Technische Universität München

Parallel Programming (TUM) SS2021 Final Project

Examiner: Prof. Martin Schulz
Project Authors: Shubham Khatri, Maximilian Stadler
Teaching Assistants: Bengisu Elis, Vincent Bode

Project 04 – BH

General Guidelines and Rules

- Your work needs to be submitted through the LRZ GitLab. You will be granted access to your repositories at the kickoff.
- You will submit your solutions in three steps with three different due dates. The individual tasks and their due dates may be found below.
- There are separate repositories under your GitLab group for each task in this document, namely OMP, MPI, Hybrid, Bonus and Documentation. Please make sure to submit your solutions to the corresponding repositories.
- Keep in mind that only the master branches of each repository will be considered as a valid submission and graded. You are expected to submit a single solution file in the repositories that contains your solution code with the best speed-up you have achieved until the deadline. Make sure this single file on the master branch contains the solution that you want to have graded.
- In case you use or adapt code from a work other than yours (internet tutorials, public repositories etc.) cite the source in a code comment. Doing otherwise will be considered as plagiarism and will be taken into account when grading your code.
- For your project presentation, further instructions are provided in the presentation template.
- **Note:** You must not not change the algorithm itself but perform the optimization and parallelization on the algorithm described above.
- To be considered a valid submission, your code should compile on the submission server. Therefore, be careful with the libraries used in your code. In case of unexpected compilation errors, contact your responsible tutor.
- You will be assigned a responsible tutor. Your tutor is available for answering questions and to help resolve any issues you might encounter.
- Write a brief README for each repository which explains the changes or additions in your code other than performance improvements. If you implemented new functions and data structures in your solution, explain these in your README.

Algorithm : Barnes Hut

One of the interesting problems in the domain of Astrophysics is the motion of stars in a galaxy. The study of the movement of stars over a long time (millions to billions of years) can help us determine the different stages between formation and the death of a galaxy. In order to perform such a study we consider a galaxy with a massive body (usually a black hole) at the center and billions of stars orbiting around it. Each of these stars has a relatively very small weight compared to the centre of the galaxy. The motion of the stars and the centre of the galaxy is governed by the overall force exerted at each of these bodies due to gravitational interaction with the others. The gravitational force[1] on a body due to another can be computed using following

$$F = Gm_1m_2\frac{\vec{p_1} - \vec{p_2}}{|\vec{p_1} - \vec{p_2}|^3} \tag{1}$$

Now let us assume that there are N bodies in the galaxy, and we want to compute the overall force on the body i then using the above equation, we can compute the overall force as follows

$$F_i = \sum_{j=1}^{N} Gm_im_j\frac{\vec{p_i} - \vec{p_j}}{|\vec{p_i} - \vec{p_j}|^3} \tag{2}$$

Once the overall force on a star is known, we can perform a simple forward Euler integration[2] to obtain the star's final position and velocity. For the sake of simplicity, we describe the integration steps here

$$\begin{aligned} v_i(t + dt) &= v_i(t) + dt * F_i \\ p_i(t + dt) &= p_i(t) + dt * v_i(t) \end{aligned} \tag{3}$$

Where

- $v_i(t)$ is velocity of body i at any time t
- $p_i(t)$ is position of body i at any time t
- dt is the time step size used for integration

The most trivial way to simulate the movement of the galaxy is by computing the pairwise gravitational interaction. Then for a particular star, we compute the force exerted on it by all the other stars, which requires $N - 1$ computations, and we perform this computation N times to find the overall force exerted on each star. This brute force method results in a computational complexity of $\mathcal{O}(N^2)$, which is prohibitively large when we are simulating billions of bodies. To overcome this issue Barnes Hut[3] algorithm was developed, which has a computational complexity of $\mathcal{O}(N \log N)$.

The fundamental idea used in the Barnes Hut method is the division of the simulation space into octrees[4], as shown in Figure 1. This results in an overall reduction in the computational complexity since all the stars are assigned to particular cells and only the stars that are in nearby cells need to be treated individually, and stars in distant cells can be treated as a single large object centred at the cell's centre of mass. Figure 2 shows an example of octree construction for a 2 galaxy interaction problem.

[1] https://en.wikipedia.org/wiki/Newton%27s_law_of_universal_gravitation
[2] https://en.wikipedia.org/wiki/Euler_method
[3] https://en.wikipedia.org/wiki/Barnes-Hut_simulation
[4] https://en.wikipedia.org/wiki/Octree

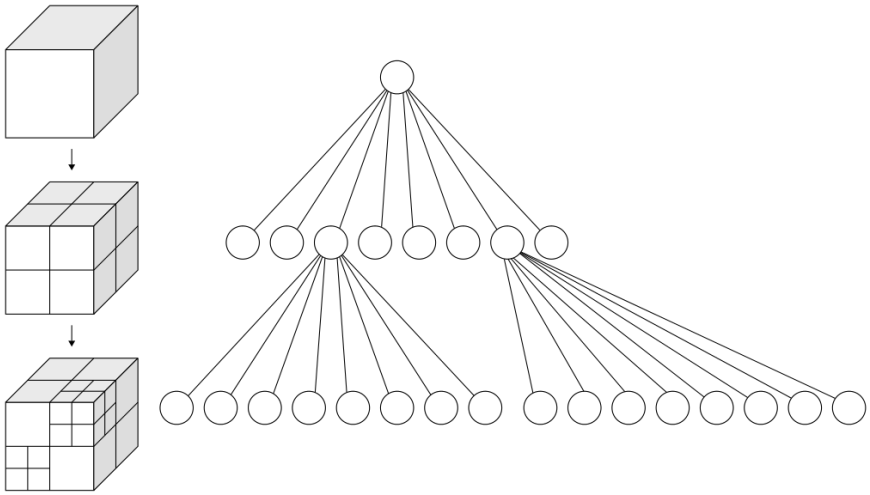
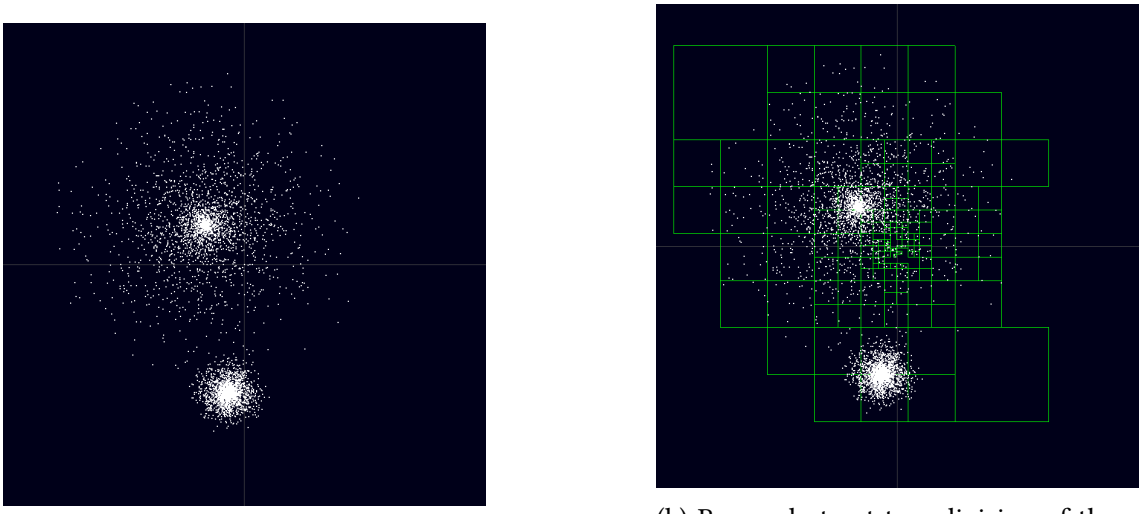


Figure 1: The plot shows the division of a $3D$ space in octrees.



(a) Distribution of stars in two galaxy system

(b) Barnes hut oct tree division of the space. The forces are computed using this division

Figure 2: Octree construction for a 2 galaxy system.

1 Octree Construction

For this specific case, we construct an octree such that each cell has the information of the children nodes, the combined centre of mass (of all children), and aggregate mass (of all children). Then we can describe the overall algorithm for the construction of Octree as follows

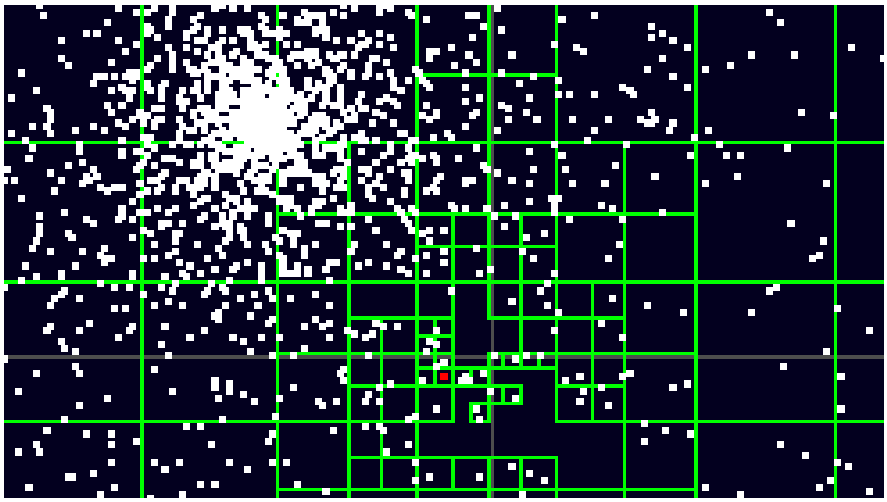


Figure 3: Zoomed region of Figure 2b

Algorithm 1: Octree Construction

Data: Array of points P with corresponding mass M
Result: Octree O

1 Start with root node as current node. Set the mass and position of tree to the mass and position of current node.;
2 If the given point is not in cuboid represented by current node, stop insertion with error. ;
3 Determine the appropriate child node, such as Up North East, to store the point. ;
4 If the child node is empty node, replace it with a node with representing the point. Stop insertion. ;
5 If the child node has a point, recompute the center of mass and total mass of the node using the current node and the existing child node. Reallocate (by calling insert on this) the point in child node to a subsequent child node by determining the appropriate region. And add the current node in appropriate region at the same level of hierarchy as the reallocated node.;
6 Repeat for all node ;

2 Force Computation

Once the tree is constructed, the force computation is done top-down, starting from the root node. For example, for a star i, the distance between the star i and the centre of mass the root is computed. If this distance is sufficiently large, the entire tree is considered as one big object, with its position being the centre of mass of the tree and mass being the accumulated mass of all the objects in the tree. In case the distance is not large, the same process is repeated for the 8 child nodes. This process is performed recursively until each of the tree nodes are sufficiently far or have only one star in them. Figure 3 (zoomed view of Figure 2b) shows the octree regions used for force computation for the redpoint. We can see that some distant regions have high star density and are used as one object for force computation.

For more detailed understanding of the method you can check out N-body Simulation from Princeton University[5] or The Barnes-Hut Galaxy Simulator by Ingo Berg[6]

[5] https://physics.princeton.edu//~fpretori/Nbody/intro.htm
[6] https://beltoforion.de/en/barnes-hut-galaxy-simulator/

Implementation : Short Documentation

For this exercise we use an interface pattern, where the main is oblivious of the type of parallelization being used. Following is the structure in which the serial code is organized.

- **main.cpp**:
  - The main calls a Compute kernel and outputs the performance metric (speedup).
  - You don't need to do anything here
- **Interface.cpp**
  - This method adds a level of abstraction between the compute kernel and main.
  - Interface performs a conditional compilation and chooses between different kind of parallelization method that are implemented. For example when OMP compute kernel is implemented we can compile the code with *BUILDOMP* flag and the interface would choose the OMP kernel for parallel implementation.
  - You don't need to do anything here
- ***_interface.cpp**:
  - We have 3 interfaces corresponding to the 3 cases: omp_interface.cpp, mpi_interface.cpp and hybrid_interface.cpp
  - This interface performs data initialization for parallel case as well as set up the data structure necessary for each kind of parallelization strategy.
  - *This interface calls the actual kernel that has to be implemented by you.*
  - You don't need to do anything here in case of OMP and MPI implementation, but you should implement the interface for Hybrid case
- ***bh.cpp**:
  - We have 3 kernels corresponding to the 3 cases: ompbh.cpp, mpibh.cpp and hybridbh.cpp
  - These must be implemented by you following the algorithm and data structure pattern provided in the serial part. You are allowed to perform necessary and justifiable changes in data structure (This should be well reasoned and justified in presentation and code documentation).
  - You should implement this.
- **serial.cpp**: This file is deliberately kept lengthy to keep a unified single file submission framework. For the ease of understanding and to avoid clutter namespacing is used. You should follow similar convention in the parallel implementation
  - This provides a serial implementation of the Barnes Hut Algorithm.
  - The class *Octree* implements the tree construction method.
  - ~~The *TreeForceCompute* method implements the force computation from the octree.~~ The *BodyInteract* method in *Utility.cpp* implements the force computation from the octree. (note: this method overloads *BodyInteract* method for pairwise force computation)

  - ~~The *PairwiseForceCompute* method computes the force between two stars.~~
    The *BodyInteract* method in *Utility.cpp* implements the pairwise force computation. (note: this method is overloaded by *BodyInteract* method for octree force computation)
  - ~~The *UpdatePosition* computes the new position of each star using Forward Euler Integration.~~
    The *PositionUpdate* method computes the new position of each star by calling *Integrate* method from *Utility.cpp*.
  - The *UpdateStep* method computes the overall force on each body at every time step and updates the position. This is where the octree is constructed and the main logic is implemented.
  - You can perform optimization in this.

- **utility.cpp**:
  - This file contains the methods that are used for setting up the problem such as initialization and command line parsing.
  - *ParseArgs* implements command line parsing and returns the number of bodies and number of simulation steps.
  - *Initializer* initializes the memory buffer with position and mass of the stars.
  - You don't need to do anything here. Any optimization here will not be taken into account on the submission server but you are welcome to be creative and report your findings in the final presentation.
    The header file marks a 4 functions which are open to optimization, if you optimize them then please provide their implementation in your solution file using namespace of *solver*, for example OMP, MPI, Hybrid. Also describe the optimization performed and use them at appropriate location in your parallelized code.

- **Constants.h**:
  - This file implements the basic data structures and defines several constants that are necessary for this problem.
  - *vec3* is used to represent $3D$ vectors.
  - *body* is used to definition of any body with position, velocity, acceleration and mass.
  - You don't need to do anything here. Any optimization here will not be taken into account on the submission server. But you are welcome to be creative and report your findings in the final presentation.
    It should be noted that redefinition of the constants would be considered an invalid submission!

- **Makefile**:
  - Implements a recipe (compilation commands) for different kind of parallel implementation.
  - You can use commands such as *make ompbh*, *make mpibh* and *make hybridbh* to compile the source code and build the executable for the three case respectively.

- **Execution instructions**: For an executable ,say*bh*, following command runs the program

```
$ ./bh -n <number of bodies> -t <number of timesteps>
```

Tasks

1. **General:**

   This task includes preliminary optimisations on the sequential code without parallelising. You are not expected to submit a separate file for this task but by optimising the sequential version provided, this task aims to help you achieve better speed up for following parallelisation tasks .

   Go through the above algorithm and analyze the given sequential code. Where do you see possible improvements through parallelization? Attempt to optimize the sequential part of the code before jumping to parallelization part. Explain the logic for each optimization step.

   Note: It is mandatory that you do not change the algorithm itself but perform the optimization and parallelization on the algorithm described above. Changing the algorithm would result in invalid submission.

2. **OMP:** *Due 15.06.2021 23:59, Submission file "ompbh.cpp"*

   Parallelize the computations using OMP. Parallelize as much as possible as long as it does not harm performance. Attempt to parallelize the octree construction, force computation and position updates. You do not have to parallelize the problem initialization. Explain the parallelization strategies used and the logic behind them. It is recommended that you use a diagram to show parallelization strategy and the data required at each step of parallelized algorithm.

   Remark: With an improved sequential code and 4 OMP threads, you should expect a speedup of at least 1.6 for 32768 bodies.

3. **MPI:** *Due 22.06.2021 23:59, Submission file "mpibh.cpp*

   Assume (hypothetically) that you have to run your code on a cluster without shared memory access and only one core per compute node. Parallelize the computations using MPI. Attempt to parallelize the octree construction, force computation and position updates. Again there is no need to parallelize the problem initialization. Explain the parallelization strategy, type of communication and the logic behind it. It is recommended to use a diagram to show parallelization strategy and the data required in each step of parallelized algorithm.

   Remark: Assume that the initial position of the bodies and their masses are only available at RANK 0.

4. **Hybrid (MPI + OMP):** *Due 29.06.2021 23:59, Submission file "hybridbh.cpp*

   Assume (hypothetically) that you have to run your code on a cluster with several distributed nodes each having multiple cores with shared memory. How can you leverage this architecture in a hybrid approach using MPI and OMP? Attempt to parallelize the octree construction, force computation and position updates. Again there is no need to parallelize the problem initialization. Explain the parallelization strategy, type of communication and the logic behind it. It is recommended to use a diagram to show parallelization strategy and the data required in each step of parallelized algorithm. Also justify when and if it makes sense to use hybrid parallelism for this algorithm.

   For this task, you are also required to implement the hybrid interface that will connect your implementation of the solver to the main of package. You should use the files *mpi_interface.cpp* and *mpi_interface.h* as reference and perform a similar implementation in *hybrid_interface.cpp* and *hybrid_interface.h*. You are not required to submit these files but they will be necessary for you to test your implementation.

Note: Please stick to the problem initialization pattern described in the *mpi_interface.cpp* file, using a different problem initialization could cause your submission to fail or could increase the runtime.

5. **Bonus (Optional):** *Due 29.06.2021 23:59, Submission file "bonusge.cpp*

   Given your hybrid implementation (MPI + OMP), can you observe any speedup using one of the following of your choosing: SIMD intrinsics, OMP GPU offloading , CUDA or HIP ? Explain how and provide necessary details.

   Note: Implement this using intrinsic operations and not OMP SIMD.

6. **Presentation Slides:** *Due 06.07.2021 23:59, Submission file "PPSS21_final_project.pdf or .ptx*

   Use the template provided in the Documentation repository for your presentation slides. Submit your presentation into the Documentation repository.