Parallel Programming Concepts
WS 2012 / 2013

Assignment 3 (Submission deadline: Feb 7th 2013, 23:59 CET)

General Rules

The assignment solutions have to be submitted at:

https://www.dcl.hpi.uni-potsdam.de/submit/

Our automated submission system is intended to give you feedback about the validity of your file upload. A submission is considered as accepted if the following rules are fulfilled:

- You did not miss the deadline.
- Your file upload can be decompressed with a zip / tar decompression tool.
- Your submitted solution contains only the source code files and a Makefile for Linux 2.6 64-bit. Please leave out any Git/Mercurial repository clones or SVN/CVS meta-information.
- Your solution can be compiled using the "make" command, without entering a separate subdirectory after decompression.
- Your program runs without expecting any kind of keyboard input or GUI interaction.
- Our assignment-specific validation script accepts your program output/generated files.

If something is wrong, you will be informed via email (console output, error code). Re-uploads of corrected solutions are possible until the end of the deadline.

Assignment 3

The third assignment covers GPU Computing as a special kind of data parallel processing, as well as, the actor model with Scala. The GPU tasks can be implemented using CUDA or OpenCL. (OpenCL allows you to run your code on CPUs as well, which may be helpful for local testing.)

Two out of four tasks must be solved correctly in order to pass the assignment. Documentation should be done inside the source code.

Students can submit solutions either **alone or as team of max 2 persons**.

Hasso Plattner Institute Operating Systems and Middleware Group
Frank Feinbube, Dr. Peter Tröger, Prof. Dr. Andreas Polze

Task 3.1: Conway's Game of Life

Develop an OpenCL-based or CUDA-based implementation of Conway's Game of Life[1]. You can use one of the "Hello-World"-Examples from the course home page as a basis for your implementation:

http://www.dcl.hpi.uni-potsdam.de/teaching/parProg/helloOCL.zip
http://www.dcl.hpi.uni-potsdam.de/teaching/parProg/helloCUDA.zip

Input

Your program has to be named as "conway" and take three arguments, an *input file* indicating the initial setup of the world, the *number of rounds* the Game of Life should be played, and the *zombie*-flag indicating the state of the cells outside the game area. The input file is a text file containing only 0s for dead cells and 1s for living cells. Each column of the file represents a column in the world; each line in the file represents a row in the world. The number of columns will always be equal to the number of rows. The third parameter indicates if the cells that are outside of the field are considered dead (=0) or alive (=1). This value is static for the whole runtime of the game.

```
Example: ./conway world.txt 99 0

Content of world.txt:
000
111
000
```

Output

The program must terminate with exit code 0 and produce an output file with the name of "output.txt" in the same directory. This file has to have the same format as the input file.

```
Example content of output.txt (for the input given above):
010
010
010
```

The result with lowest average runtime will be celebrated in the lecture.

Task 3.2: Wator

In this assignment, you should develop an implementation of the Wator[2] simulation game in Scala 2.*[3] The game rules should be implemented as described on the web page. The game field consists of `n x n` grid of cells, modeling an ocean. Each cell can either contain water, a fish, or a shark. The initial distribution of sharks and fish on the grid should be random. The simulation runs in rounds ("generations"). The evaluation order per generation is implementation-specific.

[1] http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
[2] http://de.wikipedia.org/wiki/Wator
[3] http://www.scala-lang.org/downloads

Develop a command-line program in Scala, which implements the simulation. Start with a serial version. The code should iterate over an ocean grid data structure and check each cell for its content and the appropriate activity.

Test you code with fixed initial distributions, e.g. were 1/3 of the ocean space is filled with fish, 1/3 with sharks and 1/3 with water. Step through your simulation with very small sizes, to make sure that the rules are implemented correctly. Measure the generation rate per second with different grid sizes / parameter constants (see "Rules of the Game") and a random initial distribution.

Modify your code so that the simulation parallelizes with Scala[4]. The goal is to maximize the number of generations being computed per second.

In order to coordinate the execution, implement a global barrier for all activities that marks the end of the current simulation generation computation. Measure the generation rate per second with same configurations as for your serial version. What is the performance difference to the serial version with different parameters settings? Think about the way how the simulation semantics change by the parallel implementation. Document you thoughts shorty using the web interface.

Rules of the Game

Fishes and sharks follow specific rules for their activity per simulation round. The grid should be understood as a flattened torus, meaning that the upper border is connected to the lower border, and the left border is connected to the right border. It is therefore not possible to leave the ocean alive.

A fish first checks the surrounding cells in random (!) order, and moves to the first identified free neighbor cell. Every fish has an egg counter that increases by one each simulation round. If a pre-defined number of eggs is reached, a new fish is born on the first identified free neighbor cell, and the egg counter is reset. If no cell is free, no new fish is born, and the egg counter remains the same.

A shark first checks the surrounding cell in random order. If a fish is found on a neighbor cell, the shark moves to this cell and eats the fish. If no food is available, the shark just moves to a free neighbor field. Every shark has a starvation counter, which increases with each round. If a pre-defined constant limit for the starvation counter is reached, the shark dies. New sharks appear under the same model as the fishes.

Both, the fish and the shark egg time limit and the starvation counter limit should be parameters to your application.

Parallelization Strategies

There are different parallelization strategies, for example:

- Each fish, resp. shark is modeled by an actor.
- Each cell is modeled by an actor.
- Groups of cells/animals are modeled by an actor.

[4] www.scala-lang.org/docu/files/ScalaByExample.pdf

Visualization

If you want to, you can visualize your Wator simulation through the Java graphics API, which is directly accessible form Scala code. An imperfect code skeleton with all the necessary Java Swing initialization is provided at:

http://www.dcl.hpi.uni-potsdam.de/teaching/parProg/wator_fragment.scala

Experiment with the synchronization between display rendering loop and simulation computation loop. Consider the problem that Java Swing component updates are only allowed from the original AWT event dispatching thread.

Suggestions and improvements for the provided code skeleton are welcome.

Input

Your program has to be named as "Wator" and take five arguments, an *input file* indicating the initial setup of the world, the *number of rounds* the simulation should run, the egg time limit for the fish, the egg time limit for the sharks, and the starvation time for the sharks. The input file is a text file containing only one of the three letters: w (for water), f (for fish), and s (for a shark. Each column of the file represents a column in the world; each line in the file represents a row in the world. The number of columns will always be equal to the number of rows. (As the game is played on a torus there are no cells outside of the field.)

```
Example: scala -classpath . Wator world.txt 99 10 20 5

Content of world.txt:
wfww
wwsw
wffw
wfwf
```

Output

The program must terminate with exit code 0 and produce an output file with the name of "output.txt" in the same directory. This file has to have the same format as the input file.

```
Example content of output.txt (actual result depends on chance):
wwww
wwww
wwww
wwww
```

Task 3.3: Wator – Crossing the Rubicon

Try to modify your parallelized solution so that no global barrier is needed, while the concept of simulation rounds is kept. This could, for example, being done by letting each cell keep a history of states from earlier simulation time stamps. Rules, file names, parameters, input format, and output format are equal to Task 3.2.

Task 3.4: Wator – Virtus Tentamine Gaudet

Develop an OpenCL-based or CUDA-based implementation of the Wator simulation. You are free to choose a simplified solution for the generation of random numbers, e.g. a pre-computed array/ring buffer of random values. Rules, file names, parameters, input format, and output format are equal to Task 3.2. The program will be called like this:

```
Example execution: ./wator world.txt 99 10 20 5
```