Università degli Studi di Trieste

dipartimento di matematica e geoscienze

Corso di Laurea Magistrale in Data Science and Scientific Computing

High Performance Computing: Assignment 3

Student:
Pietro Morichetti - SM3500414

Anno Accademico 2019-2020

Exercise 2: The N-Body problem

One of the possible purpose of the HPC is to make more efficient some physic or chemical informatic simulation and this report is focused on the propose of the "The N-Body problem". This issue requires to perform some program that manages a universe of physical particles inside a "unary" cube, where these objects are subject to the interaction among all of them. So, this kind of iteration will bring a sequence of evolution of the particles system in which all the particles will changes our features like the energy, the velocity and in particular the position that will be considered as the expression of evolution.

The target of the N-Body problem is to simulate as best as possible a really behaviour of a really particles system, and it could be very complex to realize in a informatic environment. For this reason exists several physical assumption to make it more manageable, that are showed below:

- All particles are incorporeal, and it means that they may be overlapped at a cert evolution's time or that doesn't exist any kind of impact among them;
- The next variation of time of the evolution has to be considered as the lower variation time among all the variation recordered by each particles; it means that all the particles will consider the same delta_t to perform the next step of the evolution;
- do not worry about the non-conservation of energy and momentum;

These and others semplification are reported in the text of the third assignment.

This specific N-Body problem may be take on in several way, in this report is taken on by using the OpenMPI library, the OpenMP library and a merged version of the previous two in a Hybrid method.

Design of the project

The design of the task provide for a division of the assignment in two different codes that they will be brifly explained in this section, more details about their operation will be showed in the specific section.

Generation phase of the particles system - The first one program is *generate_particles_system.cc*, it is implemented in C++ with the support of the MPI library and it is used to create the particles space, i.e the cube in which these objects are living. The program is executed by several nodes that here may will be called "MPI process" (tasks, or only process if it is clear from the context), and they will manage the particles system in synchronous logic.

Therefore, the program requires that the particles space is divided in a number of equivalent slices equals to the number of MPI process, and each slice is assigned to one and only one process. Inside these portions of the universe, the tasks have to generate a fixed number of particles fairly distributed among all of them. In particular, the distribution of the particles on the space, like others features, has been made by using some of the distribution functions of the statistic and probability fields: for example the normal distribution for the first velocity values.

At the end of the generation phase, all the processes write on the same binary file in parallel way, with the aid of the following expression [2]:

$$if \quad myid < R : 20 + myid \cdot my\_chunk \cdot (int)sizeof(particle) \qquad (1)$$

$$if \quad myid \geq R : 20 + (myid \cdot my\_chunk + R) \cdot (int)sizeof(particle) \qquad (2)$$

In addition of the first condition, this file is provided by a *meta data* that contain information like the number of particles, the number of files in which this file is splitted (oin this case zero, just one file) or the floating-point of the data: therefore, the total size of the meta_data is 20 bytes.

So, this file will preserve all the particles system, i.e. all the information associated with each one particle.

Evolution phase of the particles system - The second program is implemented in several way and it explores the functionality of the MPI and OMP libraries, for this reason, in the dedicated folder, it has different names in reference of the program version. In any case, all these C++ implementation have been done considering, more or less, the same logical path presented below.

All processes (MPI process or thread process, it does not matter) have to establish the particles number of the universe, and to download the particles system from the data_ic binary file. After this operation, the processes define the amount of particles that have to manage or, more accurately, that have to be evolved: the procedure is the same as the generation assignment in the generation program. In particular, the evolution phase provides to define the new values for the particles features by following these formulas [1].

The last formula is very important because it is used to syncronize all the particles at the same instant (less than a certain upperbound Δt_{max}), indeed it is obtained by the following constrain [3]:

$$\left|\frac{\Delta v}{v}\right| \leq \epsilon \rightarrow \left|\frac{a \cdot \Delta t}{v}\right| \leq \epsilon \Rightarrow \Delta t \leq \frac{\|v\|}{\|a\|} \cdot \epsilon = \Delta t_{max} \qquad (3)$$

By each particles you establish the potential next interval of time, but only the lower is selected i.e. the delta_t of the particles with the higher variation on the acceleration.

Execution Order	**Particle Feature**	**Formula**
1	Force	$F_q = m_g \cdot G \cdot \sum_{i=0, i \neq q}^{N_p} \frac{m_i \cdot (r_q - r_i)}{\left\| r_q - r_i \right\|^3}$
2	Acceleration	$a_q = \frac{F_q}{m_q}$
3	Velocity	$\Delta v_q\left(t\right) = v_q\left(t\right) + a_q\left(t\right) \cdot \Delta t$
4	Position	$\Delta r_q\left(t\right) = r_q\left(t\right) + \left(v_q\left(t\right) + \Delta v_q\left(t\right)\right) \cdot \Delta t$
5	Energy	$E_q = \frac{1}{2} \cdot m_q \cdot \left\| v_q \right\|^2 + G \cdot \sum_{i=0, i \neq q}^{N_p} \frac{m_i}{\left\| r_q - r_i \right\|}$
6	"Potential" next delta_t	$\Delta t \leq \frac{\left\| v \right\|}{\left\| a \right\|} \cdot \epsilon = \Delta t_{max}$

Table 1: Sequence of formulas used to evaluate the updated particles features and they are provided by the assignment.

When all the processes have update their own subset of particles, it is time to share the information among all of them, to update all the particles system and to decide the new delta_t for the next evolution.

It could be of interest to consider that the hybrid version is bit different compared to the others two versions, indeed the tasks that involves in the management of the problem are both MPI process and thread process. So, this special case has to be carefully supervised because will have to synchronize all the tasks and avoid any memory contention or dead lock.

Therefore, the MPI process, in the Hybrid version, deals with the downloading, uploading and sharing the particle system information. While, the thread process deals essentially in the computational part.

Note, the programs requires that after a fixed number of evolution all the processes saves the general status of the system into a new one binary file to prevent data lost.

Methods used within the programs

This section has all the information on the functions built that are distribuited among the different codes; about that, keep in mind that may happen that few functions belong to cert codes instead of others (it will clarify in the following parts).

struct, operators function At the outside of the mains function there are several elements as for example untyped constants used to simplify debugging procedure, or to (un)able some prints function to highlight specific information. Moreover, there are some struct to gather similar information and some overwrite function to customize particular standard operators to simplify daily operation upon the particles.

norm function - This function performs the norm two between two specific particles, i.e. the Eucledian distance in R^3 (in this particular case).

build_mpi_data_type function - This function builds two new personalized MPI_Data_type i.e. the MPI_particle_type for the single particle and the MPI_metanode_type for the meta data of the file. They are used to allow the transfer of particles among the MPI processes and to write directly on the file, without consider any complexity associated with the offsetting of the pointers.

build_group function - This function and the next one are very important because are used to create four groups of MPI processes, these groups are necessary to establish who to communicate to. The reason behind it is that if the processes have to manage the possible reminder, then they will have different size of the problem and it could be an issue in the share phase of the information. The groups are a trick to solve this hurdle.

In order to have a clear idea of this groups, it could be useful to consider an example showed in the figure 1 (the example will be used also later).

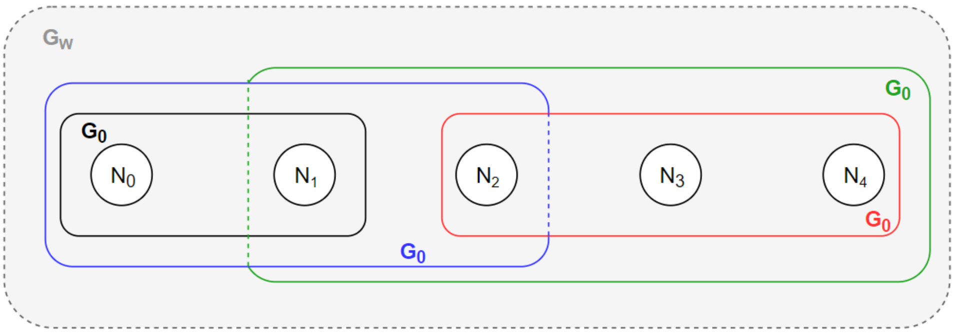

Figure 1: MPI processes divided in groups.

The picture shows five MPI process $N_0, \ldots, N_4$ and four different groups $G_0, \ldots, G_3$ in which they are devided. So, these groups consider:

- **G**$_0$ - all the processes that have to manage some reminder, i.e. who have the greater size of the sub problem;
- **G**$_1$ - all the processes that have to manage no one reminder, i.e. who have the smaller size of the sub problem;
- **G**$_2$ - all the members of the Group$_0$ plus the process with the smaller ID among the processes that have not manage any reminder;
- **G**$_3$ - all the members of the Group$_1$ plus the process with the greater ID among the processes that have to manage some reminder.

Groups	**Access Conditions**	**MPI Process Passed**	**Notes**
G_0	myid <R	N_0, N_1	who has to manage some reminder
G_1	myid >- R	N_2, N_3, N_4,	who doesn't have to manage some reminder
G_2	myid <- R && R >0	N_0, N_1, N_2,	$G_0 + N_2$
G_3	myid >- R - 1 && R >0	N_0, N_1, N_2, N_4	$G_1 + N_1$

Table 2: How the processes are divided among each groups.

Groups	**Group's size**	**Members**
G_0	R (= 2)	i (= 0, 1)
G_1	N_t - R (= 3)	i + R (= 2, 3, 4)
G_2	R + 1 (= 3)	i (= 0, 1, 2)
G_3	N_t - R + 1 (= 4)	i + R - 1 (= 1, 2, 3, 4)

Table 3: Size and members for each groups, in particular the "i" character is refered to the index of the specific *for* loop inside the *specific_groups* function, while the value defined inside the parenthesys are refered to the value for the example.

The right selection of the size and the elements of each groups are made by the *specific_group* function and, after it, the groups are finalized through a couple of MPI functions. In particular, each process will carry out it only belong to their own groups.

Note, actually exist another one group that is called Group_G and it is the global groups that contains all the MPI process; so the others groups are a sub groups of this one.

specific_group function - This function, as the previous one, is very important for the creation of the four processes groups. In particular, this function is used to set the sizes and the members of each groups, considering several constrains that are reported in the following table 2.

At this point, different processes have access different *if-body* condition and they are able to define sizes and members of their own groups; it is done as showed in the table 3.

But, the constrains to access in the groups, the sizes of each groups and which are the processes allowed to be a part of groups are explained in very different way into the codes. Indeed, this function is not splitted in few

	General Constrains	**General Group's size**	**General Members**
	myid <R && R_artificial >0	size_array_group	j + value_array_group
Groups	**Constrains**	**Group's size**	**Value_array_group**
G_0	myid <R && 1 >0	R	0
G_1	- myid - 1 <- R && 1 >0	N_t - R	R
G_2	myid - 1 <R && R >0	R + 1	0
G_3	- myid <2 - R && R >0	N_t - R + 1	R - 1

Table 4: Effective constrains, size and members for each groups, keep in mind that the variable "R_artificial" is used to be able to devide the $G_{0,1}$ and $G_{2,3}$; while the variable "j" is used instead of the variable "i" in the previous table. In the end, the variable "value_array_group" is used to specify the different members of each groups because the j variable starts always from zero.

if-condition but it is only one that is the result of the union of all these *if-condition* (this choice is made to make more compact the codes but paying in readability).So, it means that the codes for this function is designed in the following table 4.

Actually the real different between the previous definition and the actual definition is only a changes on the verse of inequality and, in consequence, the signs.

All the parameters presented into the table are part of the arguments of this function; so at the end of it, all processes have define each parameters to build the groups that it is performed by the *build_group* function.

force_computing - The first step of the evolution phase, in terms of computation, is to determine the forces among all the particles, or better, among only one specific particle at a time and all the others. Briefly, this function does a sequence of mathematical steps to evaluate a portion of the force's formula and energy's formula; the rest of the formula is performed in the *evolution_computation* function.

Actually, this function performs just a partial computation fo the total force bear on the particle, indeed this function is called two times and in sequence. This choice is made because the first calls of the function are considered all the particles with index less than the index of the particle under analysis, while the second calls considers all the particles after hers index, as showed in the figure 2.

Evidently, it may possible to do it in only one single *for-loop* including within it a *if-condition* to check if the particle under analysis is different by another one. However, the chosen proposed is more performance, inasmuch it carry out the same number of iteration but with the advantage of avoid any branch misses.

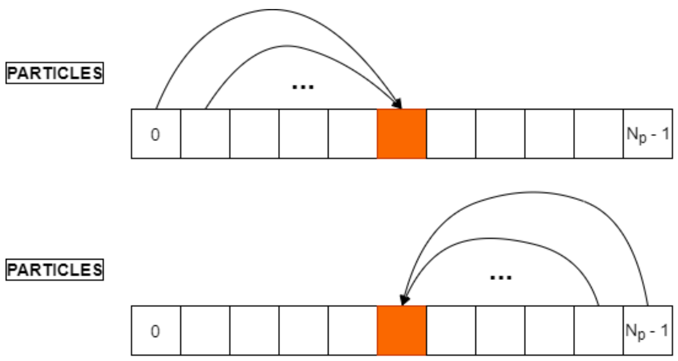

Figure 2: Forces's interaction among one specific particle and the all others; the figure abow rapresents the first calls of the function, while the figure below rapresentes the second calls of the function.

evolution_computation - This function performs the update of all the particle's features as the new velocity, the new position and the new energy.

deallocate_memories - This function is used to free all the memories that are reserved to the program, as for example dynamic array, file pointer or customized MPI data type. For this reason it keep in input several pointers of different types and it is called only at the end of the program.

check_workers - This function is used at the beggining of the evolution program and serves to check if the number of all processes that involved in the program is accetable, or in case if could be optimized or just wrong.

print_array - This function is used to print out the status of the system or a specific portion of it. However, it does not show all the information of the particles, but just the current place in the space, inasmuch this special feature is able to be perfectly the system.

Note, This method is called just before the computation for the next step of the evolution, and it requires a pointer to an array of particles and hers size.

generation main function - Each process determines the exact byte from which starts to write on the file, in particular this value is private for the process and it keep in mind to jump beyond the meta data memory region: the first 20 bytes of the file. Moreover, statistical distribution function are used to define the spatial coordinates and the initial velocity for all the particles, but with the constrains contained in the assignment. In the end, inside the portion of codes dedicated to build the particles, there is a specific condition to be sure that all the particles are not made in the same point; at least at the generation phase of the system (it is relaxed in evolution phase).

evolution MPI main function - Keep in mind that it is the main function of the MPI version, for the other versions, few observation are needed.

In this function reside all the system as the shape of an array of particle, called *array_all_particle*. Even if this array store all the system, each processes are responsible only for a small part of it, i.e. that they have to update only a fixed smaller number of particles. Indeed, the processes sets the number of particles that have to be managed and, in case, considering any reminder. Then, the following task is to create the groups by the specific function, and to define the offset by starts to read/write on the binary file that contain the particles system (or another file in case to checkpointing).

At this point, each process start the very evolution phase calling the specific computational function. For this reason, they hold track of the changes into another array of particles called *array_particle*, a kind of support vector to avoid changes directly into the array of all particles, as figure 3 shown.

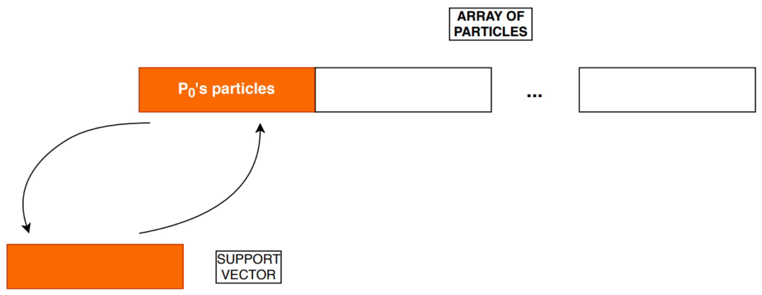

Figure 3: Interaction between the array of particles and the support vector used to take track of the P_0's particles update.

Then, the processes set up the potential next range of time to has to pass before the next evolution of the system. This variable time, called *tmp_delta_t_min*, is compared with all the same variable among the processes, and only the smaller will be considered as the effective range of time for the next evolution. In other words, this particular variable is used to synchronize all the processes on the same range of time, and to carry on the system at the next evolution.

Once the processes have decided it, they share their own particles among them through some MPI function as explained belove:

- MPI_Allgather - members of G_0 share among them, and members of G_1 also among them;
- MPI_Bcast - root process of the G_2 share only and exatly the subset of particles that it takes from the G_1, and the root process of the G_3

share only and exatly the subset of particles that it takes from the G_0.

At the end of these functions, all the processes have precisely the same array of particles and the system is just updated (figure 4).

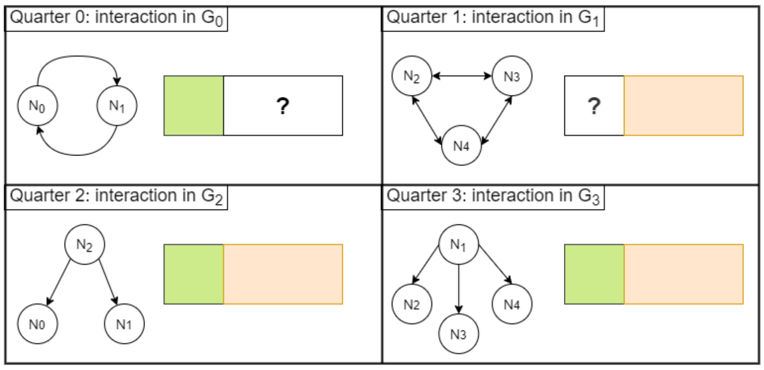
Quarter 0: interaction in G_0
N_0
N_1
?
Quarter 1: interaction in G_1
N_2
N3
N4
?
Quarter 2: interaction in G_2
N_2
N0
N1
Quarter 3: interaction in G_3
N_1
N2
N3
N4

Figure 4: In refered to the previous example: how is defined the interaction in the groups, in particular the quarters 0 and 1 is used the MPI_Allgather function, while in the quarters 2 and 3 is used the MPI_Bcast function; at the end, the particles system is shared everywhere.

Moreover, the main function is designed to be safe against unexpected disruption of the calculator: at the end of the system updating exist an *if-condition* that is needed to store the current status of the particle system in another binary file, called *check_point*. However this safe procedure is allowed only each three evolution of the system (this value is dediced a priori), because this kind of operation are very expensive for the calculators. So, it means that procedure is only a semi-safe operation, inasmuch if the unexpected disruption happen before the checkpointing procedure, the last information of the system are lost and it requires to re-start from the last checkpoint.

Finally, if the system reached the last evolution phase, the MPI processes free all the memories alloccated and they abort in the right way.

As introduced at the beginning of this description, the main function to OMP and Hybrid are slightly different, in particular for the OMP the universe particles is shared among threads, it means that the communication phase is not needed. While in the Hybrid version, both MPI processes and threads exist and they have to manage communication phase and any reminders that may appear.

Conclusions

In order to have a global view on the functionality of the functions and which programs use them and which no, as showed in table 5.

Function	**Input**	**Output**	**Code**	**Note**
norm	vector of coordinate	double	A	Define norm of a vector
check_workers	processes and problem's size	int	E	Check if it possible to exec simulation
build_mpi_data_type	New MPI data Type	-	A	Create a specialized MPI data type
specific_group	MPI process ID, couple of reminders and group's misure	int	E	Define size and members of each groups
build_group	number of processes, reminder, groups and communicators	int	E	Create the 4 groups among the MPI processes
deallocate_memories	All memories allocated	-	E	Free all the memories allocated during the execution of code
force_computing	range of particles array's of particles and the force of particle i	-	E	Partial calculation of the force and energy for a specific particle
evolution_computation	particle features, times and constants of the system, array's of particles, index of specific particle	-	E	Fill the support vector with the update of the system
print_array	array of particles and its size	-	A	Print of the particle's position

Table 5: This table summarize all function used in the programs, in particular each of them could appear in all programs (A) or only in the evolution programs (E). Moreover, the input field might not contain exactly all the very inputs argument of the code, to focus only on the significant parameters.

Moreover, might be useful the following picture that shows the logical path of the evolution algorithm's exectuion (figure 5).

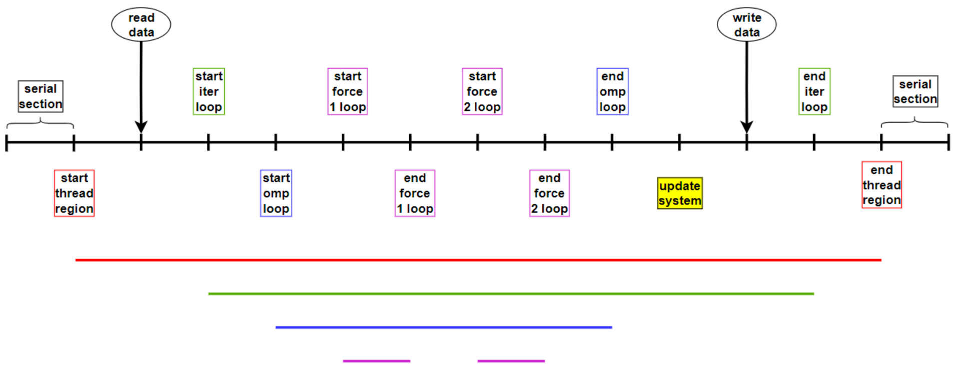

Figure 5: The logical execution of the evolution program by the point of view of the loops; in particular it is referered to the OMP and Hybrid version, while for the MPI version is possibile to ignore the thread region and changinf the *omp-loop*. Moreover, the picture shows some critical point of the code as the read and write operation and the update of the particles system.

The figure shows the direct operation of the nested sequence loops for the OMP case of the algorithm; but in anyway, it is also valid for the MPI and Hybrid versions, ignoring the references to the thread region or *omp-loop* and by using the MPI functions to read/write within the file and to share the particles.

Eventually, by this result you proof that it is possible to solve the N-Body problem by using the MPI and OMP libraries; as the picture 6 showed below.

Additional work could be done as:

- Optimize codes by using directly the pointers of the arrays, structs or function;
- Create a specialized library to gather all the built functions or customized operators;
- Try to delete the assumptions relaxed in order to look alike at the reality.

Moreover, might be of interest to make some kind of analisys as for instance the strong scaling test and the weak scaling test, but they require the serial version of the problem.

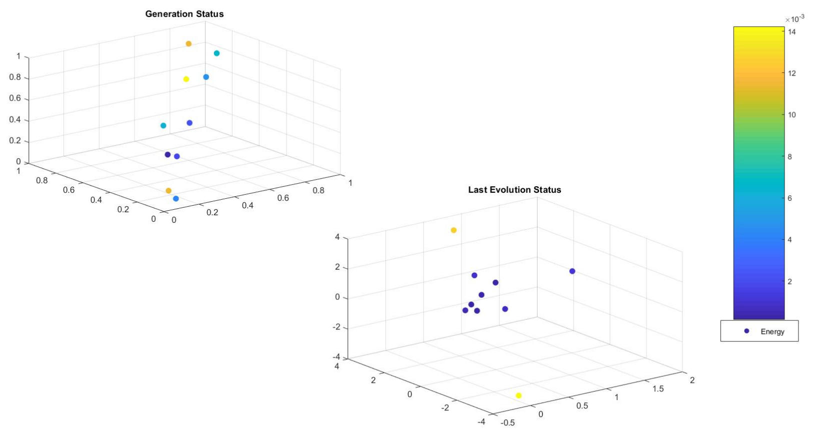

Figure 6: The Universe of particles includes just ten particles in the initial condition and in the last evolution of the system; note as the position and their own energy is changed.