Parallel Programming & Heterogeneous Computing

Summer Term 2019

Assignment 5 (Submission deadline: 2019-07-23, 23:59 CEST)

Discussion and Feedback: 2019-07-25, 13:30 CEST, Comm-Zone C-1

The fifth assignment covers programming for shared-nothing systems with using the Message Passing Interface (MPI) and Actors (either Erlang or Scala).

For the MPI tasks, your Makefile has to compile your sources with `mpicc`/`mpic++`. We recommend the OpenMPI or MPICH implementations, which are available on most Unix distributions. The validation system uses OpenMPI.

For the Actor tasks, your Makefile has to compile your sources using `erlc` or `scalac`. Provide a wrapper script to abstract the call.

Two out of the four tasks have to be solved to pass this assignment.

General Rules

The assignment solutions have to be submitted to:

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. Please leave out any Git clones, backup files or compiled executables.
- 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 each assignment. Documentation should be done inside the source code.

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

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.

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. When all block values are computed in a round, 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.

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 executable 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.
- *Optional parameter*: The name of a file (in the same directory) containing coordinates. 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 -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 hotspots.csv:

```
x,y,startround,endround
5,2,0,20
15,5,5,15
```

Example coords.csv:

```
x,y
5,2
10,5
```

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 the simulation terminated. Each value in the field is encoded the following way:

- A block with a value larger than 0.9 has to be represented as "*X*".
- All other values are incremented 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
11124442111111222111
11122222111112222211
11111211111112232211
01111111111111222111
```

If your program was called with a coordinate file, then this file simply contains a list of exact values requested through the coordinate file, rounded to four decimal points.

```
Example content of output.txt with coordinate file

1.0000
0.0306
```

Task 5.2: MPI Collective

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

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

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

Input

Your executable has to be named `mpiavg` and has 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.

```
Example: mpirun -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 print two double numbers: first the integer-precision average, and then the double-precision average with a 6 digit precision.

```
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 input lengths and comm sizes.

Task 5.3: Wa-Tor with Actors

Develop an implementation of the Wa-Tor[1] simulation game using Actors in Erlang or Scala. The game field is a `n x n` grid of cells, modeling an ocean. Each cell contains either 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, 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 *Game Rules*) and a random initial distribution.

Modify your code so that the simulation parallelizes with Scala. 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 your thoughts shortly in a "`README.txt`".

[1] https://en.wikipedia.org/wiki/Wa-Tor

Game Rules

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 thus not possible to leave the ocean alive.

A fish first checks the surrounding cells in **random** order and moves to the first identified free neighboring 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 neighboring 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 neighboring cell, the shark moves to this cell and eats the fish. If no food is available, the shark just moves to a free neighboring 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.

Input

Provide a wrapper script named `wator`, that runs either the Erlang beam-file or Scala class-file. Your program has to 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 a grid of the three letters: `w` (water), `f` (fish), or `s` (shark). 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: ./wator world.txt 99 10 20 5

scala -classpath . Wator world.txt 99 10 20 5
erl -noshell -s init stop -run wator world.txt 99 10 20 5


Example content of world.txt:
wfww
wwsw
wffw
wfwf
```

Output

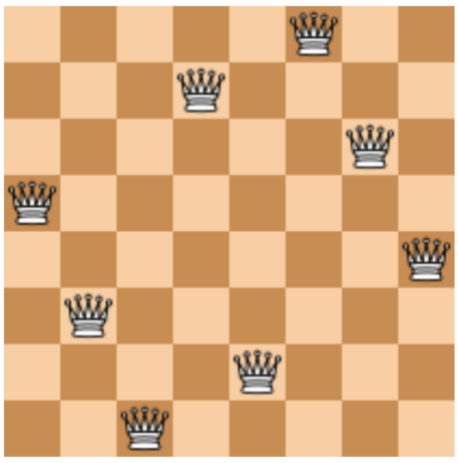
The program must terminate with exit code 0 and produce an output file named `output.txt` in the same directory. This file has to have the same format as the input file.

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

Task 5.4: NQueens with Actors

Implement an NQueens[2] solver using Actors in Erlang or Scala. 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

Provide a wrapper script named `nqueens`, that runs either the Erlang beam-file or Scala class-file. Your script takes the number of queens as single argument. This number is equal to the number of rows and columns of the board the queens are to be placed on.

```
Example: ./nqueens 8

scala -classpath . NQueens 8
erl -noshell -s init stop -run nqueens main 8
```

Output

The program must terminate with exit code 0 and print the number of valid distinct solutions to place the queens on an according board.

```
92
```

[2] https://en.wikipedia.org/wiki/Eight_queens_puzzle