High Performance Computing
Additional Assignment B

Chiara Moret

Course of A.Y. 2021-2022 - Data Science & Scientific Computing

1 Introduction

We want to implement a program to compute a *density* and a *potential* function over N_p particles, whose coordinates belong to the cube $[0,1) \times [0,1) \times [0,1)$. On the same cube a regular 3D grid is also defined, with grid number N; this means that there are $N \times N \times N$ nodes in the grid. The functions are defined as follows:

$$density(n) = P \qquad with\ P = |\{p\ s.t.\ dist(p,n) \leq R\}|$$

$$potential(p_i) = \sum_{j=0, j\neq i}^{N_p} \frac{1}{dist(p_i, p_j)}$$

Here n denotes nodes, p denotes particles, and $dist(a,b)$ is the euclidean distance.

2 Implementation

Two main parallel codes are provided:

- A MPI implementation of the process `MPIdensity.c`
- An openMP implementation of the process `oMPdensity.c`

These programs may accept 2 or 3 input arguments: either `N R [input.bin]`, or only `N R`. If only `N R` are provided, the program will generate a random number between 10 and 1000 of particles, otherwise the particles will be read from the binary file `[input.bin]`.

As outputs, two binary files are produced: `densities.bin`, containing the density for all grid points; and `potentials.bin`, containing the potentials for all points. When not passing an input file, the randomly generated particles' coordinates will not be saved in a separate file but will still be available inside `potentials.bin`.

In addition to these we also provide some additional code:

- `particles.c` will generate N_p random variables; it accepts N_p as the only input argument and it produces a `particles.bin` file as output.
- `readBinary.c` can be used to print the results in a human-readable way; it accepts as input only binary files with the following names: `particles.bin`, `densities.bin`, `potentials.bin`; it displays the information accordingly.

Finally the bash file `density.sh` will compile all of the scripts.

2.1 Implementation choices

We chose to perform parallelization among the particles rather than the grid nodes. This allowed us to introduce a `box` function: it computes the coordinates of the nodes which may have particle p in their sphere of concern. The idea is that only the nodes for which $|x_p - x_n| <= R$ and $|y_p - y_n| <= R$ and $|z_p - z_n| <= R$ are candidates. We can find the index of those nodes in the `grid` array and therefore greatly reduce the number of nodes we need to iterate over.

To store the grid coordinates we only need an array of size N, since the grid is square. For each coordinate, the first value is always 0 and the last one is always 1.

The array of densities is very large: $N \times N \times N$, making the code memory bound; to alleviate this problem we introduced a basic form of blocking, slicing the array across the x axis; we therefore have to deal with a maximum array size of $N \times N \times blockSize$. These densities are then written into the `densities.bin` file and the memory can be freed for the next block.

2.2 openMP

If no input file is provided the code will generate a random number N_p and then N_p particles will be generated in parallel; to achieve this in a thread-safe manner `erand48()` is used, seeding with the thread id. If an input file is given, the data is read into memory serially.

The `densities.bin` file is created and populated with the grid number N.

The number and size of the blocks are computed so they can be iterated over: an array is created to store the density of each grid point in the block. The threads then loop in parallel over the particles and update the density array when the particle falls within the sphere of radius R around the gridpoint. Note that we only need to verify the condition nodes which fall within the box of the particle. The array is written into the density file and discarded. The file can be closed once all blocks have been iterated over.

The array of potentials is created outside the blocking structure, and is updated with the potential relative to the each particle, using reduction.

Finally the file `potentials.bin` is created, populate and closed. Note that all of the I/O operations are done in serial sections.

2.3 MPI

If no input file is provided the root will generate a random number N_p, divisible by the number of processors. N_p is broadcasted so that each processor can compute their share of the particles. Finally these particles are gathered by all processors. If an input file is given, the data is read into memory by the root and then broadcasted.

Note that the processors need all of the particles' coordinates to compute the potentials, however each of them only takes charge of a limited number of particles which they actually will loop over. For this reason in both cases we compute the indexes `start` and `stop`, which will determine which section of the particles' array the processor will deal with.

The root then creates the `densities.bin` file and populates it with the grid number N.

The number and size of the blocks are computed so they can be iterated over: each process creates an array of local densities to collect the density of each grid point in the block. These are local values: since each processor only takes care of a limited number of points we need to sum the local arrays element-wise to have an actual count of how many particles there are in sphere around a node. The local arrays are gathered to the root, which stores their sum in a global densities array and then writes them in `densities.bin`. All of the density arrays are then freed. The root can close the file once all blocks have been iterated over.

The potential for a given particle is computed during the iteration over the first block. The result is stored in an array of local potentials. These are concatenated into a global potentials' array by the root, which will create, populate, and close the `potentials.bin` file.

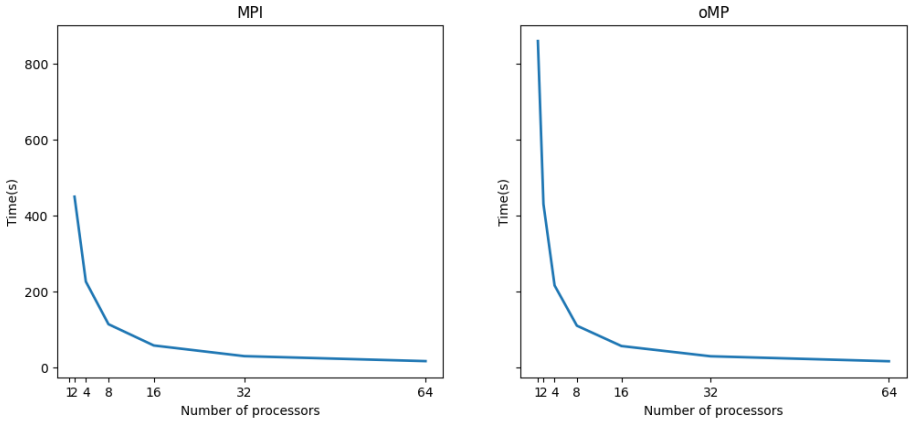
3 Requirement 1 - Scaling

3.1 Scaling N_p

Here are the results of scaling the number of particles, while both the grid number and the number of threads/processors stay the same. Here $N = 500$ and $n_{procs} = 64$. As suggested the radius varies as $R = \frac{1}{N_p^{\frac{1}{3}}}$.

N_p	MPI(avg)	openMP(avg)
10	4.180237	4.108899
10^2	2.938282	2.200009
10^3	2.933405	2.800883
10^4	3.106871	1.613888
10^5	11.618778	11.614266
10^6	871.159664	837.665143

Figure 1: Scaling the number of processors with $N_p = 10000$, $N = 1000$ $R = 0.05$.



We weren't able to run $N_p = 10000000$ particles within a reasonable time, however, that might have been achievable if we also scaled the number of processors, as both implementantions appear to scale well when increasing the number of processors (see fig 3.1).

3.2 Scaling N

Here are the results of scaling the grid number, while the number of particles, the radius, and the number of threads/processors stay the same. Here $N_p = 10000$ $R = 0.05$and $n_{procs} = 64$. We kept the size of the blocks consistent for both programs

N	MPI(avg)	openMP(avg)
32	0.169767	0.705871
64	0.178016	0.773215
128	0.181122	0.554437
256	0.504272	0.917894
512	3.223173	1.903027
1024	24.468546	14.122300
2048	–	106.938133
4096	–	930.740043

The table represents the upper limit of N we were able to reach before incurring in memory issues, for $blockSize = 100$. For both implementations we were able to reach higher values of N than what is reported, by decreasing the value of the blocks' size. It should be noted that in all trials we were able to run the

openMP version with larger values of N with respect to the MPI version with the same parameters.

4 Improvements

- The scalability of N depends strongly on the block size (it also depend on how many particles are given as input), and there is an upper limit given by $blockSize = 1$. Full blocking across all 3 axis would allow us to decrease the size of the density array even further, and therefore expand the range of acceptable N. As a downside, we expect to see a decrease in performance as for each block we need to iterate over the full array of particles; this would be very evident as N_p grows.
- Specific parallel I/O could be used to parallelize the writing operation (for example MPI-IO).