Oblivious Algorithms for Multicores and Network of Processors

Rezaul Alam Chowdhury*, Francesco Silvestri†, Brandon Blakeley* and Vijaya Ramachandran*

*Department of Computer Sciences
University of Texas
Austin, TX 78712, USA
{shaikat,blakeley,vlr}@cs.utexas.edu

† Department of Information Engineering
University of Padova
Via Gradenigo 6/B, 35131 Padova, Italy
silvest1@dei.unipd.it

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

***Keywords*-multicore; cache; network; oblivious; algorithm; Gaussian elimination paradigm; list ranking**

I. Introduction

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

This work was supported in part by NSF Grant CCF-0514876 and NSF CCF-0850775; the second author was also supported in part by the European Union under the FP6-IST/IP Project AEOLUS, and by University of Padova under the Strategic Project STPD08JA32. Part of this work was done while the second author was visiting the Department of Computer Sciences, University of Texas, Austin.

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

Efficient algorithms for multicores must address both *caching issues* and shared-memory *parallelism*, and in recent years, a number of models, algorithms and schedulers for multicores have been proposed. In its simplest form, a multicore is modeled as a collection of processing elements or *cores* sharing an arbitrarily large main memory containing all data and featuring one level of cache which could be either private (e.g., [5]–[8]) or shared among all the cores (e.g., [7], [9]). Since multicores are evolving towards a hierarchy of caches, a 3-level model was introduced in [10], which consists of a collection of cores, each with a private L_1 cache, sharing an arbitrarily large main memory through a shared L_2 cache; in [11] a multi-level version of this model is briefly introduced. Algorithms for this model are given in [10], [11]. The Multi-BSP [12] is a hierarchical shared-memory model that uses latency and gap parameters in a bulk synchronous manner. Most of the multicore algorithms in the literature are resource-aware, that is, they make use of some machine parameters in their specifications.

A. Our Results

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

Multi-BSP [12]).

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

We observe that our notion of MO algorithms is complementary to that of NO algorithms [4]: the former is defined in a shared-memory model, while the latter is defined in a distributed-memory model with point-to-point communications. Nevertheless, in many cases, a problem can be solved by MO and NO algorithms through similar strategies, and in fact, our MO algorithms for matrix transposition and FFT are adapted from their NO counterparts in [4]. We further reinforce this connection by deriving efficient NO algorithms for GEP, list ranking, connected components, and other graph problems by adapting our MO algorithms. These results enrich the body of known efficient NO algorithms and raise hopes for a unified notion of obliviousness in parallel computation. A summary of our key results appears in Table II at the end of the paper.

B. *Paper Organization*

In Section II we present the HM model. In Section III we describe three types of scheduler hints to support multicore-obliviousness, and illustrate them with MO algorithms for matrix transposition, sparse matrix dense vector multiplication, sorting, and FFT. In Section IV we review the NO framework. We present efficient MO and NO algorithms for important applications of GEP in Section V, and for list ranking, connected components, and other graph problems in Section VI. More details and several additional results are in the full paper [13].

II. THE HM MODEL

The hierarchical multi-level multicore (HM) model with h levels consists of a collection of cores P_i, $1 \leq i \leq p$, with 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 cache 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., $p'_i = p/q_i = \prod_{j=1}^{i} p_j$. We assume that $p_1 = 1$, that is, each core has a private cache at the first level. We also assume that $p_h = 1$, so that the top two levels $h-1$ and h represent a possible sequential cache hierarchy at the highest level [1], and that $C_i \geq c_i \cdot p_i \cdot C_{i-1}$ for all i, for a suitable constants $c_i \geq 1$. The constraints on relative cache sizes imply the following upper bound on the number of cores, $p = \prod_{i=1}^{h} p_i = \prod_{i=2}^{h-1} p_i \leq K \cdot C_{h-1}/C_1$, where $K = \prod_{i=2}^{h-1} (1/c_i)$. Figure 1 shows the model for $h = 5$. More details of the model are in the full paper [13].

The performance of an HM algorithm is determined by the number of parallel steps executed assuming all cores run at the same rate (*parallel time complexity*), and the maximum number of block transfers into and out of any single L_i, $i \in [1, h-1]$ (*cache complexity*).

The multicore model in [10] is the HM model with $h = 3$. As noted there for the 3-level multicore model, there is an inherent tension between cache-efficient scheduling for private 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 fine-grained schedule where all cores work on the portion of the data present in the shared L_2 cache is effective. This tension is magnified when the number of levels h increases.

Recently, multicore algorithms for sorting were given in [5], [14], [15], and the algorithms in [14], [15] claim fairly good performance on a multi-level cache hierarchy. However, this claimed good performance is still a factor of p'_i worse than the best possible for each cache level-i, and is obtained by analyzing a simple assignment to each core of a proportionate slice of the level-i cache above it, for each i. In contrast, the MO algorithms we present fully exploit the higher level caches, rather than just a small fraction of the best possible.

III. MULTICORE-OBLIVIOUS ALGORITHMS ON THE HM MODEL

For the effective use of multicores we need to have efficient *multicore-oblivious (MO)* algorithms, i.e., algorithms that do not use specific values for multicore parameters such as number of cores, number of levels of caches or their sizes, block sizes, etc., yet perform efficiently across a wide variety of multicores. To address this challenge, we propose some simple enhancements to HM algorithms in the form of instructions or 'hints' in the algorithm that are meant to be interpreted and used by the run-time scheduler to decide how to schedule parallel tasks generated during execution. Our initiative to introduce these special instructions for the run-time scheduler is in the spirit of the recent trend towards 'multiresolution' languages [16], 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

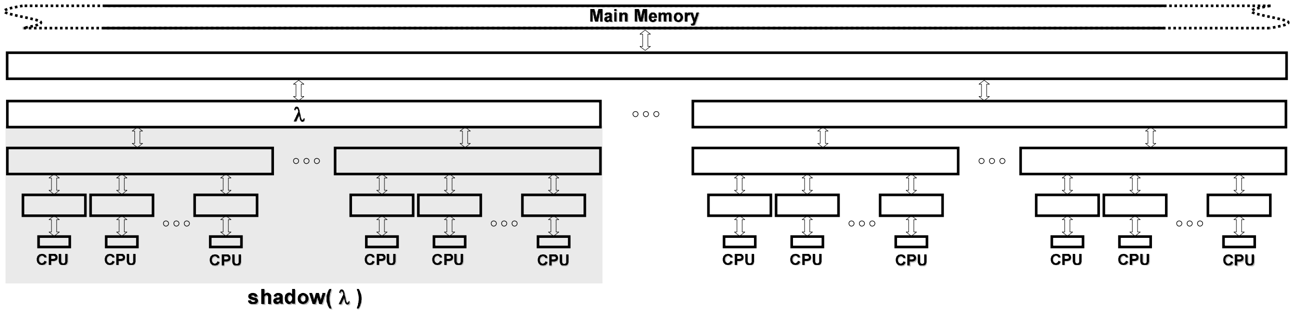

Figure 1. The HM model illustrated for $h = 5$. The dark area below the L_3 cache λ covers the caches and cores under the *shadow of* λ (see Section III).

in order to extract efficient performance of algorithms on multicores with multi-level caches.

The multicore-oblivious algorithms we present use two main types of scheduler hints, namely, CGC (coarse-grained contiguous) and SB (space-bound), and a third one that combines these two (CGC⇒SB). Of these CGC is useful for scheduling computations involving parallel **for** loops, and SB and CGC⇒SB are useful for algorithms that recursively spawn parallel tasks. In the following, the $shadow$ of a level-i cache λ includes (1) those p'_i cores that share λ as their level-i cache, and (2) all level-j $(< i)$ caches that lie between these cores and λ. Figure 1 shows an example. We also introduce the notion of an *anchor* cache. When we say that a task τ is *anchored* at (or is *assigned* to) a cache λ it implies that λ 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. Though more than one cache can satisfy the space requirement of τ, any given task can be anchored at only one cache at any given time, and this is chosen by the scheduler based on additional criteria. For example, under the SB scheduler λ must be at the smallest possible cache level under the shadow of the task that spawned τ.

A technical point: Whenever feasible, the scheduler respects block boundaries during the decomposition and distribution of tasks to avoid ping-ponging. Ping-ponging arises when writes to a block by a core are interleaved by reads/writes by other cores. This results in an excessive number of block transfers. We also assume that the hardware support that causes ping-ponging is at the size of B_1.

A. Coarse-Grained Contiguous (CGC) Scheduling

The *coarse-grained contiguous (CGC) scheduling* distributes an ordered collection of parallel subtasks in contiguous chunks across a sequence of contiguous cores. Each subtask is anchored at the L_1 cache subtending the core to which it is assigned. CGC is used to decompose a parallel **for** loop acting on a contiguous chunk of data into segments, each of which is executed by a core in parallel to others. Consider a parallel loop with index variable k running from 1 to t, and assume we wish to use all p cores to execute it under CGC scheduling. Then CGC will decompose the values taken by k into p contiguous segments of the same or almost the same length (within ± 1 of one another), and the j-th such segment will be assigned to the j-th core from the left. It is ensured that each segment will scan input data of length at least B_1 even if that results in some idle cores. CGC can also be applied to loops appearing in a task that is already anchored (by some other scheduler, e.g., SB in Section III-B) at some cache λ at a level higher than L_1. In that case, the subtasks are assigned only to cores under $shadow(\lambda)$.

MO-MT(A, n)
Input: an $n \times n$ matrix A in row-major;
Output: the transpose A^T, again in row-major.
[CGC] pfor $1 \leq k \leq n^2$ **do**
  $i := \lceil k/n \rceil;\ j := 1 + (k-1) \mod n;$
  $I[i, j] := A[\beta^{-1}(i, j)]$
[CGC] pfor $1 \leq k \leq n^2$ **do**
  $i := \lceil k/n \rceil;\ j := 1 + (k-1) \mod n;$
  $A^T[i, j] := I[\beta(j, i)]$

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

Matrix Transposition: Figure 2 gives a matrix transposition algorithm (based on an algorithm in [4]) scheduled with CGC. Here, **pfor** denotes a parallel **for** loop, and $\beta(i, j)$ is the ordered pair of integers (i', j') obtained by *bitwise interleaving* of the binary representations of indices i and j (see [13]). We assume that $\beta(i, j)$ and $\beta^{-1}(i, j)$ can be computed in constant time by the hardware.

The proof of the following theorem is based on the observation that MO-MT disperses a sequence of t entries in row-major order within a constant number of sequences of at most t^2 entries in the bit-interleaved order. Note that the parallelization of the recursive optimal cache-oblivious algorithm for matrix transposition in [1] would take $\Theta(\log n)$ parallel steps and hence would not achieve optimal *constant* critical pathlength of our algorithm.

Theorem 1. MO-MT *correctly transposes the* $n \times n$ *matrix* A *in* $\mathcal{O}(n^2/p + B_1)$ *parallel steps with* $\mathcal{O}(n^2/(q_i B_i) + B_i)$ *cache misses at each of the* q_i *caches in the level-*i *of the cache hierarchy, for all* $i \in [1, h-1]$*, assuming that all caches are tall, and* $n^2 \geq C_{h-1}$*. This result is optimal.*

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. Similarly, in the block S_A that contains $A[\beta^{-1}(i_1, j_1)]$, 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 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 in step 1 has the same cache complexity as a scan, except that up to B_l^2 elements remain unused at the end in a level-l cache, for each l. This gives the desired bound for step 1. The analysis for step 2 is similar. Optimality of this result follows from the lower bound on sequential cache complexity of matrix transposition.

Finally, if $n^2 < p \cdot B_1$, the scheduler will assign a block of data to $\lceil n^2/B_1 \rceil$ cores in order to respect the L_1 block boundaries. Hence in this situation, the parallel time is $\mathcal{O}(B_1)$. Thus the overall parallel time is bounded by $\mathcal{O}(n^2/p + B_1)$. Since the total work is $\Omega(n^2)$, the parallel speed-up is optimal for $n^2 \geq p \cdot B_1$, and thus for $n^2 \geq C_{h-1} \geq p \cdot {B_1}^2$. ∎

Scans, including prefix sums, on an input of size n can be scheduled with CGC in $\mathcal{O}(B_1 \log n)$ parallel steps [13].

B. 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 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.e., that lie in λ'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 in $(C_{i-1}, C_i]$ which are to be executed under $shadow(\lambda)$. When the current task assigned to λ completes, the first task τ in $Q(\lambda)$ is dequeued and executed while anchored at λ. When τ forks a task τ' and $s(\tau') \leq C_{i-1}$, τ' is assigned to the least loaded cache under $shadow(\lambda)$ at the smallest level $j \leq i-1$ such that $s(\tau') \leq C_j$, otherwise is enqueued in $Q(\lambda)$.

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. Section V presents an important application I-GEP (that include matrix multiplication and other problems), that is scheduled using SB.

C. 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 recursive forking from 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 (by either SB or CGC⇒SB) 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. It then distributes the subtasks evenly across the caches in level t, for $t = \max(i,j)$, under the shadow of λ. Although this distribution step has some similarity to CGC, unlike CGC subtasks can be anchored at a level-t cache for $t > 1$ in CGC⇒SB.

Multiple tasks can be anchored at λ for simultaneous execution provided the total space needed by all such tasks is bounded from above 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 $\omega_n = e^{2\pi\sqrt{-1}/n}$. In Figure 3, we present MO-FFT, obtained by adapting the cache-oblivious FFT algorithm in [1] to the HM model. MO-FFT can also be viewed as an adaptation of the network-oblivious FFT algorithm given in [4]. We use two types of scheduling in MO-FFT: CGC and CGC⇒SB.

The following theorem gives the performance bounds of MO-FFT in the HM model.

Theorem 2. *When run on an input of size* $n \geq C_{h-1}$*,* MO-FFT *terminates in* $\mathcal{O}\left((n/p + B_1)\log n\right)$ *parallel steps, and incurs* $\mathcal{O}\left((n/(q_i B_i))\log_{C_i} n\right)$ *cache misses at each of the*

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

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

Figure 3. MO-FFT: multicore-oblivious in-place FFT.

q_i caches in level-i ($i \in [1, h-1]$) of the cache hierarchy, provided all caches are tall. These bounds are optimal.

Proof: Since the CGC computations in lines 3, 4, 6, 7, 9 and 10 all have $\mathcal{O}(B_1)$ critical pathlength, the critical pathlength of MO-FFT is $\mathcal{T}_\infty(n) = \mathcal{T}_\infty(n_1) + \mathcal{T}_\infty(n_2) + \mathcal{O}(B_1) = \mathcal{O}(B_1 \log n)$. Hence, the number of parallel steps using p cores $\mathcal{T}_p(n) = \mathcal{T}_1(n)/p + \mathcal{T}_\infty(n) = \mathcal{O}((n/p + B_1) \log n)$ provided a core is not left idling when parallel tasks are available. Since CGC⇒SB executes all tasks 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 $C_i \ge p_i' B_1$, for all i, that is, when each core gets at least one block in the CGC computations at each recursive step of the tasks anchored at caches. Since FFT must perform $\Omega(n \log n)$ work, the speed-up is optimal for $n \ge p \cdot B_1$.

In order to compute the cache complexity of MO-FFT, consider any level-i cache λ. Observe that cache-misses according to CGC are incurred in steps 3, 4, 6, 5, 9 and 10, while steps 5 and 8 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. At each of these levels $\mathcal{O}(n/(q_i B_i))$ cache-misses are incurred by the algorithm at λ, and no additional cache-misses are incurred once the data fits into the cache. Thus the total number of cache-misses at λ is $\mathcal{O}\left((n/(q_i B_i)) \log_{C_i} n\right)$. The optimality of this bound follows from a straight-forward extension of the cache-miss lower bound proved in [15] for the computation DAG of MO-FFT on a two-level multicore model to the multi-level model. ∎

Sorting: Sample Partition Merge Sort (SPMS) is a very recent multicore-oblivious algorithm for sorting on the simple multicore model with just private caches [15]. SPMS runs in $\mathcal{O}(n \log n)$ sequential time and $\mathcal{O}((n/B) \log_C n)$ sequential cache misses (assuming a tall cache of size C and block size B) on an input of length n. The algorithm has critical pathlength $\mathcal{O}(\log n \log \log n)$, a linear space bound

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

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

Figure 4. MO-SpM-DV: multicore-oblivious sparse matrix and dense vector multiplication.

for all tasks, and can be scheduled optimally on a multicore with private caches.

While more complex than MO-FFT, SPMS has exactly the same structure, except that the CGC steps, which in MO-FFT use MO-MT and other constant ($\mathcal{O}(B_1)$) parallel time computations, instead consist of a constant number of applications of prefix sums and other *balanced parallel computations* ('BP' computations) [15] that can be scheduled under CGC. These BP computations are used to decompose an original problem of size n into $\sqrt{n}$ independent subproblems. The overall problem of size n is solved by a sequence of two recursive calls to subproblems of size $\sqrt{n}$ (similar to the two CGC⇒SB steps in MO-FFT). Because of this similarity in the structure of the computation, the results for MO-FFT under our scheduler translate to SPMS. However, the parallel time increases from $\mathcal{O}(B_1 \log n)$ to $\mathcal{O}(B_1 \log n \log \log n)$ due to the use of prefix sums CGC computations (which has $\mathcal{O}(B_1 \log n)$ critical pathlength) instead of the constant depth MO-MT used in MO-FFT. This gives us the following result.

Theorem 3. *When run on an input of size $n \ge C_{h-1}$, SPMS terminates in $\mathcal{O}((n/(p \log \log n) + B_1) \log n \log \log n)$ parallel steps and incurs $\mathcal{O}\left((n/(q_i B_i)) \log_{C_i} n\right)$ cache misses at each of the q_i caches in level-i ($i \in [1, h-1]$) of the cache hierarchy, provided all caches are tall. This result is optimal for $p \le n/(B_1 \log \log n)$.*

The optimality of the cache-miss bound stated in Theorem 3 can be established by extending the cache-miss lower bound for any comparison-based sorting algorithm on a two-level multicore model proved in [15] to the multi-level model.

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

Sparse Matrix Dense Vector Multiplication (SpM-DV): In the full paper we show that CGC⇒SB efficiently schedules the separator-based sparse matrix dense vector multiplication algorithm given in [10] (see MO-SpM-DV in Figure 4), provided the matrix has a support graph with good separators. A class of graphs closed under the subgraph relation is said to satisfy a $f(n)$-*edge separator theorem* if there exist constants $\alpha \in [\frac{1}{2}, 1)$ and $\beta > 0$ such that every n-node graph G from the class can be partitioned into two vertex-disjoint subgraphs containing at most αn vertices each and with no more than $\beta f(n)$ edges of G crossing the partition [17]. The *support graph* of an $n \times n$ matrix A is defined to be the graph with vertex set $\{1, \ldots, n\}$ and edge set $\{(i,j)|A[i,j] \neq 0\}$. We say that A satisfies an $f(n)$-edge separator theorem if its support graph satisfies such a theorem. A separator tree of A is constructed by applying the separator theorem to the whole support graph to get two components, and then recursively applying the theorem to each component until only a single node remains at each leaf of the tree.

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

Theorem 4. *If $n \geq C_{h-1}$, 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 $\mathcal{O}(n/p + B_1 + \log(n/B_1))$ parallel steps, and incurs $\mathcal{O}\left((n/q_i)(1/B_i + 1/C_i^{1-\epsilon})\right)$ cache misses at each of the q_i caches in level $i \in [1, h-1]$ of the hierarchy.*

Proof: It is not difficult to see that under the CGC⇒SB scheduler each task anchored at C_1 will have space bound $\Omega(B_1)$ (since $n > C_{h-1} \geq p \cdot C_1 \geq p \cdot B_1$), and that at each level i cache λ, $\Theta(n/(q_i C_i))$ tasks will be anchored whose parents have space bound too large for λ. Once such a task τ is anchored at λ all its descendant subtasks will be executed completely under the shadow of λ. Hence, the total number of cache misses incurred at λ will be the sum of the cache misses incurred by these 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 incur a cache miss. However, according to the edge separator 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, $\mathcal{Q}_i(n) = \mathcal{O}(n/(q_i C_i) \cdot (C_i/B_i + {C_i}^{\epsilon}))$ $= \mathcal{O}\left((n/q_i) \cdot \left(1/B_i + 1/C_i^{1-\epsilon}\right)\right)$.

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

IV. Review of Network-Obliviousness

A network-oblivious (NO) algorithm $\mathcal{A}$ [4] is an algorithm designed for the $\mathsf{M}(N)$ model, where N is a suitable function of the input size and represents the maximum number of processors for which the computation is designed. An $\mathsf{M}(N)$ is a complete network of N *processing elements* (*PEs*) each consisting of a CPU and an unbounded local memory. $\mathcal{A}$ consists of a sequence of synchronous supersteps: in a superstep a PE performs operations on local data and sends messages to other PEs. The complexity of $\mathcal{A}$ is then evaluated on $\mathsf{M}(p, B)$, for $p \leq N$ and $B \geq 1$. The $\mathsf{M}(p, B)$ is defined by two parameters and, intuitively, 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. An NO algorithm can be naturally executed on $\mathsf{M}(p, B)$ for every $p \leq N$ and B by stipulating that each processor carries out the operations of N/p consecutive PEs. The *communication* (resp., *computation*) *complexity* of $\mathcal{A}$ is the sum over all supersteps of the maximum number of blocks sent/received (resp., operations performed) by a processor in each superstep.

Under some reasonable assumptions [4], NO algorithms with optimal communication complexity on any $\mathsf{M}(p, B)$ exhibit optimal communication time on a variant of the *Decomposable-BSP* model [18], denoted as $\mathsf{D\text{-}BSP}(P, \boldsymbol{g}, \boldsymbol{B})$ where $\boldsymbol{g} = (g_0, \ldots, g_{\log P - 1})$ and $\boldsymbol{B} = (B_0, \ldots, B_{\log P - 1})$. A D-BSP is essentially an $\mathsf{M}(P, \cdot)$ machine where processors are recursively partitioned into 2^i clusters of size $P/2^i$ for each $0 \leq i < \log P$. The cost of a superstep s, where each processor communicates with processors within its cluster of size 2^i, is defined to be $h_s g_i$, where h_s is the maximum number of blocks of size B_i sent/received by a processor during s on $\mathsf{M}(P, B_i)$. The *communication time* of an algorithm is the sum of superstep costs.

Other than sorting and SpM-DV, the algorithms in Section III are all adapted from NO algorithms presented in [4]. For sorting, a slower NO algorithm is presented in [4] based on column sort (see also [13] for some improvements). In the next two sections we present MO algorithms for several other problems, and then adapt these algorithms to the NO framework.

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

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

Figure 5. Gaussian Elimination Paradigm (GEP).

V. Gaussian Elimination Paradigm

Let x be an $n \times n$ matrix with entries from an arbitrary domain $\mathcal{S}$, and let $f : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ be an arbitrary function. By *Gaussian Elimination Paradigm* [19] we refer to the computation in Figure 5, where the algorithm modifies x by applying a given set of *updates*, denoted by $\langle i, j, k \rangle$ for $i, j, k \in [0, n)$. We let Σ_f denote the set of updates the algorithm needs to perform. Many problems can be solved by GEP, including Floyd-Warshall's all-pairs shortest path, Gaussian Elimination and LU decomposition without pivoting, and matrix multiplication.

A cache-oblivious recursive implementation of GEP, called *I-GEP*, was presented in [19] and parallelized in [7] for multicore models with one level of cache. I-GEP consists of four recursive functions $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$ which are reproduced in the appendix for convenience. The functions accept as input four matrices $X \equiv x[I, J]$, $U \equiv x[I, K]$, $V \equiv x[K, J]$ and $W \equiv x[K, K]$, where I, J, K denote suitable intervals in $[0, n)$, and they differ on the amount of overlap X, U, V and W have among them: $\mathcal{A}$ assumes completely overlapping matrices, $\mathcal{B}$ (resp., $\mathcal{C}$) expects that $X \equiv V$ and $U \equiv W$ (resp., $X \equiv U$ and $V \equiv W$), and $\mathcal{D}$ assumes completely non-overlapping matrices (other types of overlapping are not possible). Each function performs updates in $\Sigma_f \cap (I \times J \times K)$ (i.e., each update $\langle i, j, k \rangle$ such that $x[i, j]$, $x[i, k]$, $x[k, j]$ and $x[k, k]$ are contained in X, U, V, and W, respectively) by means of eight recursive calls to $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and $\mathcal{D}$, and using suitable quadrants of X, U, V, and W as inputs. The initial call is $\mathcal{A}(x, x, x, x)$. The four functions also differ in the amount of parallelism they offer: intuitively, the less the overlap among the input matrices the more flexibility the function has in ordering its recursive calls, and thus leading to better parallelism.

I-GEP produces the correct output under certain conditions which are met by all notable instances mentioned above, and incurs $\mathcal{O}(n^3/B\sqrt{C})$ cache misses and terminates in $\mathcal{O}(n^3/p + n\log^2 n)$ parallel time [7] when executed on p cores with one cache level of size C and block length B, for both shared and distributed caches. Also presented in [7] is *C-GEP* which extends I-GEP and implements correctly any instance of GEP with no asymptotic degradation in performance. Finally, tiled I-GEP [11] runs in $\mathcal{O}(n^3/p + n)$ parallel time without increasing the cache complexity on HM [13] but is not multicore-oblivious.

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

A. Multicore-Oblivious Algorithm

Theorem 5 below states the performance of I-GEP under SB scheduler. The cache-miss bound is based on the observation that a total of $\Theta\left(n^3/(C_i\sqrt{C_i})\right)$ tasks are anchored at level-i caches incurring $\mathcal{O}(C_i/B_i + \sqrt{C_i})$ level-i cache-misses each. The parallel time follows from the observation that if C_i is larger than $p_i C_{i-1}$ by the factor stated in the theorem, the critical path of the I-GEP computation DAG has no asymptotic effect on its parallel time.

Theorem 5. *When executed under the SB scheduler, I-GEP on an $n \times n$ input, $n^2 \geq C_{h-1}$, incurs $\mathcal{O}(n^3/(q_i B_i \sqrt{C_i}))$ cache misses at each level-i cache, and terminates in $\mathcal{O}(n^3/p)$ parallel time, provided all caches are tall, and $C_i > c_i \cdot p_i \cdot C_{i-1}$ with $c_i = 2\log^2(C_i/C_{i-1})$ holds for $i \in [2, h-1]$. These bounds are optimal.*

Proof: Since I-GEP accesses data only inside recursive function calls with input size 1×1, it suffices to compute the misses incurred only by tasks with space bound $\leq C_{h-1}$. Let τ be a task anchored at a level-i cache λ. Observe that each subtask generated by τ has space bound $s(\tau)/4$, 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 descendant of τ anchored at an L_{i-1} and an L_1 cache under $shadow(\lambda)$ will have space bound larger than $C_{i-1}/4$ and $\Omega\left(B_1^2\right)$, respectively. There are $\Theta\left(n/\sqrt{C_i}\right)$, $\Theta\left(n^2/C_i - n/\sqrt{C_i}\right)$ and $\Theta\left(n^3/(C_i\sqrt{C_i}) - 2 \cdot n^2/C_i + n/\sqrt{C_i}\right)$ tasks with space bound $\Theta(C_i)$ 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 $\mathcal{O}(\sqrt{C_i} + C_i/B_i)$ misses in λ, the claimed cache-miss bound follows. Since I-GEP includes matrix multiplication, a cache-miss lower bound for I-GEP matching its upper bound can be proved by extending the lower bound result proved in [15] for matrix multiplication on a two-level multicore model to the multi-level model.

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

B. Network-Oblivious Algorithm

N-GEP is an optimal NO algorithm which performs correctly any commutative GEP computation which is correctly solved by I-GEP [7]. It exhibits space optimality, which is not yielded by a straightforward NO implementation of I-GEP, and is optimal on $\mathsf{M}(p,B)$ and on the D-BSP model for a wide range of their machine parameters. A GEP computation is *commutative* if its function f satisfies $f(f(y,u_1,v_1,w_1),\ u_2,v_2,w_2) = f(f(y,u_2,v_2,w_2),u_1,v_1,w_1)$ for each y, u_1, v_1, w_1, u_2, v_2, w_2 in $\mathcal{S}$. Not all GEP computations are commutative, however all of the instances of GEP for the aforementioned problems are commutative.

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

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

Then, N-GEP performs correctly and with a constant memory blow-up any commutative GEP computation which is correctly solved by I-GEP. The following theorem shows that N-GEP performs optimally on the $\mathsf{M}(p,B)$ and D-BSP models.

Theorem 6. *When executed on $\mathsf{M}(p,B)$ for $p \le n^2/\log^2 n$ and input size n, N-GEP exhibits optimal $\Theta(n^3/p)$ computation complexity, and $\mathcal{O}(n^2/(\sqrt{p}B) + n\log^2 n)$ communication complexity which is optimal when $p \le n^2/\log^4 n$ and $B \le n/(\sqrt{p}\log^2 n)$. Furthermore, N-GEP is optimal on a $\mathsf{D\text{-}BSP}(P,\boldsymbol{g},\boldsymbol{B})$ for $P \le n/\log n$ and $B_i = \mathcal{O}(n2^{i/2}/(P\log n))$, for each $0 \le i < \log P$.*

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

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

which solves to $\mathcal{O}(m^2/(qB) + \min\{\sqrt{q}, m\})$. Similarly, it can be proved that the communication complexity of a call to $\mathcal{B}$ (and $\mathcal{C}$ as well) or $\mathcal{A}$ is $\mathcal{O}(m^2/(qB) + m\log m)$ and $\mathcal{O}(m^2/(qB) + m\log^2 m)$, respectively. Since N-GEP consists of a call to $\mathcal{A}$ with $m = n$ and $q = p$, the upper bound on the communication complexity of N-GEP follows. Upper bounds on computation complexity and communication time can be proved in the same fashion. We remind that N-GEP correctly computes matrix multiplication without memory blow-up: then, since the communication and computation complexities of matrix multiplication without memory blow-up are $\Omega(n^2/(\sqrt{p}B))$ [20] and $\Omega(n^3/p)$, respectively, N-GEP exhibits optimal communication and computation complexities in the stated intervals of p and B. By using a result on the network-oblivious algorithm for matrix multiplication without memory blow-up in [4], it can be proved that N-GEP's communication time is optimal in the stated parameter intervals. We refer to the full paper [13]

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

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

Table I
RECURSIVE CALLS PERFORMED BY $\mathcal{D}$ AND $\mathcal{D}^*$; CALLS IN A ROUND ARE PERFORMED IN PARALLEL.

for more details. ∎

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

VI. LIST RANKING AND OTHER GRAPH ALGORITHMS

We now present MO and NO algorithms for list ranking and other graph problems. We represent a linked list of n nodes as an array where each position contains a node identifier and pointers to its successor and predecessor. The rank of a node is its distance from the end of the list, and the list ranking problem consists in determining the ranks of every node.

A. Multicore-Oblivious Algorithms

List Ranking: Our MO algorithm, named *MO-LR*, employs the list contraction technique described in [21]: this technique solves the list ranking problem by identifying an independent set S of the list[2] of size $\Theta(n)$, contracting the list by removing nodes in S, recursively solving the list ranking problem on the contracted list, and then extending the solution to the removed nodes. It is not difficult to see that list contraction and the extension of the solution to nodes in the independent set can be accomplished with $\mathcal{O}(1)$ sorts and scans using the CGC⇒SB and CGC schedules, respectively. This algorithm is derived from those in [22], [23] by adopting our MO primitives for sorting and scanning and by recursively contracting the linked list down to a constant size, instead of a variable size based on architectural parameters. We use as sorting primitive the MO algorithm SPMS given in Section III-C.

The MO algorithm, named *MO-IS*, for computing an independent set S is given in Figure 6. We observe that each step of MO-IS is solved by $\mathcal{O}(1)$ sorts and scans using the CGC⇒SB and CGC schedules, respectively. In step 1 we identify a $\log\log n$ coloring of the nodes by applying twice the deterministic coin flipping algorithm in [21] which, given a k-coloring, constructs a $(1+\log k)$-coloring: since the new color of a node is determined by only the initial colors of the node and of its successor, $\mathcal{O}(1)$ sorts and scans suffices to accomplish the coloring. During the j-th iteration of step 7 we create duplicates of successor and predecessor of each color j node v in the independent set S to indicate that v is in S. Hence, when a node is found replicated in step 6 of later iterations, it is removed and not inserted in S because at least one of its neighbors is in S. Clearly, steps 5-7 require $\mathcal{O}(1)$ sorts and scans.

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

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

Figure 6. MO-IS: multicore-oblivious algorithm for computing an independent set.

Theorem 7. *MO-LR with input size n terminates in $\mathcal{O}((n/p)\log n + (B_1\log(pB_1) + \log n\log\log n)\log(n/B_1)\log\log n)$ parallel steps, and incurs $\mathcal{O}((n/(q_iB_i))\log_{C_i} n + ((C_i/B_i)\log_{p'_i B_1}(q_iC_i)\log\log n + \log n(\log\log n)^2)\log(n/B_1))$ misses at each of the q_i caches in the level-i ($i \in [1, h-1]$) of the cache hierarchy.*

Proof: Consider the execution of MO-IS and let n_j denote the number of list nodes of color j after the coloring. At the beginning of the j-th iteration of the **for** loop, with $1 \le j \le \log\log n$, there are at most $3n_j$ nodes of color j since at most $2n_j$ nodes of color j were added in earlier iterations. A loop iteration consists of $\mathcal{O}(1)$ sorts and scans of $\Theta(n_j)$ nodes, and the j-th iteration requires $\mathcal{O}((n_j/p)\log n_j + \log n_j\log\log n_j)$ parallel steps if $n_j > pB_1$, and $\mathcal{O}(B_1\log(pB_1) + \log n_j\log\log n_j)$ otherwise because the scheduler reduces the number of active cores so that at most one core has less than B_1 nodes. It follows that MO-IS requires $\mathcal{O}((n/p)\log n + B_1\log(pB_1)\log\log n + \log n(\log\log n)^2)$ parallel steps since $\sum_{j=0}^{\log\log n-1} n_j = n$. Since the total work is $\Omega(n\log n)$, the parallel speed-up is optimal when $p \le n/((B_1 + \log\log n)\log n\log\log n)$. Similarly, we can show that the j-th iteration incurs $\mathcal{O}(n_j/(q_iB_i)\log_{C_i} n_j + (C_i/B_i)\log_{p'_i B_1}(q_iC_i) + \log n_j\log\log n_j)$ misses, where the last term is due to the critical pathlength of sorting. Then, MO-IS incurs $\mathcal{O}((n/(q_iB_i))\log_{C_i} n +$

$(C_i/B_i)\log_{p'_iB_1}(q_iC_i)\log\log n+\log n(\log\log n)^2)$ misses. The upper bounds in the theorem follow by observing that the returned independent set has size $n/3$ [22], and then solving simple recurrence relations. ∎

Note that one $\log\log n$ factor[3] in the parallel running time of MO-LR can be reduced to[4] $\log^{(k)} n$ for any integer constant $k>2$ by repeating the coloring algorithm k times (instead of twice) in step 1 of MO-IS.

Other Graph Problems: By using the CGC hint, it is straightforward to obtain as in [22]–[24] MO algorithms for Euler tour, and several tree problems such as rooting a tree, traversal numbering, vertex depth, subtree size and connected components of a forest. These algorithms have the same complexity as MO-LR.

Our MO algorithm for computing the connected components of a graph is based on the PRAM CREW algorithm in [25] adapted to adjacency lists, and uses MO-LR and tree computations to obtain the following result. Again, these algorithms are derived from the algorithms in [22], [23] by adopting our MO primitives for sorting and scanning and by recursively contracting the graphs down to a constant size, instead of variable sizes based on architectural parameters. More details are given in [13].

Theorem 8. *There exists an MO algorithm for computing the connected components of a graph of n nodes and m edges, which terminates in $\mathcal{O}(N\log N\log(N/B_1)/p + (B_1\log(pB_1)\ \log\log N + \log N(\log\log N)^2)(\log^2(N/B_1)))$ parallel steps, and incurs $\mathcal{O}(N/(q_iB_i)\log_{C_i} N\log(N/B_1)+((C_i/B_i)\log_{p'_iB_1}(q_iC_i)\log\log n + \log N(\log\log n)^2)\log^2(N/B_1))$ cache misses at each of the q_i caches in the level-i ($i\in[1,h-1]$) of the cache hierarchy, for $N=n+m$.*

B. Network-Oblivious Algorithms

As in previous MO algorithms, sorting is a fundamental primitive in the NO algorithms in this section. The NO sorting algorithm in [4], which is specified on $\mathsf{M}(n)$ (n is the input size), has optimal communication complexity on $M(p,B)$ when $p\le n^{1-\epsilon}$ for any constant $\epsilon\in(0,1)$ and $B\le\sqrt{n/p}$, but its computation complexity is suboptimal by a factor $\mathcal{O}\left((\log n)^{\log_{3/2}6}\right)$. In [13], we show that if the NO algorithm is defined on $\mathsf{M}(n^{1-\epsilon})$, for any $\epsilon\in(0,1)$, it features both optimal communication and optimal computation complexities on $M(p,B)$ when $p\le n^{1-\epsilon}$ and $B\le\sqrt{n/p}$. We use this variant in the following algorithms.

List Ranking: Let ϵ be an arbitrary constant in (0,1). The NO algorithm for list ranking, named *NO-LR*, derives from an adaptation of MO-LR to the $\mathsf{M}(n^{1-\epsilon})$ model. The NO algorithm is similar to the MO list ranking algorithm, and the main difference is in NO-IS, the NO implementation of MO-IS. Instead of storing nodes of the same color in consecutive memory locations (see step 3 of MO-IS in Figure 6), in NO-IS nodes of the same color are evenly distributed among the PEs to improve parallelism in the $\mathcal{O}(1)$ sorts within each **for** iteration. Since $\Theta(n_j)$ nodes are sorted in the j-th iteration, the NO sorting algorithm exhibits better performance when these nodes are evenly distributed among all the PEs because nodes stored in the same PE can only be accessed sequentially. The NO algorithm is similar to the MO algorithm in the other details, and we obtain the following performance bounds, using the aforementioned NO sorting algorithm.

Theorem 9. *When executed on $\mathsf{M}(p,B)$ for $p\le(n/\log\log n)^{1-\epsilon}$ and input size n, NO-LR exhibits optimal $\Theta((n/p)\log n)$ computation complexity and $\mathcal{O}(n/(pB)+((n/p)\log\log n)^{1/2}\ +\ p^\epsilon\log n\log\log n)$ communication complexity, which is optimal for $B=\mathcal{O}(\tilde{B})$, where $\tilde{B}=\min\{(n\log\log n/p)^{1/2},\ n/(p^{1+\epsilon}\log n)\}/\log\log n$. Furthermore, NO-LR is optimal on a D-BSP$(P,\boldsymbol{g},\boldsymbol{B})$ for $P\le(n/\log\log n)^{1-\epsilon}$ and $B_0=\mathcal{O}(\tilde{B})$.*

Proof: (Sketch) As noted in the proof of Theorem 7, at the beginning of the i-th iteration of the **for** loop, with $1\le j\le\log\log n$, there are at most $3n_j$ nodes of color j, and a loop iteration consists of $\mathcal{O}(1)$ sorts and scans of $\Theta(n_j)$ nodes. Since these nodes are evenly distributed among the $n^{1-\epsilon}$ PEs, sorting is executed by $\Theta(\min\{n_j,p\})$ processors of $\mathsf{M}(p,B)$ (in contrast, if nodes were stored on consecutive PEs, sorting would be executed by $\Theta(\lceil n_j/(n^{1-\epsilon}/p)\rceil)$ processors of $\mathsf{M}(p,B)$). Then, it can be proved that the communication and computation complexities of each iteration are $\mathcal{O}(n_i/(pB)+n_i^\epsilon+\sqrt{n_i/p}+n_i^{\epsilon/2}+p^\epsilon+\log p\log\log n)$ and $\mathcal{O}(n_i\log n/p+n_i^\epsilon\log n+n^{1-\epsilon}\log n\log\log n/p)$. Since $\sum_{j=0}^{\log\log n-1}n_j=n$, we have that the communication and computation complexities of NO-IS are $\mathcal{O}(n/(pB)+\sqrt{(n/p)\log\log n}+p^\epsilon\log\log n)$ and $\mathcal{O}(n\log n/p)$, respectively. The upper bounds in the theorem follow by observing that the returned independent set has size $n/3$, and then by solving simple recurrence relations. ∎

Other Graph Problems: As with MO algorithms, it is easy to derive NO algorithms with the same complexities as NO-LR for Euler tour and many tree problems. Similarly, for computing connected components in a graph we obtain the following result.

Theorem 10. *There exists an NO algorithm defined on $\mathsf{M}((n+m)^{1-\epsilon})$, where ϵ is an arbitrary constant in $(0,1)$, for computing the connected components of a graph of n nodes and m edges. On $\mathsf{M}(p,B)$ for $p\le(\tilde{N}/\Gamma)^{1-\epsilon}$, it exhibits $\mathcal{O}(\tilde{N}/(pB)+(\tilde{N}\Gamma/p)^{1/2}+p^\epsilon\Gamma\log n)$ communication and $\mathcal{O}((\tilde{N}/p)\log n)$ computation complexities, where $\tilde{N}=n+m\log n$ and $\Gamma=\log n\log\log n$.*

[3] The $\log\log n$ factor common to the 2nd and the 3rd term.

[4] For any positive integer k, $\log^{(k)} n=\log\log^{(k-1)} n$ if $k>1$; $\log^{(k)} n=\log n$ otherwise.

Problem	**Time**	**MO Cache**	**NO Communication**	**Max value of p**
Prefix sum	$\Theta(n/p)$	$\Theta(n/(q_iB_i))$ [13]	$\Theta(\log p)$ [13]	MO: $n/(B_1 \log n)$ NO: $n/\log n$
Matrix transposition	$\Theta(n^2/p)$	$\Theta(n^2/(q_iB_i))$	$\Theta(n^2/(Bp))$ [4]	MO: n^2/B_1 NO: n^2
Matrix multiplication	$\Theta(n^3/p)$	$\Theta(n^3/(q_iB_i\sqrt{C_i}))$	$\Theta(n^2/(B\sqrt{p}))$ [4]	MO: $\min\left\{n^2, C_{h-1}/\left(2^{h-2}C_1\right)\right\}$ NO: n^2
GEP	$\Theta(n^3/p)$	$\Theta(n^3/(q_iB_i\sqrt{C_i}))$	$\Theta(n^2/(B\sqrt{p}))$	MO: $\min\left\{n^2/\log^2 n, C_{h-1}/\left(C_1 \prod_{i=2}^{h-1}\left(2\log^2\left(C_i/C_{i-1}\right)\right)\right)\right\}$ NO: $n^2/\log^2 n$
FFT	$\Theta(n \log n/p)$	$\Theta(n/(q_iB_i)\log_{C_i} n)$	$\Theta(n/(pB)\log_{n/p} n)$ [4]	MO: n/B_1 NO: n
Sorting	$\Theta(n \log n/p)$	$\Theta(n/(q_iB_i)\log_{C_i} n)$	$\Theta(n/(pB))$ [13]	MO: $n/(B_1 \log\log n)$ NO: $n^{1-\epsilon}$, $\epsilon \in (0,1)$
List ranking	$\Theta(n \log n/p)$	$\mathcal{O}(n/(q_iB_i)\log_{C_i} n +$ $(\log\log n)^2 \log(n/B_i))$	$\mathcal{O}(n/(Bp)$ $+((n/p)\log\log n)^{1/2}$ $+p^{\epsilon}\log n \log\log n$	MO: $n/(B_1 + \log\log n)\log n \log\log n$ NO: $(n/\log\log n)^{1-\epsilon}$

Table II
SUMMARY OF OUR RESULTS.

VII. CONCLUSION

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

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

As mentioned earlier, the notion of MO algorithm is complementary to that of network-oblivious (NO) algorithm [4]. However there are some important differences: In going from MO to NO, we need to move from shared-memory to message passing, which also involves moving the concurrent read environment used in multicores to exclusive reads. In going from NO to MO, we need to move from message passing to shared-memory, and also need to develop methods to exploit locality at all levels of the cache hierarchy. Nevertheless, in this paper we have established several connections between efficient MO and NO algorithms: we derived NO algorithms for GEP and list ranking by suitably adapting our MO algorithms, and our MO algorithms for matrix transposition and FFT from NO algorithms in [4]. These results raise hopes for a unified notion of obliviousness in parallel computation, which may be of interest with the advent of networks of multicores, like the Blue Waters system [26].

We conclude with a summary of our main results in Table II, where some minor constraints relating to cache and network parameters are omitted. Recall that the cache complexity at the i-th level is defined as the maximum number of block transfers into and out of any single L_i cache by the p'_i cores which share L_i, while the communication complexity is the maximum number of blocks sent or received by a processor. We note that the communication complexity coincides in many cases with the average number of cache misses by a core in a level-i cache of size N/p, where N is the input size (e.g., it is n^2 for GEP).

ACKNOWLEDGMENT

The authors would like to thank Andrea Pietracaprina and Keshav Pingali for useful discussions, and the anonymous reviewers for their interesting comments.

REFERENCES

[1] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proc. 40th IEEE FOCS*, 1999, pp. 285–297.
[2] L. Arge, G. S. Brodal, and R. Fagerberg, "Cache-oblivious data structures," in *Handbook of Data Structures and Applications*. CRC Press, 2004, ch. 38.
[3] E. D. Demaine, "Cache-oblivious algorithms and data structures," 2002.
[4] G. Bilardi, A. Pietracaprina, G. Pucci, and F. Silvestri, "Network-oblivious algorithms," in *Proc. 21st IEEE IPDPS*, 2007.
[5] L. Arge, M. Goodrich, M. Nelson, and N. Sitchinava, "Fundamental parallel algorithms for private-cache chip multiprocessors," in *Proc. 20th ACM SPAA*, 2008, pp. 197–206.
[6] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul, "Concurrent cache-oblivious B-trees," in *Proc. 17th ACM SPAA*, 2005, pp. 228–237.
[7] R. A. Chowdhury and V. Ramachandran, "The cache-oblivious Gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation," in *Proc. 19th ACM SPAA*, 2007, pp. 71–80.

[8] M. Frigo and V. Strumpen, "The cache complexity of multithreaded cache oblivious algorithms," *Theory Comput. Syst.*, vol. 45, no. 2, pp. 203–233, 2009.

[9] G. Blelloch and P. Gibbons, "Effectively sharing a cache among threads," in *Proc. 16th ACM SPAA*, 2004, pp. 235–244.

[10] G. Blelloch, R. A. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch, "Provably good multicore cache performance for divide-and-conquer algorithms," in *Proc. 19th ACM-SIAM SODA*, 2008, pp. 501–510.

[11] R. A. Chowdhury and V. Ramachandran, "Cache-efficient dynamic programming algorithms for multicores," in *Proc. 20th ACM SPAA*, 2008, pp. 207–216.

[12] L. Valiant, "A bridging model for multi-core computing," in *Proc. 16th ESA*, 2008, pp. 13–28.

[13] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran, "Oblivious algorithms for multicores and network of processors," Univ. of Texas at Austin, Tech. Rep. 09-19, 2009, `http://www.cs.utexas.edu/research/publications`.

[14] G. Blelloch, P. Gibbons, and H. Simhadri, "Low depth cache-oblivious sorting," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-09-134, 2009.

[15] R. Cole and V. Ramachandran, "Resource oblivious multicore sorting," 2009, under submission.

[16] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *Int. J. of High Perf. Comp. Appl.*, vol. 21, no. 3, pp. 291–312, 2007.

[17] R. J. Lipton and R. E. Tarjan, "A separator theorem for planar graphs," *SIAM J. Applied Mathematics*, vol. 36, pp. 177–189, 1979.

[18] G. Bilardi, A. Pietracaprina, and G. Pucci, "Decomposable BSP: A bandwidth-latency model for parallel and hierarchical computation," in *Handbook of Parallel Computing: Models, Algorithms and Applications*. CRC Press, 2007, pp. 277–315.

[19] R. A. Chowdhury and V. Ramachandran, "Cache-oblivious dynamic programming," in *Proc. 17th ACM-SIAM SODA*, 2006, pp. 591–600.

[20] D. Irony, S. Toledo, and A. Tiskin, "Communication lower bounds for distributed-memory matrix multiplication," *Journal of Parallel and Distributed Computing*, vol. 64, no. 9, pp. 1017–1026, 2004.

[21] R. Cole and U. Vishkin, "Deterministic coin tossing with applications to optimal parallel list ranking," *Inf. Control*, vol. 70, no. 1, pp. 32–53, 1986.

[22] L. Arge, M. Goodrich, and N. Sitchinava, "Parallel external memory graph algorithms," in *Proc. 24th IEEE IPDPS*, 2010.

[23] Y. J. Chiang, M. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter, "External-memory graph algorithms," in *Proc. 6th ACM-SIAM SODA*, 1995, pp. 139–149.

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

[25] F. Y. Chin, J. Lam, and I. N. Chen, "Efficient parallel algorithms for some graph problems," *Commun. ACM*, vol. 25, no. 9, pp. 659–665, 1982.

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

Appendix

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

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

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

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

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

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

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

$\mathcal{D}(X, U, V, W)$
Input: $X \equiv x[I, J]$, $U \equiv x[I, K]$, $V \equiv x[K, J]$ and $W \equiv x[K, K]$, where I, J, K denote intervals (of the same length) in $[0, n)$, and $I \cap K = \emptyset$, and $J \cap K = \emptyset$.
Output: execution of all updates in $\Sigma_f \cap T$, with $T = I \times J \times K$.
Space Bound: $S(m) = 4m^2$, where $m = |I|$.

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