Improved Parallel Cache-Oblivious Algorithms for Dynamic Programming and Linear Algebra

Yan Gu
Carnegie Mellon University

Abstract

For many cache-oblivious algorithms for dynamic programming and linear algebra, we observe that the key factor that affects the cache complexity is the number of input entries involved in each basic computation cell. In this paper, we propose a level of abstraction to capture this property, and refer to it as the \( k \)-d grid computation structure. We then show the computational lower bounds for this grid structure, and propose efficient and highly-parallel algorithms to compute such grid structure that optimize the number of arithmetic operations, parallel depth, and the cache complexity in both the classic setting when reads and writes have the same cost, and the asymmetric variant that considers writes to be more expensive than reads.

Using the abstraction with the proposed algorithms as the implementation, we propose cache-oblivious algorithms for many fundamental problems with improved cache complexities in both the classic and asymmetric settings. The cache bounds are optimal in most applications we consider. Meanwhile, we also reduce the parallel depths of many problems. We believe that the novelty of our framework is of interests and leads to many new questions for future work.

1 Introduction

The ideal-cache model \[51\] is widely used in designing algorithms that optimize the communication between CPU and memory. The model is comprised of an unbounded memory and a cache of size \( M \). Data are transferred between the two levels using cache lines of size \( B \), and all computation occurs on data in the cache. An algorithm is cache-oblivious if it is unaware of both \( M \) and \( B \). The goal of designing such algorithms is to reduce the cache complexity\(1\) (or the \( I/O \) cost indistinguishably) of an algorithm, which is the number of cache lines transferred between the cache and the main memory assuming an optimal (offline) cache replacement policy. Sequential cache-oblivious algorithms are flexible and portable, and adapt to all levels of a multi-level memory hierarchy. Such algorithms are well studied \[8, 30, 42\]. Regarding parallelism, Blelloch et al. \[22\] suggest that analyzing the depth and sequential cache complexity of an algorithm is sufficient for deriving upper bounds on parallel cache complexity.

In this paper, we focus on a class of cache-oblivious algorithms that have a similar computation structure as matrix multiplication and can be coded up in nested for-loops. Their implementations are based on a divide-and-conquer approach that partitions the ranges of the loops and recurses on the subproblems until the base case is reached. Such algorithms are in the scope of dynamic programming (e.g., the LWS/GAP/RNA/Parenthesis problems) and linear algebra (e.g., matrix multiplication, Gaussian elimination, LU decomposition) \[22, 34, 37, 39, 51, 69, 88, 91, 92, 98\].

\[1\]In this paper, we refer to it as symmetric cache complexity to distinguish from the case when reads and writes have different costs.
However, there are several challenges in designing these cache-oblivious algorithms. The first issue is that the computation of these algorithms usually involves complicated data dependencies that are entangled with the divide-and-conquer approach, making the algorithm design and analysis less obvious. As a result, although these algorithms have been studied for 20 years, some of them are suboptimal regarding the sequential cache complexity and/or parallel depth, or are complicated and can be greatly simplified. For example, previous algorithms to solve the GAP problem (i.e., the GAP recurrence) and protein accordion folding shown in [34, 37, 39, 69, 88, 91] have cache complexity $\Theta(n^3/B\sqrt{M})$. However, because of the complication of data dependencies, as we show in this paper, this bound is not-optimal. Some other problems in higher (greater than three) dimensions (e.g., the RNA problem) are even harder, and directly applying the automatic systems in [34, 69] cannot yield algorithms with optimal cache complexity. In short, we lack a general method to analyze the lower and upper bounds of the cache complexity on these problems. Similar situations show up in terms of parallelism as well, and there exist spaces for improvements.

The second and more critical challenge emerges with the arrival of new non-volatile memory (NVM) technologies [62, 67]. NVMs will likely serve as the new main memories in the near future providing strong capacity-to-performance ratios. However, writes are more expensive than reads on NVMs. Roughly speaking, the reason is that writing to memory requires a change to the state of the material, while reading only requires detecting the current state. This trend poses the need to design write-efficient algorithms. However, these classic cache-oblivious algorithms based on divide-and-conquer use asymptotically the same numbers of reads and writes to the main memory, which is inefficient on asymmetric memories. Meanwhile, given the various combinations of computation structures and data dependencies, proving the lower bound and designing the individual algorithm for each specific problem can take significant effort.

To tackle the challenges, in this paper we propose a level of abstraction of the computation in these cache-oblivious algorithms. Previous algorithms are usually designed and analyzed based on the number of nested loops, or the number of the dimensions of which the input and output are stored and organized. However, we observe that the key underlying factor in determining the cache complexity of these computations is the number of input entries involved in each basic computation cell (e.g., two input values for the multiplication in matrix product). This observation and the associated solution is one of the reasons we show improvements of the bounds. We abstract this relationship as $k$-d grid computation structures (short for the $k$-d grids). A more formal definition is given Section 3. We then discuss the lower bounds to compute such $k$-d grids with and without considering the asymmetric cost between writes and reads. Based on the analysis of the lower bounds, we also propose highly-parallelized algorithms with the matching bound to compute a $k$-d grid assuming no data dependency within it.

We claim that the extra level of abstraction helps us to better understand the difficulties (lower bounds) of some problems. This is because the structures of the algorithms and recurrences in this paper (without considering data dependence) are always equivalent to one or more $k$-d grids. Hence the lower bounds on $k$-d grids apply to these algorithms and recurrences, indicating whether the previous algorithms are optimal, and if not, what improvement can be achieved. For write-efficient algorithms, we can also show the minimum weighted reads and writes required. These lower bounds are summarized in Table 1. We note that the lower bounds shown in this setting are non-trivial, and raised as an unsolved problem in the first papers in the research of asymmetric algorithms [18, 31].

For upper bounds, the data dependencies between the computations come in. However, we observe that the computations of all cache-oblivious algorithms in this paper can all be abstracted as or decomposed into multiple $k$-d grids without any local dependencies within each of them. Then by applying the efficient algorithms on $k$-d grids to these problems, we show better upper bounds as well as parallel depths. For problems we provide lower bounds in this paper, all but one algorithms can be shown optimal. We note that the decomposition is independent of whether the read and write costs are symmetric. Therefore, we will show
that once we propose an optimal algorithm on $k$-d grids in the asymmetry setting, we automatically improve the cache complexities of all the problems from the classic algorithms that do not consider such asymmetry.

We believe that the framework for analyzing cache-oblivious algorithms based on $k$-d grids provides a better understanding of these algorithms. In particular, the contributions include:

- We show algorithms with improved symmetric cache complexity on many problems, including the GAP recurrence, protein accordion folding, and RNA recurrence. We show that the previous cache bound $O(n^3/B\sqrt{M})$ for the GAP recurrence and protein accordion folding is not optimal, and we improve the bound to $O(n^2/B \cdot (n/M + \log \min\{n/\sqrt{M}, \sqrt{M}\}))$ and $\Theta(n^2/B \cdot (1 + n/M))$ respectively\footnote{The improvement is $O(\sqrt{M})$ from an asymptotic perspective (i.e., $n$ approaching infinity). For smaller range of $n$ that $O(\sqrt{M}) \leq n \leq O(M)$, the improvement is $O(n/\sqrt{M} / \log(n/\sqrt{M}))$ and $O(n/\sqrt{M})$ respectively for the two cases. (The computation fully fit into the cache when $n < O(\sqrt{M})$.)}. For RNA recurrence, we show an optimal cache complexity of $\Theta(n^4/BM)$, which improves the best existing result by $\Theta(M^{3/4})$.

- We show improved, linear parallel depth for cache-oblivious algorithms solving all-pair shortest-paths, Gaussian elimination, triangular system solver, LWS recurrences and protein accordion folding. For some of these problems, the depth bounds are achieved by previous algorithms \cite{44,89}, but they assumes a much stronger model and the algorithms are also much more complicated (more discussion in Section \ref{sec:parallel_depth}). Our approaches are under the standard nested-parallel model and are much simpler. Our algorithms are not in-place, but in Section \ref{sec:inplace} (before Theorem \ref{thm:inplace}) we show that the extra storage can bounded to arbitrarily small (any value no less than the cache size).

- We provide write-efficient cache-oblivious algorithms for all problems we discussed in this paper, including matrix multiplication, many linear algebra algorithms, all-pair shortest-paths, and a number of dynamic programming recurrences. If a write costs $\omega$ times more than a read (the formal computational model shown in Section \ref{sec:framework}), the asymmetric cache complexity is improved by a factor of $\Theta(\omega^{1/2})$ or $\Theta(\omega^{2/3})$ on each problem compared to the previous results \cite{20}. We also show that this improvement is optimal under certain assumptions (the CBCO paradigm, defined in Section \ref{sec:cbo}).

- The analysis framework is concise. In this single paper, we discuss the lower bounds and parallel algorithms on a dozen or so computations and DP recurrences, which can be further applied to dozens

| Dimension | Problems | Cache Complexity |
|-----------|----------|------------------|
| $k = 2$   | LWS/GAP*/RNA/knapsack recurrences | $\Theta\left(\frac{C}{BM}\right)$ | $\Theta\left(\frac{\omega^{1/2}C}{BM}\right)$ |
| $k = 3$   | Combinatorial MatMul / LU-decomp, Kleene’s algorithm, Parenthesis recurrence | $\Theta\left(\frac{C}{B\sqrt{M}}\right)$ | $\Theta\left(\frac{\omega^{1/3}C}{B\sqrt{M}}\right)$ |

Table 1: I/O costs of cache-oblivious algorithms based on the $k$-d grid computation structures. Here $C$ is the number of algorithmic instructions in the corresponding computation. (*) For the GAP recurrence, the upper bounds have addition terms as shown in Section \ref{sec:gap}.
of real-world problems\textsuperscript{3}. The results are shown in both settings with or without considering the asymmetric costs between reads and writes.

2 Preliminaries and Related Work

Ideal-cache model and cache-oblivious algorithms. In modern computer architecture, a memory access is much more expensive compared to an arithmetic operation due to larger latency and limited bandwidth (especially in the parallel setting). To capture the cost of an algorithm on memory access, the ideal-cache model, a widely-used cost model, is a two-level memory model comprised of an unbounded memory and a cache of size $M$. Data are transferred between the two levels using cache lines of size $B$, and all computation occurs on data in the cache. The cache complexity (or the I/O cost indistinguishably) of an algorithm is the number of cache lines transferred between the cache and the main memory assuming an optimal (offline) cache replacement policy. An algorithm on this model is cache-oblivious with the additional feature that it is not aware of the value of $M$ and $B$. In this paper, we refer to this cost as the symmetric cache complexity (as opposed to asymmetric memory as discussed later). Throughout the paper, we assume that the input and output do not fit into the cache since otherwise the problems become trivial.

The nested-parallel model and work-depth analysis. In this paper, the parallel algorithms are within the standard nested-parallel model, which is a computation model and provides easy analysis of the work-efficiency and parallelism. In this model, a computation starts and ends with a single root task. Each task has a constant number of registers and runs a standard instruction set from a random access machine, except it has one additional instruction called FORK, which can create two independent tasks one at a time that can be run in parallel. When the two tasks finish, they join back and the computation continues.

A computation can be viewed as a (series-parallel) DAG in the standard way. The cost measures on this model are the work and depth: we say work $W$ to be the total number of operations in this DAG and depth (span) $D$ equals to the longest path in the DAG. Such a computation can be scheduled on the round-synchronous PRAM model with $p$ processors in $W/p + O(D)$ round (time) using the randomized work-stealing scheduler \cite{28}. All algorithms in this paper assume concurrent reads and exclusive writes (CREW). Here we do not distinguish the extra write cost for asymmetric memory on $W$ and $D$ to simplify the description of the results, and we only capture this asymmetry using cache complexity.

Regarding parallel cache complexity, Blelloch et al. \cite{22} suggest that analyzing the depth and sequential cache complexity of an algorithm is sufficient for deriving upper bounds on parallel cache complexity. In particular, let $Q_1$ be the sequential cache complexity. Then for a $p$-processor shared-memory machine with private caches (i.e., each processor has its own cache) using a work-stealing scheduler, the total number of cache misses $Q_p$ across all processors is at most $Q_1 + O(pDM/B)$ with high probability \cite{1}. For a $p$-processor shared-memory machine with a shared cache of size $M + pBD$ using a parallel-depth-first (PDF) scheduler, $Q_p \leq Q_1$ \cite{17}. We can extend these bounds to multi-level hierarchies of private or shared caches, respectively \cite{22}.

Parallel and cache-oblivious algorithms for dynamic programming and linear algebra. Dynamic Programming (DP) is an optimization strategy that decomposes a problem into subproblems with optimal substructure. It has been studied for over sixty years \cite{4,13,41}. For the problems that we consider in this paper, the parallel DP algorithms were already discussed by a rich literature in the eighties and nighties (e.g.,

\textsuperscript{3}Like in this paper we abstract the “2-knapsack recurrence”, which fits into our $k$-d grid computation structure and applies to many algorithms.

\textsuperscript{4}In this paper, we often assume the cache size to be $O(M)$ since it simplifies the description and only affects the bounds by a constant factor.
Later work not only considers parallelism, but also optimizes symmetric cache complexity (e.g., [22, 34, 35, 37, 39, 44, 51, 69, 87, 88, 89, 91]). The algorithms in linear algebra that share the similar computation structures (but with different orders in the computation) are also discussed (e.g., [12, 27, 39, 43, 75, 92, 98]).

**Problem definitions.** Since we are showing many optimal cache-oblivious algorithms, we need formal problem definitions. It is hard to show general lower bounds that any type of computations or operations is allowed. For example, for matrix multiplication on a semiring, the only known lower bound of operations is just $\Omega(n^{3-o(1)})$ assuming SETH (more details of fine-grain complexity in [97]). Here we make no assumptions of the set of the ring, and we consider both operators to be atomic and unable to be decomposed or batched. We borrow the term **combinatorial matrix multiplication** to indicate this specific problem. Such an algorithm requires $\Theta(n^3)$ operations on square matrices of size $n$ ($n^2$ inner products). Regarding dynamic programming in Section 8, we discuss the recurrences rather than the problems, and again each operation is atomic and non-decomposable or batchable. We abstract such operation as a computation cell in Section 3.

**Algorithms with asymmetric read and write costs.** The future of main memory appears to lie in a new wave of non-volatile memory (NVM) technologies that promise persistence, significantly lower energy costs, and higher density than the DRAM technology used in today’s main memories [62, 67, 82, 102]. Despite the advantages, a key property of such memory technologies, however, is their asymmetric read-write costs: compared to reads, writes can incur much higher energy, higher latency, lower (per-module) bandwidth, and prone to wear-out [6, 9, 18, 19, 31, 33, 45, 46, 63, 66, 71, 78, 81, 85, 99, 100, 103, 104]. Moreover, because bits are stored in these technologies as at rest “states” of the given material that can be quickly read but require physical change to update, this asymmetry appears fundamental. This motivates the need for write-efficient algorithms that reduce the number of writes compared to existing algorithms. The early research studied such read-write asymmetry on NAND Flash chips [14, 48, 54, 84] and database operators [32, 93, 94].

Blelloch et al. [15, 18, 19] formally defined and analyzed several sequential and parallel computation models that take asymmetric read-write costs into account. The model Asymmetric RAM (ARAM) extends the two-level memory model and contains a parameter $\omega$, which corresponds to the cost of a write relative to a read to the non-volatile main memory. In this paper, we refer to the **asymmetric cache complexity** $Q$ as the number of write transfers to the main memory multiplied by $\omega$, plus the number of read transfers. This model captures different system consideration (latency, bandwidth, or energy) by simply plugging in a different value of $\omega$, and also allows algorithms to be analyzed theoretically and practically. Similar scheduling results (upper bounds) on parallel running time and cache complexity are discussed in [15, 19] based on work $W$, depth $D$ and asymmetric cache complexity $Q$ of an algorithm. Based on this idea, many interesting algorithms and lower bounds are designed and analyzed by various recent works [15, 16, 18, 19, 21, 23, 25, 58, 70].

In the analysis, we always assume that the input size is much larger than the cache size (which is usually the case in practice). Otherwise, both the upper and the lower bounds on cache complexity also include the term for output—$\omega$ times the output size. For simplicity, this term is ignored in the asymptotic analysis.

Carson et al. [31] also discussed algorithms using less writes. Since their results are under some different assumptions, we discuss the difference in the Appendix E.

**Discussions to previous work.** We now discuss several possible confusions of this paper. In this paper, we define and analyze the $k$-d grid computation structure. Similar grid structures and the bounds on computing them have been discussed in many previous work (e.g., [2, 10, 11, 34, 37, 39, 68, 69, 91]). However, we note that the definition in this paper is different from those papers. In the $k$-d grid, the dimension of the grid is related to the number of entries per basic computation unit (formal definition in Section 3), not the dimension of the input/output arrays. As a result, the readers should not confuse the $k$-d grid in this paper with the previous grid structures.
Since we believe that the abstraction and definition are new, the proof of lower bounds in Section 4 focuses on the simplest sequential setting with a limited-size cache, which is different from the parallel or distributed settings with infinite-size local memory discussed in previous work (e.g., [2, 10, 11, 68]). It is an interesting future work to extend the results in this paper to the more complicated settings, especially in the asymmetric setting.

Although the definition of $k$-d grid is different, in the special case when the number of input entries per basic block is the same as the dimension of input/output arrays, the analysis based on $k$-d grid provides the same sequential symmetric cache complexity as the previous work [34, 37, 39, 51, 69, 91]. The sequential symmetric versions of these algorithms based on the $k$-d grid are the same as or similar to the previous algorithms. One of such examples is the matrix multiplication [51], and we recommend the reader to use it as an instantiation to understand our improved parallel and asymmetric algorithms in Section 5 and 6. This is also the case for other linear algebra algorithms in Section 7.2 and the LWS and Parenthesis recurrences in Section 8.1 and 8.4. For the algorithms with the same bounds as previous, we give another way to describe and analyze them — this is a minor result, which we do not claim it as a contribution of this paper.

Some previous work [44, 89] achieves the same linear depth bounds in several problems discussed in this paper. We note that they assume a much stronger model (i.e., enabling the DAG to be non-nested), so their algorithms either need a specially designed scheduler [44], or cannot benefit from the scheduling results discussed in this preliminary section. Our algorithms are much simpler and under the nested-parallel model.

The dynamic programming recurrences discussed in this paper have non-local dependencies (definition given in [57]), and we point out that they are pretty different from the problems like edit distance or stencil computations (e.g., [36, 52, 59, 77]) that only have local dependencies. We did not consider other types of dynamic programming approaches like rank convergence or hybrid $r$-way DAC algorithms [38, 79, 80] that cannot guarantee processor- and cache-obliviousness simultaneously.

3 The $k$-d Grid Computation Structure

The $k$-d grid computation structure (short for the $k$-d grid) is defined as a $k$-dimensional grid $C$ of size $n_1 \times n_2 \times \cdots \times n_k$. Here we consider $k$ to be a small constant greater than 1. This computation requires $k - 1$ input arrays $I_1, \ldots, I_{k-1}$ and generate one output array $O$. Each array has $k - 1$ dimensions and is the projection of the grid out of one different dimension. Each cell in the grid represents some certain computation that requires $k - 1$ inputs and generates a temporary value. This temporary value is “added” to the corresponding location in the output array, and we assume this addition is associative. The $k - 1$ inputs of this cell are the projections of this cell removing each (but not the last) dimensions, and the output is the projection removing the last dimension. They are referred to as the input and output entries of this cell. When computing each cell, the input and output entries must be in the cache. Figure 1 illustrates such a computation in 2 and 3 dimensions.

We refer to a $k$-d grid computation structure as a square grid computation structure (short for a square grid) of size $n$ if it has size $n_1 = \cdots = n_k = n$. More concisely, we say a $k$-d grid has size $n$ if it is square and of size $n$.

For instance, when multiplying two matrices of size $n$-by-$n$ on a semiring $(\times, +)$, the computation of the classic cubic algorithm exactly matches the 3d square grid. A corresponding 2d case is when computing a matrix-vector multiplication where the matrix is implicit (i.e., we can query the value of each cell, but the matrix is not stored explicitly). Such applications are commonly seen in dynamic programming algorithms.

The key aspect to decide the dimension of a computation is the number of inputs that each basic cell element requires. For example, when multiplying two dense tensors, although each tensor may have multiple dimensions, each multiplication operation is only based on two entries, so it is a 3d grid. The cache-oblivious
Figure 1: An illustration of a 2d and a 3d grid. The left figure shows the 2d case where the input $I_1$ and output $O$ are 1d arrays, and each computation cell requires exactly one entry in $I_1$ as input, and update one entry in $O$. For the 3d case on the right, the inputs and output are 2d arrays, and each computation cell requires one entry from input $I_1$ and one from input $I_2$. The input/output entries of each cell are the projections of this cell on different 2d arrays.

algorithms discussed in this paper are based on $k$-d grids with $k = 2$ or 3, but we can also find applications with larger $k$ (e.g., a Nim game with some certain rules on multiple piles [29]).

For some dynamic programming and linear algebra algorithms in this paper, the computations can be abstracted as $k$-d grids, but a constant fraction of the grid cells are empty. We call such a grid as an $\alpha$-full grid for some constant $0 < \alpha < 1$ if at least an $\alpha \pm o(1)$ fraction of the cells are non-empty. We will show that all properties we show for a $k$-d grid also work for the $\alpha$-full case, since the constant $\alpha$ affects the analysis of neither the lower bounds nor the algorithms.

4 The Lower Bounds

We first discuss the lower bounds of the cache complexities for a $k$-d grid computation structure, which sets the target to design the algorithms in the following sections. In Section 4.1 we show the symmetric cache complexity. This is a direct extension of the classic result by Hong and Kong [61] to an arbitrary dimension. Then in Section 4.2 we discuss the asymmetric cache complexity when writes are more expensive than reads, which is more interesting and has involved analyses.

4.1 Symmetric Cache Complexity

The symmetric cache complexity of a $k$-d grid is simple to analyze, yielding to the following result:

**Theorem 4.1.** The symmetric cache complexity of a $k$-d grid computation structure with size $n$ is $\Omega\left(\frac{n^k}{M^{1/(k-1)}B}\right)$.

**Proof.** Let’s say the computations of $n^k$ cells are sequentialized in an array. For blocks of size $S = 2^k M^{k/(k-1)}$, the minimum number of inputs and outputs of each block is at least $2^{k-1}kM \geq 2M$. Since only a total amount of $M$ entries can be held in the cache at the beginning of the computation, the cache complexity to load the input/output and finish the computation for such a block is $\Omega(M/B)$. Since there are $n^k/S = \Theta(n^k M^{-k/(k-1)})$ of such blocks, the cache complexity of the entire computation is $\Omega(M/B) \cdot n^k/S = \Omega(n^k/(M^{1/(k-1)}B))$. \qed
Notice that the proof does not assume cache-obliviousness, but the lower bound is asymptotically tight by applying a sequential cache-oblivious algorithm that is based on \(2^k\)-way divide-and-conquer [51].

### 4.2 Asymmetric Cache Complexity

We now consider the asymmetric cache complexity of a \(k\)-d grid computation structure. Unfortunately, this case is significantly harder than the symmetric setting. Again for simplicity we first analyze the square grid of size \(n\), which can be extended to the more general cases similar to [51].

Interestingly, there is no specific pattern that a cache-oblivious algorithm has to follow. Some existing algorithms use “buffers” to support cache-obliviousness (e.g., [7]), and many others use a recursive divide-and-conquer framework. For the recursive approaches, when the cache complexity of the computation is not leaf-dominated (like various sorting algorithms [18, 51]), a larger fan-out in the recursion is more preferable (usually set to \(O(\sqrt{n})\)). Otherwise, when it is leaf-dominated, existing efficient algorithms all pick a constant fan-out in the recursion in order to reach the base case as soon as possible. All problems we discuss in this paper are in this category, so we make our analysis under the following constraints. More discussion about this constraint is given in Section 9.

**Definition 1 (CBCO paradigm).** We say a divide-and-conquer algorithm is under the constant-branching cache-oblivious (CBCO) paradigm if it has an input-value independent computational DAG, such that each task has constant\(^5\) fan-outs of its successive subtasks until the base cases, and the partition of each task is decided by the (sub)problem size and independent with the cache parameters \((M, B)\).

Notice that \(\omega\) is a parameter of the main memory, instead of a cache parameter. One can define resource-obliviousness [40] so that the value of \(\omega\) is not exposed to the algorithms, but this is out of the scope of this paper.

We now prove the (sequential) lower bound on the asymmetric cache complexity of a \(k\)-d grid under the CBCO paradigm. The two places that we use this assumption in the proof are the constant branching of the recursion, and “scale-free” of the cache-oblivious algorithms: the structure in the recursive levels around the boundary of cache size is identical.

**Theorem 4.2.** The asymmetric cache complexity of \(k\)-d grid is \(\Omega\left(\frac{n^3 \omega^{1/k}}{M^{1/(k-1)}B}\right)\) under the CBCO paradigm.

**Proof.** We prove the lower bound using the same approach in Section 4.1—analyzing blocks of size \(S\) based on the sequence of the operations executed by the algorithm. The cache can hold \(M\) entries as temporary space for the computation. For the lower bound, we only consider the computation in each cell without considering the step adding the calculated value back to the output array. Again when computing each cell, the \(k\) input and output entries have to be in the cache.

For a block of operations of size \(S\), the cache needs to hold the entries in \(I_1, \ldots, I_{k-1}\) and \(O\) corresponding to the cells in this sequence at least once during the computation. The number of entries is minimized when the sequence of operations are within a \(k\)-d cuboid of size \(S = a_1 \times a_2 \times \cdots \times a_k\), where the projections on \(I_i\) and \(O\) are \((k-1)\)-d arrays with sizes \(a_1 \times \cdots \times a_{i-1} \times a_{i+1} \times \cdots \times a_k\) and \(a_1 \times \cdots \times a_{k-1}\). Namely, the number of entries is at least \(S/B \cdot 1/a_i\) for the corresponding input or output array.

Note that the input arrays are symmetric to each other regarding the access cost, but in the asymmetric setting storing the output is more expensive. As a result, the cache complexity is minimized when \(a_1 = \cdots = a_{k-1} = a\), and let’s denote \(a_k = ar\) where \(r\) is the ratio between \(a_k\) and other \(a_i\). Here we assume \(r \geq 1\)

\(^5\)It can exponentially depend on \(k\) where we assume \(k\) is a constant.
since reads are cheaper. Due to the scale-free property that $M$ and $n$ are arbitrary, $r$ should be fixed (within a small constant range) for the entire recursion.

Similar to the analysis in the proof of Theorem 4.1 for a block of size $S$, the read transfers required by the cache is $\Omega\left(\frac{n^k}{SB} \cdot \max\{a^{-1}r - M, 0\}\right)$, where $n^k/S$ is the number of such blocks, and $\max\{a^{-1}r - M, 0\}/B$ lower bounds the number of reads per block because at most $M$ entries can be stored in the cache from the previous block. Similarly, the write cost is $\Omega\left(\frac{\omega n^k}{SB} \cdot \max\{a^{-1} - M, 0\}\right)$. In total, the cost is:

$$Q = \Omega\left(\frac{n^k}{SB} \cdot \left(\max\{a^{-1}r - M, 0\} + \omega \max\{a^{-1} - M, 0\}\right)\right)$$

$$= \Omega\left(\frac{n^k}{SB} \left(\max\{S^{(k-1)/k} - 1/k - M, 0\} + \omega \max\left\{\frac{S^{(k-1)/k} - 1/k - M}{r^{(k-1)/k}} - M, 0\right\}\right)\right)$$

The second step is due to $S = O(a^k r)$.

The cost decreases as the increase of $S$, but it has two discontinuous points $S_1 = M^{k/(k-1)} / r^{1/(k-1)}$ and $S_2 = M^{k/(k-1)} r$. Therefore,

$$Q = \Omega\left(\frac{n^k}{S_1 B} S_1^{1/k} r^{1/k} + \frac{n^k}{S_2 B} S_2^{1/k} r^{1/k} + \frac{\omega n^k}{r^{(k-1)/k}}\right)$$

$$= \Omega\left(\frac{n^k}{S_1^{1/k} B} r^{1/k} + \frac{n^k}{S_2^{1/k} B} r^{1/k} + \frac{\omega}{r^{(k-1)/k}}\right) = \frac{n^3}{M^{1/k} B} \left(\frac{\omega}{M^{1/k} B} + \frac{\omega}{r^{(k-1)/k}}\right)$$

When $r = \omega^{(k-1)/k}$, $Q$ is minimized to be $\Omega\left(\frac{n^3 \omega^{1/k}}{M^{1/(k-1)} B}\right)$ and this leads to the theorem. $\square$

5 A Matching Upper Bound on Asymmetric Memory

In the sequential and symmetric setting, the classic cache-oblivious divide-and-conquer algorithms to compute the $k$-d grid (e.g., 3D case in [51]) is optimal. In the asymmetric setting, the proof of Theorem 4.2 indicates that the classic algorithm is not optimal. The key to improve the cost is the balancing factor $r = \omega^{(k-1)/k}$ that leads to more cheap reads and less expensive writes in each sub-computation.

We now show that the lower bound in Theorem 4.2 is tight by a (sequential) cache-oblivious algorithm with such asymmetric cache complexity. The algorithm is given in Algorithm 1, which can be viewed as a variant of the classic approach with minor modifications on how to partition the computation. Notice that in line 6 and 10 “conceptually” means the partitions are used for the ease of algorithm description. In practice, we can just pass the ranges of indices of the subtask in the recursion, instead of actually partitioning the arrays.

Compared to the the classic approaches (e.g., [51]) that partition the largest input dimension among $n_i$, the only underlying difference in the new algorithm is in line 4 when partitioning the dimension not related to the output array $O$, $n_k$ has to be $\omega^{(k-1)/k}$ times larger than $n_1, \cdots, n_{k-1}$. This modification introduces an asymmetry between the input size and output size of each subtask, which leads to an improvement in the cache efficiency.

For simplicity, we show the asymmetric cache complexity for square arrays (i.e., $n_1 = \cdots = n_k$) and $n = \Omega(\omega^{(k-1)/k} M)$, and the general case can be analyzed similar to [51].
Algorithm 1: ASYM-ALG($I_1, \cdots, I_{k-1}, O$)

| Input:  | $k - 1$ input arrays $I_1, \cdots, I_{k-1}$, read/write asymmetry $\omega$ |
|---------|---------------------------------------------------------------------|
| Output: | Output array $O$                                                      |

1. The computation has size $n_1 \times n_2 \times \cdots \times n_k$

2. **if** $I_1, \cdots, I_{k-1}, O$ are small enough **then**
   3. Solve the base case and **return**
   4. $i \leftarrow \text{arg max}_{1 \leq i \leq k}\{n_i x_i\}$ where $x_k = \omega^{-(k-1)/k}$ and $x_j = 1$ otherwise for $1 \leq j < k$
   5. **if** $i = k$ **then**
      6. (Conceptually) equally partition $I_1, \cdots, I_{k-1}$ into \{I_{1,a}, I_{1,b}\}, \cdots, \{I_{k-1,a}, I_{k-1,b}\} on $k$-th dimension
      7. ASYM-ALG($I_{1,a}, \cdots, I_{k-1,a}, O$)
      8. ASYM-ALG($I_{1,b}, \cdots, I_{k-1,b}, O$)
   9. **else**
      10. (Conceptually) equally partition $I_1, \cdots, I_{i-1}, I_{i+1}, \cdots, I_{k-1}, O$ into \{I_{i,a}, I_{i,b}\}, \cdots, \{I_{k-1,a}, I_{k-1,b}\}, \{O_{a}, O_{b}\} on $i$-th dimension
      11. ASYM-ALG($I_{1,a}, \cdots, I_{i-1,a}, I_{i}, I_{i+1,a}, \cdots, I_{k-1,a}, O_{a}$)
      12. ASYM-ALG($I_{1,b}, \cdots, I_{i-1,b}, I_{i}, I_{i+1,b}, \cdots, I_{k-1,b}, O_{b}$)

**Theorem 5.1.** Algorithm 1 computes the $k$-d grid of size $n$ with asymmetric cache complexity $\Theta\left(\frac{n^k \omega^{1/k}}{M^{1/(k-1)} B}\right)$.

**Proof.** We separately analyze the numbers of reads and writes required by Algorithm 1. In the sequential execution of Algorithm 1, each subproblem only requires $O(1)$ extra temporary space. Also, our analysis ignores rounding issues since they will not affect the asymptotic bounds.

When starting from the square grid at the beginning, the algorithm first partitions in the first $k - 1$ dimensions (via line 10 to 12) into $\omega^{(k-1)/k}$ subproblems (refer to as second-phase subproblems) each with size $(n/\omega^{(k-1)/k}) \times \cdots \times (n/\omega^{(k-1)/k}) \times n$, and then partition $k$ dimensions in turn.

The number of writes of the algorithm $W(n)$ (to array $O$) follows the recurrences that:

$$W'(n) = 2^k W'(n/2) + O(1)$$

and

$$W(n) = (\omega^{(k-1)/k})^{k-1} \cdot \left(W'(n/\omega^{(k-1)/k}) + O(1)\right)$$

where $W'(n)$ is the number of writes required by the second-phase subproblems with the size of $O$ being $n \times \cdots \times n$. The base case is when $W'(M^{1/(k-1)}) = O(M/B)$. Solving the recurrences gives $W'(n/\omega^{(k-1)/k}) = O\left(\frac{n^k \omega^{1-k}}{M^{1/(k-1)} B}\right)$, and $W(n) = O\left(\frac{n^k \omega^{(1-k)/k}}{M^{1/(k-1)} B}\right)$.

We can analyze the reads similarly by defining $R(n)$ and $R'(n)$. The recurrences on reads are:

$$R'(n) = 2^k R'(n/2) + O(1)$$

and

$$R(n) = (\omega^{(k-1)/k})^{k-1} \cdot \left(R'(n/\omega^{(k-1)/k}) + O(1)\right)$$
The difference occurs in the base case since the input fits into the cache when \( n = M^{1/(k-1)}/\omega^{1/k} \). Namely, \( R'(M^{1/(k-1)}/\omega^{1/k}) = O(M/B) \). Therefore, by solving the recurrences, we have \( R'(n/\omega^{(k-1)/k}) = O\left(\frac{n^k \omega^{2-k}}{M^{1/(k-1)} B}\right) \).

The overall (sequential) asymmetric cache complexity for Algorithm 1 is:

\[
Q(n) = R(n) + \omega W(n) = O\left(\frac{n^k \omega^{1/k}}{M^{1/(k-1)} B}\right)
\]

and combining with the lower bound of Theorem 4.2 proves the theorem.

Comparing to the classic approach, the new algorithm improves the asymmetric cache complexity by a factor of \( O(\omega^{(k-1)/k}) \), since the classic algorithm requires \( \Theta(n^k/(M^{1/(k-1)} B)) \) reads and writes. Again here we assume \( n^{k-1} \) is much larger than \( M \). Otherwise, the lower and upper bounds should include \( \Theta(\omega n^k/ B) \) for just writing down the output \( O \).

6 Parallelism

We now show the parallelism in computing the \( k \)-d grids. We show that the parallel versions of the cache-oblivious algorithms only has polylogarithmic depth, indicating that they are highly parallelized.

6.1 The Symmetric Case

We first discuss the classic algorithm on symmetric memory. For a square grid, the algorithm partition the \( k \)-dimensions in turn until the base case is reached.

Notice that in every \( k \) consecutive partitions, there are no dependencies in \( k - 1 \) of them, which we can take full parallelism from them. The only exception is during the partition in the \( k \)-th dimension, whereas both subtasks share the same output array \( O \) and cause write concurrence. If such two subtasks are sequentialized (like in [51]), the depth is \( D(n) = 2D(n/2) + O(1) = O(n) \).

We now introduce the algorithm with logarithmic depth. As just explained, to avoid the two subtasks from editing the same elements in the output array \( O \), our algorithm works in the following way when partitioning the \( k \)-th dimension:

1. Allocating two stack-allocated temporary arrays with the same size of the output array \( O \) before the two recursive function calls.
2. Applying computation for the \( k \)-d grid in two subtasks with different output arrays that are just allocated (with no concurrency to the other subtask).
3. When both subtasks finish, the computed values are merged back in parallel, with work proportional to the output size and \( O(\log n) \) depth.
4. Deallocating the temporary arrays.

Notice that the algorithm also works if we only allocate temporary space for one of the subtasks, while the other subtask still works on the original space for the output array. This can be a possible improvement in practice, but in high dimensional case \( (k > 2) \) it requires more complicated details to pass the pointers of the output arrays to descendant nodes, aligning arrays to cache lines, etc. Theoretically, this version does not change the bounds except for the stack space in Lemma 6.1 when \( k = 2 \).
We first analyze the cost of square grids of size $n$ in the symmetric setting, and will discuss the asymmetric setting later.

**Lemma 6.1.** The overall stack space for a subtask of size $n$ is $O(n^{k-1})$.

**Proof.** When partitioning the output ($k$-th) dimension, the algorithm allocates and computes two subtasks of size $n/2$. This leads to the following recurrence:

$$S(n) = 2S(n/2) + O(n^{k-1})$$

which solves to $S(n) = O(n^{k-1})$ when $k > 2$. When $k = 2$, we can apply the version that only allocates temporary space for one subtask, which changes the constant before $S(n/2)$ to 1, and yields to the same bound as $S(n) = O(n)$. Note that we only need to analyze one of the branches, since the temporary spaces that are not allocated in the direct ancestor of this subtask have already been deallocated, and will be reused for later computations for the current branch. ◻

With the lemma, we have the following corollary:

**Corollary 6.2.** A subtask of size $n \leq M^{1/(k-1)}$ can be computed within a cache of size $O(M)$.

This corollary indicates that this modified parallel algorithm has the same sequential cache complexity since it fits into the cache in the same level as the classic algorithm (the only minor difference is the required cache size increases by a small constant factor). Therefore we can apply the a similar analysis in [51] ($k = 3$ in the paper) to show the following lemma:

**Lemma 6.3.** The sequential symmetric cache complexity of the parallel cache-oblivious algorithm to compute a $k$-d grid of size $n$ is $O(n^3/M^{1/(k-1)}B)$.

Assuming that we can allocate a chunk of memory in constant time, the depth of this approach is simply $O((log^2 n) - O(log n)$ levels of recursion, each with $O(log n)$ depth for the additions [22].

We have shown the parallel depth and symmetric cache complexity. Therefore, we have the following result for parallel symmetric cache complexity by applying the scheduling theorem in the preliminary section.

**Corollary 6.4.** The $k$-d grid of size $n$ can be computed with the parallel symmetric cache complexity of $O(n^3/M^{1/(k-1)}B + pM log^2 n)$ with private caches, or $O(n^3/M^{1/(k-1)}B)$ with share caches of size $M + pB log^2 n$.

Lemma 6.1 shows that the extra space required is $S_1 = O(n^{k-1})$ for sequentially running the parallel algorithm. Naively the parallel space requirement is $pS_1$, which can be very large. We now show a better upper bound for the extra space.

**Lemma 6.5.** The overall space requirement of the $k$-d grid is $O(p^{1/k}n^{k-1})$.

**Proof.** We analyze the amount of space allocated in total for all processors. Lemma 6.1 indicates that if the root of the computation on one processor has the output array of size $(n^k)^{k-1}$, then the space requirement for this task is $O((n^k)^{k-1})$. There can be at most $2^k$ processors starting with their computations of size $n^{k-1}/2^{k-1}, (2^{k})^2$ of size $n^{k-1}/(2^{k-1})^2$, until $(2^{k})^q$ processors of size $n^{k-1}/(2^{k-1})^q$ where $q = \log_{2^k} p$, when each of the $p$ processors all get a task. This is the most pessimistic case that maximizes the overall space requirement, which is:

$$\sum_{h=1}^{\log_{2^k} p} O\left(\frac{n^{k-1}}{(2^{k-1})^h}\right) \cdot (2^{k})^h = p \cdot O\left(\frac{n^{k-1}}{(2^{k-1})^{\log_{2^k} p}}\right) = O\left(p^{1/k}n^{k-1}\right)$$

which gives the stated bound. ◻
We believe the space requirement for the parallel cache-oblivious algorithm is acceptable since it is asymptotically the same as the most intuitively (non-cache-oblivious) parallel algorithm that partitions the computation into \( p \) square subtasks each with size \( n/p^{1/k} \). In practice nowadays it is easy to fit several terabyte main memory onto a single powerful machine such that the space requirement can usually be satisfied. For example, when \( k = 2, p = 100 \) and the main memory can hold \( 10^{12} \) entries, the grid needs to contain \( \approx 10^{22} \) cells to exceed the memory size: such computation can be too big to run on a single shared-memory machine. For \( k \geq 3 \), the extra term \( p^{1/k} \) only affects a small range of input sizes that cannot fit. Even if it falls into this small region, we can always slightly change the algorithm to bound the extra space. We first partition the input dimensions for \( \log_2 p \) rounds to bound the largest possible output size to be \( O(n^{k-1}/p) \) that one processor can steal (similar to the case discussed in Section 6.2). Then the overall extra space is limited to \( O(n^{k-1}) \), the same as the input/output size. If needed, the constant in the big-O can also be bounded. Such change will not affect the cache complexity and the depth as long as the main memory size is larger than \( pM \).

In practice, it is always several orders of magnitude larger than \( pM \), and the extra space requirement is acceptable in most cases in practice. Throughout the rest of the paper, we focus on the cache complexity and the depth, similar to the previous work [22].

Combining all results gives the following theorem:

**Theorem 6.6.** There exists a cache-oblivious algorithm for \( k \)-d grid computation structure of size \( n \) that requires \( \Theta(n^k) \) work, \( \Theta\left(\frac{n^k}{M^{1/(k-1)}B}\right) \) symmetric cache complexity, \( O(\log^2 n) \) depth, and \( O(p^{1/k}n^{k-1}) \) main memory size.

### 6.2 The Asymmetric Case

Algorithm 1 considers the write-read asymmetry, which involves some minor changes to the classic cache-oblivious algorithm. In terms of parallelism, the changes only affect the order of the partitioning of the \( k \)-d grid in the recurrence, but not the parallel version and the analysis in Section 6.1. As a result, the depth of the parallel variant of Algorithm 1 is also \( O(\log^2 n) \). The extra space requirement is actually decreased, because the asymmetric algorithm has a higher priority in partitioning the input dimensions that does not require allocation temporary space.

**Lemma 6.7.** The space requirement of Algorithm 1 on \( p \) processors is \( O(n^{k-1}(1 + p^{1/k}/\omega^{(k-1)/k})) \).

**Proof.** Algorithm 1 first partition the input dimensions until \( q = O(\omega^{(k-1)/k}) \) subtasks are generated. Then the algorithm will partition \( k \) dimensions in turn. If \( p < q \), then each processor requires no more than \( O(n^{k-1}/q) \) extra space at any time, so the overall extra space is \( O(p \cdot n^{k-1}/q) = O(n) \). Otherwise, the worst case appears when \( O(p/q) \) processors work on each of the subtasks. Based on Lemma 6.5, the extra space is bounded by \( O((p/q)^{1/k} \cdot q \cdot n^{k-1}/q) = O(p^{1/k}n^{k-1}/\omega^{(k-1)/k}) \). Combining the two cases gives the stated bounds. \( \square \)

**Lemma 6.7** indicates that Algorithm 1 requires extra space no more than the input/output size asymptotically when \( p = O(\omega^{k-1}) \), which should always be true in practice.

The challenge arises in scheduling this computation. The scheduling theorem for the asymmetric case [15] constraints on the non-leaf stack memory to be a constant size. This contradicts the parallel version in Section 6.1. This problem can be fixed based on Lemma 6.1 that upper bounds the overall extra memory on one task. Therefore the stack-allocated array can be moved to the heap space. Once a task is stolen, the first allocation will annotate a chunk of memory with size order of \( |O| \) where \( O \) is the current output. Then all successive heap-based memory allocation can be simulated on this chunk of memory. In this manner,
the stack memory of each node corresponding to a function call is constant, which allows us to apply the scheduling theorem in \[15\].

**Theorem 6.8.** Algorithm 1 with input size \(n\) requires \(\Theta(n^k)\) work, \(\Theta\left(\frac{n^k \omega^{1/k}}{M^{1/(k-1)}B}\right)\) asymmetric cache complexity, and \(O(\log^2 n)\) depth for a \(k\)-d grid of size \(n\).

7 Numerical Algorithms and All-Pair Shortest Paths

In this section we discuss matrix multiplication, Kleene’s algorithm on all pair shortest-paths, and some linear algebra algorithms including Strassen algorithm, Gaussian elimination (LU decomposition), and triangular system solver. The common theme in these algorithms is that their computation structures are very similar to that of matrix multiplication, which is a 3d grid. Strassen algorithm is slightly different and introduced separately in Appendix A. Other algorithms are summarized in Section 7.2 and the details are given in Appendix B–D.

We show improved asymmetric cache complexity for all problems. For Gaussian elimination and triangular system solver, we show linear depth algorithms in both symmetric and asymmetric settings which are based on the parallel algorithm discussed in Section 6. There exist work-optimal and sublinear depth algorithm for APSP [90], but we are unaware of how to make it I/O-efficient. Compared to previous linear depth algorithms in [44], our algorithm is in the classic nested-parallel model and can be scheduled using the work-stealing scheduler. Also, we believe our algorithms are significantly simpler.

7.1 Matrix Multiplication

The combinatorial matrix multiplication (definition in Section 2) is one of the simplest cases of the 3d grid. Given a semiring \((\times, +)\), in matrix multiplication each cell corresponds to a “\(\times\)” operation of the two corresponding input values and the “+” operation is associative. Since there are no dependencies between the operations, we can simply apply Theorem 6.6 and 6.8 to get the following result.

**Corollary 7.1.** Combinatorial matrix multiplication of size \(n\) can be solved in \(\Theta(n^3)\) work, optimal symmetric and asymmetric cache complexity of \(\Theta\left(\frac{n^3}{B\sqrt{M}}\right)\) and \(\Theta\left(\frac{\omega^{1/3} n^3}{B\sqrt{M}}\right)\) respectively, and \(O(\log^2 n)\) depth.

7.2 All-Pair Shortest Paths, Gaussian Elimination, and Triangular System Solver

We now discuss the well-known cache-oblivious algorithms to solve all-pair shortest paths (APSP) on a graph, Gaussian elimination (LU decomposition), and triangular system solver. These algorithms share similar computation structures and can usually be discussed together. Chowdhury and Ramachandran [37, 39] categorized matrix multiplication, APSP, and Gaussian Elimination into the Gaussian Elimination Paradigm (GEP) and discussed a unified framework to analyze complexity, parallelism and actual performance. We show how the parallel depth and the asymmetric cache complexity can be improved using the algorithms we just introduced in Section 5 and 6.

We discuss the details of these cache-oblivious algorithms in the appendix. The common theme in these algorithms is that, the computation takes one or two square matrix(ces) of size \(n \times n\) as input, applies \(n^3\) operations, and generates output as a square matrix of size \(n \times n\). Each output entry is computed by an inner product of one column and one row of either the input matrices or the output matrix in some intermediate
state. Namely, the output $A_{i,j}$ requires input $B_{i,k}$ and $C_{k,j}$ for $1 \leq k \leq n$ ($A$, $B$ and $C$ may or may not be the same matrix). Therefore, we can apply the results of 3d grids on these problems.\footnote{For Gaussian Elimination $A_{k,k}$ is also required, but $A_{k,k}$ is only on the diagonal, which requires a lower-order of cache complexity to load when computing a sub-cubic of a 3d grid.} Note that some of the grids are full (e.g., Kleene’s algorithm) while others are not, but they are all $\alpha$-full and contain $O(n^3)$ operations.

The data dependencies in these algorithms are quite different from each other, but the recursions for cache complexity $Q(n)$ and depth $D(n)$ for APSP, Gaussian elimination and triangular system solver are all in the following form:

$$Q(n) = \beta Q(n/2) + \gamma Q_{3C}(n/2)$$
$$D(n) = 2D(n/2) + \delta D_{3C}(n/2)$$

where $Q_{3C}(n)$ and $D_{3C}(n)$ are the cache complexity and depth of a 3d grid of size $n$. Here, as long as the the recursive subtask fits into the cache together with the 3d grid computation and the $\beta$, $\gamma$ and $\delta$ are constants and satisfy $\beta < 8$, we can show the following bounds.

**Theorem 7.2.** Kleene’s algorithm for APSP, Gaussian elimination and triangular system solver of size $n$ can be computed in $\Theta(n^3)$ work, symmetric and asymmetric cache complexity of $O\left(\frac{n^3}{B\sqrt{M}}\right)$ and $O\left(\frac{\omega^{1/3}n^3}{B\sqrt{M}}\right)$ respectively, and $O(n)$ depth.

### 8 Dynamic Programming Recurrences

In this section we discuss a number of new results on dynamic programming (DP). To show interesting lower and upper bounds on parallelism and cache efficiency in either symmetric and asymmetric setting, we focus on the specific DP recurrences instead of the problems. We assume the operations in the recurrences are atomic, and not decomposable or batchable.

The goal of this section is to show how the DP recurrences can be viewed as and decomposed into the $k$-d grids. Then the lower and upper bounds discussed in Section\footnote{For Gaussian Elimination $A_{k,k}$ is also required, but $A_{k,k}$ is only on the diagonal, which requires a lower-order of cache complexity to load when computing a sub-cubic of a 3d grid.} and\footnote{For Gaussian Elimination $A_{k,k}$ is also required, but $A_{k,k}$ is only on the diagonal, which requires a lower-order of cache complexity to load when computing a sub-cubic of a 3d grid.} as well as the analysis of parallelism in Section\footnote{For Gaussian Elimination $A_{k,k}$ is also required, but $A_{k,k}$ is only on the diagonal, which requires a lower-order of cache complexity to load when computing a sub-cubic of a 3d grid.} can be easily applied to the computation of these DP recurrences. When the dimension of the input/output is the same as the number of entries in each grid cell, then the sequential and symmetric versions of the algorithms in this section are the same as the existing ones discussed in [34, 37, 39, 51, 91], but the others are new. Also, the asymmetric versions and most parallel versions are new. We improve the existing results on symmetric/asymmetric cache complexity, as well as parallel depth.

**Symmetric cache complexity.** We show improved algorithms on a number of problems when the number of entries per cell differs from the dimension of input/output arrays, including the GAP recurrence, protein accordion folding, and RNA recurrence. We show that the previous cache bound $O(n^3/B\sqrt{M})$ for the GAP recurrence and protein accordion folding is not optimal, and we improve the bounds in Theorem 8.2 and 8.3. For RNA recurrence, we show an optimal cache complexity of $\Theta(n^4/BM)$ in Theorem 8.2, which improves the best existing result by $O(M^{3/4})$.

**Asymmetric cache complexity.** By replacing the computation of the $k$-d grid using the asymmetric version discussed in Section\footnote{For Gaussian Elimination $A_{k,k}$ is also required, but $A_{k,k}$ is only on the diagonal, which requires a lower-order of cache complexity to load when computing a sub-cubic of a 3d grid.} we show a uniform approach to provide write-efficient algorithms for all problems in this section. We also prove the optimality of all these algorithms expect for the GAP recurrence.

**Parallelism.** The parallelism of these algorithms is provided by using the parallel algorithms discussed in Section\footnote{For Gaussian Elimination $A_{k,k}$ is also required, but $A_{k,k}$ is only on the diagonal, which requires a lower-order of cache complexity to load when computing a sub-cubic of a 3d grid.} as the underlying building blocks for computing the DP recurrences. We can achieve polylogarithmic
depth in computing the 2-knapsack recurrence, and linear depth in LWS recurrence and protein accordion folding. The linear depth for LWS can be achieved by previous work [44, 89], but they are not in the nested parallel model and does not have the guarantee by the randomized work-stealing scheduler. Meanwhile, our algorithms are arguably simpler.

### 8.1 The LWS Recurrence

The LWS (least-weighted subsequence) recurrence [60] is one of the most commonly-used DP recurrences in practice. Given a real-valued function $w(i, j)$ for integers $0 \leq i < j \leq n$ and $D_0$, for $1 \leq j \leq n$,

$$D_j = \min_{0 \leq i < j} \{D_i + w(i, j)\}$$

This recurrence is widely used as a textbook algorithm to compute optimal 1D clustering [73], line breaking [74], longest increasing sequence, minimum height B-tree, and many other practical algorithms in molecular biology and geology [56, 57], computational geometry problems [3], and more applications in [76]. Here we assume that $w(i, j)$ can be computed in constant work based on a constant size of input associated to $i$ and $j$, which is true for all these applications. Although different special properties of the weight function $w$ can lead to specific optimizations, the study of recurrence itself is interesting, especially regarding cache efficiency and parallelism.

We note that the computation structure (without considering dependencies) of this recurrence is a standard 2d grid. Each cell requires the input entry of $D_i$, computes $D_i + w(i, j)$ and updates $D_j$ as the output entry, so Theorem 4.1 and 4.2 show lower bounds on cache complexity on this recurrence (the grid is (1/2)-full).

We now introduce cache-oblivious implementation considering the data dependencies. Recent work by Chowdhury and Ramachandran [37] solves the recurrence with $O(n^2)$ work and $O(n^2/BM)$ symmetric cache complexity. The algorithm is simply a divide-and-conquer approach and we describe and extend it based on k-d grids. A task of range $(p, q)$ computes the cells $(i, j)$ such that $p \leq i < j \leq q$. To compute it, the algorithm generates two equal-size subtasks $(p, r)$ and $(r + 1, q)$ where $r = (p + q)/2$, solves the first subtask $(p, r)$ recursively, then computes the cells corresponding to $w(i, j)$ for $p \leq i \leq r < j \leq q$, and lastly solves the subtask $(r + 1, q)$ recursively. Note that the middle step exactly matches a 2d grid with no dependencies between the cells, which can be directly solved using the algorithms in Section 5. This leads to the cache complexity and depth to be:

$$Q(n) = 2Q(n/2) + Q_{2C}(n/2)$$

$$D(n) = 2D(n/2) + D_{2C}(n/2)$$

Here $2C$ denotes the computation of a 2d grid. The recurrence is root-dominated and $D(1) = 1$. This solves to the following theorem.

**Theorem 8.1.** The LWS recurrence can be computed in $\Theta(n^2)$ work, $\Theta\left(\frac{n^2}{BM}\right)$ and $\Theta\left(\frac{\omega^{1/2}n^2}{BM}\right)$ optimal symmetric and asymmetric cache complexity respectively, and $O(n)$ depth.

### 8.2 The GAP Recurrence

The GAP problem [55, 57] is a generalization of the edit distance problem that has many applications in molecular biology, geology, and speech recognition. Given a source string $X$ and a target string $Y$, we can apply a sequence of consecutive deletes corresponds to a gap in $X$, and a sequence of consecutive inserts
corresponds to a gap in $Y$. For simplicity here we assume both strings with length $n$, but the algorithms and analyses can easily adapt to the more general case. Since the cost of such a gap is not necessarily equal to the sum of the costs of each individual deletion (or insertion) in that gap, we define $w(p, q)$ ($0 \leq p < q \leq n$) to be the cost of deleting the substring of $X$ from $(p + 1)$-th to $q$-th character, and $w'(p, q)$ for inserting the substring of $Y$ accordingly.

Let $D_{i,j}$ be the minimum cost for such transformation from the prefix of $X$ with $i$ characters to the prefix of $Y$ with $j$ characters, the recurrence for $i, j > 0$ is:

$$D_{i,j} = \begin{cases} \min_{0 \leq q < j} \{D_{i,q} + w(q,j)\} \\ \min_{0 \leq p < i} \{D_{p,j} + w(p,i)\} \end{cases}$$

corresponding to either inserting or deleting a substring. The boundary is set to be $D_{0,0} = 0$, $D_{0,j} = w(0,j)$, and $D_{i,0} = w'(0,i)$. The diagonal dependency from $D_{i-1,j-1}$ can be added if required, without affecting the asymptotic analysis.

The best existing algorithms on GAP Recurrence [37, 88] have symmetric cache complexity of $O(n^3/B\sqrt{M})$. This bound seems to be reasonable, since in order to compute $D_{i,j}$, we need the input of two vectors $D_{i,q}$ and $D_{p,j}$, which is similar to matrix multiplication and other algorithms in Section 7. However, as indicated in this paper, each update in GAP only requires one entry, while matrix multiplication has two. Therefore, if we ignore the data dependencies, the first line of the GAP recurrence can be viewed as $n$ LWS recurrences, independent of the dimension of $i$ (similarly for the second line). This derives a lower bound on cache complexity to be that of an LWS recurrence multiplied by $2n$, which is $\Omega(n^3/ BM)$ (assuming $n > M$). Hence, the gap is $\Theta(\sqrt{M})$ between the lower and upper bounds.

We now discuss an I/O-efficient algorithm to reduce this gap. Unfortunately, the algorithm is not optimal, and we leave it as an open problem. Chowdhury and Ramachandran’s approach [37] is based on divide-and-conquer to compute output $D$. The algorithm recursively partitions $D$ into four equal-size quadrants $D_{00}$, $D_{01}$, $D_{10}$ and $D_{11}$, and starts to compute $D_{00}$ recursively. After this is done, it uses the computed value in $D_{00}$ to update $D_{01}$ and $D_{10}$ using the recurrence. This step can be considered to compute $2 \times n'/2$ LWS recurrences (with no data dependencies) each with size $n'/2$ (assuming $D$ has size $n' \times n'$). Then the algorithm computes $D_{01}$ and $D_{10}$ within their own ranges. After that, it updates $D_{11}$ using the results from $D_{01}$ and $D_{10}$, and solves $D_{11}$ recursively at the end.

Our modified version reorganizes the data layout and the order of computation to take advantage of our I/O-efficient and parallel algorithm on 2d grids. Since the GAP recurrence has two independent sections in different directions, we keep two copies of $D$, one organized in column major and the other in row major. Then when computing on the inter-quadrant updates (e.g., from $D_{00}$ to $D_{01}$) we start $n'/2$ parallel tasks, each to compute a 2d grid on the corresponding row or column, taking the input and output with the correct representation. This update takes the work and cache complexity shown in Theorem 8.1. We also need to keep the consistency of the two copies. After the update of a quadrant $D_{01}$ or $D_{10}$ is finished, we apply a matrix transpose [22] to update the other copy of this quadrant, and the cost of the transpose is a lower-order term. For the quadrant $D_{11}$, we wait until the two updates from $D_{01}$ and $D_{10}$ finish, and then apply the matrix transpose to update the values in each other. It is easy to check that by induction, the values in both copies in a quadrant are update-to-date after each recursion.

The updated algorithm still requires $\Theta(n^3)$ work since it does not require extra asymptotic work. The cache complexity and depth satisfy:

$$Q(n) = 4Q(n/2) + 4n \cdot Q_{2C}(n/2)$$

$$D(n) = 3D(n/2) + 2D_{2C}(n/2)$$
The base cases are $Q(\sqrt{M}) = O(M/B)$ and $Q_{2C}(m) = O(m/B)$ for $m \leq M$ for the symmetric setting. Hence, we have $Q(M) = O((M \log_2 \sqrt{M})/B)$. The top-level computation is root dominated. Therefore, if $n > M$, $Q(n) = O(n^2 Q(M)/M) + O(1) \cdot Q_{2C}(n) = O(n^2/B \cdot (n/M + \log_2 \sqrt{M}))$. Otherwise, $Q(n) = O(n^2 \log_2(n/\sqrt{M})/B)$. Similarly we can compute the asymmetric results.

**Theorem 8.2.** The GAP recurrence can be computed in $\Theta(n^3)$ work, $O(n^{\omega + 2} n)$ depth, symmetric cache complexity of

$$O \left( \frac{n^2}{B} \cdot \left( \frac{n}{M} + \log_2 \min \left\{ \frac{n}{\sqrt{M}}, \sqrt{M} \right\} \right) \right)$$

and asymmetric cache complexity of

$$O \left( \frac{n^2}{B} \cdot \left( \frac{\omega^{1/2} n}{M} + \omega \log_2 \min \left\{ \frac{n}{\sqrt{M}}, \sqrt{M} \right\} \right) \right)$$

Compared to the previous result [22] provides the following theorem. (The computation fully fit into the cache when $n < O(\sqrt{M})$.)

**Protein accordion folding.** The recurrence for protein accordion folding [91] is $D_{i,j} = \max_{1 \leq k < j-1} \left\{ D_{j-1,k} + w(i,j,k) \right\}$ for $1 \leq j < i \leq n$, with $O(n^2/B)$ cost to precompute $w(i,j,k)$. Although there are some minor differences, from the perspective of the computation structure, the recurrence can basically be viewed as only containing the first section of the GAP recurrence. As a result, the same lower bounds of GAP can also apply to this recurrence.

In terms of the algorithm, we can compute $n$ 2d grids with the increasing order of $j$ from 1 to $n$, such that the input are $D_{j-1,k}$ for $1 \leq k < j - 1$ and the output are $D_{i,j}$ for $j < i \leq n$. For short, we refer to a 2d grid as a task. However, the input and output arrays are in different dimensions. To handle it, we use the similar method as the GAP algorithm that keeps two separate matrices, one in column-major and one in row-major. They are used separately to provide the input and output for the 2d grid. We apply the transpose in a divide-and-conquer manner: once the first half of the tasks finish, we transpose all computed values from the output matrix to the input matrix (which is a square matrix), and then compute the second half of the task. Both matrix transposes in the first and second halves are applied recursively with geometrically decreasing sizes. The correctness of this algorithm can be verified by checking the data dependencies so that all required values are computed and moved to the correct positions before they are used for further computations.

The cache complexity is from two subroutines: the computations of 2d grids and matrix transpose. The cost of 2d grids is simply upper bounded by $n$ times the cost of each task, which is $O(n^2/B \cdot (1 + n/M))$ and $O(n^2/B \cdot (\omega + \omega^{1/2} n/M))$ for symmetric and asymmetric cache complexity, and $O(n \log^2 n)$ depth. For matrix transpose, the cost can be verified in the following recursions.

$$Q(n) = 2Q(n/2) + Q_{Tr}(n/2)$$

$$D(n) = 2D(n/2) + D_{Tr}(n/2)$$

where $Tr$ indicates the matrix transpose. The base case is $Q(\sqrt{M}) = O(M/B)$ and $D(1) = 1$. Applying the bound for matrix transpose [22] provides the following theorem.

**Theorem 8.3.** Protein accordion folding can be computed in $O(n^3)$ work, symmetric and asymmetric cache complexity of $\Theta \left( \frac{n^2}{B} \left( 1 + \frac{n}{M} \right) \right)$ and $\Theta \left( \frac{n^2}{B} \left( \omega + \frac{\omega^{1/2} n}{M} \right) \right)$ respectively, and $O(n \log^2 n)$ depth.

The cache bounds in both symmetric and asymmetric cases are optimal with respect to the recurrence.
8.3 RNA Recurrence

The RNA problem \[57\] is a generalization of the GAP problem. In this problem a weight function \( w(p, q, i, j) \) is given, which is the cost to delete the substring of \( X \) from \((p + 1)\)-th to \( i\)-th character and insert the substring of \( Y \) from \((q + 1)\)-th to \( j\)-th character. Similar to GAP, let \( D_{i,j} \) be the minimum cost for such transformation from the prefix of \( X \) with \( i \) characters to the prefix of \( Y \) with \( j \) characters, the recurrence for \( i, j > 0 \) is:

\[
D_{i,j} = \min_{0 \leq p < i, 0 \leq q < j} \{ D_{p,q} + w(p, q, i, j) \}
\]

with the boundary values \( D_{0,0}, D_{0,j} \) and \( D_{i,0} \). This recurrence is widely used in computational biology, like to compute the secondary structure of RNA \[96\].

While the cache complexity of this recurrence seems to be hard to analysis in previous papers, it fits into the framework of this paper straightforwardly. Since each computation in the recurrence only requires one input value, the whole recurrence can be viewed as a 2d grid, with both the input and output as \( D \). The 2d grid contains a constant fraction of the cells, so we can apply the lower bounds in Section 5 here.

Again for a matching upper bound, we need to consider the data dependency. We can apply the similar technique in GAP algorithm to partition the output \( D \) into four quadrants, compute \( D_{00} \), then \( D_{01} \) and \( D_{10} \), and finally \( D_{11} \). Each inter-quadrant update corresponds to a 1/2-full 2d grid. Here maintaining two copies of the array is not necessary with the tall-cache assumption \( M = \Omega(B^2) \). Applying the similar analysis in GAP gives the following result:

**Theorem 8.4.** RNA recurrence can be computed in \( \Theta(n^4) \) work, optimal symmetric and asymmetric cache complexity of \( \Theta\left(\frac{n^4}{BM}\right) \) and \( \Theta\left(\omega^{1/2}n^4/BM\right) \) respectively, and \( O(n^{\log_2 3}) \) depth.

8.4 Parenthesis Recurrence

The Parenthesis recurrence solves the following problem: given a linear sequence of objects, an associative binary operation on those objects, and the cost of performing that operation on any two given (consecutive) objects (as well as all partial results), the goal is to compute the min-cost way to group the objects by applying the operations over the sequence. Let \( D_{i,j} \) be the minimum cost to merge the objects indexed from \( i + 1 \) to \( j \) (1-based), the recurrence for \( 0 \leq i < j \leq n \) is:

\[
D_{i,j} = \min_{i < k < j} \{ D_{i,k} + D_{k,j} + w(i, k, j) \}
\]

where \( w(i, k, j) \) is the cost to merge the two partial results of objects indexed from \( i + 1 \) to \( k \) and those from \( k + 1 \) to \( j \). Here the cost function is only decided by a constant-size input associated to indices \( i, j \) and \( k \). \( D_{i,i+1} \) is initialized, usually as 0. The applications of this recurrence include the matrix chain product, construction of optimal binary search trees, triangulation of polygons, and many others shown in \[41, 56, 57, 101\].

The computation of this recurrence (without considering dependencies) is a \((1/3)\)-full 3d grid, which has the same lower bound shown in Corollary \[7.1\].

The divide-and-conquer algorithm that computes this recurrence is usually hard to describe (e.g., it takes several pages in \[34, 69\] although they also describe their systems simultaneously). We claim that under the view of our \( k\)-d grids, this algorithm is conceptually as simple as the other algorithms. Again this divide-and-conquer algorithm partitions the state \( D \) into quadrants, but at this time one of them \( D_{10} \) is empty since \( D_{i,j} \) does not make sense when \( i > j \). The quadrant \( D_{01} \) depends on the other two. The
algorithm first recursively computes $D_{00}$ and $D_{11}$, then updates $D_{01}$ using the computed values in $D_{00}$ and $D_{11}$, and finally recursively computes $D_{01}$. Here $D_{01}$ is square, so the recursive computation of $D_{01}$ is almost identical to that in RNA or GAP recurrence (although the labeling of the quadrants is slightly changed): breaking a subtask into four quadrants, recursively solving each of them in the correct order while applying inter-quadrant updates in the middle. The only difference is when the inter-quadrant updates are processed, each update requires two values, one in $D_{01}$ and another in $D_{00}$ or $D_{11}$. This is the reason that Parenthesis is 3d while RNA and GAP are 2d. The correctness of this algorithm can be shown inductively.

**Theorem 8.5.** The Parenthesis recurrence can be computed in $\Theta(n^3)$ work, optimal symmetric and asymmetric cache complexity of $\Theta\left(\frac{n^3}{B \sqrt{M}}\right)$ and $\Theta\left(\frac{n^{1/3} n^3}{B \sqrt{M}}\right)$ respectively, and $O(n \log^2 n)$ depth.

### 8.5 2-Knapsack Recurrence

Given $A_i$ and $B_i$ for $0 \leq i \leq n$, the 2-knapsack recurrence computes

$$D_i = \min_{0 \leq j \leq i} \{A_j + B_{i-j} + w(j, i-j, i)\}$$

for $0 \leq i \leq n$. The cost function $w(j, i-j, i)$ relies on constant input values related on indices $i$, $i-j$ and $j$. To the best of our knowledge, this recurrence is first discussed in this paper. We name it the “2-knapsack recurrence” since it can be interpreted as the process of finding the optimal strategy in merging two knapsacks, given the optimal local arrangement of each knapsack stored in $A$ and $B$. Although this recurrence seems trivial, the computation structure of this recurrence actually forms some more complicated DP recurrence. For example, many problems on trees can be solved using dynamic programming, such that the computation essentially applies the 2-knapsack recurrence a hierarchical (bottom-up) manner.

We start by analyzing the lower bound on cache complexity of the 2-knapsack recurrence. The computational grid has two dimensions, corresponding to $i$ and $j$ in the recurrence. If we ignore $B$ in the recurrence, then the recurrence is identical to LWS (with no data dependencies), so we can apply the lower bounds in Section 5 here.

Note that each update requires two input values $A_j$ and $B_{i-j}$, but they are not independent. When computing a subtask that corresponding to $(i, j) \in [i_0, i_0 + n_i] \times [j_0, j_0 + n_j]$, the projection sizes on input and output arrays $A$, $B$ and $D$ are no more than $n_j$, $n_i + n_j$ and $n_i$. This indicates that the computation of this recurrence is a variant of 2d grid, so we can use the same algorithm discussed in Section 5.

**Corollary 8.6.** 2-knapsack recurrence can be computed using $O(n^2)$ work, optimal symmetric and asymmetric cache complexity of $\Theta\left(\frac{n^2}{BM}\right)$ and $\Theta\left(\frac{\omega^{1/2} n^2}{BM}\right)$, and $O(\log^2 n)$ depth.

### 9 Conclusion and Future Work

In this paper, we abstract the $k$-d grid as a basic block to analyze the computation structure of many classic cache-oblivious algorithms. This abstraction provides a more simple and intuitive framework on better understanding the actual computation of these algorithms, proving lower bounds, and designing algorithms.

---

7Such problems can be: (1) computing a size-$k$ independent vertex set on a tree that maximizes overall neighbor size, total vertex weights, etc.; (2) tree properties such that the number of subtrees of certain size, tree edit-distance, etc.; (3) many approximation algorithms on tree embeddings of an arbitrary metric, and many more.
that are both I/O-efficient and highly parallelized. It also provides a unified framework to bound the asymmetric cache complexity of these algorithms.

Because of the new perspective to review these algorithms, we already provide many new results, but we also observe many new open problems. Among them are:

1. The only non-optimal algorithm regarding cache complexity in this paper is for the GAP recurrence. The I/O cost has an additional term of $O((n^2 \log M)/B)$. Although in practice this term will not dominate the running time (the computation has $O(n^3)$ arithmetic operations), it is theoretically interesting to know if we can get rid of this term (even without the constraints of being cache-oblivious or based on divide-and-conquer).

2. We show our algorithms in the asymmetric setting are optimal under the assumption of constant-branching (the CBCO paradigm). Since the cache-oblivious algorithms discussed in this paper are leaf-dominate, we believe this assumption is always true. We wonder if this assumption is necessary (i.e., if there exists a proof without using it, or if there are cache-oblivious algorithms on these problems with non-constant branching but still I/O-optimal).

3. The parallel symmetric cache complexity $Q_p$ on $p$ processors is $Q_1 + O(pDM/B)$, which is a loose upper bound when $D$ is large. Although it might be hard to improve this bound on any general computation under randomized work-stealing, it can be a good direction to show tighter bounds on more regular computation structures like the $k$-d grids or other divide-and-conquer algorithms. We conjecture that the additive term can be shown as optimal (i.e., $O(pM/B)$) for the $k$-d grid computation structures.

4. Due to the page limit, in this paper we mainly discussed the lower bounds and algorithms for square grid computation structures, which is the setting of the problems in this paper (e.g., APSP, dynamic programming recurrences). It is interesting to see a more general analysis on $k$-d grids with arbitrary shape, and such results may apply to other applications like the computation of tensor algebra.

Acknowledgments

This work is supported by the National Science Foundation under CCF-1314590 and CCF-1533858. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. The author thanks Guy E. Blelloch and Yihan Sun for valuable discussions on various ideas, and Yihan Sun and Yuan Tang for the preliminary version of the algorithm in Section 5.

References

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

[2] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Communication complexity of prams. Theor. Comput. Sci., 71(1):3–28, 1990.

[3] Alok Aggarwal and Maria Klawe. Applications of generalized matrix searching to geometric algorithms. Discrete Applied Mathematics, 27(1-2), 1990.
[4] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

[5] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[6] Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajech K. Gupta, and Steven Swanson. Onyx: A prototype phase change memory storage array. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2011.

[7] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1), 2003.

[8] Lars Arge, Gerth Stølting Brodal, and Rolf Fagerberg. Cache-oblivious data structures. *Handbook of Data Structures and Applications*, 27, 2004.

[9] Manos Athanassoulis, Bishwaranjan Bhattacharjee, Mustafa Canim, and Kenneth A. Ross. Path processing using solid state storage. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2012.

[10] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Communication-optimal parallel and sequential cholesky decomposition. *SIAM J. Scientific Computing*, 32(6):3495–3523, 2010.

[11] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Analysis Applications*, 32(3):866–901, 2011.

[12] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Communication costs of strassen’s matrix multiplication. *Communications of the ACM*, 57(2), 2014.

[13] Richard Bellman. *Dynamic programming*. Princeton University Press, 1957.

[14] Avraham Ben-Aroya and Sivan Toledo. Competitive analysis of flash-memory algorithms. In *European Symposium on Algorithms (ESA)*, 2006.

[15] Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Parallel algorithms for asymmetric read-write costs. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2016.

[16] Naama Ben-David, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Implicit decomposition for write-efficient connectivity algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[17] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2004.

[18] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. Sorting with asymmetric read and write costs. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2015.

[19] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. Efficient algorithms with asymmetric read and write costs. In *European Symposium on Algorithms (ESA)*, 2016.
[20] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. Sorting with asymmetric read and write costs. In arXiv preprint:1603.03505, 2016.

[21] Guy E Blelloch, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. The parallel persistent memory model. In ACM symposium on Parallelism in algorithms and architectures (SPAA), 2018.

[22] Guy E Blelloch, Phillip B Gibbons, and Harsha Vardhan Simhadri. Low depth cache-oblivious algorithms. In ACM symposium on Parallelism in algorithms and architectures (SPAA), 2010.

[23] Guy E Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallel write-efficient algorithms and data structures for computational geometry. In ACM symposium on Parallelism in algorithms and architectures (SPAA), 2018.

[24] Guy E. Blelloch, Yan Gu, and Yihan Sun. Efficient construction of probabilistic tree embeddings. In International Colloquium on Automata, Languages, and Programming (ICALP), 2017.

[25] Guy E Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. Parallel shortest paths using radius stepping. In ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2016.

[26] Guy E Blelloch, Anupam Gupta, and Kanat Tangwongsan. Parallel probabilistic tree embeddings, k-median, and buy-at-bulk network design. In ACM symposium on Parallelism in algorithms and architectures (SPAA), 2012.

[27] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In ACM symposium on Parallelism in algorithms and architectures (SPAA), 1996.

[28] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM), 46(5):720–748, 1999.

[29] Charles L Bouton. Nim, a game with a complete mathematical theory. The Annals of Mathematics, 3(1/4), 1901.

[30] Gerth Stølting Brodal. Cache-oblivious algorithms and data structures. In Scandinavian Workshop on Algorithm Theory (SWAT), volume 3111. Springer, 2004.

[31] Erin Carson, James Demmel, Laura Grigori, Nicholas Knight, Penporn Koanantakool, Oded Schwartz, and Harsha Vardhan Simhadri. Write-avoiding algorithms. In IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016.

[32] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In CIDR, 2011.

[33] Sangyeun Cho and Hyunjin Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In IEEE/ACM International Symposium on Microarchitecture (MICRO), 2009.

[34] Rezaul Chowdhury, Pramod Ganapathi, Jesmin Jahan Tithi, Charles Bachmeier, Bradley C Kuszmaul, Charles E Leiserson, Armando Solar-Lezama, and Yuan Tang. Autogen: Automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs. In ACM Symposium on Principles and Practice of Parallel Programming (PPOPP), 2016.
[35] Rezaul A. Chowdhury. Cache-efficient algorithms and data structures: Theory and experimental evaluation. *PhD Thesis, UT Austin*, 2007.

[36] Rezaul A. Chowdhury, Hai-Son Le, and Vijaya Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 7(3):495–510, 2010.

[37] Rezaul A. Chowdhury and Vijaya Ramachandran. Cache-oblivious dynamic programming. In *ACM-SIAM symposium on Discrete algorithm (SODA)*, 2006.

[38] Rezaul A. Chowdhury and Vijaya Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *ACM Symposium on Parallelism in algorithms and architectures (SPAA)*, pages 207–216. ACM, 2008.

[39] Rezaul A. Chowdhury and Vijaya Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. *Theory of Computing Systems*, 47(4), 2010.

[40] Richard Cole and Vijaya Ramachandran. Resource oblivious sorting on multicores. In *International Colloquium on Automata, Languages, and Programming*. Springer, 2010.

[41] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009.

[42] Erik D Demaine. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 8(4), 2002.

[43] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *International Parallel & Distributed Processing Symposium (IPDPS)*, 2013.

[44] David Dinh, Harsha Vardhan Simhadri, and Yuan Tang. Extending the nested parallel model to the nested dataflow model with provably efficient schedulers. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[45] Xiangyu Dong, Norman P. Joustrup, and Yuan Xie. PCRAMsim: System-level performance, energy, and area modeling for phase-change RAM. In *ACM International Conference on Computer-Aided Design (ICCAD)*, 2009.

[46] Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, Hai H. Li, and Yiran Chen. Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In *ACM Design Automation Conference (DAC)*, 2008.

[47] David Eppstein and Zvi Galil. Parallel algorithmic techniques for combinatorial computation. *International Colloquium on Automata, Languages, and Programming (ICALP)*, 1989.

[48] David Eppstein, Michael T. Goodrich, Michael Mitzenmacher, and Pawel Pszona. Wear minimization for cuckoo hashing: How not to throw a lot of eggs into one basket. In *ACM Symposium on Experimental and Efficient Algorithms (SEA)*, 2014.

[49] Michael J Fischer and Albert R Meyer. Boolean matrix multiplication and transitive closure. In *IEEE Symposium on Switching and Automata Theory*, 1971.
[50] Robert W. Floyd. Algorithm 97: Shortest path. Commun. ACM, 5(6), June 1962.

[51] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In IEEE Symposium on Foundations of Computer Science (FOCS), 1999.

[52] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In ACM International Conference on Supercomputing, pages 361–366, 2005.

[53] ME Furman. Application of a method of fast multiplication of matrices to problem of finding graph transitive closure. Doklady Akademii Nauk SSSR, 194(3), 1970.

[54] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. ACM Computing Surveys, 37(2), 2005.

[55] Zvi Galil and Raffaele Giancarlo. Speeding up dynamic programming with applications to molecular biology. Theoretical Computer Science, 64(1), 1989.

[56] Zvi Galil and Kunsoo Park. Dynamic programming with convexity, concavity and sparsity. Theoretical Computer Science, 92(1), 1992.

[57] Zvi Galil and Kunsoo Park. Parallel algorithms for dynamic programming recurrences with more than O(1) dependency. Journal of Parallel and Distributed Computing, 21(2), 1994.

[58] Yan Gu, Yihan Sun, and Guy E. Blelloch. Algorithmic building blocks for asymmetric memories. In European Symposium on Algorithms (ESA), 2018.

[59] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. Communications of the ACM, 18(6):341–343, 1975.

[60] Daniel S Hirschberg and Lawrence L Larmore. The least weight subsequence problem. SIAM Journal on Computing, 16(4), 1987.

[61] Jia-Wei Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In Proc. ACM Symposium on Theory of Computing (STOC), 1981.

[62] HP, SanDisk partner on memristor, ReRAM technology. http://www.bit-tech.net/news/hardware/2015/10/09/hp-sandisk-rram-memristor, October 2015.

[63] Jingtong Hu, Qingfeng Zhuge, Chun Jason Xue, Wei-Che Tseng, Shouzhen Gu, and Edwin Sha. Scheduling to optimize cache utilization for non-volatile main memories. IEEE Transactions on Computers, 63(8), 2014.

[64] S-HS Huang, Hongfei Liu, and Venkatraman Viswanathan. Parallel dynamic programming. IEEE transactions on parallel and distributed systems, 5(3), 1994.

[65] Shou-Hsuan Stephen Huang, Hongfei Liu, and Venkatraman Viswanathan. A sublinear parallel algorithm for some dynamic programming problems. Theoretical Computer Science, 106(2), 1992.

[66] www.slideshare.net/IBMZRL/theseus-pss-nvmw2014, 2014.

[67] Intel and Micron produce breakthrough memory technology. http://newsroom.intel.com/community/intel-newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology, July 2015.
[68] Dror Irony, Sivan Toledo, and Alexandre Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64(9):1017–1026, 2004.

[69] Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016.

[70] Riko Jacob and Nodari Sitchinava. Lower bounds in the asymmetric external memory model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017.

[71] Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[72] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. Technical report, RAND PROJECT AIR FORCE SANTA MONICA CA, 1951.

[73] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

[74] Donald E Knuth and Michael F Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11), 1981.

[75] Penporn Koanantakool, Ariful Azad, Aydn Buluç, Dmitriy Morozov, Sang-Yun Oh, Leonid Oliker, and Katherine Yelick. Communication-avoiding parallel sparse-dense matrix-matrix multiplication. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[76] Marvin K"unnemann, Ramamohan Paturi, and Stefan Schneider. On the fine-grained complexity of one-dimensional dynamic programming. *arXiv preprint arXiv:1703.00941*, 2017.

[77] Gad M Landau and Uzi Vishkin. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *ACM symposium on Theory of computing (STOC)*, pages 220–230, 1986.

[78] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *ACM International Symposium on Computer Architecture (ISCA)*, 2009.

[79] Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Efficient parallelization using rank convergence in dynamic programming algorithms. *Communications of the ACM*, 59(10):85–92, 2016.

[80] Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Low-rank methods for parallelizing dynamic programming algorithms. *ACM Transactions on Parallel Computing (TOPC)*, 2(4):26, 2016.

[81] Jasmina Malicvevic, Subramanya Dulloor, Narayanan Sundaram, Nadathur Satish, Jeff Jackson, and Willy Zwaenepoel. Exploiting nvm in large-scale graph analytics. In *Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2015.

[82] Jagan S. Meena, Simon M. Sze, Umesh Chand, and Tseung-Yuan Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters*, 9, 2014.
[83] Ian Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2), 1971.

[84] Hyoungmin Park and Kyuseok Shim. FAST: Flash-aware external sorting for mobile database systems. *Journal of Systems and Software*, 82(8), 2009.

[85] Moinuddin K. Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. *Phase Change Memory: From Devices to Systems*. Morgan & Claypool, 2011.

[86] Wojciech Rytter. On efficient parallel computations for some dynamic programming problems. *Theoretical Computer Science*, 59(3), 1988.

[87] Edgar Solomonik, Aydin Buluc, and James Demmel. Minimizing communication in all-pairs shortest paths. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2013.

[88] Yuan Tang and Shiyi Wang. Brief announcement: Star (space-time adaptive and reductive) algorithms for dynamic programming recurrences with more than O(1) dependency. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017.

[89] Yuan Tang, Ronghui You, Haibin Kan, Jahan Jahan Tithi, Pramod Ganapathi, and Rezaul A Chowdhury. Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2015.

[90] Alexandre Tiskin. All-pairs shortest paths computation in the BSP model. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 178–189, 2001.

[91] Jesmin Jahan Tithi, Pramod Ganapathi, Aakrati Talati, Sonal Aggarwal, and Rezaul Chowdhury. High-performance energy-efficient recursive dynamic programming with matrix-multiplication-like flexible kernels. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015.

[92] Sivan Toledo. Locality of reference in lu decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4), 1997.

[93] Stratis D. Viglas. Adapting the B+-tree for asymmetric I/O. In *Advances in Databases and Information Systems (ADBIS)*, 2012.

[94] Stratis D. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5), 2014.

[95] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1), 1962.

[96] Michael S Waterman and Temple F Smith. Rna secondary structure: A complete mathematical analysis. *Mathematical Biosciences*, 42(3-4), 1978.

[97] Virginia Vassilevska Williams. On some fine-grained questions in algorithms and complexity. In *International Congress of Mathematicians (ICM)*, 2018.

[98] David Womble, David Greenberg, Stephen Wheat, and Rolf Riesen. Beyond core: Making parallel computer i/o practical. In *Proceedings of the 1993 DAGS/PC Symposium*, 1993.

[99] Cong Xu, Xiangyu Dong, Norman P. Jouppi, and Yuan Xie. Design implications of memristor-based RRAM cross-point structures. In *IEEE Design, Automation and Test in Europe (DATE)*, 2011.
A Strassen Algorithm

Strassen algorithm computes matrix multiplication on a ring. Given two input matrices $A$ and $B$ and the output matrix $C = AB$, the algorithm partitions $A$, $B$ and $C$ into quadrants, applies seven recursive matrix multiplications on the sums or the differences of the quadrants, and each quadrant of $C$ can be calculated by summing a subset of the seven intermediate matrices. This can be done in $O(n\log_2 7)$ work, $O(n\log_2 7 / M \log_4 7 - 1)B$ cache complexity and $O(\log n)$ depth.

Technically the computation structure of Strassen is not a k-d grid, but we can apply a similar idea in Section 5 to reduce asymmetric cache complexity. We still use $r$ as the balancing factor between reads and writes (set to be $\omega^{2/3}$ in classic matrix multiplication). Given square input matrices, the algorithm also partition the output into $r$-by-$r$ submatrices, and then run the 8-way divide-and-conquer approach to compute the matrix multiplication. This gives the following recurrences on work ($T'$), reads, writes and depth based on the output size $n$:

$$T'(n) = 7T'(n/2) + O(n^2)$$
$$R'(n) = 7R'(n/2) + O(n^2/B)$$
$$W'(n) = 7W'(n/2) + O(\omega n^2/B)$$
$$D'(n) = D(n/2) + O(\log n)$$

with the base cases $T'(1) = r$, $R'(\sqrt{rM}) = M/B$, $W'(\sqrt{M}) = \omega M/B$, and $D'(1) = 1$. They solve to

$$R(n) = r^2 R'(n/r) = O(n^{\log_4 7 r^{2-\log_4 7} M^{1-\log_4 7} / B})$$
$$W(n) = r^2 W'(n/r) = O(\omega n^{\log_4 7 r^{2-\log_4 7} M^{1-\log_4 7} / B})$$
$$D(n) = O(\log^2 n)$$

The case when $r = \omega^{\log_4 7 - 4}$ gives the minimized cache complexity of

$$Q(n) = O(n^{\log_4 7 \omega^{\log_7 16 - 1} M^{1-\log_4 7} / B}) \approx O(n^{2.8 \omega^{0.42} / BM^{0.4}})$$

an $O(\omega^{0.58})$ improvement over the non-write-efficient version. In this setting the work is $O(n^{\log_4 7 \omega^{\log_7 64 - 2}})$, a factor of $O(\omega^{0.14})$ or $O(\omega^{1/7})$ extra work.
Algorithm 2: KLEENE(A)

**Input:** Distance matrix A initialized based on the input graph G = (V, E)
**Output:** Computed Distance matrix A

1. A₀₀ ← KLEENE(A₀₀)
2. A₀₁ ← A₀₁ + A₀₀A₀₁
3. A₁₀ ← A₁₀ + A₁₀A₀₀
4. A₁₁ ← A₁₁ + A₁₀A₀₁
5. A₁₁ ← KLEENE(A₁₁)
6. A₀₁ ← A₀₁ + A₀₁A₁₁
7. A₁₀ ← A₁₀ + A₁₁A₁₀
8. A₀₀ ← A₀₀ + A₁₀A₀₁
9. return A

B  All-Pair Shortest-Paths (APSP)

An all-pair shortest-paths (APSP) problem takes a (usually directed) graph G = (V, E) (with no negative cycles) as input. Here we discuss the Kleene’s algorithm (first mentioned in [49, 53, 72, 83], discussed in full details in [5]). Kleene’s algorithm has the same computational DAG as Floyd-Washall algorithm [50, 95], but it is described in a divide-and-conquer approach, which is already I/O-efficient, cache-oblivious and highly parallelized.

The pseudocode of Kleene’s algorithm is provided in Algorithm 2. The matrix A is partitioned into 4 submatrices indexed as \[
\begin{bmatrix}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{bmatrix}
\]
The matrix multiplication is defined in a closed semi-ring with (+, min).

Kleene’s algorithm is a divide-and-conquer algorithm to compute APSP. Its high-level idea is to first compute the APSP between the first half of the vertices only using the paths between these vertices. Then by applying some matrix multiplication we update the shortest-paths between the second half of the vertices using the computed distances from the first half. We then apply another recursive subtask on the second half vertices. The computed distances are finalized, and we use them to again update the shortest-paths from the first-half vertices.

The asymmetric cache complexity Q(n) of this algorithm follows the recursion of:

\[
Q(n) = 2Q(n/2) + 6Q_{3C}(n/2)
\]
\[
D(n) = 2D(n/2) + 2D_{3C}(n/2)
\]

Considering this cost, the recursion is root-dominated, which indicates that computing all-pair shortest paths of a graph has the same upper bound on cache complexity as matrix multiplication.

C  Gaussian Elimination

Gaussian elimination (without pivoting) is used in solving of systems of linear equations and computing LU decomposition of symmetric positive-definite or diagonally dominant real matrices. Given a linear system \(AX = b\), the algorithm proceeds in two phases. The first phase modifies A into an upper triangular matrix.
updates $B$ accordingly), which is discussed in this section. The second phase solves the values of the variables using back substitution, which is shown in Section D.

The process of Gaussian elimination can be viewed as a three nested-loops and computing the value of $A_{i,j}$ requires $A_{i,k}$, $A_{k,j}$, and $A_{k,k}$ for all $1 \leq k < i$. If required, the corresponding value of the LU decomposition matrix can be computed simultaneously. The underlying idea of divide-and-conquer approach is almost identical to Kleene’s algorithm, which partitions $A$ into four quadrants $[A_{00} \ A_{01} \ A_{10} \ A_{11}]$. The algorithm:

1. recursively computes $A_{00}$;
2. updates $A_{10}$ and $A_{01}$ using $A_{00}$;
3. updates $A_{11}$ using $A_{10}$ and $A_{01}$; and
4. recursively computes $A_{11}$.

Note that each inter-quadrant update in step (2) and (3) is a 3d grid, which gives the following recurrence:

$$Q(n) = 4Q(n/2) + 4Q_3C(n/2)$$
$$D(n) = 2D(n/2) + 3D_3C(n/2)$$

D Triangular System Solver

A Triangular System Solver computes the back substitution step in solving the linear system. Here we assume that it takes as input a lower triangular $n \times n$ matrix $T$ (can be computed using the algorithm discussed in Section C) and a square matrix $B$ and outputs a square matrix $X$ such that $TX = B$. A triangular system can be recursively decomposed as:

$$[B_{00} \ B_{01}] = [T_{00} \ 0] [X_{00} \ X_{01}]$$
$$= \begin{bmatrix} T_{00}X_{00} & T_{00}X_{01} \\ T_{10}X_{00} + T_{11}X_{10} & T_{10}X_{01} + T_{11}X_{11} \end{bmatrix}$$

such that four equally sized subquadrants $X_{00}$, $X_{01}$, $X_{10}$, and $X_{11}$ can be solved recursively. In terms of parallelism, the two subtasks of $X_{00}$ and $X_{01}$ are independent, and need to be solved prior to the other independent subtasks $X_{10}$, and $X_{11}$.

The asymmetric cache complexity $Q(n)$ of this algorithm follows the recursion of:

$$Q(n) = 4Q(n/2) + 4Q_3C(n/2)$$
$$D(n) = 2D(n/2) + 3D_3C(n/2)$$

E Discussions to Carson et al. [31]

Carson et al. also discussed algorithms using less writes in the paper [31]. They concluded that many cache-oblivious algorithms like matrix multiplication could not be write-avoiding. Their definition of write-avoiding is different from our write-efficiency, and it requires the algorithm to reduce writes without asymptotically increasing reads. Hence, their negative conclusion does not contradict the result in this paper.

We now show an example that optimal number of reads leads to worse overall asymmetric cache complexity. Let’s say a write is $n$ times more expensive than a read. A smart cache-oblivious matrix multiplication algorithm should apply $n^2$ inner products, and the asymmetric cache complexity is $O(n^3/B)$: $O(n/B)$ reads and $O(1/B)$ writes per inner product. However, the algorithms using optimal number of reads requires $O(n^3/B\sqrt{M})$ reads and writes, so the overall cache complexity is $O(n \cdot n^3/B\sqrt{M})$. The algorithm requiring more reads is a factor of $O(n/\sqrt{M})$ better on the asymmetric cache complexity. Notice
that we always assume $n = \omega(\sqrt{M})$ since otherwise the whole computation is trivially in the cache and has no cost. As a result, an algorithm with a good asymmetric cache complexity does not always need to be write-avoiding.

The algorithms on asymmetric memory in this paper all require extra reads, but can greatly reduce the overall asymmetric cache complexity compared to the previous cache-oblivious algorithms. The goal of this paper is to find the optimal cache-oblivious algorithms for any given write-read asymmetry $\omega$. 