Automatic Discovery of Efficient Divide-&-Conquer Algorithms for Dynamic Programming Problems

Pramod Ganapathi
(Stony Brook University)


ACM Transactions on Parallel Computing (TOPC), 2017
Vision

Problem specifications → Algorithms

Automatic discovery of efficient algorithms
Automation in CS

IBM Watson

Mathematica

E-mail

Navigation

Automated testing

Text-to-speech converter
Automatic programming

Problem specification

Efficient algorithm

High-performing implementation
Automatic code generation

Problem specification

Efficient algorithm

High-performing implementation
Automatic algorithm design

Problem specification
Efficient algorithm
High-performing implementation
Aim: Automatic algorithm design

Problem specification

Efficient algorithm
Automatic algorithm design is inevitable

Programming is hard

Machine architectures are changing

Problems are increasing
State-of-the-art in automatic programming

- Automated parameter tuning
- Automatic parallelizer and locality optimizer
- Program synthesis
- Machine learning
- Self-improving algorithms

No algorithms that can discover nontrivial (in structure) D&C algorithms
We focus on dynamic programming (DP) problems.

DP recurrence → Efficient DP algorithm
Dynamic programs are important

RNA/protein folding
All-pairs shortest path
Viterbi algorithm
Word wrapping

Chain matrix multiplication
Edit distance
Knapsack problem
Function approximation
D&C is the best way to implement dynamic programs

I-DP = iterative algorithms
Tiled I-DP = blocked iterative algorithms
R-DP = recursive divide-and-conquer algorithms

<table>
<thead>
<tr>
<th>Feature</th>
<th>I-DP</th>
<th>Tiled I-DP</th>
<th>R-DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache-efficiency</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache-obliviousness</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache-adaptivity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High parallelism</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Highly optimizable kernels</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Energy efficiency</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bandwidth efficiency</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Currently, D&C algorithms are designed manually

<table>
<thead>
<tr>
<th>Problem</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence alignment</td>
<td>I-DP</td>
</tr>
<tr>
<td>Parenthesis problem</td>
<td>I-DP</td>
</tr>
<tr>
<td>Gap problem</td>
<td>I-DP</td>
</tr>
<tr>
<td>Floyd Warshall’s APSP</td>
<td>I-DP</td>
</tr>
<tr>
<td>Multi-instance Viterbi</td>
<td>I-DP</td>
</tr>
<tr>
<td>Spoken word recog.</td>
<td>I-DP</td>
</tr>
<tr>
<td>Function approx.</td>
<td>I-DP</td>
</tr>
</tbody>
</table>
Computational scientists need efficient dynamic programs

- **Physicist**
  How can I efficiently evaluate my heat diffusion DP recurrence?

- **Economist**
  How can I efficiently evaluate my asset pricing DP recurrence?

- **Biologist**
  How can I efficiently evaluate my protein folding DP recurrence?
We want to automatically discover efficient and portable DP algorithms

Blackbox implementation of a DP recurrence

Efficient, portable, & robust algorithm
Parallelism and cache locality are the key factors to improve performance.

Parallelism

Cache locality
Communication is more expensive than computation

<table>
<thead>
<tr>
<th>Feature</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>RAM</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access time</td>
<td>0.5ns</td>
<td>7ns</td>
<td>20ns</td>
<td>100ns</td>
<td>1 ms</td>
</tr>
<tr>
<td>Size</td>
<td>32KB</td>
<td>256KB</td>
<td>20MB</td>
<td>32GB</td>
<td>2TB</td>
</tr>
<tr>
<td>Cost</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>$150</td>
<td>$150</td>
</tr>
<tr>
<td>Technology</td>
<td>SRAM</td>
<td>SRAM</td>
<td>SRAM</td>
<td>DRAM</td>
<td>Magnetic</td>
</tr>
</tbody>
</table>

Source: Intel and StackExchange forums
I-DP --> Autogen --> R-DP
Example: Parenthesis problem

• Parenthesization that minimizes the cost

• Applications:
  • Optimal chain matrix multiplication
  • RNA secondary structure prediction
  • Optimal binary search trees
  • Optimal polygon triangulation
  • String parsing for context-free grammar – CYK algorithm
  • Optimal natural join of database tables – Selinger algorithm
  • Maximum perimeter inscribed polygon
  • Offline job scheduling minimizing flow time of jobs
Parenthesis problem recurrence

Input: A string $s_1s_2 ... s_n$

$C[i, j] =$ cost of parenthesization from $s_i$ to $s_j$

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

Output: $C[1, n]$
Dependency structure for the parenthesis problem
Slow loop DP algorithms (I-DPs)

Loop-Parenthesis($C, n$)
1. for $i \leftarrow n - 1$ downto 1 do
2. for $j \leftarrow i + 2$ to $n$ do
3. for $k \leftarrow i$ to $j$ do
4. $C[i, j] \leftarrow \min \left( \frac{C[i, k] + C[k, j]}{w(i, k, j), C[i, j]} \right)$

Parallel-Loop-Parenthesis($C, n$)
1. for $t \leftarrow 2$ to $n - 1$ do
2. parallel for $i \leftarrow 1$ to $n - t$ do
3. $j \leftarrow t + i$
4. for $k \leftarrow i$ to $j$ do
5. $C[i, j] \leftarrow \min \left( \frac{C[i, k] + C[k, j]}{w(i, k, j), C[i, j]} \right)$

Serial I-DP

Parallel I-DP
Fast D&C DP algorithm (R-DP)

\[
A(X)
\begin{align*}
1. & \text{ if } X \text{ is a small matrix then } A\text{-loop}(X) \\
2. & \text{ else} \\
3. & \text{ parallel: } A(X_{11}), A(X_{22}) \\
4. & B(X_{12}, X_{11}, X_{22})
\end{align*}
\]

\[
B(X, U, V)
\begin{align*}
1. & \text{ if } X \text{ is a small matrix then } B\text{-loop}(X, U, V) \\
2. & \text{ else} \\
3. & B(X_{21}, U_{22}, V_{11}) \\
4. & \text{ parallel: } C(X_{11}, U_{12}, V_{21}), C(X_{22}, X_{21}, V_{12}) \\
5. & \text{ parallel: } B(X_{11}, U_{11}, V_{11}), B(X_{22}, X_{22}, V_{22}) \\
6. & C(X_{12}, U_{12}, X_{22}) \\
7. & C(X_{12}, X_{11}, V_{12}) \\
8. & B(X_{12}, U_{11}, V_{22})
\end{align*}
\]

\[
C(X, U, V)
\begin{align*}
1. & \text{ if } X \text{ is a small matrix then } C\text{-loop}(X, U, V) \\
2. & \text{ else} \\
3. & \text{ parallel: } C(X_{11}, U_{11}, V_{11}), C(X_{12}, U_{11}, V_{12}) \\
4. & \text{ parallel: } C(X_{21}, U_{21}, V_{11}), C(X_{22}, U_{21}, V_{12}) \\
5. & \text{ parallel: } C(X_{11}, U_{12}, V_{21}), C(X_{12}, U_{12}, V_{22}) \\
6. & \text{ parallel: } C(X_{21}, U_{22}, V_{21}), C(X_{22}, U_{22}, V_{22})
\end{align*}
\]
Fast D&C DP algorithm (R-DP)
I-DP $\rightarrow$ Autogen $\rightarrow$ R-DP

**Loop-Parenthesis** $(C, n)$
1. for $i \leftarrow n - 1$ downto 1 do
2.  for $j \leftarrow i + 2$ to $n$ do
3.  for $k \leftarrow i$ to $j$ do
4.    $C[i, j] \leftarrow \min \left( C[i, k] + C[k, j] + w(i, k, j), C[i, j] \right)$

**A(X)**
1. if $X$ is a small matrix then A-loop $(X)$
2. else
3. parallel: A$(X_{11})$, A$(X_{22})$
4. B$(X_{12}, X_{11}, X_{22})$

**B(X, U, V)**
1. if $X$ is a small matrix then B-loop $(X, U, V)$
2. else
3. B$(X_{21}, U_{22}, V_{11})$
4. parallel: C$(X_{11}, U_{12}, V_{21})$, C$(X_{22}, X_{21}, V_{12})$
5. parallel: B$(X_{11}, U_{11}, V_{11})$, B$(X_{22}, X_{22}, V_{22})$
6. C$(X_{12}, U_{12}, X_{22})$
7. C$(X_{12}, X_{11}, V_{12})$
8. B$(X_{12}, U_{11}, V_{22})$

**C(X, U, V)**
1. if $X$ is a small matrix then C-loop $(X, U, V)$
2. else
3. parallel: C$(X_{11}, U_{11}, V_{11})$, C$(X_{12}, U_{12}, V_{12})$
   C$(X_{21}, U_{21}, V_{11})$, C$(X_{22}, U_{21}, V_{12})$
4. parallel: C$(X_{11}, U_{12}, V_{21})$, C$(X_{12}, U_{12}, V_{22})$
   C$(X_{21}, U_{22}, V_{21})$, C$(X_{22}, U_{22}, V_{22})$
Autogen works on Fractal-DP class of DP problems

One-way sweep property: If a cell $x$ depends on another cell $y$, then $y$ is fully updated before $x$ reads from $y$

Fractal property:
Autogen works on Fractal-DP class of DP problems

One-way sweep property: If a cell $x$ depends on another cell $y$, then $y$ is fully updated before $x$ reads from $y$

Fractal property:

Fractal property satisfied for:
- 75% of 16 DP problems in CLRS book
- 60% of 40 DP problems in Lew and Mauch’s book
Core idea: Find recursive patterns from DP table updates

Generate DP table accesses for a small DP table. Find recursive patterns in DP table accesses. Build a recursive algorithm around these recursive access patterns.

\[
\begin{align*}
&((1,3), (1,1), (1,3)) \\
&((1,3), (1,2), (2,3)) \\
&((1,3), (1,3), (3,3)) \\
&((2,4), (2,2), (2,4)) \\
&\ldots \\
&\ldots \\
&((1,64), (1,64), (64,64)) \\
\end{align*}
\]

Step 1. Cell-set generation

Step 2. Algorithm-tree construction

Step 3. Algorithm-tree labeling

Step 4. Algorithm-DAG construction
Step 1. Generate a set of DP table updates

Loop-Parenthesis\((C, n)\)
1. for \(i \leftarrow n - 1\) downto 1 do
2. for \(j \leftarrow i + 2\) to \(n\) do
3. for \(k \leftarrow i\) to \(j\) do
4. \(C[i,j] \leftarrow \min\left( C[i,k] + C[k,j] + w(i,k,j), C[i,j] \right)\)

\((i,j)\) \(\in\) set \((i,k)\) \(\in\) set \((k,j)\) \(\in\) set

Cellset-Generation\((64)\)
1. for \(i \leftarrow 64 - 1\) downto 1 do
2. for \(j \leftarrow i + 2\) to 64 do
3. for \(k \leftarrow i\) to \(j\) do
4. output \(\{(i, j), (i, k), (k, j)\}\)
Step 1. Generate a set of DP table updates

Small problem, say, $n = 64$

Generate cell-dependencies in the form below

$$\langle \text{writecell}, \text{readcell}_1, \ldots, \text{readcell}_s \rangle$$

Cell-set = set of all cell dependencies
Step 2. Construct an algorithm-tree

\[ A(\langle X, X, X \rangle) \]

\[ \langle (1,3), (1,1), (1,3) \rangle \]
\[ \langle (1,3), (1,2), (2,3) \rangle \]
\[ \langle (1,3), (1,3), (3,3) \rangle \]
\[ \langle (2,4), (2,2), (2,4) \rangle \]
\[ \ldots \]
\[ \ldots \]
\[ \ldots \]
\[ \langle (1,64), (1,64), (64,64) \rangle \]

\[ X \]
Step 2. Distribute the DP table updates into buckets

**Bucketing:** Distribute the cell-dependencies into $4 \times 4 \times 4 = 64$ possible buckets

$$\{\text{writecell, readcell}_1, \text{readcell}_2\}$$

$$\{(1,3), (1,1), (1,3)\}$$
$$\{(1,3), (1,2), (2,3)\}$$
$$\{(1,3), (1,3), (3,3)\}$$
$$\{(2,4), (2,2), (2,4)\}$$
$$\ldots$$
$$\ldots$$
$$\ldots$$
$$\{(1,64), (1,64), (64,64)\}$$

$$\{(X_{11}, X_{11}, X_{11})\}$$
$$\{(X_{11}, X_{11}, X_{11})\}$$
$$\{(X_{11}, X_{11}, X_{11})\}$$
$$\{(X_{11}, X_{11}, X_{11})\}$$
$$\ldots$$
$$\ldots$$
$$\ldots$$
$$\{(X_{12}, X_{12}, X_{22})\}$$
Step 2. Distribute the DP table updates into buckets

\[
A((X, X, X)) \\
((1,3), (1,1), (1,3)) \\
((1,33), (1,1), (1,33)) \\
((1,33), (1,33), (33,33)) \\
(33,35), (33,33), (33,35), \\
\ldots
\]

\[
(X) \\

((X_{11}, X_{11}, X_{11})) \\
((X_{12}, X_{11}, X_{12})) \\
((X_{12}, X_{12}, X_{22})) \\
((X_{22}, X_{22}, X_{22}))
\]
Step 2. Combine buckets that write-to and read-from the same regions

\[ A((X, X, X)) \]

\[ \langle (1,3), (1,1), (1,3) \rangle \]
\[ \langle (1,3), (1,2), (2,3) \rangle \]
\[ \langle (1,3), (1,3), (3,3) \rangle \]
\[ \ldots \]
\[ \langle (1,64), (1,64), (64,64) \rangle \]
Step 2. Combine buckets that write-to and read-from the same regions.
Step 2. After combining buckets

\[
\begin{align*}
&\{(1,3), (1,1), (1,3)\} \\
&\{(1,3), (1,2), (2,3)\} \\
&\{(1,3), (1,3), (3,3)\} \\
&\quad \vdots \\
&\{(1,64), (1,64), (64,64)\}
\end{align*}
\]
Step 3. Give function names to nodes of the algorithm-tree

```plaintext
A((X_{11}, X_{11}, X_{11}))  
B((X_{12}, X_{11}, X_{12}), (X_{12}, X_{12}, X_{22}))  
A((X_{22}, X_{22}, X_{22}))
```
Steps 2&3. Recursively expand new functions

\[
B((X_{12}, X_{11}, X_{12}), (X_{12}, X_{12}, X_{22}))
\]

\[
\langle(1,33), (1,1), (1,33)\rangle
\]

\[
\langle(1,33), (1,2), (2,33)\rangle
\]

\[
\langle(1,33), (1,3), (3,33)\rangle
\]

\[
\langle(1,64), (1,64), (64,64)\rangle
\]
Steps 2&3. Recursively expand new functions
Steps 2&3. Labeled algorithm-tree
Step 4. Construct algorithm-DAG

Limitations with algorithm-tree:
- Order of execution
- Parallelism

Solution: Algorithm-DAGs for each function

Four rules to construct algorithm-DAGs
Step 4. Four rules to construct algorithm-DAGs

Rule 1

Rule 2

Rule 3

Rule 4
Step 4. Algorithm-DAGs for different recursive functions
Step 4. The DAGs represent an R-DP

<table>
<thead>
<tr>
<th>A(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. if X is a small matrix then A-loop(X)</td>
</tr>
<tr>
<td>2. else</td>
</tr>
<tr>
<td>3. parallel: A(X_{11}), A(X_{22})</td>
</tr>
<tr>
<td>4. B(X_{12}, X_{11}, X_{22})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B(X, U, V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. if X is a small matrix then B-loop(X, U, V)</td>
</tr>
<tr>
<td>2. else</td>
</tr>
<tr>
<td>3. B(X_{21}, U_{22}, V_{11})</td>
</tr>
<tr>
<td>4. parallel: C(X_{11}, U_{12}, V_{21}), C(X_{22}, X_{21}, V_{12})</td>
</tr>
<tr>
<td>5. parallel: B(X_{11}, U_{11}, V_{11}), B(X_{22}, X_{22}, V_{22})</td>
</tr>
<tr>
<td>6. C(X_{12}, U_{12}, X_{22})</td>
</tr>
<tr>
<td>7. C(X_{12}, X_{11}, V_{12})</td>
</tr>
<tr>
<td>8. B(X_{12}, U_{11}, V_{22})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C(X, U, V)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. if X is a small matrix then C-loop(X, U, V)</td>
</tr>
<tr>
<td>2. else</td>
</tr>
<tr>
<td>3. parallel: C(X_{11}, U_{11}, V_{11}), C(X_{12}, U_{11}, V_{12})</td>
</tr>
<tr>
<td>C(X_{21}, U_{21}, V_{11}), C(X_{22}, U_{21}, V_{12})</td>
</tr>
<tr>
<td>4. parallel: C(X_{11}, U_{12}, V_{21}), C(X_{12}, U_{12}, V_{22})</td>
</tr>
<tr>
<td>C(X_{21}, U_{22}, V_{21}), C(X_{22}, U_{22}, V_{22})</td>
</tr>
</tbody>
</table>
\textbf{I-DP} \rightarrow \textbf{Autogen} \rightarrow \textbf{R-DP}

\textbf{Loop-Parenthesis}(C, n)
1. \textbf{for} \( i \leftarrow n - 1 \) \textbf{down to} 1 \textbf{do}
2. \textbf{for} \( j \leftarrow i + 2 \) \textbf{to} \( n \) \textbf{do}
3. \textbf{for} \( k \leftarrow i \) \textbf{to} \( j \) \textbf{do}
4. \( C[i, j] \leftarrow \min \left( C[i, k] + C[k, j] + w(i, k, j), C[i, j] \right) \)

\textbf{A}(X)
1. if \( X \) is a small matrix then \textbf{A-loop}(X)
2. else
3. \textbf{parallel:} \( \textbf{A}(X_{11}), \textbf{A}(X_{22}) \)
4. \( \textbf{B}(X_{12}, X_{11}, X_{22}) \)

\textbf{B}(X, U, V)
1. if \( X \) is a small matrix then \textbf{B-loop}(X, U, V)
2. else
3. \( \textbf{B}(X_{21}, U_{22}, V_{11}) \)
4. \textbf{parallel:} \( \textbf{C}(X_{11}, U_{12}, V_{21}), \textbf{C}(X_{22}, X_{21}, V_{12}) \)
5. \textbf{parallel:} \( \textbf{B}(X_{11}, U_{11}, V_{11}), \textbf{B}(X_{22}, X_{22}, V_{22}) \)
6. \( \textbf{C}(X_{12}, U_{12}, X_{22}) \)
7. \( \textbf{C}(X_{12}, X_{11}, V_{12}) \)
8. \( \textbf{B}(X_{12}, U_{11}, V_{22}) \)

\textbf{C}(X, U, V)
1. if \( X \) is a small matrix then \textbf{C-loop}(X, U, V)
2. else
3. \textbf{parallel:} \( \textbf{C}(X_{11}, U_{11}, V_{11}), \textbf{C}(X_{12}, U_{12}, V_{12}) \)
\( \textbf{C}(X_{21}, U_{21}, V_{11}), \textbf{C}(X_{22}, U_{21}, V_{12}) \)
4. \textbf{parallel:} \( \textbf{C}(X_{11}, U_{12}, V_{21}), \textbf{C}(X_{12}, U_{12}, V_{22}) \)
\( \textbf{C}(X_{21}, U_{22}, V_{21}), \textbf{C}(X_{22}, U_{22}, V_{22}) \)
## Performance analysis

<table>
<thead>
<tr>
<th>Theory</th>
<th>Work-span model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallelism</td>
<td></td>
</tr>
<tr>
<td>Cache locality</td>
<td>Cache-oblivious model</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Practice</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel Sandy Bridge</td>
</tr>
<tr>
<td>#Cores</td>
<td>16</td>
</tr>
<tr>
<td>L1, L2, L3, RAM</td>
<td>32KB, 256KB, 20MB, 32GB</td>
</tr>
<tr>
<td>Compiler</td>
<td>Intel C++ Compiler</td>
</tr>
<tr>
<td>Parallelization</td>
<td>Intel Cilk Plus</td>
</tr>
<tr>
<td>Count cache misses</td>
<td>PAPI</td>
</tr>
<tr>
<td>I-DP</td>
<td>Parallel + nontrivial opt.</td>
</tr>
<tr>
<td>Tiled I-DP</td>
<td>Pluto-generated + nontrivial opt.</td>
</tr>
<tr>
<td>R-DP</td>
<td>Simple impl. + trivial opt.</td>
</tr>
</tbody>
</table>
### R-DPs are efficient in theory

<table>
<thead>
<tr>
<th>Problem</th>
<th>I-DP</th>
<th>R-DP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cache comp.</td>
<td>Parallelism</td>
</tr>
<tr>
<td>Parenthesis problem</td>
<td>Θ(n³)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>Gap problem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floyd-Warshall's APSP</td>
<td>Θ(n² / log n)</td>
<td></td>
</tr>
<tr>
<td>Protein folding</td>
<td>Θ(n³ / B)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>Function approx.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix multiplication*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-instance Viterbi</td>
<td>Θ(n³ t / B)</td>
<td>Θ(n²)</td>
</tr>
<tr>
<td>LCS / edit distance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spoken word recog.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binomial coefficient</td>
<td>Θ(n² / B)</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>Bitonic TSP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bubble sort*</td>
<td>Θ(1)</td>
<td></td>
</tr>
<tr>
<td>Selection sort*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insertion sort*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*B = block size, M = cache size, * = non-DP problems*
R-DPs are efficient in practice

<table>
<thead>
<tr>
<th></th>
<th>Parenthesis</th>
<th>Gap</th>
<th>FW-APSP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Runtime Performance</strong></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><strong>L2 Cache Misses</strong></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
</tbody>
</table>
R-DPs are efficient in practice

<table>
<thead>
<tr>
<th>Problem</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parenthesis</td>
<td>18x</td>
</tr>
<tr>
<td>Gap</td>
<td>17x</td>
</tr>
<tr>
<td>Floyd-Warshall’s APSP</td>
<td>6x</td>
</tr>
</tbody>
</table>

\[ n = 8192 \text{ and } p = 16 \]

speedup w.r.t. highly parallel and optimized I-DP
Overview of our work

- **AutogenWave**
  - Wave R-DP
  - DP with near-optimal parallelism
  - [SPAA 2017]

- **AutogenFractile**
  - Tiled R-DP
  - Comm.-optimal arch.-independent DP
  - [ICS 2019]

- **Autogen**
  - R-DP
  - Automatic discovery of D&C DP
  - [TOPC 2017]

- **Viterbi algorithm**
  - Efficient, portable algo. for irregular DP
  - [Euro-Par 2016]

- **AutogenTradeoff**
  - Hybrid R-DP
  - Space-adaptive cache-efficient DP
Problem-solving technique

Automation
- Complete
- Partial

Performance
- Cache locality
- Parallelism

D&C

Machine-dependent algorithms
- Cache-aware
- Processor-aware
- Space-aware
- CPUs, GPUs, Clusters

Machine-independent algorithms
- Cache-oblivious
- Processor-oblivious
- Cache-adaptive
- Processor-adaptive