

INFO 0939 Project 2026

Last updated: 30/08/2026

Goals of the Project

- Solve a physical problem modelled by a large system of ordinary differential equations, using a direct summation method coded in C.
- Parallelize the code on distributed memory systems using MPI, and on shared memory systems using OpenMP.
- Test the acceleration potential of GPUs using CUDA.
- Learn to profile the code, run weak and strong scalability analyses, and understand the performance of the algorithm on each hardware.
- Explore one or more advanced topic(s) based on your own interests.

Project statement

We consider a system of N point masses interacting through gravitation. Body i has mass m_i , position $\mathbf{r}_i(t)$ and velocity $\mathbf{v}_i(t) = \dot{\mathbf{r}}_i(t)$, and Newton's law of gravitation gives its acceleration as the sum of the contributions of all the other bodies:

$$\mathbf{a}_i = \ddot{\mathbf{r}}_i = G \sum_{j \neq i} m_j \frac{\mathbf{r}_j - \mathbf{r}_i}{\|\mathbf{r}_j - \mathbf{r}_i\|^3}, \quad i = 0, \dots, N-1, \quad (1)$$

where G is the gravitational constant. This is a system of $3N$ second-order ordinary differential equations, and no closed-form solution exists as soon as $N > 2$.

Formula (1) is singular when two bodies come arbitrarily close to each other. Since the N bodies of the simulation are not real point masses but rather represent a smooth mass distribution sampled by N particles, the singularity is removed by introducing a *softening length* ε :

$$\mathbf{a}_i = G \sum_j m_j \frac{\mathbf{r}_j - \mathbf{r}_i}{(\|\mathbf{r}_j - \mathbf{r}_i\|^2 + \varepsilon^2)^{3/2}}. \quad (2)$$

Note that the term $j = i$ can now be kept in the sum, since it contributes exactly zero. The softened interaction (2) derives from the potential energy

$$U = -G \sum_i \sum_{j > i} \frac{m_i m_j}{\sqrt{\|\mathbf{r}_j - \mathbf{r}_i\|^2 + \varepsilon^2}}, \quad (3)$$

so that the total energy $E = T + U$, with the kinetic energy $T = \frac{1}{2} \sum_i m_i \|\mathbf{v}_i\|^2$, is exactly conserved by the continuous problem, together with the total linear momentum $\mathbf{p} = \sum_i m_i \mathbf{v}_i$ and the total angular momentum $\mathbf{L} = \sum_i m_i \mathbf{r}_i \times \mathbf{v}_i$. These three invariants are the main tool at your disposal to verify that the code is correct, in serial as well as in parallel.

The equations of motion are integrated in time with the *leapfrog* scheme, in its kick-drift-kick form – the same scheme is known as velocity Verlet in the molecular dynamics literature, and as the Störmer-Verlet scheme in the numerical analysis one. Beware that the name “leapfrog” alone often refers to the staggered variant, in which the velocities permanently live at half-integer time steps, as the magnetic field did in last year's finite difference project; the form used here is the

synchronized rewriting of it, which is what one wants when positions and velocities are needed at the same instant, for instance to evaluate the kinetic energy. Denoting by $\mathbf{r}_i^n$ and $\mathbf{v}_i^n$ the approximation of the position and of the velocity of body i at time $n\Delta t$, one step of the scheme reads (see Figure 1)

$$\mathbf{v}_i^{n+1/2} = \mathbf{v}_i^n + \frac{\Delta t}{2} \mathbf{a}_i(\mathbf{r}^n) \quad (\text{kick}) \quad (4)$$

$$\mathbf{r}_i^{n+1} = \mathbf{r}_i^n + \Delta t \mathbf{v}_i^{n+1/2} \quad (\text{drift}) \quad (5)$$

$$\mathbf{v}_i^{n+1} = \mathbf{v}_i^{n+1/2} + \frac{\Delta t}{2} \mathbf{a}_i(\mathbf{r}^{n+1}) \quad (\text{kick}) \quad (6)$$

for $n = 0, 1, \dots$, where the accelerations $\mathbf{a}_i$ are evaluated with (2). The scheme is explicit and second-order accurate, and requires a single evaluation of the accelerations per time step, since the accelerations computed at the end of a step are reused at the beginning of the next one. It is also time-reversible and symplectic, which has an important practical consequence that you are asked to verify experimentally: the error on the total energy does not accumulate with time, but oscillates and remains bounded, in contrast with what a standard Runge-Kutta scheme of the same order would give.

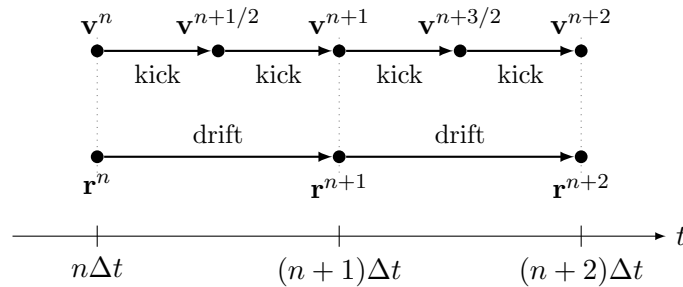


Figure 1: One step of the kick-drift-kick leapfrog scheme. Positions are known at integer time steps, and velocities are advanced by two half kicks separated by a drift. The accelerations used by the two kicks of a step are those evaluated at $\mathbf{r}^n$ and at $\mathbf{r}^{n+1}$ respectively, so that a single evaluation per step is needed: the last kick of a step and the first kick of the next one both use $\mathbf{a}(\mathbf{r}^{n+1})$, and can be merged into a single full kick.

The evaluation of the accelerations (2) makes every body interact with every other one, and completely dominates the cost of a time step.

All the computations are performed in a system of units in which $G = 1$, and the initial conditions are scaled so that the total mass and the initial size of the system are of order one; the dynamical time of the system is then also of order one. As an illustration, choosing 10 kpc as the unit of length and 10^{11} solar masses as the unit of mass gives a unit of velocity of about 207 km/s and a unit of time of about 47 Myr, which are the correct orders of magnitude for a galaxy.

Three problems are provided, summarized in Table 1. The first one is the encounter of two galaxies on a parabolic orbit with a nonzero impact parameter, each galaxy being modelled by a massive central body surrounded by a rotating disk of much lighter bodies. This is essentially the setup with which Toomre and Toomre explained the bridges and the tails observed in interacting galaxies [3]; see Figure 2. It runs in about a minute and a half in serial, and is the one to use to produce pictures and movies. The second problem is a single Plummer sphere in equilibrium, large enough to be meaningful for scaling measurements; it runs in about nine minutes in serial. The third one is the collapse of a cold uniform sphere released at rest: the density in the center increases by orders of magnitude before the sphere bounces back, which makes it the most demanding of the three for the time integration.

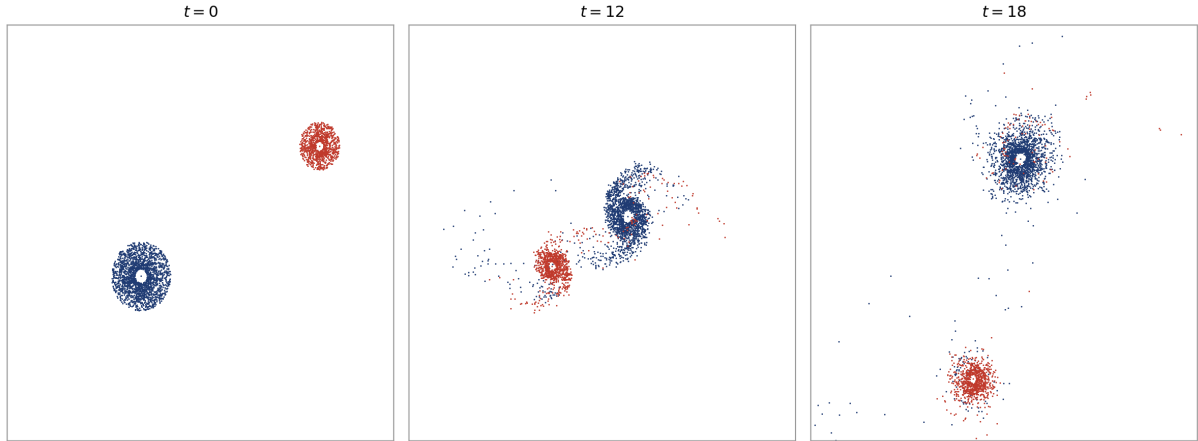


Figure 2: Problem 1: encounter of two galaxies, seen in the orbital plane. Bodies are colored according to the galaxy they initially belong to. The tidal interaction extracts bridges and tails out of the two disks.

problem_id	initial condition	N	ε	Δt	N_t
1	encounter of two disk galaxies	3 000	0.05	0.01	3 000
2	Plummer sphere in equilibrium	50 000	0.02	0.002	50
3	cold collapse of a uniform sphere	4 000	0.02	0.002	2 000

Table 1: The three problems implemented in the reference code.

Instructions

A serial C code that implements the direct summation of (2) and the leapfrog integration is available on <https://people.montefiore.uliege.be/geuzaine/INF00939/project/>. The program takes a single integer argument, selecting the problem to solve, and creates output files that can be read with Paraview (<https://paraview.org>). The particles are written as a point cloud, with the modulus of the velocity and the index of the galaxy they belong to attached to each point; use the *Point Gaussian* representation of Paraview to render them.

In a first project phase (deadline: 24/11/2026) you are asked to:

1. Experimentally study the accuracy and the stability of the leapfrog scheme, depending on the values of the parameters. How do the errors on the three invariants behave as a function of Δt , of the softening length ε , and of the duration of the simulation? Compare with the explicit Euler scheme, which you will implement for the purpose of the comparison. Verify that the scheme is time-reversible, by integrating backwards in time. Which of the three problems is the most constraining, and why?
2. Perform an analysis of the arithmetic intensity of the numerical method and determine the main limiting factor for the speed of the algorithm: is the code memory bound or CPU bound? What do you expect if you run the code on a machine capable of 200 GB/s of memory bandwidth and 2.8 TFLOPS/s of processing power? Compare your prediction with the measured performance, and explain the difference: which hardware resource actually limits the evaluation of (2)?
3. Evaluate potential bottlenecks in the serial code, based on the lectures on CPU architecture

and cache hierarchies. Discuss in particular the layout of the data in memory, the cost of the square root and of the division, the effect of the compiler options controlling the accuracy of the floating point arithmetic, and whether the innermost loop is vectorized.

4. Parallelize the serial code using MPI, by distributing the bodies over the processes. Since every body interacts with every other one, each process needs to see all the positions and masses at every time step: implement and compare at least two strategies to achieve this, for instance a collective communication of all the positions to all the processes, and a ring exchange in which each process sends the block it currently holds to its neighbour and receives a new one, $P - 1$ times per time step. How much data is communicated in each case? Can the communications be overlapped with computations?
5. Parallelize the serial code using OpenMP. Newton's third law implies that the interaction between i and j could be computed once instead of twice, halving the number of operations: what does this imply in terms of concurrent writes, and is it a good idea? Pay attention to the load balancing and to false sharing.
6. Perform a scalability analysis of the MPI and the OpenMP codes, by evaluating both the strong and the weak scaling. Choose appropriate parameter values for these scaling runs, deactivating outputs to disk. Make sure to reserve appropriate resources on the clusters (e.g. full nodes), and perform measurements multiple times to reduce uncertainty. Explain the results of the scalability study using the concepts and the tools presented during the lectures. Do you observe NUMA effects? How do your measurements compare with the performance model you can write down for each of the two communication strategies of item 4?
7. Port the code to run in single precision on a single GPU of the Lyra cluster, using CUDA. Pay attention to the reuse of the positions loaded from the global memory [4]. Compare and analyze the performance of the CPU and GPU codes.

Every run of your code, in serial and in parallel, must report the relative error on the total energy and the values of the total linear and angular momentum. Include these values in your scaling tables: a code that is fast but does not conserve what it should is not a correct code.

In a second phase (deadline: 18/12/2026), you are asked to explore replacing the direct summation by the hierarchical algorithm of Barnes and Hut [2], in which the bodies are stored in an octree and the contribution of a distant group of bodies is approximated by that of its center of mass, reducing the complexity from $O(N^2)$ to $O(N \log N)$. Study the accuracy of the approximation as a function of the opening angle, and determine for which value of N the tree code becomes faster than the direct summation. Redo the analysis of item 2 of the first phase for this algorithm: what happens to the arithmetic intensity, and what does this imply for the parallelization?

General remarks on C coding style

Your coding style will be evaluated. As such, some quality in the submitted code is expected, and you should strive to have a clean and neat coding style. In general, think about who will read your code and ask yourself if it is clear and understandable. Please beware of some common mistakes that will lead to penalties on your final grade:

- If you `malloc()` something, remember to `free()` it.

- Check return values for failure, especially from memory allocation, file handling and MPI functions. Don't assume that those calls will always succeed.
- Don't use Variable Length Arrays, i.e.

```
int function(int N) {
    int array[N];
}
```

The value of N must be known at compile time. If it is not the case, use `malloc()`.

- Don't abuse comments. If something is obvious, don't comment it. If you feel the need to comment some code, ask yourself if perhaps you can write the code in a clearer way instead of putting that comment.
- Try to use relevant and appropriate variable and function names. Be coherent in naming things, in order to make it easy to track down where data goes.
- Whenever possible avoid global variables: they are one of the factors that make it harder to make your code thread safe.
- Limit the length of your functions: if a function does not fit on your screen, perhaps you should split it into smaller pieces. Also, if you have a function that takes more than 10 parameters, perhaps it is time create a `struct`?

A The Barnes-Hut algorithm

This appendix sketches the algorithm proposed in [2], which is the subject of the first topic of the second phase. The idea is that the detailed distribution of a group of distant bodies does not matter: seen from far enough away, the group acts as a single body placed at its center of mass. The bodies are therefore stored in a tree that recursively subdivides space – a quadtree in two dimensions, an octree in three – and the sum over the $N - 1$ other bodies is replaced by a sum over a much smaller number of tree nodes, whose size grows with their distance.

Each node of the tree stores the geometry of the cube it covers, the total mass M and the center of mass $\mathbf{R}$ of the bodies it contains, and either a single body (leaf) or eight children (internal node). Building the tree and computing the moments are two traversals of cost $O(N \log N)$ and $O(N)$ respectively.

The force evaluation traverses the tree from the root for each body. A node is accepted as a whole when it is small enough compared with its distance, i.e. when $s/d \leq \theta$, where s is the edge length of the node, d the distance from the body to the center of mass of the node, and θ the *opening angle*; otherwise the node is opened and its children are examined in turn. The parameter θ controls the trade-off between cost and accuracy: $\theta = 0$ recovers the direct summation, while a large θ gives a fast and inaccurate algorithm. Because the expansion is made about the center of mass, the dipole term vanishes and the leading error is the quadrupole one, of relative size $O(\theta^2)$.

The tree must be rebuilt (or at least repaired) at every time step, since the bodies move. Each traversal visits $O(\log N)$ nodes per body for a fixed θ , hence the $O(N \log N)$ total cost. Three practical remarks are worth keeping in mind when implementing it: the tree should be stored in flat arrays indexed by integers rather than as a graph of `malloc`'ed nodes, the bodies should be sorted along a space filling curve (Morton or Hilbert order) so that neighbouring bodies are contiguous in memory and traverse nearly the same nodes, and the traversal should build an

Algorithm 1 Insertion of a body in the tree, and computation of the moments

```
1: procedure INSERT(node,  $b$ )
2:   if node is empty then
3:     store  $b$  in node; node becomes a leaf
4:   else if node is a leaf then
5:      $b' \leftarrow$  the body stored in node
6:     subdivide node into 8 children; node becomes internal
7:     INSERT(child of node containing  $b'$ ,  $b'$ )
8:     INSERT(child of node containing  $b$ ,  $b$ )
9:   else
10:    INSERT(child of node containing  $b$ ,  $b$ )

11: procedure MOMENTS(node)
12:   if node is a leaf then
13:      $M_{\text{node}} \leftarrow m_b$ ;  $\mathbf{R}_{\text{node}} \leftarrow \mathbf{r}_b$ 
14:   else
15:     for all children  $c$  of node do
16:       MOMENTS( $c$ )
17:      $M_{\text{node}} \leftarrow \sum_c M_c$ ;  $\mathbf{R}_{\text{node}} \leftarrow (\sum_c M_c \mathbf{R}_c) / M_{\text{node}}$ 
```

Algorithm 2 Acceleration of one body by tree traversal

```
1: procedure ACCELERATION(node,  $b$ ,  $\theta$ ,  $\varepsilon$ )
2:   if node is empty or node is the leaf holding  $b$  then
3:     return 0
4:    $\mathbf{d} \leftarrow \mathbf{R}_{\text{node}} - \mathbf{r}_b$ ;  $d \leftarrow \|\mathbf{d}\|$ 
5:   if node is a leaf or  $s_{\text{node}}/d \leq \theta$  then
6:     return  $GM_{\text{node}} \mathbf{d} / (d^2 + \varepsilon^2)^{3/2}$  ▷ one interaction instead of many
7:   else
8:     return  $\sum_{c \in \text{children}} \text{ACCELERATION}(c, b, \theta, \varepsilon)$ 

9: for all bodies  $b$  do
10:   $\mathbf{a}_b \leftarrow \text{ACCELERATION}(\text{root}, b, \theta, \varepsilon)$ 
```

explicit interaction list per body (or per group of nearby bodies) so that the resulting sum can be evaluated with the same vectorized kernel as the direct method. Without these precautions the implementation is dominated by pointer chasing and unpredictable branches, and it may well be slower than the direct summation for the values of N considered here.

B From Barnes-Hut to the fast multipole method

The algorithm above only approximates the *source* side of the interaction: a distant group of bodies is replaced by a single equivalent body, but the result is evaluated separately, and exactly, at each target body. This is a particle-cell interaction, and the traversal is repeated N times, once per target, which is where the $\log N$ comes from.

The fast multipole method (FMM) removes that last factor by treating the two sides symmetrically. Bodies that are close to each other see the same distant groups in nearly the same way, so their contribution can be computed once for the whole target cell, in the form of a local (Taylor) expansion of the potential about the center of that cell, and only then evaluated at the

individual bodies. The interaction thus becomes cell-cell, and the algorithm is organized around a small number of operators:

P2M, M2M	build the multipole expansion of a leaf, then merge the children's expansions upwards
M2L	convert the multipole expansion of a source cell into a local expansion of a target cell
L2L, L2P	propagate the local expansions downwards, then evaluate them at the bodies
P2P	direct summation for the near-field pairs, which are never expanded

Each cell interacts with a bounded number of source cells (its interaction list, of size 189 for the standard octree in three dimensions), and the number of cells is $O(N)$, so the total cost is $O(N)$ instead of $O(N \log N)$. The accuracy is controlled by the order p of the expansions rather than by an angle, and the error decreases like θ^{p+1} , i.e. exponentially in p at fixed geometry, which is what makes the method usable when several correct digits are required.

Seen from a distance, Barnes-Hut and the FMM are the same idea: the dense $N \times N$ interaction matrix is split into near-field blocks, which are kept dense and evaluated directly, and far-field blocks, which are numerically of low rank and are replaced by a factorized approximation. Barnes-Hut uses a rank-one approximation of the far-field blocks and a single-tree traversal; the FMM uses a rank- $O(p^2)$ approximation and a dual-tree traversal. The same construction applied to the dense matrices of a boundary element method gives the hierarchical matrices, with adaptive cross approximation playing the role of the multipole expansion.

The comparison is also interesting from the performance point of view, and is a good subject for the report. The FMM has a much larger constant, so Barnes-Hut remains competitive at low accuracy and moderate N ; but the FMM concentrates its work in the P2P and M2L kernels, which are dense, regular and blocked operations of high arithmetic intensity, whereas Barnes-Hut spends its time in an irregular traversal. In other words, the $O(N)$ method is not only asymptotically cheaper: it is also the one that maps better onto vector units and GPUs, which is why the fast solvers used in production today are of that family.

References

- [1] Sverre J. Aarseth. *Gravitational N-Body Simulations: Tools and Algorithms*. Cambridge University Press, 2003.
- [2] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.
- [3] Alar Toomre and Juri Toomre. Galactic bridges and tails. *The Astrophysical Journal*, 178:623–666, 1972.
- [4] Lars Nyland, Mark Harris and Jan Prins. Fast N-body simulation with CUDA. In *GPU Gems 3*, chapter 31, pages 677–695. Addison-Wesley, 2007. <https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-31-fast-n-body-simulation-cuda>.
- [5] Volker Springel. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364(4):1105–1134, 2005.