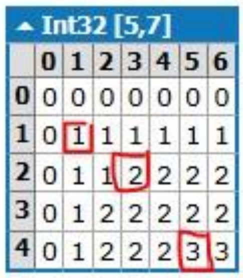
Assignment #1 – Functional Programming for Parallel Processing

Introduction

This assignment aims to consolidate your understanding of several topics taught in this course, such as applied functional programming and parallel /distributed processing. You are required to build and evaluate several versions of a simple but fundamental algorithm, Longest Common Subsequence (LCS). This algorithm is very important, e.g. in bio-informatics, and efficient parallel versions are still sought and proposed.

For example, consider strings *s1*="acba" and *s2*="abcdad". The LCS length is 3, given by the subsequence "aca" (or "aba"), as described by the following snapshots (*s1* is the NS axis and *s2* the WE axis):

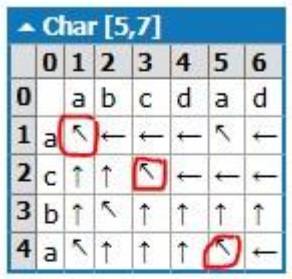
C



Int32 [5,7]

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	$\boxed{1}$	1	1	1	1	1
2	0	1	1	$\boxed{2}$	2	2	2
3	0	1	2	2	2	2	2
4	0	1	2	2	2	$\boxed{3}$	3

P



Char [5,7]

	0	1	2	3	4	5	6
0		a	b	c	d	a	d
1	a	↖	←	←	←	↖	←
2	c	↑	↑	↖	←	←	←
3	b	↑	↖	↑	↑	↑	↑
4	a	↖	↑	↑	↑	↖	←

For our purpose, we only require the LCS length, not the corresponding longest sunsequence(s). So, you won't need any backtrace array like P above.

You are required to design and develop two command-line programs and write one evaluation report.

Program Requirements

You will develop two similar programs, A1F.EXE in F#, and A1C.EXE in C#. Each of these programs will run three different versions:

1. Sequential implementation (SEQ), used as a baseline to evaluate the parallel speedup.
2. Actor implementation (ACT), used to evaluate the parallel speedup for the Actor model.

   For F#, you will use its own MailboxProcessor, which provides unbounded queues. For C#, you will use the channel library with unbounded queues.
3. CSP implementation (CSP), used to evaluate the parallel speedup for the CSP model.

   For F#, you will use the Hopac library, with size 0 buffers. For C#, you will use the channel library with size 1 buffers.

Your programs will be run and expected to show parallel speedups even on a single lab machine with several cores. However, your parallel Actors design should be amenable to run on a distributed cluster. You can read more about such topics in the Berkeley patterns for parallel computing, esp. the structured grid dwarf.

To ensure a fair playing field, you will base your design on the simple dynamic programming (DP) algorithm. However, some of the strings could be quite large. For space efficiency, you will need to consider replacing DP arrays by diagonal wave-fronts, instead of the textbook array approach. For parallel efficiency, you will need to consider a courser granularity, by splitting the virtual DP array into a grid of subarrays, defined by the intersection of horizontal and vertical bands. Finally, you will probably need to streamline your design by using sentinel values for the arrays/subarrays borders.

Figures 1, 2, and 3 schematically illustrate the progression of the diagonals wave-fronts, Fig. 1 for sequential processing, Figs. 2 and 3 for the parallel case with a courser granularity defined by horizontal and vertical bands. Green areas indicate active cells, currently on a diagonal wave-front. Yellow areas indicate already processed cells. White areas indicate unprocessed cells. Red lines indicate boundaries between bands.

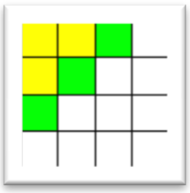


Figure 1: SEQ

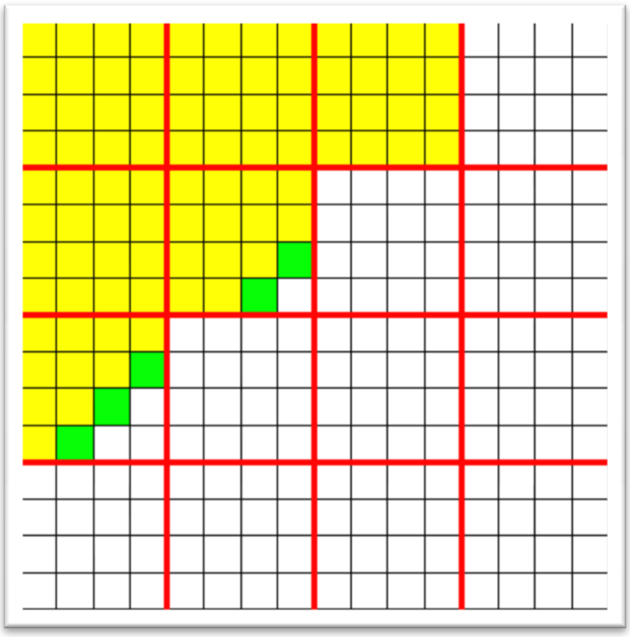


Figure 2: PAR (sync evolution)

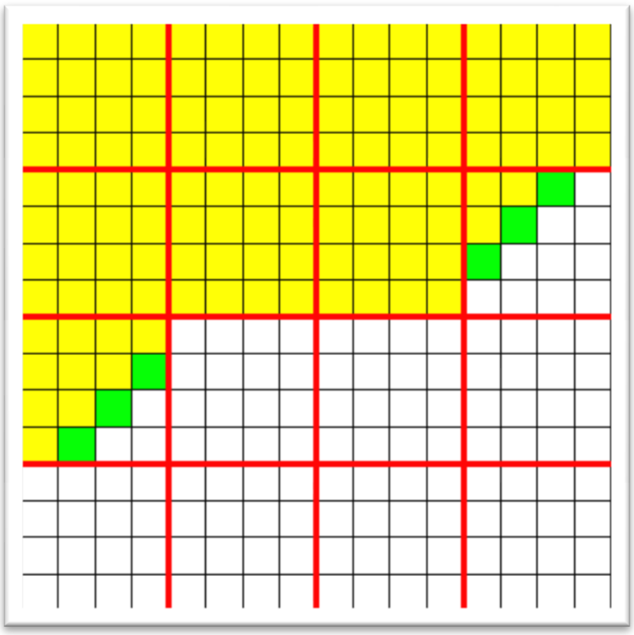


Figure 3: PAR (async evolution)

Running the Programs

Please strictly follow these conventions (for consistency with the marking tools)!

1. Invoking the F# program – one of the following command lines:

```
A1F.EXE  /S1:path-to-file1  /S2:path-to-file2  /SEQ

A1F.EXE  /S1:path-to-file1  /S2:path-to-file2  /ACT:b1,b2,t

A1F.EXE  /S1:path-to-file1  /S2:path-to-file2  /CSP:b1,b2,t
```

Where

- **file1** and **file2** are .TXT files using the UTF8 encoding and containing the two input strings, of lengths **n1**, **n2**, respectively, s.t.: 1 <= **n1**, **n2** <= 100000

- **b1** and **b2** are strictly positive numbers (1 <= **b1**, **b2** <= 100, **b1** <= **n1**, **b2** <= **n2**) that indicate the required number of horizontal and vertical bands, respectively ○ if the lengths of the input strings are not exactly divisible by b1, b2, then the bands' sizes need to be adjusted in a balanced way, with larger bands first ○ **n1** = 6, **b1** = 3 => 3 horizontal bands of lengths [2; 2; 2] ○ **n1** = 7, **b1** = 3 => 3 horizontal bands of lengths [3; 2; 2] ○ **n1** = 8, **b1** = 3 => 3 horizontal bands of lengths [3; 3; 2] ○ **n1** = 9, **b1** = 3 => 3 horizontal bands of lengths [3; 3; 3] ○ **n1** = 3, **b1** = 3 => 3 horizontal bands of lengths [1; 1; 1] ○ **n1** = 7, **b1** = 1 => 1 horizontal band of length [7]

- **t**=0,1 is a trace parameter, where **t**=1 requires the printing of partial LCS's to the standard output

2. Standard output: see appendix!

3. Error output: may contain any additional traces which you want. However, these may affect the runtime, so you should disable these for the final submission.

4. Invoking the C# program – as above, just replace A1F by A1C.

Report (15%)

The report must contain a brief literature overview and empirical evaluations of your own results. You will evaluate the relative performance of the sequential versions vs. the message-based parallel versions, using tables and plots.

Your report should be structured and written like a research article, of (normally) up to 6 pages. It should contain: title, author, abstract, introduction, good and updated literature overview, brief descriptions of your algorithms, clear empirical evaluations of the runtimes of your implementations, conclusions, and bibliography.

As indicated by the highlights, the focus should be on the literature overview and your own empirical evaluation. Note that you can start working on your report from day one: only the empirical evaluation and conclusions need to wait for your results.

Suggestion to use a good and "standard" format for scientific publications, such as the LNCS article style, either in LaTeX (preferable) or Word.

Deliverables and Submission

Submit electronically, to the ADB web dropbox, an upi.7z archive containing an upi folder with:

○ your report as PDF; ○ your .FS F# source file(s); ○ a _COMPILE.FS.BAT batch file to

compile your F# file(s) into an A1F.EXE executable.

○ your .CS C# source file(s); ○ a _COMPILE.CS.BAT batch file to compile your C# file(s)

into an A1C.EXE executable.

Please do not submit anything else: no VS or VS Code solutions/projects, no copie of the neede libraries, such as: Hopac.*.dll, System.Threading.Channels.dll, … (we already have copies of these)

Please keep your electronic dropbox receipt and follow the instructions given in our web pages, including our policy on plagiarism.

<u>Note</u>

One of the programs must be written in F#. For the other program, we highly recommend C#. However, as discussed in the class, you could also use another language of your choice, provided that some conditions are met -- most importantly:

1. The replacement language must offer similar capabilities, i.e. provide support for actors and channels (natively or via third party libraries).

2. The marking team has access to the required compilers and runtimes, or these can be easily installed on a Windows machine (sorry, we need to care about all parties).

Possible candidates: Node.js, Python, Java (Akka?), Kotlin, Golang…

Anyway, please let us know of your intentions and do not proceed without our green light. Thanks.

Assessment criteria ○ **Functional correctness**: [**black**

box style, mostly] ○ **Performance medium**: [**black**

box style]

- ○ **Special cases**: [**black box** style] string sizes not exactly divisible by band numbers ; incorrect command lines… all exceptions must be caught and error messages must pe printed
- ○ **Report**: see the description above; ensure that your report looks readable and provides convincing discussions and plots, with results which are reproducible on a typical lab machine (illustrate it with your own runtime results, even if the performance is not great)

Appendix - Samples

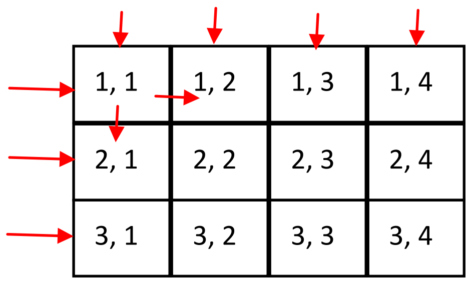
LCS-Demo.xls: A simple intuitive Excel demo (recall that Excel is a functional language)

GridDiag: A sample which demonstrates a diagonal wave sweeping over a rectangular array (the 0 sentinel values are not shown).

You may actually need to recycle three diagonals. For example, initially: *d1*=[1; 0; 0], *d2*=[2; 2; 0]; *d2*=[3; 3; 3]. Then, you could recycle *d1* to play the role of *d4*=[4; 4; 4]; *d2* for *d5*=[0; 5; 5]; *d3* for *d6*=[0; 0; 6].

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

GridActorsModel2, **GridActorsModel1:** Two versions a simple rectangular grid dataflow, with messages running from N to S and W to E. Messages contain grid coordinates. The process starts from the NW corner and ends in the SW corner. These samples are implemented using F#'s own Mailbox actors.



Other samples and required libraries will be posted in the assignment folder

Appendix - Required standard output

/SEQ: regardless of the value of the trace parameter **t**, one single line for the result, containing three numbers separated by single spaces:

1 1 LCS-length

This format ensures some consistency with the parallel case (kind of one single task)

/ACT, **/HOP**: if **t**=1, then **b1*b2** lines - one line for each subarray (intersection of a horizontal and a vertical band), containing three numbers, separated by single spaces:

1) the 0-based row index of the subarray
2) the 0-based column index of the subarray
3) the value of the subarray's SW corner cell, i.e. the partial LCS length so far

The **b1*b2** lines must be printed in the completion order of the associated tasks

Otherwise, if **t**=0, only the last line will be printed, with the final LCS value:

(b1-1) (b2-1) LCS-length

The following sample illustrates these rules. Consider the following strings and partitioning bands:

s1, **s2** = "abcd", "abbdecccbd" // **n1**, **n2** = 4, 10

b1, **b2** = 2, 3 // 6 subarrays, with indexes: (0,0), (0,1), (0,2), (1,0), (1,1), (1,2)

The computation determines the following virtual array, where the background highlights indicate the 6 subarrays, and the red numbers indicate the partial results:

		1	2	3	4	5	6	7	8	9	10
		a	b	b	d	e	c	c	c	b	d
1	a	1	1	1	1	1	1	1	1	1	1
2	b	1	2	2	**2**	2	2	**2**	2	2	**2**
3	c	1	2	2	2	2	3	3	3	3	3
4	d	1	2	2	**3**	3	3	**3**	3	3	**4**

The completion order is not deterministic, but the results should be confluent, i.e. match the model, after sorting these in the same way, e.g. on line/column order. Possible outputs for **/ACT and /CSP**, if **t=1**

0 0 2 0 1 2 1 0 3 0 2 2 1 1 3 1 2 4	0 0 2 0 1 2 1 0 3 1 1 3 0 2 2 1 2 4	0 0 2 1 0 3 0 1 2 0 2 2 1 1 3 1 2 4	0 0 2 1 0 3 0 1 2 1 1 3 0 2 2 1 2 4	0 0 2 0 1 2 0 2 2 1 0 3 1 1 3 1 2 4