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

Pramod Ganapathi (Adviser: Rezaul Chowdhury)

Our Vision

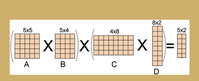
Problem specification → Efficient algorithm → High-performing implementation

Dissertation Statement

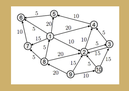
It is possible to develop algorithm(s) / framework(s) to automatically / semi-automatically discover algorithms that are simultaneously nontrivial, simple, fast, portable, and robust.

DP is Important

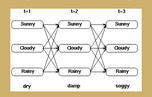
DP = dynamic programming



Chain matrix multiplication



FW APSP



Viterbi algorithm

Edit distance

DP Implementations

I-DP = iterative, TI-DP = tiled iterative, R-DP = recursive

Feature	I-DP	TI-DP	R-DP
Cache-efficiency			
Cache-obliviousness			
Cache-adaptivity	NA		
Higher parallelism			
Optimizable kernels	NA		
Energy-efficiency			
Bandwidth-efficiency			

= bad / no = good / yes

State of the Art & Science

Algorithm design was an art

Iterative →  → Recursive

Now, algorithm design is a science

Any DP implementation →  → Recursive

Autogen

Autogen is Efficient

```
LOOP-PARENTHESIS(C, n)
1. for i ← n − 1 to 0 do
2.   for j ← i + 2 to n do
3.     for k ← i + 1 to j do
4.       C[i, j] ← min ( C[i, k] + C[k, j]
                       +w(i, j, k), C[i, j] )
```

I-DP → Autogen → R-DP

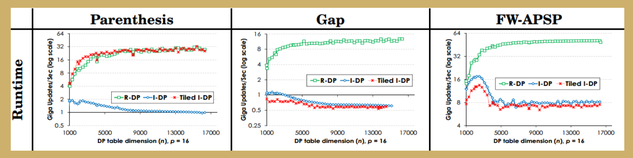
```
A_par(X)
1. if X is a small matrix then A_loop-par(X)
2. else
3.   parallel: A_par(X11), A_par(X22)
4.   B_par(X22, X11, X22)

B_par(X, U, V)
1. if X is a small matrix then B_loop-par(X, U, V)
2. else
3.   B_par(X21, U22, V11)
4.   parallel: C_par(X11, U12, X21), C_par(X22, X21, V12)
5.   parallel: B_par(X11, U11, V11), B_par(X22, U22, V22)
6.   C_par(X12, U12, X22)
7.   C_par(X12, X11, V12)
8.   B_par(X12, U11, V22)

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

R-DP is Efficient in Theory

Problem	Work (T_1)	$\mathcal{I}$-DP Serial cache comp. (Q_1)	$\mathcal{I}$-DP Span (T_∞)	$\mathcal{R}$-DP Serial cache comp. (Q_1)	$\mathcal{R}$-DP Span (T_∞)
Parenthesis problem	$\Theta(n^3)$	$\Theta(n^3)$	$\Theta(n^2)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log 3}\right)$
Floyd-Warshall's APSP 3-D	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n \log n)$	$\Theta(n^3/B)$	$\mathcal{O}\left(n \log^2 n\right)$
Floyd-Warshall's APSP 2-D	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n \log n)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n \log^2 n\right)$
LCS / Edit distance	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n)$	$\Theta(n^2/(BM))$	$\Theta\left(n^{\log 3}\right)$
Multi-instance Viterbi	$\Theta(n^3 t)$	$\Theta(n^3 t/B)$	$\Theta(nt)$	$\Theta\left(n^3 t/(B\sqrt{M})\right)$	$\Theta(nt)$
Gap problem	$\Theta(n^3)$	$\Theta(n^3)$	$\Theta(n^2)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log 3}\right)$
Protein accordion folding	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n^2)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n \log n)$
Spoken-word recognition	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n)$	$\Theta(n^2/(BM))$	$\Theta\left(n^{\log 3}\right)$
Function approximation	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n^2)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log 3}\right)$
Binomial coefficient	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n)$	$\Theta(n^2/(BM))$	$\Theta\left(n^{\log 3}\right)$
Bitonic traveling salesman	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n)$	$\Theta(n^2/(BM))$	$\Theta(n \log n)$
Matrix multiplication	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta(n)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n)$
Bubble sort	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n^2)$	$\Theta(n^2/(BM))$	$\Theta(n)$
Selection sort	$\Theta(n^2)$	$\Theta(n^2/B)$	$\Theta(n^2)$	$\Theta(n^2/(BM))$	$\Theta(n)$
Insertion sort	$\Theta(n^2)$	$\Theta(n^2/B)$	$\mathcal{O}(n^2)$	$\mathcal{O}\left(n^{\log 3}/(BM^{\log 3-1})\right)$	$\mathcal{O}(n)$

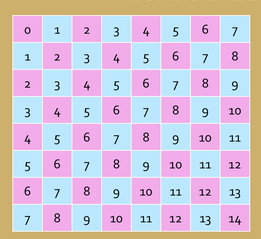
M = memory size, B = block size

R-DP is Efficient in Practice

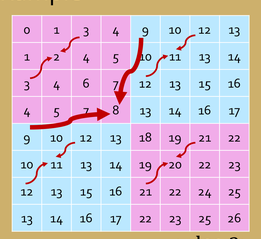


R-DP can be Made Faster

Consider LCS example



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



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

R-DP is slow due to artificial dependencies
Parallelism can be increased by removing these artificial dependencies

Autogen-Wave is Efficient++

Recursive →  → Recursive wavefront

Autogen-Wave

WR-DP = recursive wavefront

```
A_par(X)
1. if X is a small matrix then A_loop-par(X)
2. else
3.   parallel: A_par(X11), A_par(X22)
4.   B_par(X22, X11, X22)

B_par(X, U, V)
1. if X is a small matrix then B_loop-par(X, U, V)
2. else
3.   B_par(X21, U22, V11)
4.   parallel: C_par(X11, U12, X21), C_par(X22, X21, V12)
5.   parallel: B_par(X11, U11, V11), B_par(X22, U22, V22)
6.   C_par(X12, U12, X22)
7.   C_par(X12, X11, V12)
8.   B_par(X12, U11, V22)

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

R-DP → WR-DP → Autogen-Wave

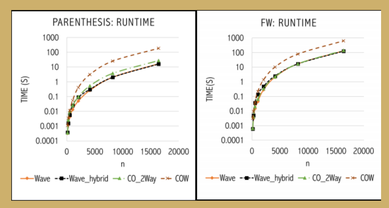
```
RECURSIVE-WAVEFRONT-PARENTHESIS()
1. w ← 0
2. while w < ∞ do w ← A(G, G, G, w)

A(X, X, X, w)
1. v_i ← ∞ for all i ∈ [1, 3]
2. if X is an n' × n' matrix then
3.   if w = S_A(X, X, X) then A_chunk(X, X, X)
4. else
5.   F_1..3 ← {A, A, B}
6.   arg_1..3 ← {(X11, X11, X11), (X22, X22, X22), }
                (X12, X11, X22)
7.   parallel: for i ← 1 to 3 do
8.     if w < S_F_i(arg_i) then v_i ← S_F_i(arg_i)
9.     elif w ≤ Ê_F_i(arg_i) then v_i ← F̂_i(arg_i, w)
10.  sync
11. return min v_i for all i ∈ [1, 3]

B(X, U, V, w)
1. v_i ← ∞ for all i ∈ [1, 8]
2. if X is an n' × n' matrix then
3.   if w = S_B(X, U, V) then B_chunk(X, U, V)
4. else
5.   F_1..8 ← {B, C, C, B, B, C, C, B}
6.   arg_1..8 ← {(X21, U22, V11), (X11, U12, V21),
                 (X22, X21, V12), (X11, U11, V11),
                 (X22, X22, V22), (X12, U12, X22),
                 (X12, X11, V12), (X12, U11, V22)}
7.   parallel: for i ← 1 to 8 do
8.     if w < S_F_i(arg_i) then v_i ← S_F_i(arg_i)
9.     elif w ≤ Ê_F_i(arg_i) then v_i ← F̂_i(arg_i, w)
10.  sync
11. return min v_i for all i ∈ [1, 8]

C(X, U, V, w)
1. v_i ← ∞ for all i ∈ [1, 8]
2. if X is an n' × n' matrix then
3.   if w = S_C(X, U, V) then C_chunk(X, U, V)
4. else
5.   F_1..8 ← {C, C, C, C, C, C, C, C}
6.   arg_1..8 ← {(X11, U11, V11), (X12, U11, V12),
                 (X21, U21, V11), (X22, U21, V12),
                 (X11, U12, V21), (X12, U12, V22),
                 (X21, U22, V21), (X22, U22, V22)}
7.   parallel: for i ← 1 to 8 do
8.     if w < S_F_i(arg_i) then v_i ← S_F_i(arg_i)
9.     elif w ≤ Ê_F_i(arg_i) then v_i ← F̂_i(arg_i, w)
10.  sync
11. return min v_i for all i ∈ [1, 8]
```

```
S_A(X, X, X)
1. return 𝔾(x_r, x_c)

S_B(X, U, V)
1. return 𝔾(x_r + n − 1, x_c)

S_C(X, U, V)
1. m ← (x_r + n − 1 + x_c)/2; ũ ← u_c + n − 1
2. if u_c > m then return max {𝔾(u_r + n − 1, u_c), 𝔾(u_c, x_c)}+1
3. elif ũ < m then return max {𝔾(u_r + n − 1, ũ), 𝔾(ũ, x_c)}+1
4. else return
   (max {𝔾(u_r + n − 1, m), 𝔾(m, x_c)} + 1):[u_c > (x_r + x_c)/2]

Ê_A(X, X, X)
1. return 𝔾(x_r, x_c + n − 1)

Ê_B(X, U, V)
1. return 𝔾(x_r, x_c + n − 1)

Ê_C(X, U, V)
1. lval ← max {𝔾(u_r, u_c), 𝔾(u_c, x_c + n − 1)}
2. rval ← max {𝔾(u_r, u_c + n − 1), 𝔾(u_c + n − 1, x_c + n − 1)}
3. return (max {lval, rval} + 1).[u_c > (x_r + x_c)/2]

𝔾(i, j)
1. if (j − i) ≤ 1 then return (j − i) else return 2 × (j − i) − 1
```

WR-DP is Efficient++ in Theory

Problem	Work (T_1)	$\mathcal{R}$-DP Serial cache comp. (Q_1)	$\mathcal{R}$-DP Span (T_∞)	$\mathcal{WR}$-DP Best serial cache comp. (Q_1)	$\mathcal{WR}$-DP Best span (T_∞)
Parenthesis problem	$\Theta(n^3)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log 3}\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n \log n)$
Floyd-Warshall's APSP 3-D	$\Theta(n^3)$	$\Theta(n^3/B)$	$\Theta\left(n \log^2 n\right)$	$\Theta(n^3/B)$	$\Theta(n \log n)$
Floyd-Warshall's APSP 2-D	$\Theta(n^3)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n \log^2 n\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n \log n)$
LCS / Edit distance	$\Theta(n^2)$	$\Theta(n^2/(BM))$	$\Theta\left(n^{\log 3}\right)$	$\Theta(n^2/(BM))$	$\Theta(n \log n)$
Gap problem	$\Theta(n^3)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log 3}\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n \log n)$
3-point stencil	$\Theta(n^2)$	$\Theta(n^2/(BM))$	$\Theta\left(n^{\log 3}\right)$	$\Theta(n^3/(BM))$	$\Theta(n \log n)$
Protein accordion folding	$\Theta(n^3)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n \log n)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n \log n)$
Spoken-word recognition	$\Theta(n^2)$	$\Theta(n^2/(BM))$	$\Theta\left(n^{\log 3}\right)$	$\Theta(n^2/(BM))$	$\Theta(n \log n)$
Function approximation	$\Theta(n^3)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta\left(n^{\log 3}\right)$	$\Theta\left(n^3/(B\sqrt{M})\right)$	$\Theta(n \log n)$
Binomial coefficient	$\Theta(n^2)$	$\Theta(n^2/(BM))$	$\Theta\left(n^{\log 3}\right)$	$\Theta(n^2/(BM))$	$\Theta(n \log n)$
Bitonic traveling salesman	$\Theta(n^2)$	$\Theta(n^2/(BM))$	$\Theta(n \log n)$	$\Theta(n^2/(BM))$	$\Theta(n \log n)$

WR-DP is Efficient++ in Practice

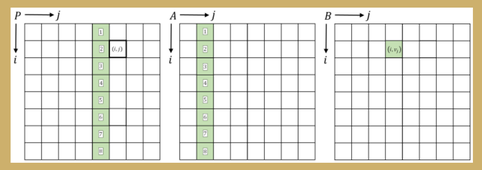


Viterbi Algorithm (VA)

Autogen does not work on data-dependent problems

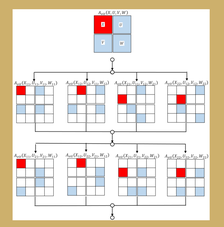
Viterbi recurrence is data-dependent

No efficient cache-oblivious algorithm known



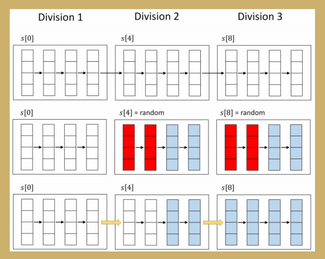
Multi-instance VA is Efficient

Get multiple instances
Get ith column vector from all instances
Run an efficient MM-like algorithm



Inefficient Parallel VA

Split timesteps into segments
Initialize segment start vectors with random values and run VA
If rank convergence occurs, VA ends fast

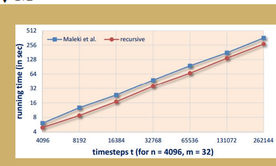


Single-instance VA is Efficient

Idea: Consider segments as instances

The serial cache complexity is $\mathcal{O}\left(\frac{n^2 t\lambda}{B\sqrt{M}} + \frac{n^2 t\lambda}{M} + \frac{n(n2^\lambda + t\lambda)}{B} + 2^\lambda\right)$

Our algorithm beats the fastest VA implementation



Conclusion

In future, computers will discover algorithms
Autogen (PPoPP 2016, Invited to TOPC journal)
Autogen-Wave (Under review)
Viterbi algorithm (Euro-Par 2016)