Conway's Game of Life in MPI

Introduction

The Game of Life is a board game. The board consist of N x M cells (N rows, M columns), each having value 1 or 0, depending on whether or not it contains an "organism". Every cell on the board has eight neighbors. In our version of the game, the world has wrap-around both horizontally and vertically, so when moving off the left of the board, you end up at the right; similarly top and bottom are connected.

Initially, some of the cells hold organisms. The cell values then change in synchronous global steps according to the following simple rules:

- Every organism with two or three neighboring organisms survives for the next generation.
- Every organism with four or more neighbors dies from overpopulation.
- Every organism with one or no neighbor dies from isolation.
- Every empty cell adjacent to exactly three occupied neighbor cells will give birth to a new organism.

For more information, check out the "Game of Life" Wikipedia page and online applet:

- http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
- http://www.bitstorm.org/gameoflife/

The parallel Game of Life is an example of what is called a parallel stencil operation and is very similar to a large number of simulation type parallel processing applications like simulating the movement of an oil spill in an ocean or simulating the spread of a forest fire, or temperature distribution in conducting material.

The goal of the assignment is to get familiarity of MPI by writing a parallel MPI program for the Game of Life, based on the sequential version. See file gol/gol-seq.c Your job is to write an MPI program that simulates the Game of Life on a varying number of compute nodes and CPU cores of DAS-5, that should be able to run significantly faster than the sequential version.

In the MPI version, every process is responsible for a portion of the game board of size (N x M)/p. The total number of processes p is determined when submitting your job; it is the number of compute nodes times the number of CPU cores per node.

The following (positional) arguments are provided to the application, e.g., "gol-par <N> <M> <T> <W> <C>":

arg	meaning
N	number of rows of the world
M	number of columns of the world
T	number of timesteps to run the simulation
W	every W timesteps, print global world state
C	every C timesteps, print number of live cells

In the parallel implementation, at each W-th step, the process with MPI rank 0 should collect the world state from all processes and print the current configuration of the entire game board on standard output. If W is set to 0, no board should be printed, so this can be used for performance experiments.

In the parallel implementation, at each C-th step, the process with MPI rank 0 should write the sum of the number of live cells at all processes on standard output. If C is set to 0, your program should only print the number of cells alive at the end, as the sequential version does.

Your program should also calculate and print the runtime (max. wall clock time over all processes) on standard error.

MPI

This assignment for the parallel programming practical requires you to implement a parallel GOL (Game Of Life) algorithm, using C and the MPI (Message Passing Interface) communication library. The application has to be run and benchmarked on the DAS-5 cluster at the VU.

The Message-Passing Interface or MPI is widely used. It is not a new programming language; rather it is a library of subprograms that can be called from C and Fortran programs. It was developed by an open, international forum consisting of representatives from industry, academia, and government laboratories. It has rapidly received widespread acceptance because it has been carefully designed to permit maximum performance on a wide variety of systems, and it is based on message passing, one of the most powerful and widely used paradigms for programming parallel systems.

The introduction of MPI also made it possible for developers of parallel software to write libraries for parallel programs that are both portable and efficient. Use of these libraries will hide many of the details of parallel programming, and, as a consequence, make parallel computing much more accessible to students and professionals in all branches of science and engineering.

Parallel GOL

In the parallel implementation of GOL, the "world" of cells should be partitioned over the processes. Every process is responsible of computing the next timesteps of its part of the world. The values of the cells at the boarder of the domains should be exchanged with the appropriate processes before each new timestep, so that your parallel implementation gives the same results as the sequential one. As mentioned, also note that the world has "wrap-around", i.e., it could be seen as the surface of a torus-like shape, represented by a 2D array.

Requirements

Implement a parallel Game Of Life (GOL) algorithm using MPI, starting from the provided sequential implementation, which is available on Canvas. The reference sequential program contains the 'stencil' operation that implements the GOL liveness rule, that should repeatedly be applied to each grid point. The algorithm uses a predetermined number of timesteps, which is given as parameter.

Data distribution at the start, counting the global sum of live cells, and gathering the data for printing the global world state must also be implemented in your program.

The base parallel version of GOL must partition the world grid such that each process gets approximately N/P consecutive rows. The application should in principle run on any number of machines, and accept any given problem size (assuming sufficient memory is available).

Limit your code changes to the essentials. Much of the existing sequential code can and should be reused in your parallel implementation. This mimics a real life situation where a large existing sequential application has to be parallelized. You do not want to spend much time to completely redesign the entire application, but you typically try to get reasonable speedups mostly reusing the existing code base. The provided sequential code for GOL is suitable for this.

Make sure you use the right MPI operations for the best performance. E.g., if you need to communicate a sequence of grid points, do this by means of a single transfer, instead of a sequence of tiny transfers. When it is efficient to use collective operations, do so. You can use the MPI_Wtime() function provided by MPI to measure the performance of computation/communication sections of your code; this may be helpful to analyze the communication overhead, which can impact parallel performance.

When doing performance measurements you should not take the time required for data distribution, data gathering and printing into account (i.e., specify 0 for the "V" argument while doing these experiments).

Measure the speedups of the GOL version on DAS-5, but using only up to 8 nodes and up to 16 cores per node. NOTE: DAS-5/VU itself has over 64 nodes, but this system is primarily used for research projects, so we want to leave enough resources to remain available for that purpose, hence the limitation to 8 compute nodes.

To limit the number of experiments and hence load of DAS-5, follow these guidelines:

- select two representative world sizes: a small sized one (but large enough that every process has some data) and a much bigger one, adapting the number of timesteps so that the sequential version completes within a few minutes;
- check performance only on a subset of all possible node/core combinations, e.g., on 1, 2, 4, 8 nodes with 1, 4, 16 cores, so in total scaling from 1x1 up to 8x16=128 cores. Spawn multiple processes per DAS-5 node using the available prun option. E.g., "prun -np 2 -8 gol-par <N> <M> <T> <W> <C>". allocates 2 nodes and spawns 8 processes per node, so in this case 16 CPU cores in total are used.
- Draw separate speedup lines per core setting, and discuss performance patterns. Does speedup change with increasing problem size, and why?

Part of the message transfers via MPI happen inside a node (using shared memory), but between compute nodes a fast interconnect network (InfiniBand) is used. This is all hidden by the MPI API and taken care of by the MPI implementation, but depending on the application, you may see performance differences depending on where communication takes place.

Don't wait with your experiments until the last week of the assignment, or you might have to wait long for other users' jobs to complete first.

Write a short report in English describing your design, implementation, and performance results for your parallel GOL solution. Describe (briefly) the way the algorithm works, focusing on the MPI implementation details. Before presenting the performance results, mention the composition of the experimental set-up:

- number of nodes and CPU cores, input parameters, application versions, etc.
- the way you measured the execution time

Include performance graphs of your application. Report speedups in the form of graphs with the total number of processes (the number of nodes times number of cores per node) on the x-axis to provide quick insight in the performance of your implementation. Highlight a number of important performance aspects and key insights in the text.

Compiling and running your applications

Refer to the DAS-5 documentation to execute your program using prun. For MPI, use the OpenMPI implementation on DAS-5, which becomes available in your environment by mean of "module load openmpi/gcc/64". You can start with the simple MPI example provided here:

- https://www.cs.vu.nl/das5/jobs.shtml

Submitting

You have to submit both the code and a report.

Important: Because we partly use automatic test scripts to test and benchmark your submissions, you must strictly follow the instructions below.

- Make sure that your submission has the exact same directory structure as the provided template.
- The parallel program should be compiled in the same directory as the sequential program. Change the Makefile to make sure that your parallel GOL version is compiled with the "make gol-par" command. The standard parallel executable must be called "gol-par".
- If you have additional executables as part of the bonus work, include an additional target in the Makefile for this.
- Make sure that your parallel program gives the exact same output as the sequential program. Because we compare your application's output with the correct output using diff, any difference (except for the run time and other diagnostics you print on standard error) could lead to a rejection of your submission. A sanity check script is available in the archive on Canvas to check this.
- Create the report as a PDF file, and make sure you place the file in the pre-created docs directory.

- Place the code and documentation directories in a directory that contains your VUnet id, full name, and the type of assignment (i.e., mpi) - e.g., jj400_JanJanssen_mpi. Archive the directory as a .tar.gz file (i.e. jj400_JanJanssen_mpi.tar.gz) and submit this archive via Canvas.

Note that passing the sanity check does not guarantee a passing grade, but guarantees the assignment can be considered ready for grading.

Documentation

- C Reference manual and other documentation
- The official MPI documentation http://www.mpi.org

Grading

A correct implementation, compliant with all the requirements above (both for code and documentation) is graded with 8.

Up to 2 bonus points (to grade 10) can be given for extra work on the following aspects:

- implement GOL also using a different data distribution by splitting up the grid column-wise instead of row-wise, and comparing the achieved performance. Discuss for which grid sizes you expect the alternative data distribution may give better performance, and evaluate this in practice. Note that column-wise distribution and communication may be a bit more challenging given the way memory for multidimensional arrays is laid out in C. See
  - https://www.rookiehpc.com/mpi/docs/mpi_type_create_resized.php
- make a more detailed evaluation of communication overheads using MPI_Wtime() around MPI primitives, comparing non-blocking and blocking MPI primitives. Do you expect much impact in this case? Discuss results, taking Amdahls and Gustavson's laws into account:
  - https://en.wikipedia.org/wiki/Amdahl%27s_law
  - https://en.wikipedia.org/wiki/Gustafson%27s_law
- If you want to work on some other extensions for the bonus points, clearly tell why you thought it was interesting, and what insights it provides compared to the basic parallel version.

Important:

You only get bonus points for extra work if is done clearly in addition of the base version, so put code for the bonus version in a separate implementation or use #ifdef BONUS_XYZ to add functionality to the base source code. Put a discussion related to the bonus work in a separate section of your report.

The points are distributed as follows:

points	assignment task
3	correct base parallel implementation, compliant with all the requirements
1	coding style (use clear code with concise comments)
4	report with discussion of your implementation and evaluation
2	bonus for extra work

TODO list

We recommend following these steps:

- Look at the sequential version of the GOL application.
- Recheck the MPI introduction presented in the earlier course in class, and/or look through the extra MPI introductory material on Canvas.

- Create a parallel application using MPI and C.
- Make sure your application runs correctly for various input parameters, also ones where the world row/column size does not exactly divide the number of processes used.
- Test the functionality and correctness of your application on DAS-5. With the sample fixed "glider" world provided, you can visually check whether the output looks OK, but it should of course work on arbitrary worlds, and for that random patterns are more suitable. Test that your parallel implementation gives exactly the same output as the sequential version. Either do this with the provided sanity-check script (which runs a few standard random worlds and compares them against pre-recorded sequential outputs) or make some tests of your own.
- Note that the random world generated depends on a particular random number implementation, which may be different depending on the platform used. If you test this on a Windows/MacOS laptop, or use a different version of Linux, it may generate a different world and therefore give different results. So test this on DAS-5, which you should use anyway for your evaluation.
- Benchmark the sequential and parallel application for selected input parameters, using both a small and a large world size.
- Write a report about your MPI assignment. Present the experimental results and discuss the achieved performance of your solution, referring to speedups graphs included. Mention the difficulties you encountered (if any) and how you solved them.
- (Optional) To get up to 2 bonus points for this assignment, check the Grading section above and implement one or two of the extensions described there.
- Check that the code and report are placed in the correct directories, make sure the Makefile is also correct, archive the entire directory, and submit the archive via Canvas.
- If needed, you can update and resubmit the code and/or report on Canvas. Only the last submission found on Canvas at the submission deadline will be checked and graded.