Report of the second assigment of HPC 2021-2022

Guglielmo Padula

January 23, 2022

The main script erases previously created files and recreates them after the calculations. The folder contains: testbinding.cc (program to test the CPU binding), kdtree.cc (the code), kdtree.sh (a bash script for doings all the necessary tests), some csv files with the test results and some png files, which are mainly graphs.

1 Introduction

The problem consists in creating an algorithm that builds a Kd-tree from a vector of data.

2 The abstract Kd-tree algorithm

The abstract algorithm is recursive:

- The median of the vector is found, along the specified sorted dimension.
- The vector is partitioned with the median as pivot (right median in case of a odd size) along the specified sorted dimension.
- The median becomes the root node of the subtree.
- The algorithm is applied recursively, with a shift of sorted dimension by one, to the right partition of the vector and to the left partition of the vector.
- The result of the right sub call becomes the right subtree of the root node, the result of the left sub call becomes the left subtree of the root node.

3 The implementation

The algorithm is an openmp-mpi hybrid (every MPI thread spawn an equal number of OMP threads). It accepts an input two numbers: the first indicates the size of the data, the second indicates the sorting algorithm to use (0 for serial mergesort, 1 for parallel-OMP mergesort and 2 for a serial quick select; all the algorithms have been written by myself). The default values are 100000000 and 2 respectively. The code is written in C++14. Thanks to templates multiple Ks can be tested (the code howewer must be recompiled when we change K, in the provided code is 2). Before the tree building the data is created using random numbers between 0 and 1. The kdtree creation can be divided in 4 phases:

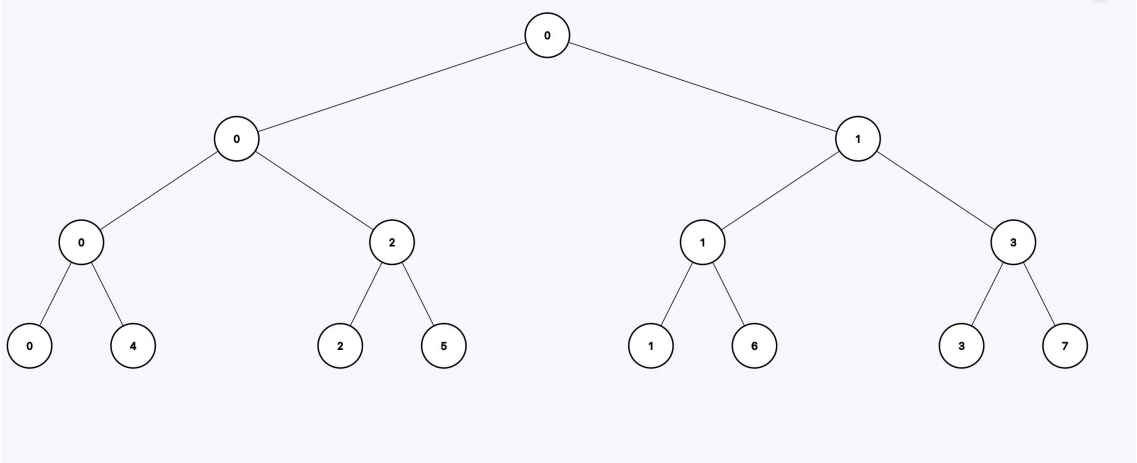
1. MPI Thread 0 sorts the vector using the selected sorting algorithm. It sends the right part of the vector to MPI Thread 1 if it exists, otherwise it procedees direcly to step 3).
2. (iterate) Every MPI Thread sorts its part of the vector using the selected sorting algorithm. Then it sends the right part of its vector to his MPI Thread son, if it exists, otherwise it proceeds to step 3).
3. All MPI Threads calculate the sub kd-tree (of the sub vectors they have received) using OMP tasks (the left and the right calculation of the tree are tasked). In this part there are no MPI communications. There is a task conditional branch to avoid task overhead.
4. (iterate) All MPI Threads serialize the sub kd-tree in a vector, send this vector to the father, the father deserializes it and connects it to left right part of the tree which was computed by itself.

3.0.1 Example with a vector of size 15 and 4 MPI threads and 1 OMP thread

Vectors are indicated using the index (start from 1).

1. MPI Thread 0 receives the data.
2. MPI Thread 0 sorts 1:15. Keeps 1:8 for itself and sends 9:15 to rank 1.
3. MPI Thread 0 sorts 1:7, keeps 1:4 for itself and sends 5:7 to rank 2. MPI Thread 1 sorts 9:15, keeps 9:12 for itself and sends 13:15 to MPI Thread 3.
4. MPI Thread 0 computes the tree of 1:3 and attaches it to left of 4, MPI Thread 1 computes the tree of 9:11 and attaches it at the left of 12, MPI Thread 2 computes the tree of 5:7, MPI Thread 3 computes the tree of 13:15.
5. MPI Thread 3 serializes the tree of 13:15 and sends it to MPI Thread 1. MPI Thread 1 deserializes it an connects it at the right of 12. MPI Thread 2 serializes the tree of 5:7 and sends it to MPI Thread 0. MPI Thread 0 deserializes it an connects it at the right of 4.
6. MPI Thread 0 connects the tree of 1:7 to the left of 8. MPI Thread 1 serializes the tree of 9:15, and sends it to MPI Thread 0 which deserializes it and connects it to the right of 8.
7. MPI Thread 0 returns the tree.

Here is the scheme father-son of 8 MPI Threads.



4 The performance model

The performance model has some assumptions: there is no OMP and MPI overhead, and all processes have the same workload (this last thing is obviously wrong for the MPI part, but all models are wrong). Another assumption is that the number of MPI processes is a power of 2 and n (the size of the data) is a power of 2 minus 1 (this assumption is only needed for calculations) . Let $S(n)$ be the time of sorting the vector. We have $T(n, omp, mpi) = S(n) + \sum_{i=1}^{log2(mpi)} S(\frac{n}{2^i}) + \frac{\sum_{i=log2(mpi)}^{log2(n+1)} 2^i S(\frac{n}{2^i})}{omp}$. Let's assume that we are using quickselect (in the appendix A we will show that it is the fastest). We then have $S(n) \approx k * n$, so after substitution we obtain $T(n) = k * (n + \sum_{i=1}^{log2(mpi)} \frac{n}{2^i} + \frac{\sum_{i=log2(mpi)}^{log2(n+1)} 2^i \frac{n}{2^i}}{omp})$. After doing some simple calculations we arrive at $T(n) = k * n * (1 + \frac{1}{mpi} + \frac{log2(n+1) - log2(mpi)}{omp})$.
Consequences:

- we expect a good strong scaling with OMP, altough not perfect, and a bad strong scaling with MPI.
- the algorithm will badly weak scale, with OMP because of the $log2(n)$ term, and with MPI becuse of the n term.

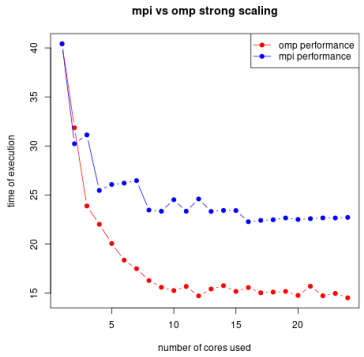
Now we will present some graphs about the real behaviour of the algorithm, along with the expected behaviour using the performance model. Note that the performance model overestimates the impact of MPI because of the simmetry assumption, and the no overhead assumptions, while the no overhead assumption for OMP is reasonable thanks to the task conditional branch. The algorithm has been compiled with the openmpi compilier using -O3 optimization and the -std=c++14 flag.

5 Simulations

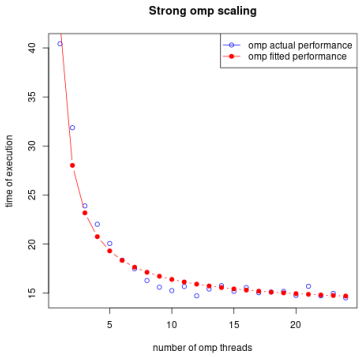
All the simulations were done with a 2D tree, and with the quickselect.

5.1 Strong scaling

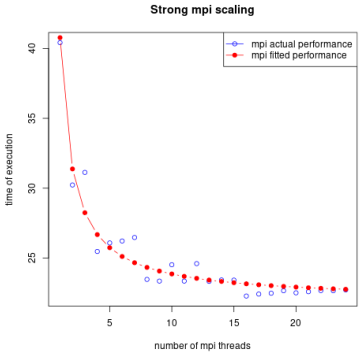
This simulation have were with n=100000000.



As the graph shows, actual performance of OMP is much better then the MPI one, because OMP doesn't need to serialize and deserialize and also has equal workload.



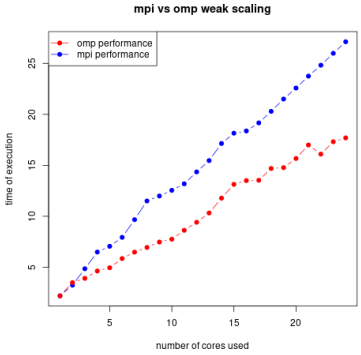
The performance model fits the the data well enough.



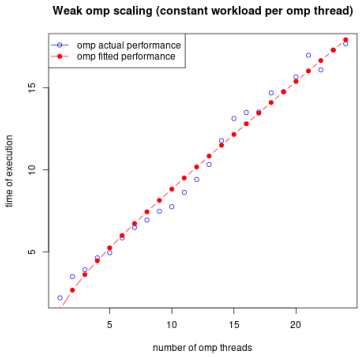
The performance model fits the the data well, but it is worse than the OMP case because of asymmetric workload and overhead.

5.1.1 Weak scaling

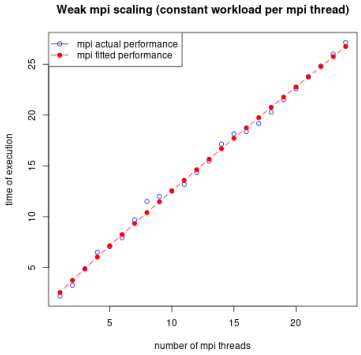
Weak scaling was done with costant workload per proc of n=500000.



The actual performance of OMP is again much better then the MPI one, because OMP doesn't need to serialize and deserialize and also has equal workload.



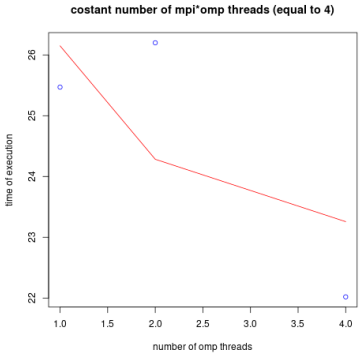
The performance model fits the the data well enough.

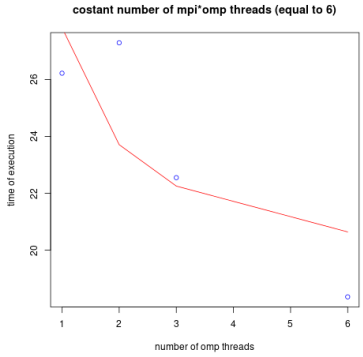


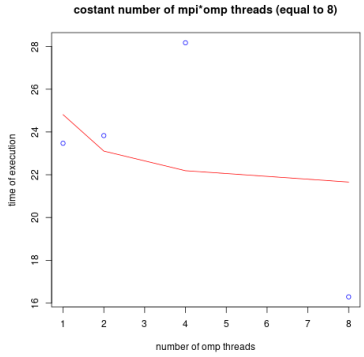
The performance model fits the the data well enough.

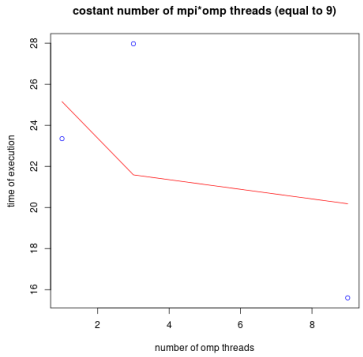
5.2 Omp vs MPI

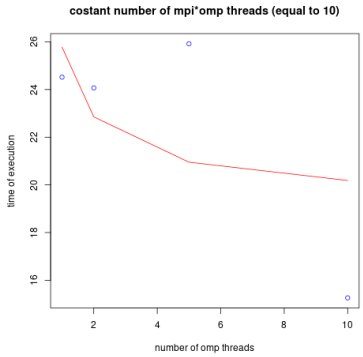
In this subsection we evaluate what is the best balance between OMP and MPI, keeping fixed the number of processors used.

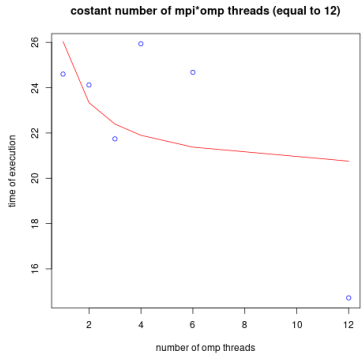


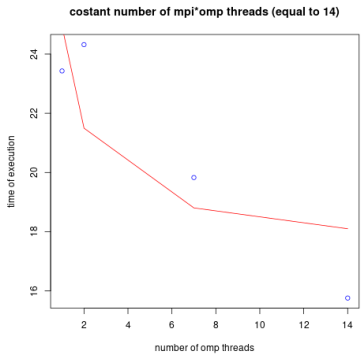


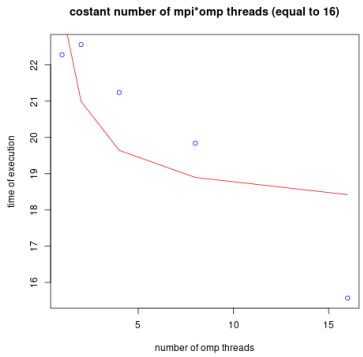


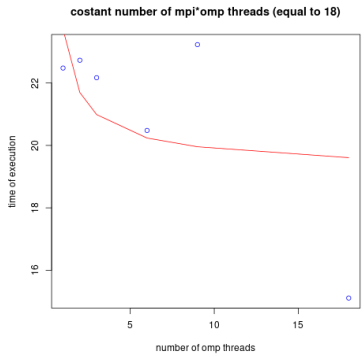

costant number of mpi*omp threads (equal to 9)
time of execution
number of omp threads

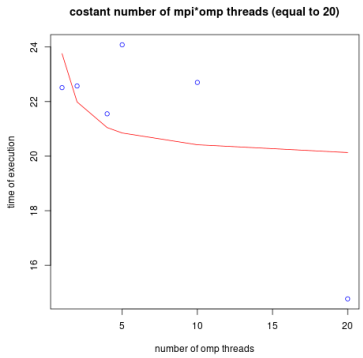

costant number of mpi*omp threads (equal to 10)
time of execution
number of omp threads

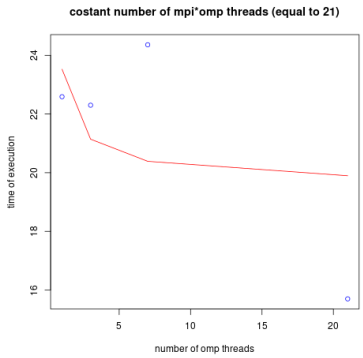

costant number of mpi*omp threads (equal to 12)
time of execution
number of omp threads

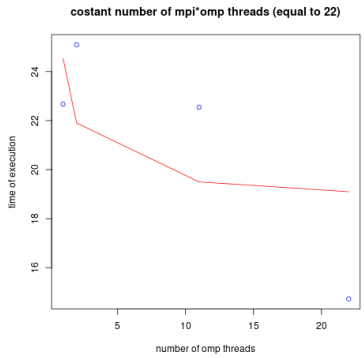

costant number of mpi*omp threads (equal to 14)
time of execution
number of omp threads

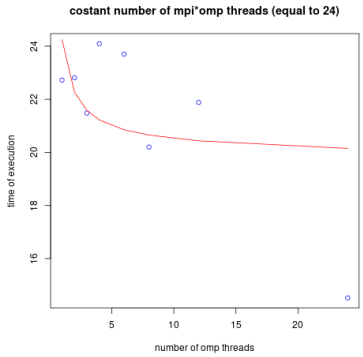

costant number of mpi*omp threads (equal to 16)
time of execution
number of omp threads


costant number of mpi*omp threads (equal to 18)
time of execution
number of omp threads


costant number of mpi*omp threads (equal to 20)
time of execution
number of omp threads


costant number of mpi*omp threads (equal to 21)
time of execution
number of omp threads


costant number of mpi*omp threads (equal to 22)
time of execution
number of omp threads



In all the cases, we see that the performance model overestimates actual performance with low OMP threads and underestimates it with high OMP threads. This happens because of the asymmetric MPI workload, which was not incorporated in the performance model.

6 Pitfall and proposals

The main pitfall of this algorithm is the asymmetric MPI workload. Also I am not entirely satisfacted of the MPI strong scaling, which is only log, which causes the algorithm not to weak scale. All my calculations are on a single node so the MPI overhead is not that big.
A possible extension of the work is to study the behaviour of the algorithm with multiple nodes (I expect a big drop in efficiency). In this algorithm the initial data is only in the MPI Thread 0, another possible extension is to try to extend the algorithm to accept initial data from different MPI Threads. Another proposal is to try to balance the MPI part of the algorithm.

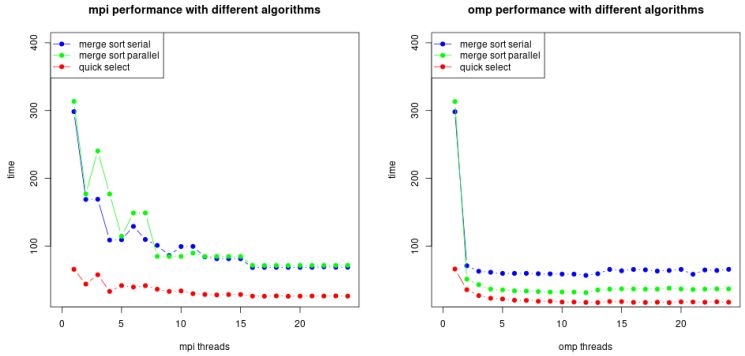
A Appendix

A.1 The testbindings cpp file

The testbindings is a program, implemented by myself, that shows to which CPU an OMP thread of a MPI thread is associated.

A.2 Comparison of algorithms

These two figures show the strong scalability differences of different algorithms:



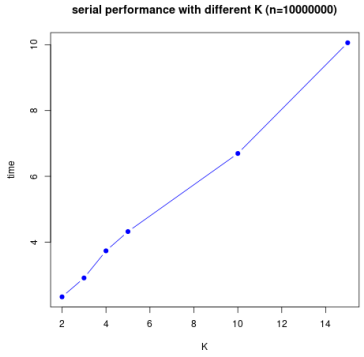
As the graph shows, the quickselect is faster than both the merge sort serial and the merge sort parallel. Parallel mergesort is OMP-only, so in the only MPI scalability study its performance are similar to the serial mergesort.

A.3 The .dot dump

In the code there are two functions exportTree and exportTreeC (the C version), both implemented by myself, which export the tree in a .dot file.

A.4 Comparison of different Ks

As already told in the introduction, we can change the number of elements (K) in every node of the tree. I studied the scaling with increasing K.



As the graph shows, time grows linearly with K.