Nested Dataflow Algorithms for Dynamic Programming Recurrences with more than O(1) Dependency

Yuan Tang
School of Computer Science, Fudan University
Shanghai, P. R. China
yuantang@fudan.edu.cn

***Abstract*—Dynamic programming problems have wide applications in real world and have been studied extensively in both serial and parallel settings. In 1994, Galil and Park developed work-efficient and sublinear-time algorithms for several important dynamic programming problems based on the closure method and matrix product method. However, in the same paper, they raised an open question whether such an algorithm exists for the general GAP problem. In this paper, we answer their question by developing the first work-efficient and sublinear-time GAP algorithm based on the closure method and Nested Dataflow method. We also improve the time bounds of classic work-efficient, cache-oblivious and cache-efficient algorithms for the 1D problem and GAP problem, respectively. It remains an interesting question if we can further bound the GAP algorithm's space and cache bounds to be asymptotically optimal without sacrificing work or time bounds.**

***Keywords*-dynamic program with more than $O(1)$ dependency, closure method, cache-oblivious method, Nested Dataflow method, work-time model**

I. Introduction

Dynamic Programming (DP) is a general problem-solving technique that solves a large problem optimally by recursively breaking down into sub-problems and solving those sub-problems optimally [4]. If a problem can be solved this way, it is said to have optimal substructure. DP has wide applications in various fields from aerospace engineering, control theory, operation research, biology, economics, and computer science [5], [6], [1].

By the nature, a problem solvable by the DP technique, is usually presented by a set of recurrence equations. Equations (1) and (2) are two such examples. In solving a set of DP recurrences, it usually boils down to update every entries of a multi-dimensional table by some order. It is this serial order that prevents an efficient parallelization.

In the literature, there are two classic ways to parallelize a DP algorithm. Based on the well-known closure method and matrix product method, Galil and Park [1] gave out several work-efficient and sublinear-time algorithms for DP recurrences with more than $O(1)$ dependency, including the 1D problem [7], the parenthesis problem which computes the minimum cost of parenthesizing n elements [8], and the secondary structure of RNA without multiple loops [9]. They raised an open problem in their paper if there exists a better algorithm for the general edit distance problem when allowing gaps of insertions and deletions [5] (the GAP problem). On the other hand, Chowdhury, Le, and Ramachandran [2], [3], [10] developed a set of cache-oblivious parallel (COP) and cache-efficient algorithm for DP problems with $O(1)$ or more than $O(1)$ dependency. Their algorithms are usually work-efficient but super-linear in time. Dinh et al [11], [12] observed that the classic COP method may introduce excessive control dependency among recursively derived sub-problems, and that excessive control dependency actually increases the time bound (critical-path length of its computational DAG) of algorithm, therefore extended the method to the Nested Dataflow (ND) method. Besides the above two big classes of DP algorithms, we will discuss other related works in Sect. IV.

Problems: Noticing that Galil and Park's approach does not give an work-efficient and sublinear-time algorithm for the general GAP problem and the latest COP approach does not achieve a sublinear time bound, we present a new framework for the parallel computation of DP recurrences with more than $O(1)$ dependency based on a novel combination of the closure method and ND method. We demonstrate the framework by working on the following two problems:

P1. Given a real-valued function $\mathrm{w}(\cdot,\cdot)$, which can be computed with no memory access in $O(1)$ time, and initial value $D[0]$, compute

$$D[j] = \min_{0 \le i < j} D[i] + w(i,j) \qquad \text{for } 1 \le j \le n \tag{1}$$

This problem was called the least weight subsequence (LWS) problem by Hirschberg and Larmore [7]. We will call it *1D* problem following the convention of Galil and Park [1]. Its applications include, but not limited to, the optimum paragraph formation and finding a minimum height B-tree. We will study the 1D problem in Sect. III-A as a prerequisites and 1D simplification of the more complicated GAP problem.

P2. Given w, w', s_{ij}, which can be computed in $O(1)$ time with no memory access, and $D[0,0] = 0$, compute

$$D[i,j] = \min \begin{cases} D[i-1,j-1] + s_{ij} \\ \min_{0 \le q < j}\{D[i,q] + w(q,j)\} \\ \min_{0 \le p < i}\{D[p,j] + w'(p,i)\} \end{cases} \tag{2}$$

Shanghai Natural Science Funding (No. 18ZR1403100)

Algorithm	Work (T_1)	Time (T_∞)	Space	Cache (Q_1)
Galil and Park's final 1D [1]	$O(n^2)$	$O(\sqrt{n}\log n)$	$O(n^2)$	$O(n^2/B)$
COP 1D	$O(n^2)$	$O(n\log n)$	$O(n)$	$O(n^2/(BM))$
space-/cache-efficient ND 1D (Theorem 5)	$O(n^2)$	$O(n)$	$O(n)$	$O(n^2/(BM))$
sublinear-time ND 1D (Theorem 6)	$O(n^2)$	$O(\sqrt{n}\log n)$	$O(n^2)$	$O(n^2/B)$
Galil and Park's final GAP[1]	$O(n^4)$	$O(\sqrt{n}\log n)$	$O(n^4)$	$O(n^4/B)$
COP GAP [2], [3]	$O(n^3)$	$O(n^{\log_2 3})$	$O(n^2)$	$O(n^3/(B\sqrt{M}))$
space-/cache-efficient ND GAP (Theorem 11)	$O(n^3)$	$O(n\log n)$	$O(n^2)$	$O(n^3/(B\sqrt{M}))$
sublinear-time ND GAP (Theorem 13)	$O(n^3)$	$O(n^{3/4}\log n)$	$O(n^3)$	$O(n^3/B)$

Fig. 1: Main results of this paper, with comparisons to typical prior works.

Notation	Explanation
DP	Dynamic Programming
COP	Cache-Oblivious Parallel
ND	Nested Dataflow
n	problem dimension
p	# of cores
ϵ_i	small constant
M	cache size
B	cache line size
T_1	work
T_∞	time (span, depth, critical path length)
T_1/T_∞	parallelism
Q_1	serial cache complexity
$a \parallel b$	task b has *no* dependency on a
$a \; ; \; b$	task b has *full* dependency on a
$a \rightsquigarrow b$	task b has *partial* dependency on a

Fig. 2: Acronyms & Notation

for $0 \le i \le m$ and $0 \le j \le n$. We assume that m and n are of the same order of magnitude. This is the problem of computing the edit distance when allowing gaps of insertions and deletions [5]. We will call it *GAP* problem following the convention of Galil and Park [1]. Its applications include, but not limited to, molecular biology, geology, and speech recognition.

Our Results (Fig. 1):

1) The 1D problem:
   a) We give out a linear $O(n)$ time and optimal $O(n^2)$ work 1D algorithm (Theorem 5 in Sect. III-A), which achieves optimal $O(n)$ space and optimal $O(n^2/(BM))$ cache bounds in a cache-oblivious fashion. This result improves over the prior cache-oblivious and cache-efficient algorithm on time bound.
   b) We give out a sublinear $O(\sqrt{n}\log n)$ time and optimal $O(n^2)$ work 1D algorithm (Theorem 6 in Sect. III-A) with non-optimal $O(n^2)$ space and $O(n^2/B)$ cache bounds. This algorithm provides new insight on how to solve DP recurrences with more than $O(1)$ dependency and is a prerequisite for the more complicated GAP problem.
2) The GAP problem:
   a) We give out a superlinear $O(n\log n)$ time, optimal $O(n^3)$ work GAP algorithm (Theorem 11 in Sect. III-B), which achieves optimal $O(n^2)$ space and optimal $O(n^3/(B\sqrt{M}))$ cache bounds in a cache-oblivious fashion. This result improves over prior cache-oblivious and cache-efficient algorithm [2], [3] on time bound.
   b) we give out the first sublinear $O(n^{3/4}\log n)$ time and optimal $O(n^3)$ work algorithm for the general GAP problem (Theorem 13 in Sect. III-B) with non-optimal $O(n^3)$ space and $O(n^3/B)$ cache bounds. This result improves over Galil and Park's sublinear time algorithm [1] on work bound, thus answers their open question.

II. Theoretical Models

Parallel Model: We adopt the work-time model [13] (also known as work-span model [14]) to calculate work and time complexities. The model views a parallel computation as a Directed Acyclic Graph (DAG). Each vertex stands for a piece of computation with no parallel construct and each directed edge represents some control or data dependency between the pair of vertices. For simplicity, we count every arithmetic operation such as multiplication, addition, and comparison uniformly as an $O(1)$ operation. The model calculates an algorithm's time complexity (T_∞) by counting the number of arithmetic operations along its DAG's critical path. Work (T_1) is then the sum over all vertices, with parallelism defined as T_1/T_∞. Time (T_∞) and work (T_1) bounds characterize the running time of a parallel algorithm on infinite number and one processor(s), respectively. We call a parallel algorithm ***work-efficient*** and / or ***cache-efficient*** if its total work T_1 and / or serial cache bound Q_1 matches asymptotically that of the best serial algorithm for the same problem, respectively. Analogously, we have the notion of ***space-efficient***. We call a parallel algorithm ***sublinear-time*** if its time bound T_∞ is sublinear to the problem dimension n, i.e. $T_\infty = o(n)$.

Memory Model: By the convention of COP algorithms, we calculate only a parallel algorithm's serial cache complexity, i.e. serialize all its parallel constructs by some order as if it were run on one single computing core, in the ideal cache model [15]. This simplification is reasonable because a corresponding parallel cache complexity under a randomized work-stealing (RWS) runtime scheduler can be derived directly by $Q_p = Q_1 + O(pT_\infty M/B)$ [16], [17]. In the rest of paper, the term "cache bound (complexity)" stands for "serial cache bound (complexity)" unless otherwise specified.

The ideal cache model has an upper level cache of size M and an unbounded lower level memory. Data exchange between the upper and lower level is coordinated by an omniscient (offline optimal) cache replacement algorithm in cache line of size B. It also assumes a tall cache, i.e. $M = \Omega(B^2)$.

To accommodate parallel execution, we further assume that the lower level memory follows CREW (Concurrent Read Exclusive Write) convention [18]. Every concurrent reads from the same memory location can be accomplished in $O(1)$ time, while n concurrent writes to the same memory cell have to be serialized by some order and take $O(n)$ total time to accomplish. By Brent's theorem [19] , the above work-time model is justified.

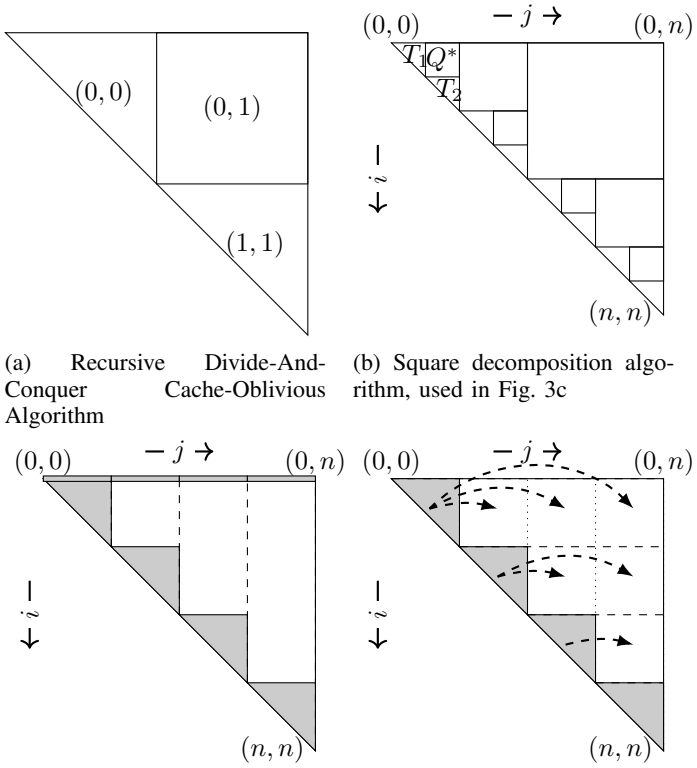
The Nested Dataflow (ND) Model: Tang et al. [11] firstly observe that classic COP method may introduce excessive control dependency among recursively derived sub-problems, which un-necessarily lengthens DAG's critical path. Dinh et al. [12] formalize an ND method based on the observation. The ND method bears some similarity to pipelining technique [13], future [20], [21], [22], [23] and / or synchronization variable [24]. But instead of explicitly chaining all vertices in a DAG beforehand, the ND method recursively , where the term "*Nested*" comes from, refines and expands the DAG on only data dependency, where the term "*Dataflow*" comes from, in a lazy fashion. Therefore, it not only shortens critical path, i.e. time bound, or equivalently maximizes parallelism, of an algorithm, but also achieves cache efficiency in a cache-oblivious fashion [15] because it keeps the recursive executing order among vertices. More discussions on the differences between the ND method and related works can be found in [12].

We describe the ND method in general as follows. The method employs a new dataflow operator "$\rightsquigarrow$" (Pronounced "Fire") to address the notion of ***partial dependency***. If we define a ***task*** as a *set* of vertices of a DAG, a partial dependency between a pair of tasks a and b , denoted by"$a \rightsquigarrow b$", indicates that a subset of subtasks of b depends on a subset of subtasks of a. That is to say, b will get notified by some signal , from either runtime system or a depending on implementation, and can start executing some subset of its subtasks when their data dependency are satisified. If an algorithm follows a divide-and-conquer framework, as many COP algorithms does, the "$\rightsquigarrow$" operator will preserve recursive executing order among subtasks of a and b to attain cache efficiency in a cache-oblivious fashion. A partial dependency will be refined, recursively if it's a recursive algorithm, by fire rules throughout computation. A partial dependency will stop refinement until both its source and sink tasks become leaf vertices (vertices with no parallel construct), in which case the "$\rightsquigarrow$" operator reduces to a classic ";" (Serial) construct. By the definition, the classic "$\|$" (Parallel) and ";" (Serial) constructs are just syntactic sugar for the two extreme cases. That is, notation "$a \parallel b$" indicates that no subtasks of b depends on any subtasks of a, i.e. *no* dependency, while "a ; b" says that all subtasks of b depend on all subtasks of a, i.e. a *full* dependency.

III. Work-Efficient and Sublinear-Time Nested Dataflow Algorithms

A. The 1D Problem

Organization: We firstly recap Galil and Park's sublinear-time algorithm [1] by Theorem 3, which serves as the foundation for later discussions; Then we address several components, i.e. a classic COP algorithm by Lemma 4, an improved ND version by Theorem 5. Finally, we present our main Theorem 6 for the 1D problem constructed from these components.



(a) Recursive Divide-And-Conquer Cache-Oblivious Algorithm

(b) Square decomposition algorithm, used in Fig. 3c

(c) The final sublinear time algorithm, using the square decomposition algorithm in Fig. 3b

(d) The ND algorithm with sublinear time and linear space bounds.

Fig. 3: Different Algorithms for the 1D problem

1) Old Work-Efficient and Sublinear-Time 1D algorithm: The following algorithm is extracted from Sect. 2 of Galil and Park's original work [1]. They firstly reduce the problem of solving the recurrences of (1) to a shortest path problem, then solve the shortest path by squaring (the closure method, i.e. general matrix multiplication on a semiring). Finally, they reduce the amount of work by the method of indirection.

The main steps are recapped as follows. It defines a matrix H as follows such that each entry $H(i, j)$ denotes the current shortest path from coordinate i to j.

$$\begin{aligned} H(i,i) &= 0 && \text{for } 0 \le i \le n \\ H(i,j) &= w(i,j) && \text{for } 0 \le i < j \le n \\ H(i,j) &= +\infty && \text{for } i > j \end{aligned} \tag{3}$$

It then defines a squaring operation on H as follows.

$$H^2(i,j) = \min_{i \le r \le j} \{H(i,r) + H(r,j)\} \quad \text{for } 0 \le i < j \le n \tag{4}$$

$H^k(i,j)$ is then the length of shortest path from i to j via at most k edges, and the closure H^* ($H^* = H^n$, according to Lemma 2 of [1]) contains the lengths of shortest paths between all pairs of coordinates. Hence, solving the 1D recurrence reduces to finding the shortest path from 0 to n, i.e. from $(0,0)$ to $(0,n)$, which is depicted by the top shaded row in Fig. 3c.

The squaring operation is basically a general matrix multiplication (MM) on a closed semiring and takes the computational overheads in Lemma 1 [14] to accomplish.

Lemma 1 ([14]): General MM of size n, i.e. an n-by-n matrix multiplies another n-by-n matrix, on a closed semiring can be computed in $O(\log n)$ time, $O(n^3)$ work, $O(n^3)$ space and $O(n^3/B)$ cache misses.

Proof: The lemma can be proved by constructing a 2-way divide-and-conquer algorithm that recursively divides a dimension-n (n-by-n-by-n) matrix multiplication (MM) into eight (8) concurrent dimension-$(n/2)$ sub-MMs by allocating temporary matrix of size n-by-n to hold intermediate results and merge the results by addition in the end of each recursion. ■

Lemma 2 (Lemma 3 of [1]): Referring to Fig. 3b, a square Q^* of size v, i.e. of size v-by-v, can be computed from two adjacent triangles T_1, T_2, and Q by $Q^* = T_1 Q T_2$ in $O(\log v)$ time, $O(v^3)$ space and work, and $O(v^3/B)$ cache bounds.

Proof: Assuming that the indices for Q^*, T_1, and T_2 are from 1 to m, i.e.

$$\begin{aligned} T_1 &= H^*(i,j) && \text{for } 1 \le i < j \le m \\ T_2 &= H^*(i,j) && \text{for } m/2 < i < j \le m \\ Q &= H(i,j) && \text{for } i \le m/2 < j \\ Q^* &= H^*(i,j) = T_1 Q T_2 && \text{for } i \le m/2 < j \end{aligned} \tag{5}$$

The semantics of Q^*'s computation stands for computing the shortest path from i to j by taking the minimum of all possible paths $[(i,p),(p,q),(q,j)]$, where (i,p) is a cell in T_1 representing the shortest path from i to p, (p,q) is a cell in Q standing for an edge from p to q, and (q,j) is a cell in T_2 representing the shortest path from q to j. Since the computation of $T_1 Q T_2$ requires two squaring operations, each of which costs $O(\log v)$ time and $O(v^3)$ work and space, and $O(v^3/B)$ cache according to Lemma 1, the lemma then follows. ■

Theorem 3 (Refinement of Theorem 2 of [1]): There is an algorithm that solves the 1D recurrences of (1) in a sublinear $O(\sqrt{n}\log n)$ time, optimal $O(n^2)$ work, $O(n^2)$ space, and $O(n^2/B)$ cache bounds.

Proof: Referring to Fig. 3c, the algorithm can be formulated as follows:

1) Dividing the top row of H^* into n/v intervals, where v is a parameter to determine later.
2) Computing n/v shaded triangles of H^* on diagonal bounded by cells (i,i), $(i,i+v)$, and $(i+v,i+v)$ for $0 \le i \le n/v - 1$, simultaneously by repeated squaring as the algorithm in Lemma 2. Observing that each top-level triangle is computed recursively in $\log v$ levels and that top-level computation dominates, the overheads to compute one top-level triangle then sum up to $O(\log^2 v)$ time, $O(v^3)$ work and space, and $O(v^3/B)$ cache misses. Summing up the overheads for n/v concurrent top-level triangles yield the costs of $O(\log^2 v)$ time, $O(nv^2)$ work and space, and $O(nv^2/B)$ cache misses.
3) Computing the top shaded row of H^* from the left-most interval to right-most in n/v iterations. For convenience, let's denote the top shaded row of H^* by f. The first interval, i.e. $f(0),\ldots,f(v)$, is given by the top row of first shaded triangle for free; For $l \in [2, n/v)$, we compute the l-th interval , i.e. $f((l-1)v+1),\ldots,f(lv)$, by the squaring of following three matrices,
   a) all previous intervals, i.e. $f(0),\ldots,f((l-1)v)$, which is a 1-by-$(l-1)v$ matrix. We call it "H^*_{-}" for convenience.
   b) the l-th interval of H, i.e. the rectangular region of H bounded by cells $(0,(l-1)v+1)$, $((l-1)v+1,(l-1)v+1)$, $((l-1)v+1,lv)$, and $(0,lv)$, which is an $(l-1)v$-by-v matrix. We call it "$H_{\square}$". Note that matrix H can be computed on-the-fly in $O(1)$ time with no memory access.
   c) the shaded triangle on the same column, i.e. the H^* region bounded by cells $((l-1)v,(l-1)v)$, $((l-1)v,lv)$, and (lv,lv), which is a v-by-v triangular matrix. We call it "$H^*_{\triangle}$".

   The first squaring of H^*_{-} with $H_{\square}$ yields an intermediate 1-by-v matrix H'_{-} in $O(\log(lv))$ time, $O(lv^2)$ space and work, and $O(lv^2/B)$ cache misses; The second squaring of the intermediate H'_{-} with $H^*_{\triangle}$ yields the final 1-by-v l-th interval in $O(\log v)$ time, $O(v^2)$ space and work, and $O(v^2/B)$ cache misses. Apparently, the first squaring dominates.

   Summing up over $l \in [2, n/v]$ iterations yields a cost of $O((n/v)\log n)$ time, $O(nv)$ space, with temporary space for squaring of Lemma 1 reused across iterations, $O(n^2)$ work, and $O(n^2/B)$ cache misses.

Summing up the overheads of the above two steps and making $v = \sqrt{n}$ yields the conclusion. ■

Though this algorithm is work-efficient, it's neither space- nor cache-efficient since a straightforward cache-oblivious algorithm requires only $O(n)$ space and incurs $O(n/B + n^2/(BM))$ cache misses.

2) Cache-Oblivious Parallel 1D Algorithm: Referring to Fig. 3a, a straightforward cache-oblivious parallel (COP) algorithm recursively divides the work into three or four quadrants depending on the shape and schedules their executing order according to the data dependencies in the granularity of *quadrants*. We denote the top-left quadrant by $(0,0)$, top-right $(0,1)$, bottom-left $(1,0)$, and bottom-right $(1,1)$. Referring

to the pseudo-code in Fig. 4a, the serialization between the computation of A_{00} and A_{01} is because the computation of A_{01} requires the results of A_{00} as input. If an algorithm does not allocate temporary space, the computation of A_{11} has to lay behind A_{01} by our CREW assumption because they output to the same region. A similar analysis applies to the pseudo-code in Fig. 4c.

Lemma 4: There is a COP algorithm that solves the 1D recurrences of (1) in $O(n \log n)$ time, optimal $O(n^2)$ work, optimal $O(n)$ space, and optimal $O(n/B + n^2/(BM))$ cache bounds.

Proof: The COP algorithm is given in Fig. 3a and we have following recurrences for its time and cache complexity, respectively. The subscripts of $\triangle$ and $\square$ in the recurrences stand for the COP-1D$_\triangle$ and CO-1D$_\square$ algorithms in Figs. 4a and 4c respectively. Equation (10) is the stop condition of cache bound. The recurrences solve to $T_{\infty,\triangle}(n) = O(n \log n)$ and $Q_{1,\triangle}(n) = O(n/B + n^2/(BM))$, with an optimal $O(n)$ space bound because the algorithm does not use temporary space.

$$T_{\infty,\triangle}(n) = 2T_{\infty,\triangle}(n/2) + T_{\infty,\square}(n/2) \quad (6)$$
$$T_{\infty,\square}(n) = 2T_{\infty,\square}(n/2) \quad (7)$$
$$Q_{1,\triangle}(n) = 2Q_{1,\triangle}(n/2) + Q_{1,\square}(n/2) \quad (8)$$
$$Q_{1,\square}(n) = 4Q_{1,\square}(n/2) \quad (9)$$
$$Q_{1,\triangle}(n) = Q_{1,\square}(n) = O(n/B) \quad \text{if } n \le \epsilon_6 M \quad (10)$$

■

COP-1D$_\triangle(A)$
1 COP-1D$_\triangle(A_{00})$;
2 CO-1D$_\square(A_{01}, A_{00})$;
3 COP-1D$_\triangle(A_{11})$;
4 **return**

(a) The COP algorithm for a triangular region

$\oplus \overset{\triangle\square}{\rightsquigarrow} \ominus = \{$
$\oplus_{00} \overset{\triangle\square}{\rightsquigarrow} \{\ominus_{00}, \ominus_{01}\}$
$, \oplus_{11} \overset{\triangle\square}{\rightsquigarrow} \{\ominus_{10}, \ominus_{11}\}\}$

(b) The fire rule of ND algorithm

CO-1D$_\square(A, B)$
1 CO-1D$_\square(A_l, B_l)$
  $\|$ CO-1D$_\square(A_r, B_r)$;
2 CO-1D$_\square(A_l, B_r)$
  $\|$ CO-1D$_\square(A_r, B_l)$;
3 **return**

(c) The COP algorithm for a rectangular region

ND-1D$_\triangle(A)$
1 ND-1D$_\triangle(A_{00}) \overset{\triangle\square}{\rightsquigarrow}$
  CO-1D$_\square(A_{01}, A_{00})$;
2 ND-1D$_\triangle(A_{11})$;
3 **return**

(d) The ND algorithm for a triangular region

Fig. 4: Pseudo-codes of cache-oblivious 1D algorithms

3) Nested Dataflow 1D algorithm: We can improve the time bound of above COP algorithm by refining the data dependency of COP-1D$_\triangle$ algorithm recursively by the ND method as shown in Fig. 4d. The "$\overset{\triangle\square}{\rightsquigarrow}$" construct in figure indicates a partial dependency between the source and sink tasks, which is specified by the fire rule in Fig. 4b. The $\oplus$ and $\ominus$ notation are wildcards to match at runtime source and sink tasks respectively. The fire rule says that the $(0,0)$, $(0,1)$ subtasks nested in the same sink task only partially depends on the $(0,0)$ subtask of the source, respectively the $(1,0)$ and $(1,1)$ subtasks of the same sink partially depends on the $(1,1)$ subtask of the source. The partial dependences will then be recursively refined by the same rule until base cases where the "$\overset{\triangle\square}{\rightsquigarrow}$" construct will reduce to a ";" construct.

Theorem 5: There is an ND algorithm that solves the 1D recurrences of (1) in $O(n)$ time, optimal $O(n^2)$ work, $O(n)$ space and $O(n^2/(BM))$ cache bounds.

Proof: Referring to Fig. 4d, the ND algorithm just reschedules subtasks to run as soon as their input data are ready so that it does not use more space or incur more cache misses than the classic COP counterpart in Fig. 4a. Its new time recurrences are as follows, which solve to $O(n)$.

$$T_{\infty,\triangle}(n) = T_{\infty,\overset{\triangle\square}{\rightsquigarrow}}(n/2) + T_{\infty,\triangle}(n/2) \quad (11)$$
$$T_{\infty,\overset{\triangle\square}{\rightsquigarrow}}(n) = 2T_{\infty,\overset{\triangle\square}{\rightsquigarrow}}(n/2) \quad (12)$$
$$T_{\infty,\overset{\triangle\square}{\rightsquigarrow}}(1) = T_{\infty,\triangle}(1) + T_{\infty,\square}(1) \quad (13)$$

■

Theorem 6: There is an ND algorithm that solves the 1D recurrences of (1) in $O(\sqrt{n} \log n)$ time, optimal $O(n^2)$ work, $O(n^2)$ space and $O(n^2/B)$ cache bounds.

Proof: We construct the algorithm based on following observations.

1) The $\overset{\triangle\square}{\rightsquigarrow}$ construct in (12) will invoke some kernel functions to compute triangular and rectangular shape regions respectively when the recursion goes down to base cases.
2) Galil and Park's algorithm in Theorem 3 computes a triangular shape region in sublinear time and optimal work.
3) By using extra temporary space, i.e. like the general MM algorithm in Lemma 1, we can have a sublinear-time and optimal-work algorithm to compute rectangular shape regions as well.
4) Combining the two sublinear-time and optimal-work kernel functions for triangular and rectangular shape regions respectively, we can stop the recursion of (12) earlier.

We allocate temporary space for the CO-1D$_\square$ algorithm in Fig. 4c and have following SUB-1D$_\square$ algorithm as shown in Fig. 5.

In Fig. 5, lines 4–5 parallelizes the task of updating A from B to four subtasks by allocating temporary space of A', which is of the same size of A. The ";" construct on line 5 is a synchronization operation so that the overall task can not proceed until all four concurrent subtasks are done. Note that the merge by addition on line 7 can be parallelized and accomplished in $O(1)$ time if counting only data dependency. The recurrence for SUB-1D$_\square$'s time bound is then revised to (14), which solves to $O(\log n)$.

$$T_{\infty,\square}(n) = T_{\infty,\square}(n/2) + O(1) \quad (14)$$

Supplying this SUB-1D$_\square$ for rectangular shape regions and Galil and Park's final algorithm for triangular shape regions

SUB-1D$_\square(A, B)$

1 // Update region A from B
2 // Allocate temporary space
3 $A' \leftarrow$ alloc(sizeof(A))
4 SUB-1D$_\square(A_l, B_l) \parallel$ SUB-1D$_\square(A_r, B_r)$
5 $\parallel$ SUB-1D$_\square(A'_l, B_r) \parallel$ SUB-1D$_\square(A'_r, B_l)$;
6 // Merge A' to A by addition
7 $A = A + A'$
8 free(A')
9 **return**

Fig. 5: A Sublinear-Time and Optimal-Work algorithm for the rectangular shape region of 1D problem

to stop the recursion of (12) earlier, we have following stop condition of (15) to replace (13), where v is a parameter to be determined later.

$$T_{\infty, \overset{\triangle\square}{\rightsquigarrow}}(v) = T_{\infty,\triangle}(v) + T_{\infty,\square}(v) \tag{15}$$

$$T_{\infty,\triangle}(v) = O(\sqrt{v}\log v) \tag{16}$$

$$T_{\infty,\square}(v) = O(\log v) \tag{17}$$

$T_{\infty, \overset{\triangle\square}{\rightsquigarrow}}(v)$ then solves to $O(\sqrt{v}\log v)$. Supplying this result into (12), we have $T_{\infty,\triangle}(n) = O(n/v \cdot \sqrt{v}\log v) + T_{\infty,\triangle}(n/2) = O(n/\sqrt{v}\log v)$. By making $v = n$, the time bound solves to $O(\sqrt{n}\log n)$. It's easy to verify that SUB-1D$_\square$ has the same space and cache bounds as Galil and Park's final algorithm for triangular regions (Theorem 3). The overall space and cache bounds then are a simple summation over all regions. ∎

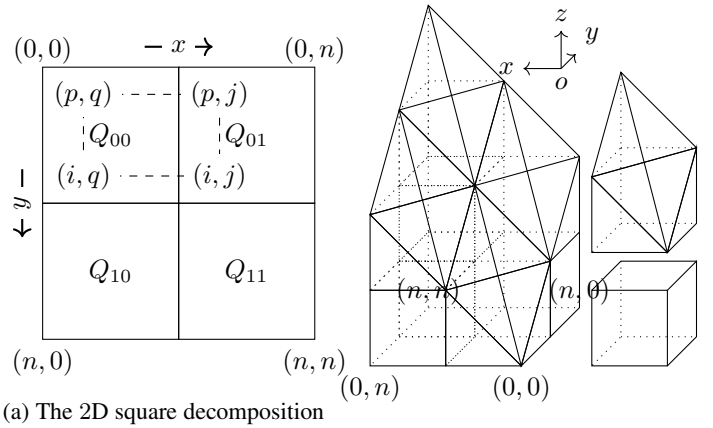
Discussions: Our result of Theorem 5 improves over prior cache-oblivious and cache-efficient algorithm of Lemma 4 on time bound without sacrificing work, space, and cache efficiency. Our result of Theorem 6 gives out another dimension of tradeoff by reducing further time bound to be sublinear but at the cost of increasing the space and cache bounds a bit , actually still be asymptotically the same as that of Galil and Park's final algorithm (Theorem 3). Moreover, Theorem 6 provides new insights into solving DP recurrences with more than $O(1)$ dependency and will show its power in solving the GAP problem in Sect. III-B.

B. The GAP Problem

Organization: We firstly recap Galil and Park's final algorithm by Theorem 9, which serves as the foundation for later discussion; Then we address the classic COP algorithm developed by Chowdhury and Ramachandran [2], [3], as well as its ND improvement by Theorem 11; Finally, we combine all components to yield the main Theorem 13.

C. Sublinear-Time but Non-Work-Efficient GAP algorithm

The key of Galil and Park's sublinear-time algorithm (Sect. 4 of [1]) is similar to that of the 1D algorithm (Sect. III-A1 of this paper). That is, firstly, they reduce the GAP problem to a shortest path problem, then solve the shortest path problem by the closure method. Finally, they reduce the amount of total work by indirection.



(a) The 2D square decomposition algorithm (b) The work of GAP

Fig. 6: The GAP Problem

The main steps as follows are essentially a 2D version of the closure method for the 1D problem in Sect. III-A1. Firstly, it defines matrix H as follows such that each entry $H(p, q, i, j)$, where $p < i$ and $q < j$, is the current shortest path from cell (p, q) to (i, j) via one intermediate cell (p, j) or (i, q) as shown in Fig. 6a. $H(\cdot, \cdot, \cdot, \cdot)$ can be viewed as a combination of two independent matrices like the one defined by (3) for the 1D problem, and is a 4D array by itself.

$$\begin{aligned} &H(i,j,i,j) = 0 && 0 \le i,j \le n \\ &H(p,q,i,j) = w(q,j) + w'(p,i) && p < i-1 \wedge q < j-1 \\ &H(p,q,i,j) = +\infty && \text{for all others} \end{aligned} \tag{18}$$

$$H(i-1, j-1, i, j) = \min\{s_{ij}, w(j-1, j) + w'(i-1, i)\}$$

It then defines a squaring operation on H as follows.

$$H^2(p,q,i,j) = \min_{\substack{p \le s \le i \\ q \le r \le j}} \{H(p,q,s,r) + H(s,r,i,j)\} \tag{19}$$

Lemma 7: The squaring operation of (19) on two 4D matrices of dimension n as defined by (18) costs $O(\log n)$ time, $O(n^6)$ work and space with $O(n^6/B)$ cache misses.

Proof: Referring to (19), since the squaring operation chooses the two coordinates of intermediate cell (s, r) independently from p to i and q to j, the update to any cell requires a $\min$ operation on $O(n^2)$ intermediate results of $+$ operations. By allocating temporary space like that of Lemma 1, it takes $O(\log n)$ time, $O(n^2)$ work and space with $O(n^2/B)$ cache misses to update one cell. Summing up over the $O(n^4)$ cells of H yields the conclusion. ∎

By Lemma 2 in [1], $H^* = I + H + H^2 + \cdots + H^n = H^n$ is then the solution to the GAP recurrence of (2) and can be computed by $\log n$ repeated squaring on H. By Lemma 7, this straightforward computation takes totally $O(\log^2 n)$ time,

$O(n^6 \log n)$ work, $O(n^6)$ space, and $O(n^6 \log n/B)$ cache misses by reusing temporary space across repeated squarings.

Following Lemma 8 reduces the total work by a 2D reduction technique (The 1D version is in the proof of Lemma 2).

Lemma 8: There is a 2D square decomposition algorithm that computes the GAP recurrences of (2) in $O(\log^2 n)$ time, $O(n^6)$ work and space, and $O(n^6/B)$ cache misses.

Proof: Referring to Fig. 6a, the algorithm computes the GAP recurrence by recursively decomposing the 2D region of H (a 2D projection of H) into four quadrants and computes H^* bottom up from the smallest squares to the largest in $\log n$ levels. At any level k, H^* of $(n/2^k)^2$ squares of size 2^k are computed as follows. To compute a square Q from four quadrants Q_{00}, Q_{01}, Q_{10}, and Q_{11}, the algorithm firstly computes $H^*_{00,01} = H^*_{00} H_{00,01} H^*_{01}$ and $H^*_{10,11} = H^*_{10} H_{10,11} H^*_{11}$ simultaneously, where $H_{00,01}$ stands for the joint H matrix striding quadrants Q_{00} and Q_{01}, and so on. Finally $H^* = H^*_{00,01} H H^*_{10,11}$, where H and H^* are over the entire region striding all four quadrants. By Lemma 7, a square of size 2^k can thus be computed in $O(k)$ time, $O(2^{6k})$ work and space , and $O(2^{6k}/B)$ cache misses. Since a square's dimension at a higher level is geometrically larger than the one at a lower level, the work, space, and cache bounds of the top-level H^* computation then dominates. The conclusion then follows. ■

Theorem 9: The final sublinear-time GAP algorithm in [1] takes $O(\sqrt{n} \log n)$ time, $O(n^4)$ work and space , and $O(n^4/B)$ cache misses.

Proof: Let $f(i, j)$ be the length of the shortest path from $(0, 0)$ to (i, j), their algorithm works as follows.

1) It computes H^* of the squares from bottom up until level k such that $2^k = v$, where v is a parameter to be determined later. Each square of $H^*(i, j, i+v, j+v)$ contains all pairs of shortest paths within the 2D region bounded between cell (i, j) and $(i + v, j + v)$. By Lemma 8, each square's computation takes $O(\log^2 v)$ time, $O(v^6)$ work and space, with $O(v^6/B)$ cache misses. Since there are $O(n^2/v^2)$ of them, which can be computed simultaneously , the costs of this step sum up to $O(n^2 v^4)$ work and space, with $O(n^2 v^4/B)$ cache misses.
2) It computes $f(i, j)$ from cell $(0, 0)$ to (n, n) at step size of v by backward diagonal, i.e. $i + j$ is constant, in $2n/v$ iterations. Squares on the same backward diagonal are computed simultaneously in the same iteration. As in the 1D version (refer to the proof of Theorem 3), computing any square $f(i, j)$ on the l-th iteration, i.e. $i + j = lv$, requires squaring of following three matrices.
   a) All previous squares of $f(0, 0) .. f(i-v, j-v)$, which can be viewed as a combination of two independent 1-by-$(l-1)v$ matrices. We denote it by a $(1 \times (l-1)v)^2$ matrix for convenience.
   b) The H values of l-th iteration, which can be viewed as a combination of two independent $(l-1)v$-by-v matrices. We denote it by a $((l-1)v \times v)^2$ matrix.
   c) The square of level-k H^* that is computed in above step (1) and is on the l-th iteration, which can be viewed as a combination of two independent v-by-v triangular matrices. We denote it by a $(v \times v)^2$ matrix.

   As in the 1D case, the squaring of $(1 \times (l-1)v)^2$ matrix with $((l-1)v \times v)^2$ matrix dominates. By Lemma 7, the squaring of one $f(i, j)$ takes $O(\log lv)$ time, $O(l^2 v^4)$ work and space, with $O(l^2 v^4/B)$ cache misses.

   Summing over the n/v iterations for n^2/v^2 squares of f, the costs of this step are $O(n/v \log n)$ time, $O(n^4)$ work and space, and $O(n^4/B)$ cache misses.

Summing up the overheads of the two steps and making $v = \sqrt{n}$ yields the bounds. ■

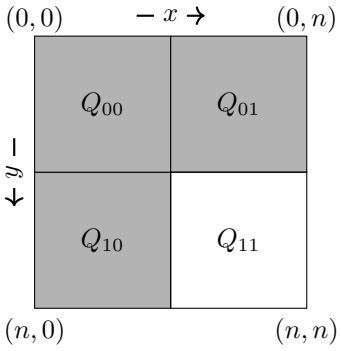
D. Cache-Oblivious Parallel GAP algorithm

```
COP-GAP△(A)
1  COP-GAP△(A00)
2  ; ((COP-GAP-H□(A01, A00)
       ; COP-GAP△(A01))
3    ‖ (COP-GAP-V□(A10, A00)
       ; COP-GAP△(A10)))
4  ; COP-GAP-V□(A11, A01)
5  ; COP-GAP-H□(A11, A10)
6  ; COP-GAP△(A11)
7  return
```

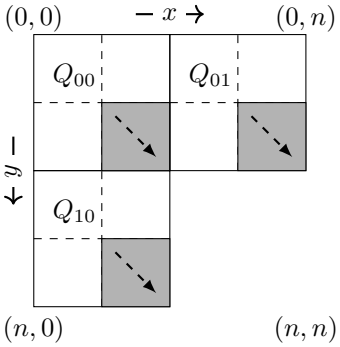
(a) The COP algorithm for a triangular region A.

```
COP-GAP-H□(A, B)
1  (COP-GAP-H□(A00, B00)
2     ; COP-GAP-H□(A00, B01))
3  ‖ (COP-GAP-H□(A01, B00)
4     ; COP-GAP-H□(A01, B01))
5  ‖ (COP-GAP-H□(A10, B10)
6     ; COP-GAP-H□(A10, B11))
7  ‖ (COP-GAP-H□(A11, B10)
8     ; COP-GAP-H□(A11, B11))
9  return
```

(b) The COP algorithm to update a rectangular region A from B.



(c) The ND algorithm



(d) The ND algorithm to compute a Γ-shape region.

Fig. 7: Pseudo-code of cache-oblivious GAP algorithms

Chowdhury and Ramachandran [2], [3] devised a cache-oblivious parallel (COP) algorithm for the GAP recurrences of (2). The algorithm separates the updates to any quadrant to two functions, one is an update from cells within the same quadrant, and the other is an update from a disjoint quadrant either horizontally or vertically. The COP-GAP$_\triangle$ function in Fig. 7a is the self-updating function, the computational shape (total work) of which is a 3D triangular analogue as shown in the upper-right corner of Fig. 6b. The COP-GAP-H$_\square(A, B)$ function in Fig. 7b is to update quadrant A from a disjoint quadrant B horizontally, the computational shape of which is a 3D cube as shown in the bottom-right corner of Fig. 6b. The update from a vertical direction is similar thus omitted.

Lemma 10: The COP algorithm in Fig. 7a computes the GAP recurrences of (2) in $O(n^{\log_2 3})$ time, optimal $O(n^2)$ space, optimal $O(n^3)$ work, and optimal $O(n^3/(B\sqrt{M}))$ cache bounds [2], [3].

Proof: **Space bound:** Since different functions updating the same quadrant are explicitly separated by synchronizations, it's easy to see that it uses no more space than the input 2D array of A.

Time and cache bound: The algorithm has the following recurrences for time and cache bounds, which solves to $O(n^{\log_2 3})$ and $O(n^3/(B\sqrt{M}))$, respectively.

$$
\begin{aligned}
T_{\infty,\triangle}(n) &= 3T_{\infty,\triangle}(n/2) + 3T_{\infty,\square}(n/2)\\
T_{\infty,\square}(n) &= 2T_{\infty,\square}(n/2)\\
Q_{1,\triangle}(n) &= 4Q_{1,\triangle}(n/2) + 4Q_{1,\square}(n/2)\\
Q_{1,\square}(n) &= 8Q_{1,\square}(n/2)\\
Q_{1,\triangle}(n) &= Q_{1,\square}(n) = O(n^2/B) \quad \text{if } n^2 \le \epsilon_8 M
\end{aligned}
$$

■

E. Nested Dataflow GAP algorithm

Theorem 11: There is an ND algorithm that solves the GAP recurrences of (2) in $O(n \log n)$ time, optimal $O(n^3)$ work, optimal $O(n^2)$ space, and optimal $O(n^3/(B\sqrt{M}))$ cache bounds.

Proof: Let's label all the quadrants recursively as in Fig. 7c. Figure 7c is the 2D projection of a 3D triangular analogue in Fig. 6b from its 3D $o - xyz$ space to the $o - xy$ plane. We have an observation that except the last (11) quadrant, i.e. $Q_{11,11,\ldots,11}$, all (11) quadrants nested in a recursion have the same amount of computation as the (00) quadrants of some of its parent's siblings, i.e. the quadrants that lie on the same backward diagonal (i.e. $x+y$ is constant). For instances, the (11) quadrant of Q_{00}, i.e. $Q_{00,11}$, has the same amount of computation as the (00) quadrants of some of its parent's siblings, i.e. $Q_{01,00}$ and $Q_{10,00}$; $Q_{01,11}$ and $Q_{10,11}$ have the same amount of computation as $Q_{11,00}$, and so on. Moreover, these quadrants on the same backward diagonal do not have any data dependencies among each other so can be scheduled to run in parallel. By this observation, except the $Q_{11,11,\ldots,11}$ quadrant, all other (11) quadrants can be pushed one level up in the recursion and run simultaneously with some of its parent's siblings as shown in Figs. 7c and 7d. And this execution pattern proceeds recursively.

The pseudo-code of ND-GAP$_\triangle$ algorithm is in Fig. 8a. The key improvement over the classic COP algorithm comes from associating every COP-GAP$_\square$ subtask with its source COP-GAP$_\triangle$ by a partially parallel "$\rightsquigarrow$" construct. The partially parallel $\rightsquigarrow$ construct will then be refined recursively by fire rule throughout computation. By fine-grain interleaving of data dependency across borders of recursion, each base case of COP-GAP$_\square$ can start computing as soon as corresponding source subtask produces the data. At the same time, the recursive executing order among subtasks is preserved. The interleaving just allows some subtasks at a higher level of recursion to start executing earlier. The computation frontier proceeds by a 2D plane whose projection on the $o - xy$ plane aligns with some backward diagonal, i.e. $x + y$ is constant. Referring to Fig. 7c, the projection of proceeding plane on the bottom $o - xy$ plane sweeps from cell $(0, 0)$ to (n, n) by backward diagonals. By contrast, a COP-GAP-H/V$_\square$ computation in Fig. 7a has to wait until the entire COP-GAP$_\triangle$ at the same recursion level finishes. That is, it introduces excessive control dependency to subtasks at lower levels.

ND-GAP$_\triangle(Q)$
1 ND-GAP$_\Gamma(Q_{00,01,10})$
  $\overset{\Gamma\square}{\rightsquigarrow}$ COP-GAP$_\square(Q_{11})$;
2 ND-GAP$_\triangle(Q_{11})$;
3 **return**

(a) The ND algorithm

$\oplus \overset{\Gamma\square}{\rightsquigarrow} \ominus = \{$
$\oplus_{00} \overset{\triangle\square}{\rightsquigarrow}_{H/V} \{\oplus_{01}, \oplus_{10}\}$
$, \{\oplus_{01}, \oplus_{10}\} \overset{\triangle\square}{\rightsquigarrow}_{H/V} \{\ominus\}\}$

(b) The fire rule of "$\overset{\Gamma\square}{\rightsquigarrow}$" computation

$\oplus \overset{\triangle\square}{\rightsquigarrow}_H \ominus = \{$
$\{\oplus_{00} \overset{\triangle\square}{\rightsquigarrow}_H \{\oplus_{01}, \oplus_{10}$
$, \ominus_{00}, \ominus_{01}\}$
$, \oplus_{01} \overset{\triangle\square}{\rightsquigarrow}_H \{\ominus_{00}, \ominus_{01}\}$
$, \oplus_{10} \overset{\triangle\square}{\rightsquigarrow}_H \{\ominus_{10}, \ominus_{11}\}\}$
$, \{\oplus_{11} \overset{\triangle\square}{\rightsquigarrow}_H \{\ominus_{10}, \ominus_{11}\}\}\}$

(c) The fire rule of "$\overset{\triangle\square}{\rightsquigarrow}$" computation

Fig. 8: Pseudo-code of ND GAP algorithm

Referring to Fig. 8, the ND algorithm proceeds as follows. Figure 8a says that the computation of a GAP recurrence without external dependency is accomplished by a partially parallel $\overset{\Gamma\square}{\rightsquigarrow}$ computation followed by a recursion on a geometrically smaller Q_{11} quadrant. Note that the partially parallel $\overset{\Gamma\square}{\rightsquigarrow}$ computation not only computes the Γ shape region of $Q_{00,01,10}$ but also invokes COP-GAP$_\square$ function (Fig. 7b) to update Q_{11} quadrant with the data from the Γ shape region. The big partially parallel $\overset{\Gamma\square}{\rightsquigarrow}$ computation will then be refined by the fire rule in Fig. 8b to two wavefronts that comprise four geometrically smaller $\overset{\triangle\square}{\rightsquigarrow}$ computations. The two wavefronts execute one after another as follows. Referring to Fig. 7d [1] , it firstly executes the partially parallel computation of $Q_{00} \overset{\triangle\square}{\rightsquigarrow}_{H/V} \{Q_{01}, Q_{10}\}$ (the first wavefront), [2] followed by $\{Q_{01}, Q_{10}\} \overset{\triangle\square}{\rightsquigarrow}_{H/V} Q_{11}$ (the second wavefront) [3]. The partially parallel computations of the first wavefront share the same source and will have their sink tasks executed simultaneously. By contrast, the second wavefront executes two distinct source tasks (ND-GAP$_\triangle$) concurrently, while sink tasks (COP-GAP$_\square$) serially because the two sinks write to the same output region of Q_{11}. After the second wavefront, Q_{11} quadrant will have been updated with the data from the Γ shape

[1] The $\oplus$ and $\ominus$ in Figs. 8b and 8c are wildcards that will match at runtime to the source and sink subtasks of corresponding $\rightsquigarrow$ construct, respectively.

[2] This is a shorthand for two partially parallel computations of $Q_{00} \overset{\triangle\square}{\rightsquigarrow}_H Q_{01}$ and $Q_{00} \overset{\triangle\square}{\rightsquigarrow}_V Q_{10}$.

[3] Similarly, this is a shorthand for two partially parallel computations of $Q_{01} \overset{\triangle\square}{\rightsquigarrow}_V Q_{11}$ and $Q_{10} \overset{\triangle\square}{\rightsquigarrow}_H Q_{11}$

region of $Q_{00,01,10}$ and can start a recursive GAP computation of ND-GAP$_\triangle$ on itself without external data dependency. The $\overset{\triangle\square}{\rightsquigarrow}$ construct recursively refines the partially parallel invocation of ND-GAP$_\triangle$ (3D triangular analogue) and COP-GAP$_\square$ (3D cube) to two parallel steps. There is a synchronization between the two parallel steps to guard correctness. Figure 8c shows the horizontal refinement , and the vertical direction is similar. Referring to Figs. 7d and 8c, $Q_{00} \overset{\triangle\square}{\rightsquigarrow} Q_{01}$ is refined to $Q_{00,00} \overset{\triangle\square}{\rightsquigarrow} \{Q_{00,01}, Q_{00,10}, Q_{01,00}, Q_{01,01}\}$ for the first parallel step and $\{Q_{00,01}, Q_{00,10}\} \overset{\triangle\square}{\rightsquigarrow} \{Q_{00,00}, Q_{00,01}, Q_{00,10}\}$ for the second parallel step. The rule of $\{\oplus_{11} \overset{\triangle\square}{\rightsquigarrow}_H \{\ominus_{10}, \ominus_{11}\}\}$ in Fig. 8c is a nested $\overset{\triangle\square}{\rightsquigarrow}$ computation on a geometrically smaller (11) quadrant, which will run concurrently with some of its parent's siblings on the same backward diagonal.

By the ND method, we have following recurrences for time bound.

$$T_{\infty,\text{ND-GAP}_\triangle}(n) = T_{\infty,\text{ND-GAP}_{\underset{\rightsquigarrow}{\Gamma\square}}}(n) + T_{\infty,\text{ND-GAP}_\triangle}(n/2) \quad (20)$$

$$T_{\infty,\text{ND-GAP}_{\underset{\rightsquigarrow}{\Gamma\square}}}(n) = T_{\infty,\text{ND-GAP}_{\underset{\rightsquigarrow}{\triangle\square}}}(n/2) + \max\{T_{\infty,\text{ND-GAP}_\triangle}(n/4), T_{\infty,\text{ND-GAP}_{\underset{\rightsquigarrow}{\triangle\square}}}(n/2)\} + T_{\infty,\text{COP-GAP}_\square}(n/2) \quad (21)$$

$$T_{\infty,\text{COP-GAP}_\square}(n) = 2T_{\infty,\text{COP-GAP}_\square}(n/2) \quad (22)$$

$$T_{\infty,\text{ND-GAP}_{\underset{\rightsquigarrow}{\triangle\square}}}(n) = 2T_{\infty,\text{ND-GAP}_{\underset{\rightsquigarrow}{\triangle\square}}}(n/2) = O(n) \quad (23)$$

$$T_{\infty,\text{ND-GAP}_{\underset{\rightsquigarrow}{\triangle\square}}}(1) = T_{\infty,\text{ND-GAP}_\triangle}(1) + T_{\infty,\text{ND-GAP}_\square}(1) \quad (24)$$

The $\max\{\ldots\}$ term of (21) says that the recursively nested (11) quadrant will be executed concurrently with the second wavefront. Referring to Fig. 7d, these are the shaded (11) quadrants. Since these (11) quadrants are geometrically smaller, their execution will be completely subsumed in the $\max\{\ldots\}$, hence will not affect the critical-path length. The last $T_{\infty,\text{COP-GAP}_\square}(n/2)$ term of (21) says that there are two partially parallel $\overset{\triangle\square}{\rightsquigarrow}$ computations of the second wavefront to update the same output quadrant (Q_{11}) so that one update has to lay behind. Note that their source tasks have no dependency on each other, thus can still execute concurrently. Equation (22) recursively expands the computation of COP-GAP$_\square$ without using any more space than inputs and output. Equation (24) says that the recursive refinement of $\overset{\triangle\square}{\rightsquigarrow}$ computation will stop and reduce to classic serial construct (";") at base cases. The time recurrences solve to $O(n \log n)$, which is asymptotically better than the classic COP algorithm in Lemma 10.

Since the ND algorithm just schedules COP-GAP$_\square$ (3D cubes) to run as soon as its sources (3D triangular analogues) produce the data without changing the recursive executing order, its work, space, and cache bounds do not change from those of Lemma 10. ■

F. Work-Efficient and Sublinear-Time GAP Algorithm

Referring to Fig. 7b, we can see that the computation of 3D cubes by COP-GAP$_\square$ can be parallelized in the same way as that of Lemma 1 by using temporary space. We then have following Lemma 12.

Lemma 12: We have a sublinear-time algorithm to compute the 3D cubes of dimension-n in Fig. 6b in $O(\log n)$ time, $O(n^3)$ work and space, and $O(n^3/B)$ cache bounds.

Combining Galil and Park's sublinear-time algorithm (Theorem 9) with the ND method of Theorem 11 for the 3D triangular analogues, and Lemma 12 for the 3D cubes, we have the first work-efficient and sublinear-time GAP algorithm as follows.

Theorem 13: There is an ND algorithm that solves the GAP recurrences of (2) in sublinear $O(n^{3/4} \log n)$ time, optimal $O(n^3)$ work, $O(n^3)$ space, and $O(n^3/B)$ cache bounds.

Proof: Referring to Fig. 6b, we employ the algorithm of Lemma 12 to compute the 3D cubes and the algorithm of Theorem 9 to compute the 3D triangular analogues. At a high level, we preserve the ND scheduling as shown by the recurrences of (20) – (23) except that we stop the recursion of (23) earlier than (24) at size v as follows, where v is a parameter to be determined later.

$$T_{\infty,\text{ND-GAP}_{\underset{\rightsquigarrow}{\triangle\square}}}(v) = T_{\infty,\text{ND-GAP}_\triangle}(v) + T_{\infty,\text{ND-GAP}_\square}(v) \quad (25)$$

To bound the parameter v, we need to bound the total work of this hybrid algorithm to be the optimal $O(n^3)$. Since there are $O((n/v)^3)$ 3D cubes, each of which takes $O(v^3)$ work (Lemma 12), and $O((n/v)^2)$ 3D triangular analogues, each of which takes $O(v^4)$ work (Theorem 9), we have (26) to bound the total work, which solves to $v = n^{1/2}$.

$$O((n/v)^3) \cdot O(v^3) + O((n/v)^2) \cdot O(v^4) = O(n^3) \quad (26)$$

Supplying this v into (25) yields $T_{\infty,\text{ND-GAP}_{\underset{\rightsquigarrow}{\triangle\square}}}(v) = O(v^{1/2} \log v) = O(n^{1/4} \log n)$. Supplying this result into (20) – (23) yields the conclusion. ■

Discussions : Our result of Theorem 11 improves over the prior cache-oblivious and cache-efficient algorithm of Lemma 10 developed by Chowdhury and Ramachandran [2], [3] on time bound without sacrificing work, space, and cache efficiency. By Theorem 13, we solve an open problem raised in Galil and Park's paper [1], i.e. we have the first work-efficient and sublinear-time GAP algorithm, though its space and cache complexities are non-optimal, which will be left as yet another open problem for future work.

IV. Conclusion and Related Works

Concluding Remarks: Galil and Park [1] gave out several work-efficient and sublinear-time algorithms for DP recurrences with more than $O(1)$ dependency. However their algorithm for the general GAP problem is not work-efficient. Classic COP approach [2], [3], [10] usually attains optimal work, space, and cache bounds in a cache-oblivious fashion, but with a super-linear time bound due to both the updating order imposed by DP recurrences and excessive control dependency introduced by their approach. In this paper, we present a new framework to parallelize a DP computation based on a novel combination of the closure method and ND method. Our

results not only improves the time bounds of classic cache-oblivious parallel algorithms without sacrificing work, space, and cache efficiency, but also gives out the first work-efficient and sublinear-time algorithm for the general GAP problem, thus provides an answer to the open problem raised by Galil and Park [1].

Open Problem: It remains an interesting problem whether it is possible to further bound the space and cache bounds of the general GAP algorithms to be asymptotically optimal in a cache-oblivious fashion while keeping its work bound to be optimal and time bound to be sublinear.

Related Works: Galil, Giancarlo, and Park [5], [6], [1] proposed to solve DP recurrences with more than $O(1)$ dependency by the methods of closure, matrix product, and indirection. Maleki et al. [25] presented in their paper that certain dynamic programming problem called "Linear-Tropical Dynamic Programming (LTDP)" can possibly obtain extra parallelism by breaking data dependencies between stages. Their approach is based on the property of rank convergence of matrix multiplication in linear algebra. Their approach in the worst case can reduce to a serial algorithm. Chowdhury and Ramachandran [26] considered a processor-aware hybrid r-way divide-and-conquer algorithms with different values of r at different levels of recursion. Their cache bounds match asymptotically that of the best serial algorithm. Chowdhury et al. [27] proposed a multicore-oblivious (MO) method for a hierarchical multi-level caching (HM) model. By the MO method, an algorithm requires no specifying of any machine parameters such as the number of computing cores, number of cache levels, cache size, or block transfer size. However, for improved performance, an MO algorithm is allowed to provide advices or "hints" to the runtime scheduler through a small set of instructions on how to schedule the parallel tasks it spawns. Shun et al. [28] proposed "priority updates" to relax the serialization of "concurrent writes" to the same memory cell, thus could possibly improve the time bounds in some cases. However, not all operations can be prioritized.

REFERENCES

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

[2] 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*, 2006, pp. 591–600.

[3] R. Chowdhury, "Cache-efficient algorithms and data structures: Theory and experimental evaluation," Ph.D. dissertation, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas, 2007.

[4] Wikipedia contributors, "Dynamic programming — Wikipedia, the free encyclopedia," 2018.

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

[6] Z. Galil and K. Park, "Dynamic programming with convexity, concavity and sparsity," *Theoretical Computer Science*, vol. 92, no. 1, pp. 49–76, Jan. 1992.

[7] D. Hirschberg and L. Larmore, "The least weight subsequence problem," *SIAM Journal on Computing*, vol. 16, pp. 628–638, 1987.

[8] F. F. Yao, "Efficient dynamic programming using quadrangle inequalities," in *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, ser. STOC '80. New York, NY, USA: ACM, 1980, pp. 429–435.

[9] M. Waterman and T. Smith, "Rna secondary structure: a complete mathematical analysis," *Mathematical Biosciences*, vol. 42, no. 3, pp. 257 – 266, 1978.

[10] R. A. Chowdhury, H.-S. Le, and V. Ramachandran, "Cache-oblivious dynamic programming for bioinformatics," *TCBB*, vol. 7, no. 3, pp. 495–510, Jul.-Sep. 2010.

[11] 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'15*, San Francisco, CA, USA, Feb.7 – 11 2015.

[12] D. Dinh, H. V. Simhadri, and Y. Tang, "Extending the nested parallel model to the nested dataflow model with provably efficient schedulers," in *SPAA'16*, Pacific Grove, CA, USA, Jul.11 – 13 2016.

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

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[15] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," *ACM Trans. Algorithms*, vol. 8, no. 1, pp. 4:1–4:22, Jan. 2012.

[16] 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)*, 2000, pp. 1–12.

[17] D. Spoonhower, G. E. Blelloch, P. B. Gibbons, and R. Harper, "Beyond nested parallelism: Tight bounds on work-stealing overheads for parallel futures," in *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '09. New York, NY, USA: ACM, 2009, pp. 91–100.

[18] P. B. Gibbons, Y. Matias, and V. Ramachandran, "The queue-read queue-write pram model: Accounting for contention in parallel algorithms," *SIAM J. Comput.*, vol. 28, no. 2, pp. 733–769, Feb. 1999.

[19] R. P. Brent, "The parallel evaluation of general arithmetic expressions," vol. 21, no. 2, pp. 201–206, Apr. 1974.

[20] R. H. Halstead, Jr., "Implementation of Multilisp: Lisp on a multiprocessor," in *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas, Aug. 1984, pp. 9–17.

[21] ——, "Multilisp: A language for concurrent symbolic computation," vol. 7, no. 4, pp. 501–538, Oct. 1985.

[22] G. E. Blelloch and M. Reid-Miller, "Pipelining with futures," in *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '97. New York, NY, USA: ACM, 1997, pp. 249–259.

[23] M. Herlihy and Z. Liu, "Well-structured futures and cache locality," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 155–166.

[24] G. E. Blelloch, P. B. Gibbons, G. J. Narlikar, and Y. Matias, "Space-efficient scheduling of parallelism with synchronization variables," in *SPAA*. ACM, 1997, pp. 12–23.

[25] 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*, ser. PPoPP'14. New York, NY, USA: ACM, 2014, pp. 219–232.

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

[27] 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*, April 2010, pp. 1–12.

[28] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons, "Reducing contention through priority updates," in *SPAA*, 2013, pp. 152–163.