TDDC78 - Programming of Parallel Computers
Lab4 - Particle Simulation using MPI

Authors:
David Tran - Davtr766
Jakob Bertlin - Jakbe457

I. Introduction

The fourth lab in the course TDDC78 introduces a particle simulation problem. The particle simulation works by instantiating particles in a box. The particles can either interact with other particles by colliding or walls.

In this assignment, the aim is to parallelize the particle simulation using MPI and verify the gas law:

$$pV = nRT$$

where p is the pressure, V is the volume (in this case area), n is the number of particles, R is a magic constant and T is the temperature, in this case the volume will be instead of area.

II. Approach

The approach to verify the gas law. It is hypothesized that the pressure should increase if the number of particles increases as n is directly proportional to p.

This report will also discuss the choice of distribution of particles between processors given the geometry domain of the implementation.

III. Implementation

This section covers the implementation of particles and the use of MPI.

A. Particles

The implementation was written in C++ and the particles was defined by a struct. The particles was managed by a vector of pairs, where each pair contains one particle and a boolean flag of whether the particle has collided with another particle/wall or not.

B. MPI

There are many ways to implement the MPI communication, however, in this report the approach was row-wise partitioning based on the amount of processors available. An illustration can be seen in figure 1.

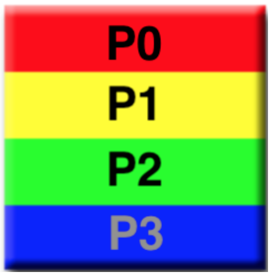

Fig. 1: Simple illustration of the box area in the particle simulation. Each chunk is assigned to one processor where the horizontal length of the chunk depends on the number of processors and the vertical length is the same as the vertical size of the box.

Every processor instantiates the same amount of particles and the particles will only move in their respective chunk area. However, if particles move over to another chunk then they have to be sent to the processor that handles that specific chunk. Therefore, the implementation requires two vectors that acts as a buffer. One vector handles particles that have to be sent up and the other vector handles particles that have to be sent down.

The communication was done using *MPI_Isend* and *MPI_recv*. The implementation is written in a way such that particles are always sent up or down even though it may not contain any particles in the buffer. The buffers were not always the same size because for each time stamp, they may contain more particles or less particles depending on the simulation. Therefore, *MPI_Probe* was used to determine how many particles were going to be received.

IV. Result

Figure 2 shows the dependence of number of cores with elapsed time and the pressure.

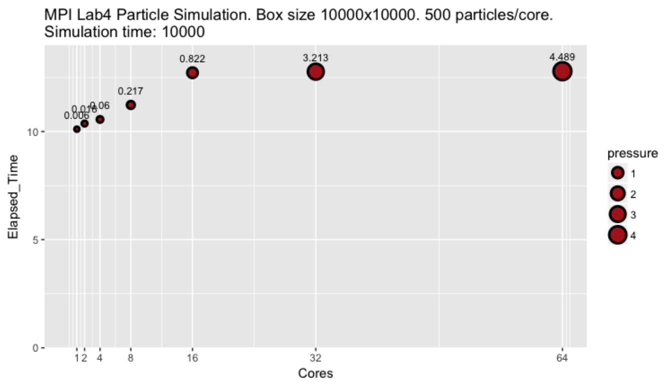

Fig. 2: Scatter plot where the x-axis are the number of cores, the y-axis is the elapsed time and the size of the points are determined by the amount of pressure.

V. Discussion

The scatter plot in section IV bolsters previous hypothesis about the gas law.

Figure 2 verifies the gas law described in section II. The pressure increases as the number of particles increases. This proves that the pressure is proportional to the number of particles. The plot also shows that the elapsed time is more or less the same. The reason for that is each processor works on the same amount of particles.

The communication of particles between processors was implemented in a way that two send operations are required, either up or down. The advantage over having a grid is that row-wise partitioning allows less communication however each communication are larger. However, in our case, less communication is better than more communication because the cost of initializing another communication operation is more expensive than sending a larger one.

APPENDIX

```cpp
#include <cstdlib>
#include <ctime>
#include <cstdio>
#include <cmath>
#include <limits>
#include <iostream>
#include <vector>
#include <mpi.h>
#include <algorithm>
#include <utility>
#include "coordinate.h"
#include "definitions.h"
#include "physics.h"


//Feel free to change this program to facilitate parallelization.

float rand1(){
  return (rand()/(float) RAND_MAX);
}

int calcRowRank(float y){
  int n_proc;
  MPI_Comm_size(MPI_COMM_WORLD, &n_proc);
  float rowSplit = (float)BOX_VERT_SIZE / n_proc;
  return (int)(y/rowSplit);
}

bool boundaryCheck(Particle &p, cord_t wall){

  if((p.y < wall.y0 || p.y > wall.y1) && (p.y > 0 && p.y < BOX_VERT_SIZE) )
    return true;
  else
    return false;
}


int main(int argc, char** argv){
  /* Define variables */
  unsigned int time_stamp = 0, time_max;
  float pressure = 0;
  int rank{}, world{}, root{0};
  std::vector<std::pair<Particle, bool>> Particles;
  std::vector<Particle> sendParticlesUp;
  std::vector<Particle> sendParticlesDown;
  cord_t wall;
  Particle *recbuf;
    double start_time{0}, end_time{0};


  /* Initialize MPI environment */
  MPI_Init(nullptr,nullptr);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &world);

    /* Create MPI data type for particles */
  MPI_Datatype MPI_Particle, oldtypes[1];
  int blockcounts[1];
  MPI_Aint offsets[1];
  offsets[0] = 0;
  oldtypes[0] = MPI_FLOAT;
  blockcounts[0] = 4;
  MPI_Type_create_struct(1, blockcounts, offsets, oldtypes, &MPI_Particle);
  MPI_Type_commit(&MPI_Particle);
  long localsum{0}, globalsum{0}, max_sent{0}, min_sent{std::numeric_limits<long>::max()}, global_max{0}, global_min{0};


  // parse arguments
  if(argc != 2) {
    fprintf(stderr, "Usage: %s simulation_time\n", argv[0]);
    fprintf(stderr, "For example: %s 10\n", argv[0]);
    exit(1);
  }

  time_max = atoi(argv[1]);

    //std::cout << "Rank " << rank << " out of " << world << "\n";

    //Initializes the wall and Particle on the root/master processor.

  // 1. set the walls
  float rowSplit = (float)(BOX_VERT_SIZE/world);
  wall.y0 = rank * rowSplit;
  wall.y1 = wall.y0 + rowSplit;
  wall.x0 = 0;
  wall.x1 = BOX_HORIZ_SIZE;


  // 2. allocate particle buffer and initialize the Particle
  Particles = std::vector<std::pair<Particle, bool>>(INIT_NO_PARTICLES);


  srand( time(nullptr) + 1234 );

  float r, a;
  for(int i=0; i<INIT_NO_PARTICLES; i++){
    // initialize random position
    Particles[i].first.x = static_cast<float>(wall.x0 + rand1()*BOX_HORIZ_SIZE);
    Particles[i].first.y = (wall.y0 + rand1()*rowSplit);

    // initialize random velocity
    r = rand1()*MAX_INITIAL_VELOCITY;
    if(r > 50) r = 49;
    a = static_cast<float>(rand1()*2*PI);
    Particles[i].first.vx = r*cos(a);
    Particles[i].first.vy = r*sin(a);
  }
  MPI_Request req[2];
  MPI_Status statUp, statDown, statRec;
  Particle *sendBufDown = new Particle[1], *sendBufUp = new Particle[1];
  int sendcountDown, sendcountUp;


    start_time = MPI_Wtime(); //start MPI::Wtime();
  /* Main loop */
  for (time_stamp=0; time_stamp<time_max; time_stamp++) { // for each time stamp


    /* Initialize values */
    for(auto& p : Particles)
      p.second = false;

    sendParticlesUp.clear();
    sendParticlesDown.clear();

    /* Main collision loop */
    for(auto p = Particles.begin(); p != Particles.end(); ++p) { // for all Particle
      if(p->second) continue;

      /* check for collisions */
      for(auto pp=p+1; pp != Particles.end(); ++pp){
        if(pp->second) continue;

        float t=collide(&(p->first), &(pp->first));
        if(t!=-1){ // collision
          p->second = pp->second = true;
          interact(&(p->first), &(pp->first), t);

          /*****************************
           * For LAB 5 TOTALVIEW Debug *
           *****************************/
          /*if(boundaryCheck((p->first), wall)) {
            if(calcRowRank(p->first.y) < rank)
              sendParticlesUp.emplace_back(p->first);
            else
              sendParticlesDown.emplace_back(p->first);

            //std::cout << "Erasing... \n" ;
            Particles.erase( p );
          }
          if(boundaryCheck(pp->first, wall)) {
            if(calcRowRank(pp->first.y) < rank)
              sendParticlesUp.emplace_back(pp->first);
            else
              sendParticlesDown.emplace_back(pp->first);

            //std::cout << "Erasing... \n" ;
            Particles.erase( pp );
          }
          */

          break; // only check collision of two Particle
        }
      }

    }


    // move Particle that has not collided with another
    for(auto p = Particles.begin(); p != Particles.end();) {
      if(!p->second){
                feuler(&(p->first), 1);
                pressure += wall_collide(&(p->first), wall);
            }

      if(boundaryCheck(p->first, wall)) {
        if (calcRowRank(p->first.y) < rank)
          sendParticlesUp.emplace_back(p->first);
        else
          sendParticlesDown.emplace_back(p->first);

        p = Particles.erase(p);
      }
      else{
          ++p;
      }

    }


    // Send to another process
    if(rank != root) {
      //std::cout << "Sending " << sendParticlesUp.size() << " from rank " << rank << "... \n" ;
      delete[] sendBufUp;
      sendcountUp = static_cast<int>(sendParticlesUp.size());
      sendBufUp = new Particle[sendcountUp];
            max_sent = sendcountUp > max_sent ? sendcountUp : max_sent;
            min_sent = sendcountUp < min_sent ? sendcountUp : min_sent;
            localsum +=sendcountUp;
      for(int i=0; i<sendParticlesUp.size(); i++)
        sendBufUp[i] = sendParticlesUp[i];

      MPI_Isend(sendBufUp, sendcountUp, MPI_Particle, rank-1, 0, MPI_COMM_WORLD, &(req[0]));
    }


    if(rank != world-1) {
      //std::cout << "Sending " << sendParticlesDown.size() << " from rank " << rank << "... \n" ;
      delete[] sendBufDown;
      sendcountDown = static_cast<int>(sendParticlesDown.size());
      sendBufDown = new Particle[sendcountDown];
            max_sent = sendcountDown > max_sent ? sendcountDown : max_sent;
            min_sent = sendcountDown < min_sent ? sendcountDown : min_sent;
      localsum += sendcountDown;
      for(int i=0; i<sendParticlesDown.size(); i++)
        sendBufDown[i] = sendParticlesDown[i];

      MPI_Isend(sendBufDown, sendcountDown, MPI_Particle, rank+1, 0, MPI_COMM_WORLD, &(req[1]));
    }


    if(rank != root){
      MPI_Probe(rank-1, MPI_ANY_TAG, MPI_COMM_WORLD, &statRec);
      int recCount{};
      MPI_Get_count(&statRec, MPI_Particle, &recCount);
      //std::cout << "Received " << recCount << " from rank: " << statRec.MPI_SOURCE <<  "\n";
      if(recCount != 0)
        recbuf = new Particle[recCount];
      MPI_Recv(recbuf, recCount, MPI_Particle, statRec.MPI_SOURCE, statRec.MPI_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      if(recCount != 0) {
        for (int i = 0; i < recCount; i++)
          Particles.emplace_back(std::make_pair(recbuf[i], false));

        delete[] recbuf;
      }
    }


    if(rank != world-1){
      MPI_Probe(rank+1, MPI_ANY_TAG, MPI_COMM_WORLD, &statRec);
      int recCount{};
      MPI_Get_count(&statRec, MPI_Particle, &recCount);
      //std::cout << "Received " << recCount << " from rank: " << statRec.MPI_SOURCE <<  "\n";
      if(recCount != 0)
        recbuf = new Particle[recCount];
      MPI_Recv(recbuf, recCount, MPI_Particle, statRec.MPI_SOURCE, statRec.MPI_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      if(recCount != 0) {
        for (int i = 0; i < recCount; i++)
          Particles.emplace_back(std::make_pair(recbuf[i], false));

        delete[] recbuf;
      }
    }

    MPI_Barrier(MPI_COMM_WORLD);

    }


  // Get total pressure from all processes
  float totalpress = 0;
  int n = (int)Particles.size(), totalp = 0;

  //std::cout << localsum << std::endl;
  MPI_Reduce(&pressure, &totalpress,  1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&localsum, &globalsum, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&min_sent, &global_min,1, MPI_LONG, MPI_MIN, 0, MPI_COMM_WORLD);
    MPI_Reduce(&max_sent, &global_max,1, MPI_LONG, MPI_MAX, 0, MPI_COMM_WORLD);
  MPI_Reduce(&n, &totalp,  1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);


  end_time = MPI_Wtime(); //start MPI::Wtime();
  if(rank == root){
    std::cout <<"Size of box: " << BOX_HORIZ_SIZE << "x" << BOX_VERT_SIZE << "\n";
    std::cout << "Total particles "  << totalp << "\n";
    printf("Average pressure = %f\n", totalpress / (WALL_LENGTH*time_max));
        printf("Elapsed time = %f seconds\n", end_time-start_time);
        std::cout << "Average sent particles = " << globalsum/time_max << "\n";
        std::cout << "Minimum particles sent = " << global_min << "\n";
        std::cout << "Maximum particles sent = " << global_max << "\n";
    }


  MPI_Finalize();
  return 0;
}
```