Fast and Cache-Oblivious Dynamic Programming with Local Dependencies

Philip Bille[1], Morten Stöckel[2]
[1] *Technical University of Denmark*
[2] *IT University of Copenhagen*

March 9 2012

The Problem
- What we have done
- Dynamic Programming with Local Dependencies
- Longest Common Subsequence
- Standard Algorithms

Re-computations and I/O's
- The FastLSA Algorithm
- The CO Algorithm

The FastCO Algorithm

Experimental Results

Round Off

The Problem

- Contribution: A new approximate string matching algorithm, which is cache oblivious and provides a time/space tradeoff useful for large input strings.
- Theoretical bounds of the new method, named FastCO, are at least as good as current state of the art methods.
- Practical experiments show our new method to be highly competetive for large input strings.

Alignments

- An alignment of strings $X = \{tcgacggc\}$ and $Y = \{tatcagta\}$:

$$\begin{array}{cccccccccc} t & c & g & a & - & c & g & g & c & - \\ | & & & | & & | & & | & & \\ t & - & - & a & t & c & a & g & t & a \end{array}$$

- Score of an alignment: Summation of scores of aligned characters.
- Applications in bioinformatics, dictionaries, subversion etc. where n *can be large*.

The Problem

- Problem input: *Strings* X and Y, consisting of *characters* from Σ, where $|X| = |Y| = n$.
- Access to score function $S : (\sigma + 1) \times (\sigma + 1) \mapsto \mathbb{R}$.
- Problem output: An *alignment* of X and Y which is optimal wrt. score function S.

Alignments

- An alignment of strings $X = \{tcgacggc\}$ and $Y = \{tatcagta\}$:

$$\begin{array}{cccccccccc} t & c & g & a & - & c & g & g & c & - \\ | & & & | & & | & & | & & \\ t & - & - & a & t & c & a & g & t & a \end{array}$$

- Score of an alignment: Summation of scores of aligned characters.
- Applications in bioinformatics, dictionaries, subversion etc. where n *can be large*.

Solving by Dynamic Programming

- Problem input: strings X and Y, where $|X| = |Y| = n$.
- Input strings form *DPM* c of $n + 1 \times n + 1$ values.
- Invariant of c: $c\,[i, j]$ for $0 \leq i, j \leq n$ holds the score of the optimal solution for strings $X[1 \ldots i]$ and $Y[1 \ldots j]$.

Filling the DPM

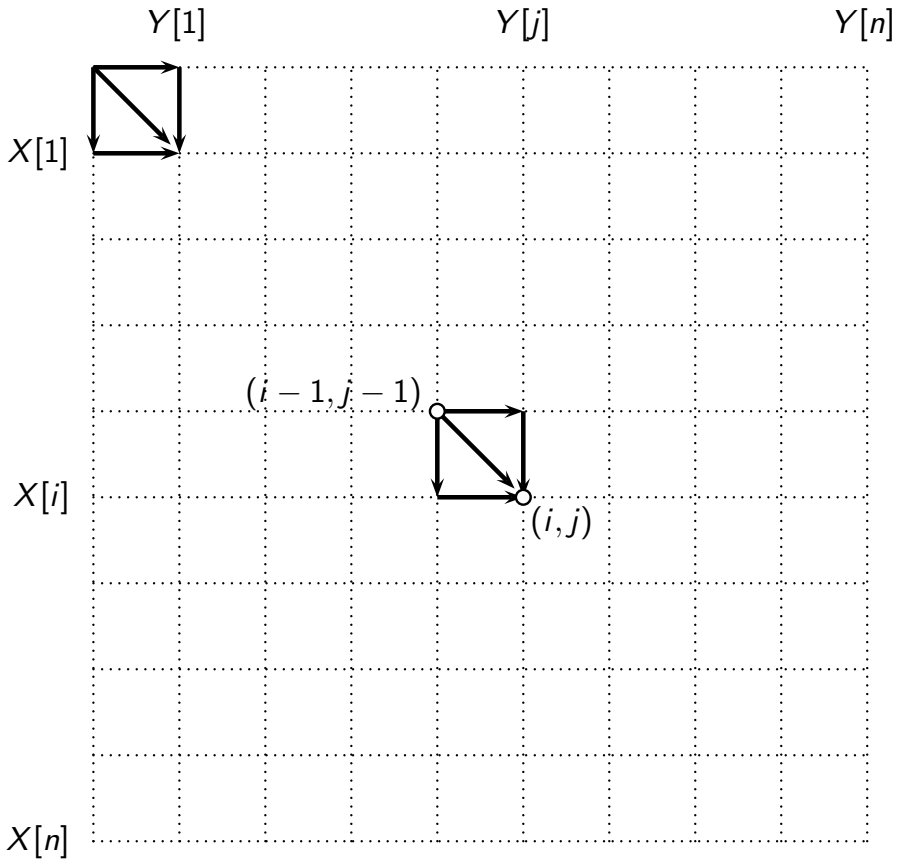
- Each value of c for $0 \leq i, j \leq n$ follows recurrence 1.

$$c[i,j] = \begin{cases} h((i,j)) & \text{if } j = 0 \vee i = 0, \\ f((i,j), X[i], Y[j]) & \text{if } i,j > 0 \end{cases} \quad (1)$$

- Functions f and h compute the values in constant time and make use of S.
- Function f derives the current value from one of three adjacent values.

Generating a Solution

- The solution can be seen as a *path* through c, consisting of all c-value derivations from upper left to bottom right.
- Graph theoretical view: Optimal path in weighted grid graph with $n+1 \times n+1$ nodes.

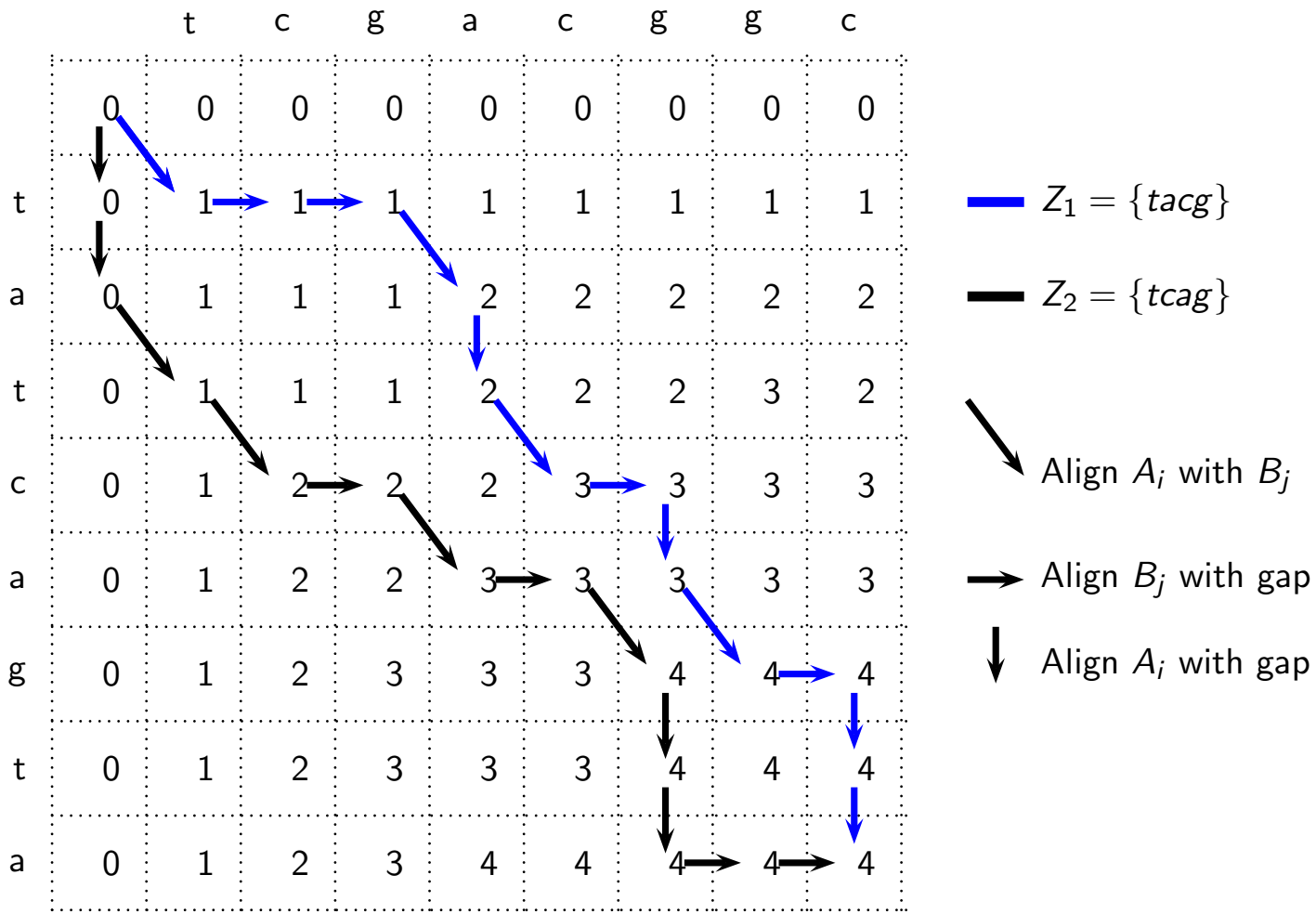


Chosen problem: LCS

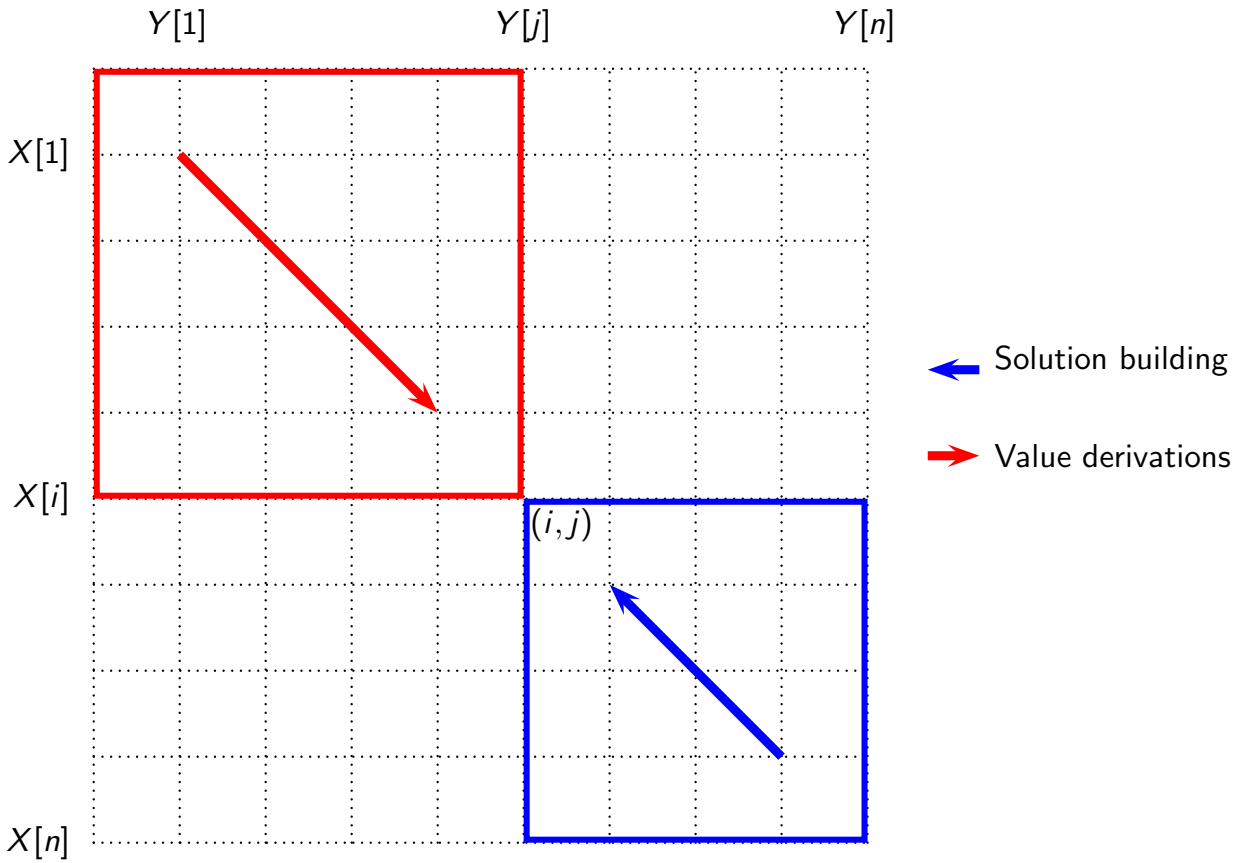
- LCS: Score function yields 1 for equal characters, 0 otherwise and goal is to maximize the alignment score.
- DPM values follow recurrence 2.

$$c[i,j] = \begin{cases} 0 & \text{if } j = 0 \vee i = 0, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \wedge X[i] = Y[j], \\ \max \begin{cases} c[i,j-1] \\ c[i-1,j] \end{cases} & \text{if } i,j > 0 \wedge X[i] \neq Y[j] \end{cases} \quad (2)$$

An LCS example



Why is this difficult?

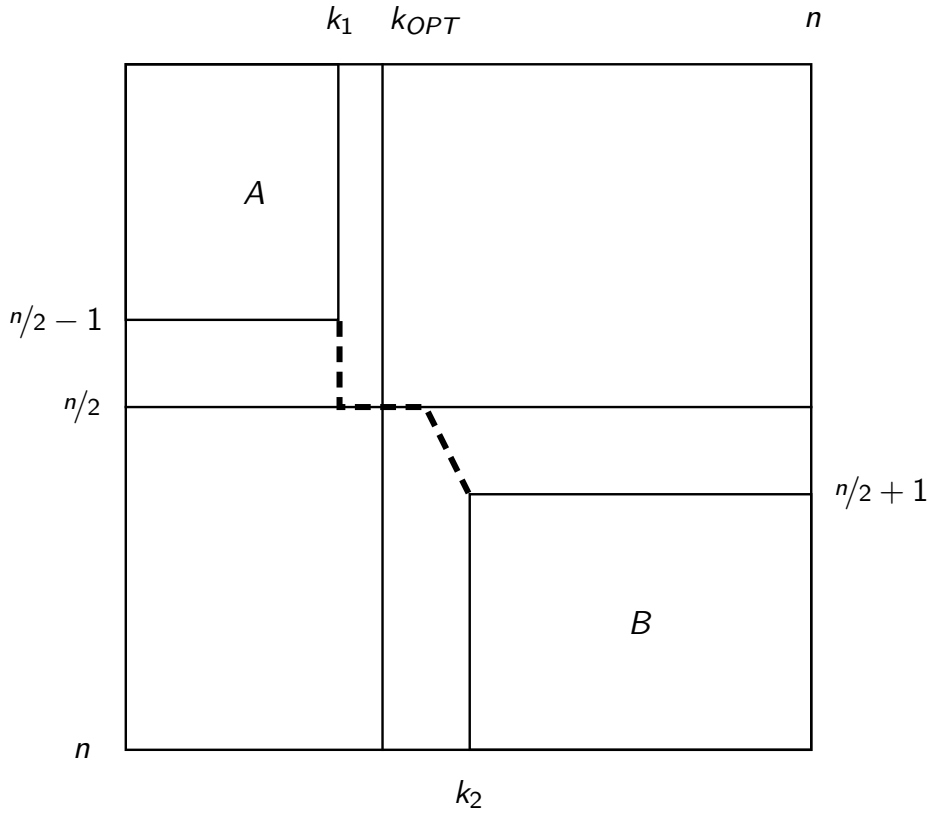


Dynamic Programming

1. Forward pass: Compute and store all $n+1 \times n+1$ values in a row wise manner using recurrence 2 for each value.
2. Backward pass: Starting from $c[n,n]$, follow derivations to achieve solution path through c.
3. Time: $O\left(n^2\right)$
4. Space: $O\left(n^2\right)$
5. I/O's: $O\left(\frac{n^2}{B}+\frac{n}{B}+1\right)$

Hirschberg's Algorithm

- Divide and conquer: Calculate a split such that the current problem can be computed by two smaller disjoint problems.
- Time: $\sum_{i=1}^{\log n} cn^2/2^{i-1} \leq 2cn^2 = O\left(n^2\right)$
- Space: $O\left(n\right)$
- I/O's: $O\left(\frac{n^2}{B} + \frac{n}{B} + 1\right)$



The Problem
- What we have done
- Dynamic Programming with Local Dependencies
- Longest Common Subsequence
- Standard Algorithms

Re-computations and I/O's
- The FastLSA Algorithm
- The CO Algorithm

The FastCO Algorithm

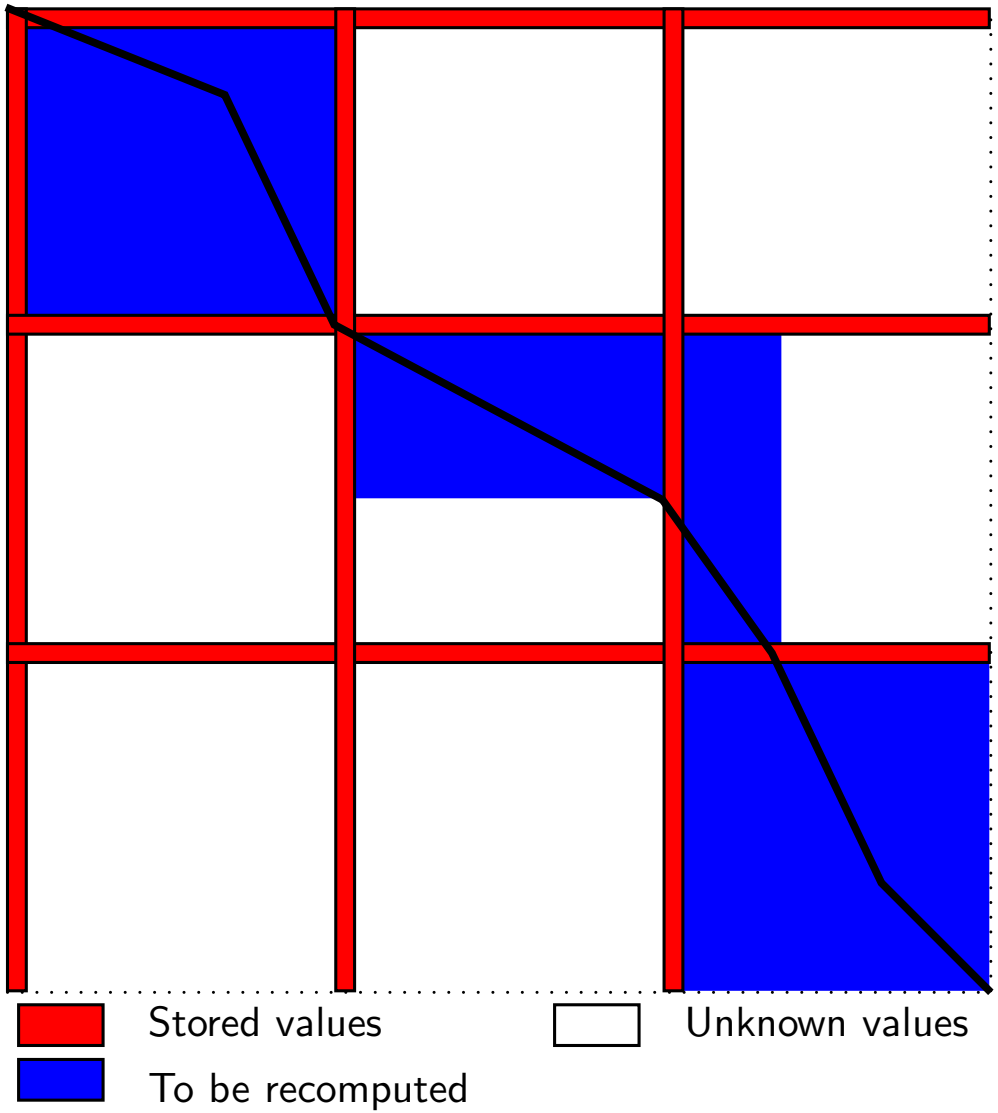
Experimental Results

Round Off

FastLSA Intuition

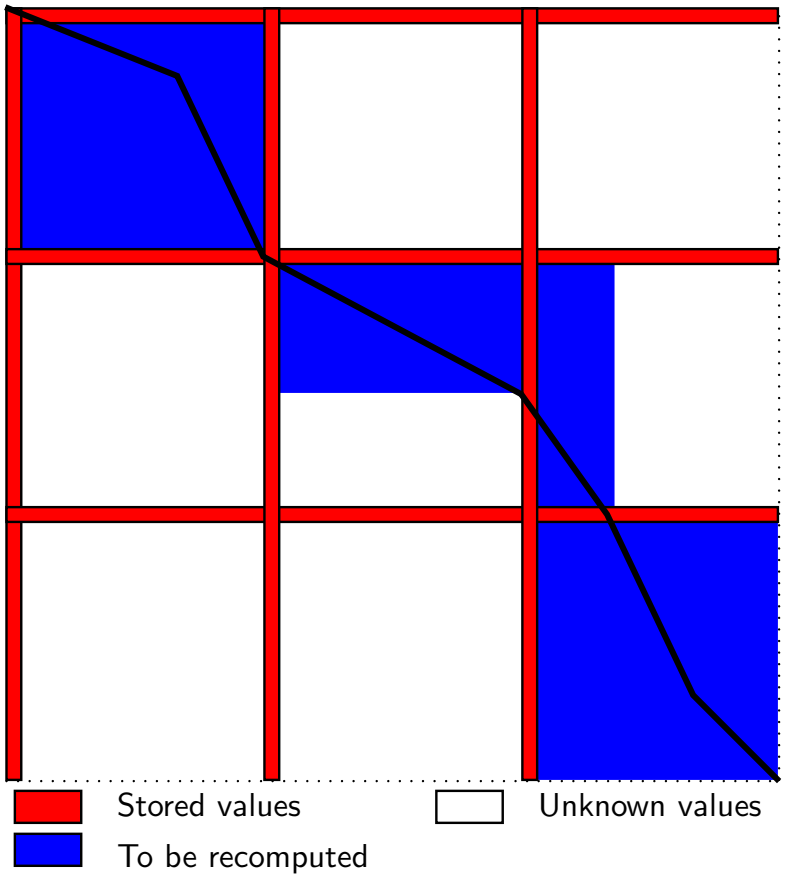
- Inefficiency of Hirschberg: Linear space means quadratic re-computations.
- FastLSA by Driga et al.: Split DPM in k parts in both directions. Store k "important" rows and columns to achieve a faster backward pass. If current problem matrix size is less than D compute directly, recurse otherwise.
- Provides a time/space tradeoff through parameter k.

FastLSA time/space tradeoff



FastLSA Complexities

- Forward: $O\left(n^2\right)$
- Backward: $O\left(n^2/k\right)$
- Space: $O\left(kn + D\right)$
- I/O's: $O\left(\frac{n^2}{B} + \frac{n}{B} + 1\right)$



Wasting I/O's

- For each character in X, Y is accessed from left to right.
- Incurs $O(\frac{n}{B})$ I/O's per row.
- $O(\frac{n^2}{B})$ total I/O's.
- Issue: Y continuously accessed and thus brought into memory.

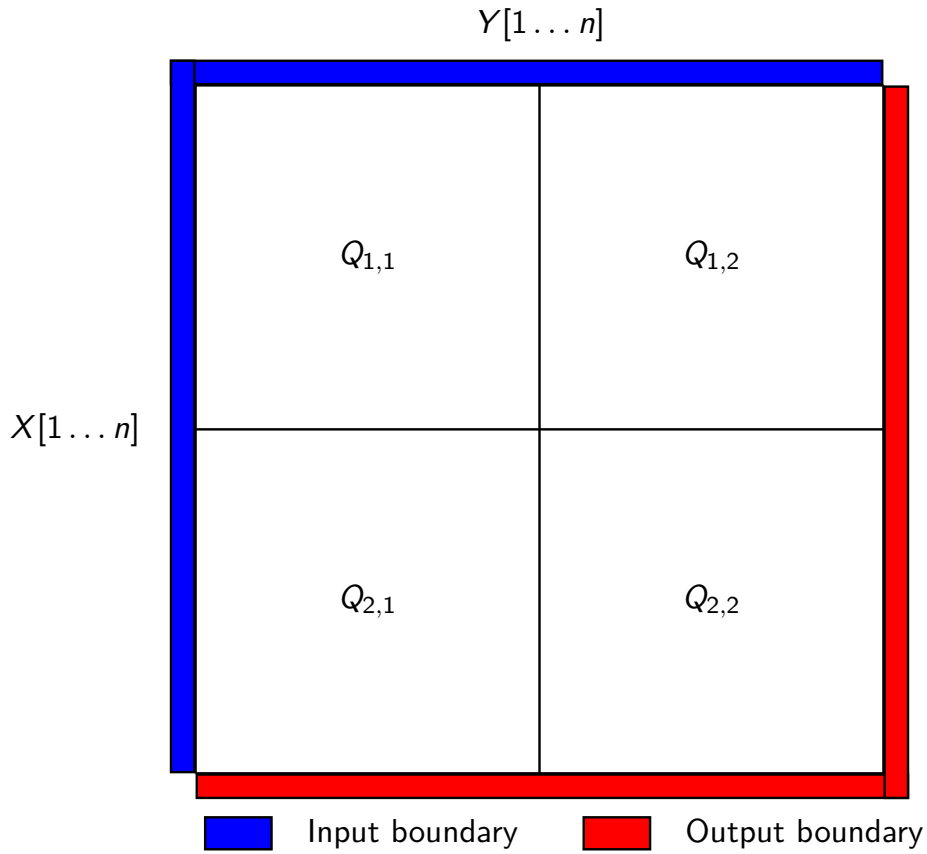
		t	c	g	a	c	g	g	c
	0	0	0	0	0	0	0	0	0
t	0	1	1	1	1	1	1	1	1
a	0	1	1	1	2	2	2	2	2
t	0	1	1	1	2	2	2	3	2
c	0	1	2	2	2	3	3	3	3
a	0	1	2	2	3	3	3	3	3
g	0	1	2	3	3	3	4	4	4
t	0	1	2	3	3	3	4	4	4
a	0	1	2	3	4	4	4	4	4

CO Intuition

- Deals with I/O inefficiency of left-right computation.
- Algorithm CO by Chowdhury et al.: If input matrix is constant size compute directly, otherwise split DPM in 4 quadrants and recurse. Compute boundaries of quadrants in a cache efficient manner to achieve better I/O complexity.

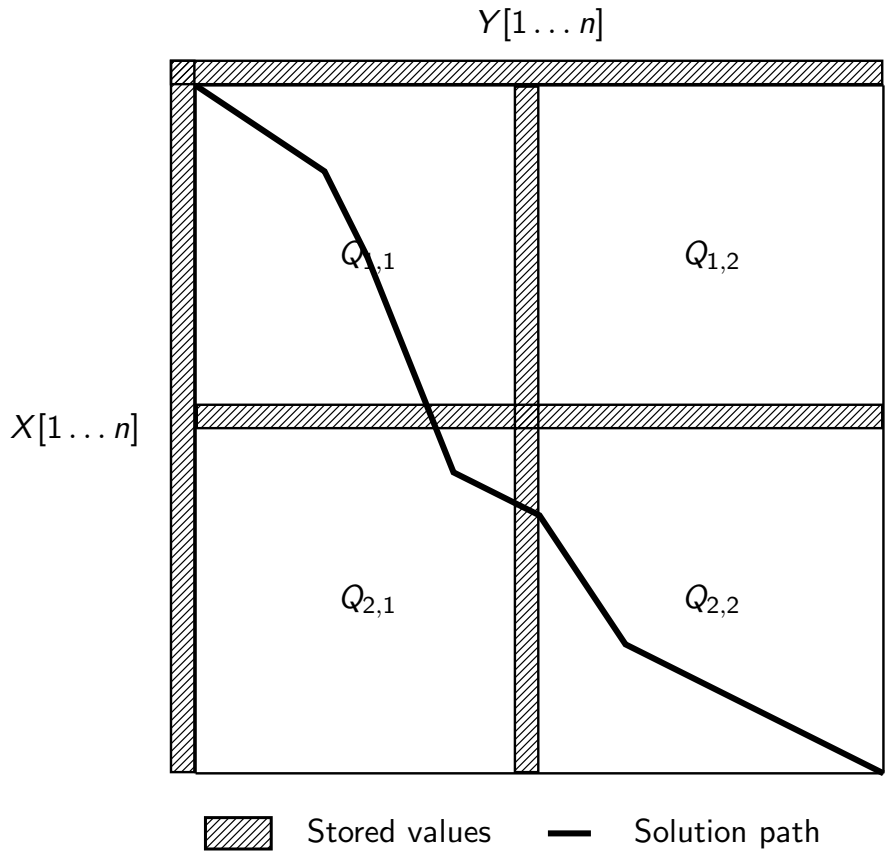
Computing Boundaries

- Problem: Given input boundary and strings, compute output boundary.
- Divide DPM into 4 quadrants and recursively compute quadrants in a left-right top-down order. Pass and combine boundaries.
- I/O's: $O\left(\frac{n^2}{BM}\right)$



CO Complexities

- Forward: $O\left(n^2\right)$
- Backward: $O\left(n^2\right)$
- Space: $O\left(n\right)$
- I/O's: $O\left(\frac{n^2}{BM} + \frac{n}{B} + 1\right)$



The Problem
- What we have done
- Dynamic Programming with Local Dependencies
- Longest Common Subsequence
- Standard Algorithms

Re-computations and I/O's
- The FastLSA Algorithm
- The CO Algorithm

The FastCO Algorithm

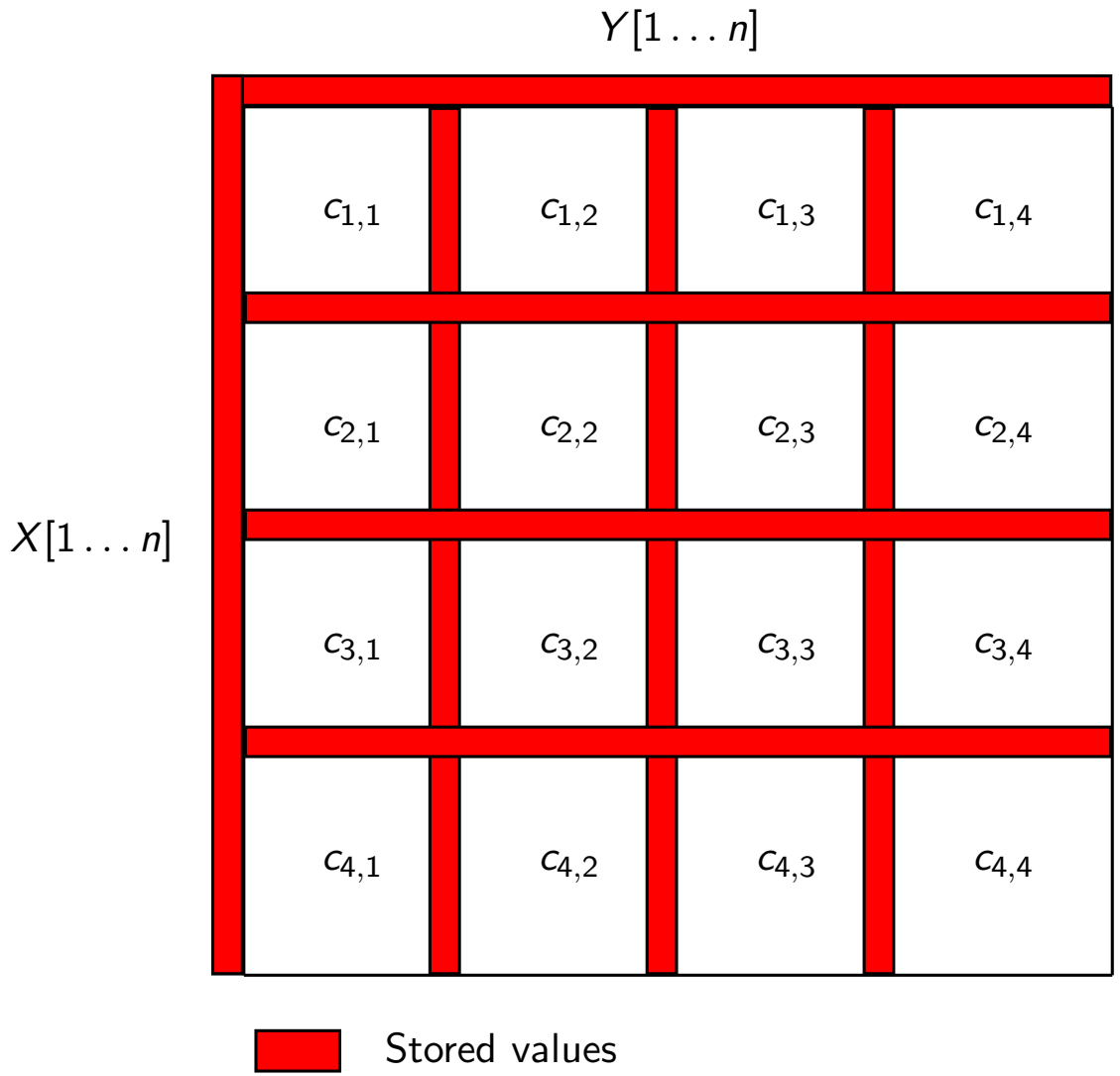
Experimental Results

Round Off

FastCO Intuition

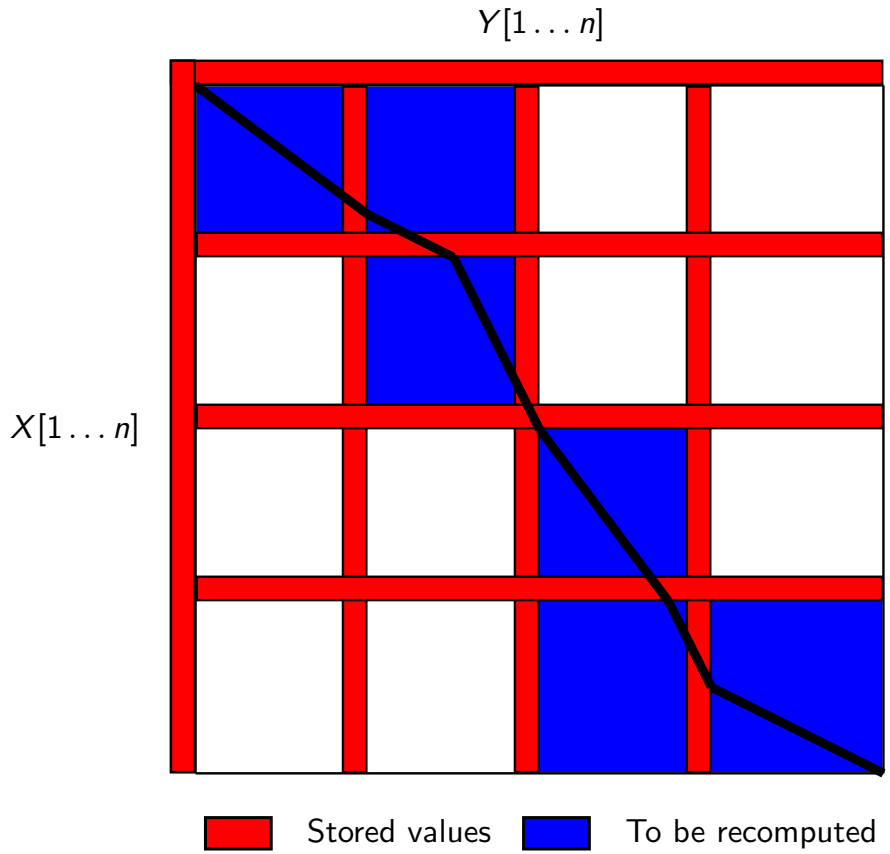
- FastCO: Combine the two ideas from CO and FastLSA to create a method better than both of them.
- Generalize CO to use the time/space tradeoff through parameter k.
- Result: A cache oblivious time/space tradeoff for dynamic programming with local dependencies.

FastCO Forward Pass



FastCO Backward Pass

- Compute $O(k)$ submatrices each of size $O\left(\frac{n^2}{k^2}\right)$.
- Backward: $O\left(\frac{n^2}{k}\right)$

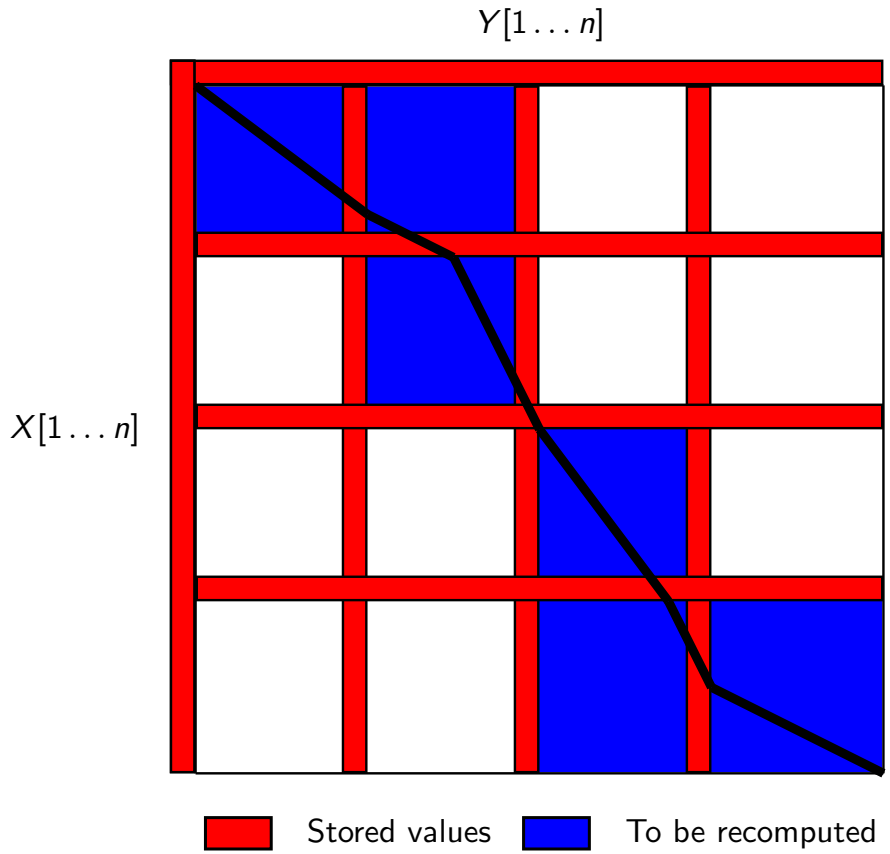


FastCO I/O's

- I/O complexity now dependant on k.

$$I(n,k) = O(\frac{n^2 k}{MB} + \frac{n}{B} + 1)$$

- Typically $k = O(1)$



Algorithm Overview

Algorithm	Forward Pass	Backward Pass	Space	I/O	CO
FullMatrix	$O\left(n^2\right)$	$O(n)$	$O\left(n^2\right)$	$O\left(\frac{n^2}{B}+\frac{n}{B}+1\right)$	Yes
Hirschberg	$O\left(n^2\right)$	$O\left(n^2\right)$	$O(n)$	$O\left(\frac{n^2}{B}+\frac{n}{B}+1\right)$	Yes
FastLSA$_k$	$O\left(n^2\right)$	$O\left(\frac{n^2}{k}+n\right)$	$O(nk+D)$	$O\left(\frac{n^2}{B}+\frac{n}{B}+1\right)$	No
CO	$O\left(n^2\right)$	$O\left(n^2\right)$	$O(n)$	$O\left(\frac{n^2}{BM}+\frac{n}{B}+1\right)$	Yes
FastCO$_k$	$O\left(n^2\right)$	$O\left(\frac{n^2}{k}+n\right)$	$O(nk)$	$O\left(\frac{n^2k}{BM}+\frac{n}{B}+1\right)$	Yes

The Problem
- What we have done
- Dynamic Programming with Local Dependencies
- Longest Common Subsequence
- Standard Algorithms

Re-computations and I/O's
- The FastLSA Algorithm
- The CO Algorithm

The FastCO Algorithm

Experimental Results

Round Off

Overview

- All algorithms were implemented in C++.
- Standardized DNA strings of sizes up to 2^{21} as input.
- Three architectures: Intel M, Intel I7, AMD X2.
- FastLSA: $D = 1000 \times 1000$ for Intel I7 and AMD X2, 500×500 for Intel M.
- k values used: 2^i for $i \in \{3, 4, 5\}$.

Intel I7 Results

- 2.66GHz. 32KB L1, 256KB L2, 8MB L3 cache. 4GB memory

n	HB	CO	FLSA_8	FLSA_{16}	FLSA_{32}	FCO_8	FCO_{16}	FCO_{32}
2^{16}	0.016h	0.012h	0.009h	0.009h	**0.008h**	0.009h	0.009h	**0.008**h
2^{17}	0.063h	0.049h	0.0387h	0.036h	**0.033h**	0.036h	0.035h	0.034h
2^{18}	0.251h	0.194h	0.150h	0.144h	**0.132h**	0.143h	0.136h	0.134h
2^{19}	1.003h	0.775h	0.584h	0.559h	**0.529h**	0.565h	0.539h	0.530h
2^{20}	4.059h	3.129h	2.320h	2.290h	2.127h	2.238h	2.258h	**2.100h**
2^{21}	16.105h	12.297h	9.544h	9.022h	8.741h	9.036h	8.611h	**8.355h**

AMD X2 Results

- 2.5GHz. 64KB L1, 512KB L2 cache. 4GB

n	HB	CO	FLSA_8	FLSA_{16}	FLSA_{32}	FCO_8	FCO_{16}	FCO_{32}
2^{16}	0.017h	0.009h	0.010h	0.010h	0.010h	**0.007h**	**0.007h**	**0.007h**
2^{17}	0.069h	0.037h	0.041h	0.039h	0.038h	0.028h	0.027h	**0.026h**
2^{18}	0.278h	0.149h	0.169h	0.159h	0.156h	0.114h	0.108h	**0.104h**
2^{19}	1.123h	0.597h	0.685h	0.640h	0.624h	0.455h	0.430h	**0.418h**
2^{20}	4.474h	2.389h	2.752h	2.574h	2.498h	1.846h	1.721h	**1.671h**
2^{21}	17.949h	9.442h	11.007h	10.337h	9.950h	7.278h	6.873h	**6.685h**

Cachegrind, L1 and L2 misses

- 64KB L1, 512KB L2 cache

L1 cache misses $\times 10^6$

n	HB	CO	FCO_8	FCO_{16}	FCO_{32}	$FLSA_8$	$FLSA_{16}$	$FLSA_{32}$
2^{16}	1,090	**0.855**	1.682	1.980	2.070	1,293	1,162	1,154
2^{17}	4,639	**1.952**	3.823	4.566	7.49	5,156	4,834	4,626
2^{18}	18,866	**5.916**	11.23	12.29	17.41	20,640	19,4654	18,789
2^{19}	76,654	**19.85**	37.27	39.59	46.55	83,201	77,630	75,355

L2 cache misses $\times 10^6$

2^{16}	604.6	**0.345**	1.038	1.373	1.711	1,151	1,146	1,150
2^{17}	3,207	**0.594**	1.949	2.49	5.2	4,575	4,579	4,581
2^{18}	16,517	**1.312**	4.385	5.067	9.373	18,741	18,303	18,290
2^{19}	71,629	**3.416**	11.689	15.12	18.792	82,131	75,219	73,117

The Problem
- What we have done
- Dynamic Programming with Local Dependencies
- Longest Common Subsequence
- Standard Algorithms

Re-computations and I/O's
- The FastLSA Algorithm
- The CO Algorithm

The FastCO Algorithm

Experimental Results

Round Off

- Two ideas taken from each their algorithm were used to create an algorithm theoretically better than both of them.
- Experiments showed FastCO performing equally good or better across all three sample architectures.
- Cache efficiency once again shown to be important when dealing with large data.

Fast and Cache-Oblivious Dynamic Programming with Local Dependencies

Philip Bille[1], Morten Stöckel[2]
[1] *Technical University of Denmark*
[2] *IT University of Copenhagen*

March 9 2012