GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Project report

Visualization of Circle Collisions using Quad- and Octa-trees

prepared by

Eliah Windolph, Evgeni Uschakov

Georg-August-University Göttingen

Course title:	Practical Course on High-Performance Computing
Supervisor:	Prof. Dr. Julian Kunkel
Submission date:	30.09.2023

Contents

1 Introduction

1.1 Motivation

Detecting and handling circle collisions is a computation-intensive task. Due to the underlying processes, including the calculation of Euclidean distances, formulation and assignment of resulting trajectories, as well as the resolution of overlaps after collisions, the implementation needs to be addressed with care. As the number of circles and the subsequent amount of collisions per second increases, the demand for fast, efficient and reliable algorithms and data structures grows. For that there exists a compelling solution using spatial trees, such as quad- for 2D and octa-trees for 3D applications, as well as higher-dimensional trees as needed. This data structure provides a well suited space partitioning approach for this task and is being used in engineering and robotics for safety and precision when dealing with collision detection as well as in physical simulations such as n-body simulations. Well-documented tests can provide detailed insights into further possible improvements of this approach and its impact of adding parallel processes in Message Passing Interface (MPI) based solutions, enabling an evaluation of the achievable scalability and efficiency.

1.2 Scope of the project

This project aims to create a physical simulation, but instead of focusing on the correctness and realism, we focus on the scaling of the simulation. The resulting application should be able to visualize an n-body simulation with object collision, excluding gravity. For this to be possible we decided for two simplifications: first, limiting the simulation to two dimensions, and second, exclusively check collisions between circles for which the collision checks are simpler than for other objects (see chapter 1.3). We choose C as our programming language since it should be optimal for high performance applications.

We first look into different solutions for performing circle collision and their implementation with MPI. In chapter 5 we talk about challenges in their development before we move on to the performance chapter. There we compare the average iterations per second (IPS) achieved with variable amounts of circles, processes and other parameters, between the introduced implementations. We then look into these results and show each strengths and weaknesses of our implemented ideas, before giving an outlook of possible next steps and improvements.

1.3 Collisions

Since we only look at collisions in 2D it is simple to see if two circles collide. We just have to get the distance between the two centers and check if it is smaller or equal to the sum of the radii of the circles. Mathematically, it looks like this:

$$\sqrt{(c1.x - c2.x)^2 + (c1.y - c2.y)^2} \leq c1.r + c2.r \tag{1}$$

If we now add the simplification that all circles have the same size we get:

$$\sqrt{(c1.x - c2.x)^2 + (c1.y - c2.y)^2} \leq 2r \tag{2}$$

Now we know how to check if two circles are colliding, so the next question is how do we solve the collision impact and calculate the new velocities. For our main objective of analysing the scalability and efficiency, we use elastic collision, which means no kinetic energy is lost in thermal energy or deformation. That means:

$$\begin{gathered}\sum E_{kin} = \sum E'_{kin} \\ \frac{m_1}{2}v_1^2 + \frac{m_2}{2}v_2^2 = \frac{m_1}{2}{v'_1}^2 + \frac{m_2}{2}{v'_2}^2 \\ \frac{m_1}{2}(v_1^2 - {v'_2}^2) = \frac{m_2}{2}({v'_2}^2 - v_2^2)\end{gathered} \tag{3}$$

Additionally we know:

$$\begin{gathered}\sum p = \sum p' \\ m_1v_1 + m_2v_2 = m_1v'_1 + m_2v'_2 \\ m_1(v_1 - v'_1) = m_2(v'_2 - v_2)\end{gathered} \tag{4}$$

If we put all of this together and add the simplification that the masses of the two circles are the same we get the following result:

$$\begin{gathered}v'_1 = v_2 \\ v'_2 = v_1\end{gathered} \tag{5}$$

The problem with this is that we only check each frame if two circles collide and change their position based on the velocity. Therefore it is possible that the circles overlap and become stuck in a deadlock situation, as the velocities are swapped every frame while the overlap remains unresolved. In order to circumvent this problem we adjusted the collision calculation.

The following pseudocode shows a vague implementation of our `checkCollision` function.

```
1  def checkColission(Circle circle1, Circle cirlce2):
2      double dx = circle2->posX - circle1->posX
3      double dy = circle2->posY - circle1->posY
4      if fabs(dx) > circleSize or fabs(dy) > circleSize then
5          continue
6      end
7      double distSquared = dx * dx + dy * dy
8      if distSquared < circleSize * circleSize then
9          double dist = sqrt(distSquared)
10         double overlap = (circleSize - dist) / 2.0
11         dx /= dist
12         dy /= dist
13
14         circle1->posX -= overlap * dx
15         circle1->posY -= overlap * dy
16         circle2->posX += overlap * dx
17         circle2->posY += overlap * dy
18
19         double dvx = circle2->velX - circle1->velX
20         double dvy = circle2->velY - circle1->velY
21         double dot = dvx * dx + dvy * dy
22
23         circle1->velX += dot * dx
24         circle1->velY += dot * dy
25         circle2->velX -= dot * dx
26         circle2->velY -= dot * dy
27
28         circle1->velX *= friction
29         circle1->velY *= friction
30         circle2->velX *= friction
31         circle2->velY *= friction
32     end
33 function end
```

In line 4 we make a small check, if the positional distance between the x or y coordinates is bigger than 2 times the radius (`circleSize` = diameter). If false, the circles cannot collide. We then get over to check if the two circles collide by using the formula 2 squared. Then we resolve the overlap, change the velocities and add a friction factor. Even though we technically used elastic collisions we thought it would make a more interesting simulation if the circles could slow down over time with a optional parameter friction (for no friction we can set it to 1).

2 Naive Implementation

2.1 Idea

The idea of the naive implementation is to calculate the distances for each circle pair. This is done efficiently by only checking all circles with the ones having a bigger id than the circle for which we currently check collisions. This reduces a lot of useless computation. If a circle with id 1 collides with circle 5, we calculate the collision for both circles by manipulating the velocities and resolving possible overlaps for both circles at once. Now there is no need to detect the collision between its reverse order, circle 5 with circle 1. We added this optimization in order to avoid calculating collisions twice for each circle pair. Another way of solving this problem would be to just change the attributes of the circle which initiates the check which will be needed later in the MPI versions. This brings us to the following main loop, shown in pseudocode 1 below.

```
class Naive Implementation:
    def mainLoop:
        for i from 0 to numCirlces-1 do
            for j from i+1 to numCircles do
                checkCollision(i, j)
            end
        end
        for i from 0 to numCirlces do
            move(i)
        end
    function end
class end
```

Algorithm 1: The naive implementation

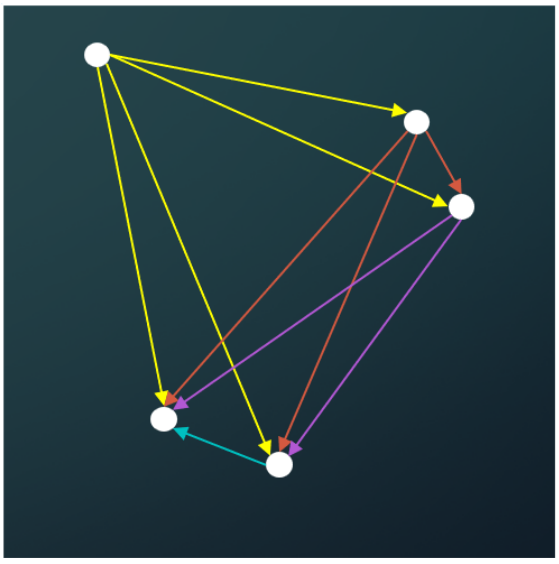

Figure 1: Naive Implementation

2.2 MPI Extension

The idea behind its MPI solution is to parallelize the main loop. Instead of iterating over all circles we split the array into equal sized parts so that each process handles the same amount of circles. Since every circle calculates the collision with each other circle, we have to make sure that every process has the newest data from all circles of the current iteration. For that reason, we chose the MPI function `MPI_Allgather`. Each process sends their section of the circles-array to all other processes, so that every rank receives all updated data for each section (see figure 2). Furthermore we need to make sure that our `checkCollision` implementation only changes the data of the circle which initiated the check and detect collisions with each other circle (instead of just the ones with higher ids, like in the non-MPI implementation). Since two circles might be checked by different processes and each process only sends its own circle data, collisions between two circles from different sections might get lost.

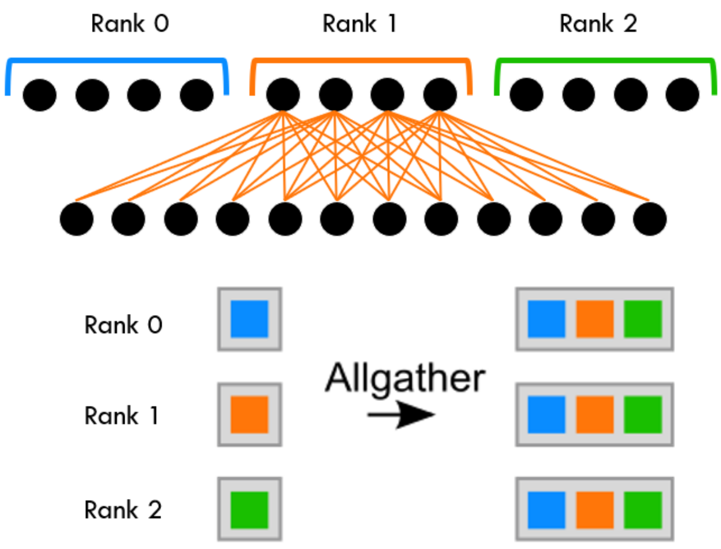

Figure 2: Naive Implementation with MPI

3 Static Tree

3.1 Idea

The idea behind the static tree approach is to reduce the number of collision checks by introducing space partitioning. This is done by dividing the whole field into multiple sub-areas. When assigning each circle to its part on the field, based on its location, we can be certain that circles from different areas are too far apart from each other in order for collision to occur. Therefore, instead of calculating the distances between all circles, we reduced the problem by checking circles inside each sub-area, which we from now on call cell.
The importance of space partitioning becomes clear, when shown as a performance comparison on an example. Lets assume there are $n = 100$ circles. The brute force approach would result in $n^2 = 100^2 = 10000$ collision checks. Our naive implementation with and without MPI reduces it to $\frac{n^2}{2} = \frac{100^2}{2} = 5000$. When adding rigid space partitioning using a grid of size 5x5, the number of collision checks drops down to around $5 \cdot 5 \cdot \frac{(\frac{100}{5 \cdot 5})^2}{2} = \frac{10000}{50} = 200$, assuming the circles are equally distributed across the area, and every circle is inside exactly one grid-cell. When enhancing the resolution of this grid to the point where the cells have the size of a circle and assigning each circle the location of its respective cell, there are no collision checks needed, until a circle leaves its cell and overlaps with one that is already occupied. A good balance between the consumption of memory and the number of collision checks needs to be found in order for the solutions to work most efficient. A problem following the space partitioning approach are circles along borders between grid cells. Then all overlapped cells have to calculate the collisions for these circles. The worst-case scenario occurs, when every circle overlaps with 4 grid cells.

Another issue emerges when the circles are not evenly distributed or many more circles are added. In that case some cells may need to perform much more work, while others don't contain any circles. This issue happens due to the missing load balancing of the static grid approach.

3.2 Implementation

In order to minimize these problems, we need a clever way of achieving a well balanced grid with dynamic splitting, so that every cell surrounds roughly the same number of circles. For this we use a quad-tree. In this approach each cell has a position (x,y) and a width and height. Based on this we know the area each cell covers. Additionally it will store two pointers, one to an array which contains its circles and one for optional subcells, generated by a split operation.

Before diving into the specifics of our quadtree implementations, a couple topics need to be addressed first. Performing basic array operations directly on circle objects takes much time. Copying a circle to a different memory location thousands of times per second seems to be not efficient. To prevent this, the array elements can be replaced by basic integers being the indices of the circles in a predefined global array. Since there can only be one version of a circle to draw and perform circle collision with, copying a circle to a different memory location becomes a simple assignment of an integer to the desired field. This also reduces the sizes of memory allocations on runtime, which should increase the performance substantially. Furthermore hashsets can be used for storing distinct circles inside the cells for faster lookup times.

We will always have the so named root cell which has position (0,0) and size (ScreenWidth, ScreenHeight). Adding circles to the tree works by inserting them into the root cells circles array. Once it exceeds a maximum, a parameter called `maxCirclesPerCell`, the cell splits by creating four subcells dividing the area into four parts (see pseudocode 2 and the examples in figures 3 and 4).

```
def split(cell pointer):
    subcells = createSubcells(cell)
    for int i = 0; i < cell.numCirclesInCell; i++ do
        int circle_id = cell.circle_ids[i]
        for int j = 0; j < 4; j++ do
            struct Cell* subcell = &cell.subcells[j]
            if !isCircleOverlappingCellArea(circle_id, subcell) then
                continue
            end

            addCircleToCell(circle_id, subcell)
        end
    end
    free(cell.circle_ids)
    cell.circle_ids = null
function end
```

Algorithm 2: The split function

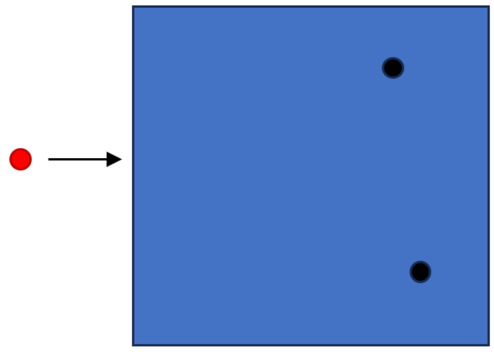

Figure 3: Before split

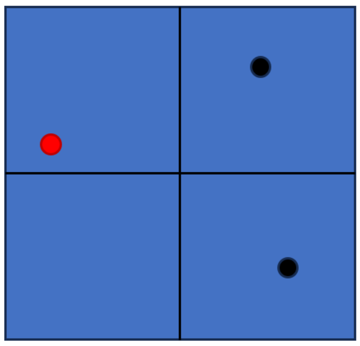

Figure 4: After split

By doing so we will end up with smaller cells that ideally each contain less circles than our maximum. If we now add new circles, the root cell will check in which child's area the circle belongs to and inserts it there. This part can be seen in pseudocode 3. Note that subcells can split too if they reach their maximum, becoming parents themselves.

```
def AddCircleToTree(circle, cell):
    if cell.isLeaf: then
        if cell.numCirclesInCell < maxCirlcesPerCell then
            cell.circles.append(circle)
        end
        else
            split(cell)
            addCircleToTree(circle, cell)
        end
    end
    else
        for i from 0 to 4: do
            if isCircleOverlappingCellArea(circle, cell.subcells[i]) then
                addCircleToTree(circle, cell.subcells[i])
            end
        end
    end
function end
```

Algorithm 3: Add circle to tree function

Now the main loop consists out of generating the tree by iterating over all circles and adding each to the root cell. Then it calculates the collisions inside of all generated leaf-cells, updates the velocities and resolves overlaps, before deleting the tree and repeating this process for every frame.

A full example of the tree structure can be seen in figure 5.

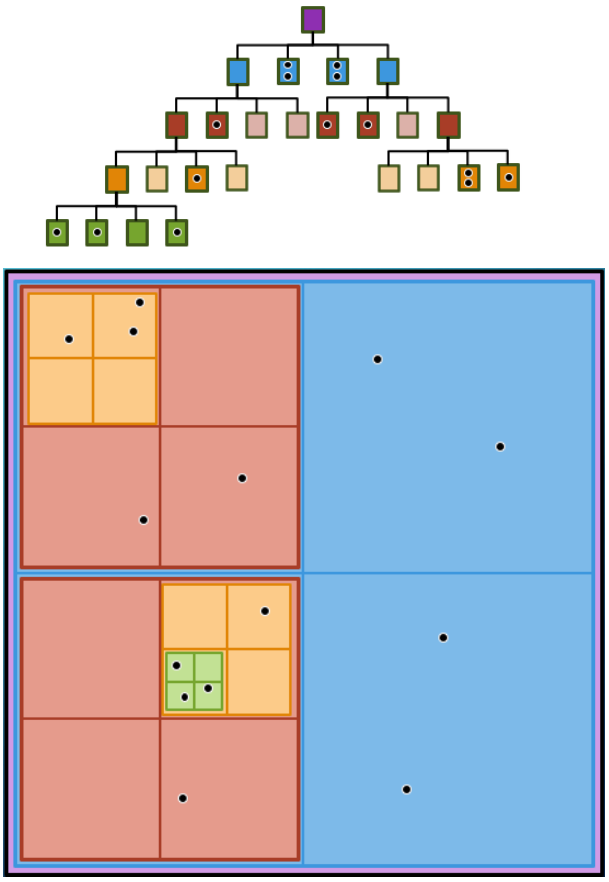
Figure 5: Visualized tree (maxCirclesPerCell=2, numCircles=14)

4 Dynamic Tree

4.1 Idea

After exploring the static tree approach, it becomes evident that there is no need to always fully rebuild the tree since it creates huge workloads by redundantly performing identical calculations. In order to achieve a better solution, we thought of a new approach by splitting and collapsing cells, accomplishing a dynamic tree capable of autonomously managing its cells. If a cell contains more than `maxCirclesPerCell` circles, it splits into four subcells as previously done. However, once one or more circles leave the area of a parent cell, it collapses, deleting its subcells and merging their circles into the parent cell. That way it fixes the issue at hand, but other difficulties emerge. In order to detect collapsable cells, we need to keep track of the amounts of distinct circles inside the parent cell areas. If a circle moves to the border of multiple cells, it is being stored in each cell it overlaps. That way, each cell detects collisions but the parent cell might misinterpret the copies as different circles, which leads to non-collapsing cells.

4.2 Implementation

Everytime we recursively update the tree, each cell checks whether its circles remained inside its area. Once a leaving circle has been detected, the function `deleteCircle` is being called on the rootCell. This deletes the circle from every cell containing it but is not overlapping anymore. Simply removing the circle from this cells array and then inserting it into other cells would lead to a lot of repeated work. If a circle leaves two cells at the same time, as shown in figures 3 and 4, both cells detect this leaving circle and perform identical operations to find the next cell for it. Moreover, once the adding process inserts this circle into a currently unvisited cell, it will get checked again. So instead we delete outdatedly assigned circles in every cell that is close to it by using the helper function `isCircleCloseToCellArea`. After deleting the circle, it is added to its right cell by the function `addCircleToParentCell`. This traverses the tree in bottom-up direction by using a parentCell reference in the subcells. If a circle is still not in the area of the parentCell, then the counter numCirclesInCell gets reduced by one. If a parentCell surrounds the circle, it is being inserted there by calling `addCircleToCell`, which leads to the according leaf-cell. If only a part of it was inside the parentCell, then a copy must be sent further up the tree continuing the traversal, until it reaches the rootCell which would be the worst-case. At the end of every update call, the code compares the `numCirclesInCell` of the current cell to `maxCirclesPerCell`. If it is smaller or equal, then collapse is called on this cell. That way, a problem shows up when circles are being added to their right circles, which might lead to an unwanted split operation, since some circles of the splitted and not yet updated cell might have left the `parentCell`, following up with a collapse operation once the traversal reaches it. In order to fix this issue, the collapse and split functions would have to run after udpating the tree. Our tests showed no FPS gain using either approach, since such a function requires a second traversal of the tree for every frame.

On an `addCircleToCell` call, the correct cell for the new circle is found in $O(\log n + k)(n =$ number of leaf-cells, $k =$ number of circles inside the found cell) time, assuming the circles are equally distributed across the field, since its just one path down the tree and a check if a copy of this circle is already inside it. The function `deleteCircle` has in addition to the lookup time, the time complexity $O(n)$ with n

being the number of circles inside the found cell, due to the removal of a circle and the rearrangement of the following circles inside the array. The `split` operation needs to create four sub-cells in a leaf-cell and insert the circles of it into them in $O(n)$ time. In contrary, the runtime of the `collapse` function is more difficult to estimate. Since it could start from any `parentCell`, it needs to traverse over its whole sub-tree. Every circle inside the visited leaf-cells need to be inserted into the collapsing cell. Due to the fact, that we only collapse cells of `maxCirclesPerCell` circles or less, there are only this many to be added. So the time complexity in this case is $O(n+k)$ with n being the number of leaf-cells in the sub-tree and $k \leq$ `maxCirclesPerCell` being the number of circles to be inserted. An `addCircleToParentCell` call takes the time of iterating the bottom-up path combined with multiple `addCircleToCell` runs. The amount of calls is determined by the overlaps of the circle with multiple cells along different paths.

4.3 MPI Extension

We thought of multiple ways on how MPI can improve the performance of our dynamic tree.

One idea is to split the field in multiple areas surrounding parts of the circles. That way we basically replace the first split of the root cell by manually creating sub-cells operating on different processes. These can not be collapsed, since they are no real cells but simple rectangular parts of the whole field managing their own root cells of size (AreaWidth, AreaHeight) at their assigned positions. These areas are stored inside a processes array, and get synchronized with all processes using MPI_Bcast. That way every process knows the assignments of the areas on the screen to its process ranks. We still need to communicate which circles are in which area. For this our first approach is to use the process 0 as a master rank. It will handle the communication and distribute the circles. Each other process calculates its tree with the circles that are inside their respective area, checks for collisions in their tree, update the circle positions, and send the updated data back to process 0. This master rank collects all updated circles, creates the new frame for the visualization, and sends the data back to all processes after assigning the ranks to each circle based on their new position. This approach has multiple flaws. First of all, the process 0 has to run through the circle array of each process to update its global circles array. This scales bad with higher amounts of processes, since in addition to waiting for the master to finish, each worker process generates multiple MPI calls per frame. Another problem is the missing load balancing for specific cases. If a lot of circles are in a few areas (i.e. if we add gravity most circles will fall to the bottom), their processes will have to calculate a lot of collision checks while others run idle. Furthermore we have a communication bottleneck, since all communication flows through the process 0, instead of messaging the corresponding processes directly. So another idea is to use direct messaging (DM) for communicating leaving and entering circles across areas.

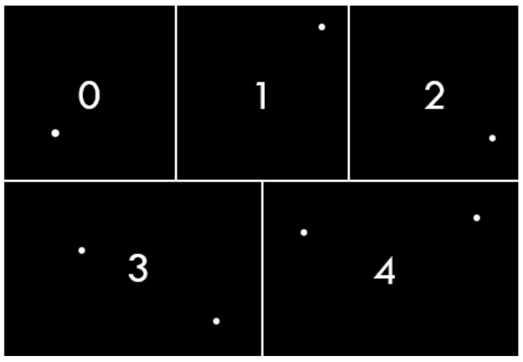

Figure 6: MPI approach visualization with 5 processes

A different approach is comparable with the MPI-version of our naive implementation. Each process will get an equal amount of circles for which it will calculate the collisions. The drawback is that all processes need to build the whole tree, even if none of its circles are in sub-trees. Another problem is that circles do not know in which cells they are, so we have to find the cell in which the circle is, by traversing trough our tree. This can be solved by adding a mapping from circles to cells. We did however not implement this version.

5 Challenges

In this chapter we will discuss challenges we had with the project. The main challenges where in the implementation of the dynamic tree approach. There are many edge cases one has to think about. In addition to that, exceptions where thrown multiple iterations later than the real error occurred, which made the debugging process even harder. One small example of one of these edge cases is the following:

Assume a circle is on the edge of two cells, which both have the same parent. Now the circle is leaving both cells in the same frame like you can see in figure 7.

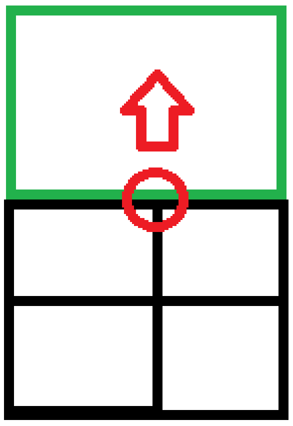

Figure 7: An example edge case

Our first implementation just checked if a circle left the cell. If it has, we reduce the counter for the circles and check if the circle also left our parent. If so we reduce its counter as well. We would do this until we found a cell which contains the circle and return. The problem now is that both cells will recognize that they lost a circle and will both tell their parent to check if it contains it. The parent will both times see that the circle in question is out of its scope and reduce the amount of circles by 1, so 2 in total, even though it only lost 1 circle. This would not always throw an exception just yet. Maybe the parent has still more than `maxCirclesPerCell` and therefore wont collapse. It may collapse multiple iterations later which made finding this bug much harder, since everything looked fine apart from the wrong counter. In addition, this was especially hard to debug in the MPI-versions due to the much more difficult tracking of variables of multiple processes on runtime.

Another problem occurs on the communication of circles between processes in the MPI-implementation. A circle is only being sent when it fully leaves the root cell area of a process. While overlapping two areas, a collision will only be registered by the one process, the circle is assigned to. Circles from different processes will not affect this circles trajectory on visible collision. But since the diameter is chosen to be 1, and the position difference in the x- and y-axis per update is at most 1, this only occurs if the circles switch areas at the same position on the border and their decimal place value inconveniently matches.

6 Performance

In this chapter we will look at the performance of the different implementations. In order to compare them we use the average iterations per second (IPS) which is comparable with the more known average frames per second, but since we did not output the visualization, we chose the former. The time frame of measuring the IPS starts on the first update call and ends once 5 seconds have passed.

6.1 Naive Implementation

As can be seen in figure 8, most of the workload is caused by collision checks. The communication after the initialization (the previously mentioned MPI_Allgather) needs only a relatively small amount of time.

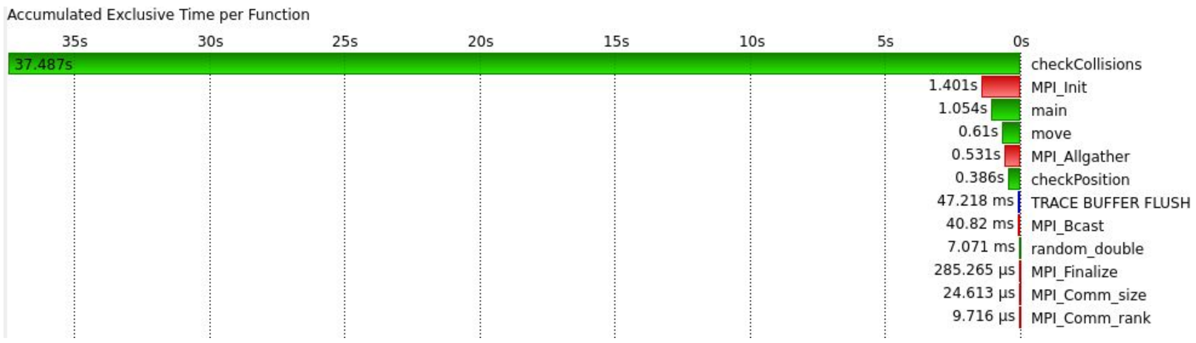

Figure 8: Naive workload

6.2 Static Tree

As figure 9 shows, we greatly reduced the time it takes for performing collision checks and shifted it towards the tree building functions, of which the three biggest are `addCircleToCell`, `isCirlceOverlappingCellArea`, and `split`.

The reason why `addCircleToCell` takes so long is, since it needs to traverse the whole tree at least to one leaf while checking in each recursive call if the circle is in all four possible sub-trees.

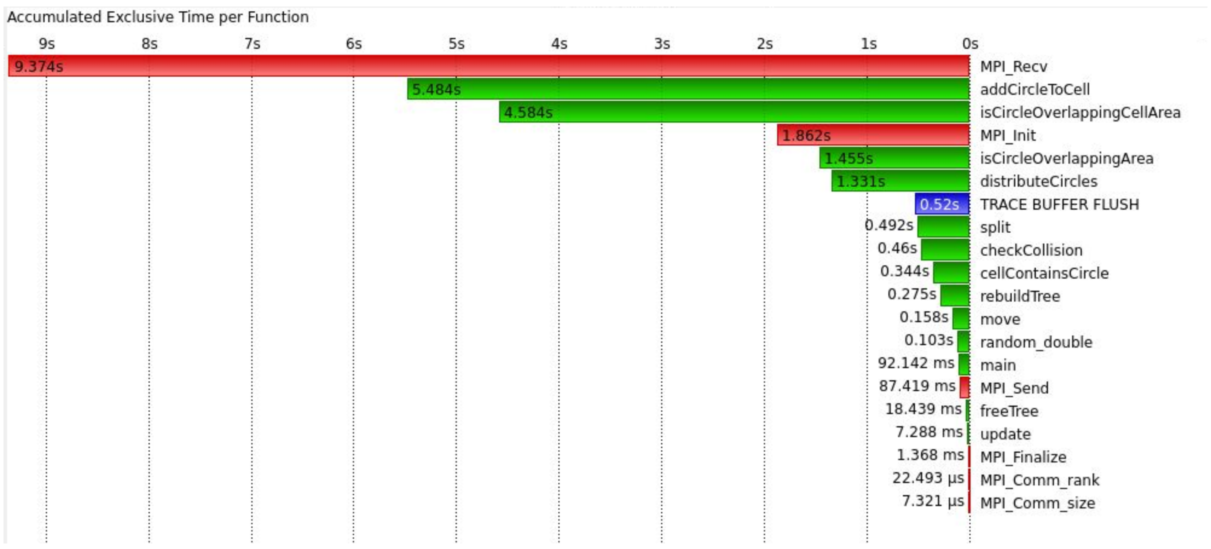

Figure 9: Static Tree Workload

The reason why `MPI_Recv` takes up so much time is easier so understand once we take a look at the workload distribution of the different processes. This can be seen in figure 10.

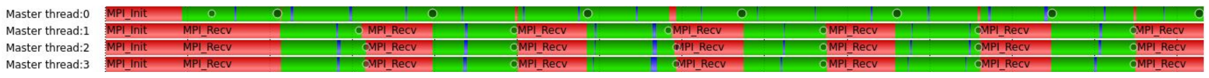

Figure 10: Static Tree Workload per process

Process 0, which is responsible for the circle distribution, has to do way more work than the other processes which leads to long waiting times in these. A better way to distribute the circles or the work could greatly benefit this approach.

6.3 Dynamic Tree

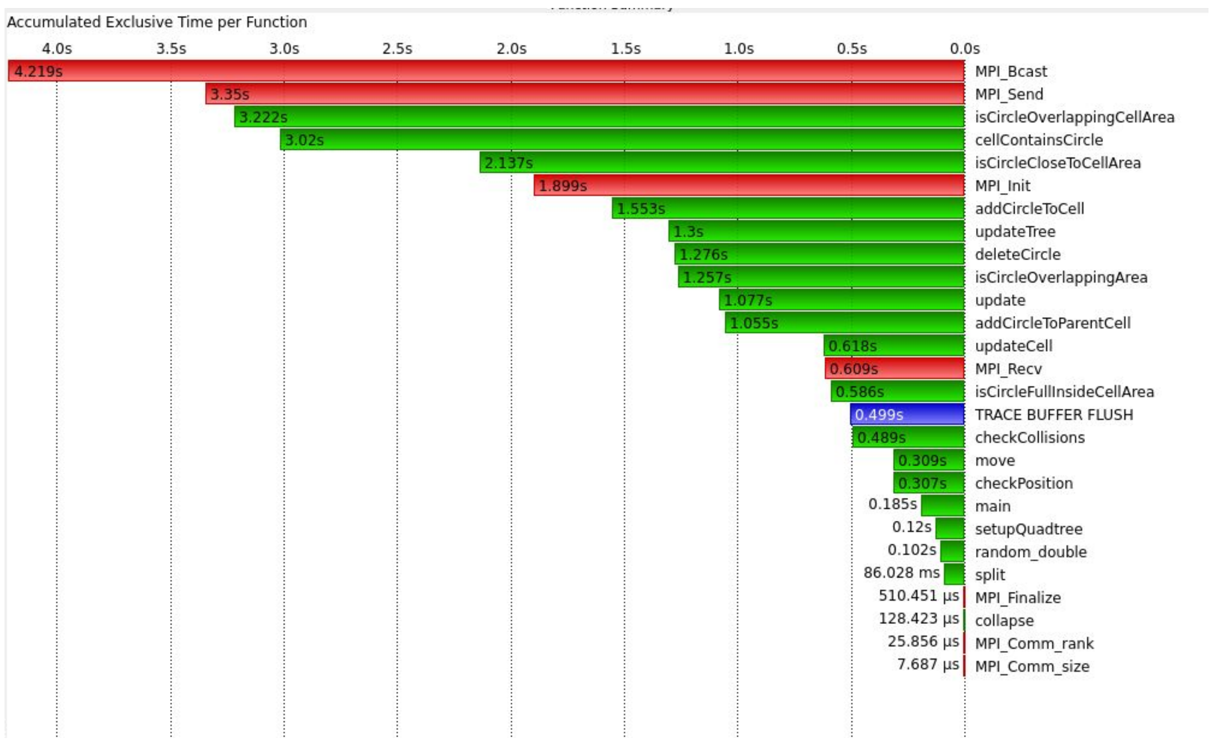

Figure 11: Dynamic Tree Workload

As you can see in figure 11 and 12 the dynamic tree approach suffers from the same problem as the static tree. The work is not balanced well enough so processes are waiting for the new data instead of calculating the new collisions.

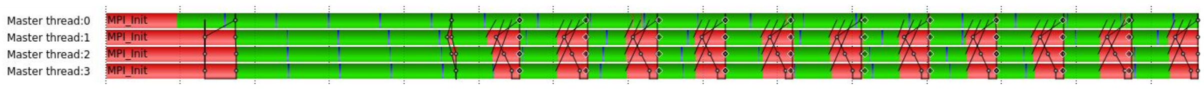

Figure 12: Dynamic Tree Workload per process

6.4 Dynamic Tree with MPI

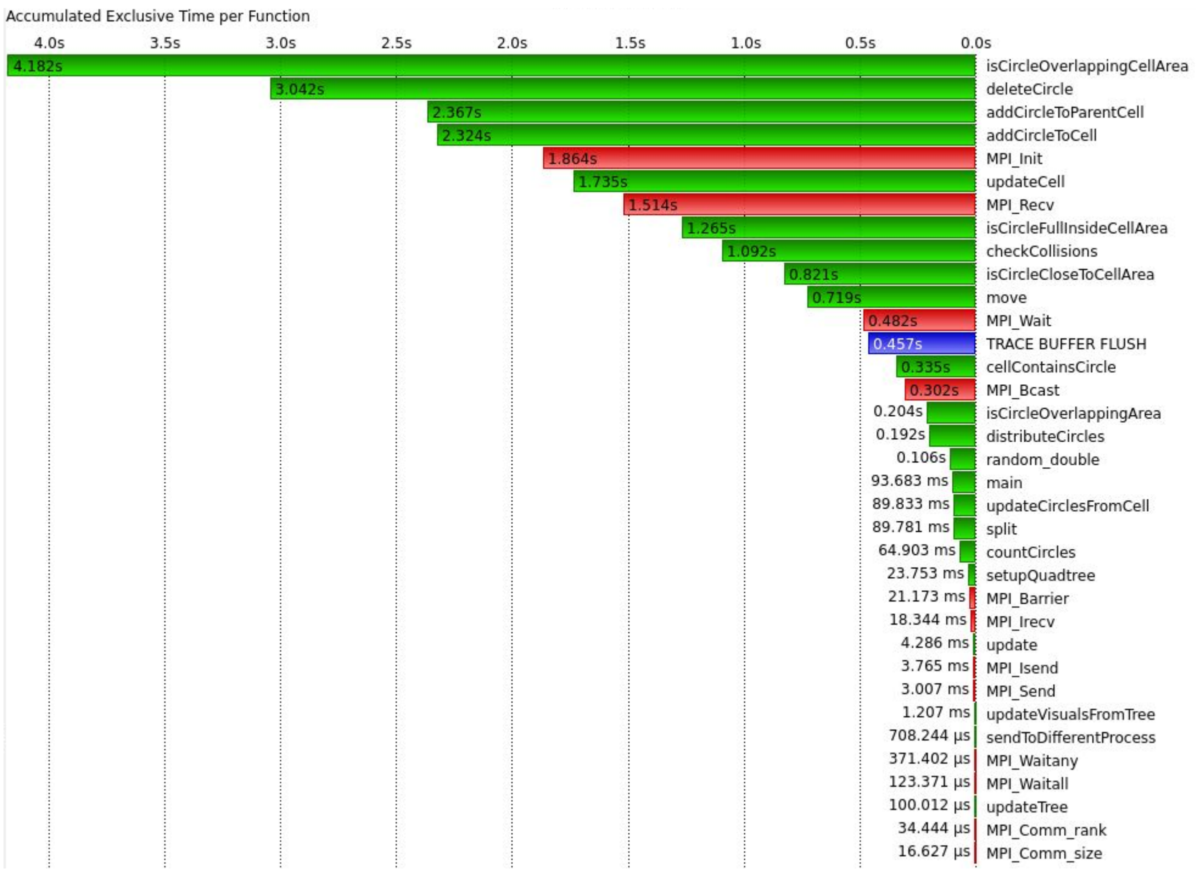

Figure 13: Dynamic Tree DM Workload

As can be seen in figure 13 and 14, the Direct Messaging approach distributes the work much better. In the beginning, process 0 still distributes all circles to the corresponding circles which means the other processes have to wait. But after that, the communication is no longer a bottleneck. This means less idle time which should result in more actual time used for the calculations. The performance gain will be discussed in the next section.

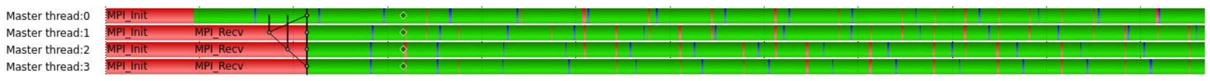

Figure 14: Dynamic Tree DM Workload per process

6.5 Comparison

In all the following tests, we used the average IPS over 5s in order to compare the performance of the algorithms. A higher IPS counter means the version could in reality create more frames in the same time and is therefore better.

In the benchmark in figure 15 you can see, we kept the amount of cores fixed at 1 and tested how the IPS develops if we change the amount of circles. Not surprising is, that all implementations get worse with more circles, since all the implementations need to handle more circles and check for more collisions. Additionally we can see that our trees have a big advantage over the naive implementations, which gets smaller the more circles we get.

The more interesting test case is the one in which we increased the amount of processes to see the scaling of each approach. The results can be seen in figure 16. The naive approach as well as the static and dynamic trees actually get worse with more processes. The reason for this is the additional workload process 0 has to handle. The DM and Hash-set (HS) approaches on the other hand get better performances with increasing processes. We assume that the peaks come from different loads on the cluster we ran the benchmarks on. Even tough we ran the benchmarks multiple times we still got peaks. But we can see a clear performance gain using bigger numbers of processes.

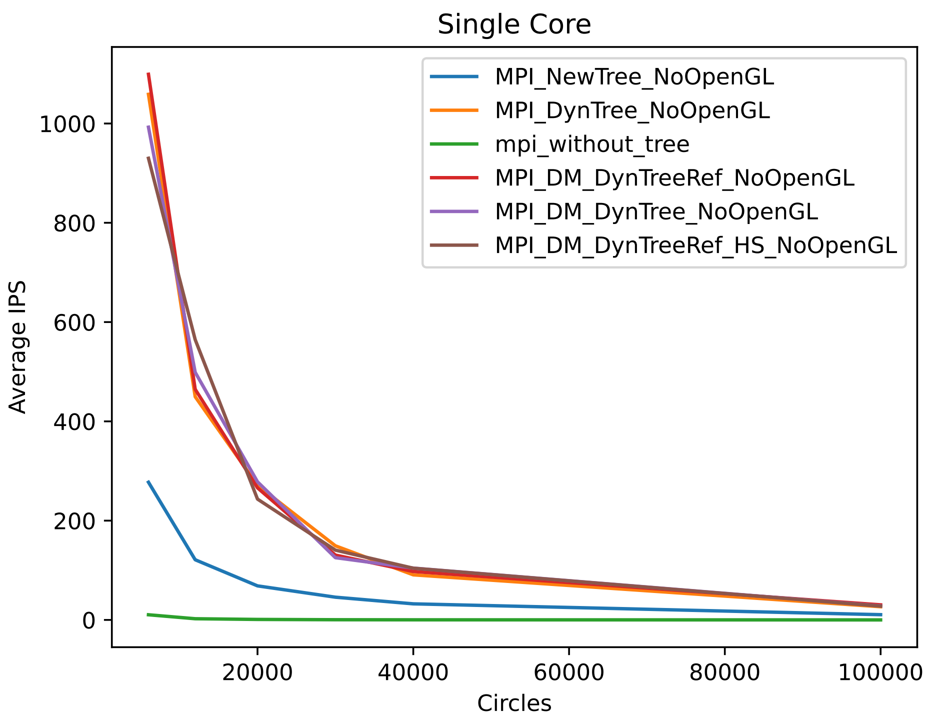

Figure 15: Single core comparison

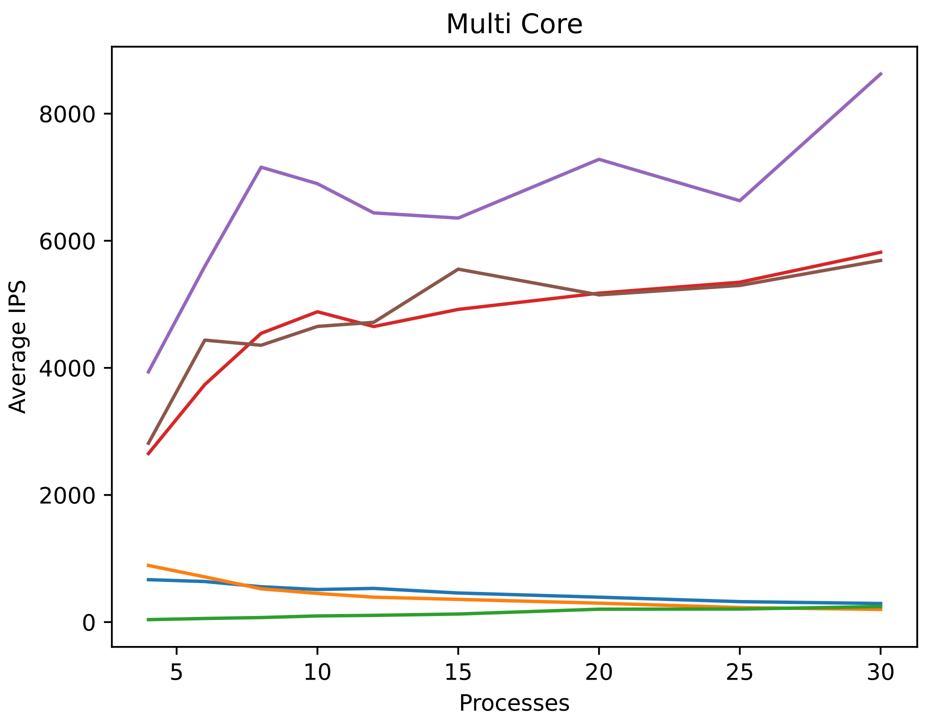

Figure 16: Multi core comparison

7 Conclusion

7.1 Summary

We first gave a small analysis of our problem and our simplifications. Then we explained our 3 different versions for the collision handling, namely the naive, static tree, and dynamic tree implementation. We explained the advantages and disadvantages of each version and provided an insight into their performances. We looked at the implementation in the single process as well as in the multi process case in addition to a variable circle amount to get a better understanding of the scaling of each approach. Furthermore we looked at the workload distribution to identify flaws and possible improvements. The following visual 17 shows our final result with 1 million circles. The circles are barely visible at such high numbers and the red lines represent the borders of the processes.

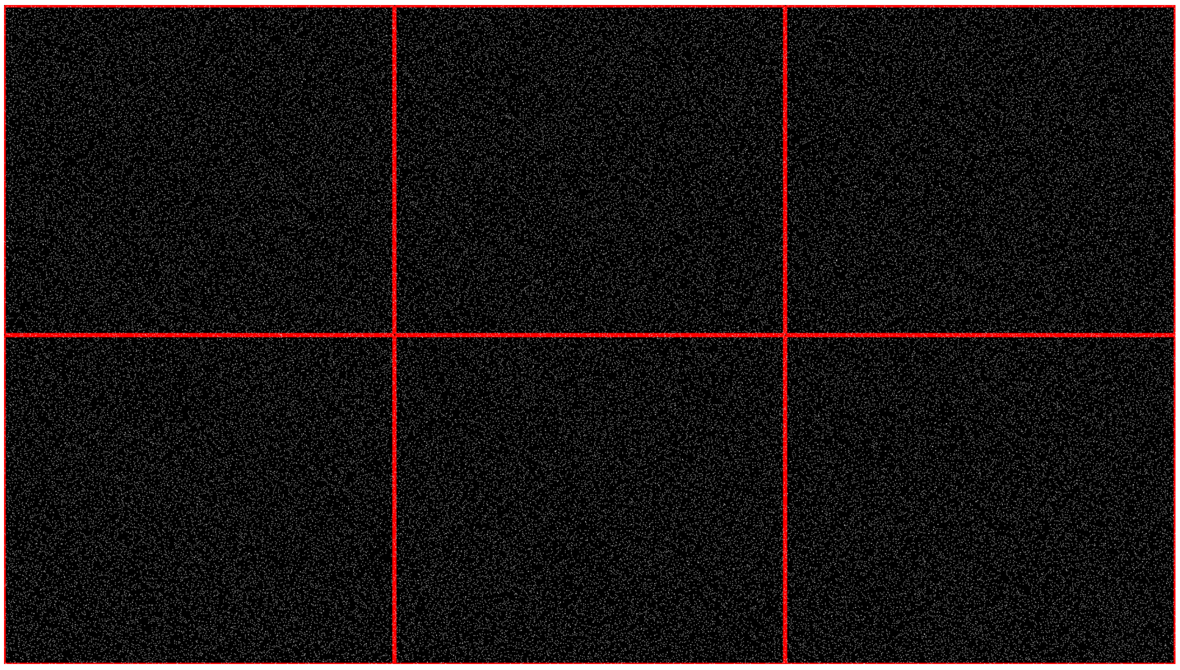

Figure 17: Final result

7.2 Outlook

As mentioned earlier, some parts of the code could be improved or get some more attention by further analyzing the reachable IPS. Examples are the order of the collision and split operations inside the update function of the dynamic tree as well as a deeper dive into the hash-set solution, since it was expected to show more noticeable improvements.
Splitting cells in four equal parts is not always the best solution of performing space partitioning. It might be more efficient to split cells at high concentrated circle locations inside it. That way there are much less split operations needed, since it always divides the cells space based on its circles distribution.
All solutions can be coded into 3-dimensional versions by changing the tree from having 4 to 8 sub-cells and the formulas of the circle collision. In addition, the visuals need to be implemented as well.5