Efficient Longest Common Subsequence Computation using Bulk-Synchronous Parallelism

Peter Krusche

Department of Computer Science
University of Warwick

June 2006

Outline

1. Introduction
   - Motivation
   - The BSP Model

2. Problem Definition and Algorithms
   - The Standard Algorithm
   - Standard Algorithm
   - Bit-Parallel Algorithm
   - The Parallel Algorithm

3. Experiments
   - Experiment Setup
   - Predictions
   - Speedup

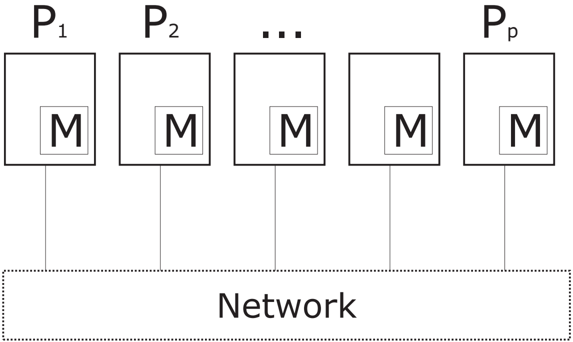
Motivation

Computing the (Length of the) Longest Common Subsequence is representative of a class of dynamic programming algorithms for string comparison. Hence, we want to

- Start with a fast sequential algorithm.
- Examine the suitability of BSP as a programming model for such problems.
- Compare different BSP libraries on different systems.
- Examine performance predictability.

The BSP Computer

- *p* identical processor/memory pairs (computing nodes), computation speed *f*
- Arbitrary interconnection network, latency *l*, bandwidth *g*



BSP Programs

- Programs are SPMD
- Execution takes place in *supersteps*
  - Communication may be delayed until the end of the superstep
  - Separates communication and computation
- Cost/Running time Formula :

$$T = f \cdot W + g \cdot H + l \cdot S$$

BSP Programs

- Programs are SPMD
- Execution takes place in *supersteps*
    - Communication may be delayed until the end of the superstep
    - Separates communication and computation
- Cost/Running time Formula :

$$T = f \cdot W + g \cdot H + l \cdot S$$

BSP Programming

‘BSP-style’ programming using a conventional communications library (MPI/Cray shmem/...)

- Barrier synchronizations for creating superstep structure
- Message passing or remote memory access for communication

Using a specialized library (The Oxford BSP Toolset/PUB/CGMlib/...)

- Optimized barrier synchronization functions and message routing
- Higher level of abstraction / nicer looking code.

BSP Programming

‘BSP-style’ programming using a conventional communications library (MPI/Cray shmem/...)

- Barrier synchronizations for creating superstep structure
- Message passing or remote memory access for communication

Using a specialized library (The Oxford BSP Toolset/PUB/CGMlib/...)

- Optimized barrier synchronization functions and message routing
- Higher level of abstraction / nicer looking code.

Previous Work

- Daniel S. Hirschberg.

  A linear space algorithm for computing maximal common subsequences.

  Communications of the ACM, 18(6):341–343, 1975.

- Maxime Crochemore, Costas S. Iliopoulos, Yoan J. Pinzon, and James F. Reid.

  A fast and practical bit-vector algorithm for the Longest Common Subsequence problem.

  Information Processing Letters, 80(6):279–285, 2001.

- C. E. R. Alves, E. N. Cáceres, and F. Dehne.

  Parallel dynamic programming for solving the string editing problem on a CGM/BSP.

  In *Proceedings of the 14th Annual ACM symposium on Parallel Algorithms and Architectures*, pp. 275–281, 2002.

Previous Work

- Daniel S. Hirschberg.

  A linear space algorithm for computing maximal common subsequences.

  Communications of the ACM, 18(6):341–343, 1975.

- Maxime Crochemore, Costas S. Iliopoulos, Yoan J. Pinzon, and James F. Reid.

  A fast and practical bit-vector algorithm for the Longest Common Subsequence problem.

  Information Processing Letters, 80(6):279–285, 2001.

- C. E. R. Alves, E. N. Cáceres, and F. Dehne.

  Parallel dynamic programming for solving the string editing problem on a CGM/BSP.

  In *Proceedings of the 14th Annual ACM symposium on Parallel Algorithms and Architectures*, pp. 275–281, 2002.

Previous Work

Daniel S. Hirschberg.

A linear space algorithm for computing maximal common subsequences.

Communications of the ACM, 18(6):341–343, 1975.

Maxime Crochemore, Costas S. Iliopoulos, Yoan J. Pinzon, and James F. Reid.

A fast and practical bit-vector algorithm for the Longest Common Subsequence problem.

Information Processing Letters, 80(6):279–285, 2001.

C. E. R. Alves, E. N. Cáceres, and F. Dehne.

Parallel dynamic programming for solving the string editing problem on a CGM/BSP.

In *Proceedings of the 14th Annual ACM symposium on Parallel Algorithms and Architectures*, pp. 275–281, 2002.

Problem Definition

Definition (Input data)

Let $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ be two strings on an alphabet Σ of constant size.

Definition (Subsequences)

A *Subsequence* U of X: U can be obtained by deleting zero or more elements from X.

Definition (Longest Common Subsequences)

A $LCS\,(X, Y)$ is any string which is subsequence of both X and Y *and* has maximum possible length. Length of these sequences: $LLCS\,(X, Y)$.

Problem Definition

Definition (Input data)

Let $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ be two strings on an alphabet Σ of constant size.

Definition (Subsequences)

A *Subsequence* U of X: U can be obtained by deleting zero or more elements from X.

Definition (Longest Common Subsequences)

A *LCS* (X, Y) is any string which is subsequence of both X and Y *and* has maximum possible length. Length of these sequences: *LLCS* (X, Y).

Problem Definition

Definition (Input data)

Let $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ be two strings on an alphabet Σ of constant size.

Definition (Subsequences)

A *Subsequence* U of X: U can be obtained by deleting zero or more elements from X.

Definition (Longest Common Subsequences)

A *LCS* (X, Y) is any string which is subsequence of both X and Y *and* has maximum possible length. Length of these sequences: *LLCS* (X, Y).

The Dynamic Programming Matrix

Definition (Matrix $L_{0\ldots m,0\ldots n}$)

$$L_{i,j} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ L_{i-1,j-1} + 1 & \text{if } x_i = y_j, \\ max(L_{i-1,j}, L_{i,j-1}) & \text{if } x_i \neq y_j \ . \end{cases}$$

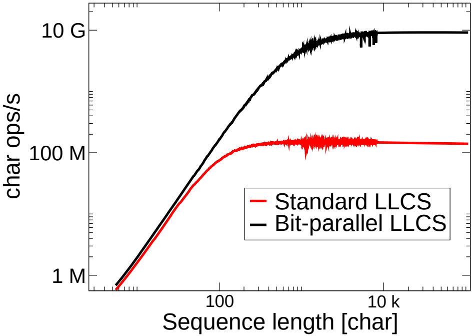
Theorem (Hirschberg, ’75)

$L_{i,j} = LLCS(x_1x_2\ldots x_i, y_1y_2\ldots y_j)$. The values in this matrix can be computed in $O(mn)$ time and space.

Bit-Parallel Algorithm

Bit-parallel computation has same asymptotic complexity but processes ω entries of L in parallel (ω : machine word size).

Example (this leads to substantial speedup)



How does it work?

- $\Delta L(i,j) = L(i,j) - L(i-1,j) \in \{0,1\}$
- $\Delta L(i,j)$ is computed columnwise using machine-word parallel operations :

$$\begin{aligned}\sim \Delta L(i,j) \quad &\leftarrow \quad (\sim \Delta L(i,j) + \\ &\qquad (\sim \Delta L(i,j-1) \ \mathbf{and}\ M(x_j))) \\ &\qquad \mathbf{or}\ (\sim \Delta L(i,j-1)\ \mathbf{and}(\sim M(x_j)))\end{aligned}$$

- M maps characters to bit strings of length m,

$$M(\sigma \in \Sigma)_i \quad = \quad 1 \Leftrightarrow x_i = \sigma$$

How does it work?

- $\Delta L(i,j) = L(i,j) - L(i-1,j) \in \{0,1\}$
- $\Delta L(i,j)$ is computed columnwise using machine-word parallel operations :

$$\begin{aligned} \sim \Delta L(i,j) \quad \leftarrow \quad & (\sim \Delta L(i,j) + \\ & (\sim \Delta L(i,j-1) \ \mathbf{and}\ M(x_j))) \\ & \mathbf{or}\ (\sim \Delta L(i,j-1)\ \mathbf{and} (\sim M(x_j))) \end{aligned}$$

- M maps characters to bit strings of length m,

$$M(\sigma \in \Sigma)_i \quad = \quad 1 \Leftrightarrow x_i = \sigma$$

How does it work?

- $\Delta L(i,j) = L(i,j) - L(i-1,j) \in \{0,1\}$
- $\Delta L(i,j)$ is computed columnwise using machine-word parallel operations :

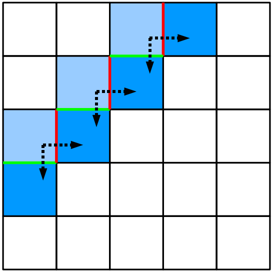
$$\begin{aligned}\sim \Delta L(i,j) \quad &\leftarrow \quad (\sim \Delta L(i,j) + \\ &\qquad (\sim \Delta L(i,j-1) \textbf{ and } M(x_j))) \\ &\qquad \textbf{or } (\sim \Delta L(i,j-1) \textbf{ and}(\sim M(x_j)))\end{aligned}$$

- M maps characters to bit strings of length m,

$$M(\sigma \in \Sigma)_i \quad = \quad 1 \Leftrightarrow x_i = \sigma$$

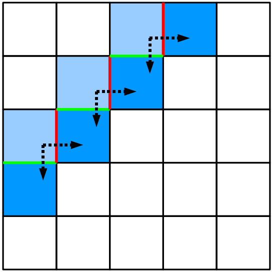
The Parallel Algorithm

- Matrix L is partitioned into a grid of rectangular blocks of size $(m/G) \times (n/G)$ (G : grid size)
- Blocks in a wavefront can be processed in parallel
- Assumptions:
  - Strings of equal length $m = n$
  - Ratio $\alpha = \frac{G}{p}$ is an integer



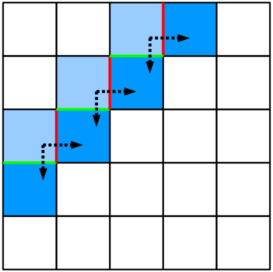
The Parallel Algorithm

- Matrix L is partitioned into a grid of rectangular blocks of size $(m/G) \times (n/G)$ (G : grid size)
- Blocks in a wavefront can be processed in parallel
- Assumptions:
  - Strings of equal length $m = n$
  - Ratio $\alpha = \frac{G}{p}$ is an integer



The Parallel Algorithm

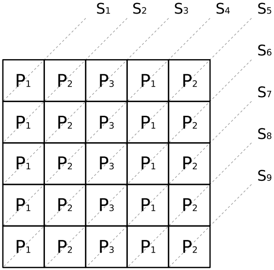
- Matrix L is partitioned into a grid of rectangular blocks of size $(m/G) \times (n/G)$ (G : grid size)
- Blocks in a wavefront can be processed in parallel
- Assumptions:
  - Strings of equal length $m = n$
  - Ratio $\alpha = \frac{G}{p}$ is an integer



Parallel Cost Model

- When $G > p$, there can be multiple stages for one block-wavefront
- Running time

$$\begin{aligned} T(\alpha) &= f \cdot (p\alpha(\alpha+1) - \alpha) \cdot \left\lceil \frac{n}{\alpha p} \right\rceil^2 \\ &+ g \cdot \alpha(\alpha p - 1) \left\lceil \frac{n}{\alpha p} \right\rceil \\ &+ l \cdot (2\alpha p - 1) \cdot \alpha \end{aligned}$$



Experiments: Systems Used

- **`aracari:`** IBM cluster, 2-way SMP Pentium3 1.4 GHz nodes (Myrinet 2000)
- **`argus:`** Linux cluster, 2-way SMP Pentium4 Xeon 2.6 GHz nodes (100Mbit Ethernet)
- **`skua:`** SGI Altix shared memory machine, Itanium-2 1.6 GHz processors

Experimental values of f and f'

Simple Algorithm (f)

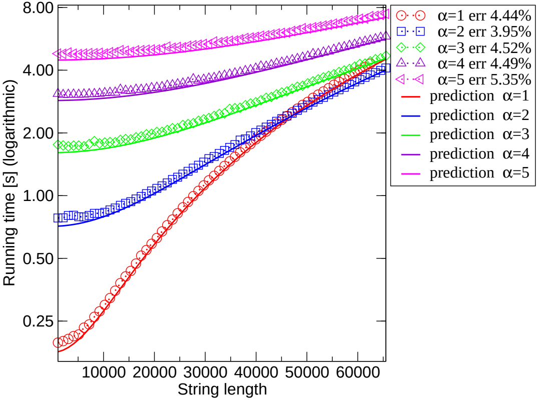
`skua`	0.008 ns/op	130 M op/s
`argus`	0.016 ns/op	61 M op/s
`aracari`	0.012 ns/op	86 M op/s

Bit-Parallel Algorithm (f')

`skua`	0.00022 ns/op	4.5 G op/s
`argus`	0.00034 ns/op	2.9 G op/s
`aracari`	0.00055 ns/op	1.8 G op/s

Prediction Results

Good Results (LLCS) . . . (e.g. `aracari` MPI, 32 processors)

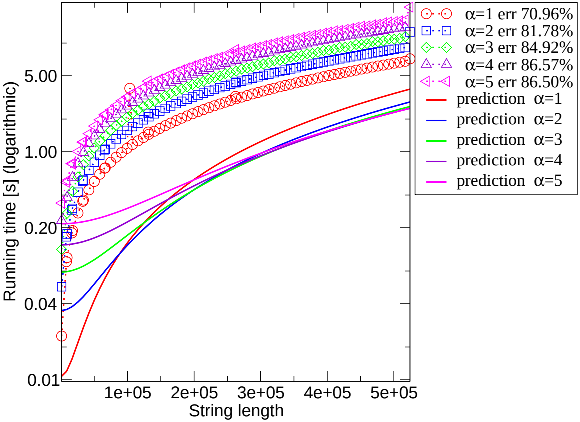


Good Predictions

- . . . on all distributed memory systems, using both bit-parallel and standard algorithm
- . . . on the shared memory system only for larger problem sizes, and for the standard algorithm

Prediction Results

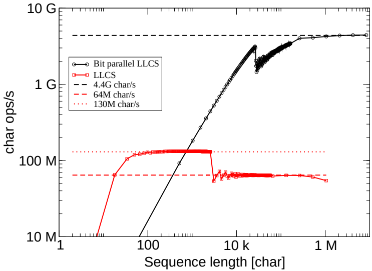
Not so good ones. . . (Oxtool, skua, 32 processors)



What happened?

Cache size effects prevent prediction of computation time. . .

Sequential computation performance on **skua**



Other Problems when Predicting Performance

- Setup costs only covered by parameter l
  - ⇒ difficult to measure
  - ⇒ Problems when communication size is small
- PUB has performance break-in when communication size reaches a certain value
- Busy communication network can create ‘spikes’

Speedup for the Bit-Parallel Version

- Speedup lower than for the standard version
- However, overall running times for same problem sizes are shorter
- Can only expect parallel speedup for larger problem sizes
- Latency is problematic, as computation times are low.

Result Summary

	Oxtool	**PUB**	**MPI**
Shared memory (skua)			
LLCS (standard)	●●●	●	●●
LLCS (bit-parallel)	●●	●●●	●
Distributed memory, Ethernet (argus)			
LLCS (standard)	●●●	●●	●
LLCS (bit-parallel)	●●●	●●	●
Distributed memory, Myrinet (aracari)			
LLCS (standard)	●●●	●●	●
LLCS (bit-parallel)	●●	●●○	●

Result Summary

	Oxtool	**PUB**	**MPI**
Shared memory (skua)			
LLCS (standard)	●●●	●	●●
LLCS (bit-parallel)	●●	●●●	●
Distributed memory, Ethernet (argus)			
LLCS (standard)	●●●	●●	●
LLCS (bit-parallel)	●●●	●●	●
Distributed memory, Myrinet (aracari)			
LLCS (standard)	●●●	●●	●
LLCS (bit-parallel)	●●	●●○	●

Result Summary

	Oxtool	**PUB**	**MPI**
Shared memory (skua)			
LLCS (standard)	●●●	●	●●
LLCS (bit-parallel)	●●	●●●	●
Distributed memory, Ethernet (argus)			
LLCS (standard)	●●●	●●	●
LLCS (bit-parallel)	●●●	●●	●
Distributed memory, Myrinet (aracari)			
LLCS (standard)	●●●	●●	●
LLCS (bit-parallel)	●●	●●○	●

Summary

- BSP algorithms are efficient for dynamic programming.
- Implementations benefit from a low latency implementation (Oxtool/PUB)
- Very good predictability

Outlook

Technical improvements

- Different modeling of bandwidth allows better predictions
- Using assembly can double bit-parallel performance
- Lower latency possible by using subgroup synchronization

Algorithmic improvements

- Extraction of LCS possible, using post processing step or other algorithm
- Implementation of all-substrings LLCS (which has many applications)
- Design and study of subquadratic algorithms

Outlook

Technical improvements

- Different modeling of bandwidth allows better predictions
- Using assembly can double bit-parallel performance
- Lower latency possible by using subgroup synchronization

Algorithmic improvements

- Extraction of LCS possible, using post processing step or other algorithm
- Implementation of all-substrings LLCS (which has many applications)
- Design and study of subquadratic algorithms