Autogen: Automatic Discovery of Efficient Recursive Divide-&-Conquer Algorithms for Solving Dynamic Programming Problems

REZAUL CHOWDHURY, PRAMOD GANAPATHI, and STEPHEN TSCHUDI,
Stony Brook University
JESMIN JAHAN TITHI, Intel Corporation
CHARLES BACHMEIER, CHARLES E. LEISERSON, and ARMANDO SOLAR-LEZAMA,
MIT
BRADLEY C. KUSZMAUL, Oracle
YUAN TANG, Fudan University

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.

CCS Concepts: • **Theory of computation → Dynamic programming**; **Divide and conquer**; **Shared memory algorithms**;

Additional Key Words and Phrases: Autogen, automatic discovery, dynamic programming, recursive, divide-and-conquer, cache-efficient, parallel, cache-oblivious, energy-efficient

Chowdhury and Ganapathi were supported in part by NSF grants CCF-1162196, CCF-1439084, and CNS-1553510. Tschudi was supported by an NSF REU supplement for CNS-1553510. 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) [2, 52], which is supported by NSF grant ACI-1053575.

Authors' addresses: R. Chowdhury, Computer Science Department, Stony Brook University, Stony Brook, NY 11794, USA; email: rezaul@cs.stonybrook.edu; P. Ganapathi, Bengaluru, Karnataka, India; email: pramod@learningisbeautiful.in; S. Tschudi, Google Inc., S. Tschudi (stschudi), 1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA; email: stschudi42@gmail.com; J. J. Tithi, Intel Corporation, SC12, 3600 Juliette Ln, Santa Clara, CA 95054, USA; email: jesmin.jahan.tithi@intel.com; C. Bachmeier, C. E. Leiserson, and A. Solar-Lezama, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA 02139, USA; emails: {cabach, cel, asolar}@csail.mit.edu; B. C. Kuszmaul, 37 Vaille Ave Lexington, MA 02421, USA; email: kuszmaul@gmail.com; Y. Tang, chool of Software, Shanghai Key Laboratory of Intelligent Information Processing, Fudan University, Shanghai, China; email: yuantang@fudan.edu.cn.

https://doi.org/10.1145/3125632

ACM Reference format:
Rezaul Chowdhury, Pramod Ganapathi, Stephen Tschudi, Jesmin Jahan Tithi, Charles Bachmeier, Charles E. Leiserson, Armando Solar-Lezama, Bradley C. Kuszmaul, and Yuan Tang. 2017. Autogen: Automatic Discovery of Efficient Recursive Divide-&-Conquer Algorithms for Solving Dynamic Programming Problems. *ACM Trans. Parallel Comput.* 4, 1, Article 4 (October 2017), 30 pages.
https://doi.org/10.1145/3125632

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 [5, 17, 46] 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 [4, 20, 27, 56] and in many other application areas including operations research [28], compilers [36], sports [19, 41], games [45], economics [42], finance [40], and agriculture [33].

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, thus imposing a 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, not on cache sizes, and thus the algorithms remain *cache oblivious*[3] [23] and more *portable* than cache-aware tiled iterative codes. Unlike tiled looping codes, these algorithms are also *cache adaptive* [6]—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 [24] in which we are given a sequence of characters $S = s_1 \cdots s_n$ and we are required to compute the minimum cost of parenthesizing S. Let $G[i,j]$ denote the minimum cost of parenthesizing $s_i \cdots s_j$. Then the

[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.

Table 1. Work (T_1), Serial Cache Complexity (Q_1), Span (T_∞), and Parallelism (T_1/T_∞) of I-DP and R-DP Algorithms for Several DP Problems

		I-DP			R-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 [14]	$\Theta(n^3)$	$\Theta(n^3)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^3/(B\sqrt{M}))$	$\Theta(n^{\log_2 3})$	$\Theta(n^{3-\log_2 3})$
Floyd-Warshall's APSP 3D [15]	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n \log n)$	$\Theta(n^2/\log n)$	$\Theta(n^3/B)$	$O(n \log^2 n)$	$\Theta(n^2/\log^2 n)$
Floyd-Warshall's APSP 2D [15]	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n \log n)$	$\Theta(n^2/\log n)$	$\Theta(n^3/(B\sqrt{M}))$	$\Theta(n \log^2 n)$	$\Theta(n^2/\log^2 n)$
LCS / Edit distance [13]	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2/(BM))$	$\Theta(n^{\log_2 3})$	$\Theta(n^{2-\log_2 3})$
Multi-instance Viterbi [10]	$\Theta(n^3 t)$	$\Theta(n^3 t/B)$	$\Theta(nt)$	$\Theta(n^2)$	$\Theta(n^3 t/(B\sqrt{M}))$	$\Theta(nt)$	$\Theta(n^2)$
Gap problem [8]	$\Theta(n^3)$	$\Theta(n^3)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^3/(B\sqrt{M}))$	$\Theta(n^{\log_2 3})$	$\Theta(n^{3-\log_2 3})$
Protein accordion folding [51]	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^3/(B\sqrt{M}))$	$\Theta(n \log n)$	$\Theta(n^2/\log n)$
Spoken-word recognition [43]	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2/(BM))$	$\Theta(n^{\log_2 3})$	$\Theta(n^{2-\log_2 3})$
Function approximation	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^3/(B\sqrt{M}))$	$\Theta(n^{\log_2 3})$	$\Theta(n^{3-\log_2 3})$
Binomial coefficient [35]	$O(n^2)$	$O(n^2/B)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2/(BM))$	$\Theta(n^{\log_2 3})$	$\Theta(n^{2-\log_2 3})$
Bitonic traveling salesman [17]	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n^2/(BM))$	$\Theta(n \log n)$	$\Theta(n/\log n)$
Matrix multiplication [23]	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^3/(B\sqrt{M}))$	$\Theta(n)$	$\Theta(n^2)$
Bubble sort [9]	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n^2)$	$\Theta(1)$	$\Theta(n^2/(BM))$	$\Theta(n)$	$\Theta(n)$
Selection sort [9]	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n^2)$	$\Theta(1)$	$\Theta(n^2/(BM))$	$\Theta(n)$	$\Theta(n)$
Insertion sort [9]	$O(n^2)$	$O(n^2/B)$	$O(n^2)$	$\Theta(1)$	$O(n^{\log_2 3}/(BM^{\log_2 3-1}))$	$O(n)$	$\Omega(n)$

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 $M = \Omega(B^d)$ when $\Theta(n^d)$ is the size of the DP table. On p cores, the running time is $T_p = O(T_1/p + T_\infty)$ and the parallel cache complexity is $Q_p = O(Q_1 + p(M/B)T_\infty)$ 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 R-DP is $O(n^{\log_2 3})$.

DP table $G[0:n, 0:n]$ is filled up using the following recurrence:

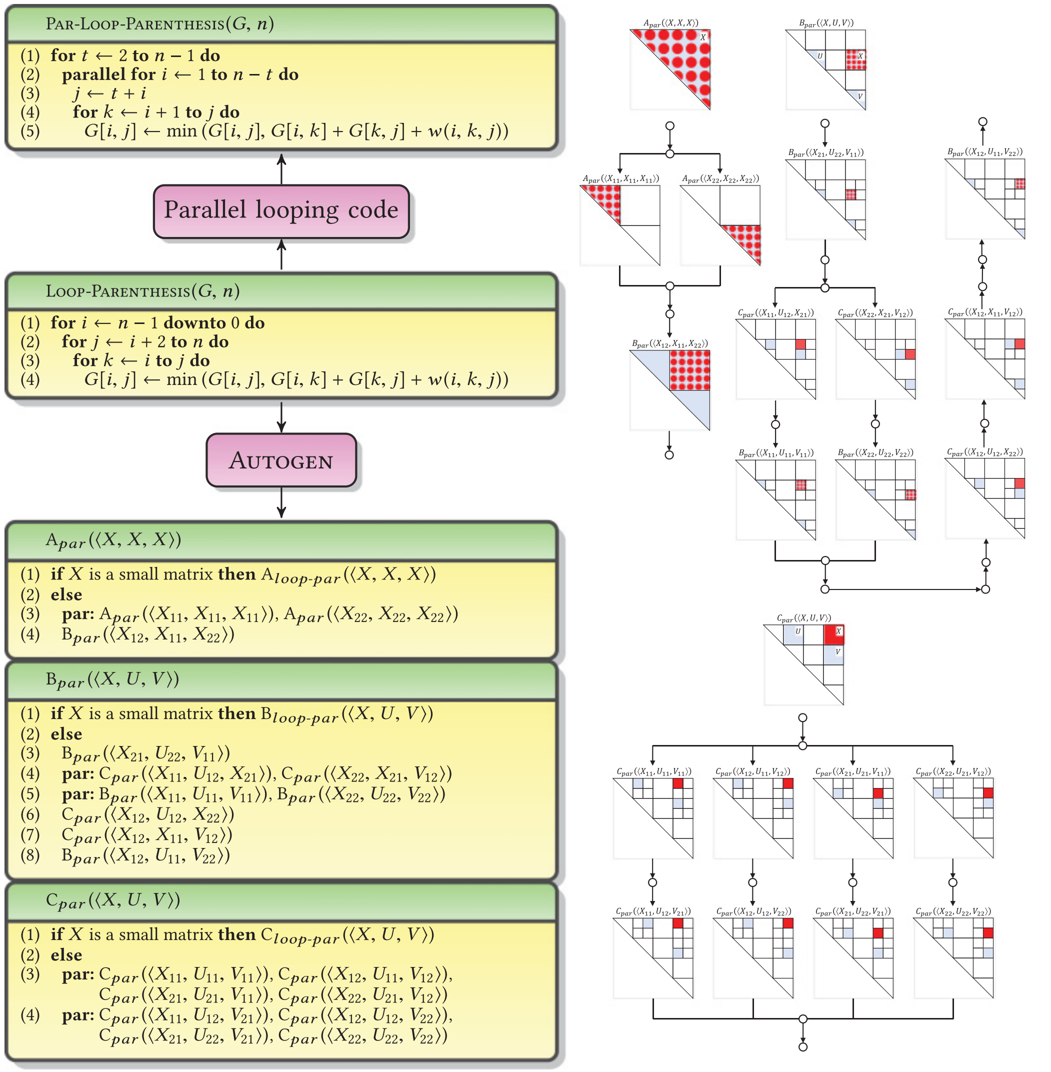
$$G[i,j] = \begin{cases} \infty & \text{if } 0 \le i = j \le \ n, \\ v_j & \text{if } 0 \le i = j-1 < n, \\ \min\limits_{i \le k \le j} \{(G[i,k] + G[k,j]) + w(i,k,j)\} & \text{if } 0 \le i < j-1 < n, \end{cases} \quad (1)$$

where the v_js 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 G. Assuming that G is of size $n \times n$ and G 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 $\Theta(n^3)$ cache misses in the ideal-cache model [23]. 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* ([17], 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 $\Theta(n^3)/\Theta(n^2) = \Theta(n)$. If the size M of the cache is known, the code can be tiled to improve its cache performance to $\Theta(n^3/B\sqrt{M}))$, 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.

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



```
Par-Loop-Parenthesis($G$, $n$)
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
      $G[i, j] \leftarrow \min(G[i, j], G[i, k] + G[k, j] + w(i, k, j))$
```

```
Loop-Parenthesis($G$, $n$)
for $i \leftarrow n - 1$ downto 0 do
  for $j \leftarrow i + 2$ to $n$ do
    for $k \leftarrow i$ to $j$ do
      $G[i, j] \leftarrow \min(G[i, j], G[i, k] + G[k, j] + w(i, k, j))$
```

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

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

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

Fig. 1. Left upper half: A parallel looping code that evaluates Rec. 1. Left lower half: 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 A$_{par}$($\langle G, G, G\rangle$), where G is an $n \times n$ DP table and n is a power of 2. The iterative base-case kernel of a function F$_{par}$ is F$_{loop\text{-}par}$. Right: Pictorial representation of the recursive divide-and-conquer algorithm discovered by Autogen. While cells in a dark red block are updated using data only from light blue blocks (C$_{par}$), cells in a red dotted block are updated using data from that block itself (A$_{par}$) as well as from light blue blocks (B$_{par}$).

Finally, Figure 1 shows the type of algorithm 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 $\Theta(n^3/(B\sqrt{M}))$ cache misses, which can be shown to be optimal. Second, its parallelism is $\Theta(n^{3-\log_2 3}) = \omega(n^{1.41})$, which is asymptotically greater than the $\Theta(n)$ parallelism achieved by the parallel looping code. Third, since the algorithm uses recursive blocking, it can

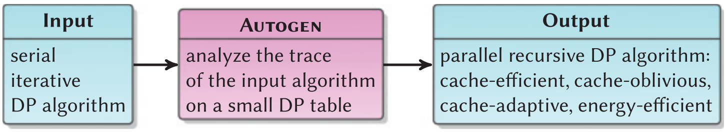


Fig. 2. Input and output of Autogen.

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 $C_{loop\text{-}par}$ is highly optimizable like a matrix multiplication algorithm, and the total time spent inside $C_{loop\text{-}par}$ asymptotically dominates the time spent inside $A_{loop\text{-}par}$ and $B_{loop\text{-}par}$ [51]. Hence, reasonably high performance can be achieved simply by optimizing $C_{loop\text{-}par}$.

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 eight-core 2.7GHz Intel Sandy Bridge processors ($2 \times 8 = 16$ cores in total), per-core 32KB private L1 cache and 256KB private L2 cache, and per-socket 20MB shared L3 cache, and 32GB 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 $8,000 \times 8,000$, the recursive algorithm without any nontrivial hand-optimizations ran more than 15 times faster than the nontiled 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 on the same socket each using only two cores, the nontiled looping code slowed down by almost a factor of 2 compared to a single instance running on two 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 nontiled looping code suffered because of bandwidth saturation, the tiled looping code suffered because of its inability to adapt to cache sharing.

In this article, we present Autogen—an algorithm that for any problem from a very wide class of DP problems can *automatically discover* an efficient cache-oblivious parallel recursive divide-and-conquer algorithm from a naïve serial iterative implementation (or any black-box implementation) of the DP recurrence (see Figure 2). 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 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. Autogen effectively eliminates the need for human involvement in the design of efficient cache-oblivious parallel algorithms for all known and yet-to-be-identified problems in that class.

Our contributions. Our major contributions are as follows:

(1) **[Algorithmic.]** We present 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. 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 Autogen.

(2) **[Experimental.]** We have implemented a prototype of Autogen that 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.

A preliminary version of this work appeared in the proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2016) [12]. In the current version, we have added three new sections: Section 5 on a more space- and time-efficient version of the Autogen algorithm, which we call the *deductive* Autogen; Section 6.3 on extending Autogen to handle nonorthogonal regions; and Section 7.4 on experimental results comparing inductive and deductive Autogen. We have also added a section named "Conclusions" (Section 8).

2 RELATED WORK

Systems for autogenerating fast iterative DP implementations exist. The Bellman's GAP compiler [26] converts declarative programs into optimized C++ code. A semiautomatic synthesizer [38] exists that uses constraint 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 [47] 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. DPX10 [55] is also a DAG and pattern-based system like EasyPDP, but it targets distributed-memory machines and dynamic programs for filling out two-dimensional DP tables with $\mathcal{O}(1)$ dependency per cell (e.g., the Smith-Waterman algorithm for sequence alignment [56]). EasyHPS [18] uses the master-slave paradigm in which the master scheduler distributes computable subtasks among its slaves, which in turn distribute subsubtasks among slave threads. A pattern-based system exists [37] that uses generic programming techniques such as class templates to solve problems in bioinformatics. Parallelizing plugins [39] 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 Autogen but completely different in many aspects is Pochoir [48, 49]. While Pochoir tailors the implementation of the same cache-oblivious algorithm to different stencil computations, Autogen discovers a (possibly) brand new efficient parallel cache-oblivious algorithm for every new DP problem it encounters.

Recently, a subset of the authors of the current article developed another system called Bellmania [30] that uses solver-aided tactics to derive parallel recursive divide-and-conquer implementations of DP algorithms. Bellmania includes a high-level language for specifying DP algorithms and a calculus that facilitates gradual transformation of these specifications into efficient recursive implementations. A user interactively guides the step-by-step transformation process while an SMT-based back-end verifies each step.

The algorithms generated by Autogen are based on two-way recursive divide-and-conquer, which have optimal serial cache complexity but often have lower parallelism compared with iterative wavefront algorithms due to artificial dependencies among subtasks [50]. Recently we showed how to systematically transform these two-way recursive algorithms into recursive wavefront algorithms that use closed-form formulas to compute at what time each recursive function must be launched in order to achieve high parallelism without losing cache performance [11, 25].

Compiler technology for automatically converting iterative versions of matrix programs to serial recursive versions is known [3]. The approach relies on heavy machineries such as dependence analysis (based on integer programming) and polyhedral techniques. 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 generates a provably correct parallel recursive algorithm for solving the same DP.

3 THE AUTOGEN ALGORITHM

In this section, we describe the Autogen algorithm.

*Definition 3.1 (**I − DP/R − DP/Autogen**).* Let P be a given DP problem. An I-DP is an iterative (i.e., loop-based) algorithm for solving P. An R-DP is a cache-oblivious parallel recursive divide-and-conquer algorithm (if it exists) for P. Autogen is our algorithm for autogenerating an R-DP from a given I-DP for P.

I-DP
Autogen
R-DP

We make the following assumption about an R-DP.

Assumption 1 (Number of Functions). *The number of distinct recursive functions in an R-DP is upper bounded by a constant (e.g., the R-DP in Figure 1 has three distinct recursive functions).*

This assumption implies that the number of distinct recursive functions in an R-DP is independent of the size of any DP problem on which the R-DP will be run. As a result, the R-DP generated by our Autogen algorithm will not be tied to one specific problem size. We are not aware of any R-DP for which Assumption 1 is not true.

Algorithm. The four main steps of 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 I-DP on a DP table of small size. See Section 3.1.
(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 Section 3.2.
(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 R-DP. See Section 3.3.
(4) [**Algorithm-DAG construction.**] For every unique function of the R-DP, we construct a DAG that shows both the order in which the child functions are to be executed and the parallelism involved. See Section 3.4.

Example. Autogen works for arbitrary d-dimensional ($d \geq 1$) 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 Autogen by applying it on an I-DP for the parenthesis problem, which updates a two-dimensional DP table. The solution is described by Recurrence 1, which is evaluated by the serial I-DP. In the rest of the section, we show how Autogen discovers the R-DP shown in Figure 1 from this serial I-DP.

3.1 Cell-Set Generation

A *cell* is a spatial grid point in a DP table identified by its d-dimensional coordinates. A d-dimensional DP table G 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 I-DP performs the following update:

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

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

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

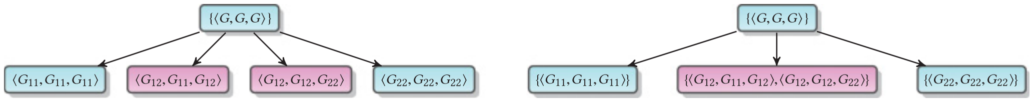
3.2 Algorithm-Tree Construction

Given an I-DP, a tree representing a hierarchy of recursive divide-and-conquer functions that is used to find a potential R-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 G and reads from tables $G^1, \ldots, G^s$ (one or more of them can be the same as G). The algorithm tree is constructed as follows.

At level 0, the only regions possible are the entire tables $G, G^1, \ldots, G^s$. 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 $G^1, \ldots, G^s$, the only possible region tuple is $\langle G, G^1, \ldots, G^s \rangle$. 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 the parenthesis problem, as all the write and read cells belong to the same DP table C, the root node will be $\{\langle G, G, G \rangle\}$.

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 G into four quadrants: G_{11} (top-left), G_{12} (top-right), G_{21} (bottom-left), and G_{22} (bottom-right). Similarly, each G^i for $i \in [1, s]$ is divided into four quadrants: $G^i_{11}, G^i_{12}, G^i_{21}$, and G^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 $\langle g, g_1, \ldots, g_s \rangle$ is found to have $g \in G_k$ and $g_i \in G^i_{k_i}$ for $i \in [1, s]$ and $k, k_i \in \{11, 12, 21, 22\}$, then we say that $\langle g, g_1, \ldots, g_s \rangle$ belongs to region tuple $\langle G_k, G^1_{k_1}, \ldots, G^s_{k_s} \rangle$. 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 $\langle G_{11}, G_{11}, G_{11} \rangle$, $\langle G_{22}, G_{22}, G_{22} \rangle$, $\langle G_{12}, G_{11}, G_{12} \rangle$, and $\langle G_{12}, G_{12}, G_{22} \rangle$.

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.


$\{\langle G, G, G\rangle\}$
$\langle G_{11}, G_{11}, G_{11}\rangle$
$\langle G_{12}, G_{11}, G_{12}\rangle$
$\langle G_{12}, G_{12}, G_{22}\rangle$
$\langle G_{22}, G_{22}, G_{22}\rangle$
$\{\langle G, G, G\rangle\}$
$\{\langle G_{11}, G_{11}, G_{11}\rangle\}$
$\{\langle G_{12}, G_{11}, G_{12}\rangle, \langle G_{12}, G_{12}, G_{22}\rangle\}$
$\{\langle G_{22}, G_{22}, G_{22}\rangle\}$

(a) Before applying Rule 1.

(b) After applying Rule 1.

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

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 $\langle G_{12}, G_{11}, G_{12}\rangle$ and $\langle G_{12}, G_{12}, G_{22}\rangle$ are combined into a single node $\{\langle G_{12}, G_{11}, G_{12}\rangle, \langle G_{12}, G_{12}, G_{22}\rangle\}$. The other two nodes are $\{\langle G_{11}, G_{11}, G_{11}\rangle\}$ and $\{\langle G_{22}, G_{22}, G_{22}\rangle\}$. 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.

3.3 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 next.

Output Fingerprints. The output fingerprints of nodes in an algorithm tree help in identifying if the region tuples belonging to two different nodes in the tree are decomposed and distributed among their respective child nodes in the same way. Since a node will correspond to an R-DP function with its region tuples as inputs and its child nodes will correspond to recursive function calls, the output fingerprints indicate if two functions make recursive function calls that look similar based on what inputs they receive.

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 and r_i are octant ids (e.g., $w, r_i \in \{11, 12, 21, 22\}$ for two-dimensional matrices, and $w, r_i \in \{111, 112, 121, 122, 211, 212, 221, 222\}$ for three-dimensional matrices) such that $\langle W_w, R_{r_1}, \ldots, R_{r_s}\rangle$ is a region tuple, where $i \in [1, s]$. For example, suppose a node u has only one region tuple $\langle X, Y, Z, Y\rangle$ and two child nodes u_1 and u_2 with region tuples $\langle X_{11}, Y_{11}, Z_{22}, Y_{12}\rangle$ and $\langle X_{22}, Y_{21}, Z_{11}, Y_{22}\rangle$, respectively. Then u's output fingerprint will be $\{\langle 11, 11, 22, 12\rangle, \langle 22, 21, 11, 22\rangle\}$.

Input Fingerprints. The input fingerprints are used to determine if the region tuples associated with any two given algorithm-tree nodes look structurally similar based on how many times each (sub)matrix appears in each region tuple and at what locations. Thus, they check the structural similarity (order and repetitions) between the inputs received by the R-DP functions corresponding to those two nodes.

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, consider two nodes u_1 and u_2 with region tuples $\langle X, Y, Z, Z, Y\rangle$ and $\langle Z, X, U, U, X\rangle$, respectively. Both of them will have the same input fingerprint $\langle 1, 2, 3, 3, 2\rangle$.

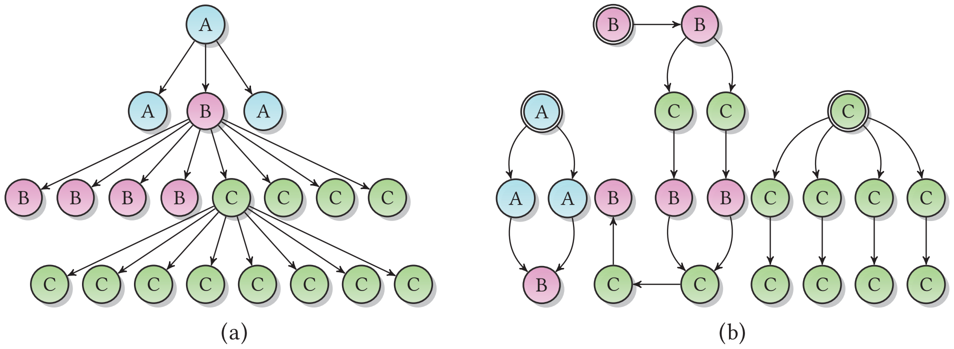


Fig. 4. (a) A small part of the labeled algorithm tree for the parenthesis problem that only shows three expanded nodes. (b) Algorithm DAGs for the three functions A, B, and C in the parenthesis problem showing the order of execution of functions.

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

Threshold Level. 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 label the nodes of the tree with function names level by level until we reach level l and at that point we stop. We call l the *threshold level* and it has a constant upper bound as per Assumption 1.

It is easy to see that once we reach a level l in which no new functions are found, no new functions will be found in higher levels. Since we already encountered each of the functions found in level l in one of the lower levels (levels 0 to $l - 1$), the functions it recursively calls were also encountered in one of those levels. Now level $l + 1$ cannot contain any new functions simply because it will only contain functions recursively called by functions in level l. The same argument also applies to all levels higher than $l + 1$.

The labeled algorithm tree for the parenthesis problem is given in Figure 4(a).

3.4 Algorithm-DAG Construction

In this step, we construct a DAG for every function. An algorithm tree does not give information on (1) the sequence in which a function calls other functions and (2) 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 F, we define $\mathbf{W}$(F) and $\mathbf{R}$(F) as the write region and the set of read regions of the region tuples in 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 $\mathbf{W}(T) \notin \mathbf{R}(T)$; that is, 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 F calls two functions F_1 and F_2, then the *function ordering* between F_1 and F_2 will be one of the following: (1) $\mathrm{F}_1 \rightarrow \mathrm{F}_2$—that is, F_1 is called before F_2; (2) $\mathrm{F}_2 \rightarrow \mathrm{F}_1$—that is, F_2 is called before F_1; (2) $\mathrm{F}_1 \leftrightarrow \mathrm{F}_2$—that is, either $\mathrm{F}_1 \rightarrow \mathrm{F}_2$ or $\mathrm{F}_2 \rightarrow \mathrm{F}_1$; and (4) $\mathrm{F}_1 || \mathrm{F}_2$—that is, F_1 can be run in parallel with F_2.

If a function F calls two functions F_1 and F_2, then the order in which F_1 and F_2 are executed is determined by the following rules.

Rule 2. *If $\boldsymbol{W}(F_1) \neq \boldsymbol{W}(F_2)$ and $\boldsymbol{W}(F_1) \in \boldsymbol{R}(F_2)$, then $F_1 \rightarrow F_2$.*

Rule 3. *If $\boldsymbol{W}(F_1) = \boldsymbol{W}(F_2)$and F_1 is flexible but F_2 is not, then $F_1 \rightarrow F_2$.*

Rule 4. *If $\boldsymbol{W}(F_1) = \boldsymbol{W}(F_2)$ and both F_1 and F_2 are flexible, then $F_1 \leftrightarrow F_2$.*

Rule 5. *If F_1 and F_2 satisfy none of the rules 2, 3, and 4, then $F_1||F_2$.*

Rule 2 states that if F_2 reads from a region F_1 writes to, then F_1 must execute before F_2. This rule ensures that these two functions do not violate the one-way sweep property (Property 1 in Section 4), which says that unless a DP table cell is fully updated (i.e., will not be modified later), it cannot be used to update another cell. We will formally define this property in Section 4 when we prove the correctness of the Autogen algorithm.

Rule 3, too, is needed to prevent the violation of the one-way sweep property. It says that if F_2 reads from and writes to a region that is also written to but not read from by F_1, then F_1 must execute before F_2.

Rule 4 says that if F_1 and F_2 writes to the same region but neither of them reads from that region, then it does not matter in which order those two functions execute. The associativity and commutativity properties of the update operator $\oplus$ ensure that both $\mathrm{F}_1 \rightarrow \mathrm{F}_2$ and $\mathrm{F}_2 \rightarrow \mathrm{F}_1$ produce the same result. We do not execute these two functions in parallel in order to avoid race conditions.

Rule 5 implies that if F_1 and F_2 write to different regions and neither of them reads from a region that the other one or itself is writing to, then those two functions can safely execute in parallel.

Based on these rules, we construct a DAG-like structure (possibly with some cycles) for every function. We then modify each such structure by deleting redundant edges from it as per the following rules and we end up with a DAG.

The following rule removes cycles created by Rule 4 by explicitly fixing an order of execution between F_1 and F_2, that is, by choosing either $\mathrm{F}_1 \rightarrow \mathrm{F}_2$ or $\mathrm{F}_2 \rightarrow \mathrm{F}_1$.

Rule 6. *In the algorithm tree, if a function F calls two functions F_1 and F_2, then in the DAG of F, we create nodes d_1 and d_2 corresponding to F_1 and F_2 (if they are not already present), respectively. If $F_1 \rightarrow F_2$, then we add a directed edge from d_1 to d_2. On the other hand, if $F_1 \leftrightarrow F_2$, then we add a directed edge either from d_1 to d_2 or from d_2 to d_1, but not both.*

The following rule removes redundant shortcut edges.

Rule 7. *Let d_1, d_2, and d_3 be three nodes in a DAG. If there are directed edges from d_1 to d_2, from d_2 to d_3, and from d_1 to d_3, then mark the directed edge from d_1 to d_3 provided neither of the two other edges (i.e., from d_1 to d_2 and from d_2 to d_3) are already marked. Mark all such edges in the DAG until no more edges can be marked. Finally, delete all marked edges from the DAG.*

The set of all modified DAGs for all functions represents an R-DP for the given I-DP. The algorithm DAGs for the parenthesis problem is given in Figure 4(b).

Threshold Problem Size and Reruns

Suppose the R-DP algorithm for the problem includes at least m distinct functions. From Section 3.3, the threshold level has an upper bound of m. In order to make sure that the cell set used by Autogen captures the dependency patterns of the DP accurately, we should use a DP table with each dimension of size at least $n = 2^{m+k}$ to generate the cell set, where k is a problem-specific natural number.

Empirically, we have found that the number of functions required to represent an R-DP algorithm for most problems is at most four. Considering $m = 4$ and $k = 2$, we get $n = 64$. If we are unable to generate all the functions, we increase the value of k and build the algorithm tree again. We continue this process until we generate functions that call no new functions. Such a threshold value of n is called the *threshold problem size* for the given DP problem.

Space/Time Complexity of Autogen

We analyze the space and time complexities of Autogen using parameters d, s, and l, where d is the number of dimensions, $1 + s$ is the cell-tuple size, and l is the threshold level.

Let the size of the small DP table Autogen uses to generate the cell set be n^d. Let Δ be an upper bound on the maximum number of cells a cell depends on. Then Autogen generates $\mathcal{O}(n^d \Delta)$ cell tuples. Hence, the total space complexity of Autogen is $\mathcal{O}(n^d \Delta ds)$. To construct an algorithm tree, the cell-tuples are scanned $\mathcal{O}(l)$ times. Other parts of the algorithm take asymptotically less time than the time taken for algorithm-tree construction. So, the total time complexity of Autogen is $\mathcal{O}(n^d \Delta dsl)$.

4 CORRECTNESS & CACHE COMPLEXITY

In this section, we give a proof of correctness for Autogen and analyze the cache complexity of the autodiscovered R-DPs.

4.1 Correctness of Autogen

We prove that if an I-DP satisfies the following two properties, then Autogen can be applied on the I-DP to get a correct R-DP under Assumption 1.

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

Property 1 says that at most one state (i.e., the final state) of a DP table cell can be used to update other cells. Indeed, if a computation (e.g., most DP problems) can be described by a recurrence, then it already satisfies this property (see Section 6.1 for a DP problem that violates Property 1).

Property 2 (Fractal Property). *An I-DP satisfies the fractal property if the following holds. Let S_n and S_{2n} be the cell sets of the I-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). Let S'_n be the cell set generated from S_{2n} by replacing every coordinate value j with $\lfloor j/2 \rfloor$ and then retaining only the distinct tuples. Then, $S_n = S'_n$.*

Property 2 basically says that the pattern of updates performed by an I-DP is independent of the problem size. This property ensures that the R-DP generated by Autogen for a given DP will not depend on the size of the DP table used to generate the cell set in Section 3.1. Of course, the table must not be too small as otherwise the cell set may not be able to capture the dependence patterns accurately (see the paragraphs on “Threshold problem size and reruns” at the end of Section 3.4).

We are now in a position to define the class of DP problems targeted by Autogen.

*Definition 4.1 (*Fractal - DP *Class).* An I-DP is said to be in the Fractal-DP class if the following conditions hold: (1) it satisfies the one-way sweep property (Property 1), (2) it satisfies the fractal property (Property 2), and (3) the cell-tuple size $(1 + s)$ has a constant upper bound.

We prove next that if the input DP belongs to the Fractal-DP class, then Autogen generates a correct R-DP for that DP.

Theorem 4.2 (Correctness). *Given an I-DP from the Fractal-DPclass as input, Autogen generates an R-DP that is functionally equivalent to the given I-DP.*

Proof. Let the I-DP and R-DP algorithms for a problem P be denoted by I and R, respectively. We use mathematical induction to prove the correctness of Autogen in d dimensions, assuming d to be a constant. We first prove the correctness for the threshold problem size (see Section 3.3)

Table 2. T_1 and T_2 Are Two Cell-Tuples In both subtables, columns 2 to 5 represent the rour conditions for the two cell tuples. Columns I and R show the ordering of the cell tuples for I and R algorithms, respectively. The order of cell updates in R is consistent with that in Table 1

Case	$\boldsymbol{W}(T_1) \in \boldsymbol{R}(T_1)$	$\boldsymbol{W}(T_2) \in \boldsymbol{R}(T_2)$	$\boldsymbol{W}(T_1) \in \boldsymbol{R}(T_2)$	$\boldsymbol{W}(T_1) = \boldsymbol{W}(T_2)$	I	Rule	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	$T_1 \parallel T_2$

(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 Autogen is correct for $n = 2^q$, we have to show the following: (1) number of nodes in the algorithm tree is $\mathcal{O}(1)$, (2) both I and R apply the same set of cell updates, and (3) R never violates the one-way sweep property (Property 1).

(1) *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 ($\leq 2^d((2^d - 1)^s + 1)$) and the maximum number of children a node can have ($\leq 2^{2^d((2^d-1)^s+1)}$). 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 $\mathcal{O}(1)$.

(2) *Both I and R perform the same set of cell updates:* There is no cell tuple of I that is not considered by R. In Section 3.2, 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 I are considered by R.

There is no cell tuple of R that is not considered by I. Let there be a cell tuple T in R that is not present in I. As the cell tuples in 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 I should have generated the cell tuple T, which contradicts our initial assumption. Hence, by contradiction, all the cell tuples of R are considered by I.

(*c*) *R never violates the one-way sweep property* (*Property 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 R is exactly the same as that in 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 Table 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 Table 2). The relation between T_1 and T_2 can be one of the following five: (1) $T_1 = T_2$, (2) $T_1 \to T_2$ (i.e., T_1 is executed before T_2), (3) $T_2 \to T_1$ (i.e., T_2 is executed before T_1), (4) $T_1 || T_2$ (i.e., T_1 and T_2 can be executed in parallel), and (5) $T_1 \leftrightarrow T_2$ (i.e., either $T_1 \to T_2$ or $T_2 \to T_1$, both of which will produce the same results because of the associativity and commutativity properties of the update operator $\oplus$).

Columns I and R represent the ordering of the two cell tuples in I and R algorithms, respectively. Column I is filled based on the one-way sweep property (Property 1) and column 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 R and I apply the updates in exactly the same order. Hence, R satisfies the one-way sweep property.

Induction. We show that if 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 R for $r = q$. Algorithm R is a set of DAGs for different functions. Let G^n and G^{2n} represent two DP tables of size n^d and $(2n)^d$, respectively, such that $n \geq 2^q$. According to Property 2, the dependencies among the regions G^n_{11}, G^n_{12}, G^n_{21}, G^n_{22} must be exactly the same as the dependencies among the regions G^{2n}_{11}, G^{2n}_{12}, G^{2n}_{21}, G^{2n}_{22}. If they were different, then that would violate Property 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 G^n and G^{2n} are the same. Therefore, if Autogen is correct for $n = 2^r$ for some $r \geq q \in \mathbb{N}$, it is also correct for $n = 2^{r+1}$.

4.2 Cache Complexity of an R-DP

A recursive function is *closed* provided it does not call any other recursive function but itself, and it is *semiclosed* provided it only calls itself and other closed functions. A closed (semiclosed, respectively) function H is *dominating* provided no other closed (semiclosed, respectively) function of the given R-DP makes more self-recursive calls than made by H and every nonclosed (nonsemiclosed, respectively) function makes strictly fewer such calls.

Theorem 4.3 (Cache Complexity). *If an R-DP includes a dominating closed or semiclosed function F_k that calls itself recursively a_{kk} times, then the serial cache complexity of the R-DP for a DP table of size n^d is $Q_1(n, d, B, M) = O(T_1(n)/(BM^{(l_k/d)-1}) + S(n, d)/B + 1)$ under the ideal-cache model, where $l_k = \log_2 a_{kk}$, $T_1(n) =$ total work $= O(n^{l_k})$, $M =$ cache size, $B =$ block size, $M = \Omega(B^d)$, and $S(n, d) =$ space complexity $= O(n^d)$.*

Proof. Suppose the R-DP algorithm consists of a set F of m recursive functions $F_1, F_2, \ldots, F_m$. For $1 \leq i, j \leq m$, let a_{ij} be the number of times F_i calls F_j. Then, for a suitable constant $\gamma_i > 0$, the cache complexity Q_{F_i} of F_i on an input of size n^d can be computed recursively as follows:

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

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

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

If F_k is a dominating closed (semiclosed, respectively) function, then (*i*) $a_{kk} \geq a_{ii}$ for every closed (semiclosed, respectively) function F_i, and (*ii*) $a_{kk} > a_{jj}$ for every nonclosed (nonsemiclosed, respectively) function F_j. The algorithm tree must contain at least one path $P = \langle \text{F}_{r_1}, \text{F}_{r_2}, \ldots, \text{F}_{r_{|P|}} \rangle$ from its root ($= \text{F}_{r_1}$) to a node corresponding to F_k ($= \text{F}_{r_{|P|}}$). 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 $\mathcal{O}(Q_{\text{F}_k}(n))$. This result is obtained by moving upward in the tree starting from $\text{F}_{r_{|P|-1}}$, 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 $Q_{\text{F}_{r_1}}(n)$ of the R-DP algorithm is $\mathcal{O}(Q_{\text{F}_k}(n))$. This completes the proof.

It is important to note that the serial cache complexity and the total work of an R-DP algorithm are related. Let F_k be a dominating closed function that calls itself a_{kk} number of times and let $P = \langle \text{F}_{r_1}, \text{F}_{r_2}, \ldots, \text{F}_{r_{|P|}} \rangle$ be a path in the algorithm tree from its root ($= \text{F}_{r_1}$) to a node corresponding to F_k ($= \text{F}_{r_{|P|}}$). 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 the master theorem repeatedly, we can show that $T_1(n) = \mathcal{O}(n^{\log a_{kk}} \log^{q-1} n)$.

5 THE DEDUCTIVE AUTOGEN ALGORITHM

In Section 3, we developed the Autogen algorithm using an inductive approach. The major limitation of the inductive Autogen algorithm is that if the number of recursive functions in an R-DP is very high, say, 100, the length n of each dimension of the DP table used for generating the cell set can be larger than 2^{100}. Thus, both the space and time complexities of the algorithm can be prohibitively high and hence impractical.

In this section, we present a deductive method to construct the Autogen algorithm. It addresses the limitation of the inductive Autogen. We call this algorithm the deductive Autogen algorithm [25]. Deductive Autogen differs from inductive Autogen in the first two steps only.

Algorithm. The four main steps of deductive Autogen are:

(1) [*Generic iterative kernel construction.*] A very general iterative kernel is constructed from the given iterative algorithm. See Section 5.1.
(2) [*Algorithm-tree construction.*] An algorithm tree is constructed from the generic iterative kernel. See Section 5.2.
(3) [*Algorithm-tree labeling.*] Same as Section 3.3.
(4) [*Algorithm-DAG construction.*] Same as Section 3.4.

5.1 Generic Iterative Kernel Construction

In this step, we construct a generic iterative kernel from the given iterative algorithm. The generic kernel can replace the iterative kernel of each recursive function in the standard two-way R-DP corresponding to the iterative algorithm. We explain how to construct such a kernel through an example later.

We will construct a generic kernel for our parenthesis R-DP from the iterative algorithm Loop-Parenthesis given in Figure 1. The generic kernel will accept three $(n' + 1) \times (n' + 1)$ subtables of the original $(n + 1) \times (n + 1)$ DP table $G[0 : n, 0 : n]$ as inputs, namely, $X[0 : n', 0 : n']$, $U[0 : n', 0 : n']$, and $V[0 : n', 0 : n']$, where $0 \leq n' \leq n$. These are the same subtables that an R-DP function receives when it invokes this generic kernel. Since Autogen assumes that $n + 1$ is a power of 2 and in each level of recursion it divides each dimension of the input table into equal halves, we can safely assume that $n' + 1$ is also a power of 2. The kernel will update each $G[i, j] \in X$

```
GENERIC-LOOP-PARENTHESIS($X, U, V, n'$)        {$X[0:n', 0:n']$, $U[0:n', 0:n']$ and $V[0:n', 0:n']$ are
    subtables of $G[0:n, 0:n]$ and region tuple $\langle X, U, V\rangle$ is received as an input by an R-DP function that
    invokes this generic kernel. For $Z \in \{X, U, V\}$, $(z_r, z_c)$ is the index of the top-left corner cell of $Z$ in $G$.}

(1) if $x_r \neq u_r$ or $x_c \neq v_c$ or $u_c \neq v_r$ then return
(2) for $i' \leftarrow n'$ downto 0 do                              {loop directions do not change}
(3)   for $j' \leftarrow 0$ to $n'$ do                              {loop boundaries are checked inside
(4)     for $k' \leftarrow 0$ to $n'$ do                            (in Line 8) right before updating $X$}
(5)       $i \leftarrow x_r + i'$                                   {$x_r = u_r \implies u_r + i' = x_r + i' (= i)$}
(6)       $j \leftarrow x_c + j'$                                   {$x_c = v_c \implies v_c + j' = x_c + j' (= j)$}
(7)       $k \leftarrow u_c + k'$                                   {$u_c = v_r \implies v_r + k' = u_c + k' (= k)$}
(8)       if $j \geq i + 2$ and $i \leq k$ and $k \leq j$ then      {loop boundaries from LOOP-PARENTHESIS of Figure 1}
(9)         $X[i', j'] \leftarrow \min(X[i', j'], U[i', k'] + V[k', j'] + w(i, k, j))$    {$X[i', j']$ maps to $G[i, j]$,
                                                                    $U[i', k']$ to $G[i, k]$, and $V[k', j']$ to $G[k, j]$}
```

Fig. 5. The generic iterative kernel for the parenthesis problem derived from LOOP-PARENTHESIS of Figure 1.

using $G[i,k] \in U$ and $G[k,j] \in V$ following Recurrence 1 for $i, j, k \in [0, n]$. When $n' = n$, that is, $X \equiv U \equiv V \equiv G$, the generic kernel will apply exactly the same set of updates on G as applied by LOOP-PARENTHESIS and in exactly the same order.

We assume that each pair of subtables from X, U, and V are either completely overlapping or completely disjoint, which follows from the way AUTOGEN recursively divides G into subtables and passes the subtables to R-DP functions through recursive calls. Let (x_r, x_c) be the index of the top-left corner cell of X in G; that is, cell $X[i,j]$ corresponds to cell $G[x_r + i, x_c + j]$. Similarly, (u_r, u_c) for U and (v_r, v_c) for V. When $n' = 0$, that is, $X \equiv G[x_r, x_c]$, $U \equiv G[u_r, u_c]$, and $V \equiv G[v_r, v_c]$, Recurrence 1 and thus LOOP-PARENTHESIS in Figure 1 imply that X will be updated provided

$$(1)\ x_r = u_r (= i), \quad (2)\ x_c = v_c (= j), (3)\ u_c = v_r (= k),$$

$$(4)\ j \geq i + 2, (5)\ i \leq k, \text{and } (6)\ k \leq j.$$

Observe that conditions (1) to (3) trivially hold when we make the initial R-DP function call with $X \equiv U \equiv V \equiv G$. The way G is recursively decomposed and passed to recursive functions ensures that those conditions continue to hold for each function call that updates X at least once.

Conditions (4) to (6) are enforced by the loop bounds in LOOP-PARENTHESIS. But in the generic kernel those conditions will be moved inside the innermost loop, leaving each loop to span the entire range from 0 to n' without changing the direction of any loop. Those conditions will be checked right before applying each update on X to make sure that the generic kernel does not apply an update not allowed by Recurrence 1.

The resulting generic looping kernel that we call GENERIC-LOOP-PARENTHESIS is shown in Figure 5.

The $\Theta(n^h)$ work required for a DP problem can be mapped to a $\Theta(n^h)$ hypercubic grid. Then, the conditions satisfied by the indices (e.g., in Line 8 of GENERIC-LOOP-PARENTHESIS in Figure 5) can be viewed as choosing half-spaces in that hypercubic grid. The grid points that satisfy all the conditions form a polyhedron in the h-dimensional integer space. Updates will be applied on points in that polyhedron.

5.2 Algorithm-Tree Construction

The major difference between the algorithm-tree construction phases of inductive and deductive AUTOGEN lies in the way region-tuple dependencies are found. While the inductive version explicitly generates the cell set and recursively distributes its contents among region tuples in order to determine which region-tuple dependencies exist (see Section 3.2), the deductive version identifies such dependencies without ever explicitly generating and storing the entire cell set. Instead, deductive AUTOGEN determines whether a given region-tuple dependency exists or not either by

Table 3. Dependencies of Quadrants in the Parenthesis Problem

Dependency	Exists?	Dependency	Exists?	Dependency	Exists?	Dependency	Exists?
$\langle G_{11}, G_{11}, G_{11}\rangle$	√	$\langle G_{12}, G_{11}, G_{11}\rangle$	✗	$\langle G_{21}, G_{11}, G_{11}\rangle$	✗	$\langle G_{22}, G_{11}, G_{11}\rangle$	✗
$\langle G_{11}, G_{11}, G_{12}\rangle$	✗	$\langle G_{12}, G_{11}, G_{12}\rangle$	√	$\langle G_{21}, G_{11}, G_{12}\rangle$	✗	$\langle G_{22}, G_{11}, G_{12}\rangle$	✗
$\langle G_{11}, G_{11}, G_{21}\rangle$	✗	$\langle G_{12}, G_{11}, G_{21}\rangle$	✗	$\langle G_{21}, G_{11}, G_{21}\rangle$	✗	$\langle G_{22}, G_{11}, G_{21}\rangle$	✗
$\langle G_{11}, G_{12}, G_{22}\rangle$	✗	$\langle G_{12}, G_{12}, G_{22}\rangle$	√	$\langle G_{21}, G_{12}, G_{22}\rangle$	✗	$\langle G_{22}, G_{12}, G_{22}\rangle$	✗
$\langle G_{11}, G_{12}, G_{11}\rangle$	✗	$\langle G_{12}, G_{12}, G_{11}\rangle$	✗	$\langle G_{21}, G_{12}, G_{11}\rangle$	✗	$\langle G_{22}, G_{12}, G_{11}\rangle$	✗
$\langle G_{11}, G_{12}, G_{12}\rangle$	✗	$\langle G_{12}, G_{12}, G_{12}\rangle$	✗	$\langle G_{21}, G_{12}, G_{12}\rangle$	✗	$\langle G_{22}, G_{12}, G_{12}\rangle$	✗
$\langle G_{11}, G_{12}, G_{21}\rangle$	✗	$\langle G_{12}, G_{12}, G_{21}\rangle$	✗	$\langle G_{21}, G_{12}, G_{21}\rangle$	✗	$\langle G_{22}, G_{12}, G_{21}\rangle$	✗
$\langle G_{11}, G_{12}, G_{22}\rangle$	✗	$\langle G_{12}, G_{12}, G_{22}\rangle$	✗	$\langle G_{21}, G_{12}, G_{22}\rangle$	✗	$\langle G_{22}, G_{12}, G_{22}\rangle$	✗
$\langle G_{11}, G_{21}, G_{11}\rangle$	✗	$\langle G_{12}, G_{21}, G_{11}\rangle$	✗	$\langle G_{21}, G_{21}, G_{11}\rangle$	✗	$\langle G_{22}, G_{21}, G_{11}\rangle$	✗
$\langle G_{11}, G_{21}, G_{12}\rangle$	✗	$\langle G_{12}, G_{21}, G_{12}\rangle$	✗	$\langle G_{21}, G_{21}, G_{12}\rangle$	✗	$\langle G_{22}, G_{21}, G_{12}\rangle$	✗
$\langle G_{11}, G_{21}, G_{21}\rangle$	✗	$\langle G_{12}, G_{21}, G_{21}\rangle$	✗	$\langle G_{21}, G_{21}, G_{21}\rangle$	✗	$\langle G_{22}, G_{21}, G_{21}\rangle$	✗
$\langle G_{11}, G_{21}, G_{22}\rangle$	✗	$\langle G_{12}, G_{21}, G_{22}\rangle$	✗	$\langle G_{21}, G_{21}, G_{22}\rangle$	✗	$\langle G_{22}, G_{21}, G_{22}\rangle$	✗
$\langle G_{11}, G_{22}, G_{11}\rangle$	✗	$\langle G_{12}, G_{22}, G_{11}\rangle$	✗	$\langle G_{21}, G_{22}, G_{11}\rangle$	✗	$\langle G_{22}, G_{22}, G_{11}\rangle$	✗
$\langle G_{11}, G_{22}, G_{12}\rangle$	✗	$\langle G_{12}, G_{22}, G_{12}\rangle$	✗	$\langle G_{21}, G_{22}, G_{12}\rangle$	✗	$\langle G_{22}, G_{22}, G_{12}\rangle$	✗
$\langle G_{11}, G_{22}, G_{21}\rangle$	✗	$\langle G_{12}, G_{22}, G_{21}\rangle$	✗	$\langle G_{21}, G_{22}, G_{21}\rangle$	✗	$\langle G_{22}, G_{22}, G_{21}\rangle$	✗
$\langle G_{11}, G_{22}, G_{22}\rangle$	✗	$\langle G_{12}, G_{22}, G_{22}\rangle$	✗	$\langle G_{21}, G_{22}, G_{22}\rangle$	✗	$\langle G_{22}, G_{22}, G_{22}\rangle$	√

solving an integer programming feasibility testing problem or by explicitly checking each cell tuple belonging to the given region tuple to see if the corresponding update is allowed by the underlying DP recurrence.

Consider the parenthesis problem as an example. Let $\mathrm{A}(\langle G, G, G\rangle)$ be the initial R-DP function call that takes the entire DP table $G[0:n, 0:n]$ as input. The R-DP will divide G into four quadrants: G_{11}, G_{12}, G_{21}, and G_{22}. As shown in Table 3, the total number of possible region-tuple dependencies at this level is $4 \times 4 \times 4 = 64$, though only four of them exist. The region-tuple dependencies that exist are then recursively broken down into subregion-tuple dependencies as needed. Continuing in this way, we can create the entire algorithm tree.

We will now focus on computing the existence of a region-tuple dependency. Let $X[0:n', 0:n']$, $U[0:n', 0:n']$ and $V[0:n', 0:n']$ be arbitrary $(n'+1) \times (n'+1)$ subtables of the $(n+1) \times (n+1)$ DP table G. For $Z \in \{X, U, V\}$, let (z_r, z_c) be the coordinates of the top-left corner cell of Z in G, where both z_r and z_c are functions of n (and so is n'). Now asking whether region-tuple dependency $\langle X, U, V\rangle$ exists is the same as asking whether there exist i, j, k, n such that $G[i,j] \in X$, $G[i,k] \in U$, and $G[k,j] \in V$, that is, if there exists an n such that U and V include cells that can be used to update X according to Recurrence 1. Formally, we ask if $i, j, k, n \geq 0$ exist such that

$$\begin{aligned}
&(1)\ x_r \leq i \leq x_r + n' \text{ and } u_r \leq i \leq u_r + n',\\
&(2)\ x_c \leq j \leq x_c + n' \text{ and } v_c \leq j \leq v_c + n',\\
&(3)\ u_c \leq k \leq u_c + n' \text{ and } v_r \leq k \leq v_r + n',\\
&(4)\ j \geq i + 2, (5)\ i \leq k, \text{ and } (6)\ k \leq j.
\end{aligned}$$

Observe that the previous conditions are similar to the ones used for designing the generic looping kernel in Section 5.1, but the first three conditions have a more general form because we now assume that the locations of X, U, and V in G are not necessarily related.

```
REGION-DEPENDENCY $(X, U, V, n')$ {$X[0 : n', 0 : n']$, $U[0 : n', 0 : n']$ and $V[0 : n', 0 : n']$
        are subtables of $G[0 : n, 0 : n]$. For $Z \in \{X, U, V\}$,
        $(z_r, z_c)$ is the index of the top-left corner cell of $Z$ in $G$.}

if $x_r \neq u_r$ or $x_c \neq v_c$ or $u_c \neq v_r$ then return FALSE
for $i' \leftarrow n'$ downto $0$ do
  for $j' \leftarrow 0$ to $n'$ do
    for $k' \leftarrow 0$ to $n'$ do
      $i \leftarrow x_r + i'$
      $j \leftarrow x_c + j'$
      $k \leftarrow u_c + k'$
      if $j \geq i + 2$ and $i \leq k$ and $k \leq j$ then return TRUE
return FALSE
```

Fig. 6. Looping code for checking region-tuple dependencies in the parenthesis R-DP.

Clearly, the region-tuple dependency checking problem earlier can be cast as an integer programming feasibility testing problem. Since both the number of variables and the number of inequalities are bounded from above by constants, the space needed to encode the problem (i.e., input length) is $\Theta(1)$. Hence, the integer programming problem given earlier can be checked for feasibility in $\Theta(1)$ time (see, e.g., [31, 34]).

Alternatively, we can modify GENERIC-LOOP-PARENTHESIS from Figure 5 to check for region-tuple dependencies in a brute-force manner. We check only cell tuples belonging to the given region tuple until we either find one that is allowed by the underlying DP recurrence (meaning the region-tuple dependency exists) or exhaust the entire set (meaning the given region-tuple dependency does not exists). The resulting algorithm, which we call REGION-DEPENDENCY, is given in Figure 6. We first choose a small value for n' large enough to capture all types of dependency relationships implied by the underlying DP recurrence. We then keep n' fixed to that value for all levels of the algorithm-tree construction phase while increasing n as we go deeper into the tree by setting $n = 2^t n'$ at level t. This approach is the opposite of what inductive AUTOGEN does (see Section 3), which keeps n fixed but reduces n' as it goes deeper into the algorithm tree.

Space/Time Complexity of Deductive Autogen

We analyze the space and time complexities of the deductive AUTOGEN using the three parameters d, s, and l, where d is the number of DP table dimensions, $1 + s$ is the cell-tuple size, and l is the threshold level. Let the total number of functions in the output R-DP be m. Then $m \geq l$.

The number of subregion-tuple dependencies for a given region-tuple dependency at any level is $2^{d(1+s)}$. If we consider a total of l levels, then the number of region-tuple dependencies will be $O(m2^{d(1+s)})$. The time taken to check the existence of a region-tuple dependency is $g(d, s)$ for some function g. Hence, the total time taken to construct the algorithm tree is $O(m2^{d(1+s)}g(d, s))$. The time taken to construct the DAGs is asymptotically smaller than the time taken to create the algorithm tree. Therefore, the total time taken for a deductive AUTOGEN algorithm to generate an R-DP is $O(m2^{d(1+s)}g(d, s))$. If we run this algorithm serially, then the space usage is $O(m2^{1+s})$.

6 EXTENSIONS OF AUTOGEN

In this section, we briefly discuss how to extend AUTOGEN to **(1)** handle one-way sweep property (Property 1) violation and **(2)** sometimes reduce the space usage of the generated R-DP algorithms.

6.1 Handling One-Way Sweep Property (Property 1) violation

The following three-step procedure works for dynamic programs that compute paths over a closed semiring in a directed graph [54]. Floyd-Warshall's algorithm for finding all-pairs shortest path (APSP) [22] belongs to this class and is shown in Figure 7.

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

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

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

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

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

(a)

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

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

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

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

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

(b)

Fig. 7. (a) An autogenerated R-DP algorithm from the cubic space Floyd-Warshall's APSP algorithm. In the initial call to $\mathrm{A}^{3D}_{FW}(\langle X, Y, X, X \rangle)$, X points to $G[1..n, 1..n, 1..n]$ and Y points to an n^3 matrix whose topmost plane is initialized with $G[1..n, 1..n, 0]$. (b) An R-DP algorithm obtained by projecting the 3D matrix $G[1..n, 1..n, 0..n]$ accessed by the algorithm in column (a) to its 2D base $G[1..n, 1..n, 0]$.

(i) Project I-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 onto a higher-dimensional space, leading to a correct I-DP that satisfies the one-way sweep property.

(ii) Autodiscover R-DP from I-DP. AUTOGEN is applied on the higher-dimensional I-DP that satisfies Property 1 to discover an R-DP in the same higher-dimensional space.

(iii) Project R-DP back to original dimension. The autogenerated R-DP is projected back to the original lower-dimensional space. One can show that the projected R-DP correctly implements the original I-DP [15, 17].

6.2 Space Reduction

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

Example. The LCS problem [13, 29] asks one to find the longest of all common subsequences [17] 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 $\Theta(n^2)$ space I-DP, we generate an R-DP for the problem that contains four recursive functions. The autogenerated R-DP still uses $\Theta(n^2)$ space and incurs $O(n^2/B)$ cache misses. One can reason as follows in order to reduce the space usage of this R-DP and thereby improving its cache performance.

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

6.3 Non-Orthogonal Regions

In this section, we generalize the definition of a region to include nonrectangular areas as well. We restrict ourselves to two-dimensional DP tables. As per the original definition of a region, the entire DP table was divided orthogonally into four equal quadrants, which are then recursively subdivided into subquadrants in the same way until each subdivision contained only a single cell. Each such (sub)quadrant was called a region. This definition of a region may not work correctly for DP recurrences that fill out nonrectangular areas of a DP table, for example, for the Cocke-Younger-Kasami (CYK) algorithm for parsing context-free grammars [16, 32, 57]. We extend the definition of a region to handle such DP problems.

We define a term called compute shape that will be used subsequently.

Definition 6.1 (Compute Cells, Compute Shape). The set of all cells that will be computed in a DP table for a given DP problem by its DP algorithm is called compute cells. The geometric shape and area the compute cells represent is called compute shape.

Given a DP problem that does not violate Property 1, the fixed polygonal compute shape S with q vertices $p_1, p_2, \ldots, p_q$ is found. The shape S is scaled down by a factor of 2 to S' having vertices $p'_1, p'_2, \ldots, p'_q$ such that $\forall i$, p'_i corresponds to p_i. The shape S' can move as opposed to S, which is fixed on the DP table. The shape S' is moved inside the fixed shape S without rotating such that S' is completely contained in S and for some i, $p_i = p'_i$ and that common area is denoted by S'_i. If all the cells inside S'_i, for some i, depend on the cells inside S'_i alone, then the area S'_i is called a region, more specifically a self-dependent region. The disjoint areas when all self-dependent regions are removed from S are also called regions. If we obtain parallelograms for regions, then they can be divided into four equal parallelograms using lines parallel to the sides of the parallelogram, and each of those smaller parallelograms will be regions at the next level.

As an example, Figure 8 shows the use of the generalized definition of regions to divide compute shapes in two DP problems with nonorthogonal regions.

7 EXPERIMENTAL RESULTS

This section presents empirical results showing the performance benefits and robustness of Autogen-discovered algorithms.

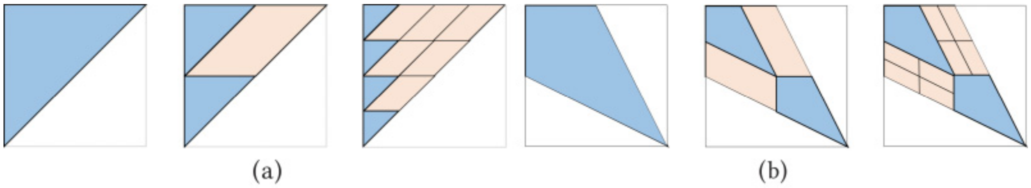


Fig. 8. Nonorthogonal regions for the first three levels depicted for (a) Cocke-Younger-Kasami (CYK) algorithm [16, 32, 57] and (b) spoken word recognition [43].

7.1 Experimental Setup

All our experiments were performed on a multicore machine with dual-socket eight-core 2.7GHz[5] Intel Sandy Bridge processors (2 × 8 = 16 cores in total) and 32GB RAM. Each core was connected to a 32KB private L1 cache and a 256KB private L2 cache. All cores in a processor shared a 20MB L3 cache. All algorithms were implemented in C++. We used the Intel Cilk Plus extension to parallelize and Intel C++ Compiler v13.0 to compile all implementations with optimization parameters `-O3 -ipo -parallel -AVX -xhost`. PAPI 5.3 [1] was used to count cache misses, and the MSR (Model-Specific Register) module and `likwid` [53] were used for energy measurements. We used `likwid` for the adaptivity (Figure 13) experiments. All `likwid` measurements were end to end (i.e., captured everything from the start to the end of the program).

Given an iterative description of a DP in the Fractal-DP class, our Autogen prototype generates pseudocode of the corresponding R-DP algorithm in the format shown in Figure 1. We implemented such autodiscovered R-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 R-DP implementation switched to an iterative kernel when the problem size became sufficiently small (e.g., 64 × 64). All our R-DP implementations were the straightforward implementation of the pseudocode with only *trivial hand-optimizations*. With nontrivial hand-optimizations, R-DP algorithms can achieve even more speedup (see [51]). Trivial optimizations included:

(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) write optimization in the basecase—if each iteration of an innermost loop updated the same location of the DP table, we performed all those updates in a local variable instead of modifying the DP table cell over and over again, and updated that cell only once using the updated local variable after the loop terminated;
(iii) using registers for variables that are accessed many times inside the loops; and
(iv) using `#pragma` directives to autovectorize/autoparallelize code.

Nontrivial optimizations that we did not apply included:

(i) using Z-morton row-major layout (see [51]) to store the matrices,
(ii) using pointer arithmetic and converting all multiplicative indexing to additive indexing, and
(iii) using explicit vectorization.

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

The major optimizations applied on I-DP codes included 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 (version 0.11.3) [7]—a state-of-the-art polyhedral compiler—to generate parallel tiled iterative codes for the parenthesis problem, gap problem, and Floyd-Warshall's APSP (2D). Optimized versions of these codes are henceforth called *tiled I-DP*. After analyzing the autogenerated codes, we found that the parenthesis implementation had temporal locality as it was tiled along 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 R-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 $\Theta(n^3/B)$, matching the cache complexity of its I-DP counterpart. Similarly, the $\Theta(n^3)$ cache complexity of tiled gap matched that of I-DP gap.

The major optimizations we applied on the parallel tiled codes generated by Pluto included:

(i) use of `#pragma ivdep`, `#pragma parallel`, and `#pragma min loop count(B)` directives;
(ii) write optimizations (as was used for basecases of R-DP);
(iii) use of empirically determined best tile sizes; and
(iv) rigorous optimizations using pointer arithmetic, additive indexing, and so forth.

The type of trivial copy optimization we used in R-DP did not improve spatial locality of the autogenerated tiled I-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 the autogenerated tiled code for the parenthesis problem had to be reversed in order to avoid violation of dependency constraints.

All algorithms we tested were in-place; that is, they used only a constant number of extra memory/register locations in addition to the given DP table. The copy optimization required the use of a small local submatrix per thread, but its size was also independent of the input DP table size. None of our optimizations reduced space usage. The write optimization avoided 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 R-DP, I-DP, and tiled I-DP implementations for all three problems when each of the programs was run on a dedicated machine. We show that R-DP outperformed I-DP in terms of runtime, scalability, cache misses, and energy consumption. Next, we show how the performance of R-DP, I-DP, and tiled I-DP implementations changed in a multiprogramming environment when multiple processes shared cache space and bandwidth.

7.2 Single-Process Performance

Figures 9 to 11 show detailed performance results of I-DP, tiled I-DP, and R-DP implementations of the parenthesis problem (Figure 9), the gap problem (Figure 10), and Floyd-Warshall's APSP (Figure 11). For each of the three problems, our R-DP implementation outperformed its I-DP counterpart, and for $n = 8,192$, the speedup factors with respect to parallel I-DP on 16 cores were around 18, 17, and 6 for parenthesis, gap, and Floyd-Warshall's APSP, respectively. For parenthesis and gap

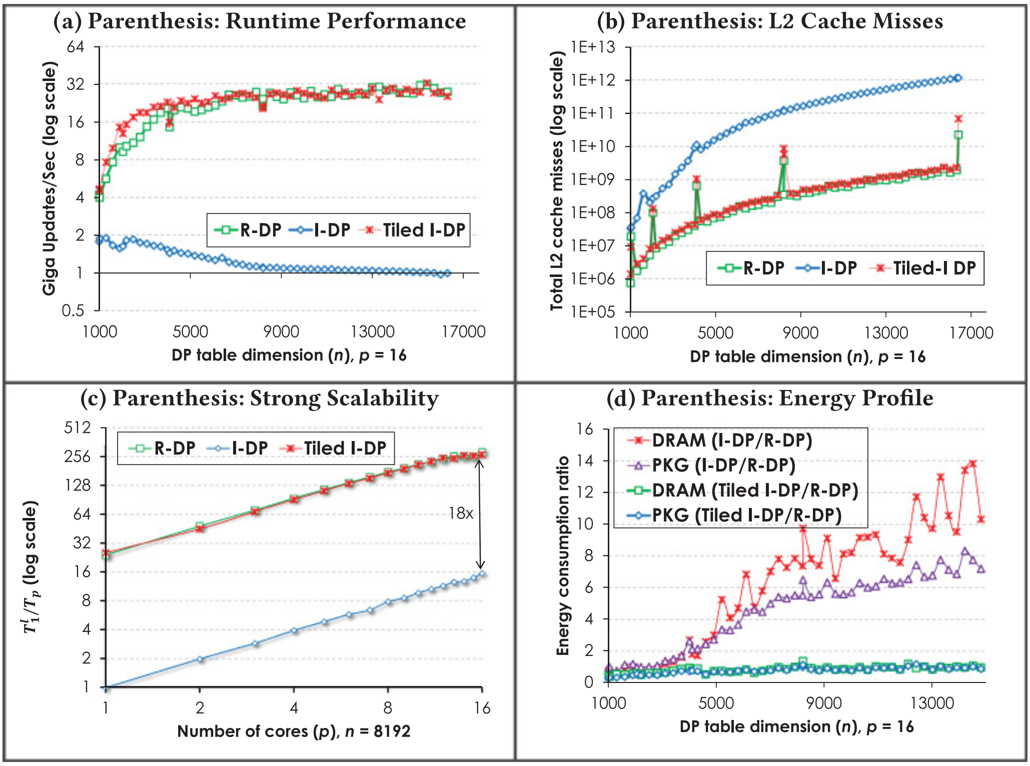


Fig. 9. Performance comparison of I-DP, R-DP, and tiled I-DP for the parenthesis problem: (a) Giga updates per second achieved by all algorithms, (b) L2 cache misses for each program, (c) strong scalability with #cores, p when n is fixed at 8, 192 (in this plot T_1^I denotes the running time of I-DP when $p = 1$), and (d) ratios of total joule energy consumed by Package (PKG) and DRAM. Here, tiled I-DP is an optimized version of the parallel tiled code generated by Pluto [7].

problems, I-DP consumed 5.5 times more package energy and 7.4 times more DRAM energy than R-DP when $n = 8,192$. For Floyd-Warshall's APSP, those two factors were 7.4 and 18, respectively.

For the parenthesis problem, tiled I-DP (i.e., our optimized version of Pluto-generated parallel tiled code) and R-DP had almost identical performance for $n > 6,000$. For $n \leq 6,000$, R-DP was slower than tiled I-DP, but for larger n, R-DP was marginally (1–2%) faster on average. Though tiled I-DP and R-DP had almost similar L2 cache performance, Figure 12 shows that R-DP incurred noticeably fewer L1 and L2 cache misses than those incurred by tiled I-DP, which helped R-DP to eventually fully overcome the overhead of recursion and other implementation overheads. This happened because the tile size of tiled I-DP was optimized for the L2 cache, but R-DP, being cache oblivious, was able to adapt to all levels of the cache hierarchy simultaneously [23].

As explained in Section 7.1 for the gap problem, tiled I-DP had suboptimal cache complexity, matching that of I-DP. As a result, tiled I-DP's performance curves were closer to those of I-DP than R-DP, and R-DP outperformed it by a wide margin. This was similar for Floyd-Warshall's APSP. However, in case of a gap problem, tiled I-DP incurred significantly fewer L3 misses than I-DP (not shown in the plots), and as a result, it consumed less DRAM energy. The opposite was true for Floyd-Warshall's APSP.

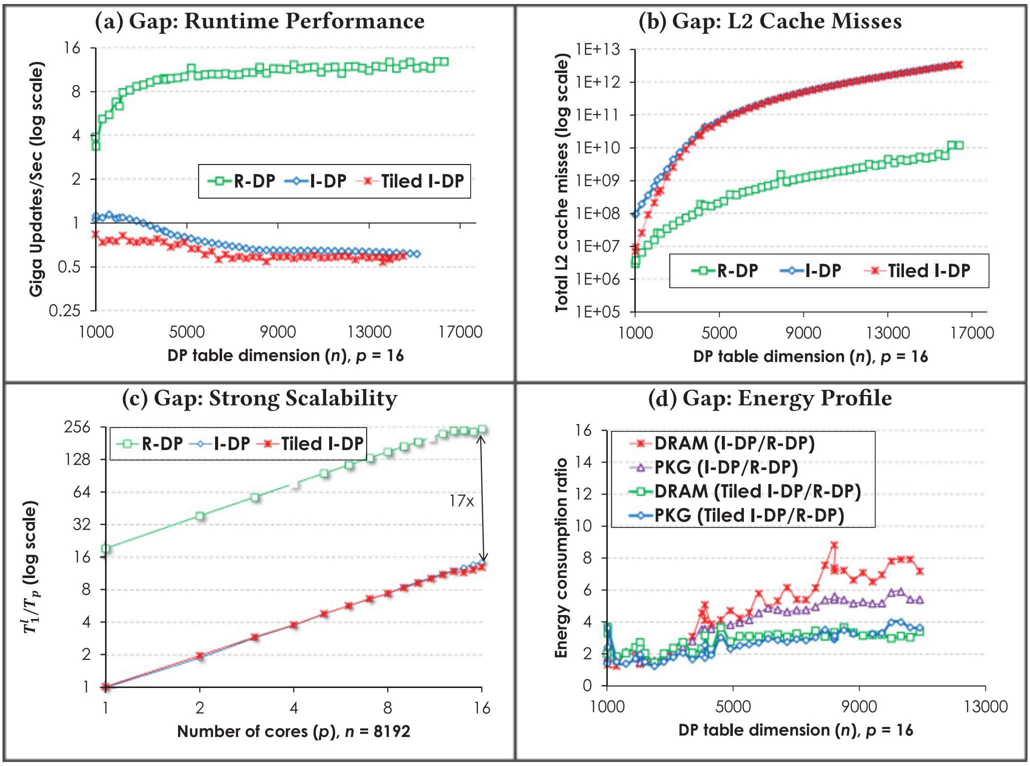


Fig. 10. Performance comparison of I-DP, R-DP, and tiled I-DP for the gap problem: (a) Giga updates per second achieved by all algorithms, (b) L2 cache misses for each program, (c) strong scalability with #cores, p when n is fixed at $8, 192$ (in this plot T_1^I denotes the running time of I-DP when $p = 1$), and (d) ratios of total joule energy consumed by Package (PKG) and DRAM. Here, tiled I-DP is an optimized version of the parallel tiled code generated by Pluto [7].

7.3 Multiprocess Performance

R-DP algorithms are more robust than both I-DP and tiled I-DP. Our empirical results show that in a multiprogramming environment, R-DP algorithms were less likely to significantly slow down when the available shared cache/memory space was reduced (unlike tiled code with temporal locality), and less likely to suffer when the available bandwidth was reduced (unlike standard I-DP code and tiled I-DP without temporal locality). Figures 13 and 14 show the results for the parenthesis problem. We saw similar trends for our other benchmark problems (e.g., FW-APSP).

We performed experimental analyses of how the performance of a program (R-DP, I-DP, and tiled I-DP) changed if multiple copies of the same program were run on the same multicore processor (Figure 13). We ran up to four instances of the same program on an eight-core Sandy Bridge processor with two threads (i.e., cores) per process. The block size of the tiled code was optimized for best performance with two threads. With four concurrent processes, I-DP slowed down by 55% and tiled I-DP by 130%, but R-DP lost only 17% of its performance (see Figure 13). The slowdown of the tiled code resulted from its inability to adapt to the loss of the shared cache space, which increased its L3 misses by a factor of 4.6 (see Figure 13). On the other hand, L3 misses incurred by R-DP increased by less than a factor of 1.6. Since I-DP did not have any temporal locality, loss of ache space did not significantly change the number of L3 misses it incurred. But I-DP already

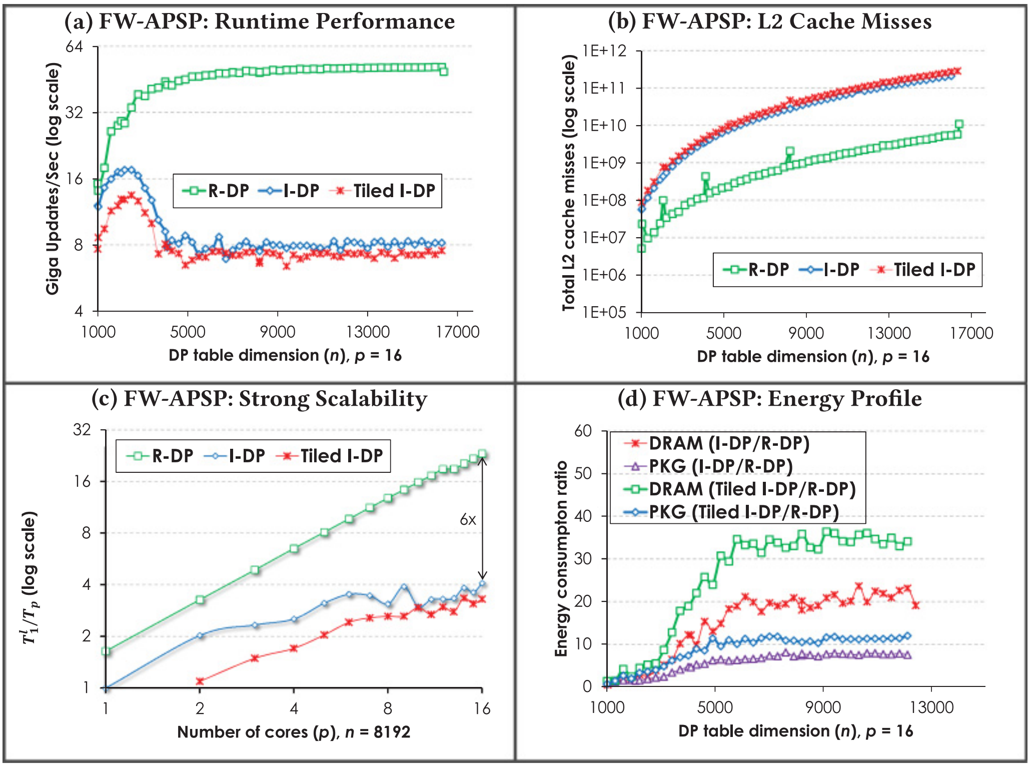


Fig. 11. Performance comparison of I-DP, R-DP, and tiled I-DP for the Floyd-Warshall's APSP: (a) Giga updates per second achieved by all algorithms, (b) L2 cache misses for each program, (c) strong scalability with #cores, p when n is fixed at 8, 192 (in this plot T_1^I denotes the running time of I-DP when $p = 1$), and (d) ratios of total joule energy consumed by Package (PKG) and DRAM. Here, tiled I-DP is an optimized version of the parallel tiled code generated by Pluto [7].

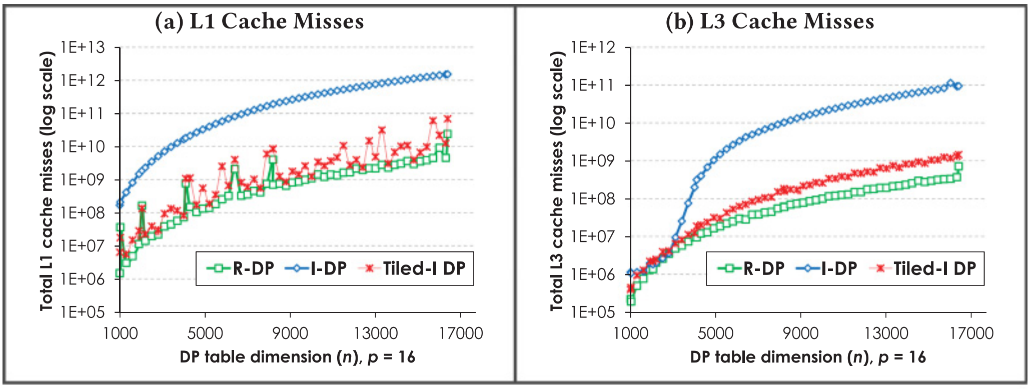


Fig. 12. The plots show the L1 and L3 cache misses incurred by the three algorithms for solving the parenthesis problem. L2 cache misses are shown in Figure 9(b).

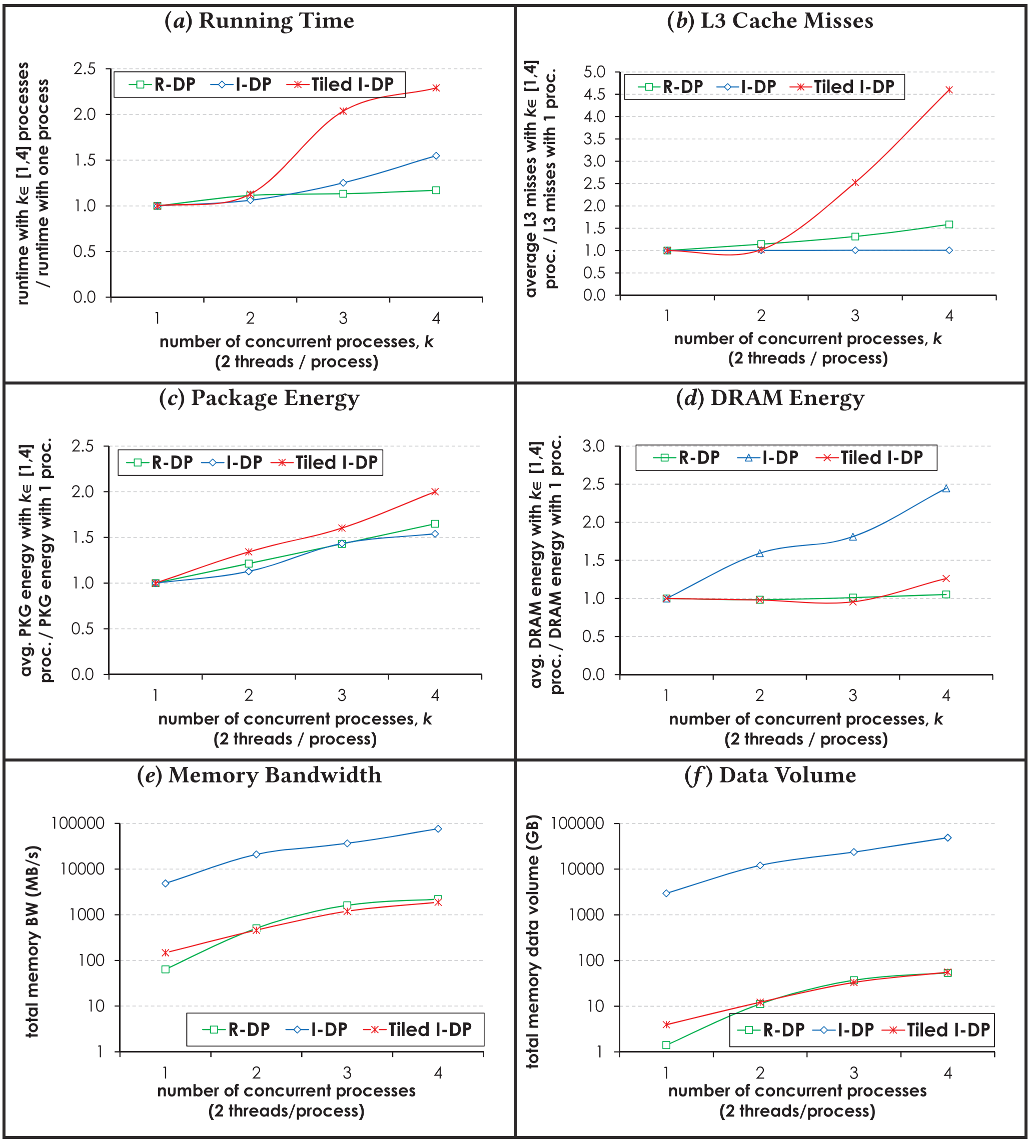


Fig. 13. The plots show how the performances of standard looping, tiled looping, and recursive codes for the parenthesis problem (for $n = 2^{13}$) are affected as multiple instances of the same program are run on an eight-core Intel Sandy Bridge with 20MB shared L3 cache.

incurred 3700 times more L3 misses than R-DP, and with four such concurrent processes the pressure on the DRAM bandwidth increased considerably (see Figure 13), causing significant slowdown of the program.

We also report changes in energy consumption of the processes as the number of concurrent processes increases (Figure 13). 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

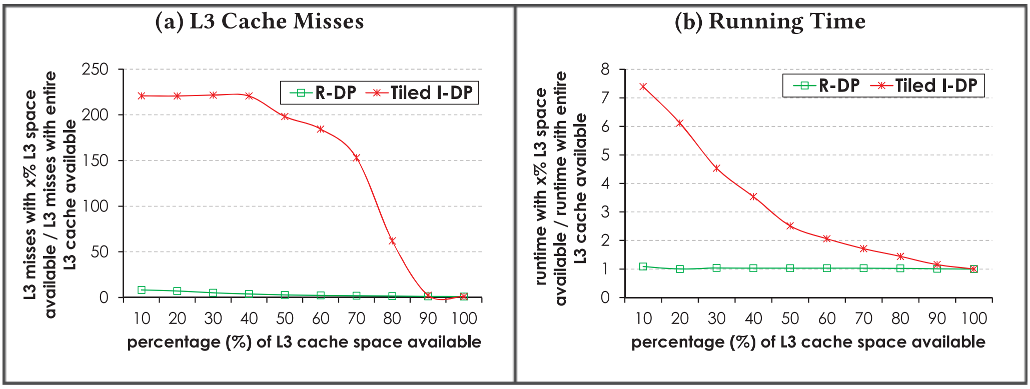


Fig. 14. The plots show how changes in the available shared L3 cache space affect (a) the number of L3 cache misses and (b) the serial running time of the tiled looping code and the recursive code solving the parenthesis problem for $n = 2^{13}$. The code under test was run on a single core of an eight-core Intel Sandy Bridge processor with 20MB shared L3 cache. A multithreaded `Cache Pirate` [21] was run on the remaining cores.

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 looked similar to that of package energy. A single instance of tiled I-DP consumed 5% less energy than an R-DP instance, while I-DP consumed 9 times more energy. Average package and PP0 energy consumed by tiled I-DP increased at a faster rate than that by R-DP as the number of processes increased. This happened because both its running time and L3 performance degraded faster than R-DP, both of which contributed to energy performance. However, since for I-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 R-DP's with fewer than three processes. However, as the number of processes increased, energy consumption increased for I-DP at a faster rate, and perhaps because of the DRAM bandwidth contention its DRAM energy consumption increased significantly.

We measured the effect of reducing the available shared L3 cache space on running times and L3 cache misses of serial R-DP and serial tiled I-DP[6] (shown in Figure 14). In this case, the serial tiled-I-DP algorithm was running around 50% faster than the serial R-DP code. The `Cache Pirate` tool [21] 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 R-DP the increase was only 17%. As a result, the tiled I-DP slowed down by over 50%, while for R-DP the slowdown was less than 3%. Thus, R-DP automatically adapted to cache sharing [6], but the tiled I-DP did not. This result can be found in the second column of Figure 13.

7.4 Inductive Vs. Deductive Autogen

In this section, we compare the performance of inductive (Section 3) and deductive (Section 5) versions of Autogen. For constructing algorithm trees in deductive Autogen, we used the approach based on the generic iterative kernel (i.e., Region-Dependency in Section 5.2). Table 4 compares the time taken by the programs to generate R-DP algorithms for the parenthesis problem, Floyd-Warshall's APSP (3D version), and the gap problem. Both programs were run on a multicore

[6] With tile size optimized for best serial performance.
[7] `Cache Pirate` allows only a single program to run, and does not reduce bandwidth.

Table 4. Comparison of the Time Needed by the Inductive and the Deductive Versions of Autogen to Generate R-DP Algorithms for Three Dynamic Programming Problems (Both Programs Were Run on a Multicore Machine with Dual-Socket 8-Core 2.7GHz Intel Sandy Bridge Processors ($2 \times 8 = 16$ Cores in Total) and 32GB RAM)

Problem	Inductive Autogen (Section 3)	Deductive Autogen (Section 5)
Parenthesis problem [14]	0.04s	0.01s
Floyd-Warshall's APSP 3-D [15]	0.03s	0.02s
Gap problem [8]	0.16s	0.02s

machine with 16 cores (dual-socket Intel Sandy Bridge) and 32GB RAM. Though the running times were low for both programs, the deductive version clearly outperformed the inductive version.

We also ran both programs on Sankoff's $\Theta(n^6)$ time and $\Theta(n^4)$ space DP for combined alignment and folding of RNA secondary structures (particularly, Recurrence 15 in [44]), where n is the length of the RNA sequences. While deductive Autogen was able to generate a correct R-DP in 0.63s, inductive Autogen ran out of space in the 32GB RAM before it could identify all recursive functions.

8 CONCLUSIONS

We have presented the Autogen algorithm that can generate provably correct and highly efficient parallel recursive algorithms for a wide class of DP problems called the Fractal-DP. The input to Autogen is a black-box implementation of any correct algorithm (e.g., an inefficient serial iterative algorithm) for solving the given DP problem. Autogen uses the input black-box implementation to discover the access pattern of the given DP problem and learn a decomposition that can be used to generate a recursive divide-and-conquer algorithm for solving it. The process is fully automatic.

Autogen has been designed for DP problems that are data oblivious; that is, DP table accesses are independent of the data values in the table itself. However, there are important DP problems such as knapsack, Viterbi, and subset sum that do not fall into that category. We are now actively working on extending Autogen to handle such data-dependent DP problems as well as classes of data-oblivious DP problems beyond Fractal-DP.

ACKNOWLEDGMENTS

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

REFERENCES

[1] PAPI. 2017. Performance Application Programming Interface (PAPI). http://icl.cs.utk.edu/papi/.

[2] XSEDE. 2017. Extreme Science and Engineering Discovery Environment (XSEDE). http://www.xsede.org/.

[3] Nawaaz Ahmed and Keshav Pingali. 2000. Automatic generation of block-recursive codes. In *Proceedings of the 6th European Conference on Parallel Processing (Euro-Par'00)*. 368–378.

[4] Vineet Bafna and Nathan Edwards. 2003. On de novo interpretation of tandem mass spectra for peptide identification. In *Proceedings of the 7th Annual International Conference on Research in Computational Molecular Biology (RCMB'03)*. 9–18.

[5] Richard Bellman. 1957. *Dynamic Programming*. Princeton University Press.

[6] Michael Bender, Roozbeh Ebrahimi, Jeremy Fineman, Golnaz Ghasemiesfeh, Rob Johnson, and Samuel McCauley. 2014. Cache-adaptive algorithms. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'14)*. 958–971.

[7] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. *ACM SIGPLAN Notices* 43, 6 (2008), 101–113.

[8] Rezaul Chowdhury. 2007. *Cache-Efficient Algorithms and Data Structures: Theory and Experimental Evaluation*. Ph.D. Dissertation. Department of Computer Sciences, The University of Texas, Austin, Texas.

[9] Rezaul Chowdhury and Pramod Ganapathi. Divide-and-Conquer Variants of Bubble, Selection, and Insertion Sorts. Unpublished manuscript.

[10] Rezaul Chowdhury, Pramod Ganapathi, Vivek Pradhan, Jesmin Jahan Tithi, and Yunpeng Xiao. 2016. An efficient cache-oblivious parallel Viterbi algorithm. In *Proceedings of the 22nd European Conference on Parallel Processing (Euro-Par'16)*. 574–587.

[11] Rezaul Chowdhury, Pramod Ganapathi, Yuan Tang, and Jesmin Jahan Tithi. 2017. Provably efficient scheduling of cache-oblivious wavefront algorithms. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'17)*. 339–350.

[12] Rezaul Chowdhury, Pramod Ganapathi, Jesmin Jahan Tithi, Charles Bachmeier, Bradley Kuszmaul, Charles Leiserson, Armando Solar-Lezama, and Yuan Tang. 2016. AutoGen: Automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)*. Article 10.

[13] Rezaul Chowdhury and Vijaya Ramachandran. 2006. Cache-oblivious dynamic programming. In *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'06)*. 591–600.

[14] Rezaul Chowdhury and Vijaya Ramachandran. 2008. Cache-efficient dynamic programming algorithms for multicores. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'08)*. 207–216.

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

[16] John Cocke. 1969. *Programming Languages and Their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sciences, New York University.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.

[18] Jun Du, Ce Yu, Jizhou Sun, Chao Sun, Shanjiang Tang, and Yanlong Yin. 2013. EasyHPS: A multilevel hybrid parallel system for dynamic programming. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW'13)*. 630–639.

[19] F. C. Duckworth and A. J. Lewis. 1998. A fair method for resetting the target in interrupted one-day cricket matches. *Journal of the Operational Research Society* 49, 3 (1998), 220–227.

[20] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. 1998. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.

[21] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. 2011. Cache pirating: Measuring the curse of the shared cache. In *Proceeding of the 40th International Conference on Parallel Processing (ICPP'11)*. 165–175.

[22] Robert W. Floyd. 1962. Algorithm 97: Shortest path. *CACM* 5, 6 (1962), 345.

[23] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS'99)*. 285–297.

[24] Zvi Galil and Kunsoo Park. 1994. Parallel algorithms for dynamic programming recurrences with more than $O(1)$ dependency. *J. Parallel and Distrib. Comput.* 21, 2 (1994), 213–222.

[25] Pramod Ganapathi. 2016. *Automatic Discovery of Efficient Divide-&-Conquer Algorithms for Dynamic Programming Problems*. Ph.D. Dissertation. Department of Computer Science, Stony Brook University.

[26] Robert Giegerich and Georg Sauthoff. 2011. Yield grammar analysis in the Bellman's GAP compiler. In *Proceedings of the 11th Workshop on Language Descriptions, Tools and Applications (LDTA'11)*. Article 7.

[27] Dan Gusfield. 1997. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press.

[28] Frederick S. Hillier and Gerald J. Lieberman. 2010. *Introduction to Operations Research* (9th ed.). McGraw-Hill.

[29] Daniel S. Hirschberg. 1975. A linear space algorithm for computing maximal common subsequences. *Commun. ACM* 18, 6 (1975), 341–343.

[30] Shachar Itzhaky, Rohit Singh, Armando Solar-Lezama, Kuat Yessenov, Yongquan Lu, Charles Leiserson, and Rezaul Chowdhury. 2016. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*. ACM, 145–164.

[31] Ravi Kannan. 1987. Minkowski's convex body theorem and integer programming. *Mathematics of Operations Research* 12, 3 (1987), 415–440.

[32] Tadao Kasami. 1965. *An efficient recognition and syntax-analysis algorithm for context-free languages*. Technical Report. Department of Electrical Engineering, Hawaii University, Honolulu.

[33] John O. S. Kennedy. 1981. Applications of dynamic programming to agriculture, forestry and fisheries: Review and prognosis. *Review of Market Agricultural Economics* 49, 3 (1981), 141–173.

[34] Hendrik W. Lenstra Jr. 1983. Integer programming with a fixed number of variables. *Mathematics of Operations Research* 8, 4 (1983), 538–548.

[35] Anany Levitin. 2011. *Introduction to the Design and Analysis of Algorithms* (3rd ed.). Pearson.

[36] Art Lew and Holger Mauch. 2006. *Dynamic Programming: A Computational Tool*. Studies in Computational Intelligence, Vol. 38. Springer.

[37] Weiguo Liu and Bertil Schmidt. 2004. A generic parallel pattern-based system for bioinformatics. In *Proceedings of the 10th European Conference on Parallel Processing (Euro-Par'04)*. Springer, 989–996.

[38] Yewen Pu, Rastislav Bodik, and Saurabh Srivastava. 2011. Synthesis of first-order dynamic programming algorithms. *ACM SIGPLAN Notices* 46, 10 (2011), 83–98.

[39] Raphael Reitzig. 2012. Automated Parallelisation of Dynamic Programming Recursions. *Masters Thesis: University of Kaiserslautern* (2012).

[40] Alexander A. Robichek, Edwin J. Elton, and Martin J. Gruber. 1971. Dynamic programming applications in finance. *The Journal of Finance* 26, 2 (1971), 473–506.

[41] David Romer. 2002. *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.

[42] John Rust. 1996. Numerical dynamic programming in economics. *Handbook of Computational Economics* 1 (1996), 619–729.

[43] Hiroaki Sakoe and Seibi Chiba. 1978. Dynamic programming algorithm optimization for spoken word recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing* 26, 1 (1978), 43–49.

[44] David Sankoff. 1985. Simultaneous solution of the RNA folding, alignment and protosequence problems. *SIAM Journal on Applied Mathematics* 45, 5 (1985), 810–825.

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

[46] Moshe Sniedovich. 2010. *Dynamic Programming: Foundations and Principles*. CRC Press.

[47] Shanjiang Tang, Ce Yu, Jizhou Sun, Bu-Sung Lee, Tao Zhang, Zhen Xu, and Huabei Wu. 2012. EasyPDP: An efficient parallel dynamic programming runtime system for computational biology. *IEEE Transactions on Parallel and Distributed Systems* 23, 5 (2012), 862–872.

[48] Yuan Tang, Rezaul Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The Pochoir stencil compiler. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)*. 117–128.

[49] Yuan Tang, Rezaul Chowdhury, Chi-Keung Luk, and Charles E. Leiserson. 2011. Coding stencil computations using the Pochoir stencil-specification language. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Parallelism (HotPar'11)*.

[50] Yuan Tang, Ronghui You, Haibin Kan, Jesmin Jahan Tithi, Pramod Ganapathi, and Rezaul A. Chowdhury. 2015. Cache-oblivious wavefront: Improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency. *ACM SIGPLAN Notices* 50, 8 (2015), 205–214.

[51] Jesmin Tithi, Pramod Ganapathi, Aakrati Talati, Sonal Agarwal, and Rezaul Chowdhury. 2015. High-performance energy-efficient recursive dynamic programming with matrix-multiplication-like flexible kernels. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS'15)*.

[52] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkens-Diehr. 2014. XSEDE: Accelerating scientific discovery. *Computing in Science and Engineering* 16, 5 (2014), 62–74. DOI: http://dx.doi.org/10.1109/MCSE.2014.80

[53] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 39th International Conference on Parallel Processing Workshops (ICPPW'10)*. 207–216.

[54] Jeffrey D. Ullman, Alfred V. Aho, and John E. Hopcroft. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading.

[55] Chen Wang, Ce Yu, Shanjiang Tang, Jian Xiao, Jizhou Sun, and Xiangfei Meng. 2016. A general and fast distributed system for large-scale dynamic programming applications. *Parallel Computing* 60 (2016), 1–21.

[56] Michael S. Waterman. 1995. *Introduction to Computational Biology: Maps, Sequences and Genomes*. Chapman & Hall Ltd.

[57] Daniel H. Younger. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control* 10, 2 (1967), 189–208.

Received January 2017; revised June 2017; accepted July 2017