AUTOGEN: Automatic Discovery of Cache-Oblivious Parallel Recursive Algorithms for Solving Dynamic Programs

Rezaul Chowdhury *
Pramod Ganapathi
Jesmin Jahan Tithi

Department of Computer Science,
Stony Brook University, NY, USA.
{rezaul,pganapathi,jtithi}
@cs.stonybrook.edu

Charles Bachmeier
Bradley C. Kuszmaul
Charles E. Leiserson
Armando Solar-Lezama

Computer Science and Artificial
Intelligence Laboratory, MIT, MA, USA.
{cabach,bradley,cel}@mit.edu
asolar@csail.mit.edu

Yuan Tang*

School of Software,
Shanghai Key Laboratory of Intelligent
Information Processing,
Fudan University, Shanghai, China.
yuantang@fudan.edu.cn

Abstract

We present AUTOGEN — an algorithm that for a wide class of dynamic programming (DP) problems automatically discovers highly efficient cache-oblivious parallel recursive divide-and-conquer algorithms from inefficient iterative descriptions of DP recurrences. AUTOGEN analyzes the set of DP table locations accessed by the iterative algorithm when run on a DP table of small size, and automatically identifies a recursive access pattern and a corresponding provably correct recursive algorithm for solving the DP recurrence. We use AUTOGEN to autodiscover efficient algorithms for several well-known problems. Our experimental results show that several autodiscovered algorithms significantly outperform parallel looping and tiled loop-based algorithms. Also these algorithms are less sensitive to fluctuations of memory and bandwidth compared with their looping counterparts, and their running times and energy profiles remain relatively more stable. To the best of our knowledge, AUTOGEN is the first algorithm that can automatically discover new nontrivial divide-and-conquer algorithms.

Keywords AutoGen, automatic discovery, dynamic programming, recursive, divide-and-conquer, cache-efficient, parallel, cache-oblivious, energy-efficient, cache-adaptive

1. Introduction

AUTOGEN is an algorithm for automatic discovery of efficient recursive divide-and-conquer *dynamic programming* (DP) algorithms for multicore machines from naïve iterative descriptions of the dynamic programs. DP (Bellman 1957; Sniedovich 2010; Cormen et al. 2009) is a widely used algorithm design technique that finds optimal solutions to a problem by combining optimal solutions to its overlapping subproblems, and explores an otherwise exponential sized search space in polynomial time by saving solutions to subproblems in a table and never recomputing them. DP is extensively used in computational biology (Bafna and Edwards 2003; Durbin et al. 1998; Gusfield 1997; Waterman 1995), and in many other application areas including operations research, compilers (Lew and Mauch 2006), sports (Romer 2002; Duckworth and Lewis 1998), games (Smith 2007), economics (Rust 1996), finance (Robichek et al. 1971) and agriculture (Kennedy 1981).

Dynamic programs are described through recurrence relations that specify how the cells of a DP table must be filled using already computed values for other cells. Such recurrences are commonly implemented using simple algorithms that fill out DP tables iteratively. These loop-based codes are straightforward to implement, often have good *spatial cache locality*[1], and benefit from hardware prefetchers. But looping codes suffer in performance from poor *temporal cache locality*[2]. Iterative DP implementations are also often *inflexible* in the sense that the loops and the data in the DP table cannot be suitably reordered in order to optimize for better spatial locality, parallelization, and/or vectorization. Such inflexibility arises because the codes often read from and write to the same DP table, and thus imposing strict read-write ordering of the cells.

Recursive divide-and-conquer DP algorithms (see Table 1) can often overcome many limitations of their iterative counterparts. Because of their recursive nature such algorithms are known to have excellent (and often optimal) temporal locality. Efficient implementations of these algorithms use iterative kernels when the problem size becomes reasonably small. But unlike in standard loop-based DP codes, the loops inside these iterative kernels can often be easily reordered, thus allowing for better spatial locality, vectorization, parallelization, and other optimizations. The sizes of the iterative kernels are determined based on vectorization efficiency and overhead of recursion, and not on cache sizes, and thus the algorithms remain *cache-oblivious*[3] (Frigo et al. 1999) and more *portable* than cache-aware tiled iterative codes. Unlike tiled looping codes these algorithms are also *cache-adaptive* (Bender et al. 2014) — they passively self-adapt to fluctuations in available cache space when caches are shared with other concurrently running programs.

For example, consider the dynamic program for solving the parenthesis problem (Galil and Park 1994) in which we are given a sequence of characters $S = s_1 \cdots s_n$ and we are required to

* corresponding author

[1] Spatial locality — whenever a cache block is brought into the cache, it contains as much useful data as possible.

[2] Temporal locality — whenever a cache block is brought into the cache, as much useful work as possible is performed on this data before removing the block from the cache.

[3] Cache-oblivious algorithms — algorithms that do not use the knowledge of cache parameters in the algorithm description.

PPoPP '16 March 12-16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2851141.2851167

compute the minimum cost of parenthesizing S. Let $C[i, j]$ denote the minimum cost of parenthesizing $s_i \cdots s_j$. Then the DP table $C[0:n, 0:n]$ is filled up using the following recurrence:

$$C[i,j] = \begin{cases} \infty & \text{if } 0 \leq i = j \leq n, \\ v_j & \text{if } 0 \leq i = j-1 < n, \\ \min\limits_{i \leq k \leq j} \{(C[i,k] + C[k,j]) + w(i,k,j)\} & \text{if } 0 \leq i < j-1 < n; \end{cases} \tag{1}$$

where the v_j's and function $w(\cdot, \cdot, \cdot)$ are given.

Figure 1 shows a serial looping code LOOP-PARENTHESIS implementing Recurrence 1. Though the code is really easy to understand and write, it suffers from poor cache performance. Observe that the innermost loop scans one row and one column of the same DP table C. Assuming that C is of size $n \times n$ and C is too large to fit into the cache, each iteration of the innermost loop may incur one or more cache misses leading to a total of $\Theta\left(n^3\right)$ cache misses in the ideal-cache model (Frigo et al. 1999). Such extreme inefficiency in cache usage makes the code bandwidth-bound. Also this code does not have any parallelism as none of the three loops can be parallelized. The loops cannot also be reordered without making the code incorrect[4] which makes the code difficult to optimize.

Figure 1 shows the type of parallel looping code PAR-LOOP-PARENTHESIS one would write to solve Recurrence 1. We can analyze its parallel performance under the *work-span model* ((Cormen et al. 2009), chapter 27) which defines the *parallelism* of a code as T_1/T_∞, where T_p ($p \in [1, \infty)$) is the running time of the code on p processing cores (without scheduling overhead). Clearly, the parallelism of PAR-LOOP-PARENTHESIS is $\Theta\left(n^3\right)/\Theta\left(n^2\right) = \Theta\left(n\right)$. If the size M of the cache is known the code can be tiled to improve its cache performance to $\Theta\left(n^3/\left(B\sqrt{M}\right)\right)$, where B is the cache line size. However, such rigid cache-aware tiling makes the code less portable, and may contribute to a significant loss of performance when other concurrently running programs start to use space in the shared cache.

Finally, Figure 1 shows the type of algorithm $\mathcal{AUTOGEN}$ would generate from the serial code. Though designing such a parallel recursive divide-and-conquer algorithm is not straightforward, it has many nice properties. First, the algorithm is cache-oblivious, and for any cache of size M and line size B it always incurs $\Theta\left(n^3/\left(B\sqrt{M}\right)\right)$ cache misses which can be shown to be optimal. Second, its parallelism is $\Theta\left(n^{3-\log_2 3}\right) = \omega\left(n^{1.41}\right)$ which is asymptotically greater than the $\Theta\left(n\right)$ parallelism achieved by the parallel looping code. Third, since the algorithm uses recursive blocking, it can passively self-adapt to a correct block size (within a small constant factor) as the available space in the shared cache changes during runtime. Fourth, it has been shown that function $\mathcal{C}_{loop\text{-}par}$ is highly optimizable like a matrix multiplication algorithm, and the total time spent inside $\mathcal{C}_{loop\text{-}par}$ asymptotically dominates the time spent inside $\mathcal{A}_{loop\text{-}par}$ and $\mathcal{B}_{loop\text{-}par}$ (Tithi et al. 2015). Hence, reasonably high performance can be achieved simply by optimizing $\mathcal{C}_{loop\text{-}par}$.

We ran the recursive algorithm and the parallel looping algorithm from Figure 1 both with and without tiling on a multicore machine with dual-socket 8-core 2.7 GHz Intel Sandy Bridge processors ($2 \times 8 = 16$ cores in total), per-core 32 KB private L1 cache and 256 KB private L2 cache, and per-socket 20 MB shared L3 cache, and 32 GB RAM shared by all cores. All algorithms were implemented in C++, parallelized using Intel Cilk Plus extension, and compiled using Intel C++ Compiler v13.0. For a DP table of size 8000×8000, the recursive algorithm without any nontrivial hand-optimizations ran more than 15 times faster than the non-tiled looping code, and slightly faster than the tiled looping code when each program was running all alone on the machine. When we ran four instances of the same program (i.e., algorithm) on the same socket each using only 2 cores, the non-tiled looping code slowed down by almost a factor of 2 compared to a single instance running on 2 cores, the tiled looping code slowed down by a factor of 1.5, and the recursive code slowed down by a factor of only 1.15. While the non-tiled looping code suffered because of bandwidth saturation, the tiled looping code suffered because of its inability to adapt to cache sharing.

In this paper, we present $\mathcal{AUTOGEN}$ — an algorithm that for a very wide class of DP problems can *automatically discover* efficient cache-oblivious parallel recursive divide-and-conquer algorithms from naïve serial iterative descriptions of DP recurrences (see Figure 2). $\mathcal{AUTOGEN}$ works by analyzing the set of DP table locations accessed by the input serial algorithm when run on a DP table of suitably small size, and identifying a recursive fractal-like pattern in that set. For the class of DP problems handled by $\mathcal{AUTOGEN}$ the set of table locations accessed by the algorithm is independent of the data stored in the table. The class includes many well-known DP problems such as the parenthesis problem, pairwise sequence alignment and the gap problem as well as problems that are yet to be encountered. $\mathcal{AUTOGEN}$ effectively eliminates the need for human involvement in the design of efficient cache-oblivious parallel algorithms for all present and future problems in that class.

Our contributions. Our major contributions are as follows:

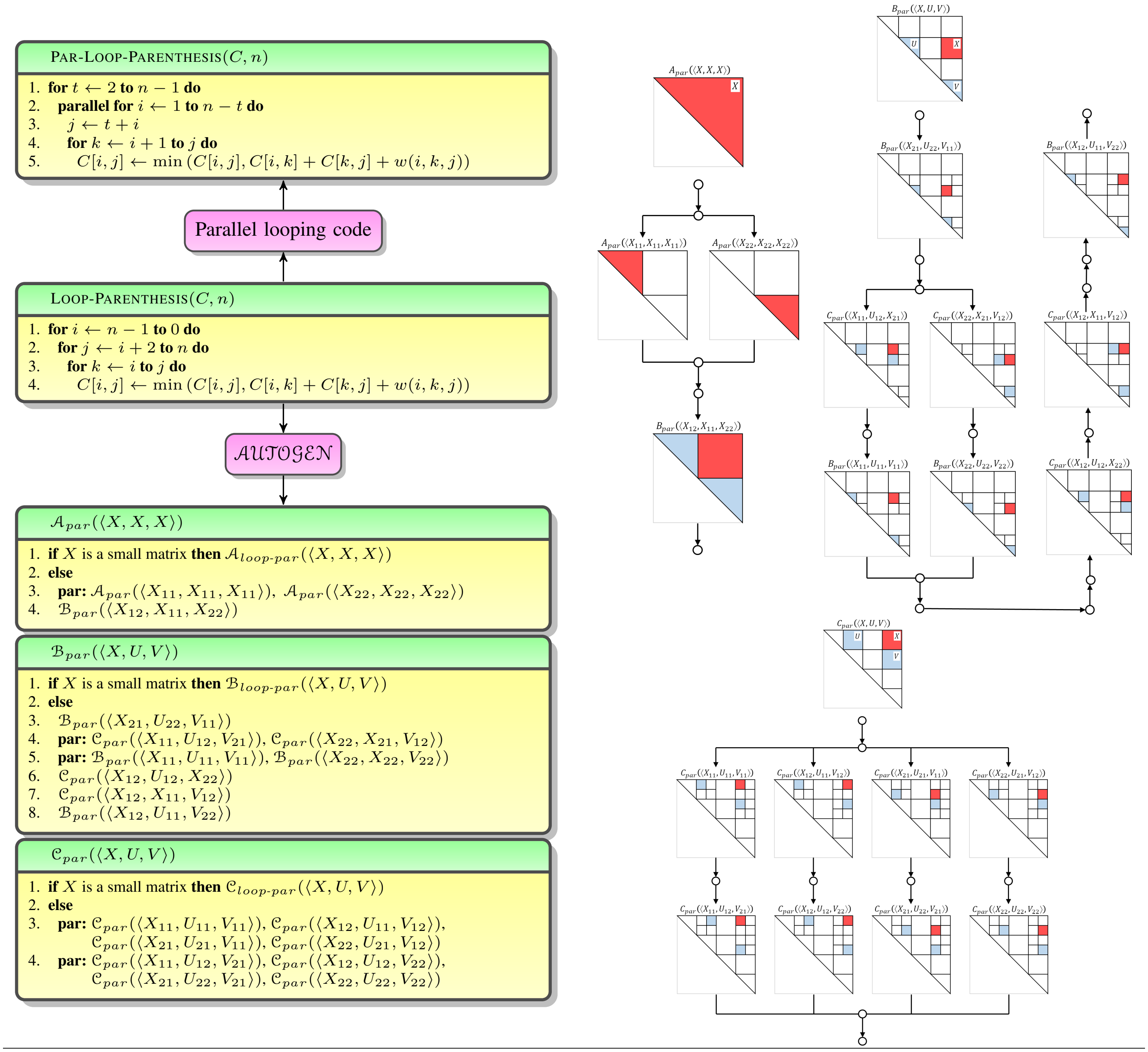
(1) **[Algorithmic]** We present $\mathcal{AUTOGEN}$ — an algorithm that for a wide class of DP problems automatically discovers highly efficient cache-oblivious parallel recursive divide-and-conquer algorithms from iterative descriptions of DP recurrences. $\mathcal{AUTOGEN}$ works by analyzing the DP table accesses (assumed to be independent of the data in the table) of an iterative algorithm on a table of small size, finding the dependencies among different orthants of the DP table recursively, and constructing a tree and directed acyclic graphs that represent a set of recursive functions corresponding to a parallel recursive divide-and-conquer algorithm. We prove the correctness of the algorithms generated by $\mathcal{AUTOGEN}$.

(2) **[Experimental]** We have implemented a prototype of $\mathcal{AUTOGEN}$ which we have used to autogenerate efficient cache-oblivious parallel recursive divide-and-conquer algorithms (pseudocodes) from naïve serial iterative descriptions of several DP recurrences. We present experimental results showing that several autogenerated algorithms without any nontrivial hand-tuning significantly outperform parallel looping codes in practice, and have more stable running times and energy profiles in a multiprogramming environment compared to looping and tiling algorithms.

Related work. Systems for auto-generating fast iterative DP implementations (not algorithms) exist. The Bellman's GAP compiler (Giegerich and Sauthoff 2011) converts declarative programs into optimized C++ code. A semi-automatic synthesizer (Pu et al. 2011) exists which uses contraint-solving to solve linear-time DP problems such as maximal substring matching, assembly-line optimization and the extended Euclid algorithm.

There are systems to automatically parallelize DP loops. EasyPDP (Tang et al. 2012) requires the user to select a directed acyclic graph (DAG) pattern for a DP problem from its DAG patterns library. New DAG patterns can be added to the library. EasyHPS (Du et al. 2013) uses the master-slave paradigm in which the master scheduler distributes computable sub-tasks among its slaves, which in turn distribute subsubtasks among slave threads. A pattern-based system exists (Liu and Schmidt 2004) that uses generic programming techniques such as class templates to solve problems in bioin-

[4] compare this with iterative matrix multiplication in which all 6 permutations of the three nested loops produce correct results



```
PAR-LOOP-PARENTHESIS$(C, n)$
for $t \leftarrow 2$ to $n - 1$ do
  parallel for $i \leftarrow 1$ to $n - t$ do
    $j \leftarrow t + i$
    for $k \leftarrow i + 1$ to $j$ do
      $C[i, j] \leftarrow \min\left(C[i, j], C[i, k] + C[k, j] + w(i, k, j)\right)$
```

```
LOOP-PARENTHESIS$(C, n)$
for $i \leftarrow n - 1$ to $0$ do
  for $j \leftarrow i + 2$ to $n$ do
    for $k \leftarrow i$ to $j$ do
      $C[i, j] \leftarrow \min\left(C[i, j], C[i, k] + C[k, j] + w(i, k, j)\right)$
```

```
$\mathcal{A}_{par}(\langle X, X, X \rangle)$
if $X$ is a small matrix then $\mathcal{A}_{loop\text{-}par}(\langle X, X, X \rangle)$
else
  par: $\mathcal{A}_{par}(\langle X_{11}, X_{11}, X_{11} \rangle)$, $\mathcal{A}_{par}(\langle X_{22}, X_{22}, X_{22} \rangle)$
  $\mathcal{B}_{par}(\langle X_{12}, X_{11}, X_{22} \rangle)$
```

```
$\mathcal{B}_{par}(\langle X, U, V \rangle)$
if $X$ is a small matrix then $\mathcal{B}_{loop\text{-}par}(\langle X, U, V \rangle)$
else
  $\mathcal{B}_{par}(\langle X_{21}, U_{22}, V_{11} \rangle)$
  par: $\mathcal{C}_{par}(\langle X_{11}, U_{12}, V_{21} \rangle)$, $\mathcal{C}_{par}(\langle X_{22}, X_{21}, V_{12} \rangle)$
  par: $\mathcal{B}_{par}(\langle X_{11}, U_{11}, V_{11} \rangle)$, $\mathcal{B}_{par}(\langle X_{22}, X_{22}, V_{22} \rangle)$
  $\mathcal{C}_{par}(\langle X_{12}, U_{12}, X_{22} \rangle)$
  $\mathcal{C}_{par}(\langle X_{12}, X_{11}, V_{12} \rangle)$
  $\mathcal{B}_{par}(\langle X_{12}, U_{11}, V_{22} \rangle)$
```

```
$\mathcal{C}_{par}(\langle X, U, V \rangle)$
if $X$ is a small matrix then $\mathcal{C}_{loop\text{-}par}(\langle X, U, V \rangle)$
else
  par: $\mathcal{C}_{par}(\langle X_{11}, U_{11}, V_{11} \rangle)$, $\mathcal{C}_{par}(\langle X_{12}, U_{11}, V_{12} \rangle)$,
       $\mathcal{C}_{par}(\langle X_{21}, U_{21}, V_{11} \rangle)$, $\mathcal{C}_{par}(\langle X_{22}, U_{21}, V_{12} \rangle)$
  par: $\mathcal{C}_{par}(\langle X_{11}, U_{12}, V_{21} \rangle)$, $\mathcal{C}_{par}(\langle X_{12}, U_{12}, V_{22} \rangle)$,
       $\mathcal{C}_{par}(\langle X_{21}, U_{22}, V_{21} \rangle)$, $\mathcal{C}_{par}(\langle X_{22}, U_{22}, V_{22} \rangle)$
```

Parallel looping code
$\mathcal{AUTOGEN}$
$A_{par}(\langle X, X, X \rangle)$
X
$A_{par}(\langle X_{11}, X_{11}, X_{11} \rangle)$
$A_{par}(\langle X_{22}, X_{22}, X_{22} \rangle)$
$B_{par}(\langle X_{12}, X_{11}, X_{22} \rangle)$
$B_{par}(\langle X, U, V \rangle)$
U
X
V
$B_{par}(\langle X_{21}, U_{22}, V_{11} \rangle)$
$B_{par}(\langle X_{12}, U_{11}, V_{22} \rangle)$
$C_{par}(\langle X_{11}, U_{12}, X_{21} \rangle)$
$C_{par}(\langle X_{22}, X_{21}, V_{12} \rangle)$
$C_{par}(\langle X_{12}, X_{11}, V_{12} \rangle)$
$B_{par}(\langle X_{11}, U_{11}, V_{11} \rangle)$
$B_{par}(\langle X_{22}, U_{22}, V_{22} \rangle)$
$C_{par}(\langle X_{12}, U_{12}, X_{22} \rangle)$
$C_{par}(\langle X, U, V \rangle)$
U
X
V
$C_{par}(\langle X_{11}, U_{11}, V_{11} \rangle)$
$C_{par}(\langle X_{12}, U_{11}, V_{12} \rangle)$
$C_{par}(\langle X_{21}, U_{21}, V_{11} \rangle)$
$C_{par}(\langle X_{22}, U_{21}, V_{12} \rangle)$
$C_{par}(\langle X_{11}, U_{12}, V_{21} \rangle)$
$C_{par}(\langle X_{12}, U_{12}, V_{22} \rangle)$
$C_{par}(\langle X_{21}, U_{22}, V_{21} \rangle)$
$C_{par}(\langle X_{22}, U_{22}, V_{22} \rangle)$

Figure 1. Left upper half: A parallel looping code that evaluates Rec. 1. Left lower half: $\mathcal{AUTOGEN}$ takes the serial parenthesis algorithm as input and automatically discovers a recursive divide-and-conquer cache-oblivious parallel algorithm. Initial call to the divide-and-conquer algorithm is $\mathcal{A}_{par}(\langle C, C, C \rangle)$, where C is an $n \times n$ DP table and n is a power of 2. The iterative base-case kernel of a function $\mathcal{F}_{par}$ is $\mathcal{F}_{loop\text{-}par}$. Right: Pictorial representation of the recursive divide-and-conquer algorithm discovered by $\mathcal{AUTOGEN}$. Data in the dark red blocks are updated using data from light blue blocks.

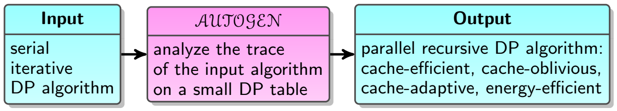


Figure 2. Input and output of $\mathcal{AUTOGEN}$.

formatics. Parallelizing plugins (Reitzig 2012) use diagonal frontier and row splitting to parallelize DP loops.

To the best of our knowledge, there has been no previous attempt to automate the process of discovering efficient cache-oblivious and cache-adaptive parallel recursive algorithms by analyzing the memory access patterns of naïve serial iterative algorithms. The work that is most related to $\mathcal{AUTOGEN}$, but completely different in many aspects is Pochoir (Tang et al. 2011a,b). While Pochoir tailors the implementation of the same cache-oblivious algorithm to different stencil computations, $\mathcal{AUTOGEN}$ discovers a (possibly) brand new efficient parallel cache-oblivious algorithm for every new DP problem it encounters.

Compiler technology for automatically converting iterative versions of matrix programs to serial recursive versions is described in (Ahmed and Pingali 2000). The approach relies on heavy machineries such as dependence analysis (based on integer programming) and polyhedral techniques. $\mathcal{AUTOGEN}$, on the other hand, is a much simpler stand-alone algorithm that analyzes the data access pattern of a given naïve (e.g., looping) serial DP code when run on a small example, and inductively generates a provably correct parallel recursive algorithm for solving the same DP.

		$\mathcal{I}$-$\mathcal{DP}$			$\mathcal{R}$-$\mathcal{DP}$		
Problem	Work (T_1)	Serial cache comp. (Q_1)	Span (T_∞)	Parallelism (T_1/T_∞)	Serial cache comp. (Q_1)	Span (T_∞)	Parallelism (T_1/T_∞)
Parenthesis problem (Chowdhury and Ramachandran 2008)	$\Theta(n^3)$	$\Theta(n^3)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log 3}\right)$	$\Theta\left(n^{3-\log 3}\right)$
Floyd-Warshall's APSP 3-D (Chowdhury and Ramachandran 2010)	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n\log n)$	$\Theta(n^2/\log n)$	$\Theta(n^3/B)$	$\mathcal{O}(n\log^2 n)$	$\Theta(n^2/\log^2 n)$
Floyd-Warshall's APSP 2-D (Chowdhury and Ramachandran 2010)	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n\log n)$	$\Theta(n^2/\log n)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n\log^2 n)$	$\Theta(n^2/\log^2 n)$
LCS / Edit distance (Chowdhury and Ramachandran 2006)	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2/(BM))$	$\Theta\left(n^{\log 3}\right)$	$\Theta\left(n^{2-\log 3}\right)$
Multi-instance Viterbi (Chowdhury et al.)	$\Theta(n^3 t)$	$\Theta(n^3 t/B)$	$\Theta(nt)$	$\Theta(n^2)$	$\Theta\left(n^3 t/(B\sqrt{M})\right)$	$\Theta(nt)$	$\Theta(n^2)$
Gap problem (Chowdhury 2007)	$\Theta(n^3)$	$\Theta(n^3)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log 3}\right)$	$\Theta\left(n^{3-\log 3}\right)$
Protein accordion folding (Tithi et al. 2015)	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n\log n)$	$\Theta(n^2/\log n)$
Spoken-word recognition (Sakoe and Chiba 1978)	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2/(BM))$	$\Theta\left(n^{\log 3}\right)$	$\Theta\left(n^{2-\log 3}\right)$
Function approximation	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log 3}\right)$	$\Theta\left(n^{3-\log 3}\right)$
Binomial coefficient (Levitin 2011)	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2/B)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2/(BM))$	$\Theta\left(n^{\log 3}\right)$	$\Theta\left(n^{2-\log 3}\right)$
Bitonic traveling salesman (Cormen et al. 2009)	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2/(BM))$	$\Theta(n\log n)$	$\Theta(n/\log n)$
Matrix multiplication (Frigo et al. 1999)	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n)$	$\Theta(n^2)$
Bubble sort (Chowdhury and Ganapathi)	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n^2)$	$\Theta(1)$	$\Theta(n^2/(BM))$	$\Theta(n)$	$\Theta(n)$
Selection sort (Chowdhury and Ganapathi)	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n^2)$	$\Theta(1)$	$\Theta(n^2/(BM))$	$\Theta(n)$	$\Theta(n)$
Insertion sort (Chowdhury and Ganapathi)	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2/B)$	$\mathcal{O}(n^2)$	$\Theta(1)$	$\mathcal{O}\left(n^{\log 3}/(BM^{\log 3-1})\right)$	$\mathcal{O}(n)$	$\Omega(n)$

Table 1. Work (T_1), serial cache complexity (Q_1), span (T_∞), and parallelism (T_1/T_∞) of $\mathcal{I}$-$\mathcal{DP}$ and $\mathcal{R}$-$\mathcal{DP}$ algorithms for several DP problems. Here, n = problem size, M = cache size, B = block size, and p = #cores. By T_p we denote running time on p processing cores. We assume that the DP table is too large to fit into the cache, and $M = \Omega\left(B^d\right)$ when $\Theta\left(n^d\right)$ is the size of the DP table. On p cores, the running time is $T_p = \mathcal{O}(T_1/p + T_\infty)$ and the parallel cache complexity is $Q_p = \mathcal{O}(Q_1 + p(M/B)T_\infty)$ with high probability when run under the randomized work-stealing scheduler on a parallel machine with private caches. The problems in the lower section are non-DP problems. For insertion sort, T_1 for $\mathcal{R}$-$\mathcal{DP}$ is $\mathcal{O}\left(n^{\log 3}\right)$.

2. The $\mathcal{AUTOGEN}$ Algorithm

In this section, we describe the $\mathcal{AUTOGEN}$ algorithm.

Definition 1 ($\mathcal{I}$-$\mathcal{DP}$/$\mathcal{R}$-$\mathcal{DP}$/$\mathcal{AUTOGEN}$)**.** *Let $\mathcal{P}$ be a given DP problem. An $\mathcal{I}$-$\mathcal{DP}$ is an iterative (i.e., loop-based) algorithm for solving $\mathcal{P}$. An $\mathcal{R}$-$\mathcal{DP}$ is a cache-oblivious parallel recursive divide-and-conquer algorithm (if exists) for $\mathcal{P}$. $\mathcal{AUTOGEN}$ is our algorithm for auto-generating an $\mathcal{R}$-$\mathcal{DP}$ from a given $\mathcal{I}$-$\mathcal{DP}$ for $\mathcal{P}$.*

We make the following assumption about an $\mathcal{R}$-$\mathcal{DP}$.

Assumption 1 (**Number of functions**)**.** *The number of distinct recursive functions in an $\mathcal{R}$-$\mathcal{DP}$ is upper bounded by a constant (e.g., the $\mathcal{R}$-$\mathcal{DP}$ in Figure 1 has 3 distinct recursive functions).*

Algorithm. The four main steps of $\mathcal{AUTOGEN}$ are:

(1) [**Cell-set generation.**] A cell-set (i.e., set of cell-dependencies representing DP table cells accessed) is generated from a run of the given $\mathcal{I}$-$\mathcal{DP}$ on a DP table of small size. See §2.1.

(2) [**Algorithm-tree construction.**] An algorithm-tree is constructed from the cell-set in which each node represents a subset of the cell-set and follows certain rules. See §2.2.

(3) [**Algorithm-tree labeling.**] The nodes of the tree are labeled with function names, and these labels represent a set of recursive divide-and-conquer functions in an $\mathcal{R}$-$\mathcal{DP}$. See §2.3.

(4) [**Algorithm-DAG construction.**] For every unique function of the $\mathcal{R}$-$\mathcal{DP}$, we construct a directed acyclic graph (DAG) that shows both the order in which the child functions are to be executed and the parallelism involved. See §2.4.

Example. $\mathcal{AUTOGEN}$ works for arbitrary d-D ($d \geq 1$) DP problems under the assumption that each dimension of the DP table is of the same length and is a power of 2. For simplicity of exposition, we explain $\mathcal{AUTOGEN}$ by applying it on an $\mathcal{I}$-$\mathcal{DP}$ for the parenthesis problem, which updates a 2-D DP table. The solution is described by Recurrence 1 which is evaluated by the serial $\mathcal{I}$-$\mathcal{DP}$. In the rest of the section, we show how $\mathcal{AUTOGEN}$ discovers the $\mathcal{R}$-$\mathcal{DP}$ shown in Figure 1 from this serial $\mathcal{I}$-$\mathcal{DP}$.

2.1 Cell-set generation

A *cell* is a spatial grid point in a DP table identified by its d-D coordinates. A d-D DP table C is called a level-0 *region*. The orthants of identical dimensions of the level-0 region are called level-1 regions. Generalizing, the orthants of level-i regions are called level-$(i+1)$ regions.

We assume that each iteration of the innermost loop of the given $\mathcal{I}$-$\mathcal{DP}$ performs the following update:

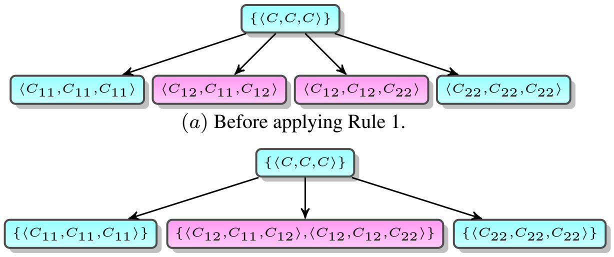
$$C[x] \leftarrow f(C^1[y_1], C^2[y_2], \ldots, C^s[y_s]) \text{ or}$$
$$C[x] \leftarrow C[x] \oplus f(C^1[y_1], C^2[y_2], \ldots, C^s[y_s]),$$

where $s \geq 1$; x is a cell of table C; y_i is a cell of table C^i; $\oplus$ is an associative operator (such as min, max, $+$, $\times$); and f is an arbitrary function. We call the tuple $\langle C[x], C^1[y_1], \ldots, C^s[y_s]\rangle$ a *cell-tuple*. Let $C[X], C^1[Y_1], \ldots, C^s[Y_s]$ be regions such that $x \in X$, and $y_i \in Y_i$. Then we call the tuple $\langle C[X], C^1[Y_1], \ldots, C^s[Y_s]\rangle$ a *region-tuple*. In simple words, a cell-tuple (resp. region-tuple) gives information of which cell (resp. region) is being written by reading from which cells (resp. regions). The size of a cell-/region-tuple is $1+s$. For any given $\mathcal{I}$-$\mathcal{DP}$, the set of all cell-tuples for all cells in its DP table is called a *cell-set*.

Given an $\mathcal{I}$-$\mathcal{DP}$, we modify it such that instead of computing its DP table, it generates the cell-set for a problem of suitably small size, generally $n = 64$ or 128. For example, for the parenthesis problem, we choose $n = 64$ and generate the cell-set $\{\langle C(i,j), C(i,k), C(k,j)\rangle\}$, where C is the DP table, $0 \leq i < j-1 < n$, and $i \leq k \leq j$.

2.2 Algorithm-tree construction

Given an $\mathcal{I}$-$\mathcal{DP}$, a tree representing a hierarchy of recursive divide-and-conquer functions which is used to find a potential $\mathcal{R}$-$\mathcal{DP}$ is called an *algorithm-tree*. The way we construct level-i nodes in an algorithm-tree is by analyzing the dependencies between level-i regions using the cell-set. Every node in the algorithm-tree represents a subset of the cell-set satisfying certain region-tuple dependencies. Suppose the algorithm writes into DP table C, and reads from tables $C^1, \ldots, C^s$ (they can be same as C). The algorithm-tree is constructed as follows.


$\{\langle C,C,C\rangle\}$
$\langle C_{11},C_{11},C_{11}\rangle$
$\langle C_{12},C_{11},C_{12}\rangle$
$\langle C_{12},C_{12},C_{22}\rangle$
$\langle C_{22},C_{22},C_{22}\rangle$
$\{\langle C,C,C\rangle\}$
$\{\langle C_{11},C_{11},C_{11}\rangle\}$
$\{\langle C_{12},C_{11},C_{12}\rangle,\langle C_{12},C_{12},C_{22}\rangle\}$
$\{\langle C_{22},C_{22},C_{22}\rangle\}$

(a) Before applying Rule 1.

(b) After applying Rule 1.

Figure 3. First two levels of the algorithm-tree for the parenthesis problem before and after applying Rule 1.

At level 0, the only regions possible are the entire tables $C, C^1, \ldots, C^s$. We analyze the cell-tuples of the cell-set to identify the region-tuples at this level. As all the write cells belong to C and all the read cells belong to $C^1, \ldots, C^s$, the only possible region-tuple is $\langle C, C^1, \ldots, C^s\rangle$. We create a node for this region-tuple and it forms the root node of the algorithm-tree. It represents the entire cell-set. For example, for parenthesis problem, as all the write and read cells belong to the same DP table C, the root node will be $\{\langle C, C, C\rangle\}$.

The level-1 nodes are found by distributing the cell-tuples belonging to the root node among region-tuples of level 1. The level-1 regions are obtained by dividing the DP table C into four quadrants: C_{11} (top-left), C_{12} (top-right), C_{21} (bottom-left), and C_{22} (bottom-right). Similarly, each C^i for $i \in [1, s]$ is divided into four quadrants: $C^i_{11}, C^i_{12}, C^i_{21}$, and C^i_{22}. The cell-tuples of the cell-set are analyzed to find all possible nonempty region-tuples at level 1. For example, if a cell-tuple $\langle c, c_1, \ldots, c_s\rangle$ is found to have $c \in C_k$ and $c_i \in C^i_{k_i}$ for $i \in [1, s]$ and $k, k_i \in \{11, 12, 21, 22\}$, then we say that $\langle c, c_1, \ldots, c_s\rangle$ belongs to region-tuple $\langle C_k, C^1_{k_1}, \ldots, C^s_{k_s}\rangle$. Different problems will have different nonempty region-tuples depending on their cell dependencies. For the parenthesis problem, there are four nonempty level-1 region-tuples and they are $\langle C_{11}, C_{11}, C_{11}\rangle$, $\langle C_{22}, C_{22}, C_{22}\rangle$, $\langle C_{12}, C_{11}, C_{12}\rangle$, and $\langle C_{12}, C_{12}, C_{22}\rangle$.

Sometimes two or more region-tuples are combined into a node. The region-tuples that write to and read from the same region depend on each other for the complete update of the write region. The following rule guarantees that such region-tuples are processed together to avoid incorrect results.

Rule 1 (Combine region-tuples). *Two region-tuples at the same level of an algorithm-tree that write to the same region X are combined into a single node if they also read from X.*

For example in Figure 3, for the parenthesis problem, at level 1, the two region-tuples $\langle C_{12}, C_{11}, C_{12}\rangle$ and $\langle C_{12}, C_{12}, C_{22}\rangle$ are combined into a single node $\{\langle C_{12}, C_{11}, C_{12}\rangle, \langle C_{12}, C_{12}, C_{22}\rangle\}$. The other two nodes are $\{\langle C_{11}, C_{11}, C_{11}\rangle\}$ and $\{\langle C_{22}, C_{22}, C_{22}\rangle\}$. The three nodes represent three mutually disjoint subsets of the cell-set and have different region-tuple dependencies. Once we find all level 1 nodes, we recursively follow the same strategy to find the nodes of levels ≥ 2 partitioning the subsets of the cell-set further depending on their region-tuple dependencies.

2.3 Algorithm-tree labeling

Two nodes of the algorithm-tree are given the same function name provided they have the same output fingerprints as well as the same input fingerprints as defined below.

The *output fingerprint* of a node is the set of all output fingerprints of its region-tuples. The output fingerprint of a region-tuple is defined as the set of all its subregion-tuples present in the child nodes. A subregion-tuple of a region-tuple $\langle W, R_1, \ldots, R_s\rangle$ is defined as a tuple $\langle w, r_1, \ldots, r_s\rangle$ where $w, r_i \in \{11, 12, 21, 22\}$ such that $\langle W_w, R_{r_1}, \ldots, R_{r_s}\rangle$ is a region-tuple, where $\forall i \in [1, s]$.

The *input fingerprint* of a node is the set of all input fingerprints of its region-tuples. The input fingerprint of a region-tuple $\langle X_1, \ldots, X_{1+s}\rangle$ is a tuple $\langle p_1, \ldots, p_{1+s}\rangle$, where $\forall i \in [1, 1+s]$, p_i is the smallest index $j \in [1, i]$ such that $X_j = X_i$.

For example, in the parenthesis problem, nodes $\{\langle C_{12}, C_{11}, C_{12}\rangle, \langle C_{12}, C_{12}, C_{22}\rangle\}$ and $\{\langle C_{1221}, C_{1122}, C_{1221}\rangle, \langle C_{1221}, C_{1221}, C_{2211}\rangle\}$ are given the same function name because they have the same output and input fingerprints.

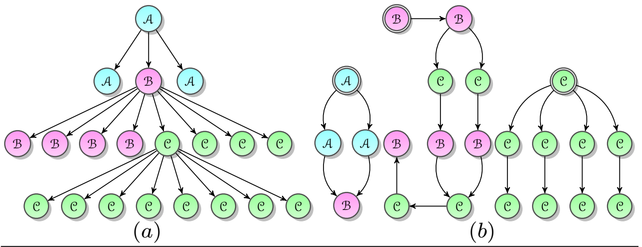


Figure 4. (a) A small part of the labeled algorithm-tree for the parenthesis problem. Due to space constaints, only three nodes are expanded. (b) Algorithm-DAGs for the three functions $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ in the parenthesis problem showing the order of execution of functions.

In an algorithm-tree, at least one new function is invoked at every level starting from level 0 till a certain level l, beyond which no new functions are invoked. We call l the *threshold level* and it is upper bounded by a constant as per Assumption 1. The labeled algorithm-tree for the parenthesis problem is given in Figure 4(a).

2.4 Algorithm-DAG construction

In this step, we construct a directed acyclic graph (DAG) for every function. An algorithm-tree does not give information on (a) the sequence in which a function calls other functions, and (b) the parallelism involved in executing the functions. The DAGs address these two issues using the rules that follow.

We define a few terms before listing the rules. Given a function $\mathcal{F}$, we define $\mathbf{W}(\mathcal{F})$ and $\mathbf{R}(\mathcal{F})$ as the write region and the set of read regions of the region-tuples in $\mathcal{F}$, respectively. For a region-tuple $T = \langle W, R_1, \ldots, R_s\rangle$, we define $\mathbf{W}(T) = W$ and $\mathbf{R}(T) = \{R_1, \ldots, R_s\}$. A region-tuple T is called *flexible* provided $\mathbf{W}(T) \notin \mathbf{R}(T)$, i.e., the region-tuple does not write to a region it reads from. A function is called flexible if all of its region-tuples are flexible. If a function $\mathcal{F}$ calls two functions $\mathcal{F}_1$ and $\mathcal{F}_2$, then the *function ordering* between $\mathcal{F}_1$ and $\mathcal{F}_2$ will be one of the following three: (a) $\mathcal{F}_1 \rightarrow \mathcal{F}_2$ i.e., $\mathcal{F}_1$ is called before $\mathcal{F}_2$, (b) $\mathcal{F}_1 \leftrightarrow \mathcal{F}_2$ i.e., either $\mathcal{F}_1 \rightarrow \mathcal{F}_2$ or $\mathcal{F}_2 \rightarrow \mathcal{F}_1$, and (c) $\mathcal{F}_1 || \mathcal{F}_2$ i.e., $\mathcal{F}_1$ can be run in parallel with $\mathcal{F}_2$.

If a function $\mathcal{F}$ calls two functions $\mathcal{F}_1$ and $\mathcal{F}_2$, then the order in which $\mathcal{F}_1$ and $\mathcal{F}_2$ are executed is determined by the following rules.

Rule 2. *If $\mathbf{W}(\mathcal{F}_1) \neq \mathbf{W}(\mathcal{F}_2)$ and $\mathbf{W}(\mathcal{F}_1) \in \boldsymbol{R}(\mathcal{F}_2)$, then $\mathcal{F}_1 \rightarrow \mathcal{F}_2$.*

Rule 3. *If $\mathbf{W}(\mathcal{F}_1) = \mathbf{W}(\mathcal{F}_2)$, $\mathcal{F}_1$ is flexible but $\mathcal{F}_2$ is not, then $\mathcal{F}_1 \rightarrow \mathcal{F}_2$.*

Rule 4. *If $\mathbf{W}(\mathcal{F}_1) = \mathbf{W}(\mathcal{F}_2)$ and both $\mathcal{F}_1$ and $\mathcal{F}_2$ are flexible, then $\mathcal{F}_1 \leftrightarrow \mathcal{F}_2$.*

Rule 5. *If $\mathcal{F}_1$ and $\mathcal{F}_2$ satisfy none of the rules 2, 3 and 4, then $\mathcal{F}_1 || \mathcal{F}_2$.*

We modify the constructed DAGs by deleting redundant edges from them following a specific set of rules (omitted due to space constraints). The set of all modified DAGs for all functions represents an $\mathcal{R}$-$\mathcal{DP}$ for the given $\mathcal{I}$-$\mathcal{DP}$. The algorithm-DAGs for the parenthesis problem is given in Figure 4(b).

3. Correctness & Cache Complexity

In this section, we give a proof of correctness for $\mathcal{AUTOGEN}$ and analyze the cache complexity of the autodiscovered $\mathcal{R}$-$\mathcal{DP}$s.

3.1 Correctness of $\mathcal{AUTOGEN}$

We prove that if an $\mathcal{I}$-$\mathcal{DP}$ satisfies the following properties, then $\mathcal{AUTOGEN}$ can be applied on the $\mathcal{I}$-$\mathcal{DP}$ to get a correct $\mathcal{R}$-$\mathcal{DP}$.

’

Case	$\boldsymbol{W}(T_1) \in \boldsymbol{R}(T_1)$	$\boldsymbol{W}(T_2) \in \boldsymbol{R}(T_2)$	$\boldsymbol{W}(T_1) \in \boldsymbol{R}(T_2)$	$\boldsymbol{W}(T_1) = \boldsymbol{W}(T_2)$	$\mathcal{I}$	Rule	$\mathcal{R}$
1	✓	✓	✓	✓	$T_1 = T_2$	–	$T_1 = T_2$
2	✓	✓	✓	✗	$T_1 \rightarrow T_2$	2	$T_1 \rightarrow T_2$
3	✓	✓	✗	✓	–	–	–
4	✓	✓	✗	✗	$T_1 \parallel T_2$	5	$T_1 \parallel T_2$
5	✓	✗	✓	✓	–	–	–
6	✓	✗	✓	✗	$T_1 \rightarrow T_2$	2	$T_1 \rightarrow T_2$
7	✓	✗	✗	✓	$T_2 \rightarrow T_1$	3	$T_2 \rightarrow T_1$
8	✓	✗	✗	✗	$T_1 \parallel T_2$	5	$T_1 \parallel T_2$
9	✗	✓	✓	✓	$T_1 \rightarrow T_2$	3	$T_1 \rightarrow T_2$
10	✗	✓	✓	✗	$T_1 \rightarrow T_2$	2	$T_1 \rightarrow T_2$
11	✗	✓	✗	✓	–	–	–
12	✗	✓	✗	✗	$T_1 \parallel T_2$	5	$T_1 \parallel T_2$
13	✗	✗	✓	✓	–	–	–
14	✗	✗	✓	✗	$T_1 \rightarrow T_2$	2	$T_1 \rightarrow T_2$
15	✗	✗	✗	✓	$T_1 \leftrightarrow T_2$	4	$T_1 \leftrightarrow T_2$
16	✗	✗	✗	✗	$T_1 \parallel T_2$	5	$T_1 \parallel T_2$

Table 2. T_1 and T_2 are two cell-tuples. Columns 2-5 represent the four conditions for the two cell-tuples. Columns $\mathcal{I}$ and $\mathcal{R}$ show the ordering of the cell-tuples for $\mathcal{I}$ and $\mathcal{R}$ algorithms, respectively. The order of cell updates of $\mathcal{R}$ is consistent with $\mathcal{I}$.

Property 1 (One-way sweep). *An $\mathcal{I}$-$\mathcal{DP}$ for a DP table C is said to satisfy the one-way sweep property if the following holds: $\forall$ cells $x, y \in C$, if x depends on y, then y is fully updated before x reads from y.*

Property 2 (Fractal property). *An $\mathcal{I}$-$\mathcal{DP}$ satisfies the fractal property if the following holds. Let S_n and S_{2n} be the cell-sets of the $\mathcal{I}$-$\mathcal{DP}$ for DP tables $[0..n-1]^d$ and $[0..2n-1]^d$, respectively, where $n \geq 2^k$ (k is the problem-specific natural number). Generate the cell-set S'_n from S_{2n} by replacing every coordinate value j with $\lfloor j/2 \rfloor$ and then retaining only the distinct tuples. Then, $S_n = S'_n$.*

Definition 2 ($\mathcal{FRACTAL}$-$\mathcal{DP}$ **class).** *An $\mathcal{I}$-$\mathcal{DP}$ is said to be in the $\mathcal{FRACTAL}$-$\mathcal{DP}$ class if the following conditions hold: (a) the $\mathcal{I}$-$\mathcal{DP}$ satisfies the one-way sweep property (Prop. 1), (b) the $\mathcal{I}$-$\mathcal{DP}$ satisfies the fractal property (Prop. 2), and (c) The cell-tuple size $(1+s)$ is upper-bounded by a constant.*

Theorem 1 (Correctness). Given an $\mathcal{I}$-$\mathcal{DP}$ from the $\mathcal{FRACTAL}$-$\mathcal{DP}$ class as input, $\mathcal{AUTOGEN}$ generates an $\mathcal{R}$-$\mathcal{DP}$ that is functionally equivalent to the given $\mathcal{I}$-$\mathcal{DP}$.

Proof. Let the $\mathcal{I}$-$\mathcal{DP}$ and $\mathcal{R}$-$\mathcal{DP}$ algorithms for a problem $\mathcal{P}$ be denoted by $\mathcal{I}$ and $\mathcal{R}$, respectively. We use mathematical induction to prove the correctness of $\mathcal{AUTOGEN}$ in d-D, assuming d to be a constant. We first prove the correctness for the threshold problem size (see Section 2.3), i.e., $n = 2^q$ for some $q \in \mathbb{N}$ and then show that if the algorithm is correct for $n = 2^r$, for any $r \geq q$ then it is also correct for $n = 2^{r+1}$.

Basis. To prove that $\mathcal{AUTOGEN}$ is correct for $n = 2^q$, we have to show the following three: (a) Number of nodes in the algorithm-tree is $\mathcal{O}(1)$, (b) Both $\mathcal{I}$ and $\mathcal{R}$ apply the same set of cell updates, and (c) $\mathcal{R}$ never violates the one-way sweep property (Prop. 1).

(a) *The size of the algorithm-tree is $\mathcal{O}(1)$.*
A node is a set of one or more region-tuples (see Rule 1). Two nodes with the same input and output fingerprints are given the same function names. The maximum number of possible functions is upper bounded by the product of the maximum number of possible nodes at a level $\left(\leq 2^d((2^d-1)^s+1)\right)$ and the maximum number of children a node can have $\left(\leq 2^{2^d((2^d-1)^s+1)}\right)$. The height of the tree is $\mathcal{O}(1)$ from Assumption 1 and the threshold level definition. The maximum branching factor (or the maximum number of children per node) of the tree is also upper bounded by a constant. Hence, the size of the algorithm-tree is $\mathcal{O}(1)$.

(b) *Both $\mathcal{I}$ and $\mathcal{R}$ perform the same set of cell updates.*
There is no cell-tuple of $\mathcal{I}$ that is not considered by $\mathcal{R}$. In §2.2, we split the entire cell-set into subsets of cell-tuples, subsubsets of cell-tuples and so on to represent the different region-tuples. As per the rules of construction of the algorithm-tree, all cell-tuples of $\mathcal{I}$ are considered by $\mathcal{R}$.

There is no cell-tuple of $\mathcal{R}$ that is not considered by $\mathcal{I}$. Let there be a cell-tuple T in $\mathcal{R}$ that is not present in $\mathcal{I}$. As the cell-tuples in $\mathcal{R}$ are obtained by splitting the cell-set into subsets of cell-tuples, subsubsets of cell-tuples and so on, the original cell-set should include T. This means that $\mathcal{I}$ should have generated the cell-tuple T, which contradicts our initial assumption. Hence, by contradiction, all the cell tuples of $\mathcal{R}$ are considered by $\mathcal{I}$.

(c) *$\mathcal{R}$ never violates the one-way sweep property (Prop. 1).*
We prove that for any two cell-tuples T_1 and T_2, the order of execution of T_1 and T_2 in $\mathcal{R}$ is exactly the same as that in $\mathcal{I}$ if changing the order may lead to violation of the one-way sweep property. The relationship between the tuples T_1 and T_2 can be defined exhaustively as shown in Tab. 2 with the four conditions: $\boldsymbol{W}(T_1) \in$ (or $\notin$) $\boldsymbol{R}(T_1)$, $\boldsymbol{W}(T_2) \in$ (or $\notin$) $\boldsymbol{R}(T_2)$, $\boldsymbol{W}(T_1) \in$ (or $\notin$) $\boldsymbol{R}(T_2)$, and $\boldsymbol{W}(T_1) =$ (or $\neq$) $\boldsymbol{W}(T_2)$. A few cases do not hold as the cell-tuples cannot simultaneously satisfy paradoxical conditions, e.g., cases 3, 5, 11 and 13 in Tab. 2. The relation between T_1 and T_2 can be one of the following five: (i) $T_1 = T_2$, (ii) $T_1 \rightarrow T_2$ i.e. T_1 is executed before T_2, (iii) $T_2 \rightarrow T_1$ i.e. T_2 is executed before T_1, (iv) $T_1 || T_2$ i.e. T_1 and T_2 can be executed in parallel, and (v) $T_1 \leftrightarrow T_2$ i.e. either $T_1 \rightarrow T_2$ or $T_2 \rightarrow T_1$.

Columns $\mathcal{I}$ and $\mathcal{R}$ represent the ordering of the two cell-tuples in $\mathcal{I}$ and $\mathcal{R}$ algorithms, respectively. Column $\mathcal{I}$ is filled based on the one-way sweep property (Prop. 1) and column $\mathcal{R}$ is filled based on the four rules 2, 3, 4, and 5. It is easy to see that for every case in which changing the order of execution of T_1 and T_2 may lead to the violation of the one-way sweep property, both $\mathcal{R}$ and $\mathcal{I}$ apply the updates in exactly the same order. Hence, $\mathcal{R}$ satisfies the one-way sweep property.

Induction. We show that if $\mathcal{AUTOGEN}$ is correct for a problem size of $n = 2^r$ for some $r \geq q \in \mathbb{N}$, it is also correct for $n = 2^{r+1}$.

From the previous arguments we obtained a correct algorithm $\mathcal{R}$ for $r = q$. Algorithm $\mathcal{R}$ is a set of DAGs for different functions. Let C^n and C^{2n} represent two DP tables of size n^d and $(2n)^d$, respectively, such that $n \geq 2^q$. According to Prop. 2, the dependencies among the regions C^n_{11}, C^n_{12}, C^n_{21}, C^n_{22} must be exactly same as the dependencies among the regions C^{2n}_{11}, C^{2n}_{12}, C^{2n}_{21}, C^{2n}_{22}. If they were different, then that would violate Prop. 2. Hence, the region-tuples for the two DP tables are the same. Arguing similarly, the region-tuples remain the same for the DP tables all the way down to the threshold level. In other words, the algorithm-trees for the two problem instances are exactly the same. Having the same algorithm-trees with the same dependencies implies that the DAGs for DP tables C^n and C^{2n} are the same. Therefore, if $\mathcal{AUTOGEN}$ is correct for $n = 2^r$ for some $r \geq q \in \mathbb{N}$, it is also correct for $n = 2^{r+1}$. □

3.2 Cache complexity of an $\mathcal{R}$-$\mathcal{DP}$

A recursive function is *closed* provided it does not call any other recursive function but itself, and it is *semi-closed* provided it only calls itself and other closed functions. A closed (resp. semi-closed) function $\mathcal{G}$ is *dominating* provided no other closed (resp. semi-closed) function of the given $\mathcal{R}$-$\mathcal{DP}$ makes more self-recursive calls than made by $\mathcal{G}$ and every non-closed (resp. non-semi-closed) function makes strictly fewer such calls.

Theorem 2 (**Cache complexity**). *If an $\mathcal{R}$-$\mathcal{DP}$ includes a dominating closed or semi-closed function $\mathcal{F}_k$ that calls itself recursively a_{kk} times, then the serial cache complexity of the $\mathcal{R}$-$\mathcal{DP}$ for a DP table of size n^d is*
$Q_1(n, d, B, M) = \mathcal{O}\left(T_1(n)/\left(BM^{(l_k/d)-1}\right) + S(n, d)/B + 1\right)$
under the ideal-cache model, where $l_k = \log_2 a_{kk}$, $T_1(n) =$ total work $= \mathcal{O}\left(n^{l_k}\right)$, $M =$ cache size, $B =$ block size, $M = \Omega\left(B^d\right)$, and $S(n, d) =$ space complexity $= \mathcal{O}\left(n^d\right)$.

Proof. Suppose the $\mathcal{R}$-$\mathcal{DP}$ algorithm consists of a set $\mathcal{F}$ of m recursive functions $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_m$. For $1 \leq i, j \leq m$, let a_{ij} be the number of times $\mathcal{F}_i$ calls $\mathcal{F}_j$. Then for a suitable constant $\gamma_i > 0$, the cache complexity $Q_{\mathcal{F}_i}$ of $\mathcal{F}_i$ on an input of size n^d can be computed recursively as follows:

$$Q_{\mathcal{F}_i}(n) = \begin{cases} \mathcal{O}\left(n^{d-1} + n^d/B\right) & \text{if } n^d \leq \gamma_i M, \\ \sum_{j=1}^{m} a_{ij} Q_{\mathcal{F}_j}\left(n/2\right) + \mathcal{O}\left(1\right) & \text{otherwise.} \end{cases}$$

If $\mathcal{F}_k$ is a closed function, then $Q_{\mathcal{F}_k}(n) = a_{kk} Q_{\mathcal{F}_k}\left(n/2\right) + \mathcal{O}\left(1\right)$ for $n^d > \gamma_k M$. Solving the recurrence, we get the overall (for all values of n^d) cache complexity as $Q_{\mathcal{F}_k}(n) =$
$\mathcal{O}\left(n^{l_k}/(BM^{(l_k/d)-1}) + n^d/B + 1\right)$, where $l_k = \log_2 a_{kk}$.

If $\mathcal{F}_k$ is a dominating semi-closed function, then $Q_{\mathcal{F}_k}(n) = a_{kk} Q_{\mathcal{F}_k}\left(n/2\right) + o\left(n^{l_k}/(BM^{(l_k/d)-1})\right)$ for $n^d > \gamma_k M$. For all sizes of the DP table this recurrence also solves to
$\mathcal{O}\left(n^{l_k}/(BM^{(l_k/d)-1}) + n^d/B + 1\right)$.

If $\mathcal{F}_k$ is a dominating closed (resp. semi-closed) function then (i) $a_{kk} \geq a_{ii}$ for every closed (resp. semi-closed) function $\mathcal{F}_i$, and (ii) $a_{kk} > a_{jj}$ for every non-closed (resp. non-semi-closed) function $\mathcal{F}_j$. The algorithm-tree must contain at least one path $P = \langle \mathcal{F}_{r_1}, \mathcal{F}_{r_2}, \ldots, \mathcal{F}_{r_{|P|}} \rangle$ from its root $(= \mathcal{F}_{r_1})$ to a node corresponding to $\mathcal{F}_k$ $(= \mathcal{F}_{r_{|P|}})$. Since $|P|$ is a small number independent of n, and by definition $a_{r_i r_i} < a_{r_{|P|} r_{|P|}}$ holds for every $i \in [1, |P| - 1]$, one can show that the cache complexity of every function on P must be $\mathcal{O}\left(Q_{\mathcal{F}_k}(n)\right)$. This result is obtained by moving upwards in the tree starting from $\mathcal{F}_{r_{|P|-1}}$, writing down the cache complexity recurrence for each function on this path, substituting the cache complexity results determined for functions that we have already encountered, and solving the resulting simplified recurrence. Hence, the cache complexity $Q_{\mathcal{F}_{r_1}}(n)$ of the $\mathcal{R}$-$\mathcal{DP}$ algorithm is $\mathcal{O}\left(Q_{\mathcal{F}_k}(n)\right)$. This completes the proof.

It is important to note the serial cache complexity and the total work of an $\mathcal{R}$-$\mathcal{DP}$ algorithm are related. Let $\mathcal{F}_k$ be a dominating closed function that calls itself a_{kk} number of times and let $P = \langle \mathcal{F}_{r_1}, \mathcal{F}_{r_2}, \ldots, \mathcal{F}_{r_{|P|}} \rangle$ be a path in the algorithm-tree from its root $(= \mathcal{F}_{r_1})$ to a node corresponding to $\mathcal{F}_k$ $(= \mathcal{F}_{r_{|P|}})$. Let q out of these $|P|$ functions call themselves a_{kk} times and q is maximized over all possible paths in the algorithm-tree. The work can be found by counting the total number of leaf nodes in the algorithm-tree. Using Master theorem repeatedly we can show that $T_1(n) = \mathcal{O}\left(n^{\log a_{kk}} \log^{q-1} n\right)$. □

4. Extensions of $\mathcal{AUTOGEN}$

In this section, we briefly discuss how to extend $\mathcal{AUTOGEN}$ to (i) handle one-way sweep property (Prop. 1) violation, and (ii) sometimes reduce the space usage of the generated $\mathcal{R}$-$\mathcal{DP}$ algorithms.

4.1 Handling one-way sweep property (Prop. 1) violation

The following three-step procedure works for dynamic programs that compute paths over a closed semiring in a directed graph (Ullman et al. 1974). Floyd-Warshall's algorithm for finding all-pairs shortest path (APSP) (Floyd 1962) belongs to this class and is shown in Figure 5.

(i) **Project $\mathcal{I}$-$\mathcal{DP}$ to higher dimension.** Violation of the one-way sweep property means that some cells of the DP table are computed from cells that are not yet fully updated. By allocating space to retain each intermediate value of every cell, the problem is transformed into a new problem where the cells depend on fully updated cells only. The technique effectively projects the DP on to a higher dimensional space leading to a correct $\mathcal{I}$-$\mathcal{DP}$ that satisfies the one-way sweep property.

(ii) **Autodiscover $\mathcal{R}$-$\mathcal{DP}$ from $\mathcal{I}$-$\mathcal{DP}$.** $\mathcal{AUTOGEN}$ is applied on the higher dimensional $\mathcal{I}$-$\mathcal{DP}$ that satisfies Prop. 1 to discover an $\mathcal{R}$-$\mathcal{DP}$ in the same higher dimensional space.

(iii) **Project $\mathcal{R}$-$\mathcal{DP}$ back to original dimension.** The autogenerated $\mathcal{R}$-$\mathcal{DP}$ is projected back to the original dimensional space. One can show that the projected $\mathcal{R}$-$\mathcal{DP}$ correctly implements the original $\mathcal{I}$-$\mathcal{DP}$ (Cormen et al. 2009; Chowdhury and Ramachandran 2010).

4.2 Space reduction

$\mathcal{AUTOGEN}$ can be extended to analyze and optimize the functions of an autogenerated $\mathcal{R}$-$\mathcal{DP}$ for a possible reduction in space usage. We explain through an example.

Example. The LCS problem (Hirschberg 1975; Chowdhury and Ramachandran 2006) asks one to find the longest of all common subsequences (Cormen et al. 2009) between two strings. In LCS, a cell depends on its three adjacent cells. Here, we are interested in finding the length of the LCS and not the LCS itself. Starting from the standard $\Theta\left(n^2\right)$ space $\mathcal{I}$-$\mathcal{DP}$, we generate an $\mathcal{R}$-$\mathcal{DP}$ for the problem that contains four recursive functions. The autogenerated $\mathcal{R}$-$\mathcal{DP}$ still uses $\Theta\left(n^2\right)$ space and incurs $\mathcal{O}\left(n^2/B\right)$ cache misses. $\mathcal{AUTOGEN}$ can reason as follows in order to reduce the space usage of this $\mathcal{R}$-$\mathcal{DP}$ and thereby improving its cache performance.

The autogenerated $\mathcal{R}$-$\mathcal{DP}$ has two functions of the form $\mathcal{F}(n) \mapsto \{\mathcal{F}(n/2), \mathcal{F}(n/2), \mathcal{G}(n/2)\}$, where $\mathcal{G}$ is of the form $\mathcal{G}(n) \mapsto \{\mathcal{G}(n/2)\}$. Given their dependencies, it is easy that in $\mathcal{G}$, the top-left cell of bottom-right quadrant depends on the bottom-right cell of the top-left quadrant. Also, in $\mathcal{F}$, the leftmost (resp. topmost) boundary cells of one quadrant depends on the rightmost (resp. bottommost) quadrant of adjacent quadrant. When there is only a dependency on the boundary cells, we can copy the values of the boundary cells, which occupies $\mathcal{O}\left(n\right)$ space, between different function calls and we no longer require quadratic space. At each level of the recursion tree $\mathcal{O}\left(n\right)$ space is used, and the total space for the parallel $\mathcal{R}$-$\mathcal{DP}$ algorithm is $\mathcal{O}\left(n \log n\right)$. This new $\mathcal{R}$-$\mathcal{DP}$ algorithm will have a single function and its cache complexity improves to $\mathcal{O}\left(n^2/(BM)\right)$. Space usage can be reduced further to $\mathcal{O}\left(n\right)$ by simply reusing space between parent and child functions. Cache complexity remains $\mathcal{O}\left(n^2/(BM)\right)$.

5. Experimental Results

This section presents empirical results showing the performance benefits and robustness of $\mathcal{AUTOGEN}$-discovered algorithms.

5.1 Experimental setup

All our experiments were performed on a multicore machine with dual-socket 8-core 2.7 GHz[5] Intel Sandy Bridge processors (2 × 8 = 16 cores in total) and 32 GB RAM. Each core was connected

[5] All energy, adaptivity and robustness experiments were performed on a Sandy Bridge machine with a processor speed 2.00GHz.

```
LOOP-FLOYD-WARSHALL-APSP-3D($D, n$)
for $k \leftarrow 1$ to $n$
  for $i \leftarrow 1$ to $n$
    for $j \leftarrow 1$ to $n$
      $D[i, j, k] \leftarrow \min\left(D[i, j, k-1], D[i, k, k-1] + D[k, j, k-1]\right)$
```

```
$\mathcal{A}^{3D}_{FW}(\langle X, Y, X, X \rangle)$
if $X$ is a small matrix then $\mathcal{A}^{3D}_{loop\text{-}FW}(\langle X, Y, X, X \rangle)$
else
  $\mathcal{A}^{3D}_{FW}(\langle X_{111}, Y_{112}, X_{111}, X_{111} \rangle)$
  par: $\mathcal{B}^{3D}_{FW}(\langle X_{121}, Y_{122}, X_{111}, X_{121} \rangle), \mathcal{C}^{3D}_{FW}(\langle X_{211}, Y_{212}, X_{211}, X_{111} \rangle)$
  $\mathcal{D}^{3D}_{FW}(\langle X_{221}, Y_{222}, X_{211}, X_{121} \rangle)$
  $\mathcal{A}^{3D}_{FW}(\langle X_{222}, X_{221}, X_{222}, X_{222} \rangle)$
  par: $\mathcal{B}^{3D}_{FW}(\langle X_{212}, X_{211}, X_{222}, X_{212} \rangle), \mathcal{C}^{3D}_{FW}(\langle X_{122}, X_{121}, X_{122}, X_{222} \rangle)$
  $\mathcal{D}^{3D}_{FW}(\langle X_{112}, X_{111}, X_{122}, X_{212} \rangle)$
```

```
$\mathcal{B}^{3D}_{FW}(\langle X, Y, U, X \rangle)$
if $X$ is a small matrix then $\mathcal{B}^{3D}_{loop\text{-}FW}(\langle X, Y, U, X \rangle)$
else
  par: $\mathcal{B}^{3D}_{FW}(\langle X_{111}, Y_{112}, U_{111}, X_{111} \rangle), \mathcal{B}^{3D}_{FW}(\langle X_{121}, Y_{122}, U_{111}, X_{121} \rangle)$
  par: $\mathcal{D}^{3D}_{FW}(\langle X_{211}, Y_{212}, U_{211}, X_{111} \rangle), \mathcal{D}^{3D}_{FW}(\langle X_{221}, Y_{222}, U_{211}, X_{121} \rangle)$
  par: $\mathcal{B}^{3D}_{FW}(\langle X_{212}, X_{211}, U_{222}, X_{212} \rangle), \mathcal{B}^{3D}_{FW}(\langle X_{222}, X_{221}, U_{222}, X_{222} \rangle)$
  par: $\mathcal{D}^{3D}_{FW}(\langle X_{112}, X_{111}, U_{122}, X_{212} \rangle), \mathcal{D}^{3D}_{FW}(\langle X_{122}, X_{121}, U_{122}, X_{222} \rangle)$
```

```
$\mathcal{C}^{3D}_{FW}(\langle X, Y, X, V \rangle)$
if $X$ is a small matrix then $\mathcal{C}^{3D}_{loop\text{-}FW}(\langle X, Y, X, V \rangle)$
else
  par: $\mathcal{C}^{3D}_{FW}(\langle X_{111}, Y_{112}, X_{111}, V_{111} \rangle), \mathcal{C}^{3D}_{FW}(\langle X_{211}, Y_{212}, X_{211}, V_{111} \rangle)$
  par: $\mathcal{D}^{3D}_{FW}(\langle X_{121}, Y_{122}, X_{111}, V_{121} \rangle), \mathcal{D}^{3D}_{FW}(\langle X_{221}, Y_{222}, X_{211}, V_{121} \rangle)$
  par: $\mathcal{C}^{3D}_{FW}(\langle X_{122}, X_{121}, X_{122}, V_{222} \rangle), \mathcal{C}^{3D}_{FW}(\langle X_{222}, X_{221}, X_{222}, V_{222} \rangle)$
  par: $\mathcal{D}^{3D}_{FW}(\langle X_{112}, X_{111}, X_{122}, V_{212} \rangle), \mathcal{D}^{3D}_{FW}(\langle X_{212}, X_{211}, X_{222}, V_{122} \rangle)$
```

```
$\mathcal{D}^{3D}_{FW}(\langle X, Y, U, V \rangle)$
if $X$ is a small matrix then $\mathcal{D}^{3D}_{loop\text{-}FW}(\langle X, Y, U, V \rangle)$
else
  par: $\mathcal{D}^{3D}_{FW}(\langle X_{111}, Y_{112}, U_{111}, V_{111} \rangle), \mathcal{D}^{3D}_{FW}(\langle X_{121}, Y_{122}, U_{111}, V_{121} \rangle),$
       $\mathcal{D}^{3D}_{FW}(\langle X_{211}, Y_{212}, U_{211}, V_{111} \rangle), \mathcal{D}^{3D}_{FW}(\langle X_{221}, Y_{222}, U_{211}, V_{121} \rangle)$
  par: $\mathcal{D}^{3D}_{FW}(\langle X_{112}, X_{111}, U_{122}, V_{212} \rangle), \mathcal{D}^{3D}_{FW}(\langle X_{122}, X_{121}, U_{122}, V_{222} \rangle),$
       $\mathcal{D}^{3D}_{FW}(\langle X_{212}, X_{211}, U_{222}, V_{212} \rangle), \mathcal{D}^{3D}_{FW}(\langle X_{222}, X_{221}, U_{222}, V_{222} \rangle)$
```

(a)

```
LOOP-FLOYD-WARSHALL-APSP($D, n$)
for $k \leftarrow 1$ to $n$
  for $i \leftarrow 1$ to $n$
    for $j \leftarrow 1$ to $n$
      $D[i, j] \leftarrow \min\left(D[i, j], D[i, k] + D[k, j]\right)$
```

```
$\mathcal{A}_{FW}(\langle X, X, X \rangle)$
if $X$ is a small matrix then $\mathcal{A}_{loop\text{-}FW}(\langle X, X, X \rangle)$
else
  $\mathcal{A}_{FW}(\langle X_{11}, X_{11}, X_{11} \rangle)$
  par: $\mathcal{B}_{FW}(\langle X_{12}, X_{11} X_{12} \rangle), \mathcal{C}_{FW}(\langle X_{21}, X_{21}, X_{11} \rangle)$
  $\mathcal{D}_{FW}(\langle X_{22}, X_{21}, X_{12} \rangle)$
  $\mathcal{A}_{FW}(\langle X_{22}, X_{22}, X_{22} \rangle)$
  par: $\mathcal{B}_{FW}(\langle X_{21}, X_{22}, X_{21} \rangle), \mathcal{C}_{FW}(\langle X_{12}, X_{12}, X_{22} \rangle)$
  $\mathcal{D}_{FW}(\langle X_{11}, X_{12}, X_{21} \rangle)$
```

```
$\mathcal{B}_{FW}(\langle X, U, X \rangle)$
if $X$ is a small matrix then $\mathcal{B}_{loop\text{-}FW}(\langle X, U, X \rangle)$
else
  par: $\mathcal{B}_{FW}(\langle X_{11}, U_{11}, X_{11} \rangle), \mathcal{B}_{FW}(\langle X_{12}, U_{11}, X_{12} \rangle)$
  par: $\mathcal{D}_{FW}(\langle X_{21}, U_{21}, X_{11} \rangle), \mathcal{D}_{FW}(\langle X_{22}, U_{21}, X_{12} \rangle)$
  par: $\mathcal{B}_{FW}(\langle X_{21}, U_{22}, X_{21} \rangle), \mathcal{B}_{FW}(\langle X_{22}, U_{22}, X_{22} \rangle)$
  par: $\mathcal{D}_{FW}(\langle X_{11}, U_{12}, X_{21} \rangle), \mathcal{D}_{FW}(\langle X_{12}, U_{12}, X_{22} \rangle)$
```

```
$\mathcal{C}_{FW}(\langle X, X, V \rangle)$
if $X$ is a small matrix then $\mathcal{C}_{loop\text{-}FW}(\langle X, X, V \rangle)$
else
  par: $\mathcal{C}_{FW}(\langle X_{11}, X_{11}, V_{11} \rangle), \mathcal{C}_{FW}(\langle X_{21}, X_{21}, V_{11} \rangle)$
  par: $\mathcal{D}_{FW}(\langle X_{12}, X_{11}, V_{12} \rangle), \mathcal{D}_{FW}(\langle X_{22}, X_{21}, V_{12} \rangle)$
  par: $\mathcal{C}_{FW}(\langle X_{12}, X_{12}, V_{22} \rangle), \mathcal{C}_{FW}(\langle X_{22}, X_{22}, V_{22} \rangle)$
  par: $\mathcal{D}_{FW}(\langle X_{11}, X_{12}, V_{21} \rangle), \mathcal{D}_{FW}(\langle X_{21}, X_{22}, V_{12} \rangle)$
```

```
$\mathcal{D}_{FW}(\langle X, U, V \rangle)$
if $X$ is a small matrix then $\mathcal{D}_{loop\text{-}FW}(\langle X, U, V \rangle)$
else
  par: $\mathcal{D}_{FW}(\langle X_{11}, U_{11}, V_{11} \rangle), \mathcal{D}_{FW}(\langle X_{12}, U_{11}, V_{12} \rangle),$
       $\mathcal{D}_{FW}(\langle X_{21}, U_{21}, V_{11} \rangle), \mathcal{D}_{FW}(\langle X_{22}, U_{21}, V_{12} \rangle)$
  par: $\mathcal{D}_{FW}(\langle X_{11}, U_{12}, V_{21} \rangle), \mathcal{D}_{FW}(\langle X_{12}, U_{12}, V_{22} \rangle),$
       $\mathcal{D}_{FW}(\langle X_{21}, U_{22}, V_{21} \rangle), \mathcal{D}_{FW}(\langle X_{22}, U_{22}, V_{22} \rangle)$
```

(b)

Figure 5. (a) An autogenerated $\mathcal{R}$-$\mathcal{DP}$ algorithm from the cubic space Floyd-Warshall's APSP algorithm. In the initial call to $\mathcal{A}^{3D}_{FW}(\langle X, Y, X, X \rangle)$, X points to $D[1..n, 1..n, 1..n]$ and Y points to an n^3 matrix whose topmost plane is initialized with $D[1..n, 1..n, 0]$. (b) An $\mathcal{R}$-$\mathcal{DP}$ algorithm obtained by projecting the 3D matrix $D[1..n, 1..n, 0..n]$ accessed by the algorithm in column (a) to its 2D base $D[1..n, 1..n, 0]$.

to a 32 KB private L1 cache and a 256 KB private L2 cache. All cores in a processor shared a 20 MB L3 cache. All algorithms were implemented in C++. We used Intel Cilk Plus extension to parallelize and Intel® C++ Compiler v13.0 to compile all implementations with optimzation parameters `-O3 -ipo -parallel -AVX -xhost`. PAPI 5.3 (PAP) was used to count cache misses, and the MSR (Model-Specific Register) module and `likwid` (Treibig et al. 2010) were used for energy measurements. We used `likwid` for the adaptivity (Figure 8) experiments. All `likwid` measurements were end-to-end (i.e., captures everything from the start to the end of the program).

Given an iterative description of a DP in the $\mathcal{FRACTAL}$-$\mathcal{DP}$ class, our $\mathcal{AUTOGEN}$ prototype generates pseudocode of the corresponding $\mathcal{R}$-$\mathcal{DP}$ algorithm in the format shown in Figure 1. We implemented such autodiscovered $\mathcal{R}$-$\mathcal{DP}$ algorithms for the parenthesis problem, gap problem, and Floyd-Warshall's APSP (2-D). In order to avoid overhead of recursion and increase vectorization efficiency the $\mathcal{R}$-$\mathcal{DP}$ implementation switched to an iterative kernel when the problem size became sufficiently small (e.g., when problem size reached 64 × 64). All our $\mathcal{R}$-$\mathcal{DP}$ implementations were the straightforward implementation of the pseudocode with only *trivial hand-optimizations*. With nontrivial hand-optimizations $\mathcal{R}$-$\mathcal{DP}$ algorithms can achieve even more speedup (see (Tithi et al. 2015)). Trivial optimizations include: (i) copy-optimization – copying transpose of a column-major input matrix inside a basecase to a local array, so that it can be accessed in unit stride during actual computation, (ii) using registers for variables that are accessed many times inside the loops, (iii) write optimization in the basecase – if each iteration of an innermost loop updates the same location of the DP table we perform all those updates in a local variable instead of modifying the DP table cell over and over again, and update that cell only once using the updated local variable after the loop terminates, and (iv) using `#pragma` directives to auto-vectorize / auto-parallelize code. Nontrivial optimizations that we did not apply include: (i) using Z-morton row-major layout (see (Tithi et al. 2015)) to store the matrices, (ii) using pointer arithmetic and converting all multiplicative indexing to additive indexing, and (iii) using explicit vectorization.

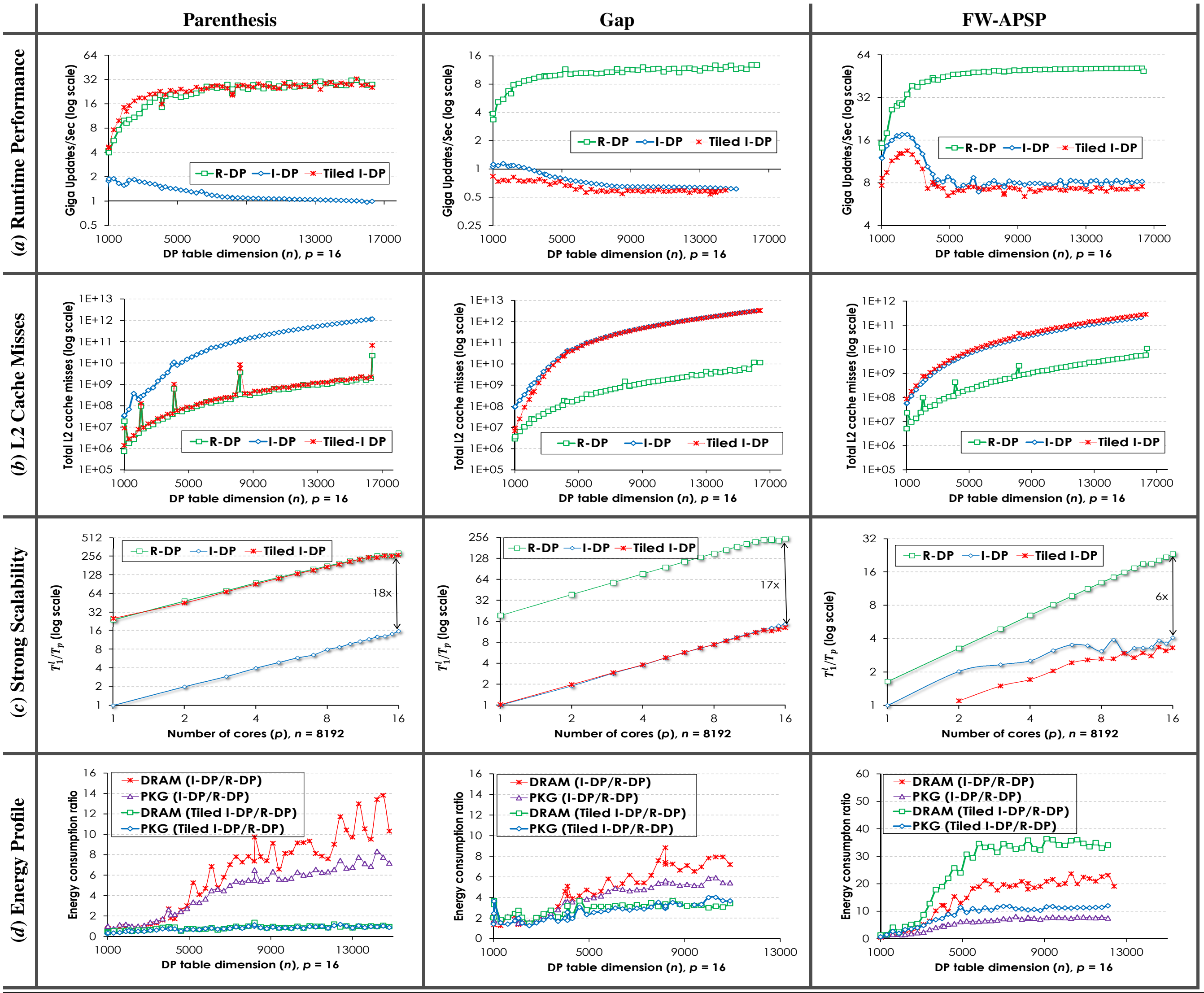


Figure 6. Performance comparison of $\mathcal{I}\text{-}\mathcal{DP}$, $\mathcal{R}\text{-}\mathcal{DP}$ and tiled $\mathcal{I}\text{-}\mathcal{DP}$: (a) Giga updates per second achieved by all algorithms, (b) L2 cache misses for each program, (c) strong scalability with #cores, p when n is fixed at 8192 (in this plot T_1^l denotes the running time of $\mathcal{I}\text{-}\mathcal{DP}$ when $p = 1$), and (d) ratios of total joule energy consumed by Package (PKG) and DRAM. Here, tiled $\mathcal{I}\text{-}\mathcal{DP}$ is an optimized version of the parallel tiled code generated by Pluto (Bondhugula et al. 2008).

The major optimizations applied on $\mathcal{I}\text{-}\mathcal{DP}$ codes include the following: parallelization, use of pragmas (e.g., `#pragma ivdep` and `#pragma parallel`), use of 64 byte-aligned matrices, write optimizations, pointer arithmetic, and additive indexing.

We used Pluto (Bondhugula et al. 2008) – a state-of-the-art polyhedral compiler – to generate parallel tiled iterative codes for the parenthesis problem, gap problem, and Floyd-Warshall's APSP (2-D). Optimized versions of these codes are henceforth called *tiled* $\mathcal{I}\text{-}\mathcal{DP}$. After analyzing the autogenerated codes, we found that the parenthesis implementation had temporal locality as it was tiled across all three dimensions, but FW-APSP and gap codes did not as the dependence-based standard tiling conditions employed by Pluto allowed tiling of only two of the three dimensions for those problems. While both parenthesis and FW-APSP codes had spatial locality, the gap implementation did not as it was accessing data in both row- and column-major orders. Overall, for any given cache level the theoretical cache-complexity of the tiled parenthesis code matched that of parenthesis $\mathcal{R}\text{-}\mathcal{DP}$ assuming that the tile size was optimized for that cache level. But tiled FW-APSP and tiled gap had nonoptimal cache complexities. Indeed, the cache complexity of tiled FW-APSP turned out to be $\Theta\left(n^3/B\right)$ matching the cache complexity of its $\mathcal{I}\text{-}\mathcal{DP}$ counterpart. Similarly, the $\Theta\left(n^3\right)$ cache complexity of tiled gap matched that of $\mathcal{I}\text{-}\mathcal{DP}$ gap.

The major optimizations we applied on the parallel tiled codes generated by Pluto include (i) use of `#pragma ivdep`, `#pragma parallel`, and `#pragma min loop count(B)` directives; (ii) write optimizations (as was used for basecases of $\mathcal{R}\text{-}\mathcal{DP}$); (iii) use of empirically determined best tile sizes, and (iv) rigorous optimizations using pointer arithmetic, additive indexing, etc. The type of trivial copy optimization we used in $\mathcal{R}\text{-}\mathcal{DP}$ did not improve spatial locality of the autogenerated tiled $\mathcal{I}\text{-}\mathcal{DP}$ for the gap problem as the code did not have any temporal locality. The code generated for FW-APSP had only one parallel loop, whereas two loops could be parallelized trivially. In all our experiments we used two parallel loops for FW-APSP. The direction of the outermost loop of

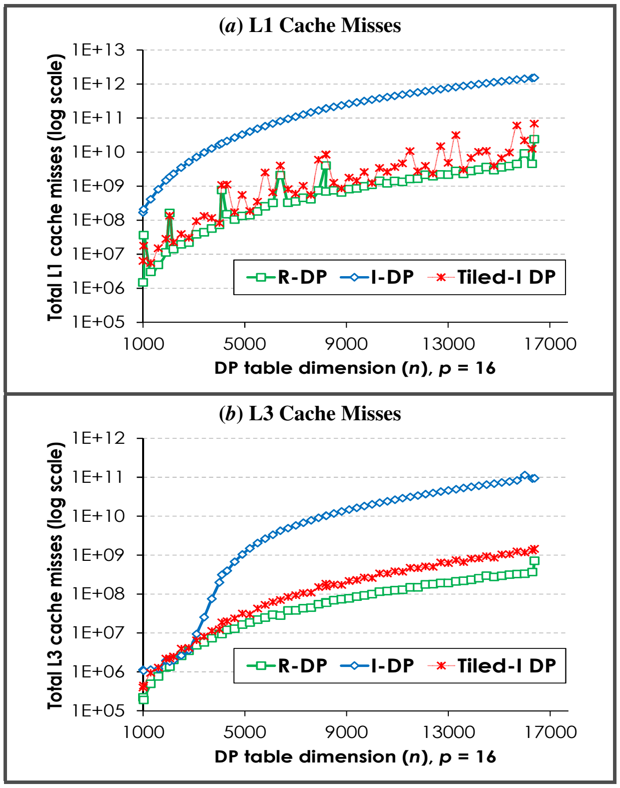


Figure 7. The plots show the L1 and L3 cache misses incurred by the three algorithms for solving the parenthesis problem. L2 cache misses are shown in Figure 6(b).

the autogenerated tiled code for the parenthesis problem had to be reversed in order to avoid violation of dependency constraints.

All algorithms we have tested are in-place, that is, they use only a constant number of extra memory/register locations in addition to the given DP table. The copy optimization requires the use of a small local submatrix per thread but its size is also independent of the input DP table size. None of our optimizations reduces space usage. The write optimization avoids directly writing to the same DP table location in the memory over and over again by collecting all those updates in a local register and then writing the final value of the register to the DP cell.

In the following part of the section, we first show performance of $\mathcal{R}$-$\mathcal{DP}$, $\mathcal{I}$-$\mathcal{DP}$ and tiled $\mathcal{I}$-$\mathcal{DP}$ implementations for all three problems when each of the programs runs on a dedicated machine. We show that $\mathcal{R}$-$\mathcal{DP}$ outperforms $\mathcal{I}$-$\mathcal{DP}$ in terms of runtime, scalability, cache-misses, and energy consumption. Next, we show the how the performance of $\mathcal{R}$-$\mathcal{DP}$, $\mathcal{I}$-$\mathcal{DP}$ and tiled $\mathcal{I}$-$\mathcal{DP}$ implementations change in a multiprogramming environment when multiple processes share cache space and bandwidth.

5.2 Single-process performance

Figure 6 shows detailed performance results of $\mathcal{I}$-$\mathcal{DP}$, tiled $\mathcal{I}$-$\mathcal{DP}$ and $\mathcal{R}$-$\mathcal{DP}$ implementations. For each of the three problems, our $\mathcal{R}$-$\mathcal{DP}$ implementations outperformed its $\mathcal{I}$-$\mathcal{DP}$ counterpart, and for $n = 8192$, the speedup factors w.r.t. parallel $\mathcal{I}$-$\mathcal{DP}$ on 16 cores were around 18, 17 and 6 for parenthesis, gap and Floyd-Warshall's APSP, respectively. For pathenthesis and gap problems $\mathcal{I}$-$\mathcal{DP}$ consumed 5.5 times more package energy and 7.4 times more DRAM energy than $\mathcal{R}$-$\mathcal{DP}$ when $n = 8192$. For Floyd-Warshall's APSP those two factors were 7.4 and 18, respectively.

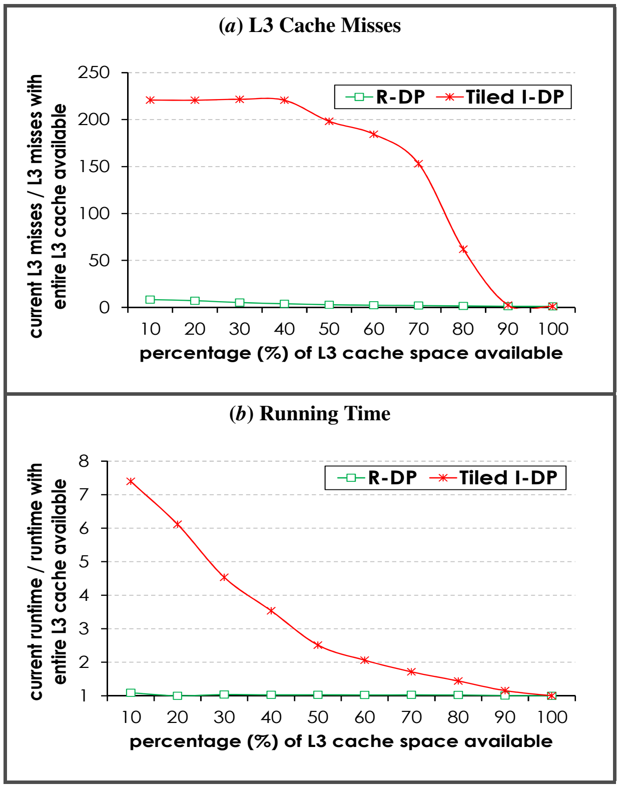


Figure 9. The plots show how changes in the available shared L3 cache space affect (a) the number of L3 cache misses, and (b) the serial running time of the tiled looping code and the recursive code solving the parenthesis problem for $n = 2^{13}$. The code under test was run on a single core of an 8-core Intel Sandy Bridge processor with 20MB shared L3 cache. A multi-threaded `Cache Pirate` (Eklov et al. 2011) was run on the remaining cores.

For the parenthesis problem tiled $\mathcal{I}$-$\mathcal{DP}$ (i.e., our optimized version of Pluto-generated parallel tiled code) and $\mathcal{R}$-$\mathcal{DP}$ had almost identical performance for $n > 6000$. For $n \leq 6000$, $\mathcal{R}$-$\mathcal{DP}$ was slower than tiled $\mathcal{I}$-$\mathcal{DP}$, but for larger n, $\mathcal{R}$-$\mathcal{DP}$ was marginally $(1$ - $2\%)$ faster on average. Observe that though tiled $\mathcal{I}$-$\mathcal{DP}$ and $\mathcal{R}$-$\mathcal{DP}$ had almost similar L2 cache performance, Figure 7 shows that $\mathcal{R}$-$\mathcal{DP}$ incurred noticably fewer L1 and L2 cache misses than those incurred by tiled $\mathcal{I}$-$\mathcal{DP}$ which helped $\mathcal{R}$-$\mathcal{DP}$ to eventually fully overcome the overhead of recursion and other implementation overheads. This happened because the tile size of tiled $\mathcal{I}$-$\mathcal{DP}$ was optimized for the L2 cache, but $\mathcal{R}$-$\mathcal{DP}$ being cache-oblivious was able to adapt to all levels of the cache hierarchy simultaneously (Frigo et al. 1999).

As explained in Section 5.1 for the gap problem tiled $\mathcal{I}$-$\mathcal{DP}$ had suboptimal cache complexity matching that of $\mathcal{I}$-$\mathcal{DP}$. As a result, tiled $\mathcal{I}$-$\mathcal{DP}$'s performance curves were closer to those of $\mathcal{I}$-$\mathcal{DP}$ than $\mathcal{R}$-$\mathcal{DP}$, and $\mathcal{R}$-$\mathcal{DP}$ outperformed it by a wide margin. Similarly for Floyd-Warshall's APSP. However, in case of gap problem tiled $\mathcal{I}$-$\mathcal{DP}$ incurred significantly fewer L3 misses than $\mathcal{I}$-$\mathcal{DP}$ (not shown in the plots), and as a result, consumed less DRAM energy. The opposite was true for Floyd-Warshall's APSP.

5.3 Multi-process performance

$\mathcal{R}$-$\mathcal{DP}$ algorithms are more robust than both $\mathcal{I}$-$\mathcal{DP}$ and tiled $\mathcal{I}$-$\mathcal{DP}$. Our empirical results show that in a multiprogramming environment $\mathcal{R}$-$\mathcal{DP}$ algorithms are less likely to significantly slowdown when the available shared cache/memory space reduces (unlike

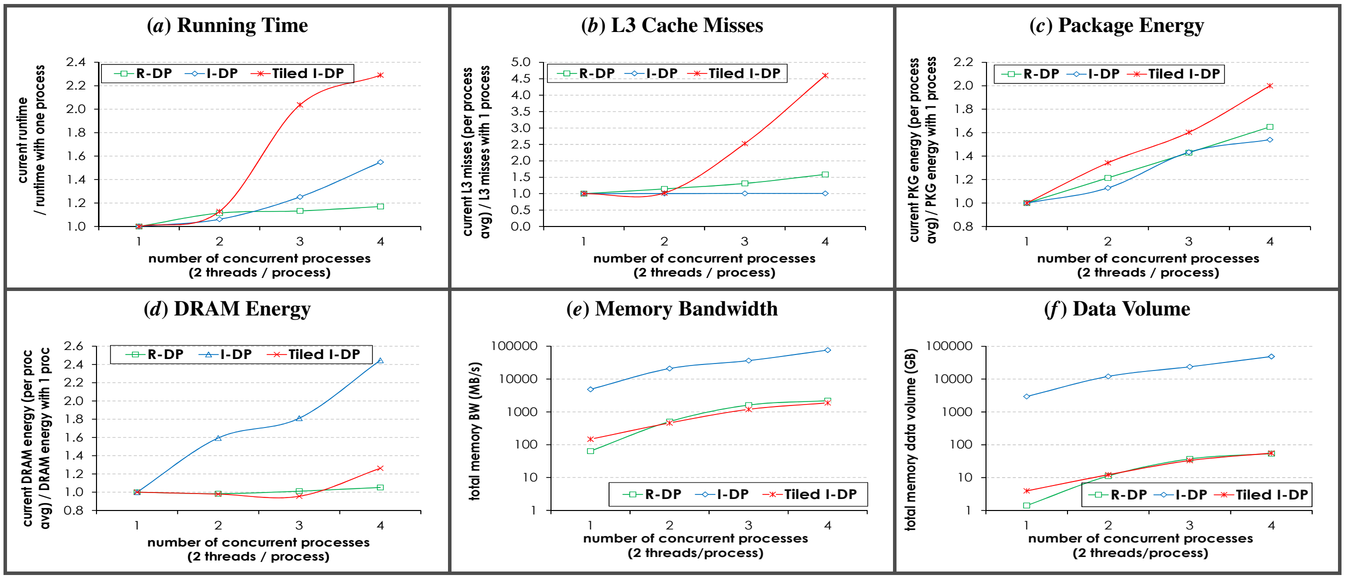


Figure 8. The plots show how the performances of standard looping, tiled looping and recursive codes for the parenthesis problem (for $n = 2^{13}$) are affected as multiple instances of the same program are run on an 8-core Intel Sandy Bridge with 20MB shared L3 cache.

tiled code with temporal locality), and less likely to suffer when the available bandwidth reduces (unlike standard $\mathcal{I}$-$\mathcal{DP}$ code and tiled $\mathcal{I}$-$\mathcal{DP}$ without temporal locality). Figures 8 and 9 show the results. For lack of space we have included only results for the parenthesis problem. We have seen similar trends for our other benchmark problems (e.g., FW-APSP).

We have performed experimental analyses of how the performance of a program ($\mathcal{R}$-$\mathcal{DP}$, $\mathcal{I}$-$\mathcal{DP}$, and tiled $\mathcal{I}$-$\mathcal{DP}$) changes if multiple copies of the same program are run on the same multicore processor (Figure 8). We ran up to 4 instances of the same program on an 8-core Sandy Bridge processor with 2 threads (i.e., cores) per process. The block size of the tiled code was optimized for best performance with 2 threads. With 4 concurrent processes $\mathcal{I}$-$\mathcal{DP}$ slowed down by 82% and tiled $\mathcal{I}$-$\mathcal{DP}$ by 46%, but $\mathcal{R}$-$\mathcal{DP}$ lost only 17% of its performance (see Figure 8). The slowdown of the tiled code resulted from its inability to adapt to the loss in the shared cache space which increased its L3 misses by a factor of 4 (see Figure 8). On the other hand, L3 misses incurred by $\mathcal{R}$-$\mathcal{DP}$ increased by less than a factor of 2.5. Since $\mathcal{I}$-$\mathcal{DP}$ does not have any temporal locality, loss of cache space did not significantly change the number of L3 misses it incurred. But $\mathcal{I}$-$\mathcal{DP}$ already incurred 90 times more L3 misses than $\mathcal{R}$-$\mathcal{DP}$, and with 4 such concurrent processes the pressure on the DRAM bandwidth increased considerably (see Figure 8) causing significant slowdown of the program.

We also report changes in energy consumption of the processes as the number of concurrent processes increases (Figure 8). Energy values were measured using `likwid-perfctr` (included in `likwid`) which reads them from the MSR registers. The energy measurements were end-to-end (start to end of the program). Three types of energy were measured: `package energy` which is the energy consumed by the entire processor die, `PP0 energy` which is the energy consumed by all cores and private caches, and finally `DRAM energy` which is the energy consumed by the directly-attached DRAM. We omitted the PP0 energy since the curves almost always look similar to that of package energy. A single instance of tiled $\mathcal{I}$-$\mathcal{DP}$ consumed 5% less energy than an $\mathcal{R}$-$\mathcal{DP}$ instance while $\mathcal{I}$-$\mathcal{DP}$ consumed 9 times more energy. Average package and PP0 energy consumed by tiled $\mathcal{I}$-$\mathcal{DP}$ increased at a faster rate than that by $\mathcal{R}$-$\mathcal{DP}$ as the number of processes increased. This happened because both its running time and L3 performance degraded faster than $\mathcal{R}$-$\mathcal{DP}$ both of which contribute to energy performance. However, since for $\mathcal{I}$-$\mathcal{DP}$ L3 misses did not change much with the increase in the number of processes, its package and PP0 energy consumption increased at a slower rate compared to $\mathcal{R}$-$\mathcal{DP}$'s when number of processes is less than 3. However, as the number of processes increases, energy consumption increases for $\mathcal{I}$-$\mathcal{DP}$ at a faster rate, and perhaps because of the DRAM bandwidth contention its DRAM energy increased significantly.

We have measured the effect on running times and L3 cache misses of serial $\mathcal{R}$-$\mathcal{DP}$ and serial tiled $\mathcal{I}$-$\mathcal{DP}$[6] when the available shared L3 cache space is reduced (shown in Figure 9). In this case, the serial tiled-$\mathcal{I}$-$\mathcal{DP}$ algorithm was running around 50% faster than the serial $\mathcal{R}$-$\mathcal{DP}$ code. The `Cache Pirate` tool (Eklov et al. 2011) was used to steal cache space[7]. When the available cache space was reduced to 50%, the number of L3 misses incurred by the tiled code increased by a factor of 22, but for $\mathcal{R}$-$\mathcal{DP}$ the increase was only 17%. As a result, the tiled $\mathcal{I}$-$\mathcal{DP}$ slowed down by over 50% while for $\mathcal{R}$-$\mathcal{DP}$ the slowdown was less than 3%. Thus $\mathcal{R}$-$\mathcal{DP}$ automatically adapts to cache sharing (Bender et al. 2014), but the tiled $\mathcal{I}$-$\mathcal{DP}$ does not. This result can be found in the second column of Figure 8.

Acknowledgments

Chowdhury and Ganapathi were supported in part by NSF grants CCF-1162196 and CCF-1439084. Kuszmaul and Leiserson were supported in part by NSF grants CCF-1314547, CNS-1409238, and IS-1447786, NSA grant H98230-14-C-1424, and FoxConn. Solar-Lezama's work was partially supported by DOE Office of Science award #DE-SC0008923. Bachmeier was supported by MIT's Undergraduate Research Opportunities Program (UROP).

Part of this work used the Extreme Science and Engineering Discovery Environment (XSEDE) (XSE; Towns et al. 2014), which is supported by NSF grant ACI-1053575.

We thank Uday Bondhugula and anonymous reviewers for valuable comments and suggestions that have significantly improved the paper.

[6] with tile size optimized for best serial performance

[7] `Cache Pirate` allows only a single program to run, and does not reduce bandwidth.

References

Performance Application Programming Interface (PAPI). `http://icl.cs.utk.edu/papi/`.

XSEDE: Extreme Science and Engineering Discovery Environment. `http://www.xsede.org/`.

N. Ahmed and K. Pingali. Automatic generation of block-recursive codes. In *Euro-Par*, pages 368–378, 2000.

V. Bafna and N. Edwards. On de novo interpretation of tandem mass spectra for peptide identification. In *Proc. RCMB*, pages 9–18, 2003.

R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

M. Bender, R. Ebrahimi, J. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In *SODA*, 2014.

U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *ACM SIGPLAN Notices*, 43(6):101–113, 2008.

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

R. Chowdhury and P. Ganapathi. Divide-and-conquer variants of bubble, selection, and insertion sorts. Unpublished manuscript.

R. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proc. SODA*, pages 591–600, 2006.

R. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *Proc. SPAA*, pages 207–216, 2008.

R. Chowdhury and V. Ramachandran. The cache-oblivious Gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. *TOCS*, 47(4):878–919, 2010.

R. Chowdhury, P. Ganapathi, V. Pradhan, J. J. Tithi, and Y. Xiao. An efficient cache-oblivious parallel viterbi algorithm. Unpublished manuscript.

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

J. Du, C. Yu, J. Sun, C. Sun, S. Tang, and Y. Yin. EasyHPS: A multilevel hybrid parallel system for dynamic programming. In *Proc. IPDPSW*, pages 630–639, 2013.

F. C. Duckworth and A. J. Lewis. A fair method for resetting the target in interrupted one-day cricket matches. *JORS*, 49(3):220–227, 1998.

R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache pirating: Measuring the curse of the shared cache. In *Proc. ICPP*, pages 165–175, 2011.

R. W. Floyd. Algorithm 97: shortest path. *CACM*, 5(6):345, 1962.

M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. FOCS*, pages 285–297, 1999.

Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than $O(1)$ dependency. *JPDC*, 21(2):213–222, 1994.

R. Giegerich and G. Sauthoff. Yield grammar analysis in the Bellman's GAP compiler. In *Proc. LDTA*, page 7, 2011.

D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *CACM*, 18(6):341–343, 1975.

J. O. S. Kennedy. Applications of dynamic programming to agriculture, forestry and fisheries: Review and prognosis. *Rev Market Agr Econ*, 49 (03), 1981.

A. Levitin. *Introduction to the Design and Analysis of Algorithms*. Pearson, third edition, 2011.

A. Lew and H. Mauch. *Dynamic Programming: A Computational Tool*, volume 38. Springer, 2006.

W. Liu and B. Schmidt. A generic parallel pattern-based system for bioinformatics. In *Proc. Euro-Par*, pages 989–996. Springer, 2004.

Y. Pu, R. Bodik, and S. Srivastava. Synthesis of first-order dynamic programming algorithms. *ACM SIGPLAN Notices*, 46(10):83–98, 2011.

R. Reitzig. Automated parallelisation of dynamic programming recursions. *Masters Thesis: University of Kaiserslautern*, 2012.

A. A. Robichek, E. J. Elton, and M. J. Gruber. Dynamic programming applications in finance. *JF*, 26(2):473–506, 1971.

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.

J. Rust. Numerical dynamic programming in economics. *Handbook of Computational Economics*, 1:619–729, 1996.

H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Trans Acoust Speech*, 26(1):43–49, 1978.

D. K. Smith. Dynamic programming and board games: A survey. *EJOR*, 176(3):1299–1318, 2007.

M. Sniedovich. *Dynamic Programming: Foundations and Principles*. CRC press, 2010.

S. Tang, C. Yu, J. Sun, B.-S. Lee, T. Zhang, Z. Xu, and H. Wu. EasyPDP: An efficient parallel dynamic programming runtime system for computational biology. *TPDS*, 23(5):862–872, 2012.

Y. Tang, R. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The Pochoir stencil compiler. In *Proc. SPAA*, pages 117–128, 2011a.

Y. Tang, R. Chowdhury, C.-K. Luk, and C. E. Leiserson. Coding stencil computations using the Pochoir stencil-specification language. In *Proc. HotPar*, 2011b.

J. Tithi, P. Ganapathi, A. Talati, S. Agarwal, and R. Chowdhury. High-performance energy-efficient recursive dynamic programming with matrix-multiplication-like flexible kernels. In *Proc. IPDPS*, 2015.

J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkens-Diehr. Xsede: Accelerating scientific discovery. *Computing in Science and Engineering*, 16(5):62–74, 2014. ISSN 1521-9615. doi: http://doi.ieeecomputersociety.org/10.1109/MCSE.2014.80.

J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proc. ICPPW*, pages 207–216, 2010.

J. D. Ullman, A. V. Aho, and J. E. Hopcroft. The design and analysis of computer algorithms. *Addison-Wesley, Reading*, 4:1–2, 1974.

M. S. Waterman. *Introduction to Computational Biology: Maps, Sequences and Genomes*. Chapman & Hall Ltd., 1995.