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

Pramod Ganapathi
(Stony Brook University)

Rezaul Chowdhury, Pramod Ganapathi, Stephen Tschudi, Jesmin Jahan Tithi, Charles Bachmeier, Charles E Leiserson, Armando Solar-Lezama, Bradley C Kuszmaul, and Yuan Tang.
ACM Transactions on Parallel Computing (TOPC), 2017

Vision

Problem specifications

Algorithms

Automatic discovery of efficient algorithms

Automation

ATMs

Vending machines

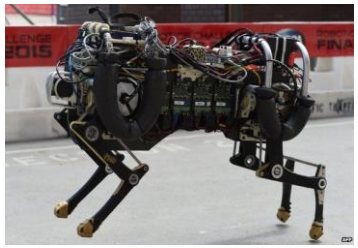
Ticket machines

e-commerce

Manufacturing

House appliances

Self-driving cars



Robots

Automation in CS

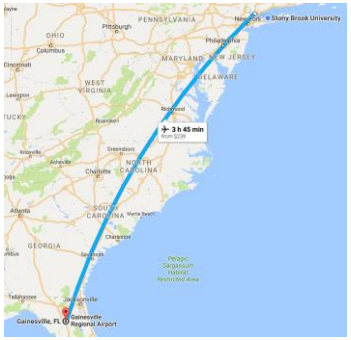
IBM Watson

Wolfram Mathematica

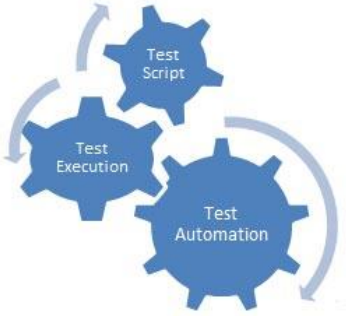
Mathematica

Gmail

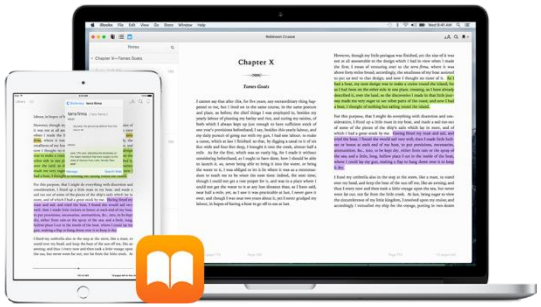
E-mail



Navigation

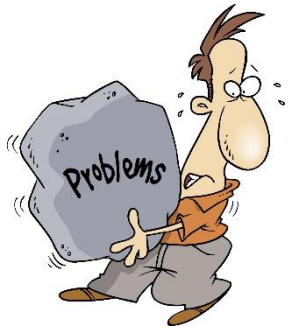


Automated testing



Text-to-speech converter

Automatic programming



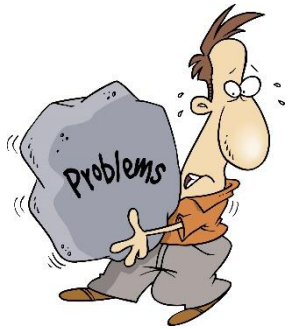
```
static boolean discover(String base, int idx)
{  int n = base.length();
   if (idx == n)     // We have a full one
   {  int digit[] = new int[n], sum = 0, k;

      for (k = 0; k < n; k++)
         digit[k] = base.charAt(k) - '0';
      for (k = sum = 0; k < n; k++)
         sum += (int) Math.pow(digit[k], n);
      if (sum != Integer.valueOf(base))
         return false;
      // Report the successful Armstrong number
      System.out.printf("%d^%d", digit[0], n);
      for (k = 1; k < n; k++)
         System.out.printf(" + %d^%d", digit[k], n);
      System.out.printf(" = %d\n", sum);
      return true;
   }
   else
   {  String  front = base.substring(0, idx),
              back  = base.substring(idx+1);
      char    digit = base.charAt(idx);
      boolean rtn = false;

      for ( ; digit <= '9'; digit++ )
         // Avoid short-circuit evaluation's disabling recursion
         rtn = discover(front + digit + back, idx+1) || rtn;
      return rtn;
   }
}
```

Problem specification

Efficient algorithm

High-performing implementation

Automatic code generation



```
static boolean discover(String base, int idx)
{  int n = base.length();
   if (idx == n)     // We have a full one
   {  int digit[] = new int[n], sum = 0, k;

      for (k = 0; k < n; k++)
         digit[k] = base.charAt(k) - '0';
      for (k = sum = 0; k < n; k++)
         sum += (int) Math.pow(digit[k], n);
      if (sum != Integer.valueOf(base))
         return false;
      // Report the successful Armstrong number
      System.out.printf("%d^%d", digit[0], n);
      for (k = 1; k < n; k++)
         System.out.printf(" + %d^%d", digit[k], n);
      System.out.printf(" = %d\n", sum);
      return true;
   }
   else
   {  String  front = base.substring(0, idx),
              back  = base.substring(idx+1);
      char    digit = base.charAt(idx);
      boolean rtn = false;

      for ( ; digit <= '9'; digit++ )
         // Avoid short-circuit evaluation's disabling recursion
         rtn = discover(front + digit + back, idx+1) || rtn;
      return rtn;
   }
}
```

Problem specification

Efficient algorithm

High-performing implementation

Automatic algorithm design

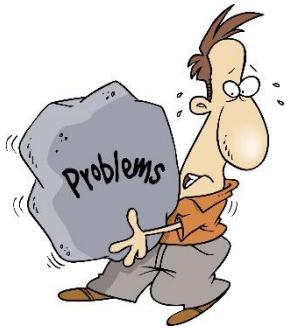
```
static boolean discover(String base, int idx)
{  int n = base.length();
   if (idx == n)     // We have a full one
   {  int digit[] = new int[n], sum = 0, k;

      for (k = 0; k < n; k++)
         digit[k] = base.charAt(k) - '0';
      for (k = sum = 0; k < n; k++)
         sum += (int) Math.pow(digit[k], n);
      if (sum != Integer.valueOf(base))
         return false;
      // Report the successful Armstrong number
      System.out.printf("%d^%d", digit[0], n);
      for (k = 1; k < n; k++)
         System.out.printf(" + %d^%d", digit[k], n);
      System.out.printf(" = %d\n", sum);
      return true;
   }
   else
   {  String  front = base.substring(0, idx),
              back  = base.substring(idx+1);
      char    digit = base.charAt(idx);
      boolean rtn = false;

      for ( ; digit <= '9'; digit++ )
         // Avoid short-circuit evaluation's disabling recursion
         rtn = discover(front + digit + back, idx+1) || rtn;
      return rtn;
   }
}
```

Problem specification

Efficient algorithm

High-performing implementation

Aim: Automatic algorithm design



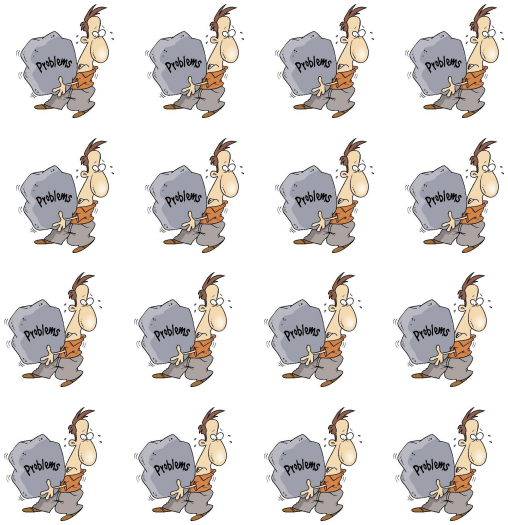
Problem specification

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

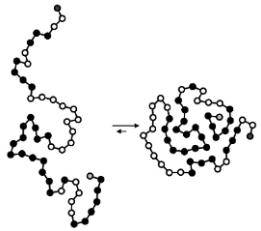
No algorithms that can discover nontrivial (in structure) D&C algorithms

We focus on dynamic programming (DP) problems

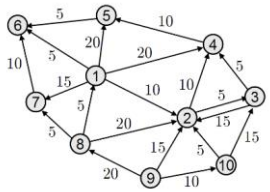
DP recurrence

Efficient DP algorithm

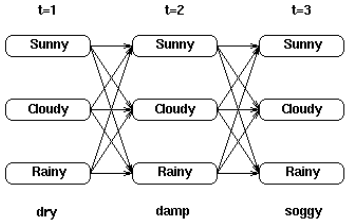
Dynamic programs are important



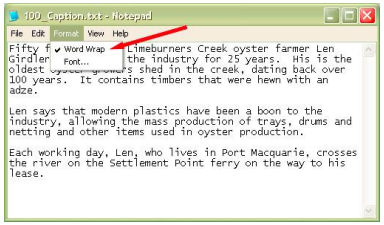
RNA/protein folding



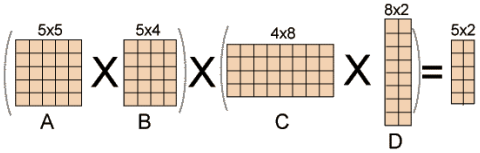
All-pairs shortest path



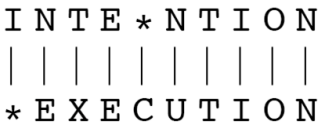
Viterbi algorithm



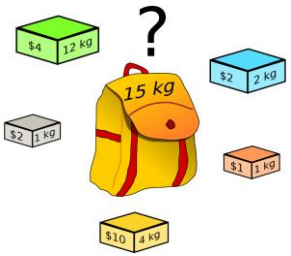
Word wrapping



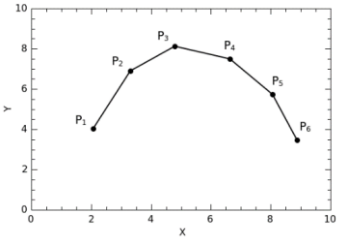
Chain matrix multiplication



Edit distance



Knapsack problem



Function approximation

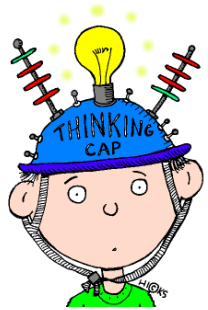
D&C is the best way to implement dynamic programs

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

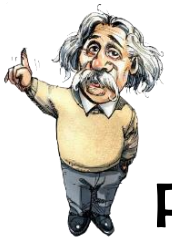
Feature	I-DP	Tiled I-DP	R-DP
Cache-efficiency			
Cache-obliviousness			
Cache-adaptivity			
High parallelism			
Highly optimizable kernels			
Energy efficiency			
Bandwidth efficiency			

Currently, D&C algorithms are designed manually

Sequence alignment	I-DP	⇒	⇒	R-DP
Parenthesis problem	I-DP	⇒	⇒	R-DP
Gap problem	I-DP	⇒	⇒	R-DP
Floyd Warshall's APSP	I-DP	⇒	⇒	R-DP
Multi-instance Viterbi	I-DP	⇒	⇒	R-DP
Spoken word recog.	I-DP	⇒	⇒	R-DP
Function approx.	I-DP	⇒	⇒	R-DP



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?

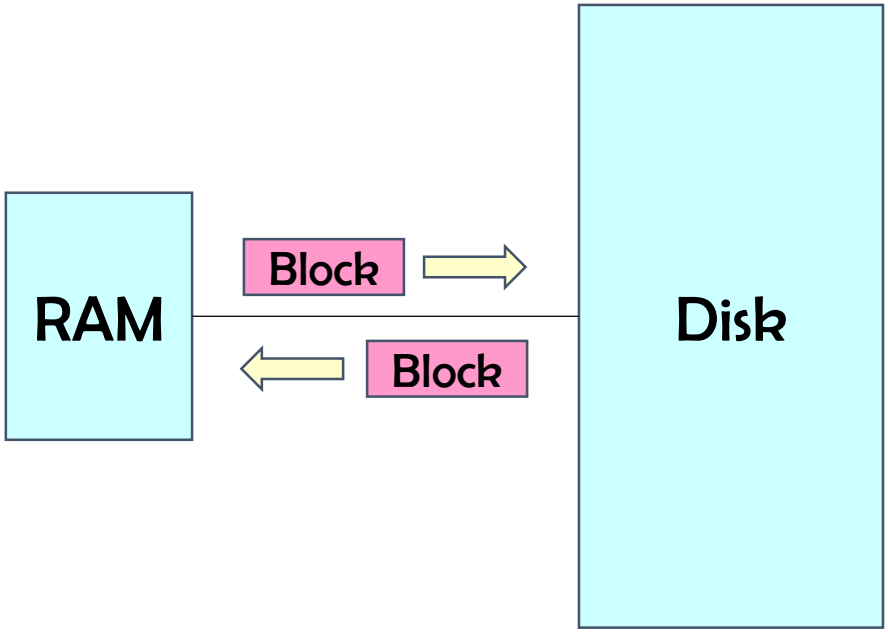
We want to automatically discover efficient and portable DP algorithms

Blackbox implementation of a DP recurrence

Efficient, portable, & robust algorithm

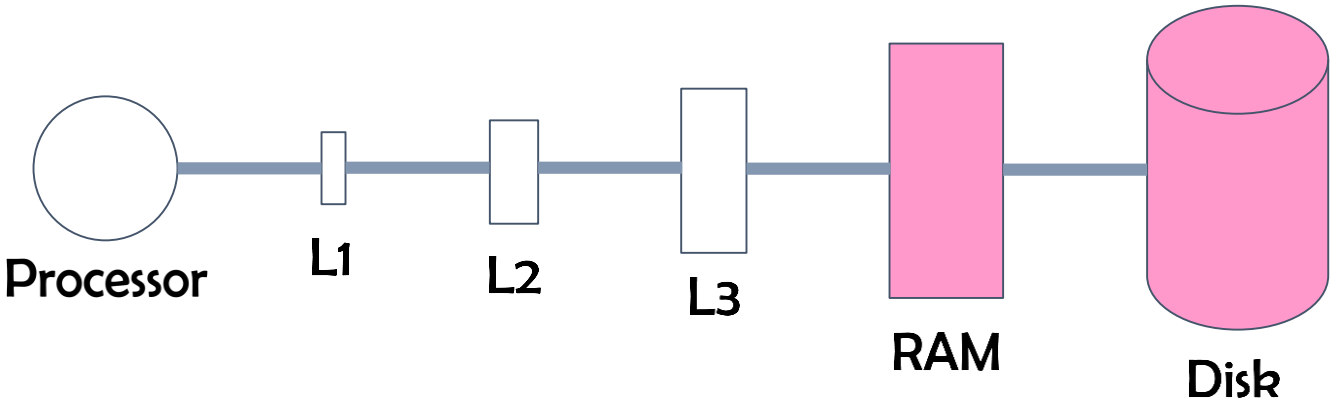
Parallelism and cache locality are the key factors to improve performance

Parallelism



Cache locality

Communication is more expensive than computation



Feature	L1	L2	L3	RAM	Disk
Access time	0.5ns	7ns	20ns	100ns	1 ms
Size	32KB	256KB	20MB	32GB	2TB
Cost	-	-	-	$150	$150
Technology	SRAM	SRAM	SRAM	DRAM	Magnetic

Source: Intel and StackExchange forums

I-DP -> Autogen -> R-DP

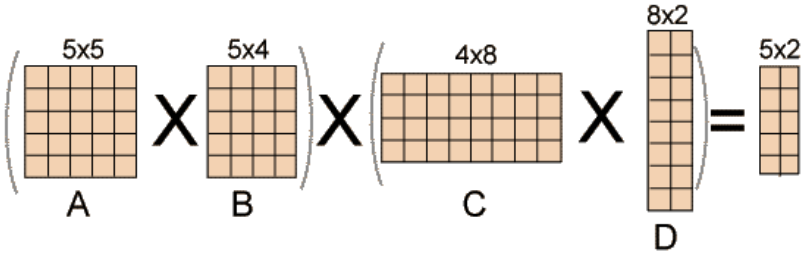
I-DP

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

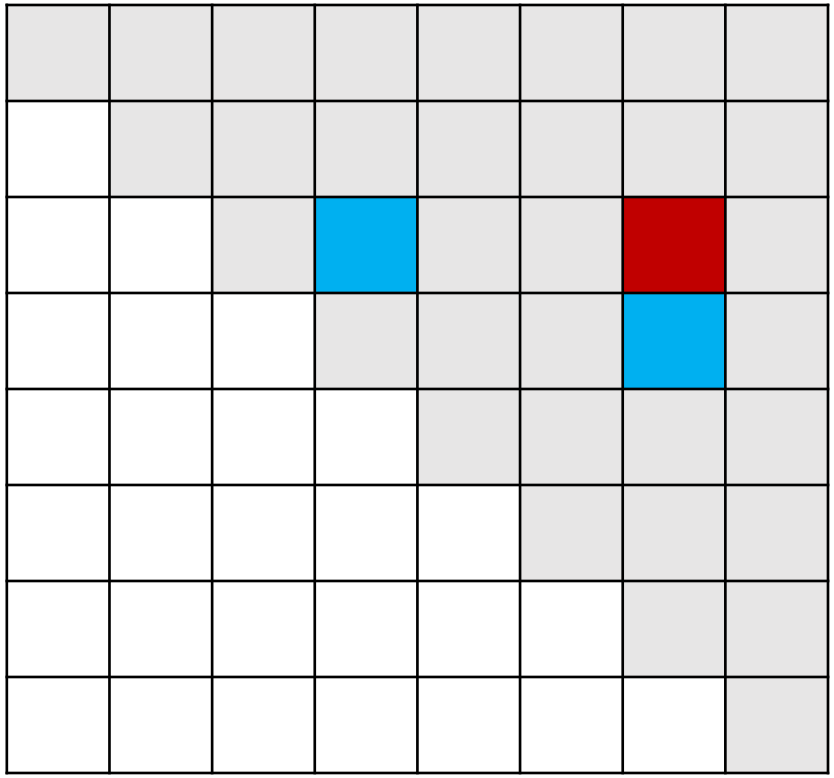
Input: A string $s_1 s_2 \ldots s_n$

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

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

Output: $C[1,n]$

Dependency structure for the parenthesis problem



	Not computed
	Computed
	Write cell
	Read cell

Slow loop DP algorithms (I-DPs)

```
Loop-Parenthesis(C, n)
1.  for i ← n − 1 downto 1 do
2.    for j ← i + 2 to n do
3.      for k ← i to j do
4.        C[i,j] ← min ( C[i,k]+C[k,j] )
                       ( +w(i,k,j),C[i,j] )
```

Serial I-DP

```
Parallel-Loop-Parenthesis(C, n)
1.  for t ← 2 to n − 1 do
2.    parallel for i ← 1 to n − t do
3.      j ← t + i
4.      for k ← i to j do
5.        C[i,j] ← min ( C[i,k]+C[k,j] )
                       ( +w(i,k,j),C[i,j] )
```

Parallel I-DP

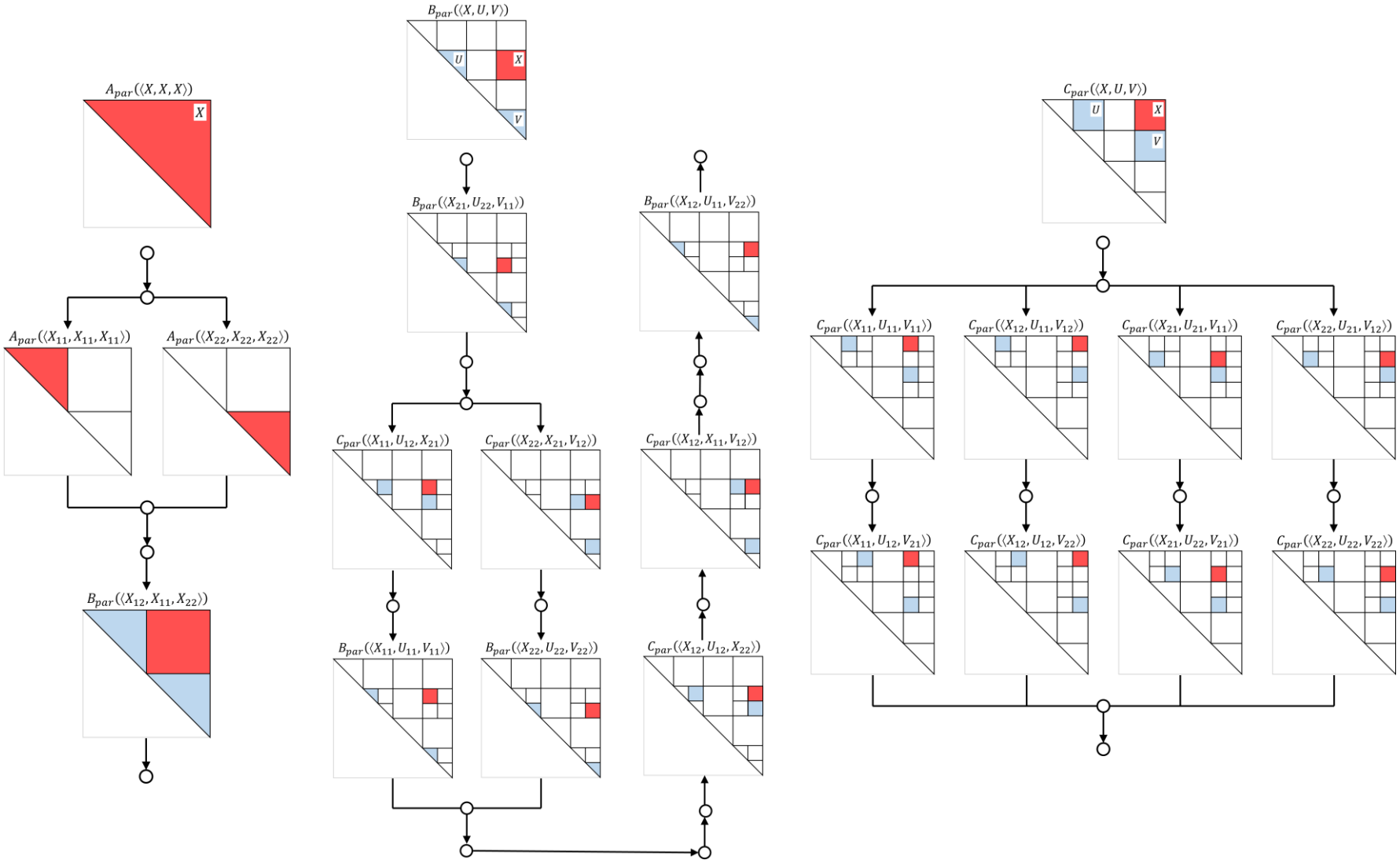
Fast D&C DP algorithm (R-DP)

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_{11}, 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})

Fast D&C DP algorithm (R-DP)



I-DP -> Autogen -> 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. $C[i,j] \leftarrow \min \begin{pmatrix} C[i,k]+C[k,j] \\ +w(i,k,j), C[i,j] \end{pmatrix}$

I-DP

Autogen

R-DP

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})

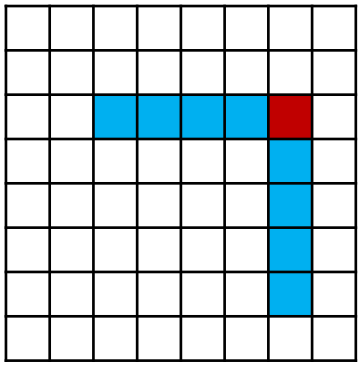
C(X, U, V)

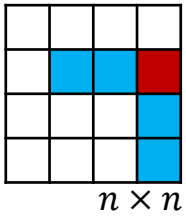
1. if X is a small matrix then C-loop(X, U, V)
2. else
3. parallel: C(X_{11}, U_{11}, V_{11}), C(X_{12}, U_{11}, V_{12})
   C(X_{21}, U_{21}, V_{11}), C(X_{22}, U_{21}, V_{12})
4. parallel: C(X_{11}, U_{12}, V_{21}), C(X_{12}, U_{12}, V_{22})
   C(X_{21}, U_{22}, V_{21}), C(X_{22}, U_{22}, V_{22})

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:

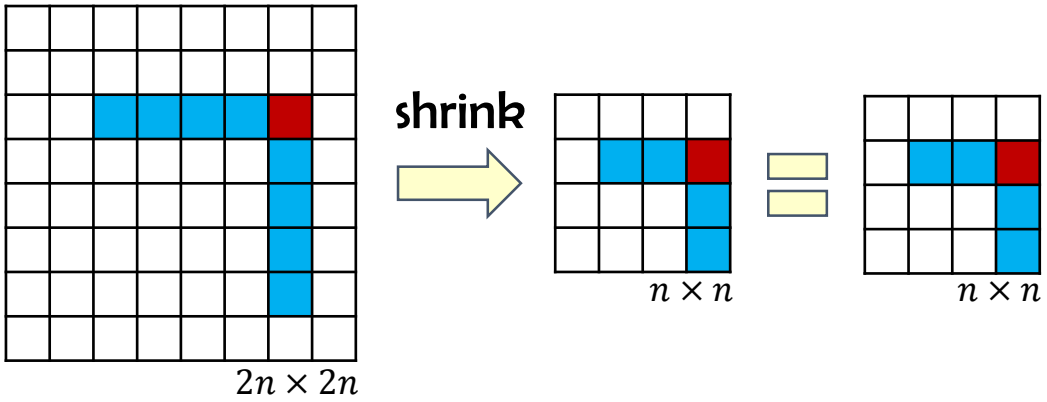




Autogen works on Fractal-DP class of DP problems

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:

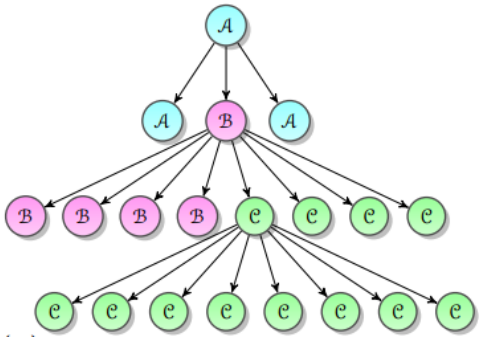
- 75% of 16 DP problems in CLRS book
- 60% of 40 DP problems in Lew and Mauch's book

Core idea: Find recursive patterns from DP table updates

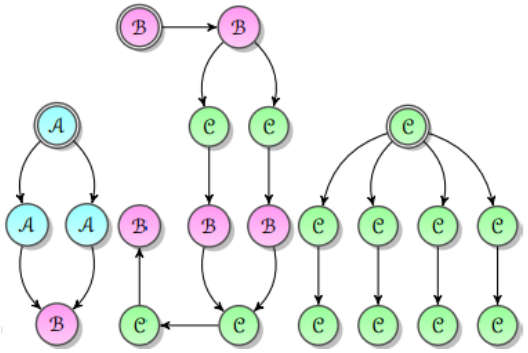
Generate DP table accesses for a small DP table. Find **recursive patterns in DP table accesses.** Build a recursive algorithm around these recursive access patterns.

$\langle(1,3),(1,1),(1,3)\rangle$
$\langle(1,3),(1,2),(2,3)\rangle$
$\langle(1,3),(1,3),(3,3)\rangle$
$\langle(2,4),(2,2),(2,4)\rangle$
..
..
..
$\langle(1,64),(1,64),(64,64)\rangle$

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. $C[i,j] \leftarrow \min \begin{pmatrix} C[i,k]+C[k,j] \\ +w(i,k,j), C[i,j] \end{pmatrix}$

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 $\langle (i,j),(i,k),(k,j) \rangle$

$\langle (1,3),(1,1),(1,3) \rangle$
$\langle (1,3),(1,2),(2,3) \rangle$
$\langle (1,3),(1,3),(3,3) \rangle$
$\langle (2,4),(2,2),(2,4) \rangle$
..
..
..
$\langle (1,64),(1,64),(64,64) \rangle$

Cell-set

Step 1. Generate a set of DP table updates

Small problem, say, $n = 64$

Generate cell-dependencies in the form below

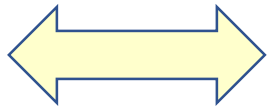
$\langle \text{writecell}, \text{readcell}_1, \ldots, \text{readcell}_s \rangle$

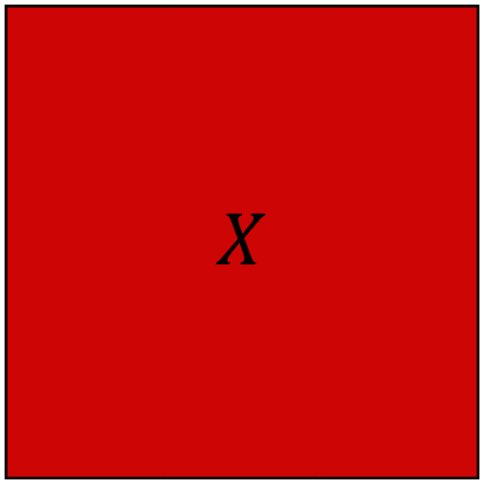
Cell-set = set of all cell dependencies

Step 2. Construct an algorithm-tree

$A(\langle X, X, X \rangle)$

$\langle(1,3), (1,1), (1,3)\rangle$
$\langle(1,3), (1,2), (2,3)\rangle$
$\langle(1,3), (1,3), (3,3)\rangle$
$\langle(2,4), (2,2), (2,4)\rangle$
..
..
..
$\langle(1,64), (1,64), (64,64)\rangle$



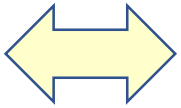


Step 2. Distribute the DP table updates into buckets

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

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

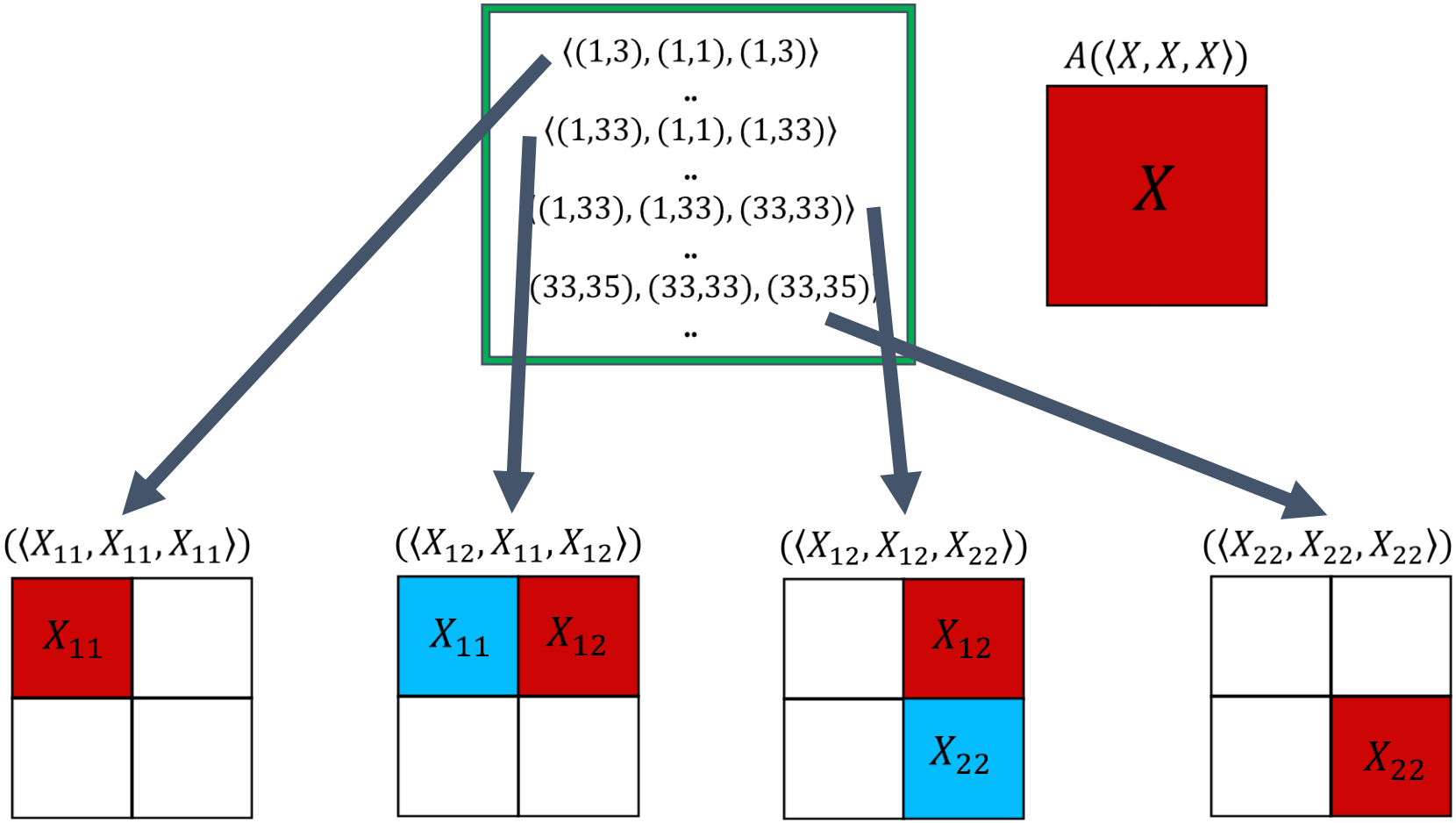
$\langle (1,3), (1,1), (1,3) \rangle$
$\langle (1,3), (1,2), (2,3) \rangle$
$\langle (1,3), (1,3), (3,3) \rangle$
$\langle (2,4), (2,2), (2,4) \rangle$
..
..
..
$\langle (1,64), (1,64), (64,64) \rangle$



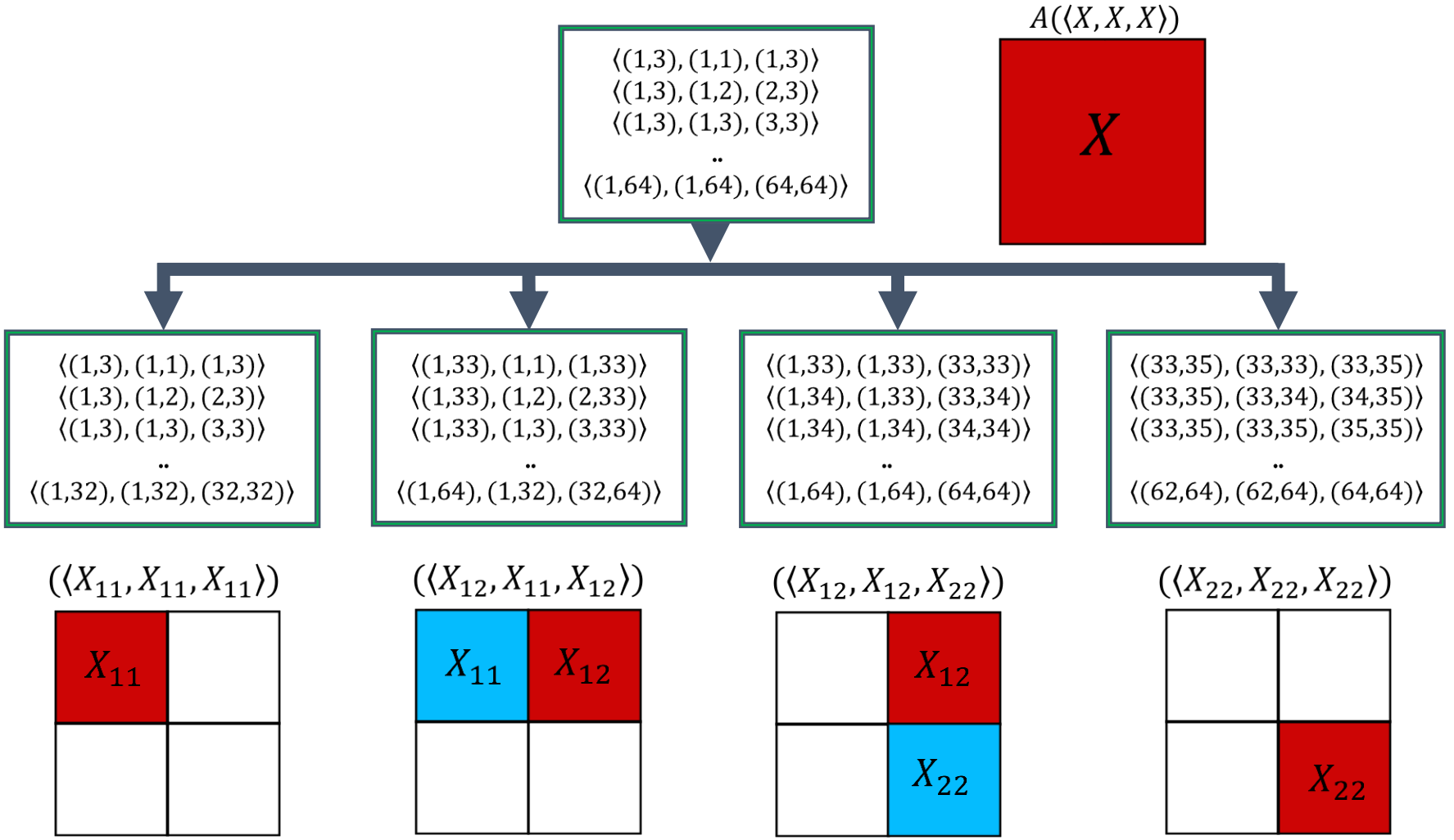
$(\langle X_{11}, X_{11}, X_{11} \rangle)$
$(\langle X_{11}, X_{11}, X_{11} \rangle)$
$(\langle X_{11}, X_{11}, X_{11} \rangle)$
$(\langle X_{11}, X_{11}, X_{11} \rangle)$
..
..
..
$(\langle X_{12}, X_{12}, X_{22} \rangle)$

X_{11}	X_{12}
X_{21}	X_{22}

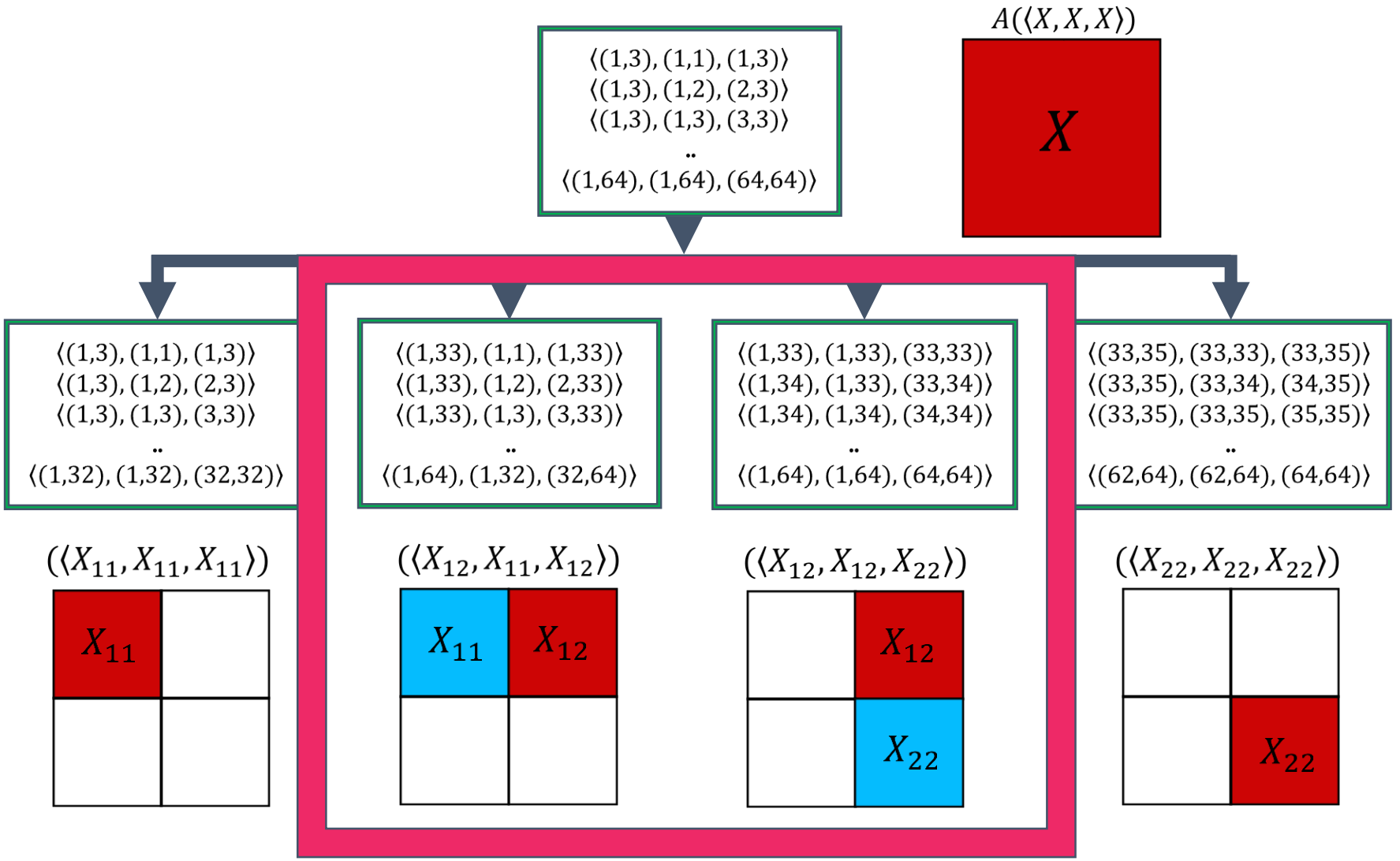
Step 2. Distribute the DP table updates into buckets



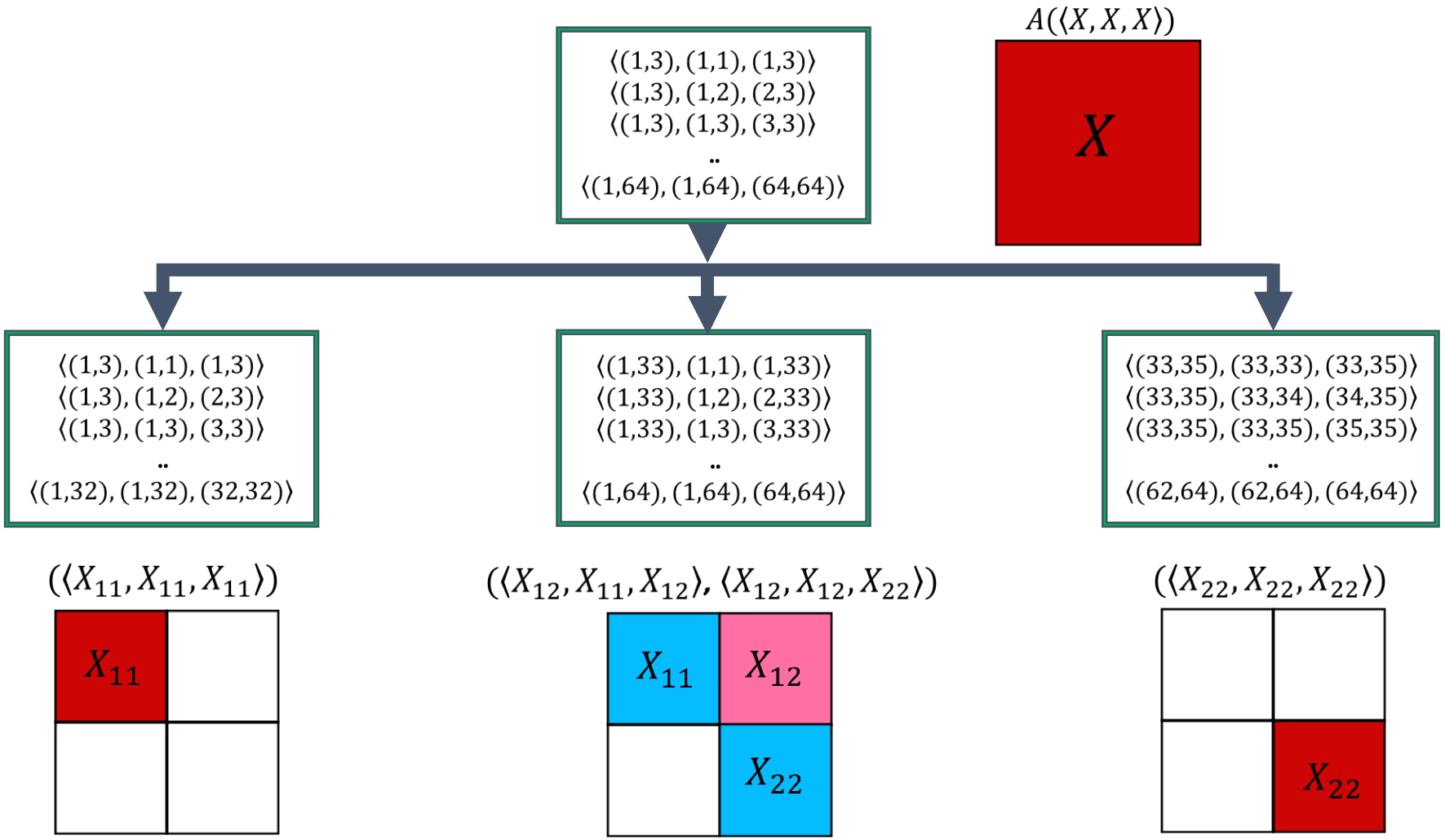
Step 2. Combine buckets that write-to and read-from the same regions



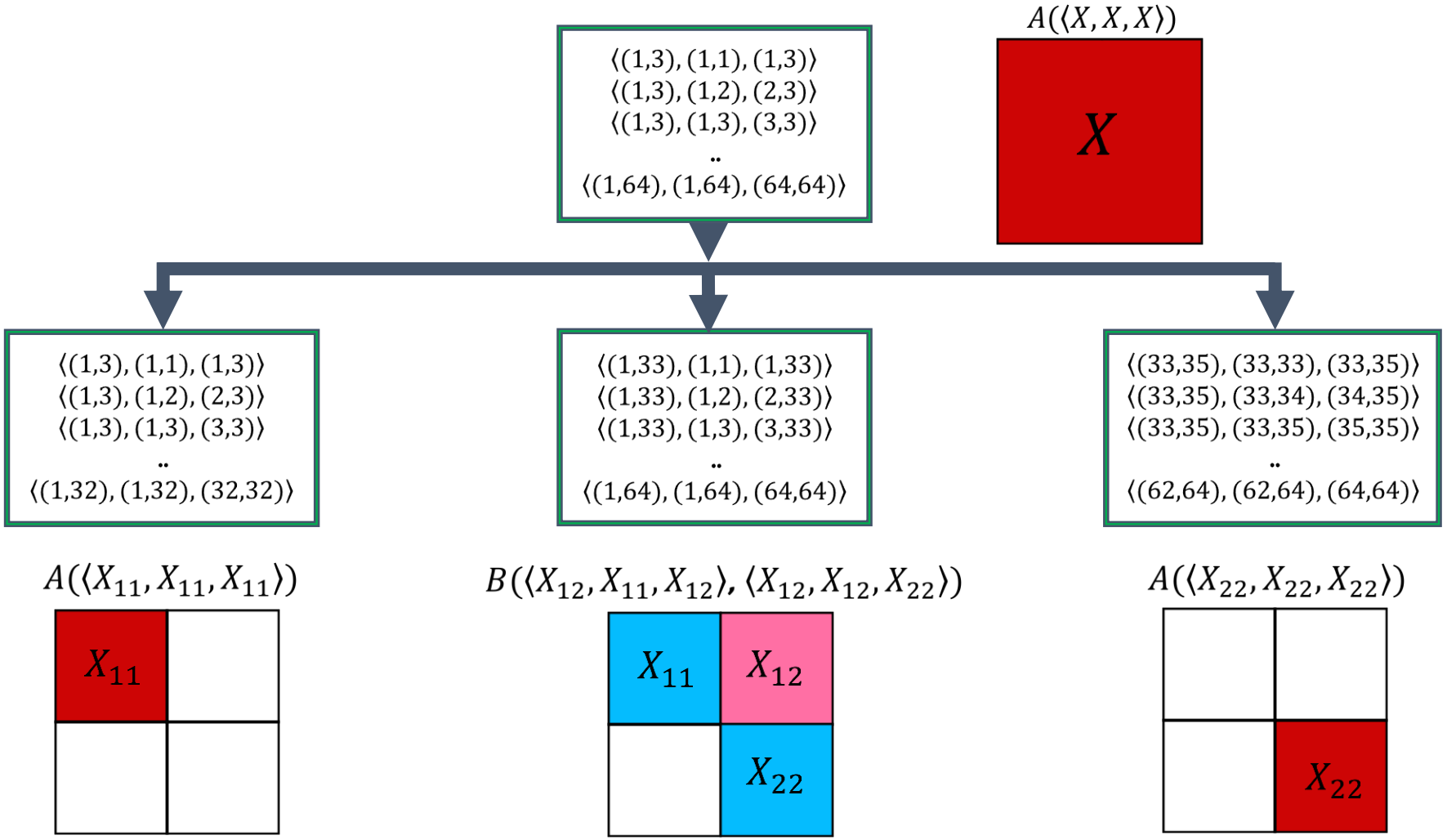
Step 2. Combine buckets that write-to and read-from the same regions



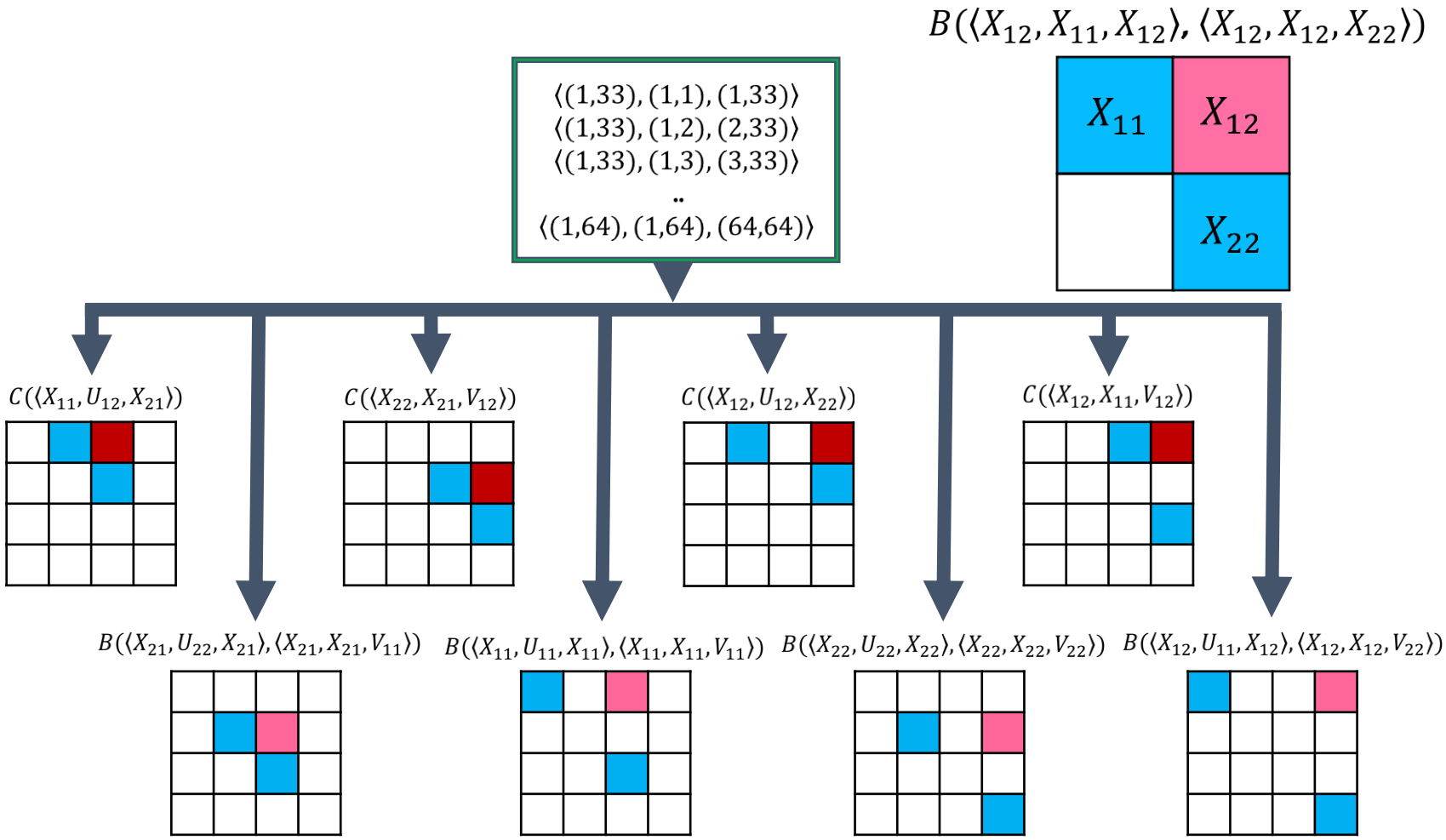
Step 2. After combining buckets



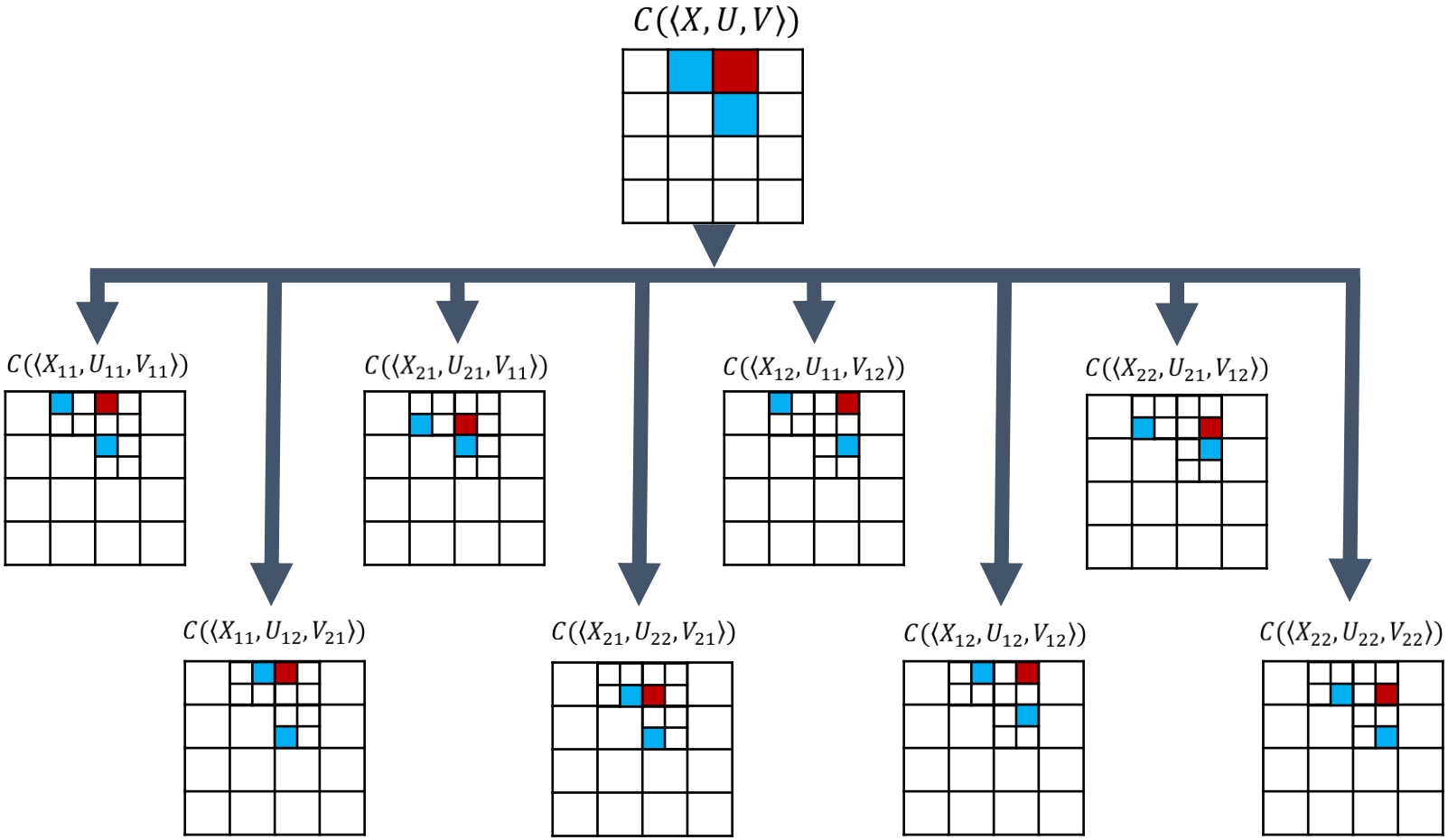
Step 3. Give function names to nodes of the algorithm-tree



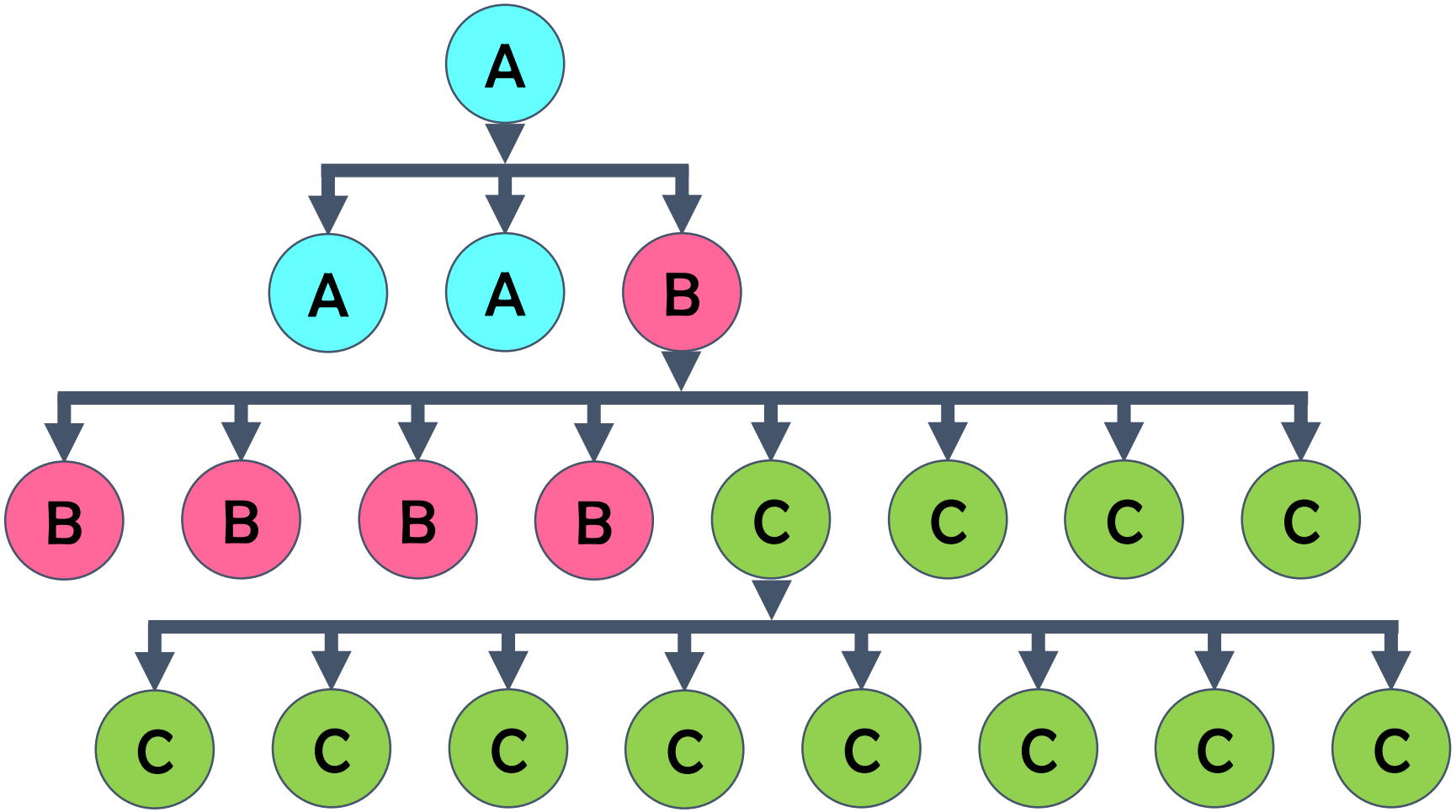
Steps 2&3. Recursively expand new functions



Steps 2&3. Recursively expand new functions



Steps 2&3. Labeled algorithm-tree



Step 4. Construct algorithm-DAG

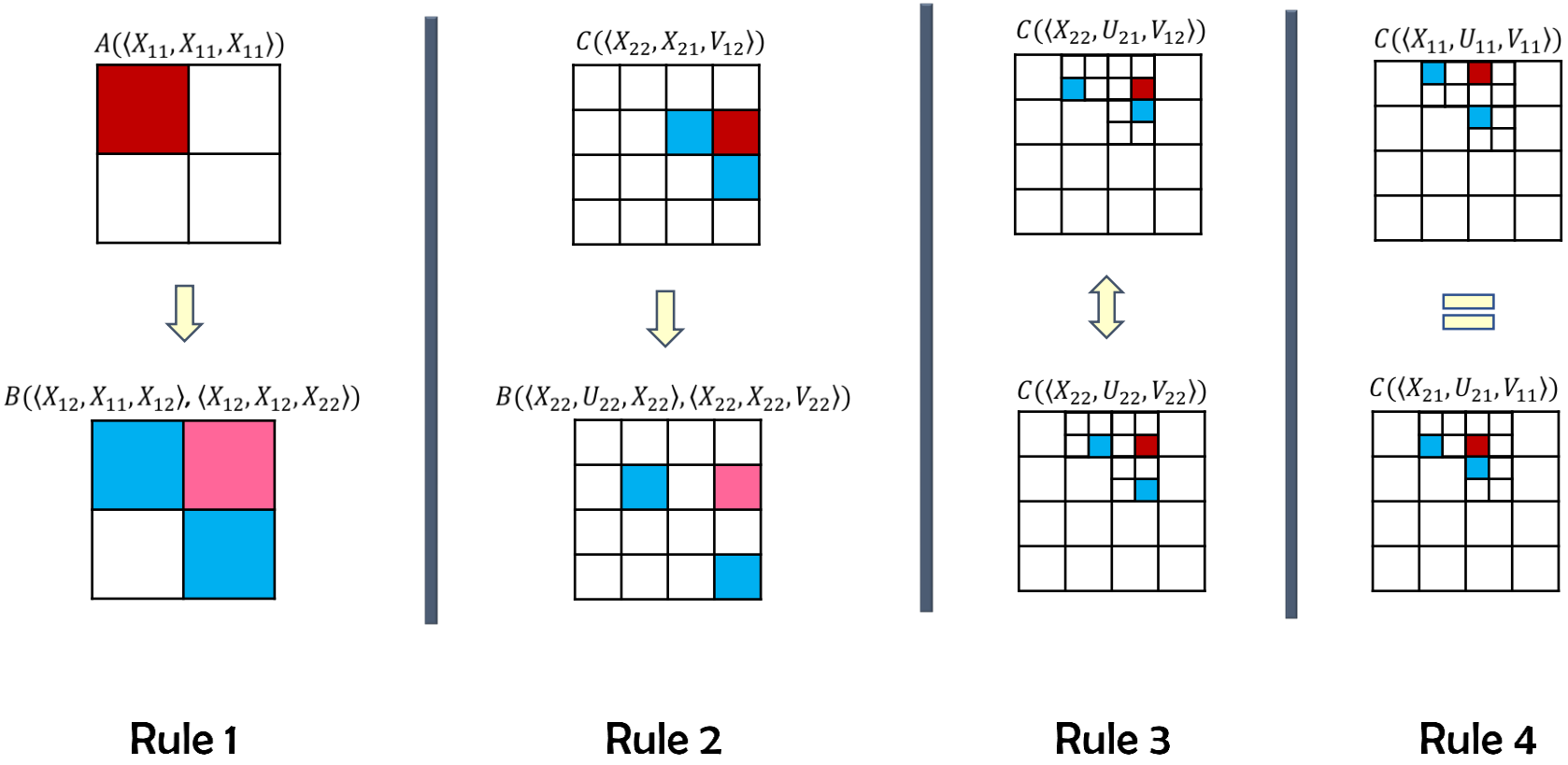
Limitations with algorithm-tree:

- Order of execution
- Parallelism

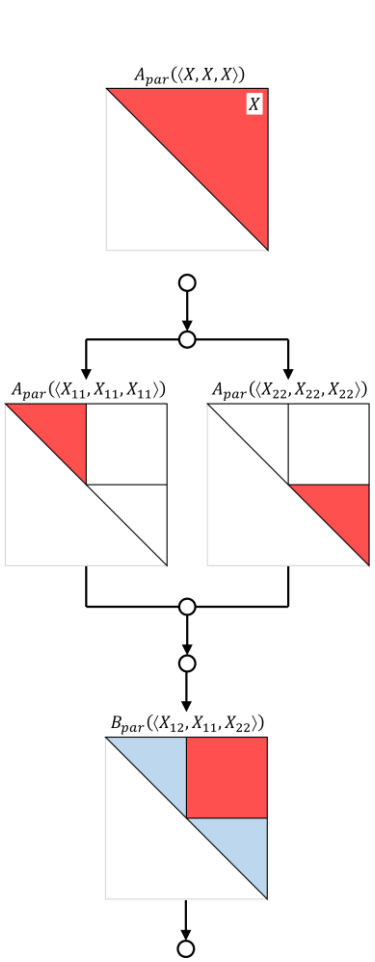
Solution: Algorithm-DAGs for each function

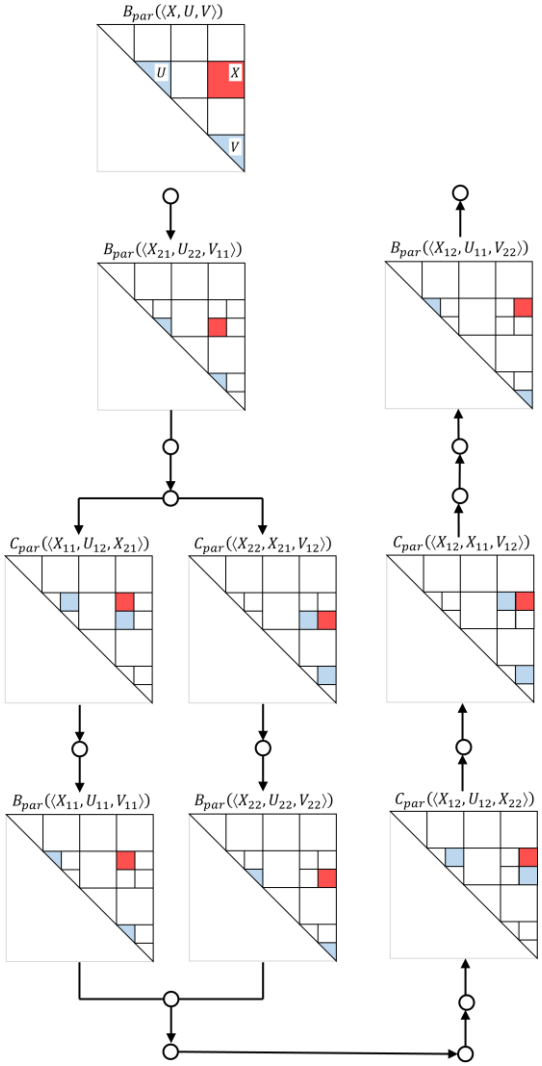
Four rules to construct algorithm-DAGs

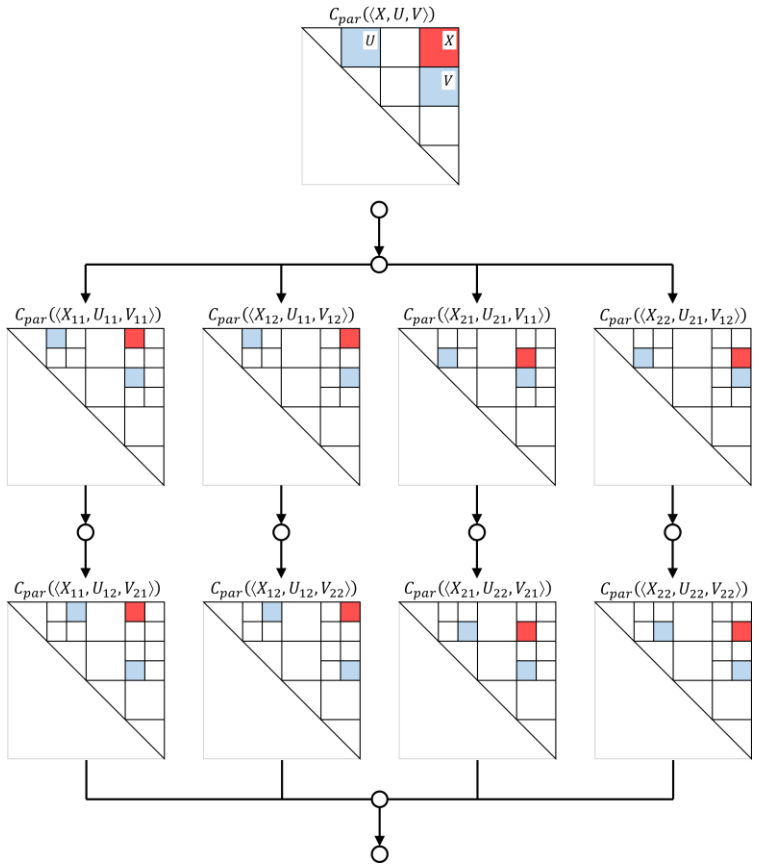
Step 4. Four rules to construct algorithm-DAGs



Step 4. Algorithm-DAGs for different recursive functions







Step 4. The DAGs represent an R-DP

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_{11}, 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})

I-DP -> Autogen -> 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. $C[i,j] \leftarrow \min \begin{pmatrix} C[i,k]+C[k,j] \\ +w(i,k,j), C[i,j] \end{pmatrix}$

I-DP

Autogen

R-DP

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_{11}, 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})

Performance analysis

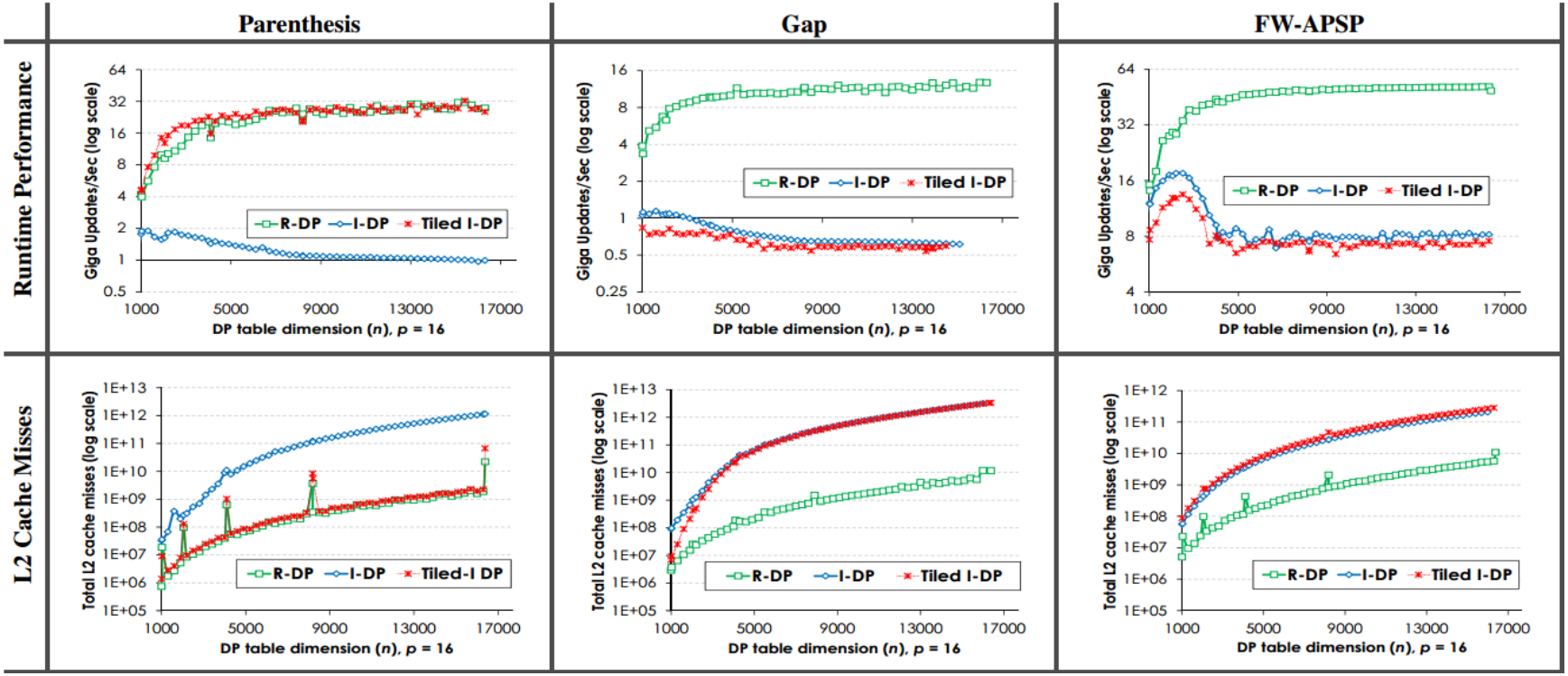
Theory	Parallelism	Work-span model
	Cache locality	Cache-oblivious model
Practice	Processor	Intel Sandy Bridge
	#Cores	16
	L1, L2, L3, RAM	32KB, 256KB, 20MB, 32GB
	Compiler	Intel C++ Compiler
	Parallelization	Intel Cilk Plus
	Count cache misses	PAPI
	I-DP	Parallel + nontrivial opt.
	Tiled I-DP	Pluto-generated + nontrivial opt.
	R-DP	Simple impl. + trivial opt.

R-DPs are efficient in theory

Problem	I-DP		R-DP	
	Cache comp.	Parallelism	Cache comp.	Parallelism
Parenthesis problem	$\Theta(n^3)$	$\Theta(n)$	$\Theta\left(n^3/\left(B\sqrt{M}\right)\right)$	$\Theta\left(n^{3-\log 3}\right)$
Gap problem				
Floyd-Warshall's APSP	$\Theta(n^3/B)$	$\Theta(n^2/\log n)$		$\Theta(n^2/\log^2 n)$
Protein folding		$\Theta(n)$		$\Theta(n^2/\log n)$
Function approx.				$\Theta\left(n^{3-\log 3}\right)$
Matrix multiplication*		$\Theta(n^2)$		$\Theta(n^2)$
Multi-instance Viterbi	$\Theta(n^3t/B)$		$\Theta\left(n^3t/\left(B\sqrt{M}\right)\right)$	
LCS / edit distance	$\Theta(n^2/B)$	$\Theta(n)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n^{2-\log 3}\right)$
Spoken word recog.				
Binomial coefficient				
Bitonic TSP				
Bubble sort*		$\Theta(1)$		$\Theta(n)$
Selection sort*				
Insertion sort*				$\Theta\left(n^{2-\log 3}\right)$

B = block size, M = cache size, $*$ = non-DP problems

R-DPs are efficient in practice



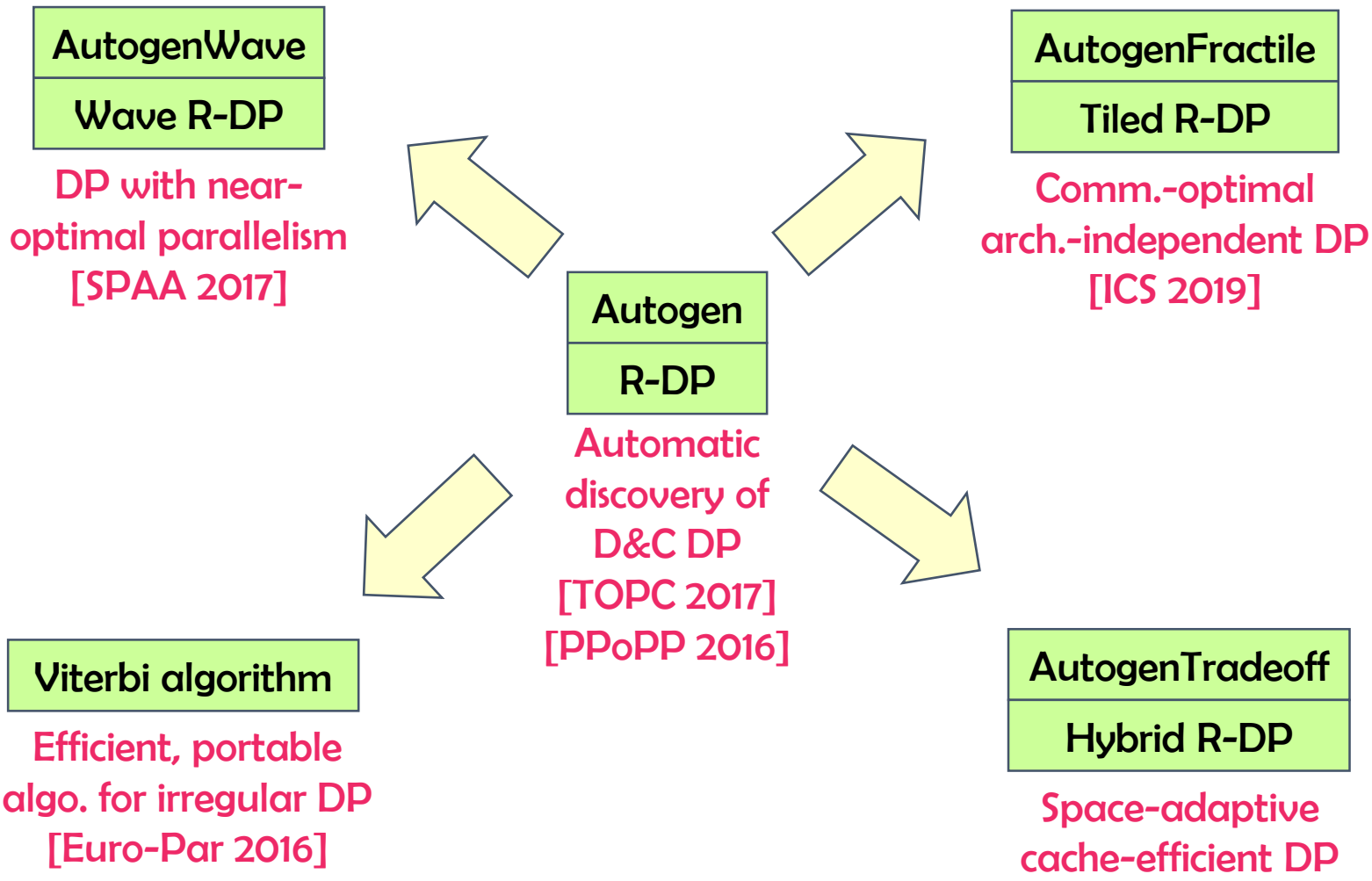
R-DPs are efficient in practice

Problem	Speedup
Parenthesis	18x
Gap	17x
Floyd-Warshall's APSP	6x

$n = 8192$ and $p = 16$

speedup w.r.t. highly parallel and optimized I-DP

Overview of our work



Problem-solving technique