Parallelising Conway's Game of Life

Rory Johnston

This project aims to implement Conway's Game of Life using a peer-to-peer message passing architecture to achieve distributed parallelism for improve performance. In brief, this is achieved by subdividing the game's (global) domain into a collection of smaller grids which are assigned to individual processors, and allowing these processors to communicate their grid's boundaries to each other to simulate a contiguous environment. Implementing a parallel version results in drastic speed-up, roughly halving (0.5198) computational time on every doubling of number of cores for a domain of $10{,}000\ (rows) \times 10{,}000\ (columns)$.

Library Features:

The library is designed to be deployed in a logical and user-friendly manner (with as many stages automated as possible) and accessible to both Microsoft (MVSC) and iOS/Linux users, where comprehensive help is given throughout the code and README.md documentation. The key features of this library are:

- Pre-processor (Python / Python for Jupyter): This prepares the environment for the next run of Conway's Game of Life by cleaning the directory structure. This program also contains the functionality to run the Game of Life on whatever image the user desires. As explained in the program, the user must simply choose the path to the input file directory (options provided within the script for different on operating system). Should the user input their own $_{\text{image}}$, the user must declare the number of cores that they intend to run the game and provide the path to the image (template given in script).
- Life (C++): Uses the Message Passing Interface (MPI) to perform Conway's Game of Life on multiple cores by dividing the environment into smaller grids and designating these to multiple cores. The cores then communicate their boundaries to each other to simulate contiguity between local domains for a periodic or fixed environment. As explained in the program and the README documentation, the user is required to enter the domain hyper-parameters at the top of the script to alter the starting conditions and the way in which the game is played.
- Post-processor (Python / Python for Jupyter): This script is fully automated, except for one initial change of file path (template given in script), and generates a .mp4 of the entire Game environment for all iterations.

Code Analysis:

This implementation of Conway's Game of Life uses the MPI standard to conduct peer-to-peer communications between cores. The memory is entirely distributed; each core retains the information it requires to perform the game and is naïve to the surrounding cores between communications. Useful information is then written into data and meta-data folders to be collated by the post-processor and subsequently visualized. An egalitarian peer-to-peer structure that distributes memory across all cores was chosen as these arrangements are best suited to coupled systems where only a relatively small amount of information needs to be communicated across boundaries between core grids. This architecture was also in part chosen for its versatility, particularly for scenarios where each node may contain information that's similar in nature but different in volume (especially in scenarios where load balancing means that each core is designated a slightly differing volume size).

This architecture was also chosen for its scalability to larger problems; unlike the Master-Slave architecture, we remove dependence on a single core and allow each one to contain its own portion of the data. This not only removes the cost of redundant storage, it also mitigates the risks associated with storing the entire grid on a single core when the domain size becomes non-trivial.

Performance Analysis:

By virtue of the system's peer-to-peer nature, there are no serialized segments in a spatial sense (the game itself is time-dependent, so we cannot decompose the temporal domain that this game occupies). By partitioning the spatial domain amongst cores, we achieve speed-up proportional to the number of cores being used, such that doubling the number of cores roughly halves the computational time.

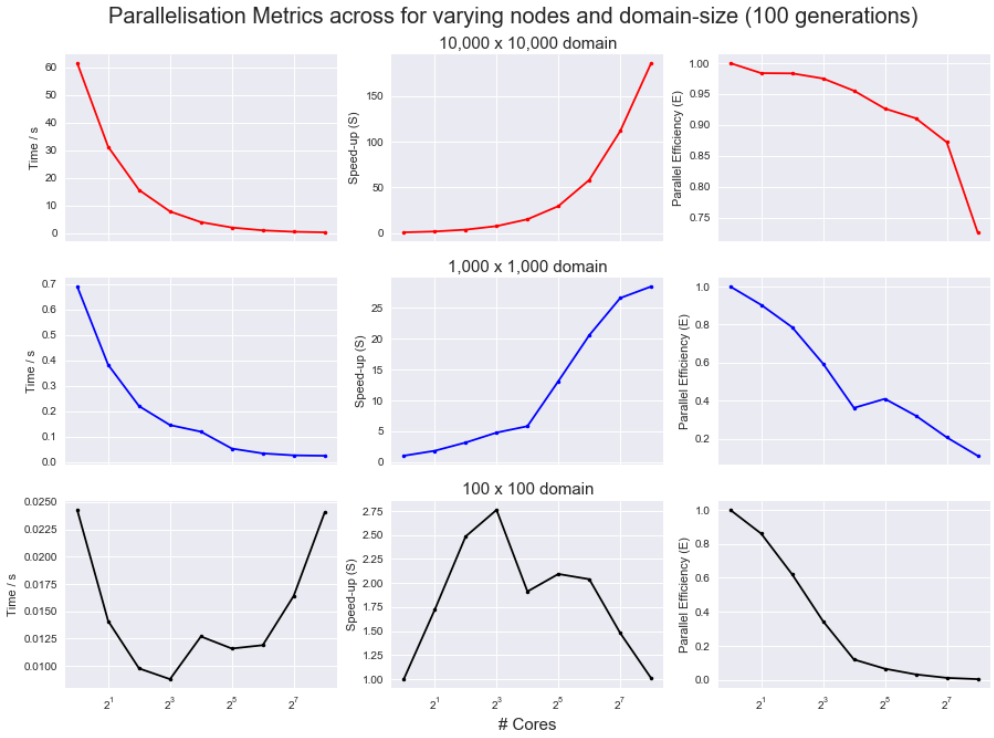

Figure 1: Parallel performance metrics for varying domain sizes using periodic boundary conditions.

For larger-domains (e.g. top row) the speed-up is extremely apparent, where the computational time falls at an average rate of 0.51981 for every doubling of cores used. The speed-up graphs (middle column) express speed-up as a fraction of parallel divided by serial computational times for each node used. From this metric, the benefits of parallelism become apparent. However, for smaller domain sizes, increasing the number of cores becomes increasingly detrimental to performance above a certain threshold (16 cores in the 100×100 case). This is due to the costs associated with communicating between cores, which becomes more significant when the boundary : volume ratios increase for smaller sub-grids. This is attested by the parallel efficiency measurements in the right column, which indicates the fraction of time that the processor is doing useful work. As the number of cores increases, the boundary : volume ratio of their local grids increases, meaning the processors spend an increasing amount of time communicating. Moreover, there is also the time penalty incurred simply by the sheer number of communications that have to be done per sub-grid, per time-step.

Review:

The performance of this program behaves as one would expect from a parallelised system until the communication cost becomes the dominant influence on performance. Although priority was given to performance throughout the construction of this program, it would have been worthwhile to have spent some more time attempting to reduce the latency associated with increasing communications, thereby aiming to increase the number of cores where the speed-up falls. For example, this was already remedied in part by deriving an MPI Type specific to each communication, but the downside of this is that eight communication types get generated on each core for each game generation – which leads to redundancy in nodes that correspond to the edges of a non-periodic global domain. Moreover, having to cycle through (at most) eight conditionals each time each core looks for a neighbour becomes burdensome. It would have been useful to have written the program to minimize these redundancies.