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

Pramod Ganapathi

Adviser: Rezaul Chowdhury

Stony Brook University

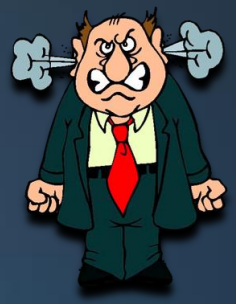
Contents

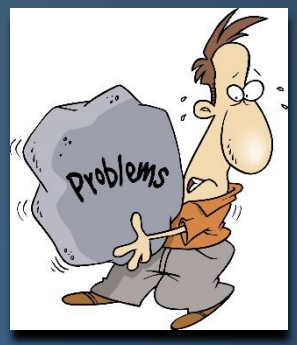
- Vision

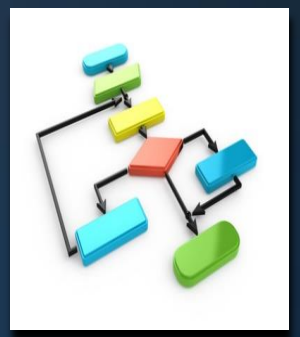
1. Autogen algorithm [PPoPP 2016, Invited to TOPC journal]
2. Autogen-Wave framework [Under review]
3. Efficient Viterbi algorithm [Euro-Par 2016]
4. Autogen-Fractile framework [Under preparation]

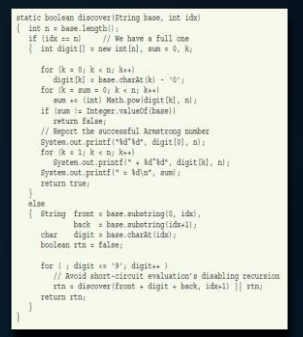
Vision

All life is problem solving









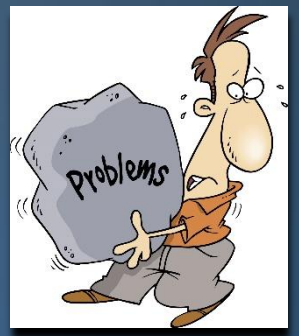
```
static boolean discover(String base, int idx)
{  int n = base.length();
   if (idx == n)      // We have a full one
   {  int digit[] = new int[n], sum = 0, k;

      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;
   }
}
```

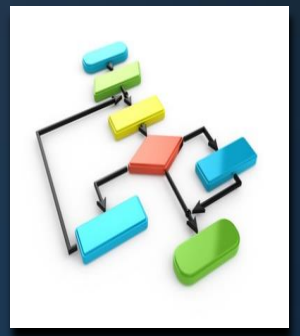
Problem specification

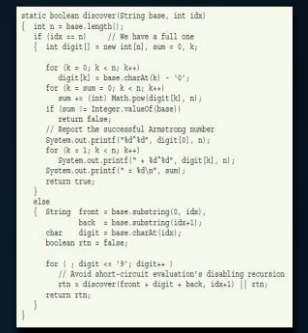
Efficient algorithm

High-performing implementation

Computers can generate codes





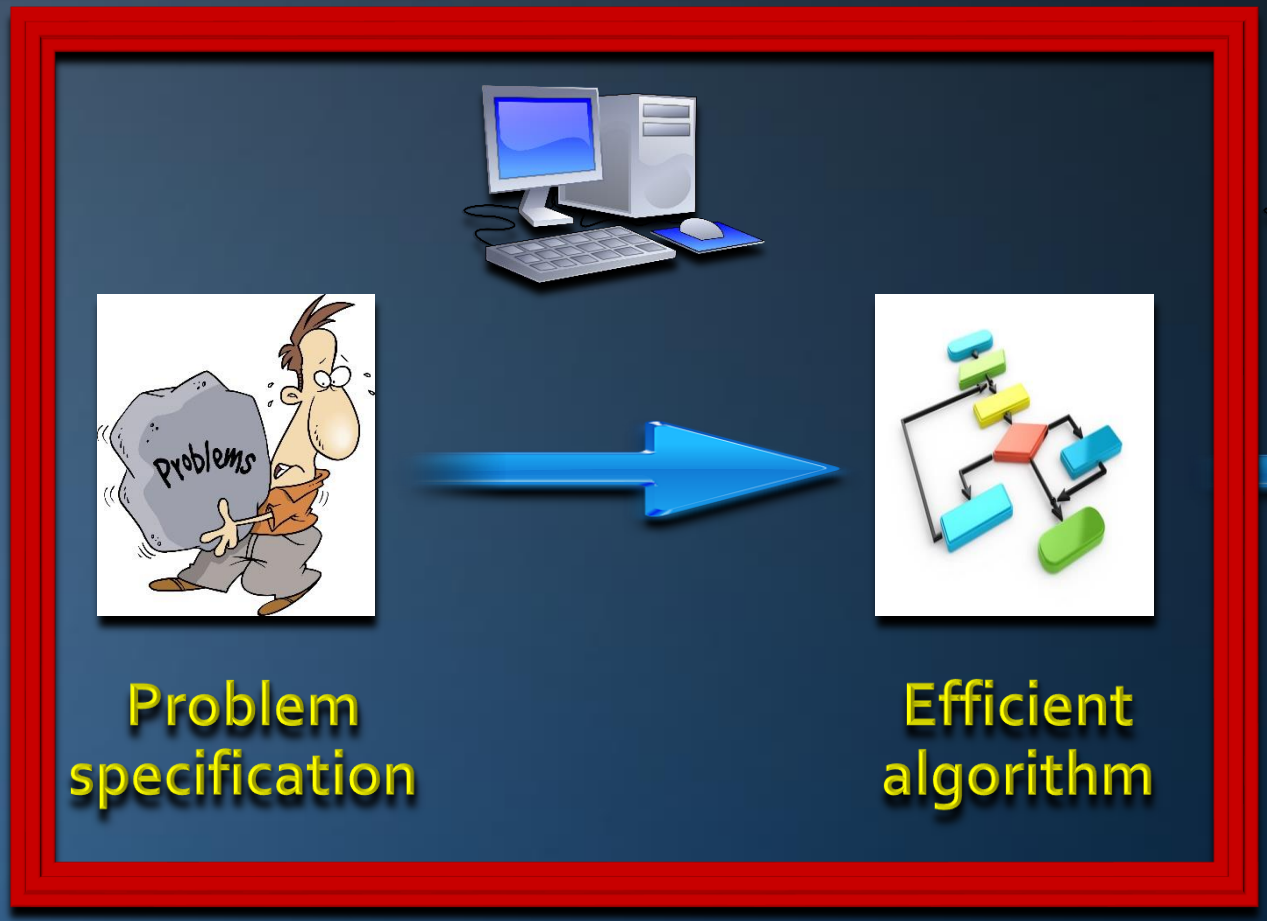


Problem specification

Efficient algorithm

High-performing implementation

Computers can discover nontrivial algorithms



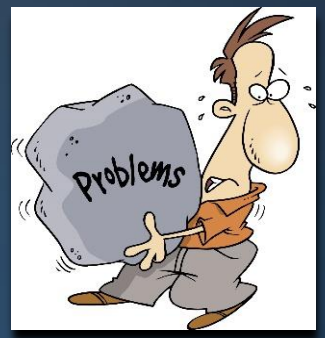
Problem specification

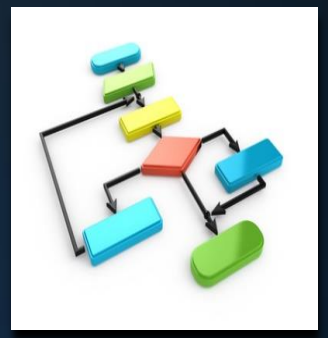
Efficient algorithm

```
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;
   }
}
```

High-performing implementation

Thesis core : Computers can discover efficient algorithms for dynamic programming (DP) problems

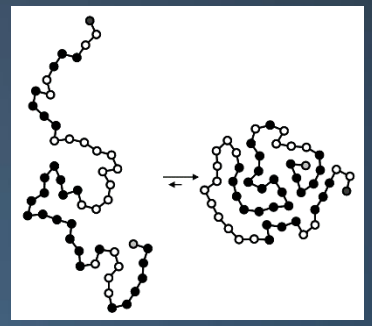




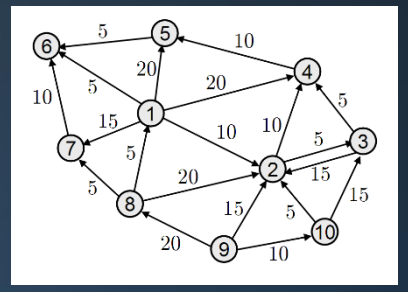
Implementation of a DP recurrence

Efficient DP algorithm

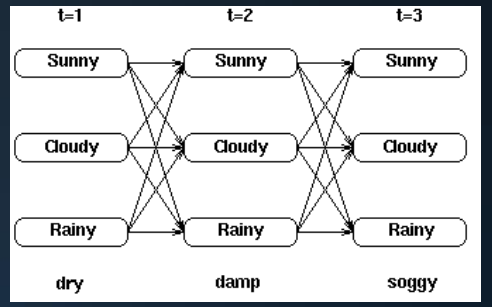
Dynamic programs are important



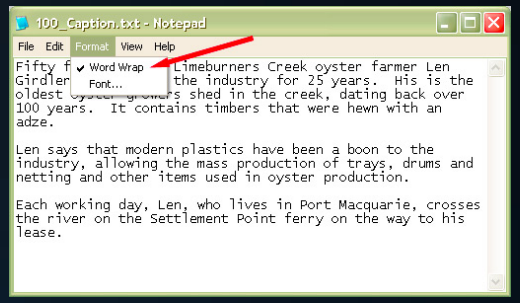
RNA / protein folding



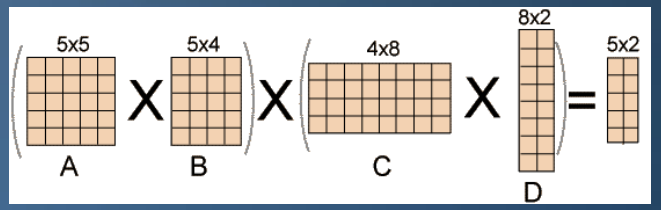
All-pairs shortest path



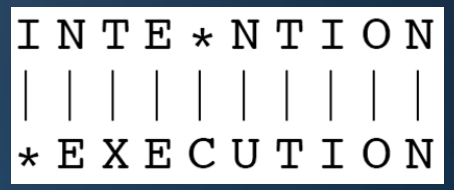
Viterbi algorithm



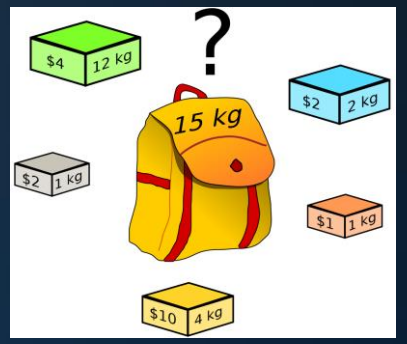
Word wrapping



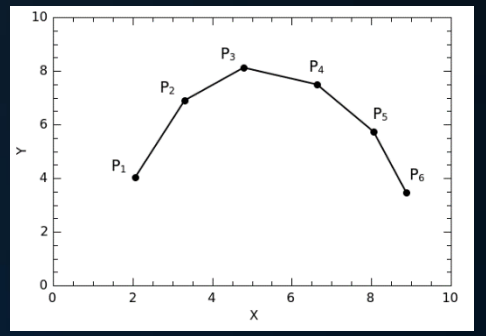
Chain matrix multiplication



Edit distance



Knapsack problem



Function approximation

Divide-&-conquer 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	NA		
Higher parallelism			*
Highly optimizable kernels	NA		
Energy efficiency			
Bandwidth efficiency			

*Typically for complicated non-local dependencies

Parallelism and locality are the key factors to improve performance

- **Parallelism**
  Multiple tasks running in parallel
- **Cache locality**
  - **Spatial locality:** Bring as much useful data as possible to cache
  - **Temporal locality:** As much useful work as possible is done to data in cache
  - Pack useful data (spatial locality) and access localized data (temporal locality)

Currently, D&C algorithms are designed by experts manually

Sequence alignment

Parenthesis problem

Gap problem

Floyd-Warshall's APSP

Multi-instance Viterbi

Spoken word recog.

Function approx.

I-DP

I-DP

I-DP

I-DP

I-DP

I-DP

I-DP

R-DP

R-DP

R-DP

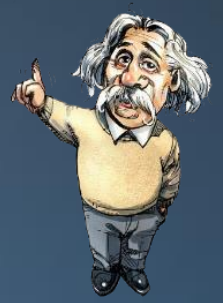
R-DP

R-DP

R-DP

R-DP

Also, researchers from other fields face difficulty in designing D&C algorithms



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 fast and portable DP algorithms

Any blackbox implementation of a DP recurrence

Efficient, portable, & robust algorithm

Algorithm / Framework

1. Autogen algorithm

[PPoPP 2016, Invited to TOPC journal (top 5%)]

We use parenthesis problem as an example

- Input: A string $s_1 s_2 \dots s_n$
- $C[i,j] =$ cost of parenthesization from s_i to s_j.
- $$C[i,j] = \begin{cases} \infty & \text{if } 0 \le i = j \le n \\ v_j & \text{if } 0 \le i = j-1 < n \\ \min\limits_{k \in [i,j]} \{C[i,k] + C[k,j] + w(i,k,j)\} & \text{if } 0 \le i < j-1 < n \end{cases}$$
- Output: $C[1,n]$

Our goal is to discover R-DP from I-DP

```
Loop-Parenthesis(C, n)
1.  for i ← n − 1 to 0 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] )
```

I-DP

Autogen

R-DP

```
A(X)
1.  if X is a small matrix then A-loop(X)
2.  else
3.    par: A(X11), A(X22)
4.    B(X12, X11, X22)
```

```
B(X, U, V)
1.  if X is a small matrix then B-loop(X, U, V)
2.  else
3.    B(X21, U22, V11)
4.    par: C(X11, U12, V21), C(X22, X21, V12)
5.    par: B(X11, U11, V11), B(X22, X22, V22)
6.    C(X12, U12, X22)
7.    C(X12, X11, V12)
8.    B(X12, U11, V22)
```

```
C(X, U, V)
1.  if X is a small matrix then C-loop(X, U, V)
2.  else
3.    par: C(X11, U11, V11), C(X12, U11, V12)
           C(X21, U21, V11), C(X22, U21, V12)
4.    par: C(X11, U12, V21), C(X12, U12, V22)
           C(X21, U22, V21), C(X22, U22, V22)
```

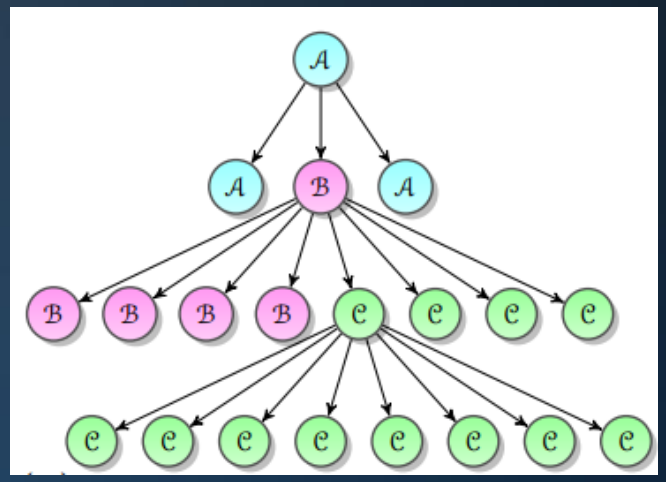
Autogen's core idea is to find recursive patterns from DP table updates

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.

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

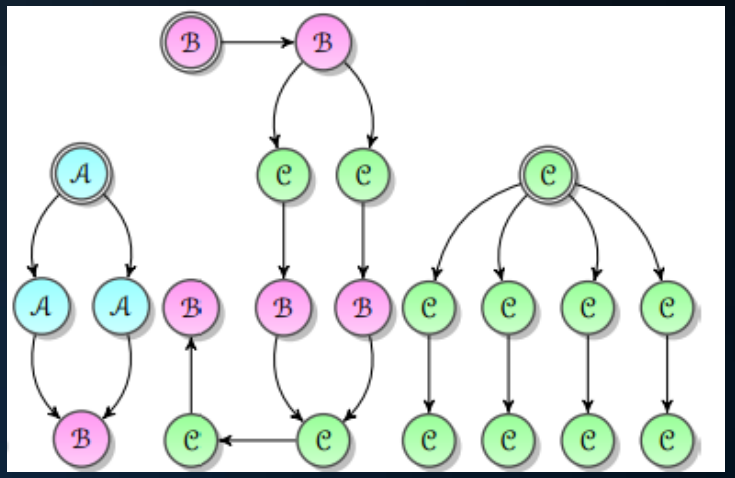
$\langle(1,64),(1,64),(64,64)\rangle$

1. Cell-set generation



2. Algorithm-tree construction

3. Algorithm-tree labeling



4. Algorithm-DAG construction

R-DPs are efficient in theory

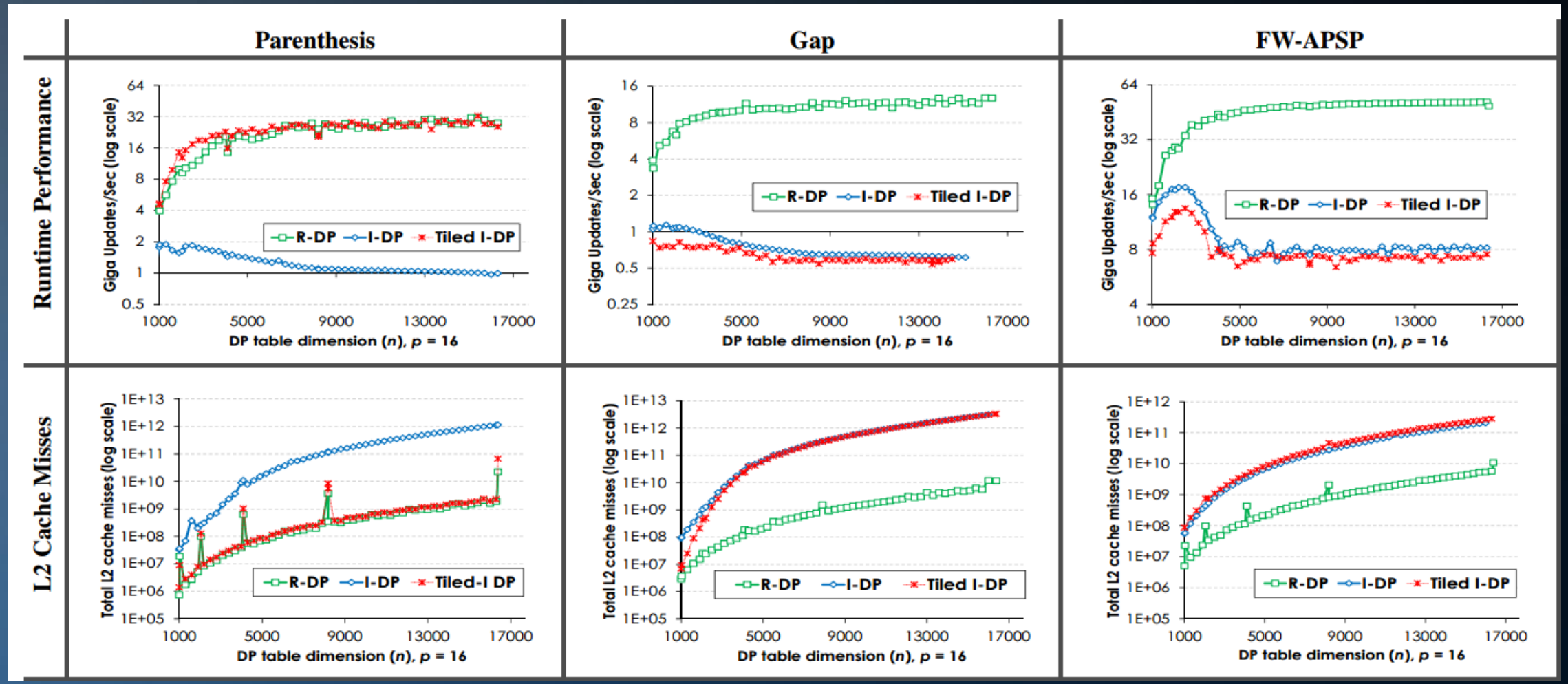
B = block size

M = cache size

$*$ = non-DP problems

Problem	Iterative DP		Recursive D&C DP	
	Serial cache complexity	Parallelism	Serial cache complexity	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 approximation				$\Theta\left(n^{3-\log 3}\right)$
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 recognition				
Binomial coefficient				
Bitonic TSP				
Bubble sort*		$\Theta(1)$		$\Theta(n)$
Selection sort*				
Insertion sort*				$\Theta\left(n^{2-\log 3}\right)$

R-DPs are efficient in practice



2. Autogen-Wave

[Under review]

R-DPs are great! But, we still can increase their parallelism

Feature	Iterative Wave	Tiled Wave	Recursive DP	Recursive Wave DP
Cache-efficiency				
Cache-obliviousness				
Cache-adaptivity	NA			
Higher parallelism			*	
Highly optimizable kernels	NA			
Energy efficiency				
Bandwidth efficiency				

* Typically for complicated non-local dependencies

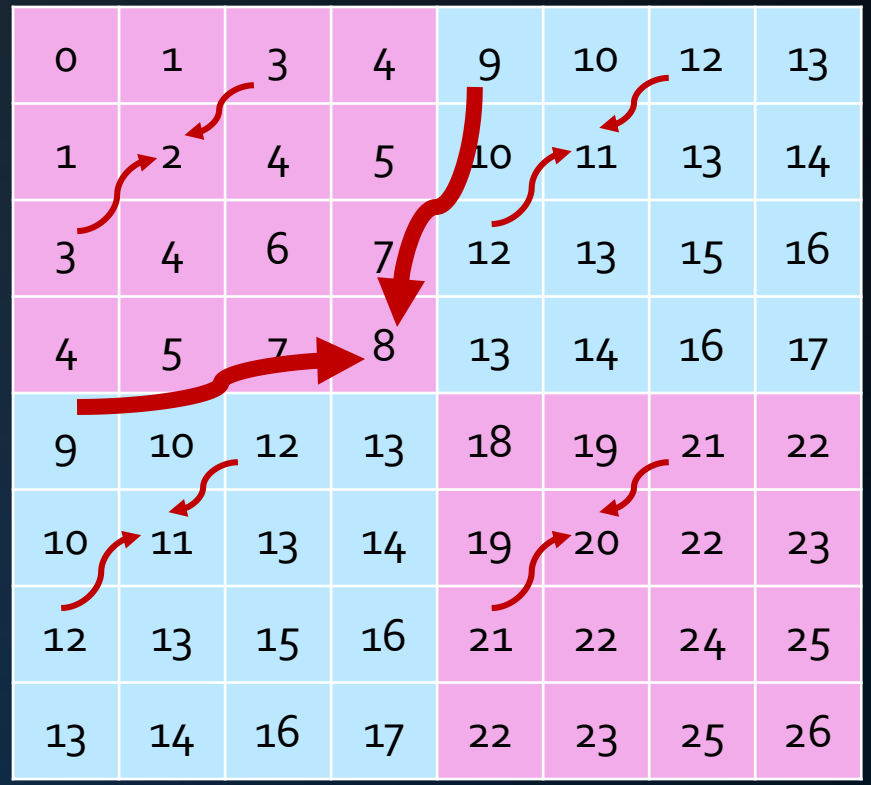
Consider the LCS DP recurrence

- Length of the longest common subsequence
- Input: A string $R[1..n]$ and $S[1..n]$
- $C[i,j] =$ length of LCS of $R[1..i]$ and $S[1..i]$
- $$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } i,j > 0 \text{ or } r_i = s_j \\ \min(C[i, j-1], C[i-1, j]) & \text{if } i,j > 0 \text{ or } r_i \neq s_j \end{cases}$$
- Output: $C[n,n]$

R-DPs are slower due to false dependencies

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14

Wave I-DP span $= \theta(n)$



R-DP span $= \theta\left(n^{\log 3}\right)$

Our goal (discover WR-DP from R-DP)

```
A(X)
1.   if X is a small matrix then A-loop(X)
2.   else
3.     A(X_11)
4.     parallel: A(X_12), A(X_21)
5.     A(X_22)
```

R-DP

Autogen-Wave

WR-DP

```
WR-DP (X)
1.   w ← 0
2.   while w < ∞ do
3.     A(X, w)
```

```
A (X, w)
1.   v_i ← 0 for i ∈ [1, 4]

2.   if X is an n' × n' matrix then
3.     if w = S_A(X) then A-chunk(X)
4.   else
5.     F_i ← {A, A, A, A}
6.     arg_i ← {X_11, X_12, X_21, X_22}
7.     parallel for i ← 1 to 4 do
8.       if w < S_Fi(arg_i) then v_i ← S_Fi(arg_i)
9.       elseif w < E_Fi(arg_i) then v_i ← F_i(arg_i)
10.    sync

11.  return Min (v_i) for i ∈ [1, 4]
```

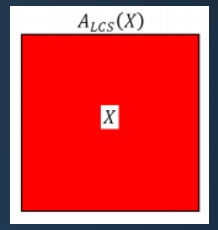
Autogen-Wave core idea: Use timing functions to execute R-DP functions as soon as they are ready

Compute mathematical formulas or algorithmic functions to find the earliest start (resp. end) time at which different R-DP functions can start (resp. end) execution. The R-DP functions are executed at their respective start times. Different functions are then called based on increasing order of start-times.

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

$$C(i,j) = i + j$$

1. Completion-time function construction



$$S_A(x_r, x_c, n) = x_r + x_c$$

$$E_A(x_r, x_c, n) = x_r + x_c + 2n - 2$$

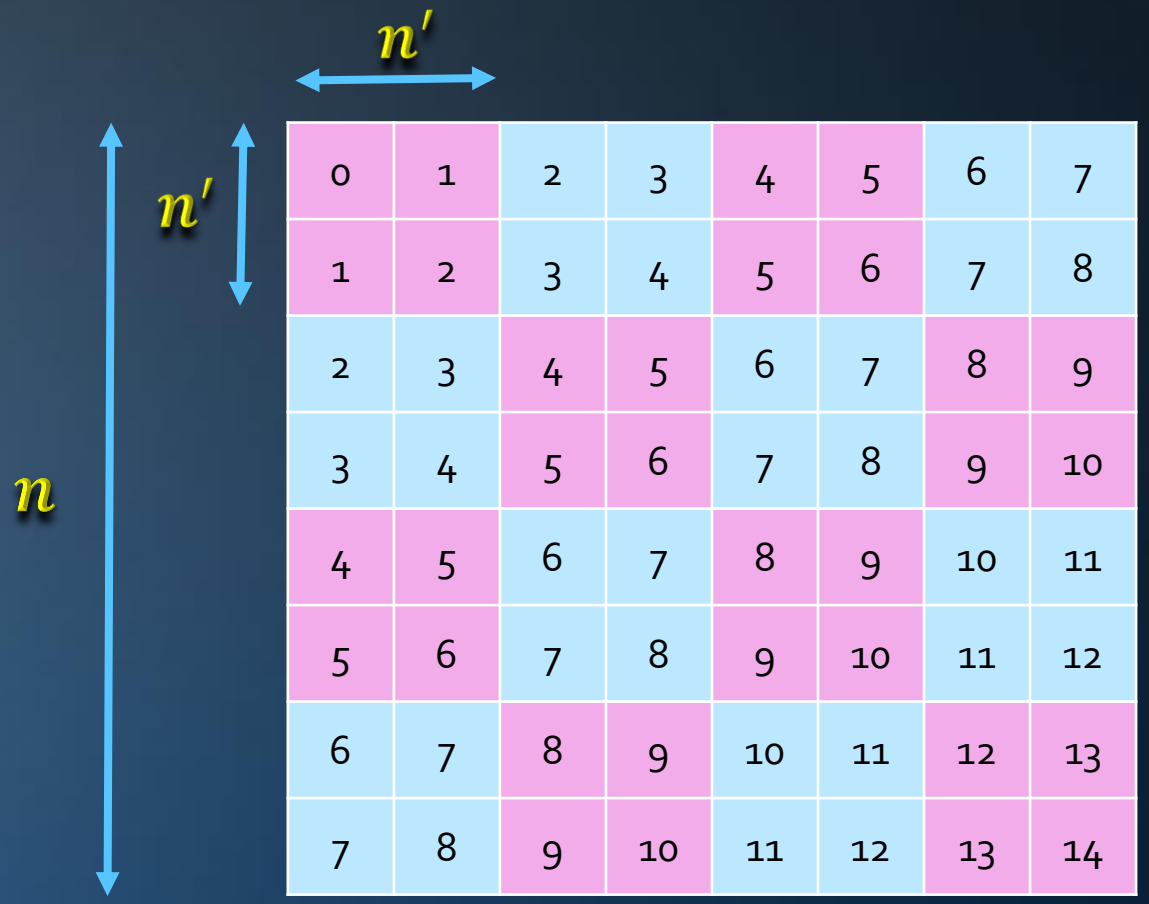
2. Start- and end-time function construction

```
WR-DP (X)
1.   w ← 0
2.   while w < ∞ do
3.     A(X, w)
```

```
A (X, w)
1.   v_i ← 0 for i ∈ [1, 4]

2.   if X is an n' × n' matrix then
3.     if w = S_A(X) then A-chunk(X)
4.   else
5.     F_i ← {A, A, A, A}
6.     arg_i ← {X_11, X_12, X_21, X_22}
7.     parallel for i ← 1 to 4 do
8.       if w < S_Fi(arg_i) then v_i ← S_Fi(arg_i)
9.       elseif w < E_Fi(arg_i) then v_i ← F_i(arg_i)
10.    sync

11.  return Min (v_i) for i ∈ [1, 4]
```

3. WR-DP algorithm derivation

WR-DP algorithm execution for LCS DP table

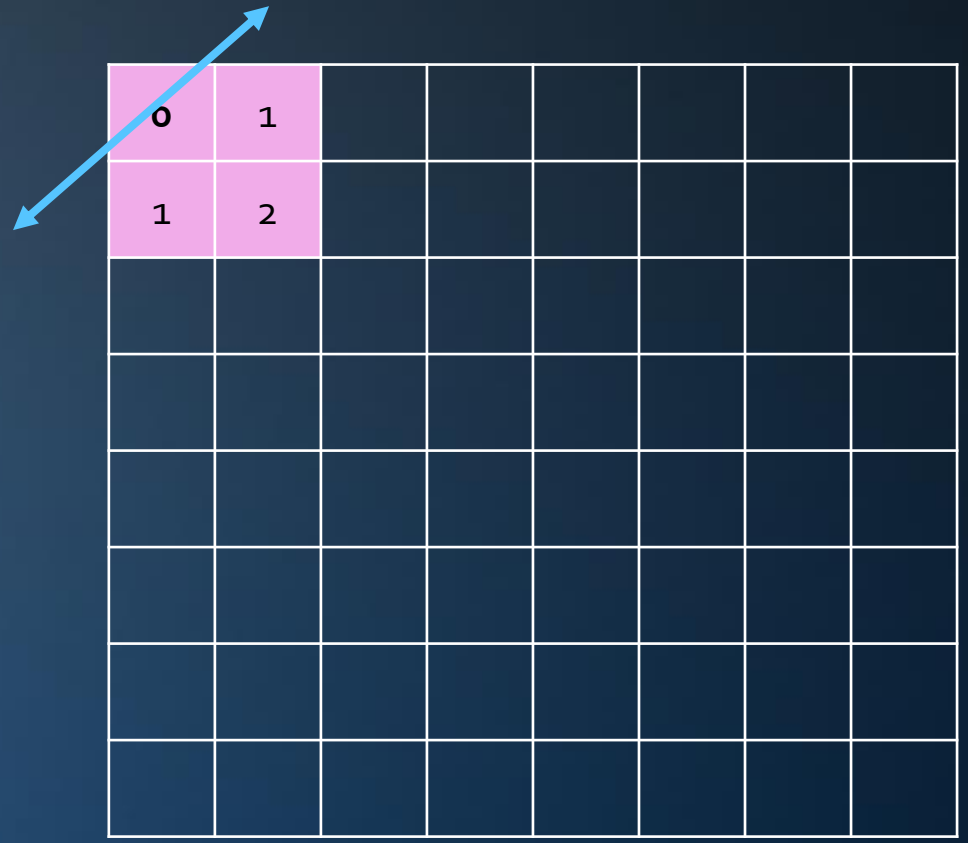


0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14

n' = base case parameter

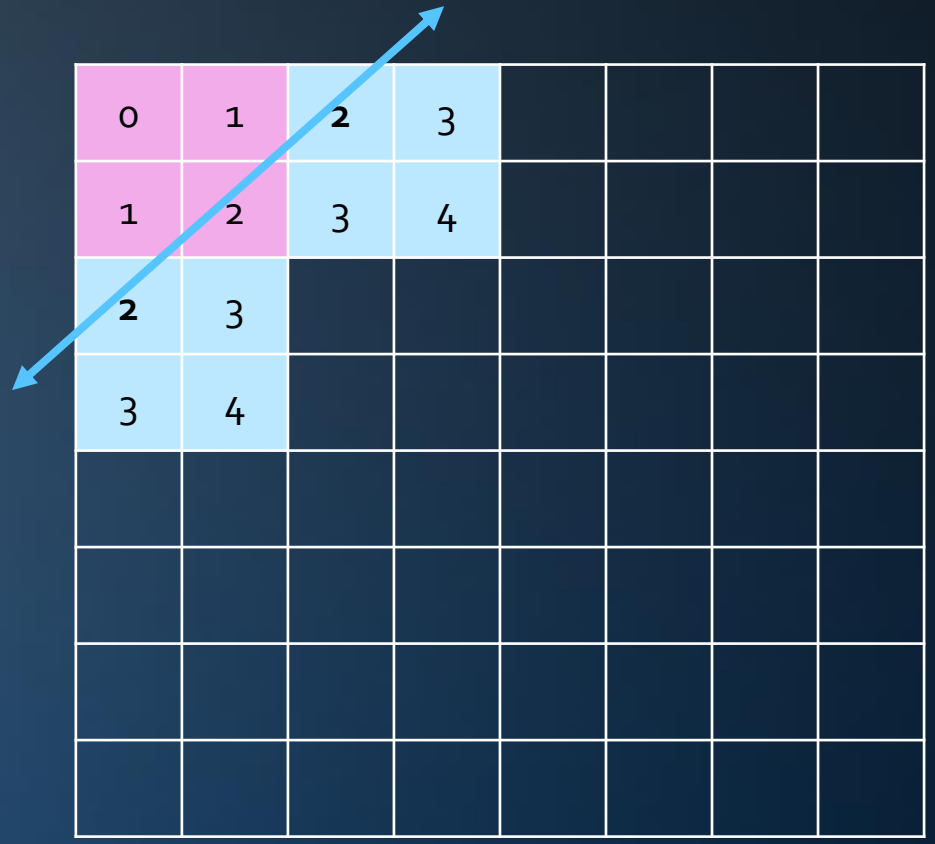
Fill all DP table chunks starting at level 0

Wavefront level $= 0$



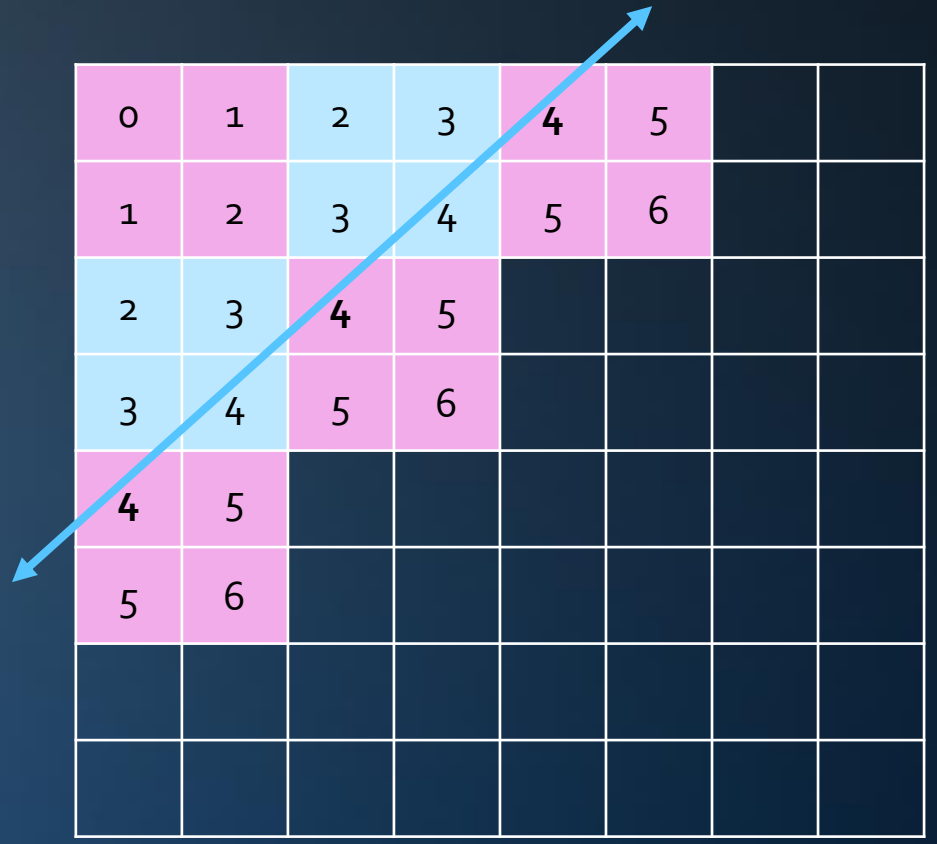
Fill all DP table chunks starting at level 2

Wavefront level $= 2$



Fill all DP table chunks starting at level 4

Wavefront level $= 4$



WR-DP algorithms are efficient++ in theory

	Work-stealing scheduler	Modified space-bounded scheduler
Span, $T_\infty(n)$	$\Theta\left(N_\infty\left(\frac{n}{n'}\right)(O(\log n)+T_\infty^R(n'))\right)$	$\Theta\left(N_\infty\left(\frac{n}{M^{1/k}}\right)(O(\log n)+T_\infty^R(M^{1/k}))\right)$
Parallel time, $T_p(n)$ w.r.t n	$O\left(\frac{T_1(n)}{p}+T_\infty(n)\right)$	$O\left(\frac{T_1(n)}{p}+T_\infty(n)\right)$
Serial cache complexity, $Q_1(n)$	$Q_1^R(n)$	$Q_1^R(n)$
Parallel cache complexity, $Q_p(n)$ w.r.t n	$O\left(Q_1(n)+p\frac{M}{B}T_\infty(n)\right)$	$O(Q_1(n))$
Extra space, $S_p(n)$	$O(p\log n)$	$O(p\log n)$
Assumptions	$T_\infty(n')=\Omega(\log n), n'\geq M^{1/k}$	$T_\infty(n')=\Omega(\log n)$

WR-DP algorithms are efficient++ in theory

B = block size

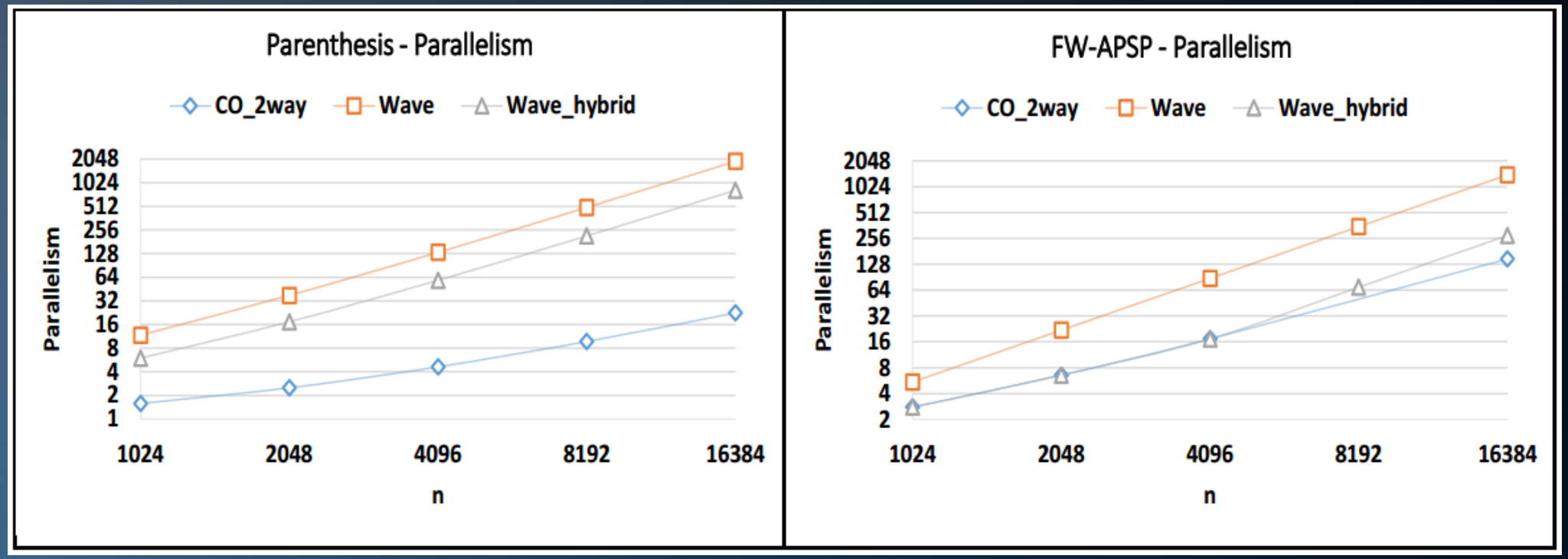
M = cache size

$*$ = non-DP problems

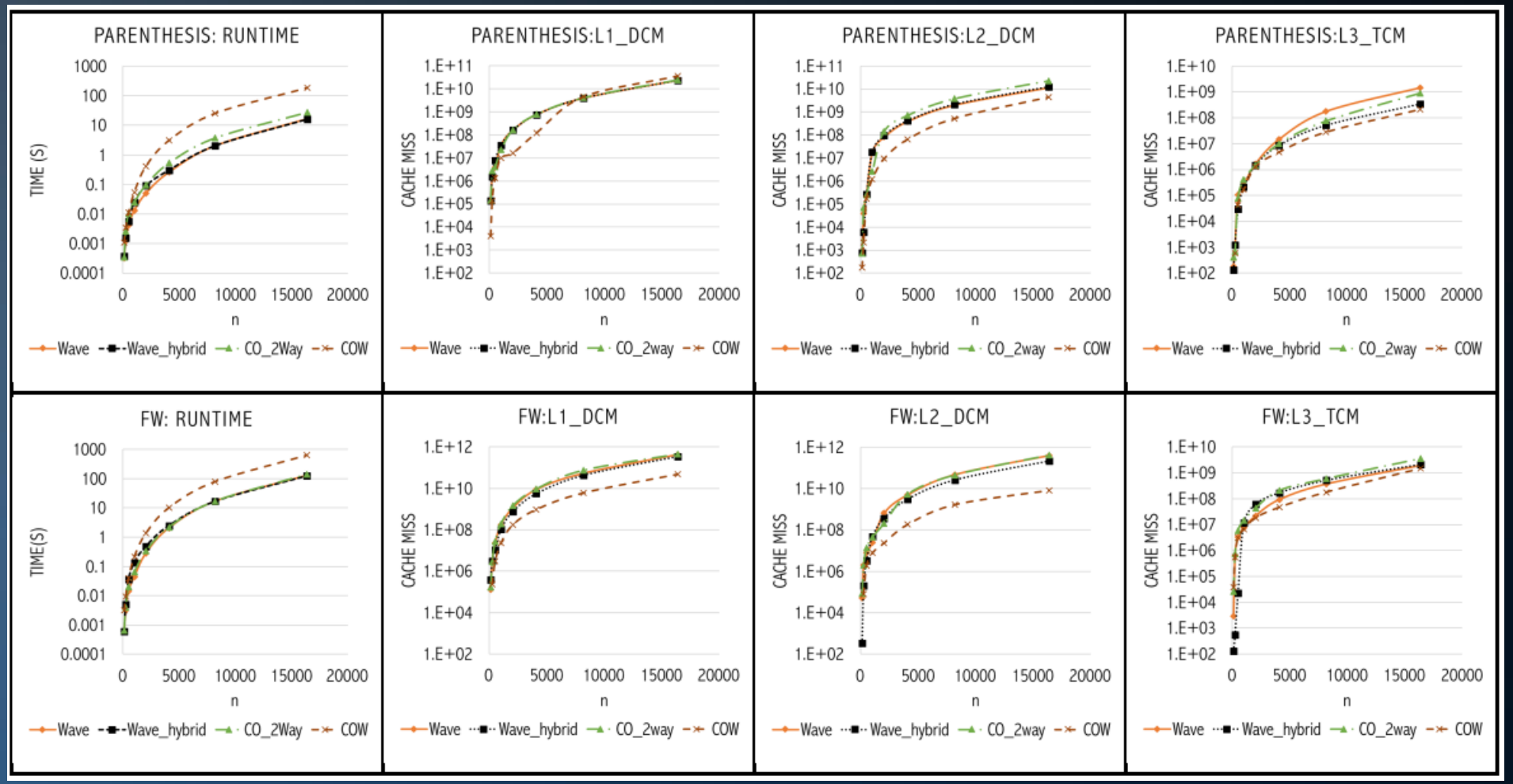
Using a modified space-bounded scheduler

Problem	R-DP		WR-DP	
	Serial cache comp.	Span	Serial cache comp.	Span
Parenthesis problem	$\Theta\left(n^3/\left(B\sqrt{M}\right)\right)$	$\Theta\left(n^{\log 3}\right)$	$\Theta\left(n^3/\left(B\sqrt{M}\right)\right)$	$\Theta\left(\frac{n}{\sqrt{M}}\left(\log n+\sqrt{M}^{\log 3}\right)\right)$
Gap problem				
Floyd-Warshall's APSP		$\Theta(n\log^2 n)$		$\Theta\left(\frac{n}{\sqrt{M}}\left(\log n+\sqrt{M}\log^2\sqrt{M}\right)\right)$
Protein folding		$\Theta(n\log n)$		$\Theta\left(\frac{n}{\sqrt{M}}\left(\log n+\sqrt{M}\log\sqrt{M}\right)\right)$
Function approximation		$\Theta\left(n^{\log 3}\right)$		$\Theta\left(\frac{n}{\sqrt{M}}\left(\log n+\sqrt{M}^{\log 3}\right)\right)$
Matrix multiplication*		$\Theta(n)$		$\Theta\left(\frac{n}{\sqrt{M}}\left(\log n+\sqrt{M}\right)\right)$
Multi-instance Viterbi	$\Theta\left(n^3t/\left(B\sqrt{M}\right)\right)$		$\Theta\left(n^3t/\left(B\sqrt{M}\right)\right)$	
		$\Theta(nt)$		$\Theta\left(\frac{nt}{\sqrt{M}}\left(\log n+\sqrt{M}\right)\right)$
LCS / edit distance	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(n^{\log 3}\right)$	$\Theta\left(n^2/(BM)\right)$	$\Theta\left(\frac{n}{M}\left(\log n+M^{\log 3}\right)\right)$
Spoken word recognition				
Binomial coefficient				
Bitonic TSP				

WR-DP algorithms have very high parallelism



WR-DP algorithms are efficient++ in practice



3. Efficient Viterbi algorithm

[Euro-Par 2016]

Viterbi has a crazy data-dependent recurrence

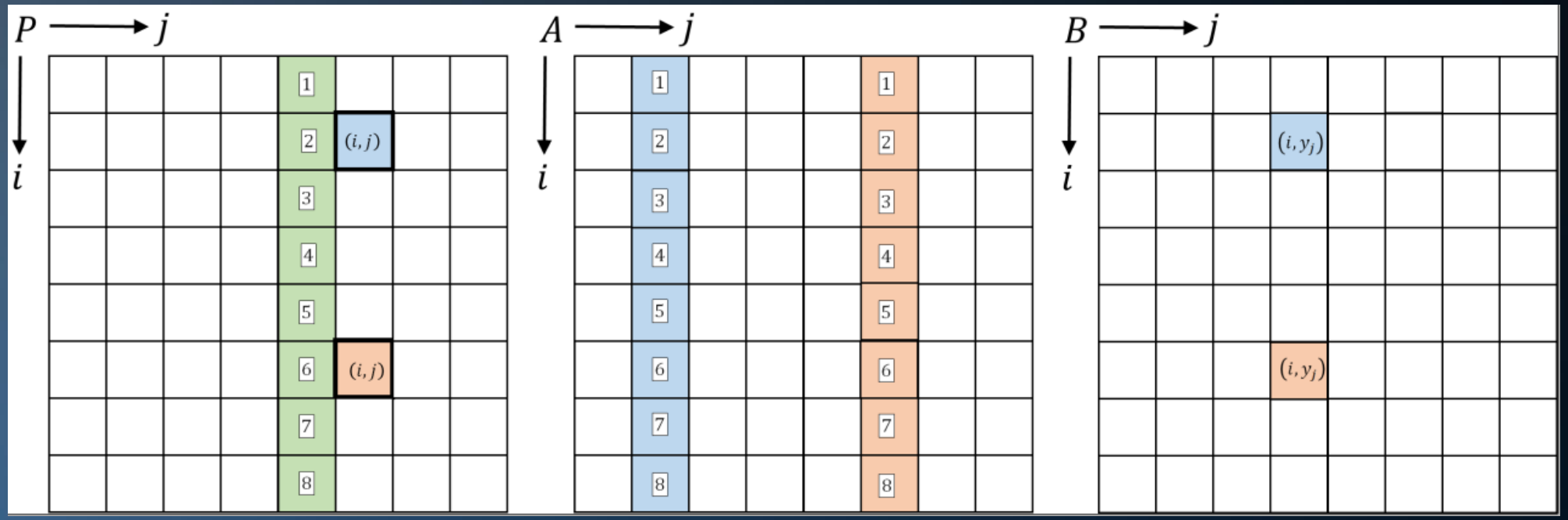
$$P[i,j] = \begin{cases} I[i] \times B[i, y_1], & \text{if } j = 1 \\ \max\limits_{k \in [1,n]} \left(P[k, j-1] \times A[k,i] \times B\left[i, y_j\right] \right), & \text{if } j > 1 \end{cases}$$

Oops.. data-dependency..

- $\{s_1, \dots, s_n\}$ = state space
- $\{o_1, \dots, o_m\}$ = observation space
- $\{y_1, \dots, y_t\}$ = observations
- $\{x_1, \dots, x_t\}$ = hidden states

- $I[i]$ = prob. that $x_1 = s_i$
- $P[i,j]$ = prob. of most likely path to reach state s_i at observation o_j
- $A[i,j]$ = transition prob. from s_i to s_j
- $B[i,j]$ = prob. of observing o_j at s_i

Viterbi dependencies are complicated



We design parallel + cache-efficient + cache-oblivious VA

	Parallel	Cache-efficient	Cache-oblivious
Existing algorithms	Yes	Yes	No
Existing algorithms	Yes	No	Yes
Our paper	Yes	Yes	Yes

Standard I-DP & R-DP for VA are not efficient

Loop-Viterbi (P, A, B)

1. for $j \leftarrow 2$ to t do
2. parallel for $i \leftarrow 1$ to n do
3. for $k \leftarrow 1$ to n do
4. $P[i,j] \leftarrow \text{Min}\begin{pmatrix} P[k,j-1] \times A[k,i] \\ \times B[i,y_j], P[i,j] \end{pmatrix}$

I-DP : Not efficient

Viterbi-D&C (P, A, B)

1. for $j \leftarrow 2$ to t do
2. $X \leftarrow P[\dots, j]; U \leftarrow P[\dots, j-1]$
3. $V \leftarrow A; W \leftarrow B[\dots, y_j]$
4. A(X, U, V, W)

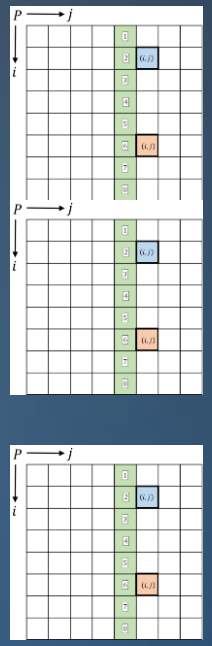
A(X, U, V, W)

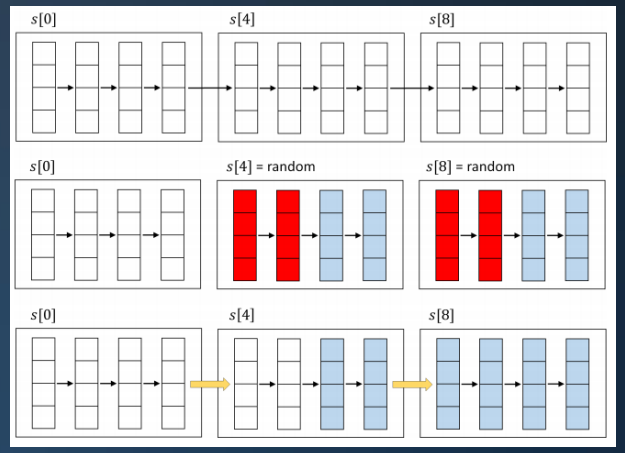
1. if V is a small matrix then A-loop(X, U, V, W)
2. else
3. par: A(X_1, U_1, V_{11}, W_1), A(X_2, U_1, V_{12}, W_2)
4. par: A(X_1, U_2, V_{21}, W_1), A(X_2, U_2, V_{22}, W_2)

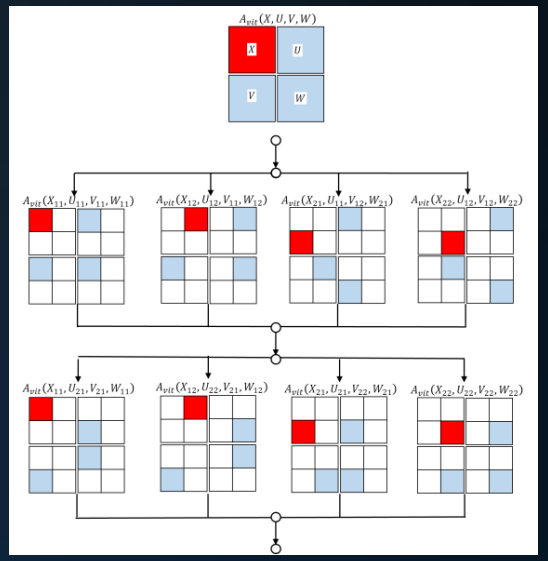
R-DP : Not efficient

Core idea: Efficient single instance VA = Efficient multi-instance VA + inefficient single instance VA

Get an efficient multi-instance VA. Combine it with the rank convergence method from PPoPP 2014.







1. Multi-instance VA

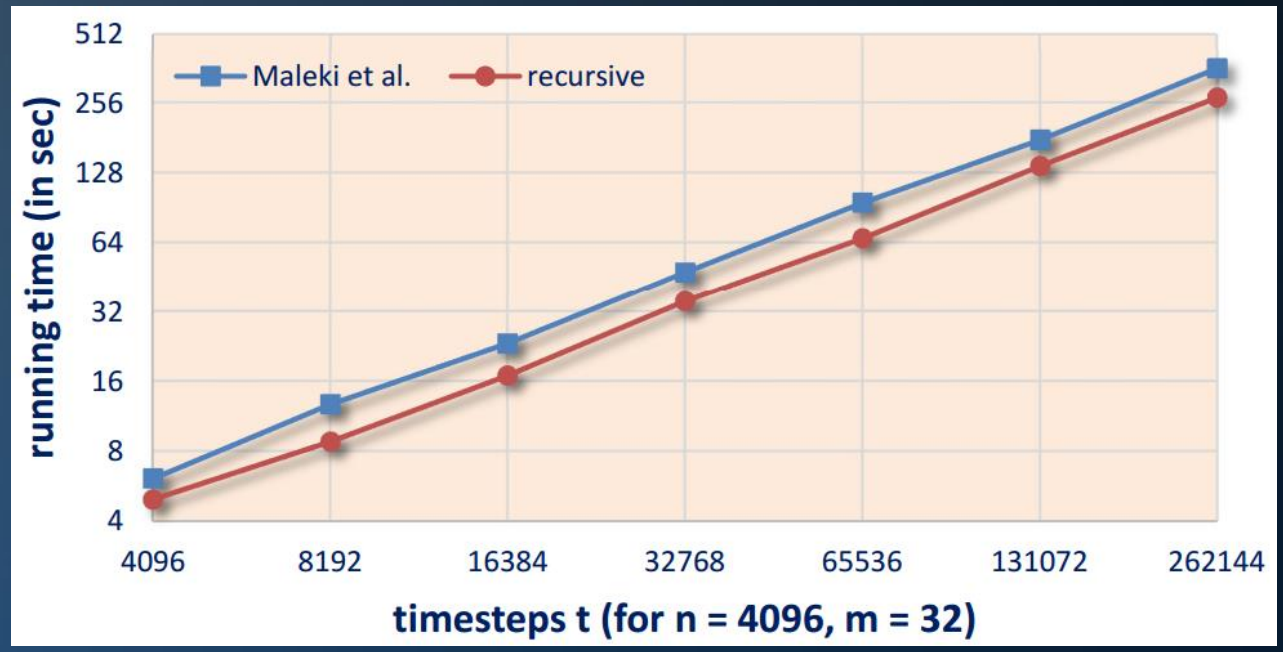
2. Rank convergence VA

3. Combine multi-instance VA & rank convergence VA

Our VA is efficient in theory & practice

Serial cache complexity $= O\left(\frac{n^2 t\lambda}{B\sqrt{M}} + \frac{n^2 t\lambda}{M} + \frac{n(n2^{\lambda}+t\lambda)}{B} + 2^{\lambda}\right)$

where, λ = number of iterations



Concluding remarks

- Autogen is the first algorithm to automatically discover nontrivial divide-and-conquer algorithms
- Autogen-Wave can be used to discover fastest portable dynamic programs

---

- Dissertation statement: It is possible to develop algorithm(s) / framework(s) to automatically / semi-automatically discover algorithms that are simultaneously non-trivial, simple, fast, portable, and robust, which can be used to solve to a wide class of DP problems

Future work

1. Autogen-Wave automation
2. Autogen for distributed systems
3. Autogen for MapReduce
4. Autogen for GPUs
5. Autogen for irregular DPs
6. Autogen for iterative wavefront algorithms
7. Autogen for planar graphs
8. Autogen for tiled iterative wavefront DP algorithms
9. Autogen for PGAS (partitioned global address space)