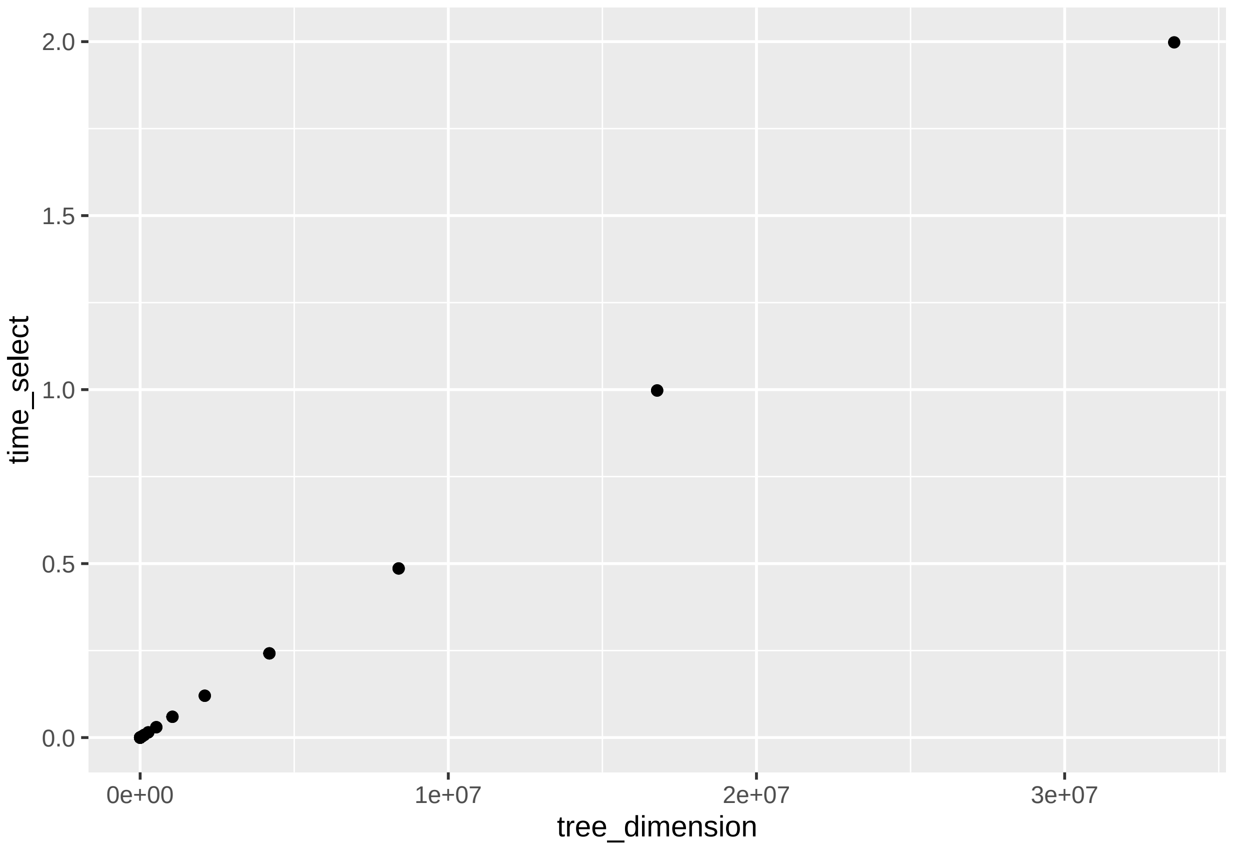
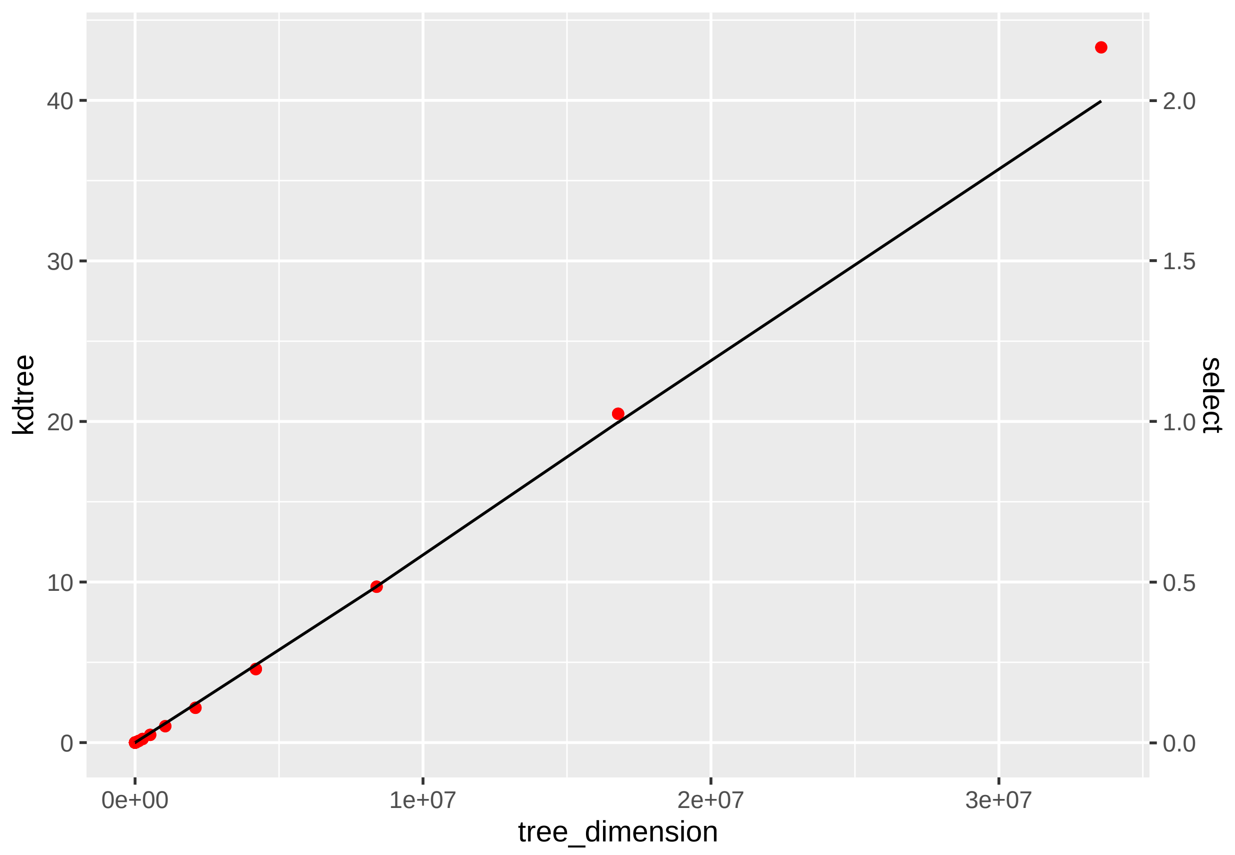
Foundations of High Performance Computing

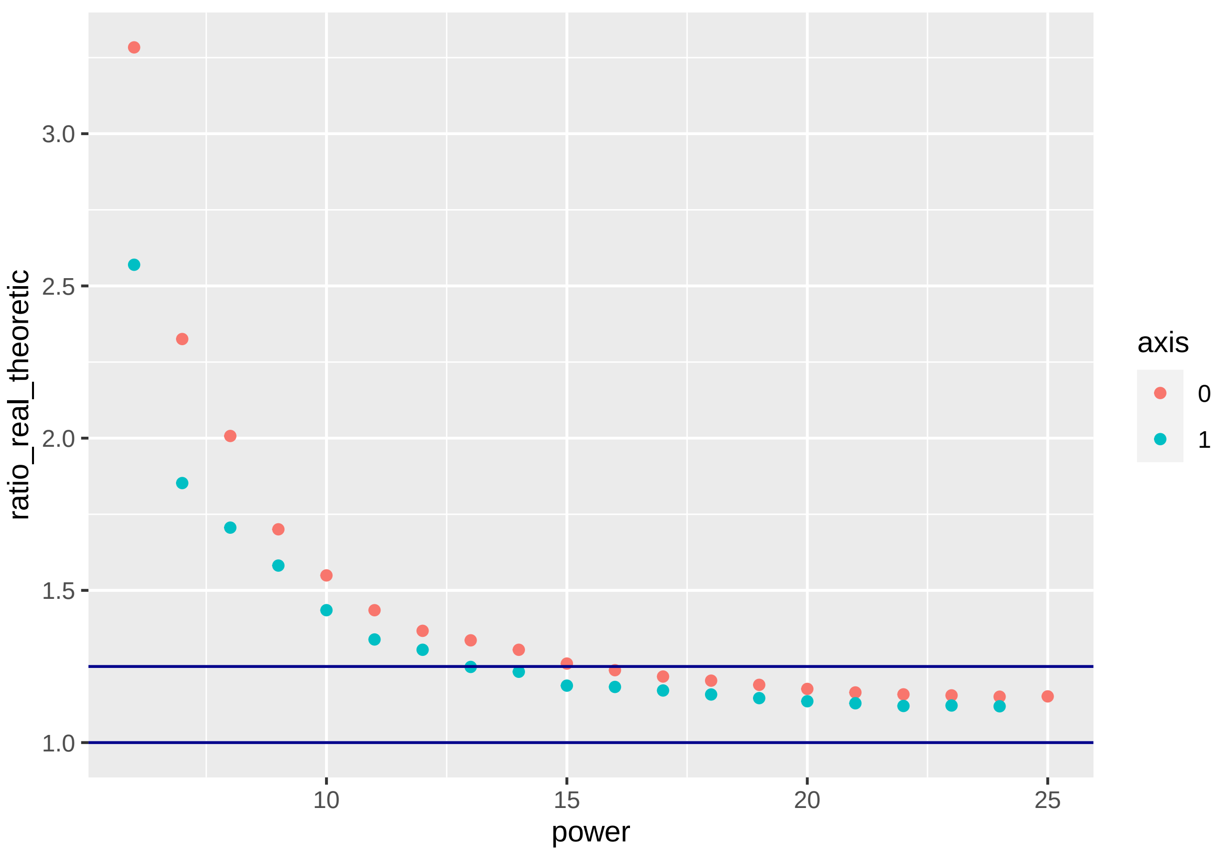
Assignment II

Lorenzo Trubian [mat. SM3500519]

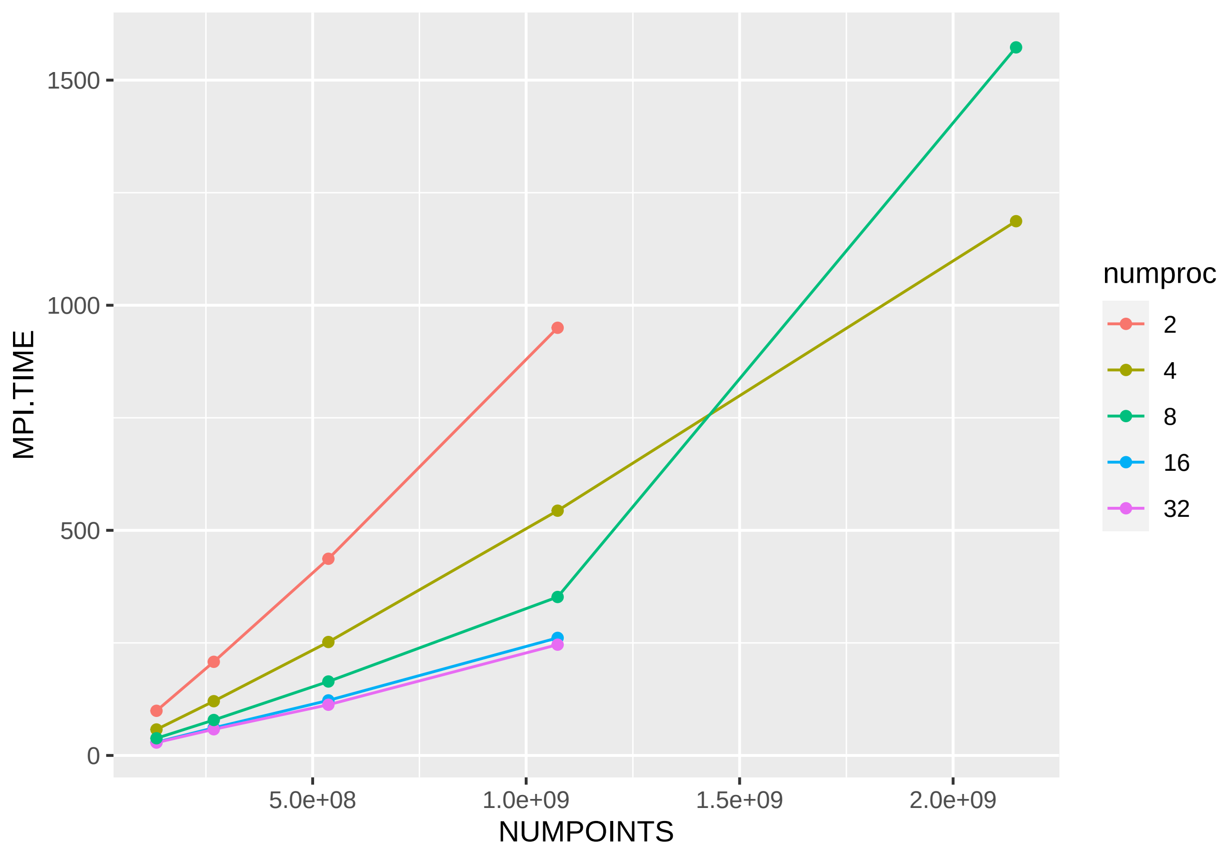
27/02/2022



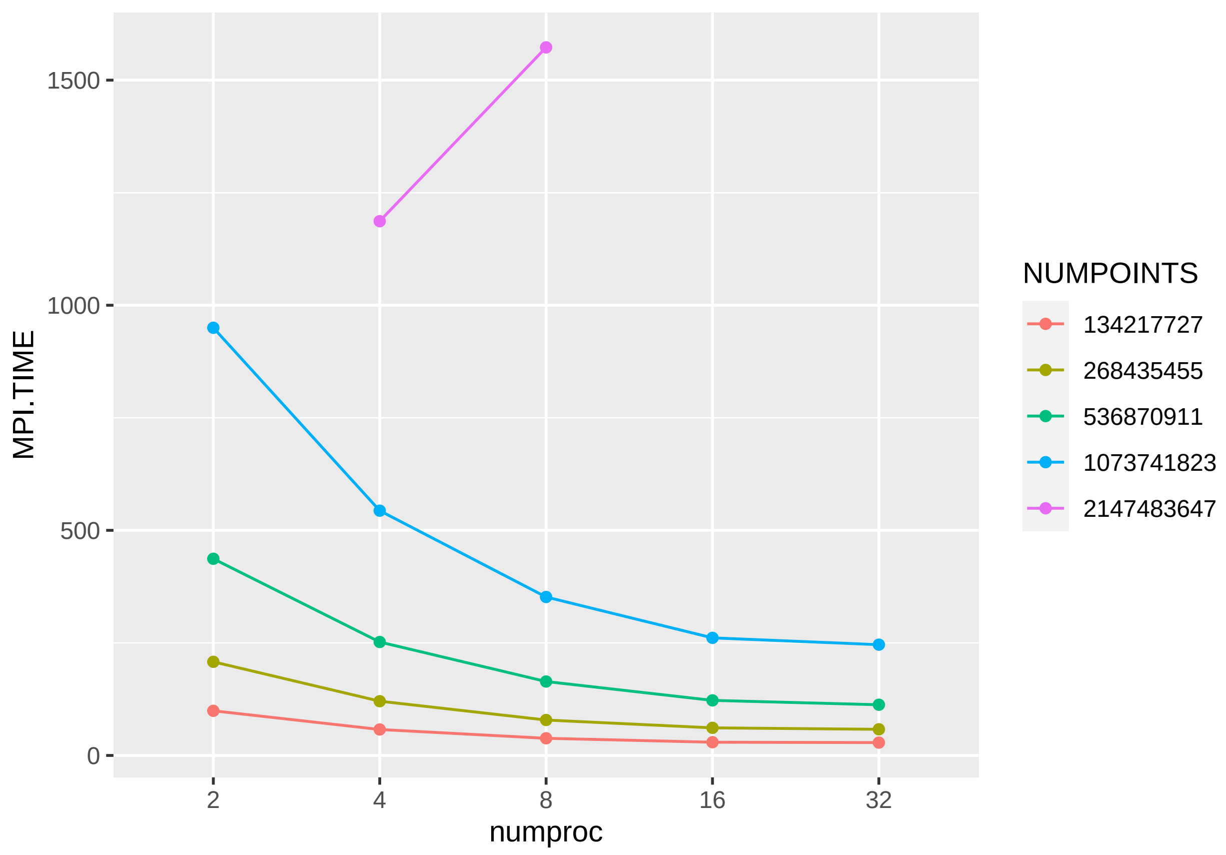

kdtree
select
tree_dimension
0
10
20
30
40
0.0
0.5
1.0
1.5
2.0
0e+00
1e+07
2e+07
3e+07



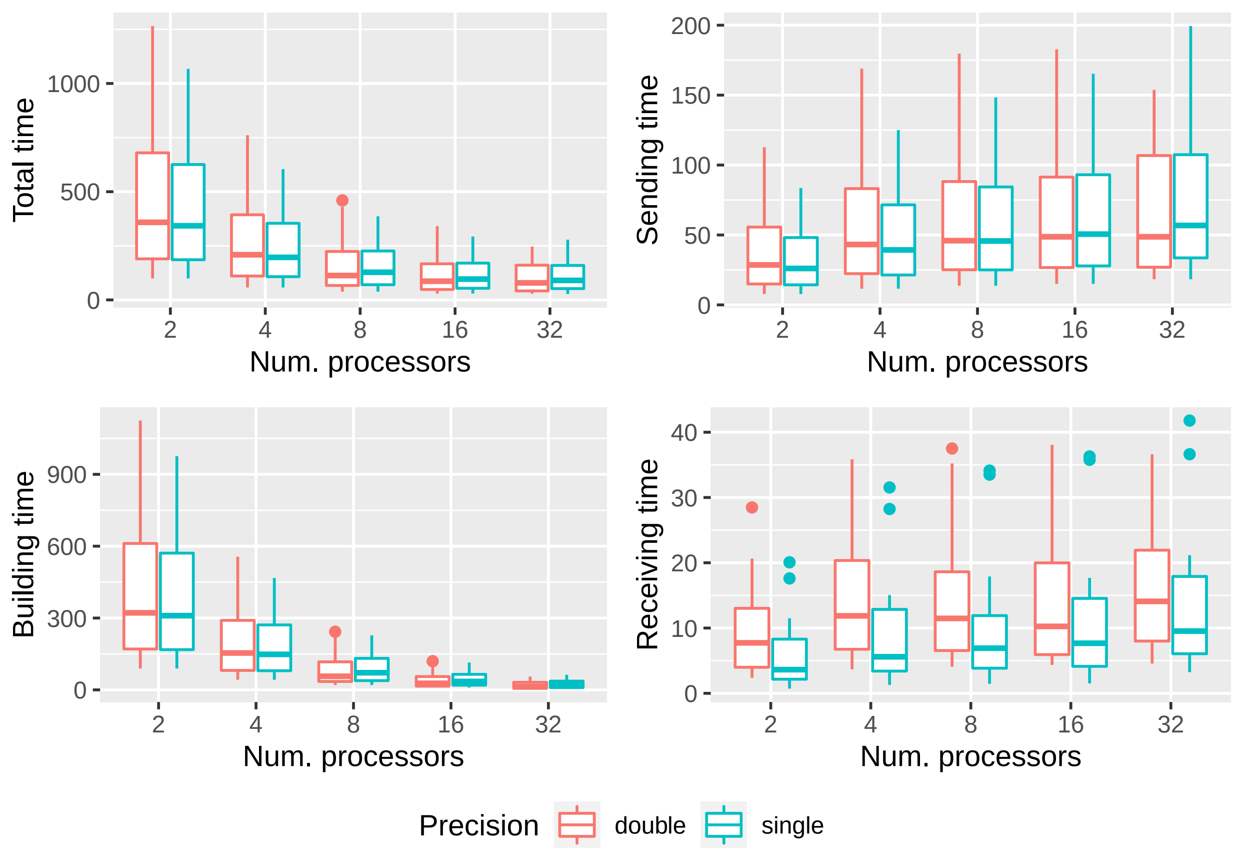
ratio_real_theoretic
3.0
2.5
2.0
1.5
1.0
10
15
20
25
power
axis
0
1



MPI.TIME
1500
1000
500
0
5.0e+08
1.0e+09
1.5e+09
2.0e+09
NUMPOINTS
numproc
2
4
8
16
32



1500
1000
500
0
MPI.TIME
2
4
8
16
32
numproc
NUMPOINTS
134217727
268435455
536870911
1073741823
2147483647



Introduction

The aim of this work was to analyze the performance improvement of building-kdtree algorithm through a parallelization. In particular, we try to investigate the difference using the distributed or shared memory approach, using MPI library in the first case and OpenMP interface in the second. Before continuing, we recall briefly the type of structure that we want to obtain. Given a d-dimensional set of points, a kdtree is a binary and balanced tree where each node contains a point which is the median (according one chosen dimension) of all the points in the sub-tree having the node as the root.

Algorithm

In order to construct the kdtree, an algorithm is needed to find the median of a list of values. In this work it is used the *median of medians* algorithm as it is described here or in the *Introduction to Algorithms* [1] to find the median of a given set of kpoint along a fixed axis. The worst-case running time of this algorithm, which will be called **select**, is linear.

We can describe the building-kdtree algorithm with the following recursive procedure:

1. given a list of kpoint and the split axis, find the median;
2. make a partition of the list around the median;
3. save the median and update the split axis;

[1]Section 9.3 of *Introduction to Algorithms* Third Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.

4. call recursively the procedure on the two sub-lists.

The **select** algorithm actually solves both the first and the second point. The described sequence of steps allow a clear parallelization: whenever a compute unit reaches the 4th step, the recursive call could easily be assigned to another compute unit. This is what is implemented, with some slight differences, both in the shared and distributed memory approach.

Implementation

Both with MPI library and OpenMP interface, the algorithm passes through two main phases:

- during the first phase the code continues to assign the two sub-list created with the 4th step to different compute unit;
- during the second phase, however, the code become completely serial and the jobs on the two sub-lists are completed by the same compute unit that creates them.

The change from the first to the second phase happens when a certain *depth* is reached. The main difference between the two implementations is the choice of the *depth*.

MPI, distributed memory

Assuming to have a power of two compute units, in order to reduce the messages sent and received by the processors, the depth is chosen to be $\log_2(\text{num-proc})$. For each iteration of the first phase of the algorithm, each processor send to another one of the two resulting lists, keeping the other for itself. In this way, at each iteration the total amount of sent data is about the half of the original set of kpoint.

OpenMP, shared memory

Due to the fact that the assignment of the work is much easier than the previous, there is no a priori fixed depth for the first phase. Hence, the code tests different depths, from 0 up to 15, using a fixed number of kpoint ($2^{24} - 1$) and chooses the three best depths with that number of threads, using each of them for the compute with the real set of kpoints.