are evenly distributed in row-major among $\min\{|\mathcal{P}|, m^2/4\}$ PEs of $\mathcal{P}$;
 4: $\tilde{X} \leftarrow X_{0,0}$; sync; {The input to the next recursive call is stored in the support matrix $\tilde{X}$}
 5: $\mathcal{A}(\tilde{X}, m/2, \mathcal{P})$;
 6: $X_{0,0} \leftarrow \tilde{X}$; sync; {The output of the recursive call is stored in $X_{0,0}$}
 7: $\tilde{X}_0 \leftarrow X_{0,1}$, $\tilde{X}_1 \leftarrow X_{1,0}$, $U' \leftarrow X_{0,0}$, $V' \leftarrow X_{0,0}$; sync;
 8: in parallel: $\mathcal{B}(\tilde{X}_0, U', m/2, \mathcal{P}_0)$, $\mathcal{C}(\tilde{X}_1, V', m/2, \mathcal{P}_1)$;
 9: $X_{0,1} \leftarrow \tilde{X}_0$, $X_{1,0} \leftarrow \tilde{X}_1$; sync;
10: $\tilde{X} \leftarrow X_{1,1}$, $U \leftarrow X_{1,0}$, $V \leftarrow X_{0,1}$, $W \leftarrow X_{0,0}$; sync;
11: $\mathcal{D}^*(\tilde{X}, U, V, W, m/2, \mathcal{P})$;
12: $\mathcal{A}(\tilde{X}, m/2, \mathcal{P})$;
13: $X_{1,1} \leftarrow \tilde{X}$; sync;
14: $\tilde{X}_0 \leftarrow X_{1,0}$, $\tilde{X}_1 \leftarrow X_{0,1}$, $U' \leftarrow X_{1,1}$, $V' \leftarrow X_{1,1}$; sync;
15: in parallel: $\mathcal{B}(\tilde{X}_0, U', m/2, \mathcal{P}_0)$, $\mathcal{C}(\tilde{X}_1, V', m/2, \mathcal{P}_1)$;
16: $X_{1,0} \leftarrow \tilde{X}_0$, $X_{0,1} \leftarrow \tilde{X}_1$; sync;
17: $\tilde{X} \leftarrow X_{0,0}$, $U \leftarrow X_{0,1}$, $V \leftarrow X_{1,0}$, $W \leftarrow X_{1,1}$; sync;
18: $\mathcal{D}^*(\tilde{X}, U, V, W, m/2, \mathcal{P})$;
19: $X_{0,0} \leftarrow \tilde{X}$; sync;
20: Delete the eight temporary matrices;
```

```
$\mathcal{B}(X, U, m, \mathcal{P})$
Input: $m \times m$ matrices $X \equiv x[I, J]$ and $U \equiv x[I, I]$, with
    $I, J$ intervals in $[0, n)$ and $I \cap J = \emptyset$; set $\mathcal{P}$ of consecutive
    PEs assigned to $\mathcal{B}$.
Output: execution of all updates $T = \Sigma_f \cap (I \times J \times I)$.
 1: if $T = \emptyset$ then return; if $m = 1$ or $|\mathcal{P}| = 1$ then Solve
    the problem sequentially with I-GEP and return;
 2: Let $\mathcal{P}_0$ and $\mathcal{P}_1$ be the partition of $\mathcal{P}$ where each set
    contains $|\mathcal{P}|/2$ consecutive numbered PEs;
 3: Allocate space for eight $m/2 \times m/2$ matrices $\tilde{X}_i$, $U_i$,
    $V_i$, and $W_i$ for $i \in \{0, 1\}$ in such a way that $\tilde{X}_i$, $U_i$,
    $V_i$ and $W_i$ are evenly distributed in row-major among
    $\min\{|\mathcal{P}|/2, m^2/4\}$ PEs of $\mathcal{P}_i$;
 4: $\tilde{X}_i \leftarrow X_{0,i}$, $U_i \leftarrow U_{0,0}$ $\forall i \in \{0, 1\}$; sync;
 5: in parallel: $\mathcal{B}(\tilde{X}_i, U_i, m/2, \mathcal{P}_i)$ $\forall i \in \{0, 1\}$;
 6: $X_{0,i} \leftarrow \tilde{X}_i$ $\forall i \in \{0, 1\}$; sync;
 7: $\tilde{X}_i \leftarrow X_{1,i}$, $U_i \leftarrow U_{1,0}$, $V_i \leftarrow X_{0,i}$, $W_i \leftarrow U_{0,0}$ $\forall i \in$
    $\{0, 1\}$; sync;
 8: in parallel: $\mathcal{D}^*(\tilde{X}_i, U_i, V_i, W_i, m/2, \mathcal{P}_i)$ $\forall i \in \{0, 1\}$;
 9: $X_{1,i} \leftarrow \tilde{X}_i$ $\forall i \in \{0, 1\}$; sync;
10: $\tilde{X}_i \leftarrow X_{1,i}$, $U_i \leftarrow U_{1,1}$ $\forall i \in \{0, 1\}$; sync;
11: in parallel: $\mathcal{B}(\tilde{X}_i, U_i, m/2, \mathcal{P}_i)$ $\forall i \in \{0, 1\}$;
12: $X_{1,i} \leftarrow \tilde{X}_i$ $\forall i \in \{0, 1\}$; sync;
13: $\tilde{X}_i \leftarrow X_{0,i}$, $U_i \leftarrow U_{0,1}$; $V_i \leftarrow X_{1,i}$, $W_i \leftarrow U_{1,1}$ $\forall i \in$
    $\{0, 1\}$; sync;
14: in parallel: $\mathcal{D}^*(\tilde{X}_i, U_i, V_i, W_i, m/2, \mathcal{P}_i)$ $\forall i \in \{0, 1\}$;
15: $X_{0,i} \leftarrow \tilde{X}_i$ $\forall i \in \{0, 1\}$; sync;
16: Delete the eight temporary matrices;
```

```
$\mathcal{C}(X, V, m, \mathcal{P})$
Input: $m \times m$ matrices $X \equiv x[I, J]$ and $V \equiv x[J, J]$, with
    $I, J$ intervals in $[0, n)$ and $I \cap J = \emptyset$; set $\mathcal{P}$ of consecutive
    PEs assigned to $\mathcal{C}$.
Output: execution of all updates in $T = \Sigma_f \cap (I \times J \times J)$.
 1: if $T = \emptyset$ then return; if $m = 1$ or $|\mathcal{P}| = 1$ then Solve
    the problem sequentially with I-GEP and return;
 2: Let $\mathcal{P}_0$ and $\mathcal{P}_1$ be the partition of $\mathcal{P}$ where each set
    contains $|\mathcal{P}|/2$ consecutive numbered PEs;
 3: Allocate space for eight $m/2 \times m/2$ matrices $\tilde{X}_i$, $U_i$,
    $V_i$, and $W_i$ for $0 \leq i \leq 1$ in such a way that $\tilde{X}_i$, $U_i$,
    $V_i$ and $W_i$ are evenly distributed in row-major among
    $\min\{|\mathcal{P}|/2, m^2/4\}$ PEs of $\mathcal{P}_i$;
 4: $\tilde{X}_i \leftarrow X_{i,0}$, $V_i \leftarrow V_{0,0}$ $\forall i \in \{0, 1\}$; sync;
 5: in parallel: $\mathcal{C}(\tilde{X}_i, V_i, m/2, \mathcal{P}_i)$ $\forall i \in \{0, 1\}$; sync;
 6: $X_{i,0} \leftarrow \tilde{X}_i$ $\forall i \in \{0, 1\}$; sync;
 7: $\tilde{X}_i \leftarrow X_{i,1}$, $U_i \leftarrow X_{i,0}$, $V_i \leftarrow V_{0,1}$, $W_i \leftarrow V_{0,0}$ $\forall i \in$
    $\{0, 1\}$; sync;
 8: in parallel: $\mathcal{D}^*(\tilde{X}_i, U_i, V_i, W_i, m/2, \mathcal{P}_i)$ $\forall i \in \{0, 1\}$;
 9: $X_{i,1} \leftarrow \tilde{X}_i$ $\forall i \in \{0, 1\}$; sync;
10: $\tilde{X}_i \leftarrow X_{i,1}$, $V_i \leftarrow V_{1,1}$ $\forall i \in \{0, 1\}$; sync;
11: in parallel: $\mathcal{C}(\tilde{X}_i, V_i, m/2, \mathcal{P}_i)$ $\forall i \in \{0, 1\}$;
12: $X_{i,1} \leftarrow \tilde{X}_i$ $\forall i \in \{0, 1\}$; sync;
13: $\tilde{X}_i \leftarrow X_{i,0}$, $U_i \leftarrow X_{i,1}$; $V_i \leftarrow V_{1,0}$, $W_i \leftarrow V_{1,1}$ $\forall i \in$
    $\{0, 1\}$; sync;
14: in parallel: $\mathcal{D}^*(\tilde{X}_i, U_i, V_i, W_i, m/2, \mathcal{P}_i)$ $\forall i \in \{0, 1\}$;
15: $X_{i,0} \leftarrow \tilde{X}_i$ $\forall i \in \{0, 1\}$; sync;
16: Delete the eight temporary matrices;
```