Improved Parallel Cache-Oblivious Algorithms for Dynamic Programming [Extend Abstract]*

Guy E. Blelloch † Yan Gu‡

Abstract

Emerging non-volatile main memory (NVRAM) technologies provide byte-addressability, low idle power, and improved memory-density, and are likely to be a key component in the future memory hierarchy. However, a critical challenge in achieving high performance is in accounting for the asymmetry that NVRAM writes can be significantly more expensive than NVRAM reads.

In this paper, we consider a large class of cache-oblivious algorithms for dynamic programming (DP) and try to reduce the writes in the asymmetric setting while maintaining high parallelism. To achieve that, our key approach is to show the correspondence between these problems and an abstraction for their computation, which is referred to as the k-d grids. Then by showing lower bound and new algorithms for computing k-d grids, we show a list of improved cache-oblivious algorithms of many DP recurrences in the asymmetric setting, both sequentially and in parallel.

Surprisingly, even without considering the read-write asymmetry (i.e., setting the write cost to be the same as the read cost in the algorithms), the new algorithms improve the existing cache complexity of many problems. We believe the reason is that the extra level of abstraction of k-d grids helps us to better understand the complexity and difficulties of these problems. We believe that the novelty of our framework is of theoretical interest and leads to many new questions for future work.

1 Introduction

The *ideal-cache model* [36] 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 [7, 24, 31], and in many cases they asymptotically match the best cache complexity for cache-aware algorithms. Regarding parallelism, Blelloch et al. [18] suggest that analyzing the depth and sequential cache complexity of an algorithm is sufficient for deriving upper bounds on parallel cache complexity.

Recently, emerging non-volatile main memory (NVRAM) technologies, such as Intel's Optane DC Persistent Memory, are readily available on the market, and provide byte-addressability, low idle power, and improved memory-density. Due to these advantages, NVRAMs are likely to become a key component in the memory hierarchy. However, a significant programming challenge arises due to an underlying asymmetry between reads and writes—reads are much cheaper than writes in terms of both latency, bandwidth, and throughput. This property requires researchers to rethink the design of algorithms and software, and optimize the existing ones accordingly to reduce the writes. Such algorithms are referred to as ***write-efficient algorithms*** [40].

Many cache-oblivious algorithms are affected by this challenge. Taking matrix multiplication as an example, the cache-aware tiling-based algorithm [4] uses $\Theta(n^3/B\sqrt{M})$ cache-line reads and $\Theta(n^2/B)$ cache-line writes for square matrices with size n-by-n. The cache-oblivious algorithm [36], despite the advantages described above, uses $\Theta(n^3/B\sqrt{M})$ cache-line reads and writes. When considering the more expensive writes, the cache-oblivious algorithm is no longer asymptotically optimal. Can we asymptotically improve the cache complexity of these cache-oblivious algorithms? Can they match the best counterpart without considering cache-obliviousness? These remain to be *open* problems at the very beginning of the study of write-efficiency of algorithms [14, 25].

In this paper, we provide the answers to these questions for a large class of cache-oblivious algorithms that have computation structures similar to 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

*Full version of this paper is available at arXiv:1809.09330

†Carnegie Mellon University

‡University of California, Riverside

[1] In this paper, we refer to it as *symmetric cache complexity* to distinguish from the case when reads and writes have different costs.

Dimension	Problems	Cache Complexity	
		Symmetric	Asymmetric
$k = 2$	LWS/GAP*/RNA/knapsack recurrences	$\Theta\left(\frac{C}{BM}\right)$	$\Theta\left(\frac{\omega^{1/2}C}{BM}\right)$
$k = 3$	Combinatorial matrix multiplication, Kleene's algorithm (APSP), Parenthesis recurrence	$\Theta\left(\frac{C}{B\sqrt{M}}\right)$	$\Theta\left(\frac{\omega^{1/3}C}{B\sqrt{M}}\right)$

Table 1: Cache complexity of the 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 7.2.

dynamic programming (e.g., the LWS/GAP/RNA/Parenthesis problems) and linear algebra (e.g., matrix multiplication, Gaussian elimination, LU decomposition) [18, 26–28, 36, 47, 55, 58].

Since we try to cover many problems and algorithms, in this paper we propose a level of abstraction of the computation in these cache-oblivious algorithms, which is referred to as the k-d grid computation structures (or k-d grids, for short). A more formal definition is given Section 3. This structure and similar ones were first used by Hong and Kung [43] (implicitly) in their seminal paper in 1981, and then by a subsequence of later work (e.g., [2, 8, 9, 28, 46]), mostly on analyzing the lower bounds of matrix multiplication and linear algebra problems in a variety of settings. By allowing the output to be the same as the input, in this paper, we show the relationship of the k-d grids and many other dynamic programming problems, and new results (algorithms and lower bounds) related to the k-d grids.

The first intellectual contribution of this paper is to draw the connection between many dynamic programming (DP) problems and k-d grids. Previous DP algorithms are usually designed and analyzed based on the number of nested loops, or the number of 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, and such a relationship will be defined formally later in Section 3. A few examples (e.g., matrix multiplication, tensor multiplication, RNA and GAP recurrences) are also provided in Section 3 to illustrate the idea. This property is reflected by the nature of the k-d grids, and the correspondence between the problems and the k-d grids is introduced in Section 7 and 8. We note that such a relationship for an DP algorithm can be much more complicated than the linear algebra algorithms, and in many cases the computation of one algorithm consists of many (e.g., $O(n)$) k-d grids.

The second intellectual contribution of this paper is a list of new results for k-d grids. We first discuss the lower bounds to compute such k-d grids considering the asymmetric cost between writes and reads (in Section 4). Based on the analysis of the lower bounds, we then propose algorithms with the matching upper bound to compute a k-d grid (in Section 5). Finally, we also show how to parallelize the algorithm in Section 6. We note that the approach for parallelism is independent of the asymmetric read-write cost so that the parallel algorithms can be applied to both symmetric and asymmetric algorithms.

In summary, we have shown the correspondence between the problems and the k-d grids, new lower and upper cache complexity bounds for computing the k-d grids in the asymmetric setting, and parallel algorithms in both symmetric and asymmetric settings. Putting all pieces together, we can show lower and upper cache complexity bounds of the original problems in both the symmetric and asymmetric settings, as well as spans (length of dependence) for the algorithms. The cache complexity bounds are summarized in Table 1, and the results of the asymmetric setting answer the open problem in [14]. The span bound is analyzed for each specific problem and given in Section 7 and the full version of this paper.

Surprisingly, even without considering the read-write asymmetry (i.e., setting the write cost to be the same as the read cost in the algorithms), the new algorithms proposed in this paper improve the existing cache complexity of several problems. We believe the reason is that the extra level of abstraction of k-d grids helps us to better understand the complexity and difficulties in these problems. Since k-d grids are used to lower bounds, they decouple the computation structures from the complicated data dependencies, which exposes some techniques to improve the bounds that were previously obscured. Also, k-d grids reveal the similarities and differences between these problems, which allows the optimizations in some algorithms to apply to other problems.

In summary, 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 new theoretical results in this paper include:

- We provide write-efficient cache-oblivious algorithms (i.e., in the asymmetric setting) for all problems we discussed in this paper, including matrix multiplication, all-pair shortest-paths, and a number of dynamic programming recurrences. If a write costs ω times more than a read (the formal computational model shown

in Section 2), 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 best previous results [16]. We also show that this improvement is optimal under certain assumptions (the CBCO paradigm, defined in Section 4.2).
- We show algorithms with improved symmetric cache complexity on many problems, including the GAP recurrence, protein accordion folding, and the RNA recurrence. We show that the previous cache complexity 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[2]. 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 the first race-free linear-span cache-oblivious algorithms solving all-pair shortest-paths, LWS recurrences, and protein accordion folding. Some previous algorithms [32, 56] have linear span, but they are not race-free and rely on a stronger model (discussed in Section 2). Our approaches are under the standard nested-parallel model, race-free, and arguably simpler. Our algorithms are not in-place, but we discuss in Section 6.1 about the extra storage needed.

We believe that the analysis framework is concise. In this single paper (and full version in [19]), we discuss the lower bounds and parallel algorithms on a dozen or so computations and DP recurrences, which can be further applied to dozens of real-world problems[3]. The results are shown in both settings with or without considering the asymmetric cost 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.[4] 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. We usually make the tall-cache assumption that $M = \Omega(B^2)$, which holds for real-world hardware and is used in the analysis in Section 7.3.

The nested-parallel model and work-span 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 span—*work* W to be the total number of operations in this DAG and span (depth) D equals to the longest path in the DAG. The randomized work-stealing scheduler can execute such a computation on the PRAM model with p processors in $W/p + O(D)$ time with high probability [22]. All algorithms in this paper are *race-free* [34]—no logically parallel parts of an algorithm access the same memory location and one of the accesses is a write. 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. [18] suggest that analyzing the span 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 [1]. For a p-processor shared-memory machine with a shared cache of size $M+pBD$ using a parallel-depth-first (PDF) scheduler, $Q_p \le Q_1$ [13]. We can extend these bounds to multi-level hierarchies of private or shared caches, respectively [18].

Parallel and cache-oblivious algorithms for dynamic programming. Dynamic Programming (DP) is an optimization strategy that decomposes a problem into subproblems with optimal substructure. It has been studied for over sixty years [5, 10, 30]. For the problems that we consider

[2] The improvement is $O(\sqrt{M})$ from an asymptotic perspective (i.e., n approaching infinity). For smaller range of n that $O(\sqrt{M}) \le n \le O(M)$, 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 $n < O(\sqrt{M})$.)

[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.

[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.

in this paper, the parallel DP algorithms were already discussed by a rich literature in the eighties and nighties (e.g., [33, 37, 39, 44, 45, 54]). Later work not only considers parallelism, but also optimizes symmetric cache complexity (e.g., [18, 26–28, 32, 36, 47, 55, 56]).

Problem definitions. Since we are showing many optimal cache-oblivious algorithms, we assume the operations in the computations to be atomic using unit cost and unable to be decomposed or batched (e.g., using integer tricks).

Algorithms with asymmetric read and write costs. Intel has already announced the new product of the Optane DC Persistent Memory, which can be bought from many retailers. The new memories sit on the main memory bus and are byte-addressable. As opposed to DRAMs, the new memories are persistent, so we refer to them as non-volatile RAMs (NVRAMs). In addition, compared to DRAMs, NVRAMs require significantly lower energy, and have good read latencies and higher density. Due to these advantages, NVRAMs are likely to be a key component in the memory hierarchy. However, a new property of NVRAMs is the asymmetric read and write cost—write operations are more expensive than reads regarding energy, bandwidth, and latency (benchmarking results in [59]). This property requires researchers to rethink the design of algorithms and software, and motivates the need for *write-efficient algorithms* [40] that reduce the number of writes compared to existing algorithms.

Blelloch et al. [11, 14, 15] 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 ω, 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 ω, plus the number of read transfers. This model captures different system considerations (latency, bandwidth, or energy) by simply plugging in a different value of ω, which allows algorithms to be analyzed theoretically and practically. Similar scheduling results (upper bounds) on parallel running time and cache complexity are discussed in [11, 15] based on work W, span D and asymmetric cache complexity Q of an algorithm. Based on this idea, many interesting algorithms and lower bounds are designed and analyzed by recent work [11, 12, 14, 15, 17, 20, 21, 41, 48].

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—ω times the output size. For simplicity, this term is ignored in the asymptotic analysis.

3 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 $I_1, \cdots, I_{k-1}$ and generates one output array O. The output array can be the same as one of the input arrays. Each array has dimension $k-1$ and is the projection of the grid removing one of the dimensions. 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 using an associative operation $\oplus$. The $k-1$ inputs of this cell are the projections of this cell removing each (but not the last) of the dimensions, and the output is the projection removing the last dimension. They are referred to as the input and output ***entries*** of this cell. Figure 1 illustrates such a computation in 2 and 3 dimensions. This structure (mostly the special case for 3d as defined below) is used implicitly and explicitly by Hong and Kung [43] and subsequence work (e.g., [2, 8, 9, 28, 46]). In this paper, we will use it as a building block to prove lower bounds and design new algorithms for dynamic programming problems. When showing the cache complexity, we assume the input and output entries must be in the cache when computing each cell.

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.

A formal definition of a square 3d grid of size n is as follows:

$$O_{i,j} = \sum_k g\big((I_1)_{i,k}(I_2)_{k,j}, i, j, k\big)$$

where $1 \le i, j, k \le n$. $g(\cdot)$ computes a value based on the two inputs $(I_1)_{i,k}$ and $(I_2)_{k,j}$ the indices, and some constant amount of data that is associated with the indices. We assume that computing $g(\cdot)$ takes unit cost. Each application of $g(\cdot)$ corresponds to a cell, and $(I_1)_{i,k}$, $(I_2)_{k,j}$ and $O_{i,j}$ are entries associated with this cell. The sum $\sum$ is based on an associative operator $\oplus$. Similarly, the definition for the 2d case is:

$$O_i = \sum_j g\big(I_j, i, j\big)$$

and we can extend it to non-square cases and for $k > 3$ accordingly.

For the DP recurrences the output array O is always the same as one of the input array(s) I. In these algorithms, some of the cells are empty to avoid cyclic dependencies. For example, in a 2d grid, we may want to restrict $1 \le j < i$. In these cases, a constant fraction of the grid cells are empty. We call such a grid an α-***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

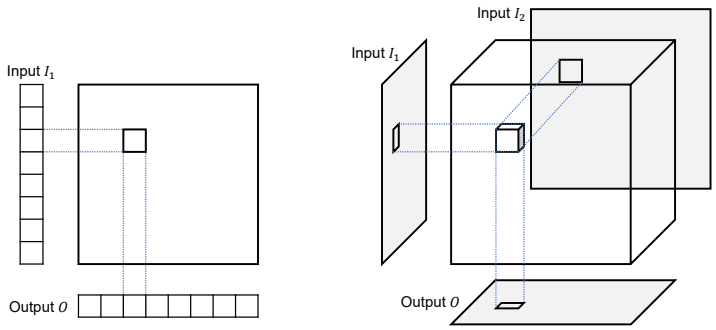

Figure 1: An illustration of a 2d and a 3d grid. The left figure shows the 2d case where the input I_1 and output O are 1d arrays, and each computation cell $g(\cdot)$ requires exactly one entry in I_1 as input, and update one entry in O. For the 3d case on the right, the inputs and output are 2d arrays, and each computation cell $g(\cdot)$ 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.

for the α-full case, since the constant α affects the analysis of neither the lower bounds nor upper bounds.

We now show some examples that can be matched to k-d grids. Multiplying two matrices of size n-by-n on a semiring $(\cdot, +)$ (i.e., $O_{i,j} = \sum_k (I_1)_{i,k}(I_2)_{k,j}$) exactly matches a 3d square grid. A corresponding 2d case is when computing a matrix-vector multiplication $O_i = \sum_j I_j \cdot f(i,j)$ where $f(i,j)$ does not need to be stored. Such applications are commonly seen in dynamic programming algorithms, and we refer readers to Section 7 first if they are not familiar with dynamic programming. For example, the widely used LWS recurrence (Section 7.1) that computes $D_j = \min_{0 \le i < j}\{D_i + w(i,j)\}$ is a 2d grid, and the associative operator $\oplus$ is min. In this case the input is the same array as the output. These are the simple cases, so even without using the k-d grid, the algorithms for them in the symmetric setting are already studied in [27, 36].

However, not all DP recurrences can be viewed as k-d grids straightforwardly. As shown above, the key aspect of deciding the dimension of a computation is the number of inputs that each basic cell $g(\cdot)$ requires. For example, when multiplying two dense tensors, although each tensor may have multiple dimensions, each multiply operation is only based on two entries and can be written in the previous 3d form, so the computation is a 3d grid. Another example is the RNA recurrence that computes a 2D array

$$D_{i,j} = \min_{0 \le p < i, 0 \le q < j}\{D_{p,q} + w(p,q,i,j)\} .$$

Assuming $w(p,q,i,j)$ can be queried on-the-fly, the computation is the simplest 2d grid. Despite that the DP table has size $O(n^2)$ and $O(n^4)$ updates in total, the computation is no harder than the simplest LWS recurrence mentioned in the previous paragraph. Similarly, in the GAP recurrence in Section 7.2, each element in the DP table is computed using many other elements similar to matrix multiplication. However, each update only requires the value of one input element and can be represented by a set of 2d grids, unlike matrix multiplication that is a 3d grid and uses the values of two input elements in each update. The exact correspondence between the k-d grid and the DP recurrences are usually more sophisticated than their classic applications in linear algebra problems, as shown in Section 7, 8 and appendices in the full paper. The cache-oblivious 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 [23]).

4 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 [43] to an arbitrary dimension, and allowing the output array to be the same as an input array. Then in Section 4.2 we discuss the asymmetric cache complexity when writes are more expensive than reads, which is more interesting and has a more involved analyses.

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

THEOREM 4.1. ([43]) *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. In a k-d grid computation structure with size n there are n^k cells. Let's sequentialize these cells in a list and consider each block of cells that considers $S = 2^k M^{k/(k-1)}$ consecutive cells in the list. The number of input entries required of each block is the projection of all cells in this block along one of the first $k-1$ dimensions (see Figure 1), and this is similar for the output. Loomis-Whitney inequality [9, 53] indicates that the overall number of input and output entries is minimized when the cells are in a square k-d cuboid, giving a total of $S^{(k-1)/k} = \left(2M^{1/(k-1)}\right)^{k-1} \ge 2M$ input and output entries. Since only a total of M entries can be held in the cache at the beginning of the computation of this block, the number of cache-line transfer for the

input/output during the computation for such a block is $\Omega(M/B)$. Since there are $n^k/S = \Theta(n^k M^{-k/(k-1)})$ such blocks, the cache complexity of the entire computation is $\Omega(M/B) \cdot n^k/S = \Omega(n^k/(M^{1/(k-1)}B))$. □

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 [36].

4.2 Asymmetric Cache Complexity We now consider the asymmetric cache complexity of a k-d grid computation structure assuming writes are more expensive. 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 [36].

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., [6]), 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 [14, 36]), a larger fan-out in the recursion is 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 and fit in the cache with maximal possible subproblem size. All problems we discuss in this paper are in this category, so we analyze under the following constraints. More discussion about this constraint is in the full version of this paper.

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 recursive subtasks until the base cases, and the partition of each task is decided by the ratio of the ranges in all dimensions of the (sub)problem and independent of the cache parameters (M and B).*

Notice that ω is a parameter of the main memory, instead of a cache parameter, so the algorithms can be aware of it. One can define resource-obliviousness [29] so that the value of ω 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 constant branching and the partition based on the ratio of the ranges in all dimensions restrict the computation pattern and lead to the "scale-free" property of the cache-oblivious algorithms—the structure or the "shape" of each subproblem in the recursive levels is similar, and only the size varies. The proof references this property when it is used.

[5] It can exponentially depend on k since we assume k is a constant.

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

Proof. We prove the lower bound using the same approach as in Section 4.1—putting all operations (cells) executed by the algorithm in a list and analyzing blocks of S cells. 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 of adding the calculated value back into the output array, which only makes the problem easier. Again when applying the computation of each cell, the k input and output entries have to be in the cache.

For a block of cells with size S, the cache needs to hold the entries in $I_1, \cdots, I_{k-1}$ and O corresponding to the cells in this block at least once during the computation. Similar to the symmetric setting discussed above, 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 entries is more expensive since they have to be written back to the asymmetric memory. 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 $r \geq 1$ 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 for Theorem 4.1, for a block of size S, the read transfers required by the cache is $\Omega\left(\frac{n^k}{SB} \cdot \max\{a^{k-1}r - M, 0\}\right)$, where n^k/S is the number of such blocks, and $\max\{a^{k-1}r - M, 0\}/B$ 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 $\Omega\left(\frac{\omega n^k}{SB} \cdot \max\{a^{k-1} - M, 0\}\right)$. In total, the cost is:

$$\begin{aligned} Q &= \Omega\left(\frac{n^k}{SB} \cdot \left(\max\{a^{k-1}r - M, 0\} + \omega \max\{a^{k-1} - M, 0\}\right)\right) \\ &= \Omega\left(\frac{n^k}{SB}\left(\max\{S^{(k-1)/k} r^{1/k} - M, 0\} \right.\right. \\ &\qquad \left.\left. + \omega \max\left\{\frac{S^{(k-1)/k}}{r^{(k-1)/k}} - M, 0\right\}\right)\right) \end{aligned}$$

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

The cost decreases with increasing 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,

$$\begin{aligned} Q &= \Omega\left(\frac{n^k}{S_1 B} S_1^{(k-1)/k} r^{1/k} + \right. \\ &\qquad \left. \frac{n^k}{S_2 B}\left(S_2^{(k-1)/k} r^{1/k} + \frac{\omega S_2^{(k-1)/k}}{r^{(k-1)/k}}\right)\right) \\ &= \Omega\left(\frac{n^k}{S_1^{1/k} B} r^{1/k} + \frac{n^k}{S_2^{1/k} B}\left(r^{1/k} + \frac{\omega}{r^{(k-1)/k}}\right)\right) \\ &= \Omega\left(\frac{n^k}{M^{1/k} B}\left(r^{1/k} + \frac{\omega}{r}\right)\right) \end{aligned}$$

Setting $r = \omega^{(k-1)/k}$ minimizes $\frac{n^k}{M^{1/k}B}\left(r^{1/k} + \frac{\omega}{r}\right)$. In this case, the lower bound of the asymmetric cache complexity Q is $\Omega\left(\frac{n^k \omega^{1/k}}{M^{1/(k-1)}B}\right)$, and this leads to the theorem. □

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 is shown in [36]) is optimal. In the asymmetric setting, the proof of Theorem 4.2 indicates that the classic algorithm is not optimal and off by a factor of $\omega^{(k-1)/k}$. This gap is captured by the balancing factor r in the proof, which 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 classic approaches (e.g., [36]) 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 (line 6–8), 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 fewer writes in total and an improvement in the cache efficiency.

For simplicity, we show the asymmetric cache complexity for square grids (i.e., $n_1 = \cdots = n_k$) and $n = \Omega(\omega^{(k-1)/k} M)$, and the general case can be analyzed similar to [36].

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 ω

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 $i \leftarrow \arg\max_{1 \le i \le k}\{n_i x_i\}$ where $x_k = \omega^{-(k-1)/k}$ and $x_j = 1$ for $1 \le j < k$
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 $\{I_{1,a}, I_{1,b}\}, \cdots, \{I_{k-1,a}, I_{k-1,b}\}, \{O_a, O_b\}$ 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 in Algorithm 1. In the sequential execution of Algorithm 1, each recursive function call 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 $\omega^{(k-1)^2/k}$ subproblems (referred 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 until the base case is reached.

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

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

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

where $W'(n)$ is the number of writes of 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 are therefore:

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

and

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

The difference from the write cost is in the base case since the input fits into the cache sooner when $n = M^{1/(k-1)}/\omega^{1/k}$. Namely, $R'(M^{1/(k-1)}/\omega^{1/k}) = O(M/B)$. By solving the recurrences, we have $R'(n/\omega^{(k-1)/k}) = O\left(\frac{n^k\omega^{2-k}}{M^{1/(k-1)}B}\right)$ and $R(n) = O\left(\frac{n^k\omega^{1/k}}{M^{1/(k-1)}B}\right)$.

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 $\Theta(\omega n^{k-1}/B)$ for storing the output O on the memory.

6 Parallelism

We now show the parallelism in computing the k-d grids. The parallel versions of the cache-oblivious algorithms only have polylogarithmic span, indicating that they are highly parallelized.

6.1 The Symmetric Case We first discuss how to parallelize the classic algorithm on symmetric memory. For a square grid, the algorithm partitions 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, so we can fully parallelize these levels with no additional cost. 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 [36]), the span 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 modifying the same elements in the output array O, our algorithm works as follows 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 using different output arrays that are just allocated (no concurrency to the other subtask).
3. When both subtasks finish, the computed values are merged (added) back in parallel, with work proportional to the output size and $O(\log n)$ span.
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 cases ($k > 2$), it requires 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. The parallel algorithm allocates memory only when partitioning the output (k-th) dimension. In this case, it allocates and computes two subtasks of size $n/2$ where n is the size of the output dimension. This leads to the following recurrence:

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

The recurrence solves to $S(n) = O(n^{k-1})$ when $k > 2$ since the recurrence is root-dominated. When $k = 2$, we can apply the version that only allocates temporary space for one subtask, which decreases the constant before $S(n/2)$ to 1, and yields $S(n) = O(n)$. 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 6.1. *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 [36] ($k = 3$ in the paper) to show the following lemma:

LEMMA 6.2. *The sequential symmetric cache complexity of the parallel cache-oblivious algorithm to compute a k-d grid of size n is $O(n^k/M^{1/(k-1)}B)$.*

Assuming that we can allocate a chunk of memory in constant time, the span of this approach is simply $O(\log^2 n)$—$O(\log n)$ levels of recursion, each with $O(\log n)$ span for the additions [18].

We have shown the parallel span and symmetric cache complexity. By applying the scheduling theorem in Section 2, we have the following result for parallel symmetric cache complexity.

Corollary 6.2. *The k-d grid of size n can be computed with the parallel symmetric cache complexity of $O(n^k/M^{1/(k-1)}B + pM\log^2 n)$ with private caches, or $O(n^k/M^{1/(k-1)}B)$ with a share cache of size $M + pB\log^2 n$.*

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

Lemma 6.3. *The overall space requirement of the parallel algorithm to compute the k-d grid is $O(p^{1/k}n^{k-1})$.*

Proof. We analyze the total space allocated for all processors. Lemma 6.1 indicates that if the root of the computation on one processor has the output array of size $(n')^{k-1}$, then the space requirement for this task is $O((n')^{k-1})$. There are in total p processors. 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 $q = \log_{2^k} p$. This case maximizes the overall space requirement for p processors, which is:

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

This shows the stated bound. ∎

Combining all results gives the following theorem:

Theorem 6.1. *There exists a cache-oblivious algorithm to compute a k-d grid of size n that requires $\Theta(n^k)$ work, $\Theta\left(\frac{n^k}{M^{1/(k-1)}B}\right)$ symmetric cache complexity, $O(\log^2 n)$ span, 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. Regarding parallelism, the changes in Algorithm 1 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 span of the parallel variant of Algorithm 1 is also $O(\log^2 n)$. The extra space upper bound is actually reduced, because the asymmetric algorithm has a higher priority in partitioning the input dimensions that does not requires allocation temporary space.

Lemma 6.4. *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 $q = O(\omega^{(k-1)^2/k})$ 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 6.3, 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. ∎

Lemma 6.4 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 [11] requires 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 [11].

Theorem 6.2. *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 $O(\log^2 n)$ span to compute a k-d grid of size n.*

7 Dynamic Programming Recurrences

In this section, we discuss a number of new results on dynamic programming (DP). To show 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 each update in the recurrences takes unit cost, just like the k-d grid in Section 3.

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 4 and 5, as well as the analysis of parallelism in Section 6, 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 [26–28, 36, 58], but the others are new. Also, the asymmetric versions and most parallel versions are new.

Symmetric cache complexity. We show improved algorithms for a number of problems when the number of entries per cell differs from the dimension of input/output arrays.

Such algorithms are for the GAP recurrence, protein accordion folding, and the 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 7.2 and 7.3. For the RNA recurrence, we show an optimal cache complexity of $\Theta(n^4/BM)$ in Theorem 7.2, which improves the best existing result by $O(M^{3/4})$.

Asymmetric cache complexity. By applying the asymmetric version for the k-d grid computation discussed in Section 5, we show a uniform approach to provide write-efficient algorithms for all DP recurrences in this section. We also show the optimality of all these algorithms regarding asymmetric cache complexity, except for the one for the GAP recurrence.

Parallelism. The parallelism of these algorithms is provided by the parallel algorithms discussed in Section 6. Polylogarithmic span can be achieved in computing the 2-knapsack recurrence, and linear span in LWS recurrence and protein accordion folding. The linear span for LWS can be achieved by previous work [32, 56], but they are not in the nested-parallel model. Meanwhile, our algorithms are arguably simpler.

7.1 LWS Recurrence We start with the simple example of the LWS recurrence, where optimal sequential upper bound in the symmetric setting is known [27]. We show new results for lower bounds, write-efficient cache-oblivious algorithms, and new span bound.

The LWS (least-weighted subsequence) recurrence [42] is one of the most commonly-used DP recurrences in practice. Given a real-valued function $w(i, j)$ for integers $0 \le i < j \le n$ and D_0, for $1 \le j \le n$,

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

This recurrence is widely used as a textbook algorithm to compute optimal 1D clustering [50], line breaking [51], longest increasing sequence, minimum height B-tree, and many other practical algorithms in molecular biology and geology [38, 39], computational geometry problems [3], and more applications in [52]. 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 of this recurrence is a standard 2d grid. Each cell $g(D_i, i, j) = D_i + w(i, j)$ 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. Chowdhury and Ramachandran [27] solve the recurrence with $O(n^2)$ work and $O(n^2/BM)$ symmetric cache complexity. This 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 $p \le i < j \le q$. 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 $p \le i \le r < j \le q$, and lastly solves the subtask $(r + 1, q)$ recursively. Note that the middle step also matches a 2d grid with no dependencies between the cells, which can be directly solved using the algorithms in Section 5. This leads the cache complexity and span 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 with base cases $Q(M) = \Theta(M/B)$ and $D(1) = 1$. This solves to the following theorem.

Theorem 7.1. *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 $O(n)$ span.*

7.2 GAP Recurrence We now consider the GAP recurrence, where the analysis of the lower bounds and the new algorithm make use of multiple grid computation. The GAP problem [37, 39] 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, other than changing one character in the string, we can apply a sequence of consecutive deletes that corresponds to a gap in X, and a sequence of consecutive inserts that corresponds to a gap in Y. For simplicity here we assume both strings have length n, but the algorithms and analyses can easily be adapted 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 $w(p, q)$ $(0 \le p < q \le n)$ as the cost of deleting the substring of X from $(p + 1)$-th to q-th character, $w'(p, q)$ for inserting the substring of Y accordingly, and $r(i, j)$ as the cost to change the i-th character in X to j-th character in Y.

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:

$$D_{i,j} = \min \begin{cases} \min_{0 \le q < j} \{D_{i,q} + w'(q, j)\} \\ \min_{0 \le p < i} \{D_{p,j} + w(p, i)\} \\ D_{i-1,j-1} + r(i, j) \end{cases}$$

corresponding to either replacing a character, inserting or deleting a substring. The base case is set to be $D_{0,0} = 0$,

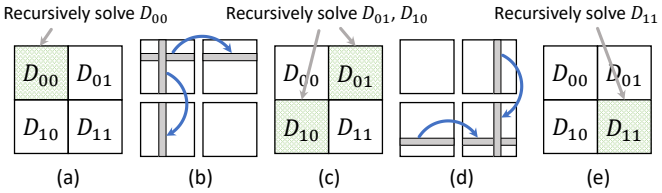

Figure 2: The new cache-oblivious algorithm for GAP recurrences (n is the input size). The algorithm has five steps. Step (a) first recursively solves the D_{00} quadrant, then Step (b) apply $n/2$ inter-quadrant column updates and $n/2$ row updates, each corresponding to a 2d grid. After that, Step (c) recursively solves D_{01} and D_{10}, Step (d) applies another n inter-quadrant updates, and finally Step (e) recursively solves D_{11}. More details about maintaining cache-efficiency is described in Section 7.2 in details.

$D_{0,j} = w(0,j)$ and $D_{i,0} = w'(0,i)$. The diagonal dependency from $D_{i-1,j-1}$ will not affect the asymptotic analysis since it will at most double the memory footprint, so it will not show up in the following analysis.

The best existing algorithms on GAP Recurrence [27, 55] have symmetric cache complexity of $O(n^3/B\sqrt{M})$. This upper 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 8. However, as indicated in Section 3, 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 between the lower and upper bounds is $\Theta(\sqrt{M})$.

We now discuss an I/O-efficient algorithm to close this gap. This algorithm is not optimal, but reduce the gap to $1 + o(1)$. How to remove the low-order term remains an open problem. The new algorithm is similar to Chowdhury and Ramachandran's approach [27] based on divide-and-conquer to compute the 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}. Then the algorithm computes D_{01} and D_{10} within their own ranges, updates D_{11} using the results from D_{01} and D_{10}, and solves D_{11} recursively at the end. The high-level idea is shown in Figure 2.

We note that in Steps (b) and (d), the inter-quadrant updates compute $2 \times (n'/2)$ LWS recurrences (with no data dependencies) each with size $n'/2$ (assuming D has size $n' \times n'$). Therefore, our new algorithm 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, one in a column and the other in a row, 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 as shown in Steps (b) and (d), we start $2 \times (n'/2)$ parallel tasks each with size $n'/2$ and compute a 2d grid on the corresponding row or column, taking the input and output with the correct representation. These updates require work and cache complexity shown in Theorem 7.1. 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 [18] to update the other copy of this quadrant by taking a min as the associative operator $\oplus$, so that the two copies of D are consistent before Steps (c) and (e). 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 up-to-date at the beginning of each recursion in Step (c) and (e).

Our new algorithm still requires $\Theta(n^3)$ work since it does not require extra asymptotic work relative to the original algorithm.. The cache complexity and span satisfy:

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

$$D(n) = 3D(n/2) + 2D_{2C}(n/2)$$

The coefficients are easily determined from the algorithm in Figure 2. We first discuss the symmetric setting. The base cases are $Q(\sqrt{M}) = O(M/B)$ and $Q_{2C}(m) = O(m/B)$ for $m \le M$. This is a "balanced" recurrence with $O(M/B)$ I/O cost per level for $\log_2 \sqrt{M}$ levels. This indicates $Q(M) = O((M/B)\log_2 \sqrt{M})$. The top-level computation is root dominated since the overall number of cells in a level decreases by a half after every recursion. Therefore, if $n > M$, $Q(n) = O(n^2 Q(M)/M) + O(n) \cdot Q_{2C}(n) = O(n^2/B \cdot (n/M + \log_2 \sqrt{M}))$, which is the base-case cost plus the top-level cost. Otherwise, all input/output for each 2d grid in the inter-quadrant update fit in the cache, so we just need to pay $O(n^2/B)$ I/O cost for $\log_2(n/\sqrt{M})$ rounds of recursion, leading to $Q(n) = O(n^2 \log_2(n/\sqrt{M})/B)$. Similarly we can show the asymmetric results by plugging in different base cases.

THEOREM 7.2. *The GAP recurrence can be computed in $\Theta(n^3)$ work, $O(n^{\log_2 3})$ span, 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 results [26–28, 47, 55, 58], the improvement on the symmetric cache complexity is asymptotically $O(\sqrt{M})$ (i.e., n approaching infinity). For smaller

range of n that $O(\sqrt{M}) \le n \le O(M)$, the improvement is $O(n/\sqrt{M}/\log(n/\sqrt{M}))$. (The computation fully fit into the cache when $n < O(\sqrt{M})$.)

Protein accordion folding. The recurrence for protein accordion folding [58] is $D_{i,j} = \max_{1\le k<j-1}\{D_{j-1,k} + w(i,j,k)\}$ for $1 \le j < i \le n$, with $O(n^2/B)$ cost to precompute $w(i,j,k)$ (only $O(n^2)$ possible values). 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 $1 \le k < j-1$ and the output are $D_{i,j}$ for $j < i \le n$. 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 a similar method to the GAP algorithm that keeps two separate copies for D, 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 $O(n\log^2 n)$ span. For matrix transpose, the cost can be derived from 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 [18] provides the following theorem.

Theorem 7.3. *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 $O(n\log^2 n)$ span.*

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

7.3 RNA Recurrence The RNA problem [39] 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:

$$D_{i,j} = \min_{\substack{0\le p<i\\ 0\le q<j}}\{D_{p,q} + w(p,q,i,j)\}$$

with the boundary values $D_{0,0}$, $D_{0,j}$ and $D_{i,0}$. This recurrence is widely used in computational biology, such as in computing the secondary structure of RNA [61].

While the cache complexity of this recurrence seems to be hard to analyze 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 is (1/4)-full, so we can apply the lower bounds in Section 5 here.

Again for a matching upper bound, we need to consider the data dependencies. We can apply the similar technique as in the 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 7.4. *The 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 $\Theta\left(\frac{\omega^{1/2}n^4}{BM}\right)$ respectively, and $O(n^{\log_2 3})$ span.*

7.4 Other DP Recurrences Due to the space limit, we overview the other results in this paper and leave the details to the full version.

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 $0 \le i < j \le n$ 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 computation of this recurrence (without considering dependencies) is a (1/3)-full 3d grid, which has the same lower bound shown in Corollary 8.1. The parallel algorithm is shown in the full version of this paper.

THEOREM 7.5. *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 $\Theta\left(\frac{\omega^{1/3}n^3}{B\sqrt{M}}\right)$ respectively, and $O(n^{\log_2 3})$ depth.*

2-Knapsack Recurrence. Given A_i and B_i for $0 \le i \le n$, the 2-knapsack recurrence computes:

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

for $0 \le i \le n$. 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.

THEOREM 7.6. *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.*

8 Matrix Multiplication and All-Pair Shortest Paths

In this section, we discuss algorithms for combinatorial matrix multiplication and Kleene's algorithm [49] on all pair shortest-paths. We show improved asymmetric cache complexity for the problems, and the APSP algorithm has linear span. We have also included some linear algebra algorithms, including Strassen's algorithm, Gaussian elimination (LU decomposition), and triangular system solver in the full version of this paper [19].

8.1 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 6.1 and 6.2 to get the following result.

Algorithm 2: KLEENE(A)

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

Output: Computed Distance matrix A

```
if |A| = 1 then return A
A00 ← KLEENE(A00)
A01 ← A01 + A00A01
A10 ← A10 + A10A00
A11 ← A11 + A10A01
A11 ← KLEENE(A11)
A01 ← A01 + A01A11
A10 ← A10 + A11A10
A00 ← A00 + A10A01
return A
```

COROLLARY 8.1. *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 $O(\log^2 n)$ span.*

The result for the symmetric case is well-known, but that for the asymmetric case is new.

8.2 All-Pair Shortest Paths We now discuss the cache-oblivious algorithms to solve all-pair shortest paths (APSP) on a graph with improved asymmetric cache complexity and linear span. Regarding the span, Chowdhury and Ramachandran [28] showed an algorithm using $O(n \log^2 n)$ span. There exist work-optimal and sublinear span algorithm for APSP [57], but we are unaware of how to make it I/O-efficient while maintaining the same span. Compared to previous linear span algorithms in [32], our algorithm is race-free and in the classic nested-parallel model. Also, we believe our algorithms are simpler. The improvement is from plugging in the algorithms introduced in Section 5 and 6 to Kleene's Algorithm.

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 [49]. Kleene's algorithm has the same computational DAG as Floyd-Warshall algorithm [35, 60], 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 semiring 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 multiplications, 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 lastly we use them to update the shortest-paths from the first-half vertices.

The cache complexity $Q(n)$ and span $D(n)$ of this algorithm follow the recursions:

$$Q(n) = 2Q(n/2) + 6Q_{MM}(n/2)$$
$$D(n) = 2D(n/2) + 2D_{MM}(n/2)$$

where $Q_{MM}(n)$ is the I/O cost of a matrix multiplication of input size n. The recursion of $Q(n)$ 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.

THEOREM 8.1. *Kleene's Algorithm to compute all-pair shortest paths of a graph of size n uses $\Theta(n^3)$ work, has symmetric and asymmetric cache complexity of $\Theta\left(\frac{n^k}{M^{1/(k-1)}B}\right)$ and $\Theta\left(\frac{n^k\omega^{1/k}}{M^{1/(k-1)}B}\right)$, and $O(n)$ span.*

Similar to some other problems in this paper, the symmetric cache complexity is well-known, but the results in the asymmetric setting as well as the parallel approach are new.

9 Conclusions and Future Work

In this paper, we have shown improved cache-oblivious algorithms of many DP recurrences in the symmetric and asymmetric settings, both sequentially and in parallel. Our key approach is to show the correspondence between the DP recurrences and the k-d grid, and new results for computing the k-d grid. We believe that this abstraction provides a simpler and intuitive framework for better understanding these algorithms, proving lower bounds, and designing algorithms that are both I/O-efficient and highly parallelized. It also provides a unified framework to bound the asymmetric cache complexity of these algorithms.

We believe that the new viewing of these problems are insightful, and based on it, we observe many new open problems. Due to the space limit, these new open problems are discussed in the full version of this paper [19].

Acknowledgments This work is supported by the National Science Foundation under CCF-1314590 and CCF-1533858. The authors thanks 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. Acar, G. Blelloch, and R. Blumofe. The data locality of work stealing. *Theory Comput. Sys.*, 35(3), 2002.

[2] A. Aggarwal, A. Chandra, and M. Snir. Communication complexity of prams. *Theor. Comput. Sci.*, 71(1):3–28, 1990.

[3] A. Aggarwal and M. Klawe. Applications of generalized matrix searching to geometric algorithms. *Discrete Applied Mathematics*, 27(1-2), 1990.

[4] A. Aggarwal and J. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9), 1988.

[5] A. Aho and J. Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

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

[7] L. Arge, G. Brodal, and R. Fagerberg. Cache-oblivious data structures. *Handbook of Data Structures and Applications*, 27, 2004.

[8] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Communication-optimal parallel and sequential cholesky decomposition. *SIAM J. Scientific Computing*, 32(6):3495–3523, 2010.

[9] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Analysis Applications*, 32(3):866–901, 2011.

[10] R. Bellman. *Dynamic programming*. Princeton University Press, 1957.

[11] N. Ben-David, G. Blelloch, J. Fineman, P. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Parallel algorithms for asymmetric read-write costs. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2016.

[12] N. Ben-David, G. Blelloch, J. Fineman, P. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Implicit decomposition for write-efficient connectivity algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

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

[14] G. Blelloch, J. Fineman, P. Gibbons, Y. Gu, and J. Shun. Sorting with asymmetric read and write costs. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2015.

[15] G. Blelloch, J. Fineman, P. Gibbons, Y. Gu, and J. Shun. Efficient algorithms with asymmetric read and write costs. In *European Symposium on Algorithms (ESA)*, 2016.

[16] G. Blelloch, J. Fineman, P. Gibbons, Y. Gu, and J. Shun. Sorting with asymmetric read and write costs. In *arXiv preprint:1603.03505*, 2016.

[17] G. Blelloch, P. Gibbons, Y. Gu, C. McGuffey, and J. Shun. The parallel persistent memory model. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2018.

[18] G. Blelloch, P. Gibbons, and H. Simhadri. Low depth cache-oblivious algorithms. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2010.

[19] G. Blelloch and Y. Gu. Improved parallel cache-oblivious algorithms for dynamic programming and linear algebra. *arXiv preprint arXiv:1809.09330*, 2018.

[20] G. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallel write-efficient algorithms and data structures for computational geometry. In *ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2018.

[21] G. Blelloch, Y. Gu, Y. Sun, and Kanat Tangwongsan. Parallel shortest paths using radius stepping. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

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

[23] C. Bouton. Nim, a game with a complete mathematical theory. *The Annals of Mathematics*, 3(1/4), 1901.

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

[25] E. Carson, J. Demmel, L. Grigori, N. Knight, P. Koanantakool, O. Schwartz, and H. Simhadri. Write-avoiding algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[26] R. Chowdhury, P. Ganapathi, J. Tithi, C. Bachmeier, B. Kuszmaul, C. Leiserson, A. Solar-Lezama, and Y. 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.

[27] R. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *ACM-SIAM symposium on Discrete algorithm (SODA)*, 2006.

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

[29] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. In *International Colloquium on Automata, Languages, and Programming*. Springer, 2010.

[30] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009.

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

[32] D. Dinh, H. Simhadri, and Y. 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.

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

[34] M. Feng and C. Leiserson. Efficient detection of determinacy races in cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[35] R. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6), June 1962.

[36] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999.

[37] Z. Galil and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64(1), 1989.

[38] Z. Galil and K. Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92(1), 1992.

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

[40] Y. Gu. *Write-Efficient Algorithms*. PhD Thesis, Carnegie Mellon University, 2019.

[41] Y. Gu, Y. Sun, and G. Blelloch. Algorithmic building blocks for asymmetric memories. In *European Symposium on Algorithms (ESA)*, 2018.

[42] D. Hirschberg and L. Larmore. The least weight subsequence problem. *SIAM Journal on Computing*, 16(4), 1987.

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

[44] S. Huang, H. Liu, and V. Viswanathan. Parallel dynamic programming. *IEEE transactions on parallel and distributed systems*, 5(3), 1994.

[45] S. Huang, H. Liu, and V. Viswanathan. A sublinear parallel algorithm for some dynamic programming problems. *Theoretical Computer Science*, 106(2), 1992.

[46] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64(9):1017–1026, 2004.

[47] S. Itzhaky, R. Singh, A. Solar-Lezama, K. Yessenov, Y. Lu, C. Leiserson, and R. 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.

[48] R. Jacob and N. Sitchinava. Lower bounds in the asymmetric external memory model. In *ACM Aymposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017.

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

[50] J. Kleinberg and E. Tardos. *Algorithm design*. Pearson Education India, 2006.

[51] D. Knuth and M. Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11), 1981.

[52] M. Künnemann, R. Paturi, and S. Schneider. On the fine-grained complexity of one-dimensional dynamic programming. *arXiv preprint arXiv:1703.00941*, 2017.

[53] L. Loomis and H. Whitney. An inequality related to the isoperimetric inequality. *Bulletin of the American Mathematical Society*, 55(10):961–962, 1949.

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

[55] Y. Tang and S. 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.

[56] Y. Tang, R. You, H. Kan, J. Tithi, P. Ganapathi, and R. 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.

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

[58] J. Tithi, P. Ganapathi, A. Talati, S. Aggarwal, and R. Chowdhury. High-performance energy-efficient recursive dynamic programming with matrix-multiplication-like flexible kernels. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015.

[59] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. Persistent memory i/o primitives. In *International Workshop on Data Management on New Hardware*, page 12. ACM, 2019.

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

[61] M. Waterman and T. Smith. Rna secondary structure: A complete mathematical analysis. *Mathematical Biosciences*, 42(3-4), 1978.