WITS UNIVERSITY

School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS3008A Course Project

Hand-out date: October 14, 2021
Due date: Monday 23:55, Nov 8, 2021

Contents

1 Introduction

1. You are expected to work individually on this project.
2. The course project consists of TWO problems, you are requested to complete both of them.
3. Hand-ins:
   (a) Source codes with `Makefile`, run scripts, job scripts (`run.sh`), and `readme` files. (**Do not submit any code which does not compile!**)
   (b) Report —
   - Parallelization approaches; you may use pseudo codes to describe your parallelization
   - Experiments and performance evaluation
   - Evaluation results and discussions.
4. In your report, proper citations and references must be given where necessary.
5. Total mark 100 which makes up 20% course weight.
6. Start early and plan your time effectively. Meet the deadline to avoid late submission penalties (20% to 40% depending on the time for overdue).

2 Due Date

Monday, 23:55, November 8, 2021 — the submission of final report (with Turnitin report) and source codes on ulWazi course site.

3 Projects

1. **Mini-project one: Parallel Conway's Game of Life.** The Conway's Game of Life can be pictured as taking place on a two-dimensional board of cells. The cells, similar to biological cells, can live or die. The rules of Game of Life is simple. They determine

   (a) when a live cell remains alive or dies from one generation to the next;

   (b) and when a dead cell comes back to life.

   The number of alive neighbouring cells in a surrounding grid points (include 8 immediate horizontal, vertical, and diagonal neighbours) that holds a live cell is used to determine its state in the next generation according to the following:

   - A live cell remains alive if there are 2 or 3 live neighbours;
   - A live cell dies if the number of live neighbours is less than 2 (loneliness) or greater than 3 (over-crowding); and
   - A dead cell will birth a new cell if there are exactly 3 live neighbours.

   Implementing a Game of Life simulation in serial is trivial. In this mini-project, you are requested to implement a parallel Game of Life simulation using MPI for a distributed memory system. To do this, we can decompose the game board into smaller sub-boards, and then evaluate the cells in each sub-board in parallel. Apparently, the parallelization can happen within each time step (or generation), not across different generations. The only problem is the border cells. To update the border cells in each sub-board, we need information from the neighbouring sub-boards. So this means some information need to be communicated before the evaluation of the border cells in the sub-boards.

   The goal of this mini-project is to apply efficient data communication between different MPI processes to achieve performance; and evaluate the performance of your implementation via differing board size, and the number of MPI processes (or machine size).

2. **Mini-project two: Parallelization of Dijkstra's Single Source Shortest Path (SSSP) Algorithm.**

   The following is a brief introduction of SSSP taken from [1, Chap. 10].

   Dijkstra's SSSP algorithm finds the shorted paths from a single vertex s to the other vertices of a weighted graph G. It is a greedy algorithm that incrementally finds the shortest paths. Given a weighted graph $G(V, E, w)$, where V is the set of vertices, E is the set of edges, and w contains the weights, Dijkstra's algorithm is shown in Algorithm 1.

   Dijkstra's algorithm is iterative (the `while` loop in Line 9). Each iteration adds a new vertex to the shortest paths. Since the value of $I[v]$ for a vertex v may change every time a new vertex u is added

Algorithm 1 Dijkstra's algorithm

```
1:  $V_T = \{s\}$;                                          ▷ Initialize $V_T$ with the source vertex
2:  for all $v \in (V - V_T)$ do
3:      if (s, v) exists then
4:          set $I[v] = w(s, v)$;
5:      else
6:          set $I[v] = \infty$;
7:      end if
8:  end for
9:  while $V_T \neq V$ do
10:     find a vertex $u$ such that $I[u] = \min\{I[v] | v \in (V - V_T)\}$;  ▷ $I[u]$ stores the minimum cost to reach $v$ from source $s$
11:     $V_T = V_T \cup u$;
12:     for all $v \in (V - V_T)$ do
13:         $I[v] = \min\{I[v], I[u] + w(u, v)\}$;
14:     end for
15: end while
```

in V_T, it is difficult to select more than one vertex to include in the shortest paths. Thus it is not easy to perform different iterations of the `while` loop in parallel. However, each iteration can be performed in parallel as follows.

Let p be the number of processes, n be the number of vertices in the graph, and assume we are using an adjacency matrix representing the graph. Then set V is partitioned into p subsets using 1D block mapping, that is, each process has n/p (assume $n\%p = 0$) consecutive vertices (or columns in the adjacency matrix), and the work associated with each subset is assigned to a different process. Let V_i be the subset of vertices assigned to process p_i where $i = 0, 1, \ldots, p-1$. Each process p_i stores the part of the array I that corresponds to V_i. Each process then computes $I_i[v]$ for its subset of vertices during each iteration of the `while` loop. The overall minimum is then obtained over all $I_i[u]$,which will be inserted into V_T, and each process updates the values of $I_i[v]$ for its subset of vertices based on this new vertex u.

Based on this naive parallelization approach, implement parallel Dijkstra's algorithm using **OpenMP and MPI**, respectively.

NOTE: For this problem, first, you need to write an efficient function to generate random graphs. To test your serial and parallel implementations of Dijkstra's algorithm, you may consider graphs that are undirected and connected. Furthermore, to evaluate the performance of various implementations, graphs with different sizes including $k \times 64$ vertices, where k is an integer and $k \geq 1$. The values of k should be chosen depending on the available computing resource. The density (the ratio between the number of edges and the number of edges in the corresponding complete graph) of the graph can be 35%, or differing densities of your choice. Each of the test graphs you used should be written to a simple text file in a format that the first line includes the number of vertices followed by the adjacency matrix in row-major order. Your program should read from an input file to obtain the input graph.

4 General marking guide

The weights of two mini-projects are distributed as mini-project one – 40% and mini-project two – 60%. Within each project, the mark will be distributed approximately as the following.

1. In each implementation of the problems listed above, you are expected to include
   - a (correct) baseline (could be a serial implementation); [20%]
   - a parallel implementation with a validation of the correctness (by comparing the outputs of serial and parallel outputs); [40%]
2. Report (only one combined report, a PDF file, of 2-4 pages is requested): [30%]
   - The main body of the report should consists of two sections, one for each of the two mini-projects.
   - Adequate writing to describe clearly the approach to solve the problems including parallelization methods, testing and validation methods.
   - Performance evaluation by i) comparing the performance (speedup) of the baseline and parallel version; ii) testing the scalability by varying the input size or number of processing elements. (Proper tables, line graphs, or bar graphs are good ways of presenting such results.)
3. Makefiles, run scripts, job scripts (where applicable), and readme files are provided. [10%]

References

[1] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison Wesley, 2003.