Cache-Oblivious Wavefront: Improving Parallelism of Recursive Dynamic Programming Algorithms without Losing Cache-Efficiency

Yuan Tang [*] Ronghui You Haibin Kan

Software School, School of Computer Science, Fudan University
Shanghai Key Laboratory of Intelligent Information Processing, Fudan University
Shanghai, P. R. China
[yuantang, 11300720164, hbkan]@fudan.edu.cn

Jesmin Jahan Tithi Pramod Ganapathi
Rezaul A. Chowdhury [†]

Department of Computer Science, Stony Brook University
Stony Brook, NY 11790, USA
[jtithi, pganapathi, rezaul]@cs.stonybrook.edu

Abstract

State-of-the-art cache-oblivious parallel algorithms for dynamic programming (DP) problems usually guarantee asymptotically optimal cache performance without any tuning of cache parameters, but they often fail to exploit the theoretically best parallelism at the same time. While these algorithms achieve cache-optimality through the use of a recursive divide-and-conquer (DAC) strategy, scheduling tasks at the granularity of task dependency introduces artificial dependencies in addition to those arising from the defining recurrence equations. We removed the artificial dependency by scheduling tasks ready for execution as soon as all its real dependency constraints are satisfied, while preserving the cache-optimality by inheriting the DAC strategy. We applied our approach to a set of widely known dynamic programming problems, such as Floyd-Warshall's All-Pairs Shortest Paths, Stencil, and LCS. Theoretical analyses show that our techniques improve the span of 2-way DAC-based Floyd Warshall's algorithm on an n node graph from $\Theta\left(n\log^2 n\right)$ to $\Theta(n)$, stencil computations on a d-dimensional hypercubic grid of width w for h time steps from $\Theta\left((d^2h)w^{\log(d+2)-1}\right)$ to $\Theta(h)$, and LCS on two sequences of length n each from $\Theta\left(n^{\log_2 3}\right)$ to $\Theta(n)$. In each case, the total work and cache complexity remain asymptotically optimal. Experimental measurements exhibit a 3 - 5 times improvement in absolute running time, 10 - 20 times improvement in burdened span by Cilkview, and approximately the same L1/L2 cache misses by PAPI.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; G.1.0 [*Mathematics of Computing*]: Numerical Analysis—Parallel Algorithms.; G.4 [*Mathematical Software*]: Algorithm design and analysis

General Terms Algorithms, Scheduling, Performance.

Keywords cache-oblivious parallel algorithm, cache-oblivious wavefront, dynamic programming, multi-core, nested parallel computation, Cilk

[*] Part of the work was done when the author was a visiting scientist at MIT CSAIL during the summer of 2014.

[†] Rezaul Chowdhury and Pramod Ganapathi were supported in part by NSF grants CCF-1162196 and CCF-1439084.

PPoPP '15, February 7–11, 2015, San Francisco Bay Area, CA, USA.
Copyright © 2015 ACM 978-1-4503-3205-7/15/02...$15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

1. Introduction

Dynamic programming (DP) [4] algorithms build optimal solutions to a problem by combining optimal solutions to many overlapping subproblems. DP algorithms exploit this overlap to explore otherwise exponential-sized problem spaces in polynomial time, making them central to many important applications ranging from logistics to computational biology [20, 30, 33, 38, 45–47, 50, 56].

State-of-the-art cache-oblivious [27] parallel (COP) algorithms for DP problems [11–13, 15, 16] often trade off parallelism for better cache performance. Those algorithms typically employ a recursive divide-and-conquer (DAC) approach. This approach allows an algorithm to achieve asymptotically optimal serial cache performance through increased "temporal locality"[1] while at the same time remain oblivious of the parameters of the cache hierarchy. In other words, these algorithms do not need to tune for different memory hierarchies, and thus are portable across machines. However, scheduling tasks at the granularity of task dependency often limits parallelism by introducing artificial dependencies among recursive subtasks in addition to those arising from the defining recurrence equations. As a result, most state-of-the-art COP DP algorithms fail to achieve optimal serial cache performance and optimal parallelism simultaneously.

Performance of a recursive DAC based COP algorithm when run under a state-of-the-art scheduler on a modern multi-core machine, depends on both its serial cache complexity and parallelism. Before we elaborate on the relationship between serial and parallel performances of these algorithms, we explain below how we will analyze their performance throughout this paper.

Nested Parallel Computations and Work-Span Model. We use the "work-span model" to analyze the performance of nested parallel computation [8, 49] that covers all recursive DAC-based DP algorithms in this paper. By T_p and Q_p we denote the running

[1] Temporal locality — whenever a cache block is brought into the cache, as much useful work as possible is performed on it before it's removed from the cache.

time and the cache complexity of the algorithm, respectively, on p processing cores. The total "work" performed by the algorithm is given by its running time T_1 on a single core. Its theoretical execution time on an unbounded number of cores is called its "span" or "critical path length" or "depth", and is denoted as T_∞. The "parallelism" of an algorithm is then given by (T_1/T_∞), which is the average amount of work it performs per step along the span. While one can always increase parallelism by increasing the total work T_1, which is not very interesting and often not useful. In this paper, we focus on reducing the span (T_∞) of a given recursive DP algorithm while keeping its T_1 fixed. Table 1 lists the notations and acronyms.

Influences of Serial Cache Complexity and Parallelism on Parallel Performance. State-of-the-art schedulers for shared-memory multicore machines guarantee good parallel performance provided the algorithm being run shows both good cache performance on a serial machine and high parallelism (i.e., low span). For example, the widely used "randomized work-stealing scheduler" [1, 9, 26] for distributed caches provides the following performance guarantees w.h.p.[2]:

Table 1: Standard acronym and notations used throughout the paper.

Symb.	Meaning
DP	Dynamic Programming
COP	Cache-Oblivious Parallel, used to denote original recursive divide-and-conquer based standard algorithm
COW	Cache-Oblivious Wavefront, used to denote new algorithms proposed in this paper
DAC	divide-and-conquer
FW	Floyd-Warshall
APSP	All-Pairs-Shortest-Paths
n	Input size or input parameter
p	Number of processing cores
M	Cache or memory size
B	Block size or cache line size or I/O transfer size
T_1	Work or serial running time
T_∞	Span or critical path length or running time on an infinite number of cores
T_p	Parallel running time on p cores
$\frac{T_1}{T_\infty}$	Parallelism
Q_1	Serial cache complexity
Q_p	Parallel cache complexity on p cores

$$T_p = O(T_1/p + T_\infty) \quad \text{and} \quad Q_p = Q_1 + O(p(M/B)T_\infty),$$

where, p is the number of processing cores, M is the cache size, and B is the cache line size. Clearly, good parallel cache performance (Q_p) requires both good serial cache performance (Q_1) and a low span (T_∞). The "parallel depth-first scheduler" [7] for shared caches, on the other hand, guarantees that

$$Q_p \le Q_1 \quad \text{provided} \quad M_p \ge M_1 + \Theta(pT_\infty),$$

where, M_p is the size of cache shared by p cores, and M_1 is the size of cache on a serial machine. The more recently proposed "space-bounded schedulers" [8, 12, 14, 17, 49] guarantee:

$$T_p = O(T_1/p + T_\infty) \quad \text{and} \quad Q_p = O(Q_1),$$

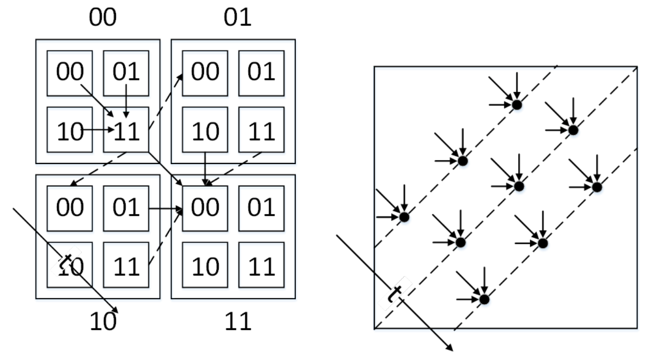
which is achieved by trading off parallelism for reduced cache misses. Again, the lower the values of Q_1 and T_∞, the better the parallel performance.

With the increase of core count on multi-core machines, and the emergence of many-core processors with cache hierarchies such as the "Intel MIC" [35], the need for algorithms with both optimal cache complexity and high parallelism continues to grow.

The Tradeoff between Q_1 and T_∞ in Recursive DAC-based DPs. We explain the tradeoff using the longest common subsequence (LCS) problem as an example.

Given two sequences $S = \langle s_1, s_2, \ldots, s_m \rangle$ and $T = \langle t_1, t_2, \ldots, t_n \rangle$, we define $X(i,j)$, $(0 \le i \le m, 0 \le j \le n)$ to be the length of the longest common subsequence (LCS) of S and T. LCS can be computed using the following defining recurrence [18] (A similar recurrence applies to the "pairwise sequence alignment with affine

[2] For an input of size n, an event E occurs w.h.p. (with high probability) if, for any $\alpha \ge 1$ and c independent of n, $Pr(E) \ge 1 - \frac{c}{n^\alpha}$. The larger the value of n, the closer $Pr(E)$ is to 1, and $\lim_{n\to\infty} Pr(E) = 1$.



(a) 2-way recursive DAC algorithm with $Q_1(n) = O\left(n^2/(BM)\right)$ and $T_\infty(n) = O\left(n^{\log_2 3}\right)$.

(b) Straightforward parallel looping (no tiling) algorithm with $Q_1(n) = O\left(n^2/B\right)$ and $T_\infty(n) = O(n)$.

Figure 1: Do we need to tune LCS between parallelism and cache efficiency? 2-way recursive DAC algorithm has best cache complexity bound but worst span, straightforward parallel looping algorithm has best span but worst cache performance, any r-way $(r \in (2,n))$ recursive DAC algorithm is in between. Note that the solid arrows in diagram denote real dependencies arising from defining recurrence and dashed arrows denote artificial dependencies introduced by the algorithm. Timeline t is the computing direction.

gap cost" problem [31]):

$$X(i,j) = \begin{cases} 0 & \text{if } i = 0 \lor j = 0 \\ X(i-1, j-1) + 1 & \text{if } i, j > 0 \land s_i = t_j \\ \max\{X(i, j-1), X(i-1, j)\} & \text{if } i, j > 0 \land s_i \ne t_j \end{cases}$$

A 2-way recursive DAC algorithm (see Figure 1a) for the LCS problem was given in [15]. We assume for simplicity of exposition that $m = n = 2^k$ for some integer $k \ge 0$. The algorithm splits the DP table X into four equal quadrants: X_{00} (top-left), X_{01} (top-right), X_{10} (bottom-left), and X_{11} (bottom-right). It then recursively computes the quadrants in the following order: X_{00} first, then X_{01} and X_{10} in parallel, and finally X_{11}, i.e. $X_{00}; X_{01} || X_{10}; X_{11}$. A recursive call on an $n' \times n'$ sub-matrix uses only $\Theta(n')$ space. When the space needed to solve a subproblem is small enough to fit into the cache, the algorithm won't incur any cache misses in addition to those needed to bring the subproblem into the cache and write it back to the memory. Thus the algorithm fully exploits the temporal cache locality, and the resulting cache complexity $Q_1(n) = O\left(n^2/(BM)\right)$ can be shown to be optimal [15]. However, the span $T_\infty(n) = \Theta\left(n^{\log_2 3}\right)$ is suboptimal since a straightforward looping algorithm that computes all entries of a diagonal in parallel and proceeds in this way diagonal by diagonal has $\Theta(n)$ span (though the cache complexity is $\Omega\left(n^2/B\right)$).

In general, assuming $m = n = r^k$ for integers $r \ge 2$ and $k \ge 0$, and performing an $r \times r$ decomposition at each level and recursively computing the r^2 subproblems, we have an r-way recursive DAC algorithm. For such an approach, Q_1 can be computed as follows.

$$Q_1(n) = \begin{cases} O(n/B + 1) & \text{if } n \le \alpha_r M, \\ r^2 Q_1(n/r) + \Theta(1) & \text{if } n > \alpha_r M; \end{cases}$$

where, $\alpha_r \in (0,1]$ is a constant automatically and implicitly determined by the r-way recursive DAC strategy [3]. The recurrence for

[3] α_r is the ratio of the subtask size and the cache size in the first level of recursion in which each individual subtask fits into the cache. This α_r is only used during the analysis of algorithm, it does not appear anywhere in the algorithm or its implementation.

span is

$$T_\infty(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ (2r-1)T_\infty(n/r) + \Theta(1) & \text{if } n > 1. \end{cases}$$

The recurrences above solve to:

$$Q_1(n) = O\left(\min\left\{n^2, \frac{n^2 r}{BM} + \frac{n^2 r^2}{M^2}\right\}\right) \text{ and } T_\infty(n) = \Theta\left(n^{\log_r(2r-1)}\right).$$

Observe that while $Q_1(n)$ is the lowest (and optimal) when $r = 2$, it grows as r increases and becomes as bad as $O\left(n^2\right)$ when r exceeds M (can be slightly improved though [4]). On the other hand, $T_\infty(n)$ has the worst value when $r = 2$, but it improves as r increases and reaches optimal value $\Theta(n)$ when $r = n$.

The change in behavior of Q_1 and T_∞ with the change of r can be explained as follows. When r is small, artificial dependency relations due to scheduling at the granularity of task dependency prevents many completely independent subtasks from executing in parallel. For example, consider the 2-way DAC algorithm in Figure 1a in which the solid arrows denote real data dependencies arising from the defining recurrence and the dashed arrows denote artificial dependencies introduced by the algorithm. The dependency of $X_{10,00}$ on $X_{00,11}$ is artificial, while the dependency of $X_{11,00}$ on $X_{00,11}$ is real, and so on. As r increases, the subproblem sizes decrease, and the number of artificial dependencies keeps decreasing which leads to increased parallelism. But large r is bad for Q_1 as that leads to smaller subproblems, as a result, the largest subproblems that fit into the cache is often much smaller than the size of the cache leading to cache under-utilization and loss of temporal locality.

In the light of discussion above, traditional wisdom may suggest that one should strike a balance point between span and cache complexity in order to get good performance in practice. Apparently, the intuition behind balance is that we can not get both optimal at the same time.

Our Contributions.

- **[Algorithmic Framework]** We introduce parallel "Cache-Oblivious Wavefront" (COW) algorithms that perform divide-and-conquer of the DP table in exactly the same way as standard cache-optimal recursive DAC-based DP algorithms, but consider a recursive subtask ready for execution as soon as all its real dependency constraints are satisfied. By performing divide-and-conquer the same way as original 2-way DAC-based algorithm, COW algorithms retain the recursive execution order of subtasks derived from the same parent task. As a consequence, COW algorithms inherit the cache-obliviousness and (serial) cache-optimality properties of the standard recursive DAC-based algorithms. By scheduling a recursive subtask ready for execution based only on its real dependency constraints, COW algorithms lead to potentially improved span. To distinguish, we name original recursive DAC-based standard algorithm as "cache-oblivious parallel" (COP) algorithm, and the new proposed algorithm "cache-oblivious wavefront" (COW) because the overall execution pattern of COW algorithms usually proceed like propagating a wave of executing and ready-to-execute subtasks through a dynamically unfolding recursive DAC tree.

  Depending on how long it takes to align a spawned subtask to the advancing wavefront, we have devised two different techniques for efficient wavefront alignment of subtasks without wasting valuable computing resources (i.e., cores).

  1. **[Eager COW]** We present in Section 2 algorithms when subtasks can start executing within $O(1)$ time of spawning. We use such algorithms for Floyd-Warshall's APSP and stencil computations among others.
  2. **[Lazy COW]** We present in Section 3 algorithms when subtasks require $\omega(1)$ time to get ready for execution after they are spawned. We use such algorithms for LCS, among others.

- **[Theoretical Analyses]** We provide theoretical analyses of potentially improved span in Sections 2 and 3 for Eager and Lazy COW algorithms respectively. The improvement in span (T_∞) comes without increasing the total work (T_1) and / or serial cache complexity (Q_1) of the original COP algorithm.
- **[Experimental Results]** We have implemented several COW algorithms and compared them in Section 4 with standard 2-way COP algorithm, tiled parallel loop, and standard parallel loop implementations of the same DP on a number different hardware platforms including 16-core and 32-core machines. Experimental measurements of absolute running times, relative speedups w.r.t. direct parallel loops, burdened span by Cilkview [34], and L1/L2 cache misses by PAPI [10] validate our claims.

We conclude the work in Section 6 with a discussion on the limitations of current approach.

2. Eager Recursion to Simulate a COW

This section introduces an algorithmic technique, called the Eager recursion, to simulate a COW algorithm for a wide range of cache-optimal recursive DAC DP algorithms including the ones that solve path problems over closed semirings [13, 15] and perform stencil computations [23–25, 53].

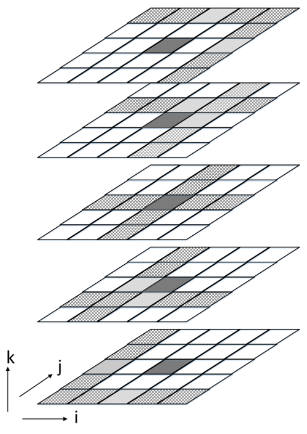
Semiring serves as a general framework for solving path problems in directed graphs [3], and both Floyd-Warshall's (FW) algorithm [22] for finding All-Pairs Shortest Paths (APSP) and Warshall's algorithm [55] for finding transitive closures [55] are instantiations of this algorithm. A stencil, on the other hand, defines the value of a grid point in a d-dimensional spatial grid at time t as a function of neighboring grid points at recent times before t. A stencil computation computes the stencil for each grid point over many time steps, and has numerous applications [6, 19, 21, 23, 25, 36, 37, 39, 42–44, 51, 53, 57].

The main insight motivating the design of the Eager recursion technique came from the following observation. Cache-optimal recursive DAC algorithms schedule the execution of recursive subtasks at the granularity of task dependency. As a result, they often delay the execution of subtasks because of some artificial dependencies arising from such a coarse-grain ordering even when the real dependency constraints are already satisfied.

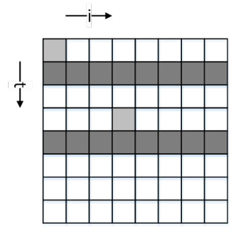
The Eager recursion technique tries to execute a subtask as soon as all its real dependency constraints are satisfied while still following the recursive DAC scheme. For example, suppose X and Y are two tasks in such an algorithm and subtask X_i of X has a real dependency on subtask Y_j of Y, and there are no other real dependencies between X and Y. Let's assume for simplicity that each subtask of X and Y takes only $O(1)$ time to execute. The standard algorithm will not let X execute until all subtasks of Y completes execution though Y_j has perhaps finished much earlier and there was no need of extra delay for X. In our Eager approach, task X is spawned at the same time as Y. Instead of the task X waiting for the completion of the entire task Y, subtasks of X are spawned in parallel with those of Y. Subtask X_i will be busy waiting for the completion of Y_j by continuously checking the wavefront data structure. When Y_j completes execution it updates the wavefront and within constant

[4] $Q_1(n)$ can be improved to $O\left(n^2/B\right)$ by suitably fixing the order of execution of subtasks.

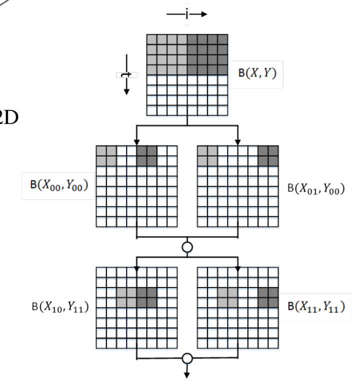
time X_i (and thus X) can advance. This Eager approach works efficiently provided the structure of computation guarantees that real dependency constraints of X are satisfied within $O(1)$ time of the start of execution of Y.



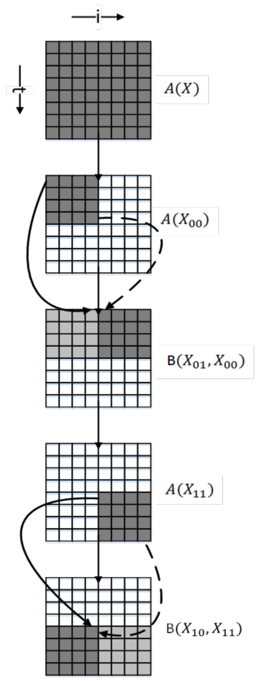
(a) Data dependency pattern of 2D FW



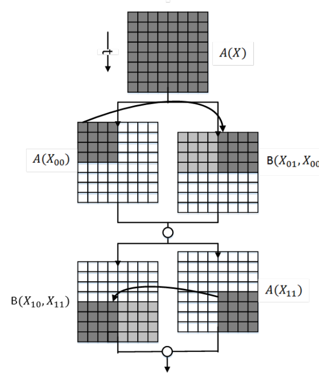
(c) Data dependency pattern of 1D FW



(d) Procedure B of 1D FW in both 2-way DAC algo. and Eager recursion



(b) Procedure A of 1D FW in 2-way DAC algo.



(e) Procedure A of 1D FW in Eager recursion.

Figure 2: Data dependency pattern of 1D/2D FW. The solid arrows indicate the real dependencies from defining recurrence. The dashed arrows indicate the artificial dependencies introduced by the algorithm. Dark-shaded cells denote the cells to update (write) in current procedure and light-shaded cells denote the dependent (read) cells. Notation $\mathrm{A}(X)$ denotes a computation on data block X with all dependent diagonal cells self-contained and $\mathrm{B}(X,Y)$ denotes a computation on data block X with all dependent diagonal cells contained in a completely disjoint data block Y.

Eager Recursion on a Simple Example. We will explain our approach using a simple synthetic benchmark as an example. This benchmark, called 1D FW, is a simplification and abstraction of original 2D FW algorithm [22] in the sense that they have a similar data dependency pattern from defining recurrence equations. The defining recurrence of 1D FW for $1 \le i, t \le n$ is as follows which assumes that $d(0,i)$ for $1 \le i \le n$ are already known.

$$d(t,i) = d(t-1,i) \oplus d(t-1,t-1) \tag{1}$$

Figures 2c and 2a show the dependency pattern of 1D FW and 2D FW side-by-side to exhibit the similarity. In both figures, dark-shaded cells denote the cells to update (write), and the light-shaded cells denote the dependent (read) diagonal cell from previous time step $(t-1)$. We can see that the update of any cell $d(t,i)$ in 1D FW depends on both the diagonal cell from previous time step, i.e. $d(t-1,t-1)$, and the cell of the same space position from previous time step, i.e. $d(t-1,i)$. The serial 1D FW problem can be solved in $\Theta\left(n^2\right)$ time using $\Theta(n)$ space.

Based on the dependency pattern in Figure 2c, we have a COP algorithm based on 2-way recursive DAC strategy adapted from [13] to solve the 1D FW with optimal work and optimal cache complexity. This algorithm comprises two recursive functions named A and B. $\mathrm{A}(X)$ denotes a computation on data block X with all dependent diagonal cells self-contained, and $\mathrm{B}(X,Y)$ denotes a computation on data block X with all dependent diagonal cells included in a completely disjoint data block Y. The recursive structure of algorithm is as follows:

$$\mathrm{A}(X): \quad \mathrm{A}(X_{00}); \mathrm{B}(X_{01}, X_{00}); \mathrm{A}(X_{11}); \mathrm{B}(X_{10}, X_{11})$$

$$\mathrm{B}(X,Y): \quad \mathrm{B}(X_{00}, Y_{00}) || \mathrm{B}(X_{01}, Y_{00}); \mathrm{B}(X_{10}, Y_{11}) || \mathrm{B}(X_{11}, Y_{11}).$$

Figures 2b and 2d are graphical illustrations of the algorithm. Observing that A is serialized by the task-level dependency, all parallelism come from B.

For this 2-way COP algorithm, we have following recurrences to compute the span $(T_\infty(n) = T_{\infty,A}(n))$, and serial cache complexity $(Q_1(n) = Q_A(n))$, respectively:

$$\begin{aligned} T_{\infty,A}(n) &= 2T_{\infty,A}(\tfrac{n}{2}) + 2T_{\infty,B}(\tfrac{n}{2}) \qquad & Q_A(n) &= 2Q_A(\tfrac{n}{2}) + 2Q_B(\tfrac{n}{2}) \\ T_{\infty,B}(\tfrac{n}{2}) &= 2T_{\infty,B}(\tfrac{n}{4}) & Q_B(\tfrac{n}{2}) &= 4Q_B(\tfrac{n}{4}) \end{aligned}$$

Assuming that $T_{\infty,A}(1) = \Theta(1)$ and $T_{\infty,B}(1) = \Theta(1)$. For $n \le \varepsilon M$, we have $Q_A(n) = O\left(\frac{n}{B}\right)$ and $Q_B(n) = O\left(\frac{n}{B}\right)$ to indicate that the recursive calculation of cache complexity should terminate as soon as A and B fit into the cache, though this terminating condition is oblivious to the algorithm design and implementation. The recurrences solve to: $T_\infty(n) = T_{\infty,A}(n) = O(n \log_2 n)$, and $Q_1(n) = Q_A(n) = O\left(n^2/(BM)\right)$.

While the cache complexity is optimal, the span is not because the span of straightforward parallel looping algorithm solving this problem is only $O(n)$. The reason behind the suboptimal span can be understood by examining Figure 2b, in which solid arrows represent the real data dependency originating from the defining recurrence Equation (1) and dashed arrows represent the artificial dependency introduced by the algorithm. For example, at each recursion level of A, the computation of first cell of X_{01} will have to wait on the last cell of X_{00} and the first cell of X_{10} will wait on the last cell of X_{11}. These dependencies do not arise from the defining recurrence but are introduced by the structure of algorithm, more precisely, the scheduling of subtasks at the granularity of task dependency. If we can reduce those artificial dependencies, we can possibly improve the span asymptotically.

The basic idea behind Eager recursion is to spawn all sibling subtasks at the same recursion level in A simultaneously, and introduce atomic operations to guard the data dependency on diagonal cells. The execution pattern of A changes to:

$$\mathrm{A}(X): \quad \mathrm{A}(X_{00}) || \mathrm{B}(X_{01}, X_{00}); \mathrm{A}(X_{11}) || \mathrm{B}(X_{10}, X_{11}).$$

A graphical illustration is shown in Figure 2e. The recursion of B doesn't change since there are no artificial dependencies there. Note that $B(X_{01}, X_{00})$ slightly lags behind $A(X_{00})$ in the same parallel block of $A(X)$ because B can not start computation until sibling A finishes its first base case due to the dependency on diagonal. Since the dependency between siblings in A occurs only on diagonal cells, and there is a strict (monotonically increasing) timing order in computing diagonal cells, atomic operations just need to be imposed on one integer that records the latest time step of computed diagonal cells. This data structure (one integer in the case of 1D FW) that records the progressing information and protected by atomic operations is called *wavefront* because the progress of algorithm usually proceeds alike a wavefront.

The span recurrence of A under Eager recursion changes to

$$T_{\infty,A}(n) = T_{\infty,B}(n/2) + T_{\infty,B}(n/2) + O(1)$$

, because B and A within the same parallel block overlaps and B depends on the diagonal data produced by A but not vice versa (Figure 2e). The $O(1)$ term in the recurrence represents the dependency cost of B (atomic busy-waiting) along time dimension on the diagonal cells computed by A at each recursion level, and is constant because if we assume perfect scheduling and constant base case size, B needs to wait for only one diagonal base case of A before it can start execution as shown in Figure 2e. The span $T_\infty(n) = T_{\infty,A}(n)$ of the entire algorithm then solves to $O(n)$, which matches the bound of straightforward parallel looping algorithm and is optimal. The improvement of performance in practice is also significant as shown in Section 4.

The cache recurrence of A under Eager recursion changes to

$$Q_A(n) = 2Q_A(n/2) + 2Q_B(n/2) + O(n),$$

where the additional term $O(n)$ accounts for the cache miss overhead during atomic busy-waiting time of all first-row subtasks in B on A for the first diagonal base case (see Figure 2e). Note that in the original 2-way COP algorithm adapted from [13], due to the dependency of B on diagonal cells computed by A, the cache complexity of fetching the diagonal cells from A to B exists anyway (included in $Q_B(n)$). The Eager recursion algorithm just charges explicitly an additive $O(1)$ (assuming base case size is constant) cost on this dependency to all first-row sub-tasks in B. The cache recurrence of B doesn't change. The recurrence solves to $O\left(n^2/(BM) + n\log(n/M)\right)$. If we assume $n/\log n >> BM$, the complexity reduces to $O\left(n^2/(BM)\right)$, the same as original algorithm.

We have measured the L1/L2 cache misses of the benchmarks in Section 4 using PAPI [10], the results of which validate our claim.

Claim 1. *The COW algorithm for* 1D *FW has only data dependency originating from the defining recurrence Equation (1).*

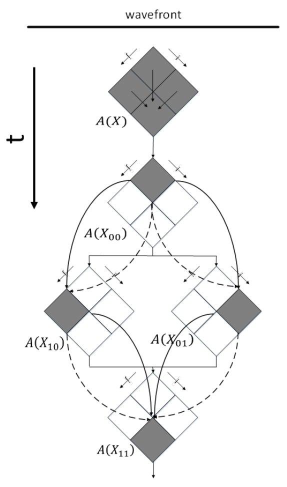
Proof of the claim is obvious from the algorithm, so is omitted. Similar ideas apply to original 2D FW and stencil computation.

Claim 2. *The COW algorithm for* 2D *FW improves the span from* $O\left(n\log^2 n\right)$ *in [13] to* $O(n)$ *with the same cache complexity bound. The COW algorithm for* d*-dimensional Stencil improves the span from* $\Theta\left((d^2h)w^{\lg(d+2)-1}\right)$ *in [53], where* h *is the height and* w *is the width of the hypercubic computing space, respectively, to* $O(h)$ *with the same cache complexity bound.*

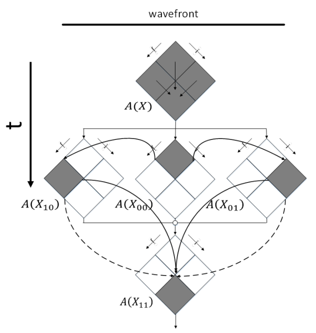
3. Lazy Recursion to Simulate a COW

This section introduces Lazy recursion – an algorithmic technique to simulate a cache-oblivious wavefront efficiently for a range of recursive DP algorithms for which the "eager" recursion technique presented in Section 2 may end up wasting too much computing resources because of the busy-waiting in practice[5]. This set of DP problems include LCS [15, 16, 18], pairwise sequence alignment with affine gap cost [31], sequence alignment with gaps (GAP problem) [15, 28, 29, 56], and the parenthesis problem [12, 29].

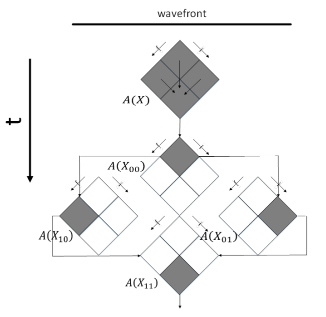
Why is the Simple Eager Technique not Efficient for LCS?



(a) Procedure A of LCS in 2-way COP algo.



(b) Procedure A of LCS in Eager COW.



(c) Procedure A of LCS in Lazy COW.

Figure 3: Comparison between classic 2-way COP, Eager, Lazy COW algorithms for LCS. The solid arrows indicate the real dependencies from defining recurrence. The dashed arrows indicate the artificial dependencies introduced by algorithm.

A classic 2-way COP algorithm (see Figure 3a) proceeds as follows: it divides the input task at each recursion level into four equally sized subtasks and executes them in diagonal order along timeline t. The execution pattern is

$$A(X): \quad A(X_{00}); A(X_{01}) || A(X_{10}); A(X_{11}).$$

This execution pattern creates artificial data dependencies among subtasks as shown by the dashed arrows in Figure 3a. For example, the first cell of X_{10} depends on the last cell of X_{00}, which is an artificial dependency. The dependencies arising from the defining recurrence are drawn as solid arrows.

If we simply adopt the Eager recursion technique introduced in Section 2 and execute the pattern as [6]

$$A(X): \quad A(X_{00}) || A(X_{01}) || A(X_{10}); A(X_{11}),$$

we still have two problems (see Figure 3b):

1. Artificial dependencies still exist, e.g., the first cell of X_{11} will have to wait for the last cells of X_{10} and X_{01} to be computed.

[5] Though does not matter in theory because when we compute T_∞ we assume an infinite number of computing cores anyways

[6] note that $A(X_{11})$ still have to lag behind $A(X_{00})$ because of the strict data dependency of the first cell of X_{11} on the last cell of X_{00}

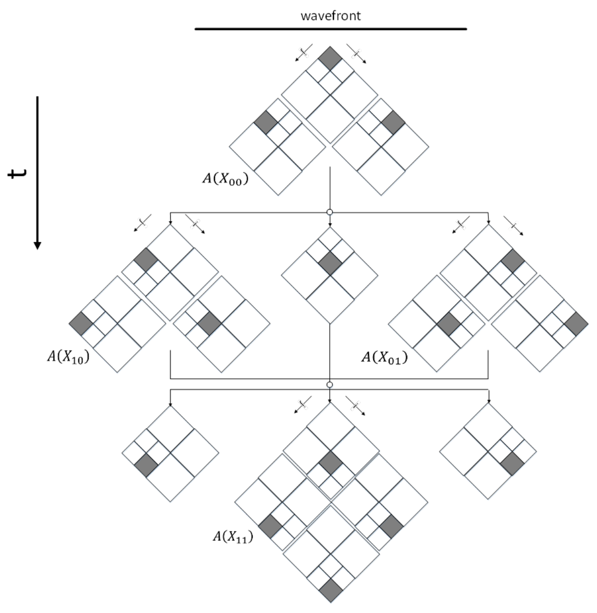


Figure 4: Execution details of procedure A of LCS in Lazy recursion.

2. X_{01} and X_{10} are spawned early, but can not start computing their first cells until X_{00} has updated at least half of its cells. In other words, the simple Eager recursion causes non-constant busy-waiting time ($O(n)$ in this case) of X_{01} and X_{10} on X_{00} along time dimension. The non-constant busy-waiting time implies that there will be fewer computing cores for producer X_{00} to produce the data on *wavefront* to unlock consumers X_{01} and X_{10}. In practice, we do not have an infinite number of computing cores. All computing cores should perform useful work instead of busy-waiting unless the busy-waiting takes only constant time. Note that when we say waiting time, we only count the waiting time along time dimension because the computation along space dimension are parallelized.

An ideal execution pattern for LCS is illustrated in Figure 3c. In the recursive execution, as soon as X_{00} updates half of its cells up to the middle line, X_{01} and X_{10} start executing, and as soon as X_{01} and X_{10} update half of their cells, X_{11} starts running. So there are overlaps of execution on timeline among all four subtasks at each recursion level.

For the ideal algorithm, the span recurrence becomes

$$T_\infty(n) = 2T_\infty(n/2) = O(n),$$

because the execution time of X_{01} and X_{10} completely overlap with those of X_{00} and X_{11}. The cache recurrence doesn't change because the DAC behavior doesn't change.

3.1 Lazy Recursion

We devised a Lazy recursion technique as shown in Figure 4 to simulate the ideal algorithm in Figure 3c.

Conceptually, the Lazy recursion approach perform the same divide-and-conquer as classic 2-way COP algorithm but schedule the execution of subtasks across different levels of DAC tree aligned to a wavefront (proof in Claim 3). In the case of LCS, referring to Figure 4, at each recursion level, except the last X_{11} each subtask pushes its X_{11} subsubtask one level up in the DAC tree to execute in parallel with the subtask's siblings. The execution pattern shown in Figure 4 is for one level of Lazy recursion

$$\begin{aligned}
\mathrm{A}(X):\ & \mathrm{A}(X_{00,\Gamma});\\
& \mathrm{A}(X_{00,11})||\mathrm{A}(X_{01,\Gamma})||\mathrm{A}(X_{10,\Gamma});\\
& \mathrm{A}(X_{01,11})||\mathrm{A}(X_{10,11})||\mathrm{A}(X_{11})
\end{aligned}$$

where $X_{00,\Gamma}$ denotes all sub-tasks in X_{00} except $X_{00,11}$ and so on. We prove later in this section that by infinite levels of Lazy recursions, the execution of all subtasks across different levels of the DAC tree are aligned to a wavefront along time dimension.

The span recurrence of LCS ($T_\infty(n) = T_{\infty,\square}(n)$) under Lazy recursion is:

$$\begin{aligned}
T_{\infty,\square}(n) &= 2T_{\infty,\Gamma}(n/2) + T_{\infty,\square}(n/2)\\
T_{\infty,\Gamma}(n) &= 2T_{\infty,\Gamma}(n/2)
\end{aligned}$$

The explanation of span recurrence is as follows. $T_{\infty,\square}$ denotes the span of procedure that computes a task without pushing its X_{11} up the DAC tree, and $T_{\infty,\Gamma}$ denotes the span of procedure that computes a subtask with its X_{11} pushed up the DAC tree and has the execution of X_{11} overlapped with its siblings. Since in the latter case, the X_{11} quadrant of a subtask always executes in parallel with that subtask's siblings which are geometrically larger, the execution time of that X_{11} quadrant is not counted in $T_{\infty,\Gamma}$. We can see from Figure 4 that for the initial recursion, span of all subtasks except the one handling X_{11} are counted as $T_{\infty,\Gamma}$. For subsequent recursions, except that the span of $X_{11,11,\ldots,11}$ will be given by $T_{\infty,\square}$, all other subtasks will be executed in an overlapping fashion and will have $T_{\infty,\Gamma}$ type span. So, the overall span of algorithm reduces to $O(n)$. In Section 4 we will see that the performance improvement in practice matches our theoretical predictions.

We argue that the cache recurrence doesn't change because if we put the child X_{11} back to its corresponding $T_{\infty,\Gamma}$ parent, the number, the shapes, and the sizes of subtasks at each recursion level doesn't change from the original 2-way COP algorithm. Note that the cache recurrence counts only the number, the shapes and the sizes of subtasks at each recursion level. If we assume an infinite number of processing cores and perfect scheduling, the atomic busy-waiting overhead of each base case will be an additive $O(1)$ because each base case needs to wait for only two (2) other base cases from previous time step to start its own computation. For example, at the bottom (last two levels) of the DAC tree, the computation of a base case X_{11} only depends on base cases X_{01} and X_{10} from the same parent recursion. In Section 4 we report L1/L2 cache misses, measured by PAPI [10], incurred by our Lazy COW algorithm showing that its cache performance is comparable to that of the standard 2-way COP algorithm.

Claim 3. *Lazy algorithm for LCS aligns subtasks across different levels of the DAC tree with a conceptual wavefront (see Figure 4) that sweeps through the entire problem space along timeline t.*

Proof. Omitted due to page limitation. □

4. Experimental Evaluation

In this section we report experimental results showing how well the COW simulation techniques, i.e., Eager and Lazy recursion, perform in practice. We compared our COW algorithms with parallel loops (without tiling), blocked parallel loops (with tiling), and classic 2-way recursive DAC (COP) implementations. Note that COW technique is an algorithmic improvement over classic COP algorithm, so the natural counterpart of our COW algorithms are the classic COP algorithms. We list as well the comparison with parallel loops and blocked parallel loops just for references. The benchmark problems that are used to evaluate our techniques are given in Table 3.

We feel that programmability of the COW algorithms are not within current scope. Without proper primitive support in either nested parallel programming model or Cilk runtime system, programming of the COW algorithms are quite tricky and requires

Table 2: Machine specifications.

Name	*Intel32*	*Intel16*
System	Intel Xeon E5-4650	Intel Xeon E5-2680
Clock	2.70 GHz	2.70 GHz
#Cores	4×8	2×8
L1 data cache	32 KB	32 KB
Last-level cache	20 MB	20 MB
Memory	1 TB	32 GB
OS	CentOS 6.3	CentOS 6.3
Compiler	icc v14.0	icc v14.0

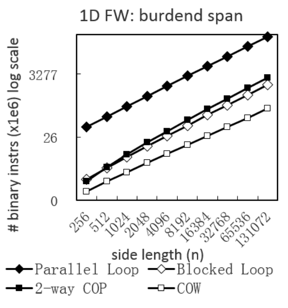
hacking into Cilk runtime system. That's also the reason why we do not give out the pseudo-code in this paper. We plan to investigate on primitive support as our future work.

Due to page limitations, we only include in this extended abstract some selected charts for 1D FW and LCS. Other charts and charts of other problems show similar patterns and trends, so are omitted.

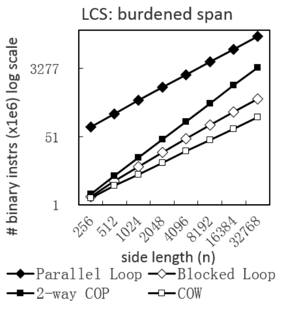
The methodology of our experiments is as follows. For a "fair" comparison, we use the same base case function throughout different algorithms of the same DP problem. For blocked loops, 2-way COP and COW, we also coarsened them to the same base case size. So the way how subtasks are scheduled becomes the main if not the only difference between COW algorithm, 2-way COP algorithm, and blocked parallel looping algorithm. To validate our claim that the COW algorithm improves parallelism without giving up cache efficiency, we measure the burdened span of all algorithms using Cilkview [34] and L1/L2 cache misses using PAPI [10]. All measurement results reported in this section are "min" of at least three (3) independent runs. We experimented on the hardware platforms listed in Table 2.

Table 3: Benchmark problems that use Eager and Lazy approaches. COP is a recursive DAC based cache-oblivious parallel algorithm, and $*$ represents this paper.

Problem	Eager recursion	Lazy recursion	2-way COP implementation
Floyd-Warshall 1D [*]	✓	–	[*]
Floyd-Warshall 2D [22, 55]	✓	–	[13]
Stencil Programs [23–25]	✓	–	[52]
LCS [15, 18]	–	✓	[15]



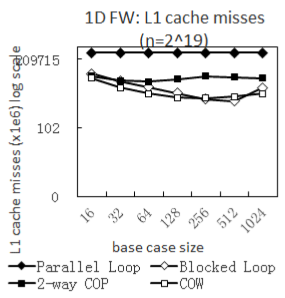
(a)



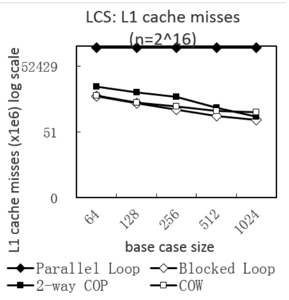
(b)

Figure 5: "Burdened span" by Cilkview of 1D FW and LCS. "Burdened span" is counted in number of binary instructions on the critical path. In figures, we fix base case size but change problem sizes.

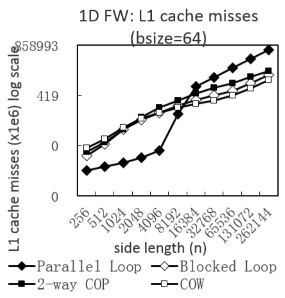
Burdened Span. Figure 5 plot the burdened span measured by Cilkview [34]. Vertical axis is "burdened span ($\times$1e6)" in log scale, and horizontal axis is the "side length n" (the problem size is then n^2) in log scale. Span is the length of critical path in the directed acyclic graph (DAG) representation of a parallel program. The



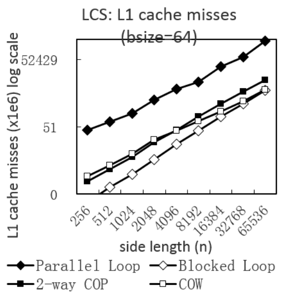
(a) Fixed problem size with variable base case sizes



(c) Fixed problem size with variable base case sizes



(b) Fixed base case size with variable problem sizes

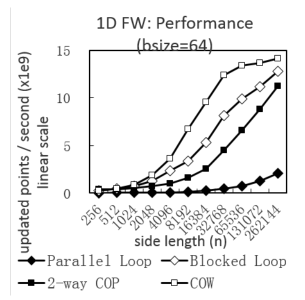


(d) Fixed base case size with variable problem sizes

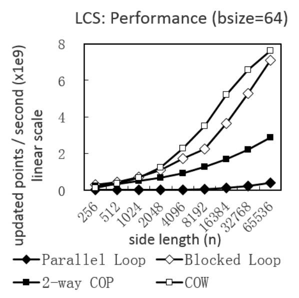
Figure 6: Comparison of cache misses (by PAPI) among parallel loops, blocked loops, 2-way COP, and COW algorithms. *bsize* is the base case size. Both vertical and horizontal axes are log scale in all sub-figures.

shorter the span, the better the parallelism an algorithm has. Burdened span is theoretical span plus scheduling overhead. In general, Figures 5a and 5b reveal that COW algorithms (solid line with empty square) always have the best span, which matches our theoretical prediction. Parallel loops (solid line with solid diamond) always have the worst burdened span because the loops are parallelized with granularity of 1, which is too fine-grain to amortize the scheduling overhead. In Figures 5a and 5b, blocked loops (solid line with empty diamond) always have shorter span than 2-way COP (solid line with solid square) because theoretical span of blocked loops for 1D FW and LCS are both $O(n)$ but theoretical span of 2-way COP for 1D FW and LCS are $O(n \log n)$ and $O\left(n^{\log_2 3}\right)$, respectively. Note that Cilkview [34] measures the span by counting the number of binary instructions on the critical path, hence the results in span charts (e.g. Figures 5a and 5b) may slightly deviate from theoretical predictions especially when the problem size is small.

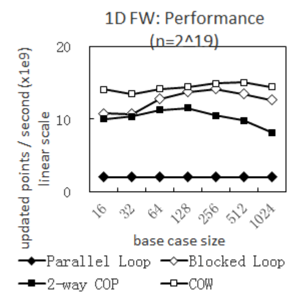
Cache Misses. Figure 6 plot the L1 cache misses measured by PAPI [10] on *Intel32*. Vertical axis is "L1 cache misses ($\times$1e6)" in log scale, and horizontal axis is "base case size" in log scale. We only show the charts of L1 cache misses. The charts of L2 cache misses show similar patterns, so are omitted. Figures 6b and 6d fixed the base case size to a small constant and varies side length, so are the problem size, to see how the L1 cache misses changes accordingly. Figures 6a and 6c, on the contrary, fixed a reasonably large problem size and varies base case sizes from small to large, to see how the L1 cache misses changes accordingly. By above two comparisons, we have a rough overview of a 3D chart of how L1 cache misses change with different problem sizes and different base case sizes for 1D FW and LCS problems. From the cache charts, we can see that blocked loops, 2-way COP, and COW algorithms have roughly the same L1 cache misses. Statistics of other



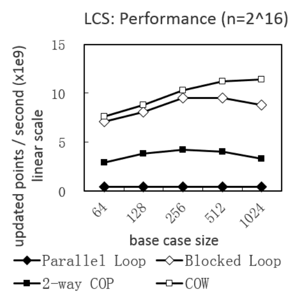
(a) Fixed base case size with variable problem sizes



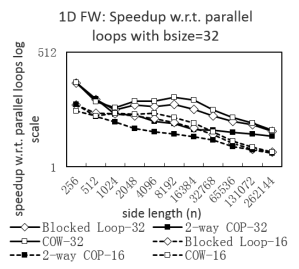
(b) Fixed base case size with variable problem sizes



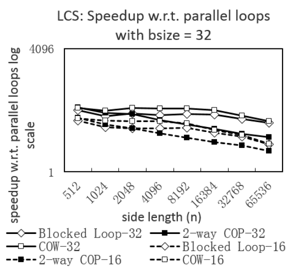
(c) Fixed problem size with variable base case sizes



(d) Fixed problem size with variable base case sizes



(e)



(f)

Figure 7: Comparison of absolute performance (updated points / second) on *Intel32* and relative performance (speedup with respect to parallel loops) on *Intel16* and *Intel32*. In all sub-figures of absolute performance, the vertical axis is linear scale and horizontal axis is log scale. In all sub-figures of relative performance, both the vertical and horizontal axes are log scale. In charts of relative performance, the suffix 32 is the speedup on *Intel32* and the suffix 16 is the speedup on *Intel16*.

problems on different machines or different level of cache show similar patterns, so are omitted.

Absolute Performance (Points Updated per Second). Figure 7 plot absolute performance (updated points/second) on *Intel32*, and relative performance (speedup w.r.t. parallel loops) on *Intel32* and *Intel16*. Similar to the methodology of showing cache charts, Figures 7a and 7b show the absolute performance (updated points / second) when we fixed base case size to a small constant and changed problem size (side length n actually) from small to large. Figures 7c and 7d show the absolute performance when we fixed a reasonably large problem size but changed base case sizes from small to large. Absolute performance were measured using the clock_gettime(CLOCK_MONOTONIC, ...) function in Linux. In figures showing absolute performance, vertical axis is "updated points/second ($\times$1e9)" in linear scale and horizontal axis is "side length (n)" in log scale. The absolute performance charts show that on *Intel32* blocked loops always have better performance than 2-way COP when using exactly the same base case function. We conjecture that on modern multi-core machine, parallelism may play a more important role in actual performance. All performance charts show that our COW algorithms beated blocked loops in almost all cases, because cache-oblivious wavefront algorithms catch up in parallelism. In Figures 7c and 7d, we can see with larger base case sizes, both COW algorithms and blocked loops tend to have better performance.

Relative Performance (Speedup w.r.t. Parallel Loops). Figures 7e and 7f plot relative performance (speedup w.r.t. parallel loops) of various algorithms on *Intel32* and *Intel16*. Vertical axis is "speedup w.r.t. parallel loops" in linear scale and horizontal axis is "side length (n)" in log scale. Suffix 32 indicates the results on *Intel32* and suffix 16 is used for results on *Intel16*. The general trend in relative performance (speedup) charts is that COW algorithms benefit when there is not enough parallelism (compared with the available number of computing cores) for classic 2-way COP algorithms. This pattern matches theoretical predictions because the key point of COW algorithms is to improve the span with approximately the same cache complexity as 2-way COP algorithms.

5. Related Work

Recursive DAC algorithms, both serial and parallel, most of them having optimal serial cache complexity have been developed, implemented, and evaluated for LCS [11, 15], pairwise sequence alignment [16], Floyd-Warshall's APSP [13], stencil computation [23–25, 52] etc.

Tiling approaches and their expressions, such as overlapped tiling, dynamic partitioning, hierarchically tiled arrays [5, 32] are heavily studied. Meng et al. [41] studies the approach of tiling plus inter-thread locality. The main difference of our work from their approach is that while we focus on a (cache, processor)-oblivious approach, Meng et al.'s approach is (processor, cache, thread)-aware. While retaining provably optimal cache complexity bounds is the main focus of our work, Meng et al.'s work does not focus on providing such theoretic guarantees.

Priority update [48] simulates a CRCW (Concurrent Read Concurrent Write) memory model when the updating operation can be prioritized to reduce memory contention among multiple writes to a single memory location. The primitive can be used to improve parallelism and overall performance of certain parallel algorithms. In all the cases we have considered, each memory location can have only one writer at any time but can have multiple readers. We still maintain the CREW (Concurrent Read Exclusive Write) memory model and doesn't require any special property from the updating operation. In addition, we are more concerned about avoiding artificial dependencies introduced by algorithm to minimize span while keeping optimum cache performance.

Maleki et al. [40] presented in their paper that certain dynamic programming problem called "Linear-Tropical Dynamic Programming (LTDP)" can possibly obtain extra parallelism based on rank convergence, a property by which the rank of a sequence of matrix products in the tropical semiring is likely to converge to 1. LTDP includes a class of important dynamic programming problems, such as Viterbi, LCS. The parallel LTDP algorithm works well in practice though the worst case is sequential. The LTDP algorithm also doesn't provide theoretical guarantee on cache-obliviousness, which is the key property of our algorithms.

Hybrid r-way DAC algorithms with different values of r at different levels of recursion have been considered in [12]. These algorithms can reach parallel cache complexity matching the best sequential cache complexity, but the algorithms then become complicated to program, processor-aware, and often cache-aware.

Current implementations of the Eager/Lazy COW simulation techniques rely on atomic operations on the data dependency path to guard the correctness of algorithms in addition to the fork-join primitives. Atomic operations are also commonly used to implement parallel task graph execution systems such as Nabbit [2], BDDT [54], etc. The key difference between COW algorithms and these task parallel graph execution systems is that these systems usually unroll the entire execution beforehand and execute all subtasks in a parallel looping fashion and as a result may lose cache efficiency. In case of a COW algorithm, recursion unfolds dynamically on-the-fly, and it inherits the recursive execution order among subtasks divided from the same parent task. So the COW algorithm retains the cache-obliviousness and cache-optimality properties of the original 2-way recursive DAC algorithm.

6. Conclusion

We have proposed two algorithmic techniques to achieve locality-preserving improvements in parallelism of standard recursive DAC-based cache-oblivious parallel DP algorithms. Our techniques work by performing the same divide-and-conquer as original COP algorithms but scheduling the execution of subtasks across different levels of DAC tree as soon as their real data dependency constraints are satisfied. The proceeding of our COW algorithms usually looks like a conceptual wavefront sweeping through a dynamically unfolded DAC tree. The resulting algorithms are called Cache-Oblivious Wavefront (COW) algorithms. They often achieve asymptotically shorter span than the original 2-way DAC based COP algorithm without losing the cache-obliviousness and cache-optimality properties of that algorithm. We have provided theoretical analyses as well as experimental results on several DP problems to validate our claims. However, COW algorithms still have the same recursive function call overhead as classic COP algorithms. In other words, while these algorithms do not need to tune for parallelism and cache efficiency, they still have to tune for base case size.

Current implementations of COW algorithms rely on atomic operations to guard the correctness of the algorithm in addition to the fork-join primitives. Atomic operations do not work well with current theory of scheduling and the nested parallel programming model. Getting rid of atomic instructions from COW algorithms without degrading the performance guarantees they provide will be one of our future research goals.

Acknowledgments

We thank Prof. Baifeng Wu and Prof. Xiaoyang Wang at Fudan University for general help on research environment. We thank Prof. Charles E. Leiserson at MIT CSAIL for coining the name of "Cache-Oblivious Wavefront". We thank all our reviewers for their insightful comments and helpful discussions. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proc. of the 12th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA 2000)*, pages 1–12, 2000.

[2] K. Agrawal, C. E. Leiserson, and J. Sukha. Executing task graphs using work stealing. In *IPDPS*, pages 1–12. IEEE, April 2010.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[4] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[5] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 48–57, New York, NY, USA, 2006. ACM.

[6] R. Bleck, C. Rooth, D. Hu, and L. T. Smith. Salinity-driven thermocline transients in a wind- and thermohaline-forced isopycnic coordinate model of the North Atlantic. *Journal of Physical Oceanography*, 22(12):1486–1505, 1992. ISSN 0022-3670.

[7] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 235–244. ACM, 2004.

[8] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 355–366, New York, NY, USA, 2011. ACM.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 1995.

[10] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. *SC Conference*, 0:42, 2000.

[11] R. Chowdhury. *Cache-efficient Algorithms and Data Structures: Theory and Experimental Evaluation*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas, 2007.

[12] R. Chowdhury and V. Ramachandran. Cache-efficient Dynamic Programming Algorithms for Multicores. In *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 207–216, 2008.

[13] R. Chowdhury and V. Ramachandran. The cache-oblivious Gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. *Theory of Computing Systems*, 47(4):878–919, 2010.

[14] R. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. *Journal of Parallel and Distributed Computing (Special issue on best papers from IPDPS 2010, 2011 and 2012)*, 73(7):911–925, 2013. A preliminary version appeared as [17].

[15] R. A. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *In Proc. of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 06*, pages 591–600, 2006.

[16] R. A. Chowdhury, H.-S. Le, and V. Ramachandran. Cache-oblivious dynamic programming for bioinformatics. *TCBB*, 7(3):495–510, July-Sept. 2010.

[17] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium*, pages 1–12, April 2010. .

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[19] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC*, pages 4:1–4:12, Austin, TX, Nov. 15–18 2008.

[20] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

[21] H. Dursun, K.-i. Nomura, W. Wang, M. Kunaseth, L. Peng, R. Seymour, R. K. Kalia, A. Nakano, and P. Vashishta. In-core optimization

of high-order stencil computations. In *PDPTA*, pages 533–538, Las Vegas, NV, July13–16 2009.

[22] R. Floyd. Algorithm 97 (SHORTEST PATH). *Commun. ACM*, 5(6): 345, 1962.

[23] M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *ICS*, pages 361–366, Cambridge, MA, June 20–22 2005.

[24] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *SPAA*, pages 271–280, 2006.

[25] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. *Theory of Computing Systems*, 45(2):203–233, 2009.

[26] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98*, pages 212–223, 1998.

[27] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–297, New York, NY, Oct. 17–19 1999.

[28] Z. Galil and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64: 107–118, 1989.

[29] Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than $O(1)$ dependency. *Journal of Parallel and Distributed Computing*, 21:213–222, 1994.

[30] R. Giegerich, C. Meyer, and P. Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.

[31] O. Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[32] J. Guo, G. Biksh, B. B. Fraguela, M. J. Garzarn, and D. Padua. Programming with tiles. In *In PPoPP 08: Proceedings of the Thirteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[33] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, 1997.

[34] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview scalability analyzer. In *SPAA*, pages 145–156, Santorini, Greece, June 13–15 2010.

[35] Intel Corporation. The Intel Many Integrated Core Architecture. `http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html`, 2011.

[36] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *MSP*, pages 36–43, Chicago, IL, June 12 2005.

[37] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC*, pages 51–60, San Jose, CA, 2006. ISBN 1-59593-578-9. . URL `http://doi.acm.org/10.1145/1178597.1178605`.

[38] J. O. S. Kennedy. Applications of dynamic programming to agriculture, forestry and fisheries: Review and prognosis. *Review of Marketing and Agricultural Economics*, 49(03), 1981.

[39] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *PLDI*, San Diego, CA, June 10–13 2007.

[40] S. Maleki, M. Musuvathi, and T. Mytkowicz. Parallelizing dynamic programming through rank convergence. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP'14, pages 219–232, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8.

[41] J. Meng, J. W. Sheaffer, and K. Skadron. Exploiting inter-thread temporal locality for chip multithreading. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010*, pages 1–12, 2010.

[42] A. Nakano, R. Kalia, and P. Vashishta. Multiresolution molecular dynamics algorithm for realistic materials modeling on parallel computers. *Computer Physics Communications*, 83(2-3):197–214, 1994. ISSN 0010-4655.

[43] A. Nitsure. Implementation and optimization of a cache oblivious lattice Boltzmann algorithm. Master's thesis, Institut für Informatic, Friedrich-Alexander-Universität Erlangen-Nürnberg, July 2006.

[44] L. Peng, R. Seymour, K.-i. Nomura, R. K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W. R. Volz, and C. C. Wong. High-order stencil computations on multicore clusters. In *IPDPS*, pages 1–11, Rome, Italy, May 23–29 2009.

[45] A. A. Robichek, E. J. Elton, and M. J. Gruber. Dynamic programming applications in finance. *The Journal of Finance*, 26(2):473–506, 1971.

[46] D. Romer. It's fourth down and what does the bellman equation say? a dynamic programming analysis of football strategy. Technical report, National Bureau of Economic Research, 2002.

[47] J. Rust. Numerical dynamic programming in economics. *Handbook of computational economics*, 1:619–729, 1996.

[48] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates. In *SPAA*, pages 152–163, 2013.

[49] H. V. Simhadri, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and A. Kyrola. Experimental analysis of space-bounded schedulers. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 30–41, New York, NY, USA, 2014. ACM.

[50] D. K. Smith. Dynamic programming and board games: A survey. *European Journal of Operational Research*, 176(3):1299–1318, 2007.

[51] A. Taflove and S. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*. Artech House, Norwood, MA, 2000. ISBN 1580530761.

[52] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir stencil compiler. In *SPAA*, San Jose, CA, USA, 2011.

[53] Y. Tang, R. A. Chowdhury, C.-K. Luk, and C. E. Leiserson. Coding stencil computation using the Pochoir stencil-specification language. In *HotPar'11*, Berkeley, CA, USA, May 2011.

[54] G. Tzenakis, A. Papatriantafyllou, H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. BDDT: block-level dynamic dependence analysis for task-based parallelism. In *Advanced Parallel Processing Technologies - 10th International Symposium, APPT 2013, Stockholm, Sweden, August 27-28, 2013, Revised Selected Papers*, pages 17–31, 2013.

[55] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.

[56] M. Waterman. *Introduction to Computational Biology*. Chapman & Hall, London, UK, 1995.

[57] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Lattice Boltzmann simulation optimization on leading multicore platforms. In *IPDPS*, pages 1–14, Miami, FL, Apr. 2008.