Technische Universität Bergakademie Freiberg
Institut für Numerische Mathematik und Optimierung
Prof. Dr. O. Rheinbach
Dr. M. Brändel

Wintersemester 2024/25
15.01.2024

Introduction to High Performance Computing and Optimization
PVL part IV - MPI halo communication

To get points for the parts of the PVL, the C++ code, the job scripts and, depending on the task, the times achieved must be submitted. The submission takes place via the upload in OPAL in the corresponding course element. Copying or typing code from fellow students is not permitted. You must write and submit your own code. The specified submission deadline must be adhered to. Your code will be tested. Faulty code and code that delivers significantly different times than specified will result in zero points for the PVL part. So make sure that the code has been compiled on the cluster and executed on the compute nodes.

Exercise 16
This serial program (`pvl04`) simulates the Game Of Life.

```
#include <iostream>
#include <random>
#include <chrono>

void readParameters(int argc, char** argv, unsigned int &N, unsigned int &t_steps,
          double &probability){
 try{
  if(argc < 4){
   throw std::invalid_argument("No probability given.");
  } else if(argc < 3){
   throw std::invalid_argument("No #timesteps given.");
  } else if(argc < 2) {
   throw std::invalid_argument("No problem size given.");
  }
  // Check if the input string is non-positive
  std::string arg = argv[1];
  if (arg[0] == '-' || arg[0] == '0') {
    throw std::invalid_argument("Problem size non-positive.");
  }
  N = std::stoi(argv[1]);
  t_steps = std::stoi(argv[2]);
  probability = std::stod(argv[3]);
 }catch(const std::invalid_argument& e){
  std::cout << "Usage ./<exe> <unsigned int = problem size>\n";
  N = 10;
  t_steps = 10;
  probability = 0.4;
 }
}

void randomInitialization(std::vector<std::vector<int>> &board, const double &probability){
 std::mt19937 generator(0); // Mersenne Twister engine
 std::bernoulli_distribution distribution(probability); // chance to be alive at the start
 for(unsigned int i = 1; i < board.size()-1; i++){
  for(unsigned int j = 1; j < board[0].size()-1; j++){
   board[i][j] = distribution(generator) ? 1 : 0;
  }
 }
}

void printBoard(std::vector<std::vector<int>> &board){
 for (const auto& row : board) {
  for (const auto& cell : row) {
    std::cout << (cell ? "O" : ".") << " "; // "O" for alive, "." for dead
  }
  std::cout << "\n";
 }
}

void computeTimestep(const std::vector<std::vector<int>> &source,
              std::vector<std::vector<int>> &target){
 unsigned int count;
 for (unsigned int i = 1; i < source.size()-1; i++){
  for (unsigned int j = 1; j < source[0].size()-1; j++){
   count =   source[i-1][j-1] + source[i-1][j] + source[i-1][j+1]
       + source[i][j-1]                    + source[i][j+1]
       + source[i+1][j-1] + source[i+1][j] + source[i+1][j+1];
   if (count < 2)      { target[i][j] = 0; }
   else if (count == 2){ target[i][j] = source[i][j]; }
   else if (count == 3){ target[i][j] = 1; }
   else                { target[i][j] = 0; }
  }
 }
}

int main(int argc, char** argv) {

  // get problem size and time steps
  unsigned int N, t_steps;
  double probability;
  readParameters(argc, argv, N, t_steps, probability);
  std::cout << "Game of Life with N = " << N << ", " << t_steps
       << " timesteps and initial probability = " << probability << "\n";

  // declare boards
  const unsigned int padding = 2;
  std::vector<std::vector<int>> boardA(N+padding,std::vector<int>(N+padding,0));
  std::vector<std::vector<int>> boardB(N+padding,std::vector<int>(N+padding,0));

  // setup boards
  randomInitialization(boardA, probability);

  if(N < 31) printBoard(boardA);

  // computeTimesteps
  auto t_start = std::chrono::high_resolution_clock::now();
  for (unsigned int i = 0; i < t_steps; i++){
    if(i%2 == 0) { // board A -> boardB
      computeTimestep(boardA,boardB);
      if(N < 31) printBoard(boardB);
    } else { // board B -> board A
      computeTimestep(boardB,boardA);
      if(N < 31) printBoard(boardA);
    }
  }
  auto t_end = std::chrono::high_resolution_clock::now();
  std::chrono::duration<double> t_duration = t_end - t_start;
  std::cout << "Computation took: " << t_duration.count() << " seconds.\n";

  return 0;
}
```

On a square board each square cell can either be alive (1) or dead (0). In the next time step, depending on the status of its neighbors, each cell either

dies: from loneliness, if it has less than two alive neighbors,

stays: dead or alive, if it has exactly two alive neighbors,

lives: if it has three alive neighbors or is becoming alive, also with three alive neighbors, or

dies: from overpopulation, if it has more than three alive neighbors.

To compute the amount of alive neighbors all eight neighbor states are analyzed. The cells beyond the boundary of the board are simulated with a padding of one cell, which is always dead. It never changes its state.

(a) Compile and test the serial program using `./<exe> <N> 10 0.5`, with $N = 1000, 2000, 4000, 8000$. Use a job script. What timings do you get?

(b) Parallelize the program using MPI: The rows of the board are evenly distributed between the ranks. Since the neighbor values of a cell are on neighboring rows, some information that rank 0 needs is on rank 1. Here you need to implement halo communication between neighboring ranks to synchronize the data that is needed.

(c) Perform a weak and strong scaling test.

Submission deadline is 23:59 12.02.2025.