Tools of high performance computing 2024

Final project

In the final project you are supposed to write a parallel program based either on *your own problem* or on one of the four problems given on the following pages. In the case of your own problem you should first discuss with the lecturer to see if its is suitable as a final project.

The deadline for submitting the report to Moodle is 20.5.2024 at 23:59

You write a report of about 10 pages on your work. The report should consists of (the number gives the weighting used in grading the report and code; maximum is 100 points)

1. Problem description 5
2. Computer science background 20
   a) Description of algorithms used
   b) Principles of parallelization
3. Documentation 20
   a) Presentation of the code
   b) Instructions for using the code
4. Benchmarking 20
   a) Details of the computing environment
   b) What did you measure, how did you measure, analysis
5. Conclusions 15
   a) Summary of the work.
   b) Possible improvements.

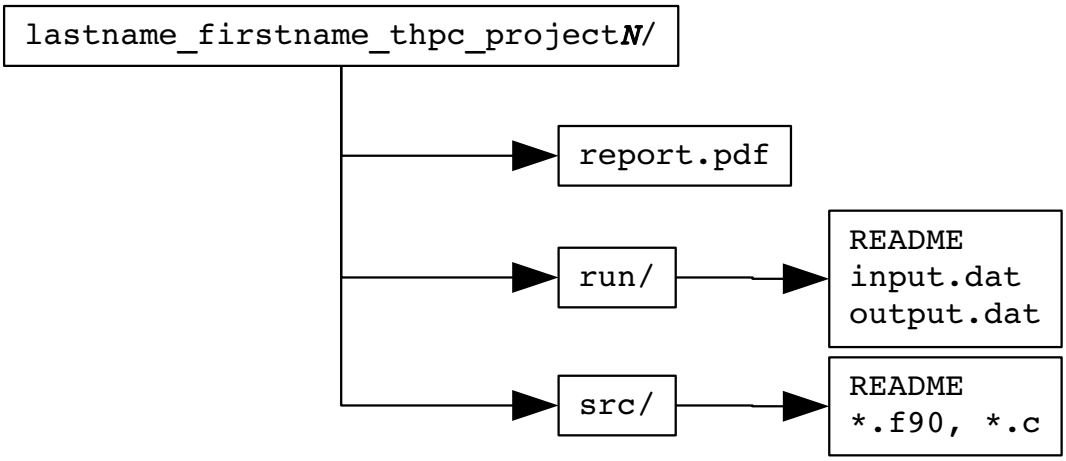
The code implementation (readability, correctness) has the weight of 20.

Send the report in PDF format. As in the case of exercises, you should also send the full source code, possible Makefile and sample input and output files. However, source code listing may not be included in the report.

Submit your report and code to Moodle in one file as a gzipped `tar` archive of a `zip` archive named as

`lastname_firstname_thpc_project`*`N`*`.tgz` or
`lastname_firstname_thpc_project`*`N`*`.zip`

where *`N`* is the project number (see below; if you have your own title the number is 3) and *containing a folder with the same name where all the files can be found*. This folder in turn, must contain the report (file `report.pdf`) and subfolders `src/` [source code and file `README` containing short compilation instructions] and `run/` [input file(s) and example output and file `README` containing short run instructions].



You may write the program either in Fortran or C/C++. For parallelization you may use either MPI, OpenMP, threads or CUDA. *Remember to ensure that the parallel code solves the problem faster than the serial one but gives the same results.*

1. Optimization using a parallel genetic algorithm: traveling salesman problem

The optimization problem is either the 2D traveling salesman problem (TSP) or one of your own interest. In genetic algorithms (GAs) the optimization is achieved by maintaining populations of possible solutions and letting only the fittest individuals of these populations survive and reproduce[1].

In TSP one has to find a route through a given set of cities with the shortest possible length.

If we denote with $\{x_i, y_i\}$, $i = 1, \ldots, N$ positions of the cities we have to minimize the length L:

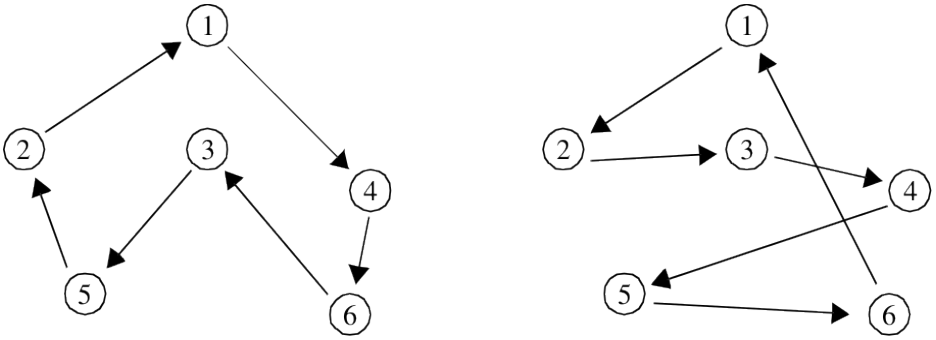
$$L^2 = \sum_{i=2}^{N} \left[(x_{R(i+1)} - x_{R(i)})^2 + (y_{R(i+1)} - y_{R(i)})^2 \right]$$

where $R(i)$, $i = 1, \ldots, N$ is the order in which the cities are traversed[2] . When applying GA to TSP we need a way to encode the route as a string. The most obvious way is to list the cities in the order they are visited in the route; i.e. the array $R(i)$. Below are shown two examples of individuals of a certain set of cities.

$$R_1 = (1, 4, 6, 3, 5, 2) \qquad R_2 = (1, 2, 3, 4, 5, 6)$$

1 In addition you may check out this thesis and, of course take a look at the Wikipedia article at https://en.wikipedia.org/wiki/Genetic_algorithm.

2 Well, to be precise we need only to specify $N-1$ elements in the route because we must return to the place where we started.



The problem in the TSP is that it is not straightforward to introduce breeding. A simple way would be to choose randomly a part of the route and swap corresponding parts in the parents to create two children. However, the children may have duplicate cities so that they are not necessarily valid routes (there are cycles in the route). Instead of randomly creating the children one can use heuristics. One possible algorithm to create one child from two parents is the following:

1. Take as the first city of the child the first one from either of the parents.
2. Choose as the second city of the child the one of the corresponding cities in the parents that is closer to the first city in the child.
3. If the new city is already included in the child choose the second city of the other parent. If this is also included in the child then choose the next city from either of the parents randomly (and in such a way that it does not introduce a cycle).
4. Go in a similar fashion through all the cities until the child has all cities.

Lets take a simple example [`d(i,j)` is the distance between cities `i` and `j`]:

```
R1=123456
R2=413265
Choose to start from R2(1) → R3=4XXXXX
Next one is either 2 or 1. Assume d(4,1)<d(4,2) → R3=41XXXX
Next one is 3 because both parents have 3 there → R3=413XXX
Next one is either 2 or 5. Assume d(3,2)<d(3,5) → R3=4132XX
Next one is either 6 or 5. Assume d(2,6)<d(2,5) → R3=41326X
The last one is 5                               → R3=413265
```

Note that periodic boundary conditions are used. That is obvious because the starting point is arbitrary: the length of the route does not depend on where it is started. So individuals

`R1=123456` and `R2=561234`

are equivalent.

Mutation of the child is performed by exchanging two cities.

GA may be parallelized either by distributing one population to processors or by letting each processor to compute its own population. In the latter case mixing of genes between the separate populations is achieved by migration[1] . In this project we choose the latter way of parallelization. In the stepping-stone model each processor sends a few of its best individuals to one of its nearest neighbors (as defined by the

1 J. Nang, K. Matsuo: A Survey of the Parallel Genetic Algorithms, Research report IIAS-RR-93-7E, Fujitsu Laboratories Ltd. (1993). (You can find the report in the Kumpula Science Library.)

processor array topology) at certain intervals. In this way the amount of communication is small and the scaling behavior of the method should be good. Of course, some global communication is necessary for obtaining global results at certain intervals and at the end of the run.

Your task: Implement the parallel TSP-GA algorithm and test its performance and scaling as a function of the number of processors. The IO and collecting of the data is handled by the root processor. In the beginning the positions of the cities are read in and in the end of the run the best route and its length is printed. In order to monitor the run it is advisable to print the so far best solution at regular intervals.

2. Parallel algorithms for solving partial differential equations: Poisson's equation in two dimensions

Assume we have the Poisson's equation in the unit square[1]:

$$\frac{\partial^2}{\partial x^2} f(x,y) + \frac{\partial^2}{\partial y^2} f(x,y) = g(x,y), \quad (x,y) \in [0,1]^2.$$

The function $g(x,y)$ is known. In addition, the boundary conditions dictate the function values at the unit square boundaries. The problem is discretized so that the unit square is divided to N parts in both directions:

$$f_{i,j} = f(\frac{i}{N}, \frac{j}{N}), \quad 0 \leq i,j, \leq N$$
$$g_{i,j} = g(\frac{i}{N}, \frac{j}{N}), \quad 0 \leq i,j \leq N.$$

The second derivative can be computed by the central difference approximation:

$$\frac{\partial^2}{\partial x^2} f(x,y) \approx \frac{1}{\Delta^2} [f(x+\Delta, y) - 2f(x,y) + f(x-\Delta, y)], \quad \Delta = \frac{1}{N}$$

A similar equation holds for y derivative and we get the equation for the interior points

$$f_{i,j} = \frac{1}{4} [f_{i+1,j} + f_{i-1,j} + f_{i,j+1} + f_{i,j-1}] - \frac{1}{4N^2} g_{i,j}$$

This is a system of $(N-1)^2$ equations and $(N-1)^2$ unknowns (boundary conditions give the function values on boundaries). This means that iterative methods developed for linear systems can be used to solve it. In the Jacobi over relaxation (JOR) method the iteration goes as

$$\begin{aligned} f_{i,j}(t+1) &= (1-\gamma) f_{i,j}(t) + \\ &\quad \frac{\gamma}{4} [f_{i+1,j}(t) + f_{i-1,j}(t) + f_{i,j+1}(t) + f_{i,j-1}(t)] - \\ &\quad \frac{\gamma}{4N^2} g_{i,j} \end{aligned}$$

$$0 < i,j < N, \; 1 < \gamma < 2.$$

Here γ is so called over relaxation parameter and its optimal value is best found by experimenting. Note that 'time' t is here only an iteration index.

Successive over relaxation (SOR) uses the Gauss-Seidel iteration algorithm:

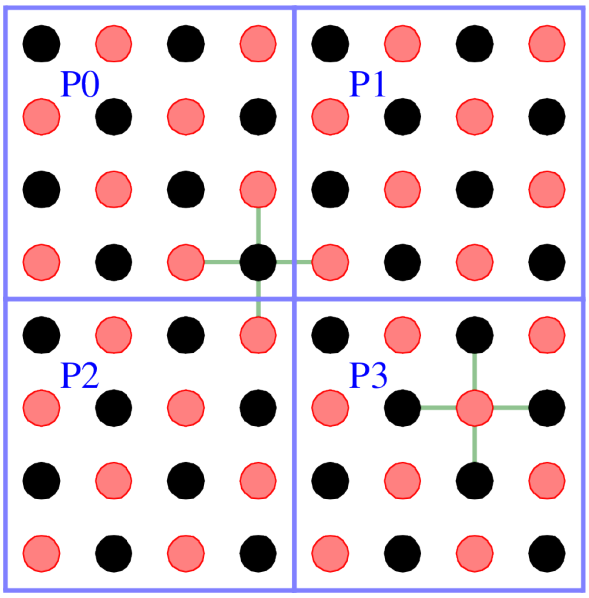
1 See e.g. paragraph 2.5 in the on-line book at http://web.mit.edu/dimitrib/www/pdc.html or chapter 19 of Numerical Recipes. You also might be interested in S. Yang, M. Gobbert: The optimal relaxation parameter for the SOR method applied to the Poisson equation in any space dimensions, *Appl. Math. Lett.* **22** (2009) 325 (http://dx.doi.org/10.1016/j.aml.2008.03.028).

$$
\begin{aligned}
f_{i,j}(t+1) \quad = \quad & (1-\gamma) f_{i,j}(t) + \\
& \frac{\gamma}{4}\left[f_{i+1,j}(t) + f_{i-1,j}(t+1) + f_{i,j+1}(t) + f_{i,j-1}(t+1)\right] - \\
& \frac{\gamma}{4N^2} g_{i,j}
\end{aligned}
$$

Note that this equation is meant to express the fact that in each iteration the newest available information is used (i.e. if there is data for $t+1$ then it is used). This means that the order in which the lattice points are traversed is important. The equation above describes a serial algorithm and can not be parallelized as such. The SOR algorithm converges fast but as in the case of JOR the optimum value of parameter γ is best found by experimenting.

Parallelization of JOR is straightforward: just do the domain decomposition of the unit square by giving each processor a small rectangle for computation. Communication between processors is needed when computing the new values of f at the boundaries of each processor's domain.

On the other hand, because of the dependencies parallelization of SOR is not so easy. One way to do it is so called red-black algorithm. In it, the lattice is divided into two sublattices by coloring every second point with red and black (see below). The algorithm first updates the red sublattice. This can be done in parallel by using the domain decomposition. Then the black points are updated using the recently calculated red points. Thus, the updating is done an alternating fashion for each sublattice at a time so that all neighbors of a lattice point have their '$t+1$' values.



Convergence criterion for the iteration may be based on the difference between two successive iterations.

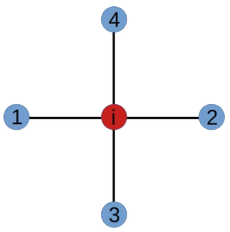
Your task: Implement the parallel Poisson's equation solver by using the SOR algorithm and test its scalability. Do the scaling tests for many system sizes. Note that you must first find out the optimum value of the over relaxation parameter γ and that its value depends on the grid size (i.e. N).

3. Simple parallel molecular dynamics simulation in two dimensions

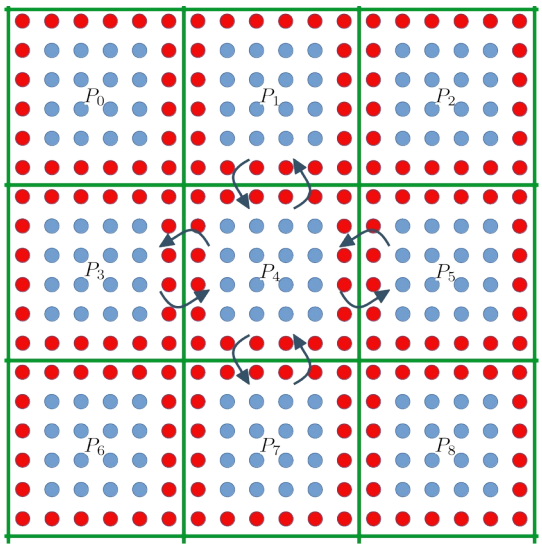
This is an extension of exercise 8 to two dimensions. Now the atoms are in a square lattice, with size `nuc`×`nuc`, (the number of atoms being `nat=nuc`^{2}), and each one interacts, or is bonded, with its four nearest neighbors. Periodic boundary conditions are applied in both directions. Attached is the serial Fortran[1] code `md2d.f90`. The interaction is the same as in exercise 8, just slightly different constants:

$$U_i = \frac{1}{2}[k_2(x_i - x_{i-1} - d)^2 + k_2(x_{i+1} - x_i - d)^2] + \frac{1}{3}[k_3(x_i - x_{i-1} - d)^3 + k_3(x_{i+1} - x_i - d)^3]$$

Remember that in this case the bonding topology does no change during the simulation. Thus, in the beginning of the run a neighbor list array `[neigh(nat,4)]` is constructed. Neighbors of each atom `[neigh(i,1:4)]` are numbered as shown on the right. Using this array the energies and forces of each atom can be easily calculated in subroutine `accel`.



Your task is make the code parallel using MPI and spatial decomposition. Each processor must communicate its edge atoms to four nearest processors as shown below.



Here the topology routines of MPI are handy. Note that here you cannot assume anything about the number of atoms. E.g. the size of the lattice if not divisible by the number of processors. This means that you must be careful when dealing with the atoms in the border processors.

Ensure that your code gives the same results[2] as the serial version and then measure its scaling as a function of number of processors. Use systems having at least 10 000 atoms.

1 Unfortunately, C++ version is not available.

2 There may be very small differences in the results. The most important thing is that the total energy of the system is conserved.

4. PIC simulation of a plasma instability

In this project, you will write a program that simulates the two-stream instability in a 1-dimensional plasma constituted of two counter-flowing beams of electrons and cold ions (not simulated). (No knowledge of the associated physics is required, requests for clarifications can be addressed to Laurent Chôné)

Your task

Implement a parallel program performing the simulation described below. Use hybrid parallelism (MPI data decomposition and OpenMP threads), and test its scalability with both increasing numbers of MPI processes and OpenMP threads.

Summary of useful parameters: (defined below)

You can adjust the number of time steps, mesh size N_x and number of particles per cell $N_{p,i}$ to find favourable scaling conditions, or fast execution during development. Other parameters are suggested as:

Δt	Δx	$N_{e,tot}/N_x$	v_0	E(x,0)
10^{-3}	1	10^{14}	2	0

Mesh

- The physical domain is a periodic line $x \in [0, L]$
- The line is discretised in a uniform mesh of $N_x + 1$ nodes $x_i = i \times \frac{L}{N_x} = i \times \Delta x$
- The line is equivalently discretised in N_x cells, where $x \in [x_i, x_{i+1}[$
- The line is periodic, so $x_{N_x+1} = x_0$

Note: normalised units are used, so Δx can be chosen of the order of 1.

With each node is associated a degree of freedom $e_i(t)$ and a basis function $\Pi(x)$ such that:

$$\Pi_i(x) = \begin{cases} 1 & \text{if } x \in [x_i, x_{i+1}[\\ 0 & \text{otherwise} \end{cases} \tag{1}$$

And the electric field on the line is:

$$E(x,t) = \sum_i e_i(t)\Pi_i(x) \tag{2}$$

Particles

Each mesh cell contains a number of particles with position and velocity $(x_p(t), v_p(t))$, which will move according to the electric force, and generate electric field according to their current.

Note: the PIC method is related to Monte-Carlo methods and subject to sampling noise. A good rule of thumb would be to use at least 10^3 particles per cells during simulations (or orders of magnitude more, the noise level scales as $\delta E \sim (N_p)^{-1/2}$). Fewer particles can be used during development/debugging.

Because particles do not interact between themselves but through the electric field, they are able to move freely along the line, and one should keep track of their position in the mesh (it is a good idea for the serial performance of the code to sort the particles by position after each time step).

Particle initialisation

The particles in the simulations do not represent individual electrons, but samples of the probability distribution function:

$$f\left(x, v, t\right) = \sum_p \frac{w}{\Delta x \Delta v} \delta(x - x_p(t)) \delta(v - v_p(t)) \tag{3}$$

where the weight w is the ratio of to total number of electrons in the simulation over the total number of simulated particles: $N_{e,tot} = w N_{p,tot}$. We consider uniformly distributed particles, so for each single cell: $N_{e,i} = \frac{N_{e,tot}}{N_x} = w N_{p,i}$.

Here let us choose $\frac{N_{e,tot}}{N_x} = 10^{14}$. You are free to choose w so that you have a reasonable number $N_{p,i}$ per particles (remember: for simulations $N_{p,i} \geq 10^3$, grow by orders of magnitudes to reduce noise).

For this particular simulations, you will initialise $N_{p,i}$ particles per cell with x_p randomly picked in $[x_i, x_{i+1}[$. Half of these particles will be initialised with a velocity $v_p(0) = v_0$, and the other half with $v_p(0) = -v_0$. That is, the simulation will contain two opposite beams of particles flowing through each other.

Time update

The particles and degrees of freedom of the electric field are evolved in time according to:

$$v_p\left(t + \frac{\Delta t}{2}\right) = v_p\left(t\right) - \frac{\Delta t}{2} \sum_i e_i\left(t\right) \Pi_i\left[x_p\left(t\right)\right] \tag{4}$$

$$e_i\left(t + \Delta t\right) = e_i\left(t\right) + \frac{w}{\Delta x} \sum_p v_p\left(t + \frac{\Delta t}{2}\right) \int_0^{\Delta t} \Pi_i\left[x_p\left(t\right) + \tau v_p\left(t + \frac{\Delta t}{2}\right)\right] \mathrm{d}\tau \tag{5}$$

$$x_p\left(t + \Delta t\right) = x_p\left(t\right) + \Delta t v_p\left(t + \frac{\Delta t}{2}\right) \tag{6}$$

$$v_p\left(t + \Delta t\right) = v_p\left(t + \frac{\Delta t}{2}\right) - \frac{\Delta t}{2} \sum_i e_i\left(t + \Delta t\right) \Pi_i\left[x_p\left(t + \Delta t\right)\right] \tag{7}$$

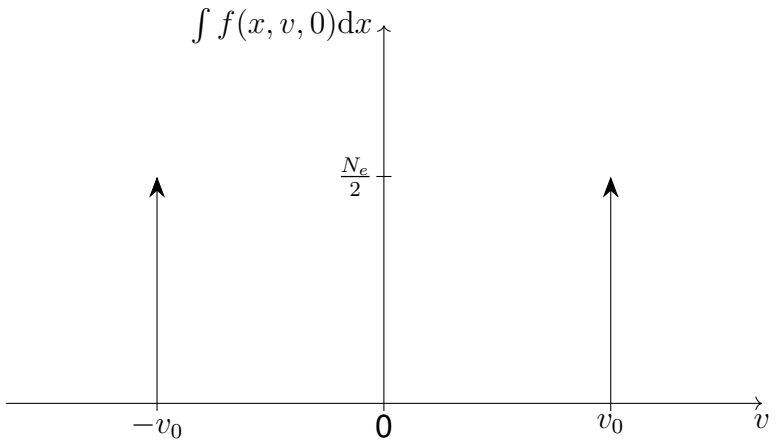


Figure 1: Probability distribution function of electrons, averaged over the simulation domain. This velocity distribution corresponds to two beams (delta functions) at $\pm v_0$. $\int f(x,v,0)\mathrm{d}x = \frac{N_e}{2\Delta v}\left[\delta\left(v-v_0\right)+\delta\left(v+v_0\right)\right]$

Note: because Π are piecewise-constant basis functions, the integral can be calculated exactly using a midpoint method. However, when the path of integration is over several elements (i.e. when the particle is crossing into a different cell during the time step), the integral should be split before each piece is evaluated with the midpoint rule.

Diagnostics

The simplest diagnostic would be to verify that the Hamiltonian is conserved (within some bounds):

$$H = \sum_i \Delta x \frac{e_i^2}{2} + \sum_p \frac{v_p^2}{2} \tag{8}$$

Periodically outputting the electric field would show the growth of a wave due to the unstable initial conditions.

Optional diagnostics

One can introduce a second kind of basis function:

$$\Lambda_i(x) = \begin{cases} \frac{x-x_{i-1}}{x_i-x_{i-1}} & \text{if } x \in [x_{i-1}, x_i[\\ \frac{x_i-x}{x_i-x_{i+1}} & \text{if } x \in [x_i, x_{i+1}[\\ 0 & \text{otherwise} \end{cases} \tag{9}$$

Which is useful to output the fluid moments (density, particle flux, total pressure):

$$\begin{aligned} n(x) &= \int f \mathrm{d}v \\ &= \sum_{i,p} \frac{w}{\Delta x} \Lambda_i(x_p) \end{aligned} \tag{10}$$

$$\begin{aligned} \Gamma(x) &= \int v f \mathrm{d}v \\ &= \sum_{i,p} \frac{w v_p}{\Delta x} \Pi_i(x_p) \end{aligned} \tag{11}$$

$$\begin{aligned} P(x) &= \int v^2 f \mathrm{d}v \\ &= \sum_{i,p} \frac{w v_p^2}{\Delta x} \Lambda_i(x_p) \end{aligned} \tag{12}$$

In general, it is not very convenient to represent the distribution function itself due to its high dimensionality, however in the present case it can be done on a 2D mesh. Assuming a Cartesian (x, v) mesh with widths $(\Delta x, \Delta v)$ (one can adjust Δv using similar techniques as to determine the number of bins in a histogram, i.e. for a given cell on the x-mesh, $N_v \approx \sqrt{N_{p,i}}$). Then, with $\Lambda_i(v)$ analogous to $\Lambda_i(x)$ the distribution function is:

$$f(x, v) = \sum_{i,p} \frac{w}{\Delta x \Delta v} \Lambda_i(x_p) \Lambda_i(v_p) \tag{13}$$

In general it is not advisable to output the phase-space coordinates of all particles, except to save the state of the simulation so that it can be restarted in a future execution of the program.