Laboratory Work
in
Programming of Parallel Computers TDDC78

Usman Dastgeer,Fredrik Berntsson,Mikhail Chalabine,Lu Li
Dept. of Computer and Information Science
Linköping University
58 183 Linköping Sweden
firstname.lastname@liu.se

1 Introduction

The purpose of the laboratory work is to get "hands on" experience in programming parallel computers. You will implement programs to solve your assignment on the different architectures using communication primitives that are characteristic for the specific paradigm on each computer. The work should be done in groups of two students. Every participant should clearly understand the assignment, work with the implementation, and understand the algorithms and solutions. Some additional information regarding the programming exercises can be found on the course homepage[1].

2 Assignments

Table 1 describes what you are supposed to do.

Lab No.	Lab Name	Sections
1	Image filter (`MPI`)	4, 4.4.1
2	Image filter (`PThreads`)	4, 4.4.2
3	Heat Equation (`OpenMP`)	5
4	Tools	6
5	Particles	7

Table 1: *Required assignments for TDDC78 course.*

2.1 Report Requirements

To pass the laboratory part of the course, you hand in reports (one lab report per laboration assignment) consisting of the following three parts[2]:

1. Description of your program(s) and how you have parallelized it (i.e., high-level overview of your parallel program). Describe also, how, and when data is communicated.
2. Execution times of your program(s) when using various number of processors or various problem-sizes (i.e., size of the input). When doing measurements, you can try the following combinations:
   - Changing number of processors (threads) while keeping the problem size constant (i.e., will measure the ability to solve a problem faster by putting more resources into it).
   - Changing problem size while keeping number of processors (threads) constant (i.e., will measure the scaling behavior of your solution when we have a fixed number of resources).
   - Changing both (problem size, number of processors/threads) in proportions (i.e., will measure the scaled speedup).

[1] http://www.ida.liu.se/~TDDC78
[2] Report for Lab 4 (i.e., about Tools) does not follow the specified structure. See Section 6 for further details.

Try to make the measurements as precise as possible. Draw curves/graphs that show the variations in the execution time. Describe results depicted in the curves/graphs. Especially explain any un-usual behavior (e.g., large performance fluctuations etc.) that you have seen in your curves/graphs.
Also, try to calculate the number of required Floating-point Operations per Second (FLOPS) or Integer Operations per Second (IOPS) needed to solve the problem. If no floating point operations are found, try to calculate integer operations per second instead. This is feasible for the image filter labs. Given this number and the execution times you should calculate the obtained MFLOPS for your implementation on the different computers.

3. Annotated source-code of your program(s). It is important that your programs, in some way, can reveal how they work both during execution and in the code (e.g. comments in the latter case).

In addition to above points, there could be some extra (reporting) requirements described for each lab in the corresponding lab section.

You can send lab report (pdf) and source-code files for each lab as a single archive file. In that case you do not need to print anything or append source-code to your report.

2.2 Demonstration Requirements

Besides submission of lab reports, you should also demonstrate each lab assignment (except Lab 4) to your laboratory assistant. During demonstration, each group member should be able to explain all of their programs.

3 Working on NSC's super-computer.

The following parallel computer at NSC is available for the laboratory assignments:

- Triolith - an advanced PC cluster.

3.1 Organizational issues

To do laboratory assignments, you need to secure an account at the Triolith. The procedure for getting an account on NSC machines (including Triolith) has been changed significantly since 2011. Please read and follow the updated instructions given on: `www.ida.liu.se/~TDDC78/nsc.shtml`.

As soon as you have your account information you can login to the NSC computer using secure shell:

```
ssh -l userName triolith.nsc.liu.se
```

In the following sections some basic information about the computer used in this course is provided. It is, however, strongly recommened to read through the user guides and FAQs available on the NSC's web pages at `www.nsc.liu.se`. See also:
NSC support: `www.nsc.liu.se/support`,
Triolith user-guide: `www.nsc.liu.se/systems/triolith` and
Triolith application software documentation: `www.nsc.liu.se/systems/triolith/software`.

3.2 Triolith Architecture

Following description is taken from NSC triolith user-guide:

The cluster Triolith currently consists of 1600 compute nodes, running the CentOS 6.x x86_64 operating system. Each compute node has two Intel® E5-2660 (2.2 GHz Sandybridge) processors with 8 cores each, i.e 16 cores per node. Hyper-threading is not enabled. 12 of the compute nodes have 256GB memory each[3], 56 have 128GB memory each, and the remaining 1532 have 32GB each. The fast interconnect is Infiniband from Mellanox® (FDR IB, 56 Gb/s) in a 2:1 blocking configuration. Each Triolith compute node has a 500GB local hard disk, of which 440GB is available as temporary scratch space for user jobs.

3.2.1 Software

As most other clusters do, Triolith is running Linux and uses SLURM resource management system with fairshare scheduler. For the development of parallel programs Intel MPI (recommended) and OpenMPI are available. There are supported compilers for C (icc), C++ (icpc) and Fortran (ifort).

Please read Triolith user guide to know more about compilation of C/C++, Fortran, OpenMP and/or MPI code. To know more about softwares available at Triolith , see application software documentation.

[3] These 12 compute nodes each with 256GB memory are available on special request.

4 Image Filter

The assignment consist of implementing two simple image transformation algorithms. The transformations are from an input image to an output image. There is source code available in the directory `~TDDC78/src/lab1,2_filters` on IDA network that provide serial implementations of the two transformations described below.

Use as much as you like of the given serial code. However, observe that the code is not cache optimized and is that the structure of the code is not intended to have any similarity to suitable structures for parallel implementations (this does not mean that it necessarily have a bad structure for a parallel implementation).

4.1 Averaging Filter

The first transformation makes the image "blurry". The algorithm works as follows:

> The value for a pixel (x, y) in the output image is the normalized weighted sum of all the pixels in a rectangle in the input image centred around (x, y).

In other words, the pixel in the output image is an average of the rectangle-shaped neighbourhood of corresponding pixel in the input image. To be more precise:

$$c_o(x_0, y_0) = \frac{\sum_{x=x_0-r}^{x_0+r} \sum_{y=y_0-r}^{y_0+r} w(x - x_0, y - y_0) c_i(x, y)}{\sum_{x=x_0-r}^{x_0+r} \sum_{y=y_0-r}^{y_0+r} w(x - x_0, y - y_0)}$$

Where

- $c_i(x, y)$ is the color of the pixel (x, y) in the input image;
- $c_o(x_0, y_0)$ is the color of the pixel (x_0, y_0) in the output image;
- r - size of the averaging rectangle
- $w(x - x_0, y - y_0)$ the weights distribution function.

The weight function used in the lab outputs coefficients of Gaussian (normal) distribution.

Your implementation should work with different sizes of the rectangle. Therefore, it is recommended that the function implementing this transformation take parameters specifying the size.

4.2 Thresholding Filter

The second transformation is a thresholding filter. The filter computes the average intensity of the whole input image and use this value to threshold the image. The result is an image containing only black and white pixels. White for those pixels in the input image that are lighter than the threshold and black for those pixels in the input image that are darker than the threshold.

4.3 Image Format

There are some images available in the directory `~TDDC78/images` on the IDA network. These images are saved in the ppm format, the binary version. (see manual pages on Sun workstations for description of this format). The serial code examples contain code to read ppm-files and write ppm and pgm-files. Your implementations should work with the images `im1.ppm`, `im2.ppm`, `im3.ppm` and `im4.ppm`. They are of different sizes but you can assume that they all are at most by 3000×3000 pixels.

4.4 Platform

You have to implement the same filter functionality using two different programming models on the same platform:

4.4.1 Triolith and MPI

In MPI, the communication can be implemented in different ways. Choose a suitable method and motivate the choice in the report you hand in. You may use C, C++ or Fortran for the lab.

Use MPI_Wtime to measure the execution time.

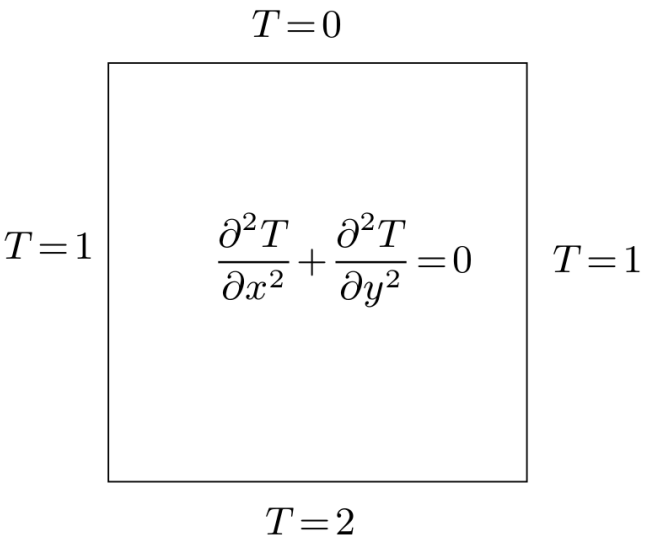

Figure 1: Boundary conditions.

4.4.2 Triolith and pthreads

In the beginning it might be easier to work on your local workstation - the pthreads library is portable. Later, when the algorithms works, move the program to the Triolith . Verify that it works and that you can utilise up to 16 processor cores. You can use C or C++ for the lab. Note that you must use the batch queue.

Use `clock_gettime()` to measure the execution time.

5 Stationary Heat Conduction Using OpenMP

In this assignment you will solve a stationary heat conduction problem on a shared memory computer (a single compute node on `triolith.nsc.liu.se`), using OpenMP. A serial code for solving the problem is given in the file `laplsolv.f90`, which is found on the course home page[4], and should be used as a starting point for your implementation. Your final parallel program should produce exactly the same results as the serial code does.

5.1 Description of the problem and the numerical method

The purpose of this section is to explain briefly the numerical details of the code that you are going to use in this excercise. It is not necessary to understand all parts of this discussion.

The problem we are going to solve is the following: Find the stationary temperature distribution in the square $[0,1]\times[0,1]$, if the temperature at the boundary is specified as in Figure 1. The stationary temperature is described by the differential equation

$$\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}=0, \qquad 0<x,y<1.$$

We introduce an equidistant grid $\{(x_i,y_j)\}_{i,j=0}^{N+1}$, as seen in Figure 2, and discretize the differential equation using finite differences. Thus the differential equation is replaced by a system of linear equations,

$$-4T_{i,j}+T_{i+1,j}+T_{i-1,j}+T_{i,j-1}+T_{i,j+1}=0, \quad 1\le i,j\le N-1, \tag{1}$$

where $T_{i,j}=T(x_i,y_j)$. The number of unknowns is N^2, and if a large number of grid points is used the problem will be too large to solve using direct methods, e.g. Gaussian Elimination. Instead the problem is solved iteratively.

Let $T_{i,j}^k$ be the approximate temperature for grid point (x_i,y_j) at the kth iteration. The next iterate $T_{i,j}^{k+1}$ is computed by

$$T_{i,j}^{k+1}=(T_{i+1,j}^k+T_{i-1,j}^k+T_{i,j-1}^k+T_{i,j+1}^k)/4, \qquad 1\le i,j\le N-1.$$

Thus the new approximation of the temperature at grid point (x_i,y_j) is obtained by taking the avarage of the values at the neighbouring gridpoints. This particular iterative method is known as the Jacobi method, see Section 5.3.

[4] www.ida.liu.se/~TDDC78/labs

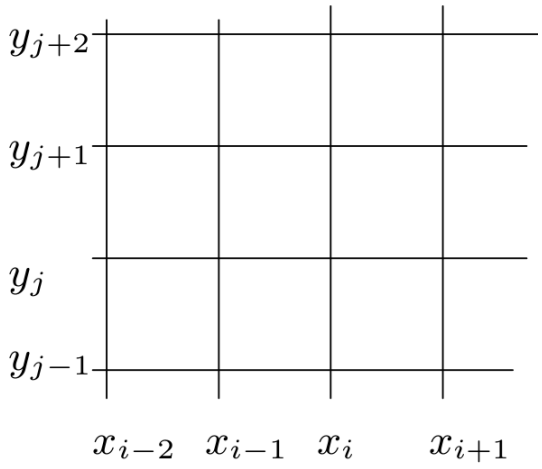

Figure 2: The computational grid.

5.2 Description of the code

The serial program is available in the course home page (see previous page for link).

The temperature data are stored in an $(n+2)\times(n+2)$ matrix,

$$T = \begin{pmatrix} T_{0,0} & T_{0,1} & \dots & T_{0,n} & T_{0,n+1} \\ T_{1,0} & T_{1,1} & \dots & T_{1,n} & T_{1,n+1} \\ \vdots & \vdots & & \vdots & \vdots \\ T_{n,0} & T_{n,1} & \dots & T_{n,n} & T_{n,n+1} \\ T_{n+1,0} & T_{n+1,1} & \dots & T_{n+1,n} & T_{n+1,n+1} \end{pmatrix},$$

where, as previously, $T_{i,j}$ denotes the temperature at grid point (x_i, y_j). Note that only the middle part of the matrix contains unknowns since the temperatures at the boundary of the square are known. The boundary data are explicitly set at the beginning of the computation.

Suppose that the matrix `T` contain the values $\{T_{i,j}^k\}$, i.e. the approximate solution at the kth iteration. The next iterate, i.e. the values $\{T_{i,j}^{k+1}\}$, can be computed by performing the following steps:

```
tmp1=T(1:n,0)
do j=1,n
  tmp2=T(1:n,j)
  T(1:n,j)=(tmp1+T(1:n,j+1)+T(0:n-1,j)+T(2:n+1,j))/4.0
  tmp1=tmp2
end do
```

The temporary vectors, `tmp1` and `tmp2`, are necessary since the old values in column j are needed for computing the new values in the $(j+1)$th column.

The assignment is to parallelize the provided serial code using OpenMP. When parallelizing, please consider the following:

1. You should only use at most $\mathcal{O}(N)$ (e.g., $2*N$ and $100*N$ are valid examples of $\mathcal{O}(N)$) additional memory where N is one side of the square. This means that you cannot create, for example, a copy of the T matrix as it would result in $O(N^2)$ extra memory usage.
2. Do not run any computations with more than 100×100 grid points before you are convinced that your code works.
3. Make sure that your parallel code produce exactly the same results as the serial code.
4. The error estimate that is used as a stopping rule for the iteration is a bit tricky to parallelize.

Use the compiler option `-openmp` when you compile your program. The number of processors is set by executing the command `export OMP_NUM_THREADS=p`, where `p` is the number of procerssors.

5.3 The Jacobi method

Usually linear systems that originate from the discretization of partial differental eqution are too large to be solved by direct methods. Instead iterative methods are used. In this section we discuss the Jacobi method, which is one of the simplest (and least efficient) iterative methods for solving linear systems of equations. For simplicity we restrict the discussion to 3×3 matrices.

Consider linear system $Ax=b$, where

$$A=\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}, \quad b=\begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}.$$

The starting point of the Jacobi method (and other iterative methods) is the splitting of the matrix A into two parts, $A=N-M$, representing the diagonal and off–diagonal elements respectively, i.e.

$$N=\begin{pmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{pmatrix}, \text{ and, } M=\begin{pmatrix} 0 & -a_{12} & -a_{13} \\ -a_{21} & 0 & -a_{23} \\ -a_{31} & -a_{32} & 0 \end{pmatrix}.$$

Using this notation the original system of equations, $Ax=b$, can be written as

$$Nx=Mx+b, \quad \text{or,} \quad x=N^{-1}Mx+N^{-1}b.$$

The Jacobi method is based on the above formula. Given an approximation x^k of the solution we compute a (hopefully) better approximation x^{k+1} by

$$x^{k+1}=N^{-1}Mx^k+N^{-1}b.$$

The Jacobi method is very inefficient and should not be used for real–life problems. However, it is as difficult to parallelize as the more sophisticated iterative methods.

6 Tools

During this lab you will try two different tools: a parallel debugger TotalView and a tracing tool Intel Trace Analyzer and Collector, ITAC.

6.1 Debugging with TotalView

The TotalView debugger is a powerful, sophisticated, and programmable tool that allows you to debug, analyze, and tune the performance of complex serial, multiprocessor, and multithreaded programs. TotalView is currently the leader in parallel programmes debugging.

Due to the license limitation it is not possible for the whole class to use the debugger simultaneously. You are therefore encouraged to use the debugger actively whenever you discover a problem with your program in any other lab.

To find out where TotalView is installed and how to use it, see:

http://www.nsc.liu.se/systems/triolith/software/triolith-software-apps-totalview.html

Moreover, extensive information on how to use different features of the debugger is available in the User Guide accessible at:

`/software/apps/totalview/8.11.0/install0/doc/pdf/TotalView_User_Guide.pdf`

and from the Help menu.

Your report should contain a short review of the features that you have used. Compare TotalView with other debuggers that you used before. Comment on ease of use, manageability for parallel application debugging, etc. Your report should also contain a successful story where you use TotalView to solve a real or artificial bug in MPI program at your choice, possibly your Lab 1 code. Include screenshots for key steps.

6.2 Tracing MPI program with ITAC

ITAC is an interactive visualization tool designed to analyze and debug parallel programs, in particular message passing programs using the MPI interface. ITAC is installed on Triolith. The full user documentation can be found on Triolith at:

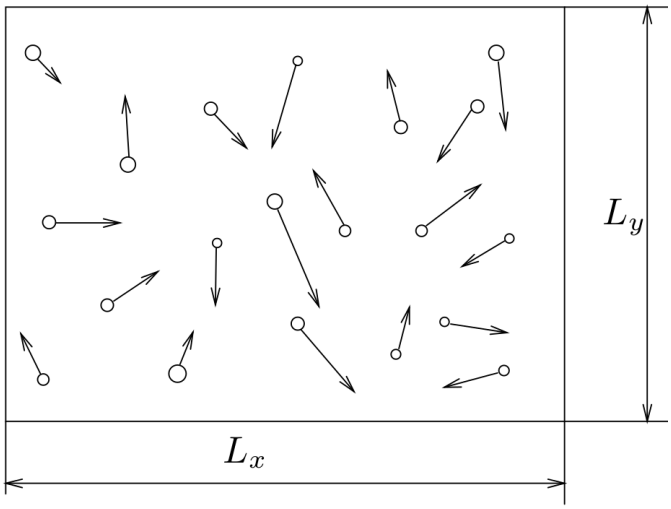

Figure 3: Gas simulation by rigid bodies.

`/software/intel/itac/$VERSION/doc`

OBS: Your task is to trace a MPI program at your choice by ITAC, possibly your Lab 1 code. Your report should contain a successful story where you use ITAC to find a performance bottleneck of your MPI program, e.g. by timeline, profiles etc. Include screenshots for key steps.

6.2.1 Running program with MPI and ITAC

For information about how to use ITAC on Triolith:

http://www.nsc.liu.se/systems/triolith/software/triolith-software-intel-itac.html

In this lab, you need to use the following functions to generate traces showing how much time the program spends working on different parts of the algorithm (collision analysis, communication, synchronization):

- use VT_classdef() and VT_funcdef() to define symbol name
- use VT_enter() and VT_end(), to record function begin and end.

Use a counter to record the pressure development on different processors:

- use VT_countdef() to define a counter at program startup
- use VT_countval() to log the value.

7 Particle Simulation

In this assignment we will do a particle simulation and verify the gas law $pV = nRT$. The particles are hard with a radius 1 and all collisions will be regarded as perfectly elastic (with the walls and other particles) and no friction is present in the box. The box will be a 2 dimensional rectangle (the collisions will be easier to handle).

Until a collision occur the particles will travel straight (no external forces) and if a collision occur the momentum and energy is conserved, by the elastic collision. From the following relationships the velocity after the collision can be found, m_1, m_2 is the mass of the particle and $\hat{v}_{(x,y)}$ is the velocity before the collision and $v_{(x,y)}$ after, see Figure 4. The law of conservation of the momentum is, after a suitable rotation of the coordinate system,

$$m_1 v_{1,x} + m_2 v_{2,x} = m_1 \hat{v}_{1,x} + m_2 \hat{v}_{2,x}$$

and the kinetic energy

$$m_1(v_{1,x}^2 + v_{1,y}^2) + m_2(v_{2,x}^2 + v_{2,y}^2) = m_1(\hat{v}_{1,x}^2 + v_{1,y}^2) + m_2(\hat{v}_{2,x}^2 + v_{2,y}^2).$$

The coordinate system is rotated so the tangent to the collision point is vertical so there is no change in the velocity in the y-direction.

When a particle hits wall the particle will bounce back with negative velocity normal to the surface.

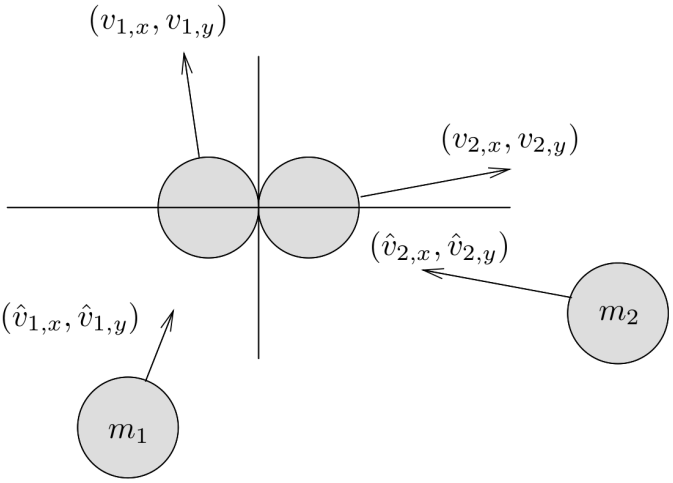

Figure 4: Interaction between two particles

With this simulation one can simulate the notion of pressure, the bouncing particles will exhibit a pressure on the walls each time they hit. Each time a particle hits a wall a momentum of $2mv_{x,y}$ will be absorbed by the wall. If we sum all collisions by a wall during t second and divide this by the circumference of the box and t, we will obtain the (two dimensional) pressure in the box. You will use this pressure to verify the pressure law $pV = nRT$, p pressure, V volume (here area), n number of moles (number of particles), R magic constant and T is the temperature, in our case the volume will be instead area.

7.1 Implementation

Write the framework for the simulation of small particles in a rectangular box. Choose and motivate a good distribution of the particles between the processors, implement the communication between the processors using MPI, count the pressure. You may use either Fortran or C/C++ as the implementation language.

7.1.1 Data types

The particles in the provided functions are represented by the following struct

```
struct part_cord {float x; float y; float vx; float vy;}
typedef struct part_cord pcord_t;
```

x, y is the position, v_x, v_y the velocity. The walls can be represented by

```
struct cord {float x0; float x1; float y0; float y1;}
typedef struct cord cord_t;
```

These datatypes are defined in the file `coordinate.h`. The particles can be stored on each processor in a fix array.

7.1.2 Functions provided

The interaction between the particles is provided by the following routines in the file `physics.c`;

```
float collide (pcord_t *p1,pcord_t *p2)
interact (pcord_t *p1,pcord_t *p2,float t)
float wall_collide (pcord_t *p, cord_t wall)
feuler (pcord_t *a, float time)
```

The routine `collide` returns -1 if there will be no collision this time step, otherwise it will return when the collision occurs. This will then be used as one of input parameter to the routine `interact`. The routine `interact` moves two particles involved in the collision. Do **not** move these particles again. `wall_collide` checks if a particle has exceeded the boundary and returns a momentum. Use this momentum to calculate the pressure. The routine `feuler` moves the a particle.

7.1.3 Important implementation issues

The files necessary for the implementation are available on the course home page[5]. Download the files to your working directory on Triolith. Note that the provided functions are written in C. One of the functions is available in Fortran, but you need to translate the others if you want to use Fortran 90. Some simplifications to the model can be made. If the particles are small compared to the box and the time step is short, the possibility that a particle will collide with more than one other particle is statistically very small, so if a particle hits another, we can update both and ignore them until the next time step (this is done in the procedure `interact`). Observe that this can be implemented without some sort of update flag array! Depending on how the particles are distributed over the processors some simplifications of the communication can be done, motivate each simplification done!

- Each time-step must be 1 time unit long.
- The initial velocity should be less then 50. Use the random number generator to generate the absolute velocity and a starting angle. (r=rand()·max_vel; θ=rand()·2π; $v_x = r\cos(\theta)$; $v_y = r\sin(\theta)$)
- Typical numbers for the simulation; number of particles = 10000 × number of processors, and area of the box = $10^4 \cdot 10^4$.
- The pressure can be found at the end of the simulation by dividing the total momentum from the routine `wall_collide` with the number of time-steps and the length of the circumference of the box.
- Think about pros and cons of data structure (arrays, linked list etc.) that you use to represent particles.
- Avoid unnecessary communication by sending all particles at once.

7.2 A short summary of the structure of the program

- Initiate particles
- Main loop: for each time-step do
  - for all particles do
    - Check for collisions.
    - Move particles that has not collided with another.
    - Check for wall interaction and add the momentum.
  - Communicate if needed.
- Calculate pressure.

7.3 Questions

Besides requirements mentioned in Section 2.1, you should measure and report the following:

1. Explain your choice of distribution of the particles over the processors. Is there an optimal relation between the distribution and the geometry of the domain? Measure this by counting particles passed between processors in each time step.
2. Verify the gas law $pV = nRT$ by changing the number of particles (n) and size of the box (V) and then measure the pressure.

[5] www.ida.liu.se/~TDDC78/labs