TU WIEN Informatics Parallel Computing

Prof. Dr. Jesper Larsson Träff
Assoc.Prof. Dr. Sascha Hunold
TU Wien, Faculty of Informatics
Research Group for Parallel Computing

# High Performance Computing

2023W

Assignment 1: Game of Life Stencil

Issue date: 2023-11-13
Due date: 2024-01-16 (23:59)

Hand-in via TUWEL. No deadline extensions will be granted. Project can be done in groups of at most three (3) participants. One hand-in per group suffices. Group registration in TUWEL required. Mark clearly if, where, and how tools like ChatGPT were used.

This project deals with efficient ways of implementing stencil computations on large matrices that are distributed across the processors in a large HPC system. The task is to explore and compare different ways of implementing a well-known 2-dimensional stencil computation on small integers using different features and communication models of MPI. In particular, the task is to provide solid evidence for comparing the different implementation approaches and assessing that certain speed-up over a sequential implementation can be achieved.

As an example, we use the *Game of Life* stencil (see for instance [1, Chapter 25] for a classical exposition to the Game of Life); but instead of living and dying in a potentially infinite 2-dimensional grid, we fold the grid around into a finite world. More concretely, we are given an $n_r \times n_c$ matrix (grid) of *cells* $C[i][j]$ that can either be *alive* (true) or *dead* (false). The next generation of alive and dead cells is computed by applying the following rule on each cell:

> A cell $C[i][j], 0 \leq i < n_r, 0 \leq j < n_c$ has 8 neighbors $C[i][j \ominus 1], C[i \oplus 1][j \ominus 1], C[i \oplus 1][j], C[i \oplus 1][j \oplus 1], C[i][j \oplus 1], C[i \ominus 1][j \oplus 1], C[i \ominus 1][j], C[i \ominus 1][j \ominus 1]$ where $x \oplus 1 = (x + 1) \bmod m$ and $x \ominus 1 = (x - 1 + m) \bmod m$ (where $m = n_r$ for rows and $m = n_c$ for columns). The cell will *survive* for the next generation, if it is alive and has a two or three (2 or 3) alive neighbors. A cell that is dead will be *reborn* for the next generation, if it has exactly three (3) alive neighbors. Cells that are not surviving or reborn, will be dead for the next generation.

For a given input configuration of live and dead cells on the $n_r \times n_c$ grid, you should be able to iterate the Game of Life for at least 1000 generations, and report the final state of the world, at least in summary by computing the total number of live and dead cells (which should add up to $n_r n_c$). You should develop first a sequential version of this Game of Life, and then estimate the speed-up of the parallel versions relative to this sequential baseline, both as *strong scaling* experiments, where the total grid size is kept constant independently of the number of MPI processes, and as *weak scaling* experiments, where instead the size of the grid per MPI process is kept constant (that is, the total grid size will increase linearly with $p$).

For stencil computations with MPI, it might be helpful to consult the script "Lecture Notes on Parallel Computing" (as always, feedback on this will be much appreciated).

## Exercise 1 (10 points)

Develop a sequential implementation as an MPI program that can run on one process of the Game of Life for a given number of generations (input parameter to your program). Develop an input generator for at least one type of initial configurations, for instance randomly allocated dead and alive cells with a given probability for being alive (input is the size of the grid $n = n_r \times n_c$ and the type of initial configuration; see sample code in TUWEL). The cell contents should be represented by a short C data type, either `char` or a bit in a word. Your implementation must be space efficient in the sense of maintaining at most two generations, the current and the next, that is at most two $n_r \times n_c$ matrices. You may consider how to do better (one matrix only with some additional rows and columns). The output should be a summary of the number of dead and alive cells at the last generation. As debug help, your program might be able to output the summary for each new generation as well as to print (parts of) the grid of alive and dead cells. You must also be able to *time the computation*: Use `MPI_Wtime` to measure the time (in micro seconds) between the start of the stencil iteration until the end of the last generation. Make it possible to repeat the experiment by giving a number of repetitions, and report the time per repetition to be able to later compute the average or mean time and perform statistical analysis. Do *not* include the time for generating the input configuration and reporting the results. Use these times as the basis for the speed-up calculations for the parallel versions.

You should explain the choices (data type) and optimizations made for your implementation. What is the asymptotic complexity per generation in terms of the matrix size $n$? Empirically, does the time depend on the configuration of the input matrix?

## Exercise 2 (10 points)

For all parallel versions, the MPI processes should be placed in a 2-dimensional periodic grid of $r \times c = p$ rows $r$ and columns $c$ processes, such that the last process in each dimension is a neighbor of the first process in that dimension. Use an MPI Cartesian communicator for this, and use `MPI_Cart_create` (and possibly `MPI_Dims_create`) to set up the process grid (there is sample code in TUWEL). Your programs must in principle work for any number of MPI processes from one to the maximum number of processes on our Hydra system; but for convenience you may assume that the size $n$ is divisible by $p$ (this means that your program, when $n$ is given as input, may not run with all $p$). At least for some $p$, you must use $r > 1$ and $c > 1$ which means that a purely one-dimensional implementation will not be acceptable.

Try creating the Cartesian communicator with and without reordering (`reorder` flag true and false) and verify, for instance with `MPI_Comm_compare`, whether there is any actual reordering of the MPI processes being done by the MPI library.

For all parallel versions, the $n = n_r \times n_c$ input grid is to be distributed evenly across the MPI processes, such that each process has a subgrid of $n_r/r \times n_c/c = n^2/p$ cells. Explain the way you choose to distribute the input (dense, consecutive submatrices or other convenient distribution).

It should be possible to generate the same input as for the sequential version. Input generation may be done either sequentially followed by a distribution of the input over the MPI processes, or in a distributed way where each process generates only the part of the input that it needs. It should be possible to experiment with both strong scaling where the total input size is independent of the number of processes, and weak scaling, where the input size per process is given and kept constant. It should also be possible to verify the output of the parallel program against se-

quentially generated output. Use this feature throughout your development to ensure as far as possible that your parallel programs are always correct (relative to the sequential version).

## Exercise 3 (15 points)

You can select either of the following three options, but one of them has to be implemented.

### Option 1

Implement a parallel version of the Game of Life using non-blocking point-to-point communication, `MPI_Isend` and `MPI_Irecv` with `MPI_Wait` or `MPI_Waitall` completion. You first have to think about which cells (rows and columns) have to be exchanged between neighboring processes, which means thinking first about how to distribute the input and which processes will be neighbors of each other.

For some communication between some processes, the data to be exchanged, for instance matrix left or right columns, will not be consecutive in memory, so simple `MPI_Isend` and `MPI_Irecv` alone will not be sufficient. You either have to pack/unpack the data into and out of intermediate, consecutive buffers, or use MPI derived datatypes (`MPI_Type_vector` will do) for specifying where the data are. Both solutions are acceptable, but you should explain your choices.

Each process should time its execution with `MPI_Wtime` as for the sequential version, and as for the sequential version it should be possible to repeat the measurements for a specified number of repetitions. The parallel time is the time for the last process to finish, assuming that all processes started at the same time. Use `MPI_Barrier` before starting the timing for each repetition, and use `MPI_Reduce` with `MPI_MAX` to find the time spent by the slowest process and report this. Report the average time over a number ($> 10$) repetitions of the experiment. How much do run times wary from experiment to experiment?

### Option 2

Instead of non-blocking point-to-point communication, implement the Game of Life using only blocking `MPI_Sendrecv` communication. Explain the necessary modifications, explain your choices, and repeat the experiments as for the previous exercise.

### Option 3

Instead of point-to-point communication, use one-sided communication instead and either `MPI_Put` or `MPI_Get` for the data transfer, and either `MPI_Win_fence` or `MPI_Win_start-MPI_Win_complete-MPI_Win_post-MPI_Win_wait` to complete the exchange. Explain your choices, and repeat the experiments as for the previous exercise.

## Exercise 4 (20 points)

An altogether different way of implementing the exchange is to use collective neighborhood communication. The exchange of neighboring cells can be done either with `MPI_Neighbor_alltoallv` (where you will most likely have to pack/unpack explicitly) or with `MPI_Neighbor_alltoallw`

(where you can reuse the `MPI_Type_vector` datatype from the other exercises). Explain your choices and any difficulties in the implementation.

To define the proper neighborhood, you have to use `MPI_Dist_graph_create_adjacent` to create a communicator where the neighboring processes for each process will be defined. The Cartesian coordinates from the first exercise and the translation function between coordinate and rank representation will be most helpful (indispensable) for this (for some additional discussion of these functionalities, see [2]).

Repeat the experiments from the previous exercises.

## What to hand in?

Your hand-in will consist of three parts.

- A short report specifying your implementations (what they can do and what they cannot do, what the input parameters are), how to run your codes, and stating what you think works (and what not). This part can be short, 1-3 pages.

  The report should describe in overview your implementations, the choices you made, why you made them (saving space, saving time, convenience, better code, ...). This part should also not be overly long, 4-7 pages.

- Your programs in C or C++. You must state how to compile so that your results can be reproduced (checked by us). The code should be readable with comments (but not over commented).

- Experimental results as part of your report with the process configurations and input sizes stated below. Compare and discuss differences between your two parallel implementations (with neighborhood collectives, and one of the other options). Use the OpenMPI library as explained in the appendix. Perform at least 10 repeated measurements and report the average running time; further analysis is optional.

Mark clearly if, where, and how tools like ChatGPT were used.

### Experiments to be conducted and reported

Document your results with the following MPI process configurations and input sizes. The runtimes should be averages of at least 10 repetitions. Further statistical analysis is optional.

1. Sequential run, $p = 1 \times 1$, $n \in \{1024^2, 10240^2\}$.
2. Strong scaling experiments:
   - $p = 1 \times 32$, $p = 8 \times 32$, $p = 16 \times 32$, and $p = 32 \times 32$, $n \in \{1024^2, 10240^2\}$
   - provide three plots (runtime, speedup compared to sequential run, parallel efficiency compared to sequential run) as functions of the number of MPI processes $p$.
3. Weak scaling experiments :
   - $p = 1 \times 32$, $p = 8 \times 32$, $p = 16 \times 32$, and $p = 32 \times 32$,
   - $n' = 1024^2$, which is the size for $p = 1 \times 1$ that needs to be increased with the number of processes
   - provide one plot (runtime) as function of the number of MPI processes $p$.

## A. How to Run on hydra

1. First, you will have to log into hydra.

```
ssh hydra
```

2. Once you are on hydra, you will have to load the MPI library

```
spack load openmpi@4.1.5
```

3. In order to run your application, you have two options: 1) use `sbatch` to submit a job to the queue or 2) use `srun` to run the application directly if enough compute nodes are free.

## B. A Step-by-step Example for Running an MPI program on hydra

You may have an MPI program like this on you local machine in a file called `mpi1.c`:

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[]){
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("Hello␣from␣process␣%d␣of␣%d\n", rank, size);

    MPI_Finalize();

    return 0;
}
```

We now copy the file to hydra using `scp` or `sftp` (`hydra` is the name of the machine used in your `.ssh/config` file):

```
scp mpi1.c hydra:~/
```

Then, we log on to hydra:

```
ssh hydra
```

Load the MPI library and compile program:

```
spack load openmpi@4.1.5
mpicc -o mpi1 -O3 mpi1.c
```

Now, run using srun:

```
stester@hydra-head:~$ srun -t 5 -p q_student -N 4 --ntasks-per-node=3 ./mpi1
Hello from process 0 of 12
Hello from process 2 of 12
Hello from process 6 of 12
Hello from process 7 of 12
Hello from process 10 of 12
```

```
Hello from process 11 of 12
Hello from process 3 of 12
Hello from process 1 of 12
Hello from process 4 of 12
Hello from process 5 of 12
Hello from process 8 of 12
Hello from process 9 of 12
```

The parameter -N denote the number of compute nodes requested, where on each compute node –ntasks-per-node processes will be created. Thus, in total, we create 12 processes.

Often, it is better to submit a job to the queuing system and wait for the result. Once the jobs has been submitted, you can log out and come back at a later point in time to collect the results. To do so, you will have to write a batch script (I give the file the name `job1.sh`):

```
#! /bin/bash

#SBATCH -p q_student
#SBATCH -N 2   # how many compute node
#SBATCH --ntasks-per-node=4 # create 4 processes per compute node
#SBATCH --cpu-freq=High
#SBATCH --time=5:00  # max time a job can run before it is killed

srun ./mpi1
```

You can now submit this job and check whether it is running:

```
stester@hydra-head:~$ sbatch job1.sh
Submitted batch job 4330052
stester@hydra-head:~$ squeue
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
4330052 q_student  job1.sh  stester  R       0:01      2 hydra[01-02]
```

With squeue, you will see the status of your job(s). In this case, the job is already running and has gotten ID 4330052. After the job has completed, there will be an output file with the result.

```
stester@hydra-head:~$ cat slurm-4330052.out
Hello from process 5 of 8
Hello from process 1 of 8
Hello from process 2 of 8
Hello from process 3 of 8
Hello from process 0 of 8
Hello from process 4 of 8
Hello from process 6 of 8
Hello from process 7 of 8
```

## References

[1] Elwyn R. Berlekamp, John H. Conway, and Richaard K. Guy. *Winning Ways for your Mathematical Plays*, volume 4. A. K. Peters, second edition, 2004.

[2] Jesper Larsson Träff, Sascha Hunold, Guillaume Mercier, and Daniel J. Holmes. MPI collective communication through a single set of interfaces: A case for orthogonality. *Parallel Computing*, 107, 2021.