Parallel Programming Concepts
WS 2013 / 2014

Assignment 5 (Submission deadline: Jan 27th 2014, 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 sub-directory after decompression.
- You 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 deadline.

50% 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**.

Assignment 5

This assignment covers Programming for Shared Nothing systems with the Message Passing Interface (MPI) and Scala Actors. For the MPI tasks, your Makefile has to compile your sources with „mpicc". The MPI stack on the test machine is OpenMPI 1.4.3.

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

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

Task 5.1: Heat Map with MPI

Implement a program that simulates heat distribution on a two-dimensional field. The simulation is executed in rounds. The field is divided into equal-sized blocks. Initially some of the blocks are cold (value=0), some other blocks are active hot spots (value=1). The heat from the hot spots then transfers to the neighbor blocks in each of the rounds, which changes their temperature value.

A round is computed as follows:

1. The value of the hot spot fields may be set to 1 again, depending on the live time of the hot spot during a given number of rounds.
2. The new value for each block per round is computed by getting the values of the eight direct neighbor blocks from the last round. The new block value is the average of these values and the own block value from the last round. Blocks on the edges of the field have neighbor blocks outside of the fields, which should be considered to have the value 0.

You have to develop a parallel application for this simulation in C / C++ using MPI. The goal is to minimize the execution time of the complete simulation. Specific optimizations for the given test hardware are not allowed, since we may have to opportunity to run your code on some larger system for the performance comparison.

Input

Your application has to be named “heatmap” and needs to accept five parameters:

- The *width* of the field in number of blocks.
- The *height* of the field in number of blocks.
- The *number of rounds* to be simulated.
- The name of a file (in the same directory) describing the *hotspots.*
- The name of a file (in the same directory) containing *coordinates*. This is the only optional parameter. If it is passed, only the values at the indicated coordinates (starting at (0, 0) in the upper left corner) are to be written to the output file.

```
Example:
mpirun --cpus-per-proc 2 -np 16 heatmap 20 7 17 hotspots.csv

mpirun -np 32 heatmap 20 7 17 hotspots.csv coords.csv
```

The *hotspots* file has the following structure:

- The first line can be ignored.
- All following lines describe one hotspot per line. The first two values indicate the position in the heat field (x, y). The hot spot is active from a start round (inclusive), which is indicated by the third value, to an end round (*exclusive!*), that is indicated by the last value of the line.

```
Example content of hotspots.csv:
x,y,startround,endround
```

```
5,2,0,20
15,5,5,15
```

With such an input file, you have to run a simulation of 17 rounds on a 20x7 field with two hotspots. The first one will be located at the coordinates (5, 2) and will be active from the first round to last round (and beyond). The second hotspot will be located at the coordinates (15, 5) and will be active starting from round 5. Starting from round 15, it will no longer be active. The starting round is inclusive, the final round is exclusive. We start counting at 0. So the first hotspot will be active at round 0,1,2…18,19; the second hotspot will be active at round 5,6,7,…13,14.

```
Example content of coords.csv:
x,y
5,2
10,5
```

With such a coordinate file, you only have to provide the values at the coordinates (5, 2) and (10, 5) as part of the output file.

Output

The program must terminate with exit code 0 and has to produce an output file with the name “output.txt” in the same directory.

If your program was called without a coordinate file, then this file represents the resulting field after simulation termination. The values in the field are encoded in the following way:

- A block with a value larger than 0.9 has to be represented as “*X*”.
- All other values must be increased by 0.09. From the resulting value, the first digit after the decimal point is added to the output picture.

```
Example content of “output.txt” without coordinate file
11112221111111111100
11123432111111111110
11124X42211111111111
11124444211111222111
11122222111112222211
11111211111112232211
01111111111111222111
```

If your program was called with a coordinate file, then this file simply represents the list of exact values requested through the coordinate file.

```
Example content of “output.txt” with coordinate file
1.0
0.030563410733335933
```

The student achieving the lowest average runtime will be announced in the lecture.

Task 5.2: MPI Collective

Implement a parallel MPI program that computes the integer-precision average and the double-precision average of a given set of double values at the same time.

The integer-precision average are to be calculated based on the integer versions of the input values. They are to be computed by reading the double values from an input file, converting them with floor() and casting them to int. The double-precision average can be computed directly using the input values.

Your program is only allowed to use collective MPI operations for the coordination of the parallel computation. MPI_Send and MPI_Receive (and their variations) are disallowed.

Input

Your application has to be named „mpiavg" and hast to accept three parameters:

- The file name of the data file that contains the input numbers.
- The number of MPI ranks to be used for the integer-precision average computation.
- The number of MPI ranks to be used for the double-precision average computation.

The data file is in the current working directory of the program. It contains one double value per line ("4.84637").

We will run your application as follows, with variations in the numerical parameters:

```
Example: mpirun --cpus-per-proc 2 -np 16 mpiavg data.txt 7 9
```

```
Example content of "data.txt"
5.666
4.3234
7.3434
2.434
1.0
```

Output

The program must terminate with exit code 0 and has to produce an output file with the name "output.txt" in the same directory. That file has to contain two double numbers: first the integer-precision average, and then the float-precision average.

```
Example content of "output.txt"
3.800000
4.153360
```

Validation

The solution is considered to be correct if all given MPI ranks are used and if the application produces correct results. We will evaluate your solution with different data amounts and comm sizes.

Task 5.3: Wator with Scala Actors

In this assignment, you should develop an implementation of the Wator[1] simulation game in Scala. 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.

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[2]. 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.

[1] http://de.wikipedia.org/wiki/Wator
[2] www.scala-lang.org/docu/files/ScalaByExample.pdf

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.

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.

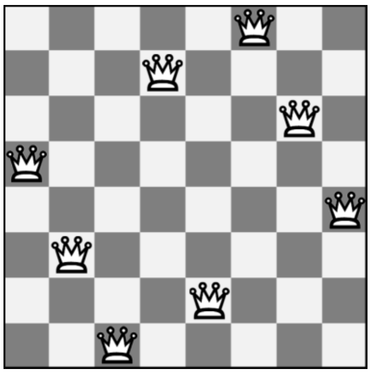
```
Example content of output.txt (actual result depends on chance):
wwww
wwww
wwww
wwww
```

Task 5.4: NQueens with Scala

Implement an NQueens[3] solver using Scala. You can use every parallelization strategy and every parallel concept supported by Scala you like.

Your program has to calculate the number of ways to arrange N non-attacking queens on an N x N board. In a valid solution, no two queens can occupy the same column, row or diagonal.

For a regular chessboard there are 92 distinct solutions. This is one of them:



Input

Your program has to be named "nqueens" and has to take one argument: the *number of queens*. This number is equal to the number of rows and columns of the board the queens are to be placed on.

```
Example: ./nqueens 8
```

Output

Your program must terminate with exit code 0 and produce an output file with the name "output.txt" in the same directory containing the number of valid distinct solutions to place the queens on an according board.

```
Example content of "output.txt":

92
```

[3] http://jsomers.com/nqueen_demo/nqueens.html