Assignment #1 v2

Functional Programming for Parallel and Distributed 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 extensively evaluate several parallel/distributed versions of a simple but fundamental dynamic programming algorithm, Checkerboard. For our purpose, we only require the max path cost between the first row and last row of a matrix of non-negative integers, without tracing an optimal path which achieves this.

More details about this problem (note that we use max instead of min):

https://en.wikipedia.org/wiki/Dynamic_programming#Checkerboard

Sample initial costs matrix **M**, here with **N1**=4 rows and **N2**=4 columns.

M	0	1	2	3
0	4	2	3	4
1	2	9	2	10
2	15	1	3	0
3	16	92	41	44

Sample max path costs matrix **R**, computed by the dynamic programming algorithm:

R	0	1	2	3
0	4	2	3	4
1	6	13	6	14
2	28	14	17	14
3	44	**120**	58	61

The program is expected to run under the following constraints, where **K** is the number of partition ranges (as discussed later):

$$1 \le N1 \le 10{,}000;\ 1 \le K \le N2 \le 10{,}000{,}000;\ 1 \le N1 \times N2 \le 10{,}000{,}000{,}000;\ 0 \le M[i1,i2],\ \forall\ i1, i2.$$

As scenarios with many long rows are also expected, your implementation should be space aware. For example, in our case, the max path cost matrix can be computed with just two alternating rows (so you don't need to reserve space for the whole R matrix). However, you may still use sentinels, as suggested by the following table, to trade a minor amount of space against speed benefits:

R	0	1	2	3	4	5
0	0	4	2	3	4	0
1	0	6	13	6	14	0
2	0	28	14	17	14	0
3	0	44	**120**	58	61	0

Overview

The assignment has two required parts and one optional part (possible bonus).

Part 1. Several basic versions, all running inside the same process.

1.1 Sequential implementation (/SEQ), which will also be the baseline for evaluating the parallel speedup.

1.2 Task based naive parallel implementation (/PAR-NAIVE), a multi-phase implementation where each phase uses a synchronous fork-join design, via .NET Parallel.For, without any explicit partitioning.

1.3 Task based parallel implementation with explicit range partitioning (/PAR-RANGE), a multi-phase implementation where each phase uses a synchronous fork-join design, via .NET Parallel.For.

1.4 Async based parallel implementation with explicit range partitioning (/ASYNC-RANGE), a multi-phase implementation where each phase uses a synchronous fork-join design, via F# Async.Parallel and Async.RunSynchronously.

1.5 Actor based parallel implementation with explicit range partitioning (/MAILBOX-RANGE), via F# MailboxProcessor.

1.6 Actor based parallel implementation with explicit range partitioning (/AKKA-RANGE), via .NET Akka.Net.

When a range is required, you need to create it, as required, either with

(i) the .NET range partitioner Partitioner.Create(int,int); or

(ii) our partitioning function, designed to ensure the given number of chunks, with sizes equal or differing by at most one (the .NET Partitioner does not ensure this).

Part 2. A report with your empirical evaluations and reflections (more details below, in section **Report**)

Part 3. Optional bonus!

You are encouraged to research related topics not directly required by this assignment. You may get a discretionary small bonus, up +10%, but you will also need to submit a report appendix with 1-3 pages. Such scenarios are entirely your choice, but please advise us before attempting a bonus. Suggestions:

- One actor based distributed version, where actors reside on different processes (on the same machine or on different machines) or on a cloud cluster. Use Akka.Remote or Akka.Cluster.
- One cloud version, using one of the high-level MBrace cloud monad: http://fsharp.org/guides/cloud/
- One GPU version, using one of the high-level F# libraries: http://fsharp.org/use/gpu/

Development Tools

For all implementations, use F# on Windows, as available in the COMPSCI labs. You can develop your program with Visual Studio, if you wish so. However, your submission must not include any VS specific folders or files. For consistency with the marking tools, your submission must be directly compilable by FSC and the resulting executable must be called A1.EXE (more details below, under Deliverables and Submission)

Running the Program

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

1. **Invoking the program** (**Part 1**):

```
A1.EXE  fname  /alg  k  v
```

Where

- **fname** is the path to a text-file describing your input. (more details below, under Input methods)
- **/alg** indicates the algorithm version, i.e. one of /SEQ, /PAR-NAÏVE, /PAR-RANGE, /ASYNC-RANGE, /MAILBOX-RANGE, /AKKA-RANGE, /ACT734-RANGE.
- **k** indicates the range partitioner: k=0, to use the .NET Partitioner.Create(int,int), or k>0, to create exactly k ranges, all of equal size or differing by at most one.
- **v** indicates the required verbosity: v=0, to avoid any printing during the innermost loops (for time performance testing), or v=1, to print progress details, as required by the output format (for accuracy testing).

2. **Input method**. You obtain the initial cost matrix, **M**, by calling the given function **Data.getData**, with one string argument, **fname**, as illustrated in the skeleton (see the Appendix).

```
let M = Data.getData fname    // string -> int[][]
```

Matrix **M** is a jagged representation of a 2D array, with same length rows. File **fname** offers a crisp text-based representation of matrix **M**. Function **getData** processes this and returns the expected cost array.

Warning! You must use the module **Data** as given, without modifying it. To ensure this, the marker will replace your submitted **Data.fs** file by our copy.

Also, you should not modify matrix **M**. For space efficiency, rows with identical contents are given as pointers to the same row object. Thus, modifying one row may actually modify a large number of rows.

3. **Output format**. The output contains mandatory text, optional text and blank lines.

3.1 Optional lines are prefixed with three dots and a space "… ". These lines may contain useful additional information, but are not required.

3.2 If **verbose** = 0, then, for each algorithm, you need to print three mandatory lines, each line prefixed by three dollar signs and a space "$$$ ":

$$$ algorithm-name …, just before the algorithm starts

$$$ duration: ddd ms, immediately after the algorithm completes

$$$ ddd …, max path sum determined by the algorithm

3.3 If **verbose** = 1, then, for each algorithm, you also need to trace the evaluation of the max path cost matrix, **R**. The following format is mandatory, as the accuracy of this matrix will be determined by tools:

row-no chunk-start-index values

This format is especially important to meaningfully trace the asynchronous actor tasks, but, for uniformity, please use this for all algorithms, even in the synchronous case.

3.4 **For example**, consider the scenario given on page 1.

Sample verbose output from the **sequential** algorithm – here one chunk per row, and all chunks start at 0:

$$$ sequential (no range)

0 0 4 2 3 4

1 0 6 13 6 14

2 0 28 14 17 14

3 0 44 120 58 61

$$$ duration: 26 ms

$$$ 120 max_sequential

As an **exception** to the general rule, for the **parallel naïve** algorithm use the same printing layout as above, i.e. print whole rows after each step (after all tasks terminate). While interesting, printing size 1 chunks may be a wasteful overkill, in this particular case…

Sample verbose output from the **actor** mailbox algorithm, for a partition with 2 chunks (0,2), (2,4) – here the chunks are interleaved, according to the relative speed of the actors involved.

```
$$$ mailbox_range: 2
0 0 4 2
0 2 3 4
1 0 6 13
1 2 6 14
2 0 28 14
3 0 44 120
2 2 17 14
3 2 58 61
$$$ duration: 70 ms
$$$ 120 max_mailbox_range
```

The checking tool will sort the traces and then reassemble the rows, obtaining the same traces as in the sequential case:

```
0 0 4 2
0 2 3 4
1 0 6 13
1 2 6 14
2 0 28 14
2 2 17 14
3 0 44 120
3 2 58 61
```

3.5 If **verbose** = 0, then traces are not printed - in the above samples, only the $$$ rows will appear.

4. The **algorithm duration** must be tightly determined with “our standard” **duration** function, as given in the skeleton (see the Appendix).

Report

Your report should be structured like an experimental research article. It should contain: title, author, abstract, introduction, literature overview, brief descriptions of your algorithms and implementations, clear and extensive empirical evaluations of the runtimes of your implementations (with tables and plots), reflections and conclusions, bibliography.

Suggestion to use a good and "standard" format for scientific publications, such as the LNCS article style, using either LaTeX (preferable) or Word. The report should contain 4-5 pages, but not more than 6 pages, so it must be crisp but convincing.

The focus should be on your own empirical evaluations and your associated interpretations and reflections. Suggested discussions, based on your empirical evidence on medium/large/very large data:

- Contrast naïve parallelisation vs. range based parallelisation. What is the optimum number of partition ranges? How close is the default partitioning to this optimum?

- Compare the Parallel.For parallelism vs. async based parallelism? Can you efficiently use F# or .NET async primitives to achieve parallelism? Many professionals do not seem aware of this…

- Make a decision table indicating the merits of each of the two considered actor based approaches. Are there any significant differences? When would you use one over the others?

Deliverables and Submission

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

- your report as PDF;
- your F# source file(s) and your app config file (if required);
- a _COMPILE.BAT batch file to compile your file(s) into an A1.EXE executable;
- you can include the needed Akka.NET libraries, but otherwise the marker will include these.

Please keep your electronic dropbox receipt and follow the instructions given in our Assignments web page (please read these carefully, including our policy on plagiarism):

http://www.cs.auckland.ac.nz/courses/compsci734s1c/assignments/

Assessment

The assessment is based first on accuracy and then on performance. We'll offer an accuracy testing script (the same as used by the marker) and we'll advertise the criteria used for performance (absolute timings and relative speedups).

Deadline

Monday 8 May 2017, 18:00! Please do not leave it for the last minute. Remember that you can resubmit and, by default, we only consider your last submission.

After this deadline, submissions will be still accepted for 6 more days, with gradually increasing penalties: for the first 3 days 0.25% for each hour late, then 0.5% for each additional hour late.

After this, no more submissions are accepted!

Appendix

1. **A1_checker_skeleton** draft project. Includes: a basic sequential algorithm, a naïve parallel algorithm, several test files and the required Data.fs input method. (published)

2. **A1_parallel_patterns_1** draft project. Includes hints for the /SEQ, /PAR-NAÏVE, /PAR-RANGE, /ASYNC-RANGE algorithms. (published)

3. **A1_parallel_patterns_2** draft project. A similar but simplified problem, with hints for the /MAILBOX-RANGE algorithms. (published)

4. **A1_checker_tut**: snippets discussed in the tutorials, including the code of our partitioner (published)

5. **A1_Verifier**: accuracy testing scripts (self-assessing kit, published)

6. **A1_Performance**: desired performance targets, for a 4 cores tablet and an 8 cores lab machine (published)