UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Superior d'Enginyeries Industrial,
Aeroespacial i Audiovisual de Terrassa

MPI Project
Armaggedon

HIGH PERFORMANCE COMPUTING FOR AEROSPACE ENGINEERING 16/17

05-06-2017

Names:

Escartín Vivancos, Guillermo

Kaloyanov Naydenov, Boyan

Serra Moncunill, Josep Maria

Introduction:

The aim of the project is to compute (with parallel computation) if a satellite orbiting the Earth has a probability to crash with some space debris, using a security distance. The satellite chosen in our simulations is the **International Space Station**.

Parallel programming idea:

Parallel discretization can be done in time or by asteroids per processor[1]. It will be done by asteroids since the crashes can be detected in a more efficient way. If it was done in a timely manner, many unnecessary computations would be done, since the crash could be the first month of simulation and many processors would already be calculating several months after that. If the number of objects is not evenly divisible by the number of processors, one of the processors will have less objects on its database and, in principle, it will have less computational work.

A crash will be found when the asteroid distance to the satellite is less than a distance of security, specified by the user. This is done so as to account for the inaccuracies of the propagator.

The program ends when one of the processors detects the first crash and communicates it to the other processors. Therefore, a communication time has to be established. This parameter slows down the program quite a lot so it has to be carefully chosen depending on the aimed simulation length. On each communication stage, the program will check if a collision has been produced in any of the processes. If several collisions have been produced, the program gets just the first one. A small drawback of this approach is that each process has to propagate the satellite orbit.

[1] During the report, the “processor” term is used to refer to an instance of the code run by certain parts of the computer (processors, cores, threads, etc.). Open-MPI and the OS manager will adapt to the computer architecture.

Development of the project:

In the development of the project there were some critical issues that had to be taken into account. Firstly, it was hard to find an available and updated database of the space debris. The first solution we came up with was to use the Asteroids Belt database and scale them down simulating the space debris. This solution was discarded due to it being unrealistic. Finally, an available and suitable database was found [1]. The database was adapted to a special csv format so that it could be used by the propagator, thus the function provided by David de la Torre to read the database was slightly modified. Moreover, some Keplerian elements were missing. This missing elements were randomised using /databases/parserCSV.js. This code used to parse and adapt the database, has some external dependencies which were installed with npm. That dependencies are stored inside /node_modules and are:

1. csv_parsing library
2. **lodash** library for easier data treatment.

Another critical point was the prediction of the positions of the space debris and the satellite. At this point, two solutions were considered: to use of the basic ideal propagator provided by David de la Torre or to include the SGP4 propagator, which is more accurate on its calculations as it takes into account the effect of perturbations caused by the Earth's shape, drag, radiation, and the gravitation effects from the sun and the moon.

Three codes have been developed in order to achieve the goal of the project, each one corresponds to different steps of the development process. The first code (**main-seq**) is a sequential code which includes the spg4. The second code, (**Armaggedon**) is a parallel code which propagates the satellite and the space debris using the propagation equations provided by David de la Torre. The final code was a parallel code which uses the spg4 propagator.

Codes:

-Sequential code (main_sequential.c):

This program solves the problem sequentially. A remarkable feature is that it includes the SGP4 propagator on its simulations. The satellite (in TLE format), the security diameter, the simulation time and the time steps are inputs of this programme provided by arguments in the command line. If one of this parameters is not defined when the code is executed, a warning is printed in the screen.

The program reads all the database using **/lib/ReadDB.h** and starts the temporal loop. The time increases according to the specified time step. The function *init_spgp4*, which passes all the orbital elements to the *sgdp4()* function, is applied to the satellite, so its trajectory is propagated. Then, the position is converted to x,y and z coordinates.

The space debris loop starts. For each object, the program checks if the object is within the limits of the SGP4 (mean motion or eccentricity limitations). The trajectory of the objects is also propagated and converted to x,y and z coordinates. The function *is_crash* computes the distance between the satellite and the object and if it is smaller than the security distance, the function returns 1. If this happens, the program prints a warning, including the object of collision and the crashed time (and also the elapsed time for the simulation). If no object crashes with the satellite for that time, the outer loop increases jumps to the next time and repeats the process. If no crash has been detected for the whole simulation, the programme notifies it.

-Armaggedon (Armaggedon.c):

Leaving out of the difficulties of including the spg4 propagator in the overall code, it was written just to understand and define the main structure of the parallel solution.

As a first step, it divides the space debris of the database by the number of processors chosen by the user. Each processor reads and saves the data of its respective space debris and the satellite (all the processors propagate the satellite trajectory) using **/lib/ReadDB.h**.

Then some variables are defined. The initial simulation time (established as a given date) and the final time (established as the initial time plus an increment of time). Also, it is important to define the time step of the propagation and the communication time of the processors. The last one will be very critical because the processors take a relatively long time to communicate, so the number of communications during a simulation will have an important impact on the final duration. Finally, the security distance is specified. Here, all this parameters are specified inside the code.

The programme starts the temporal loop at **t=TimeInitial** and increases the time with the time steps previously defined. The trajectory of the satellite is propagated by each processor using the /lib/Astrodynamics.h. Then, the processors propagate the space debris and computes the distance with the satellite (using their position with respect to the center of the Earth). If the distance is smaller than the security distance, the processor registers a collision and also saves the name of the object in the variable ***Collider*** and the time of the collision in ***CollisionTime***.

When the simulation time is a multiple of the communication time, **MPI_Allreduce** is applied in order to check the collision status of each processor. If several processors have registered a collision, a second MPI_Allreduce will find the first in "absolut"[2] time scale. The programme will stop at this point and will print on the screen the collision time and the object which produced the collision. If no collision has been detected, the programme continues with the temporal loop. Finally, if the simulation finishes with no collision, the programme notifies to the user that the satellite is safe.

-Final parallel code (main_parallel.c):

The last code is a kind of a mix of the previous codes. It starts with the MPI initialization and the definition of the same time variables than "Armaggedon": communication time, which is defined in the code, and time steps and simulation time, which are inputs of the programme. Also the satellite (in TLE format) and the security diameter must be defined

[2] Absolut refers to the simulation time, and not to the elapsed time of the running code.

when the program is executed. Again, the space debris is divided by the number of processors. All the parameters are checked and a notification is sent if something is missing or if it is in a wrong format.

The structure of the time loop was explained in the sequential code, but in this case, it will be executed by each processor with its objects. The main difference is how the collision time is registered, using the same method than Armaggedon: the function *is_crashed* computes the distance between the satellite and the object and return one if it is less than the security distance. A first **MPI_Allreduce** checks if a collision has been produced (if there is a 1 in the collision variable of any processor) and a second MPI_Allreduce saves the shortest time. If no crash is produced, the programme notifies it. Again the communication time is a critical parameter in the simulation, because the processors spend a lot of time to compare their information.

-Auxiliar code:

For the development of the current project, additional programs have been developed for the following tasks:

- Makefile. By typing “make” all the three codes get compiled only if there has been a change in any of them. The libraries are kept in separate folders.
- Database parsing and adaptation. **/databases/parserCSV.js**
- 3D plotting. **/plots/3D_plot.py** gets two csv files as argument. Each csv has the orbital positions of an object and the time. The output is a 3D plot using **matplotlib** library.
- Animation generation. **/plots/animation.py** gets two csv files as well. It generates an animation which shows the crash and saves an mp4 file, also using **matplotlib.**
- Efficiency study. **/plotting.py** is used to run the parallel code several times with different arguments and to finally obtain a plot which shows the Elapsed time according to the number of processors used.
- git system for teamwork, version control and future improvements.

Some results:

The following plots are all obtained for the same simulation for the sake of comparison purposes. Armaggedon code has been used for those since the code that uses the **SPG4** algorithm makes this a tedious task as the library prints many things to the standard output, which is used to get the Elapsed time, for instance. This happens as some of the objects are not valid for the SPG4 library, or others crash into the Earth do to the orbit decay and the program clutters the terminal output with different warnings. Rather than modifying the SPG4 library, it was decided to use the Armaggedon code for the efficiency study.

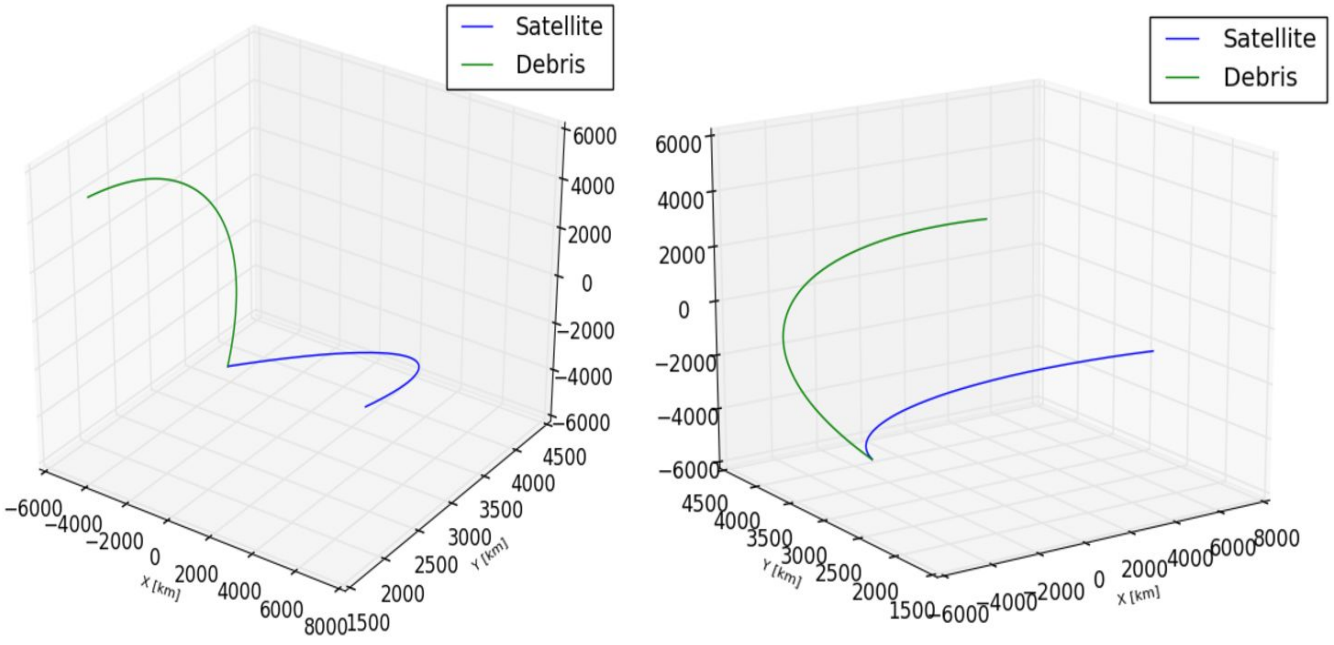

Figure 1: Collision object: FENGYUN_1C_DEB. Collision time: 1500 s after 18:00 of the 2nd of January, 2017. Security distance: 10 km. Simulation time steps: 10 s.

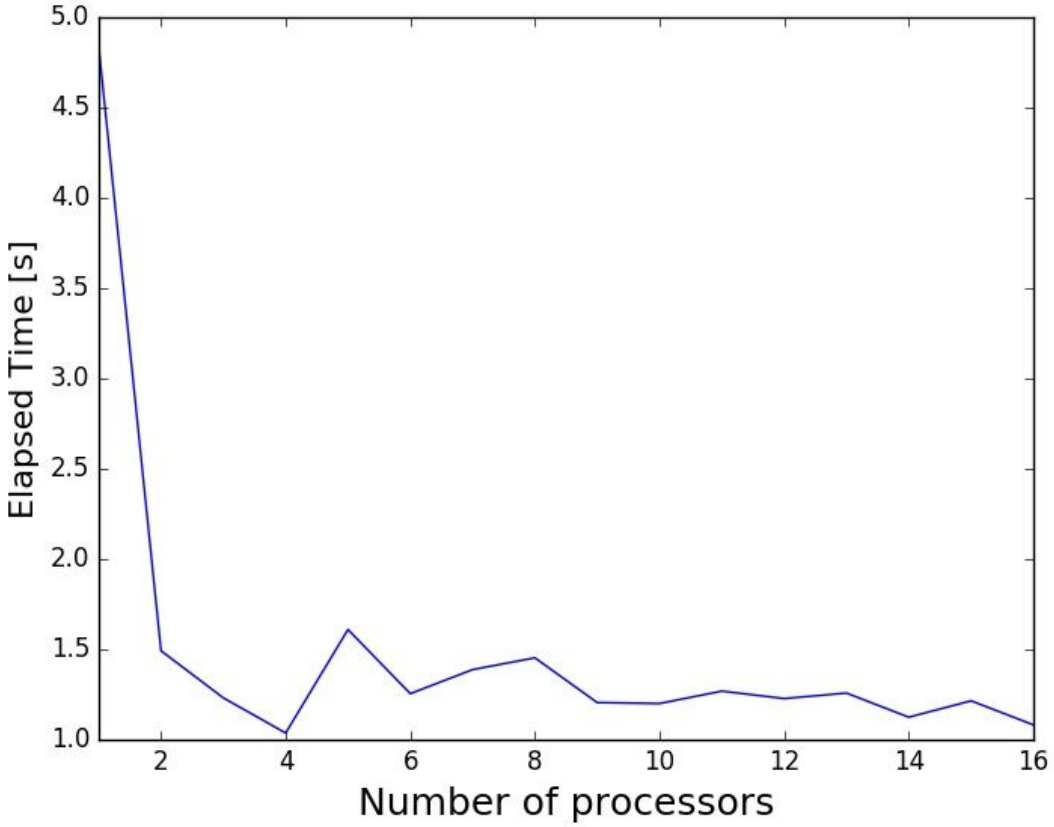

Figure 2: Efficiency study obtained with an Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz.

The used processor has two cores, and thanks to the hyper-threading technology of Intel, each core can execute up to 2 threads simultaneously, which makes a total of 4 processes to be run simultaneously. This explains why the optimal number of processors is 4.

It is also important to note that the elapsed time is decreased by a factor of 5 when comparing the sequential code to the 4 processors parallel code. This is for a half an hour orbit simulation and hence, the gain is in the scale of seconds and therefore, not quite significative. However, if longer orbit simulations were to be performed the difference could be in the hours or days time scale, where a factor of 5 is very great improvement.

A video animation is attached to the delivered file with this document, where the previous crash can be observed.

Acknowledgments:

All the contents inside the **/lib** folder are supplied by David de la Torre, teacher at UPC. All the contents inside the **/spg_lib** are downloaded from [2] and were supplied by Paul Crawford & Andrew Brooks.

Thanks to all of them for helping the open source community!

Bibliography

[1] Database: https://www.space-track.org/ [online, 08-05-17]

[2] SPG4 algorithm: http://www.sat.dundee.ac.uk/psc/sgp4.html [online, 15-05-17]

[3] Open-MPI: https://www.open-mpi.org/ [online, 04-06-17]