# POSTER: Provably Efficient Scheduling of Cache-Oblivious Wavefront Algorithms

Rezaul Chowdhury[¶] Pramod Ganapathi[¶] Yuan Tang[§] Jesmin Jahan Tithi[*]

[¶]Department of Computer Science, Stony Brook University, New York, USA.
[§]School of Software, Shanghai Key Lab of Intelligent Information Processing, Fudan University, Shanghai, China.
[*]Intel Corporation.

## Abstract

Standard cache-oblivious recursive divide-and-conquer algorithms for evaluating dynamic programming recurrences have optimal serial cache complexity but often have lower parallelism compared with iterative wavefront algorithms due to artificial dependencies among subtasks. Very recently cache-oblivious recursive wavefront (COW) algorithms have been introduced which do not have any artificial dependencies. Though COW algorithms are based on fork-join primitives, they extensively use atomic operations, and as a result, performance guarantees provided by state-of-the-art schedulers for programs with fork-join primitives do not apply.

In this work, we show how to systematically transform standard cache-oblivious recursive divide-and-conquer algorithms into recursive wavefront algorithms to achieve optimal parallel cache complexity and high parallelism under state-of-the-art schedulers for fork-join programs. Unlike COW algorithms these new algorithms do not use atomic operations. Instead, they use closed-form formulas to compute at what time each recursive function must be launched in order to achieve high parallelism without losing cache performance. The resulting implementations are arguably much simpler than implementations of known COW algorithms.

***Keywords*** recursive wavefront; cache-oblivious; parallel; divide-and-conquer; dynamic programming

## 1. Introduction

For good performance on a modern multicore machine with a cache hierarchy, algorithms must have good parallelism and should be able to use the caches efficiently at the same time. Iterative wavefront algorithms for solving *Dynamic programming (DP)* problems on these machines have optimal parallelism but often suffer due to bad cache performance. On the other hand, though standard cache-oblivious recursive divide-and-conquer DP algorithms have optimal serial cache complexity, they often have low parallelism. The tiled-iterative wavefront algorithms achieve optimality in cache complexity and achieve high parallelism but are cache-aware, and hence are not portable and do not adapt well when available cache space fluctuates during execution in a multiprogramming environment. Very recently, *cache-oblivious wavefront (COW) algorithms* (Tang et al. 2015) have been proposed that have optimal parallelism and optimal serial cache complexity. Though COW algorithms are based on fork-join primitives, they extensively use atomic operations for correctness. But current theory of scheduling nested parallel programs with fork-join primitives do not allow such atomic operations. As a result, no bounds on parallel running time and parallel cache complexity could be proved for those algorithms. Those algorithms are also very difficult to implement since they require hacking into a parallel runtime system. Extensive use of atomic locks causes too much overhead for very large and high dimensional DPs.

**Source of suboptimal parallelism in standard recursive algorithms.** Fastest iterative DP implementations have the following wavefront-like property: if an update on a DP table cell $x$ requires reading from cells $S = \{y_1, y_2, \ldots, y_s\}$, then $x$ is updated immediately after all cells in $S$ are fully updated. But a standard recursive DP algorithm may not update $x$ at the earliest possible time. Consider the standard recursive algorithm $\mathcal{A}$ for computing LCS length given in Figure 1. As per the recursive structure, the top-left quadrants of $X_{12}$ and $X_{21}$, i.e., $X_{12,11}$ and $X_{21,11}$, respectively, can only start executing when the execution of the bottom-right quadrant of $X_{11}$, i.e., $X_{11,22}$ completes. These dependencies amomg subtasks are not implied by the DP recurrence but arise from the recursive structure of the algorithm. We call these dependencies *artificial dependencies*.

**Recursive wavefront algorithms.** One can expect to improve parallelism of a standard recursive DP algorithm by removing artificial dependencies from the algorithm and allowing the smallest recursive subtasks to execute as closely to the iterative wavefront order as possible. We call such algorithms *recursive wavefront algorithms*. Indeed, such algorithms were introduced in (Tang et al. 2015), and named cache-oblivious wavefront (COW) algorithms. However, as we have already mentioned COW algorithms have several limitations that we have overcome in our current work.

In this work, we develop a generic method to schedule recursive wavefront algorithms based on timing functions. These algorithms have a structure similar to the standard

*PPoPP '17* February 04-08, 2017, Austin, TX, USA
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4493-7/17/02...$15.00
DOI: http://dx.doi.org/10.1145/3018743.3019031

recursive divide-and-conquer algorithms, but each recursive function call is annotated with start-time and end-time hints that are passed to the scheduler. Start-time (resp. end-time) is the minimum number of timesteps from the start of the program at which a task can start (resp. end) execution assuming that applying an update operation requires one timestep. The task scheduler will make sure that the algorithms are executed in a wavefront fashion using the timing functions. Indeed, the actions the scheduler is expected to take based on the timing functions is straightforward, and a programmer may choose to make some straightforward transformations of the code herself and use a scheduler that does not accept hints. The transformed code is still purely based on fork-join parallelism, and the performance bounds (e.g., parallel running time and parallel cache complexity) guaranteed by any scheduler supporting fork-join parallelism apply.

## 2. Deriving recursive wavefront algorithms

In this section, we describe how to transform a standard recursive DP algorithm into a recursive wavefront algorithm. Our transformation involves augmenting all recursive function calls with timing functions to launch them as early as possible without violating any dependency constraints implied by the DP recurrence. The timing functions are derived analytically, and do not employ locks or atomic instructions. Our transformation allows the updates to the DP table proceed in an order close to iterative wavefront, but from within the structure of a recursive algorithm. The goal is to reach the higher parallelism of an iterative wavefront algorithm while retaining the better cache performance (i.e., efficiency and adaptivity) and portability (i.e., cache- and processor-obliviousness) of a recursive algorithm.

**Transformation.** It is completed in three major steps:

**[*Construct completion-time function.*]** A closed-form formula for the timestep at which each DP cell is fully updated in wavefront order is derived based on the DP recurrence.

**[*Construct start- and end-time functions.*]** Cell completion times are used to derive closed-form formulas that give the timesteps in wavefront order at which each recursive function call should start and end execution.

**[*Derive the recursive wavefront algorithm.*]** Each recursive function call in the standard recursive algorithm is augmented with its start- and end-time functions so that it applies only the updates in any given timestep in wavefront order. We then use a variant of iterative deepening on top of this recursive algorithm to execute all timesteps efficiently.

**Example.** We will apply our method on the recursive algorithm LCS-OUTPUT-BOUNDARY given in (Chowdhury and Ramachandran, SODA'06) for computing LCS length. Let's call it algorithm $\mathcal{A}$. In LCS DP, a cell in DP table $X$ depends on three of its adjacent cells. The completion time of cell $(i, j)$ is given by $\mathfrak{C}(i, j)$, and the start-time (resp. end-time) of $A$ on DP table $X$ is given by $\mathcal{S}_{\mathcal{A}}(X)$ (resp. $\mathcal{E}_{\mathcal{A}}(X)$).

$$\mathfrak{C}(i,j) = \begin{cases} 0 & \text{if } i < 0 \,||\, j < 0 \,||\, i = j = 0, \\ \max\left(\mathfrak{C}(i-1,j), \mathfrak{C}(i,j-1), \mathfrak{C}(i-1,j-1)\right) + 1, & \text{otherwise.} \end{cases}$$

$$\mathcal{S}_{\mathcal{A}}(X) = \mathcal{E}_{\mathcal{A}}(X) = \mathfrak{C}(X) \qquad \text{if } X \text{ is a cell,}$$
$$\mathcal{S}_{\mathcal{A}}(X) = \mathcal{S}_{\mathcal{A}}(X_{11}) \text{ and } \mathcal{E}_{\mathcal{A}}(X) = \mathcal{E}_{\mathcal{A}}(X_{22}) \quad \text{if } X \text{ is not a cell.}$$

Solving, $\mathfrak{C}(i, j) = i + j$, $\mathcal{S}_{\mathcal{A}}(X) = x_r + x_c$, and $\mathcal{E}_{\mathcal{A}}(X) = x_r + x_c + 2n - 2$, where, $(x_r, x_c)$ = top-left corner of $X$.

Figure 1 shows the resulting wavefront algorithm.

```
A(X)
1. if $X$ is a cell then update $X$
2. else
3.   $\mathcal{A}(X_{11})$
4.   parallel: $\mathcal{A}(X_{12})$, $\mathcal{A}(X_{21})$
5.   $\mathcal{A}(X_{22})$
```

```
RECURSIVE-WAVEFRONT-LCS($X$)
1. $w \leftarrow 0$
2. while $w < \infty$ do
3.   $w \leftarrow \mathcal{A}_{wave}(X, w)$
```

```
$\mathcal{A}_{wave}(X, w)$
1. $v_i \leftarrow \infty$ for all $i \in [1, 4]$
2. if $X$ is an $n' \times n'$ matrix then          {for some predetermined $n'$}
3.   if $w = \mathcal{S}_{\mathcal{A}}(X)$ then $\mathcal{A}(X)$
4. else
5.   $arg_{1..4} \leftarrow \langle X_{11}, X_{12}, X_{21}, X_{22} \rangle$
6.   parallel for $i \leftarrow 1$ to 4 do
7.     if $w < \mathcal{S}_{\mathcal{A}}(arg_i)$ then $v_i \leftarrow \mathcal{S}_{\mathcal{A}}(arg_i)$
8.     elif $w \le \mathcal{E}_{\mathcal{A}}(arg_i)$ then $v_i \leftarrow \mathcal{A}_{wave}(arg_i, w)$
9. return min $v_i$ for all $i \in [1, 4]$
```

**Figure 1.** Transforming standard recursive LCS algorithm $\mathcal{A}$ into recursive wavefront algorithm RECURSIVE-WAVEFRONT-LCS.

## 3. Experimental results

We present results on LCS, the parenthesis problem, and Floyd-Warshall's APSP (FW-APSP). Table 1 compares ideal parallelism of the following algorithms computed using Intel© Cilkview scalability analyzer: $(i)$ standard 2-way recursive divide-and-conquer (`CO_2Way`), $(ii)$ recursive wavefront that switches to `CO_2Way` at some point (`wave-hybrid`), and $(iii)$ recursive wavefront that directly uses an iterative kernel (`wave`). For `wave-hybrid`, $n' = \max\{256, \text{ power of 2 closest to } n^{2/3}\}$. All implementations for parenthesis and FW-APSP switch to iterative kernels when $n = 64$, and for LCS when $n = 256$.

| Algorithm | LCS ($n = 256$K) | Parenthesis ($n = 16$K) | FW-APSP ($n = 16$K) |
|---|---|---|---|
| `CO_2Way` | 18 | 23 | 148 |
| `wave-hybrid` | 152 | 823 | 277 |
| `wave` | 510 | 1,916 | 1,404 |

**Table 1.** Projected parallelism.

We ran all implementations including original COW with atomic locks on a 2.7GHz 16-core Intel Sandy Bridge machine with 64GB RAM and two 20MB shared L3 caches. Both `wave` and `wave-hybrid` outperformed `CO_2Way` and COW in all cases. For parenthesis problem, `wave` was $2.6\times$, and `wave-hybrid` was $2\times$ faster than `CO_2Way`. For LCS the factors were $1.5\times$ and $1.7\times$, respectively, and for FW-APSP they were $1.2\times$ and $1.1\times$, respectively.

## Acknowledgments

NSF grants CCF-1439084 and CNS-1553510; XSEDE.

## References

Y. Tang, R. You, H. Kan, J. J. Tithi, P. Ganapathi, and R. A. Chowdhury. Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. In *PPoPP*, pages 205–214, 2015.