Umeå University
Department of Computing Science

Parallel Programming 7.5 p
5DV152

Final Programming Assignment
MPI Game of Life

Submitted 2017-03-26
Author: Lorenz Gerber (dv15lgr@cs.umu.se lozger03@student.umu.se)
Instructor: Lars Karlsson / Mikael Ränner

Contents

1 Introduction 1
2 Conway's Game of Life 1
3 Design and Implementation 1
4 Usage Instructions 2
4.1 Building 2
4.2 Usage 2
5 Validation 3
6 Benchmarking 3
References 5

1 Introduction

Here it was the aim to design and implement conway's game of life that can be run on a High Performance Computing system using the MPI library.

2 Conway's Game of Life

Conway simplified mathematical models from von Neuman that were about structures which had the ability to copy themselves. Published 1970, the surprising property of the program was that it represented a Turing complete system despite using only very few simple rules. It was the start of a new research field about 'cellular automata'.

Conway's game of life is represented by a regular grid of limited size that consists of individual cells which can either be death or alive. The game of life proceeds in rounds: In each round the state of each cell is evaluated individually and then updated to an eventual new state. The most important feature of this system are the rules applied for assessing and updating the state of each individual cell: Based on the eight neighboring cells, it is determined if a cells environment is life promoting or overcrowded. Hence, the variable to determine and evaluate is the number of neighboring cells alive.

3 Design and Implementation

From the description of the game itself, some preliminary design drafts were sketched. Then a short literature study about available implementations was conducted. Several programs of interest were found and inspected for inspiration [1] [2] [3]. To be flexible, it was decided to implement both randomized game start state and loading of a saved state from file. Because graphics programming was not the focus of this course a simple script in R [5] was written for visual inspection of the game state after a number of generations. 'Foster's Methodology' as reproduced in the course literature [4, p. 66] was applied for designing the parallel program:

1. Partitioning
   Two task were identified: Detecting the number of neighbors alive and updating the state for the next round. However, as it will be seen later, in the current case, it could make more sense to see the operations of each single cell as an individual task.

2. Communication
   The two first mentioned tasks above need to run in sequence and they also need to be concerted among the whole game. If a single cell is seen as an individual task, then the communication would be the step where the state of the neighboring cells is determined. The cell is locally dependent to access the eight surrounding cells.

3. Agglomeration / Aggregation
   Agglomerating the individual tasks per cell seems to make sense as they adhere to a strict temporal run sequence. However more interesting is to consider agglomerating larger numbers of individual cells. As most of the cells would have eight neighbors within the group, they could be considered to be independent with a few exceptions for the cells on the interface/border.

4. Mapping
   As the identified tasks are tightly coupled, it seems to make most sense to map larger agglomerates of cells to individual MPI nodes. Hence, communication during the game will be limited to handling the mentioned special cases of cells that are at the border of a local cell group.

For the implementation it was decided to divide the global square field along vertical lines into equal sized local rectangular fields. As such, communication between nodes included only the cells that were on the vertical borders as the horizontal borders (top and bottom) could be accessed locally. To facilitate communication, the local rectangular cell groups were padded with one layer of cells where the cell state from the next-by local border could be imported to.

This strategy resulted in a total of three tasks per game cycle:

1. update interface region
2. determine number of neighbors alive
3. update state

4 Usage Instructions

4.1 Building

The makefile that builds both the serial (`serial_life`) and MPI parallel version (`mpi_life`) is provided. In the HPC2N environment, the user only needs to load the appropriate compiler module (eg `module load openmpi/gcc/1.8.8`), go to the directory with the source code and run `make`. `make clean` was configured to clean up the directory from `.o`, `.txt` and `.pdf` files.

4.2 Usage

Running the Program

Both the serial and the parallel version use the same simple command argument parser that accepts either three or four command line arguments. These are in sequence: `n_size generations outfile [infile]`. If no infile is indicated, a random start state for the simulation is created. The only output to stdout obtained from the programs is the runtime. However, a file with the game field as a text file is written to the working directory using the filename indicated as command line argument.

Graph Output using R Script

To obtain graphical output a short R script was written. Hence to obtain a graphical result it is needed to have the R language interpreter installed on the system [5]. The script shown below can be run from the command line with the following command:
`Rscript --vanilla plot_Script.R data_file.txt graphical_output.pdf`
This command will use the output from either `serial_life` or `mpi_life` (`data_file.txt`) and render the pdf file `graphical_output.pdf`.

```
#!/usr/bin/env Rscript
args = commandArgs(trailingOnly=TRUE)
```

```
# test if there is at least one argument: if not, return an error
if (length(args)==0) {
  stop("At least one argument must be supplied (input file).n", call.=FALSE)
} else if (length(args)==1) {
  # default output file
  args[2] = "life_image.pdf"
}

## preparing data
data<-read.csv(file=args[1], sep=" ", header=FALSE)
image_data<-matrix(data[1,1]*data[1,2], data[1,1], data[1,2])

for(i in 2:dim(data)[1]){
  image_data[data[i,1], data[i,2]] <- 1
}

# plotting to file
pdf(file = args[2])
image(image_data, main="Conway's Game of Life",
        xaxt='n', yaxt='n', ylab=paste("y size: ",
        data[1,2]), xlab=paste("x size: ", data[1,1]))
dev.off()
```

5 Validation

Here the qualitative result of the serial and parallel version were evaluated to yield the same result. This was done using a bash script that runs both programs and then compares the output files using the system tool diff. Two versions of the bash script are contained in the source folder: `validation.sh` can be used on a shared-memory desktop system without batch submission. `validation_hpc2n.sh` can be submitted on the HPC2N 'Abisko' cluster system using `sbatch ./validation_hpc2n.sh`. This script includes currently the student account that was assigned to the user 'stud02'.

The typical output of a validation session is shown below:

```
Generating random 1024 x 1024 game field
running serial implementation on random data (100 generations)...
running mpi implementation using 4 processes on random data (100 generations)...
comparing the resulting datasets...
No output from 'diff', validation succeeded
```

6 Benchmarking

Initially, a run to compare the serial with the the MPI implementation running on a single process was performed. A square game of size $n = 1000$ with the same start state was run in 1000 generations with each 3 replicates. Both implementations resulted in low variation within and minimal difference between the two implementations (data not shown). Hence, for benchmarking it was decided to only use the MPI version.

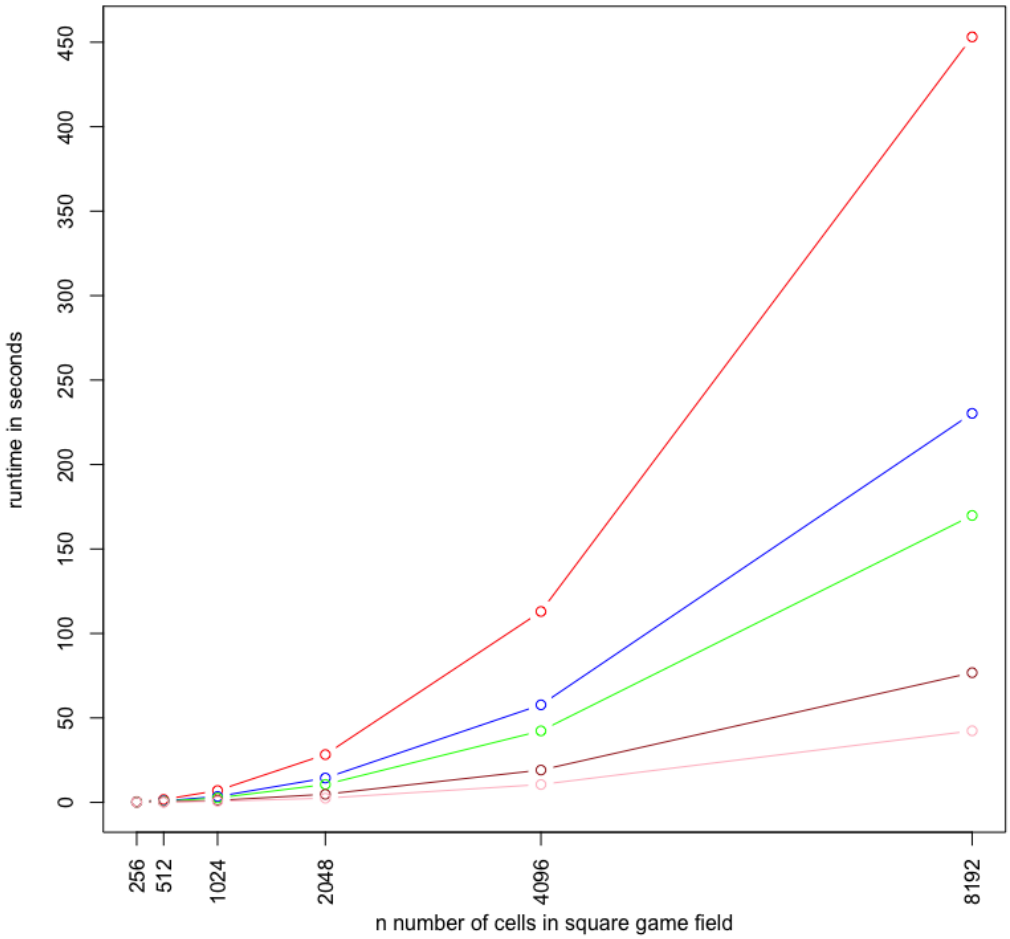

Figure 1: *This figure shows the whole benchmarking data aggregated per core and field size using the median*

The mpi implementation was run on the HPC2N 'Abisko' cluster. All timing runs included 100 game generations. MPI runs where conducted with $k = 1, 2, 4, 8, 16$ processes. From preliminary evaluation of the data using boxplots in R [5] (plots not shown) it was obvious that the variance per replication (3 technical replicates) was very low compared to the total run time. Hence it was decided to calculate median values without presenting a measure of variation for each configuration. A graphical summary of the benchmarking is shown in figure 1.

Table 1 shows a detailed numerical evaluation with obtained speedups (S), efficiencies (E) and absolute parallel overheads (O) for all configurations. Looking at the data, it sticks directly out that running two MPI processes obtained ideal speedups and efficiencies around 1 while for four MPI processes speedups and efficiencies dropped significantly. For higher even higher number of MPI processes two effects can be observed: At smaller game sizes, speedups are modest and efficiencies even drop. At larger game sizes, speedups get somewhat better again and efficiencies sometimes slightly increase or at least stay at the same level. The sharp break between two and four processes could partly be due to the intrinsic

Table 1 *The table below shows a detailed numerical evaluation of the benchmarking with speedups (S), efficiencies (E) and absolute parallel overhead times (O) for all configurations. Each configuration was measured in triplicates.*

		number of MPI processes			
n size		2	4	8	16
256	S	1.98	2.57	4.56	6.62
	E	0.99	0.64	0.57	0.41
	O	0.00	0.06	0.04	0.04
512	S	1.95	2.59	5.43	9.48
	E	0.98	0.65	0.68	0.59
	O	0.02	0.23	0.10	0.07
1024	S	2.01	2.67	5.94	11.03
	E	1.00	0.67	0.74	0.69
	O	-0.01	0.87	0.30	0.20
2048	S	1.96	2.67	5.91	11.33
	E	0.98	0.67	0.74	0.71
	O	0.27	3.53	1.25	0.73
4096	S	1.96	2.67	5.90	10.68
	E	0.98	0.67	0.74	0.67
	O	1.24	14.08	5.02	3.52
8192	S	1.97	2.67	5.90	10.69
	E	0.98	0.67	0.74	0.67
	O	3.72	56.57	20.15	14.06

fact that the number of neighbors each process corresponds with increases here: Two processes always communicate only with 1 additional neighbor as the game field folds over in both directions. For three and more processes, each process communicates always with two other processes.

Probably the degradation of speedups and efficiencies for larger number of processes is due to the very tight temporal synchronization of the game: It probably happens regularly that processes have to wait for each other due to the send/receives which are currently coded as individual events. Probably, some performance increase would be possible if one would change this communication to a collective communication strategy.

However, when looking at the absolute parallel overhead times it can also be seen that they decrease with increasing size of the game field. This is a good sign and suggests that the application is at least weakly scalable in regard of the game field size. This would however not hold true for increasing the problem size by the number of generations to be calculated.

References

[1] Game of life c serial. `https://rosettacode.org/wiki/Conway%27a_Game_of_Life`, 2017. accessed: 2017-03-23.

[2] Game of life mpi, kth summer school. `http://www.pdc.kth.se/education/summer-school/mpi-exercises/mpi-lab-codes`, 2016. accessed: 2017-03-23.

[3] Game of life with mpi. http://www.shodor.org/petascale/materials/distributedMemory/code/Life_mpi/, 2002. accessed: 2017-03-23.

[4] P.S. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufman, 2011.

[5] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.