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 03 – KDTree

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 : k-d Tree and Nearest Neighbor Search

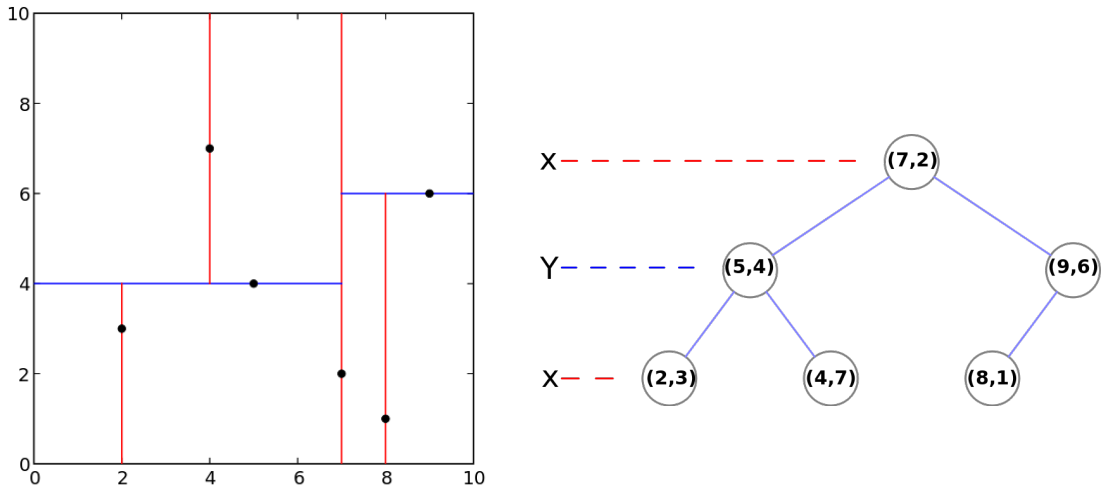
Nearest neighbor search is a crucial algorithm that is applied in many different fields like pattern recognition, computer vision, data compression, DNA sequencing, plagiarism detection, and an optimized version of the n-body problem among others.[1]. Given a point $\mathbf{x}^{(q)} \in \mathbb{R}^k$ in a k-dimensional space and a set of further points $\mathcal{V}$ occupying this space, this problem aims at finding the point $\mathbf{x}^* \in \mathcal{V}$ that minimizes the distance to the query point $\mathbf{x}^{(q)}$ (i.e. nearest neighbor) or, formally, with the Euclidean distance

$$\mathbf{x}^* = \arg\max_{x^{(j)} \in \mathcal{V}} ||\mathbf{x}^{(j)} - \mathbf{x}^{(q)}||_2. \tag{1}$$

In a naive implementation, this can be achieved by calculating the distance $d(\mathbf{x}^{(i)}, \mathbf{x}^{(q)})$ for every point and for every query in case of multiple queries. The runtime of this is $\mathcal{O}(n)$ with $n = |\mathcal{V}|$. With m query points, finding the closest point to every query point becomes $\mathcal{O}(n \cdot m)$. For many application, faster approaches are desired.

k-d Tree

A possible solution can be found in k-d trees. These are binary trees whose nodes correspond to k-dimensional points. Each point splits the input space through a hyperplane along a certain dimension d of the k-dimensional domain into two half-spaces. All points on the left of this hyperplane, will be assigned to the left child node. All points on the right to the right child node accordingly. A visualization of this simple idea is given in Figure 1 with some pseudo-code[2]given in Listing 1. Based on one root node, the input domain is subsequently split into smaller half-spaces that partition the space until all points haven been visited.



(a) Partioned Space [3]

(b) Resulting k-d-Tree [4]

Figure 1: 2-d-Tree

[1] https://en.wikipedia.org/wiki/Nearest_neighbor_search
[2] https://en.wikipedia.org/wiki/K-d_tree
[4] https://en.wikipedia.org/wiki/File:Kdtree_2d.svg
[4] https://en.wikipedia.org/wiki/File:Tree_0001.svg

```
function kdtree (list of points pointList, int depth) {
  // Select axis based on depth so that axis cycles through all valid
     values
  var int axis := depth mod k;

  // Sort point list and choose median as pivot element
  select median by axis from pointList;

  // Create node and construct subtree
  node.location := median;
  node.leftChild := kdtree(points in pointList before median, depth+1);
  node.rightChild := kdtree(points in pointList after median, depth+1);
  return node;
}
```

Listing 1: Pseudo-Code of creating a k-d tree.

Nearest Neighbor Search

Once the k-d tree has been created, you can find the nearest neighbor for a query point in $\mathcal{O}(\log(n))$ in the following way.

- starting with a root node, move down the tree recursively like you would if you inserted the node in the k-d tree (left if you are on the one side of the splitting plane, right otherwise)
- once you reach a leaf node, calculate the distance between the node point and the query and save the node point as "current best"
- unwind the recursion of the tree and subsequently refine the estimate of the "current best"
- when you arrive at the root node again, the search is complete

Implementation : Short Documentation

- You will be given a straight-forward sequential implementation of the algorithm described above in *"kdtree_sequential.cpp"*.
- For your convenience, you will find a *DEBUG* flag at the beginning of the file. If this flag is set, you will get a runtime measurement which acts as a proxy to the measurement on the submission server on your local machine.
- You can further modify the computed problem through command line arguments with the following signature: *"kdtree_sequential SEED DIM_POINTS NUM_POINTS"*.
- For evaluation on the submission server, please ensure that *DEBUG* is set to 0 or that the corresponding lines of codes are removed. In this case, your code will be evaluated on $500,000$ points of dimension 128 with a prompt specifying the seed for the random initialization (similar to the homework assignments). Note that if you also want to use this case on your local machine, you don't have to pass command line arguments to the program.
- The correctness of your code is evaluated by comparing the closest neighbors found by your algorithm for 10 query points to the results obtained from a reference solution.
- Utility code is provided in the files *"Node.cpp"*, *"Node.hpp"*, *"Utility.cpp"*, and *"Utility.hpp"*. You are not expected to change code in those files. However, feel free to do so.

Tasks

1. **General:**

   You should expect a speedup of (at least) 3 to 4 on a 6-core machine for parallel implementations.

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

   Parallelize the computations using OMP. Parallelize as much as possible as long as it does not harm performance. You do not have to parallelize the generation of the random problem. Explain which parts of the code you parallelize and why it might be better than other options. Show your work.

3. **MPI:** *Due 22.06.2021 23:59, Submission file "kdtree_mpi.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. Explain your problem decomposition, which parts of the code you parallelize and why it might be better than other options. Show your work.

   Remark: Your MPI solution should be agnostic to the number of MPI processes used when executing the code. In other words the solution should be executable by any number of MPI processes.

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

   Assume (hypothetically) that you have to run your code on a cluster with multiple compute nodes without shared memory access but several cores per compute node with these cores sharing memory access locally. How can you leverage this architecture in a hybrid approach using MPI and OMP? Modify the code from the MPI assignment accordingly. Explain which parts of the code you parallelize and why it might be better than other options. Show your work.

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

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