Implementation of a k-d tree using MPI and OpenMP

Francesco Andreuzzi

February 25, 2022

Contents

1 Introduction

k-d tree is an established data structure which enables the use of a plethora of efficient algorithms of huge practical interest, for instance *search algorithms* and *nearest neighbor queries* [1]. Another important aspect is the existence of efficient utility algorithms which make quite efficient the update of the data structure in response to changes in the dataset (deletion, insertion, . . .).

In this brief report we present and analyze a parallel implementations of k-d tree which uses two standard frameworks for parallel programming, namely OpenMP (shared memory) and MPI (distributed memory). Our implementation uses the same approach for parallelization in both cases, even though some compiler-level flags can be used to activate different code regions whose goal is to fix performance bottlenecks peculiar of a framework (for instance *false-sharing* in OpenMP). We talk briefly about this point below.

First things first: we briefly discuss the data structure and the invariants to be maintained. Afterwards we present the very simple parallel algorithm we developed, and spend a few words on the implementation. We then procede with some considerations on our expectations for the performance, and verify our hypothesis against the real data we extracted from a set of run of the code.

1.1 k-d tree

Given a set of k-dimensional points $P = \{p^1, \dots, p^n\}$ such that $p^i \in \mathbb{R}^k$ we pick a recursive definition of a k-d tree [4]. The node of the tree associated with the set P is given by:

$$\text{Node}(P) = \begin{cases} \text{null} & \text{if } P = \emptyset \\ \{p_m, \text{Node}(P_1), \text{Node}(P_2)\} & \text{otherwise} \end{cases} \tag{1}$$

where p_m is the median point of P against some axis i which is chosen via an unspecified criteria that we are going to formalize in some lines. $P_1, P_2 \subset P$ are defined as follows:

$$P_1 = \{p \in P \mid p < p_m\} \qquad P_2 = \{p \in P \mid p > p_m\}$$

i.e. they are the subsets of points of P which “fall” respectively before and after p_m along the i-th axis. For simplicity we assumed that P does not contain repeated values, however the definition is easily generalized. $\text{Node}(P_1), \text{Node}(P_2)$ in (1) are respectively the left and right branch which originate from $\text{Node}(P)$.

The axis i is chosen in such a way that the spread of the points in P is maximum along i. However, since we assumed that our points are distributed somewhat uniformly in the space $\mathbb{R}^k$, we are going to use a very simple function to determine the axis i used to ”split” the tree:

$$i(\ell) = \ell \pmod k \tag{2}$$

where ℓ is the current level of the tree (i.e. the distance of the current node from the root).

Using a visualizer tool written in Python which we developed for the occasion, we show the progression of the construction of a k-d tree for a very simple dataset. The visualization in Figure 1 renders in a very clear way the fact that points added to the tree in the level k need to satisfy constraints given by all the median points points in the parent branches in levels $0, \dots, k-1$, against all the axes considered up to this level. Each surface is the plane containing one of the points added to the tree at some point of the algorithm, and divides the volume dedicated to the corresponding branch in two halves which shall contain all the other points (respectively) in the left and right branch. This is in a certain sense a visual transposition of (1).

However, in order to communicate the structure of the tree the visualization shown in Figure 2 may be preferred, which is easier to interpret visually but conveys a smaller amount of information about the structure we imposed on the dataset.

2 Parallel algorithm and implementation

First of all we briefly describe the parallel algorithm; then we talk a little bit about some technicalities needed to glue everything in the two frameworks taken into account; finally we procede with some words and motivations regarding how we represented the dataset in our code and which data structures we used.

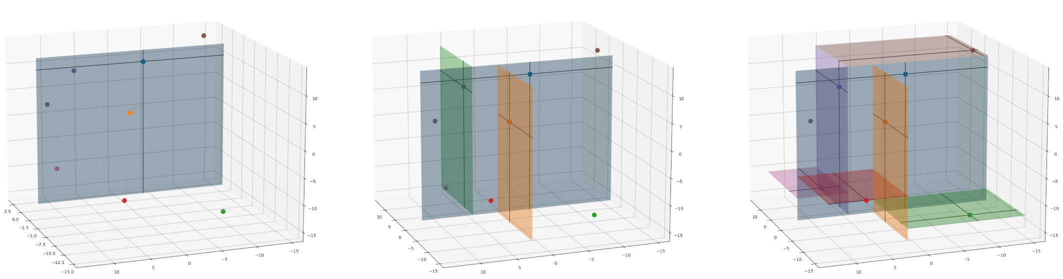
2.1 Algorithm

The following algorithm is a very simple codification of the definition proposed in Section 1:

The exact definition of "parallel worker" depends on the parallel framework used to compile and run the code:

- MPI → a parallel worker is one of the MPI processes employed to run the code;
- OpenMP → a parallel worker is an OMP thread.

It's worth noting that these kind of resources are not unlimited: in case no parallel workers are available we procede in a serial way, with two recursive calls

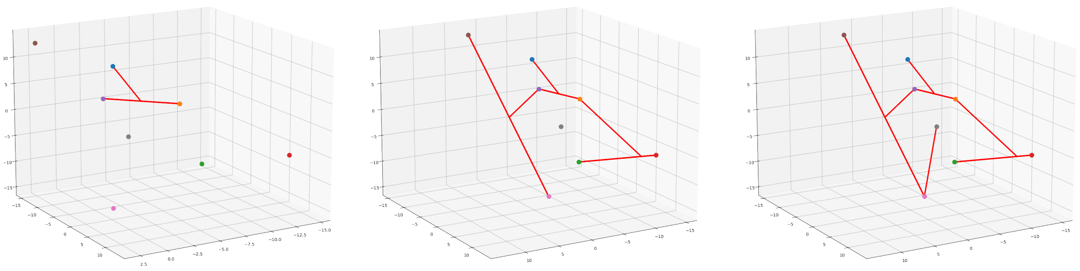


(a) Depth 0

(b) Depth 1

(c) Depth 2

Figure 1: A k-d tree (surfaces representation).



(a) Depth 0

(b) Depth 1

(c) Depth 2

Figure 2: The same k-d tree shown in Figure 1 (branches representation).

Algorithm 1: Parallel k-d tree growth

```
  Data: $P = \{p_1, \dots, p_n\}, \ell$
1 $i \leftarrow i(\ell)$ like in (2);
2 Find the median point $p_m$ against the axis $i$;
3 Assign the right branch (i.e. $P_2$) to another parallel worker (if
    available);
4 $P \leftarrow P_1, \ell \leftarrow \ell + 1$;
5 Go back to Line 1;
```

to the function which we use to generate the k-d tree for both the left and right branch. This kind of setting requires a more refined approach if MPI is used (OpenMP is quite convenient in this case), and we will now spend some words on this.

2.2 The problem of the "next" rank

Whenever we want to communicate something from an MPI process to another, we need to know precisely what we want to send and what's the process (i.e. the rank) we want to communicate with. Since in principle we are going to have multiple processes which want to communicate with multiple children processes, we needed some way to determine without conflicts the rank of an idle process, namely a process which is able to take on a branch of the two branches generated by the current process. Two ideas emerged, which we present briefly. The first one is a *master-slave* approach, i.e. to dedicate a process to the sole work of load-balancing the computation between all the available parallel workers. A process willing to delegate its right branch to another MPI process would have needed to:

1. Establish a communication with the *master*;

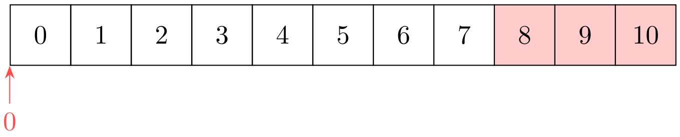
2. Receive the rank of a free MPI process;

3. Establish a communication with the chosen MPI process;

4. Send P_2.

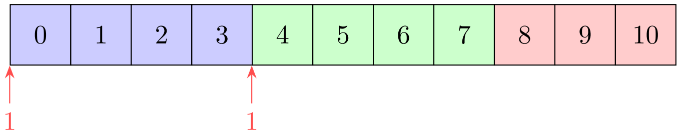
We had the feeling that this kind of setting would have incremented by 2 units the number of communications needed to delegate a branch to another process, and would have made an MPI process totally unusable to the actual purpose of growing the k-d tree; however, the code would have been most likely very elegant and understandable. Nonetheless we decided to pursue another approach: an “hardcoded” protocol. A parent process knows right off the bat the rank of a child process which is idle and willing to process a branch, and only that parent process (due to the protocol) is able to communicate with that process. The rank is computed via some very simple arithmetic operations. The code becomes less readable, but no-one is going to notice since we decided to hide this kind of complexity behind a function.

We briefly describe the protocol we employed in our code with a practical example, let’s say sat we have 11 MPI processes (0 being the main process), which we represent as follows:

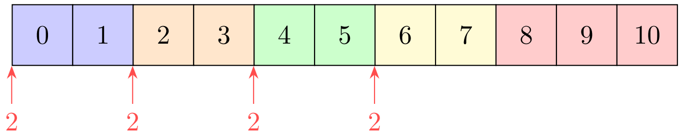


The red arrow pointing to $\boxed{0}$ means that at this point only $\boxed{0}$ is running, while the other MPI processes are idle, waiting for some branch to be assigned to them. The number below the arrow refers to the current depth of the three processed by the process that the arrow is pointing to. Also, at this point the whole dataset is assigned to $\boxed{0}$. This is equivalent to “Depth 0” in Figure 1 and 2. Note that $\boxed{8}$, $\boxed{9}$, $\boxed{10}$ are highlighted in red. We call them `surplus_workers` for reasons which will be clarified in some lines.

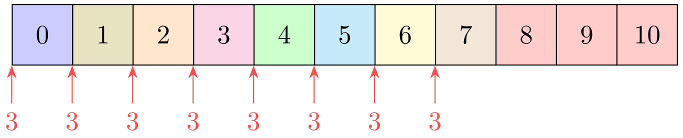
At this point $\boxed{0}$ executes Algorithm 1, which means that the left branch generated (i.e. P_1) is recursively assigned to $\boxed{0}$. How do we choose the process that receives the right branch (i.e. P_2)? We consider the set of available processes (i.e. those which are not highlighted), we split it in two equal halves and we assign P_2 to the first process of the right branch (i.e. $\boxed{4}$):



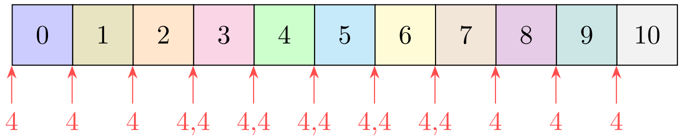
Now we have two processes working in parallel, processing two different parts of the dataset. We keep splitting the work in the exact same way:



At this point we have four processes working in parallel ($\boxed{0}$, $\boxed{2}$, $\boxed{4}$, $\boxed{6}$) on the second level of the tree. Fast forward, we reach the following situation:



We now have three idle proccesses ($\boxed{8}$, $\boxed{9}$, $\boxed{10}$), which are clearly not enough to parallelize entirely a new level. Even though the performance gain we get by using also these `surplus_workers` is not so influential (we have to wait the slowest parts of the tree to be completed) we decided to assign them left-to-right to the other processes. In this case only $\boxed{0}$, $\boxed{1}$ and $\boxed{2}$ would get a `surplus_workers` (respectively $\boxed{8}$, $\boxed{9}$ and $\boxed{10}$). Therefore the final situation is:



As you can see $\boxed{3}$, $\boxed{4}$, $\boxed{5}$, $\boxed{6}$, $\boxed{7}$ receive both the branches they spawned, while the other processes receive only one. From now on we do not have other MPI processes, therefore the computation proceeds serially.

As we mentioned earlier OpenMP does not requires these technicalities since we employed OpenMP *Tasks* to parallelize the computation. Each time we need to assign a branch to another thread we just spawn a new task, and then process the remaining branch on this thread. We employed the modifier `final` to disable spawning new tasks when we know that no more threads are available since otherwise the newly spawned task would have to wait until a new thread became available; also spawning OpenMP tasks without control usually leads to worse performance due the incremented orchestration overhead [3]. Checking whether new threads are available is done almost in the same way we presented above (even though we do not need to compute the rank of the next threads). The same mechanism of `surplus_workers` is employed to overcome the issues which emerges when the number of parallel workers is not exactly a power of 2.

2.3 Merging

Another difficulty in the usage of MPI is the fact that each process that delegated one or more branches to other MPI processes needs to recover and reconstruct a k-d tree starting from the portion of tree produced by child processes. This operation is analogous to the `merge` subroutine which is used in common implementations of *Merge-Sort* [2]. Indeed in our code the function which encapsulates this operation is called `merge_kd_trees`. Note that *merging* is not required if we use OpenMP, unless we employ some particular strategy for writing into the 1D array which represents the k-d tree in memory (for instance, see Section 2.4).

Summing up, the abstract algorithm for the growth of k-d tree (for which we presented a very general pseudocode in Algorithm 1) must be extended as follows if MPI is used:

Algorithm 2: Parallel k-d tree growth (MPI)

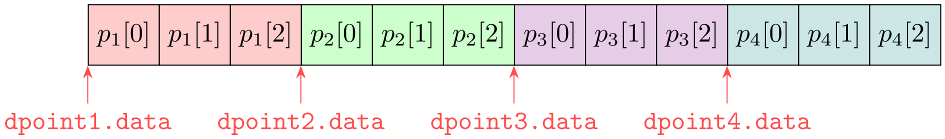
```
Data: $P = \{p_1, \dots, p_n\}, \ell$, m_rank
Run Algorithm 1;
KDT ← $k$-d tree rooted in the first fully serial level computed by m_rank;
foreach $r \in \mathcal{C}^{-1}_{m\_rank}$ do
    right_kdt ← $k$-d tree generated by process $r$;
    KDT ← merge_kd_trees(KDT, right_kdt);
end
```

As you can see we procede in a bottom-up fashion and locate the first fully serial level computed by this process (i.e. the first level of the k-d tree such that no branches have been delegated to another process). Then we procede towards the root of the k-d tree querying the processes which received some branch from `m_rank` for the result of their computation. We refer as $\mathcal{C}_{\mathtt{m\_rank}}$ to the list of child processes of `m_rank`, and $\mathcal{C}^{-1}_{\mathtt{m\_rank}}$ means that we traverse that list in reverse order (i.e. the last inserted child is the first to be queried).

2.4 Fixing *false-sharing*

Not only MPI determines more techical challenges than expected: OpenMP does too. Until now we managed to stay very high level ignoring technical details like, for instance, how the k-d tree is stored in the memory of a process. We now state that the in-memory representation is a 1D array of `std::optional<DataPoint>`. `DataPoint` is a class which contains a pointer to a 1D array of `float` or `double` values (depending on a compilation flag). This pointer points to the appropriate position inside a big 1D array of values which contains all the components of all the datapoints (note that this means that each `DataPoint` does not "own" its data, so it's not responsible of deleting). We depict this setting in an image to clarify:

dpoint1?	dpoint2?	dpoint3?	dpoint1?



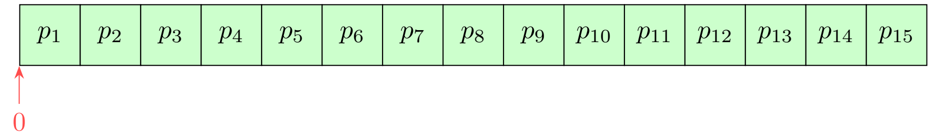
Moreover the class `DataPoint` defines come convenient methods which are used to simplify the code. We decided to use `std::optional` since in principle some values of the tree might be empty: this happens because we enforce the rule that all the nodes of the k-d tree (except the leaves) must have exactly two children. We made this decision in order to simplify the communication protocol between different MPI processes/OMP threads. However these details might be ignored since they do not determine any particular performance improvement, and are mere design choices. For this reason from now on we're going to consider the "local" k-d tree (i.e. that grown by a single parallel worker) as a 1D simple array of "points".

Inside this 1D array, 2^ℓ consecutive points represent a level of the tree, where ℓ is the depth (i.e. distance from the root) of the level. It has been experimentally observed (via the operator `sizeof`) that when the data type is `float` and each data point has three components, each item in the 1D array has size 16 bytes. We visualize this setting in the following image (each p_i in this case is an "empty spot" to be filled with a data point from the dataset):

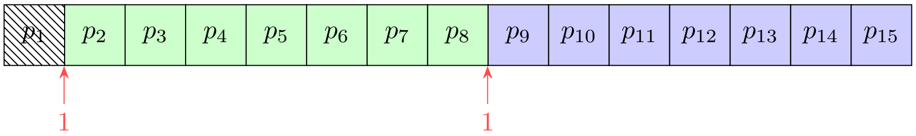
p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}	p_{14}	p_{15}
16	16	16	16	16	16	16	16	16	16	16	16	16	16	16

We now observe that the *naive* write mode is very inconvenient for OpenMP, if the 1D array which represents the k-d tree is shared among all the parallel OMP threads. "*naive* write mode" means that each thread writes all the points in their proper location inside the 1D array. This means that we could virtually observe all the OMP threads writing in the same k-d tree level at the same time, but in different cells of the array. However if we consider a cache line to be 128 bytes long, this means that we could have 8 OMP threads writing concurrently to that same memory address window, which results in very poor performance when we take into account cache coherency [3]. Note that this is definitely not a problem if we use MPI, since memory is distributed rather than shared.

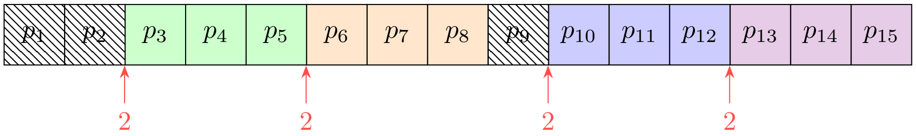
In order to fix *false-sharing* we introduced a compilation flag (`ALTERNATIVE_SERIAL_WRITE`) which allows to select a different write mode which minimzes the number of concurrent writes on addresses which might belong to the same cache line. We briefly describe this method using some images. Let's consider again the 1D array depicted above.


p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9 p_{10} p_{11} p_{12} p_{13} p_{14} p_{15}
0

The OMP thread `thread0` is assigned the dataset P and has the duty to fill the big green array above in such a way that it's easy to recover the k-d tree corresponding to P afterwards. Therefore it dermines the median point and splits P into P_1 and P_2. It also places the median point into p_1. In order to reduce the number of concurrent write on the same cache line, we divide the remaining array in two halves, and we make sure that only `thread0` (and its descendants) can write in the left part, and only `thread1` (and its descendants) can write on the right part (we assume that `thread1` takes on the task associated with right branch P_2). The situation now looks like this:


p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9 p_{10} p_{11} p_{12} p_{13} p_{14} p_{15}
1
1

We keep writing the median point on the first memory address in the subset assigned to the thread, and splitting the remaining memory between the two threads that process the left and right branches generated:


p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9 p_{10} p_{11} p_{12} p_{13} p_{14} p_{15}
2
2
2
2

This is not very meaningful when the number of points in the levels of the tree are small like in this example, but the situation changes radically as we increase the height of the k-d tree. Clearly we need to re-write the k-d tree in the expected form afterwards: this is something that may reduce, or even nullify, the performance gains which we get by removing *false-sharing*.

Another possible solution is introducing some *padding* [3], i.e. artificially augmenting the number of components for each data point in order to reduce the number of threads possibly writing on the same cache line . For instance, if we manage to increase the size of each item in the array from 16 to 64 bytes rwe would risk writing concurrently on the same cache line with two threads instead of eight.

2.5 Improving the memory access pattern

The most computationally intensive line in our code is the call to `std::nth_element` which is used to find the median element of a set of points in a branch. This function is called exactly one time per branch, and its computational complexity is $O(n)$. But rather than the computational complexity we look at the memory access pattern: a bad access pattern is capable to ruin performances potentially worse than an excessive complexity [3]. This is particularly true since the body of the function `std::nth_element` involves multiple comparisons between the data points in the branch against the chosen axis i. Since we're only interested in the i-th component for those comparisons, storing and accessing the data points in this way:

$p_1[0]$	$p_1[1]$	$p_1[2]$	$p_2[0]$	$p_2[1]$	$p_2[2]$	$p_3[0]$	$p_3[1]$	$p_3[2]$	$p_4[0]$	$p_4[1]$	$p_4[2]$

is probably suboptimal, since every time we access $p_k[i]$ we load in the cache a nearboroughood of $p_k[i]$, i.e. $p_k[i-1], p_k[i+1], \dots$. For this reason we decided to try a different arrangement in memory. A new copy of the array of data points was needed since we receive the data from the "user" in a 1D array which looks like the one in the image above. We depict the new arrangement:

$p_1[0]$	$p_2[0]$	$p_3[0]$	$p_4[0]$	$p_1[1]$	$p_2[1]$	$p_3[1]$	$p_4[1]$	$p_1[2]$	$p_2[2]$	$p_3[2]$	$p_4[2]$

Now a call to `std::nth_element` would most likely have a much smaller number of *cache misses*, since the "interesting" addresses in the array below (for a given axis i) are close to each other in memory.

This approach did not pay unfortunately, since a single experiment with 8 MPI processes and a dataset of 100'000'000 data points each having 3 components took about 163% the time needed to grow the tree before the modification on the same dataset. For this reason we decided to pause this improvement, a profiler could probably tell us what's happening inside the code.

3 Performance model and results

On the basis of the discussion we developed in Section 2 we propose the following performance model, which we are going to compare against the experimental results we obtained in several runs of our code:

$$\begin{aligned} T(P,N) = & t(\phi(P,N)) + pc(P,N) + cp(P,N) \\ & + \texttt{parallel\_split}(P,N) + w(P,N) + \texttt{merge}(P,N) \end{aligned} \tag{3}$$

This equation tries to model the time needed for a worker to grow a k-d tree. For simplicity we assume that $N = 2^k, k \in \mathbb{N}$, but it's not difficult to generalize the model with a few more technicalities. We briefly explain the meaning of the symbols in (3):

- T(P,N) is the time needed to grow a k-d tree having N elements on P parallel workers;
- t(N) is the time needed to grow serially a k-d tree having N elements;
- $\phi(P,N)$ is the number of elements in the first non-parallelizable branch generated by a worker which in principle can parallelize the generation of a k-d tree having N elements using P workers;
- $pc(P,N)$ is the time needed to communicate one branch per level to another worker, until the first non-parallelizable level is reached;
- $cp(P,N)$ is the time needed to retrieve the grown k-d tree from the workers which received a branch from this worker;
- $w(P,N)$ is the time that this worker needs to wait before the children branches complete their work;
- `merge(P,N)` is the time needed to build the k-d tree starting from the serial branch grown by this worker and the branches delegated to other workers;
- `parallel_split`(P,N) is the time needed to split the data in branches (one of them is delegated to another worker) before the first non-parallelizable level.

Note that $pc \equiv 0, cp \equiv 0$ if we use OpenMP, but this is probably a coarse approximation since there are hidden costs involved, for instance in maintaining cache coherency (we spent some words on this problem in Section 2.4). We simplify the model and assume the same cost of communication for sending a branch and receiving the corresponding grown k-d tree, which we call `comm`(P,N).

The following relation holds:

$$w(P,N) = \begin{cases} 0 & \text{if } P = 2^k, k \in \mathbb{N} \\ t(\phi(P,N)) & \text{otherwise} \end{cases}$$

The first condition means that we can parallelize in the exact same way all the regions of the k-d tree, while the latter means that some parts cannot be parallelized and therefore need to start the serial growth of the tree one level before the others (see `surplus_workers` in Section 2.2). Therefore we write $w(P,N) = \alpha(P)\phi(P,N)$ where $\alpha(P)$ is 0 if P is a power of 2, and 1 otherwise.

Recalling the serial algorithm which we employ to compute the k-d tree, the complexity is the same of *Merge-Sort* (this can be shown via the *Master theorem*), namely $t(N) \in O(N \log N)$. Therefore we assume that $t(N) = \beta N \log N$, where β is some proportionality constant.

We now look at $\texttt{parallel\_split}(P, N)$ and $\texttt{merge}(P, N)$. The former consists in the iterated application of `std::nth_element` to datasets whose size is $N, N/2, N/4, ..., N2^{-\log_2(\hat{P})}$, where $\hat{P}$ is the greatest power of 2 lower than or equal to P. Since the complexity of `std::nth_element` is $O(N)$, using the value of the summation of a geometric sum and introducing a proportionality constant γ_1, we obtain:

$$\begin{aligned}
\texttt{parallel\_split}(P, N) &= \gamma_1 \sum_{i=0}^{\log_2(\hat{P})} \frac{N}{2^i} && (4)\\
&= \gamma_1 N \sum_{i=0}^{\log_2(\hat{P})} \frac{1}{2^i} && (5)\\
&= \gamma_1 N (2 - 2^{-\log_2(\hat{P})}) && (6)\\
&= \gamma_1 N \left(2 - \frac{1}{\hat{P}}\right) && (7)
\end{aligned}$$

Since $\texttt{merge}(P, N)$ (if we use MPI) consists in a linear scan of the two branches which comprise a level, iterated for all the levels from the last parallelizable level to the root, we obtain relation very similar to (4):

$$\texttt{merge}(P, N) = \gamma_2 N \left(2 - \frac{1}{\hat{P}}\right)$$

If we use OpenMP no merging is needed (unless we employ the approach presented in Section 2.4), therefore $\texttt{merge}(P, N) \equiv 0$.

Lastly, we observe that $\phi(P, N) = \frac{N}{\hat{P}}$. At this point we can express a simplified model based on the assumptions we stated above:

$$T(P, N) = \begin{cases} (1 + \alpha(P))\beta \frac{N}{\hat{P}} \log \frac{N}{\hat{P}} + 2\texttt{comm}(P, N) + (\gamma_1 + \gamma_2) N \left(2 - \frac{1}{\hat{P}}\right) & \text{(MPI)} \\ (1 + \alpha(P))\beta \frac{N}{\hat{P}} \log \frac{N}{\hat{P}} + \gamma_1 N \left(2 - \frac{1}{\hat{P}}\right) + \texttt{hidden}(P, N) & \text{(OpenMP)} \end{cases}$$

Note that we added an “hidden cost” term in OpenMP to remind that OpenMP isn’t just “MPI *minus* communication”, and the experimental results are going to remark strongly this fact.

The model testifies that in our experimental results, as we increase the number of parallel workers P, we should observe a steep improvement of the performance each time P becomes a power of 2. However, the incidence of this improvement depends on the cost of communication in the MPI case, while in the OpenMP case the model is most likely unprecise due to unquantified hidden costs.

Another aspect which remains unexplained by this model is the decrease of execution time which we are going to observe in the following sections when we increase the number of parallel workers to a number which is not a power of 2 (for instance, see Figure 3). This is most likely due to the fact that parallel workers are not working equally fast. This means that waiting for the computation of the additional (i.e. non-parallelizable) level of (for instance) 5 or 6 workers is practically not the same of waiting for the computation of the additional level of one worker, since in the former case the probability of having a slower worker is much higher. The slower worker is a *bottleneck*, therefore increasing the number of workers (even though we do not reach the next power of 2) reduces the probability of having a bottleneck which stalls the other processes waiting for its portion of the dataset.

3.1 Software and hardware employed for the experiments

In the following sections we are going to present and discuss some experimental measurements in order to verify the fidelity of our theoretical model. The measurements have been carried out using a variable number of cores on a single *Thin* node on *Orfeo*. The code have been compiled using *gcc* 9.3.0, the MPI distribution taken into account is OpenMPI 4.0.3.

All the datasets have been generated using a small Python utility which we wrote for the purpose. The utility enables the generation of `.csv` files containing a random dataset following a *Normal* distribution, with an arbitrary number of data points (row) having an arbitrary number of components (columns). In particular for the experiments we fixed the number of components to 3, while the number of data points varies according to the experiment.

3.2 *Strong* and *weak* scalability

First of all we briefly discuss the *strong scalability* of our code, namely the performance gain given by the introduction of additional parallel workers while the problem size stays constant. As you can see in Figure 3 we observe the behavior

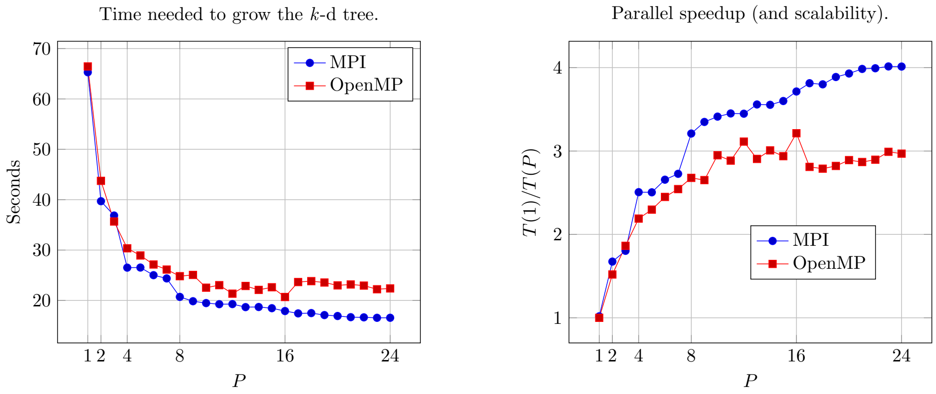


Figure 3: Strong scalability of the growth of a k-d tree ($N = 100'000'000$, $k = 3$).

we expected after the inspection of the simplified model (i.e. steep improvement everytime we add 2^k processors). Also, we observe that the hidden cost of OpenMP is definitely not negligible. The slight instability in the OpenMP line could probably mean that some OMP threads have been paused and assigned to other tasks. Note that in this case we can observe the effect we forecasted (and explained) in the last paragraph of the introduction to Section 3, i.e. the improvement of performance even if the number of parallel workers is not a power of 2.

In order to evaluate the *weak scalability* of our implementation, we generated a set $\mathcal{D} = \{d_1.csv, \ldots, d_{24}.csv\}$ comprising 24 datasets having varying sizes, such that $\texttt{len}(d_i) = i * 1000000$. With respect to [3, Chapter 5.3.2] we choose $\alpha = 1$. Then we measured for a number of parallel workers going from 1 to 24 the time needed to grow the k-d tree, and computed for each run the *parallel performance* $P^i = \frac{\text{Work}}{\text{Time}} = \frac{i*1000000}{T^i}$. Finally we computed the weak scalability $S_w^i = \frac{P^s}{P^i} = i\frac{T^s}{T^i}$ for the numbers of parallel workers taken into account (both for MPI and OpenMP). Our results are shown in Figure 4.

As expected the most "performant" points with respect to the varying workload are those where the number of parallel workers is a power of 2: in those cases the workload is equally distributed among all the workers, which results in a local maximum of the weak scalability plot. Moreover we recall the effect we talked about in the final part of the introduction of Section 3, which is observable also in the weak scalability plot: indeed we see a slight improvement (after a steep worsening) of weak scalability as the number of parallel workers increase, even if it is not a power of 2.

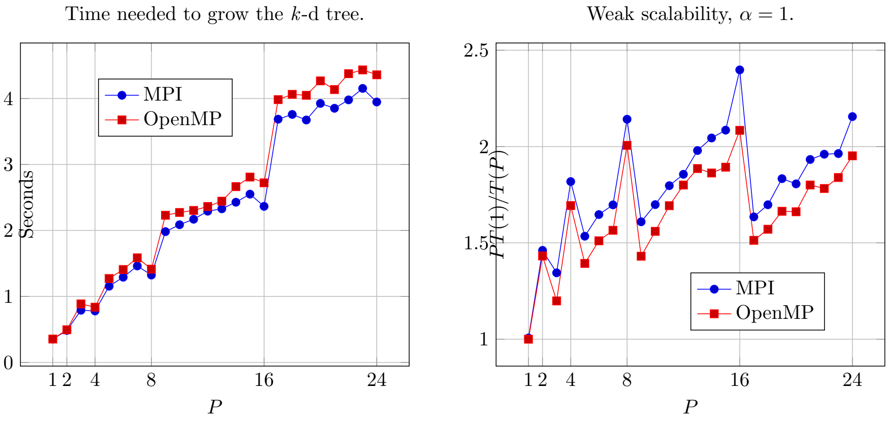


Figure 4: Weak scalability of the growth of a k-d tree for a linearly varying number of elements ($k = 3$).

3.3 More measurements

In this small section we gather several additional interesting experiments along with a brief discussion of the results.

3.3.1 Dependency on the number k of components

In this experiment we observe the dependency of the time measured during the growth of a k-d tree with a varying number of components k (N is fixed to 1000000). We also consider several numbers of parallel workers for both the MPI and OpenMP versions. The plot on the left of Figure 5 shows that the MPI version presents an approximately linear dependency of the time needed to grow the tree on the number of components in each data point, as the number of MPI processes involved increases (we have some unstable measurements when 2 processes are used). The OpenMP version by contrast is much more stable when a small number of OMP threads is used, and we can observe a less steep linear dependency on the number of components k. Note that since our parallel algorithm is agnostic on the number of components (i.e. in theory we do not expect any particular overhead when the number of components increases) the observed increased cost for big values of k is entirely due to memory management issues (like *false sharing*, moving memory, cache coherency, message passing).

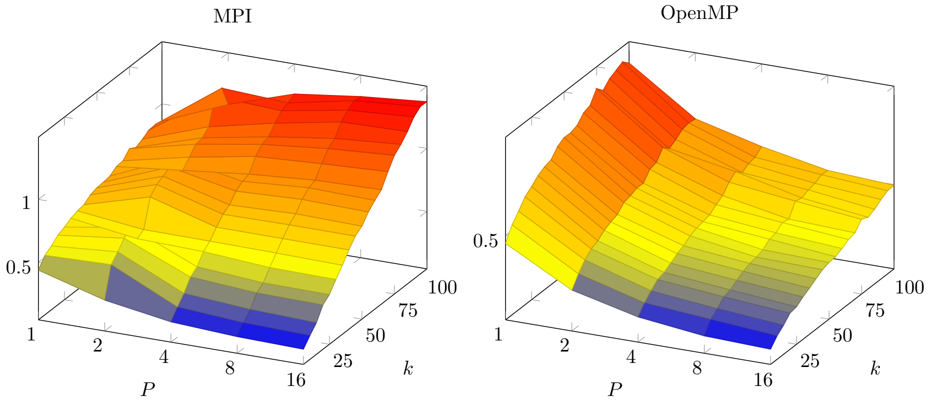


Figure 5: Performance measured for the growth of k-d tree with $N = 1000000$, varying the number of components k.

3.3.2 Compiler optimzation

We comment a little bit the effect of compiler optimization flags on the performance of the code. As you can see in Figure 6 there is a clear improvement when the flag `-O1` is used, but further flags do not enable any particular speedup. In particular `-O3 -march=native` is also somehow less regular than the other compilations. The fact that the plots for higher optimization levels are almost overlying is probably due to the fact that the most computational intensive

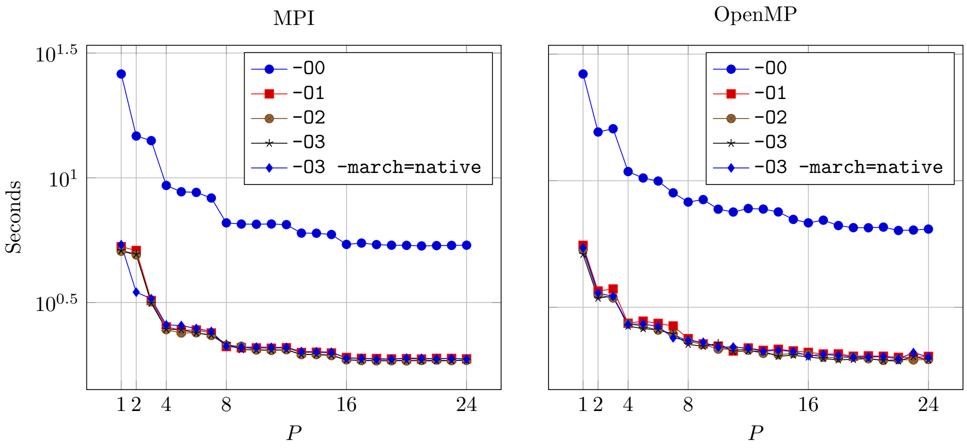


Figure 6: Performance determined by compilation with different compiler optimization flags ($N = 100'000'000$, $k = 3$).

operations in our algorithm are the calls to `std::nth_element` and memory operations, which are only slightly influenced by the compiler. We might be able to see more distinct lines if we consider smaller datasets and a small number of processors.

3.3.3 `double` VS `float`

Another interesting experiment concerns the usage of `double` instead of `float` values (which have been used until now in the experiments). Using `double` values allows the digital representation of a bigger variety of datasets, but this

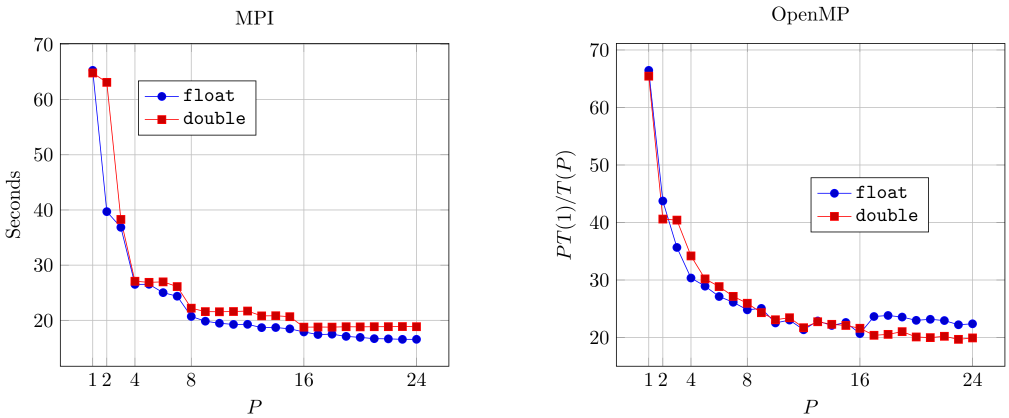


Figure 7: Time taken by the growth of a k-d tree ($N = 100'000'000$, $k = 3$).

comes at an increased computational cost. We structured the code in order to enable the user to choose which data type should be used on compile time. In Figure 7 we show the difference in time determined by the data type chosen. The difference is not so clear in the OpenMP case, but it becomes noticeable when MPI is used: this is most likely due to the fact that communication becomes much more costly in the latter case.

3.3.4 False sharing

Finally in Figure 8 we present the results obtained using the approach discussed in Section 2.4, which is aimed to fixing the problem of *false-sharing*. Since this is peculiar of the OpenMP case we are not going to show any result using MPI. As you can see the (—■—) line (i.e. the version with the fix enabled) is initially less performant than the default version, since as we mentioned earlier the approach involves an additional scan of the 1D array which represents the tree in memory. However, the approach starts paying the effort as the number of OMP threads increases, which means that the number of parallel workers which could write concurrently on the same cache line increases.

3.3.5 GPU nodes

In Figure 9 we plot the performance measured on a single GPU node going all the way from 1 to 48 CPUs. As you can see the overall performance is worse than what we measured on THIN nodes, so much that the lowest time measured is still above 20 seconds for the same dataset comprising 100’000’000 data points which we considered in Figure 3. At $P = 25$ the MPI version presents a steep descent, exactly when as we go beyond the number of processors on a single

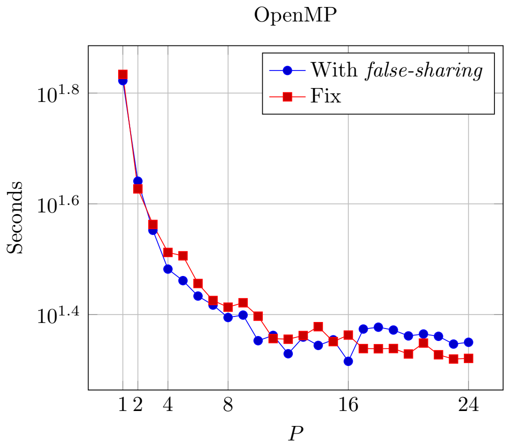


Figure 8: Results obtained applying the fix for false sharing presented in Section 2.4 ($N =$ 100’000’000, $k = 3$).

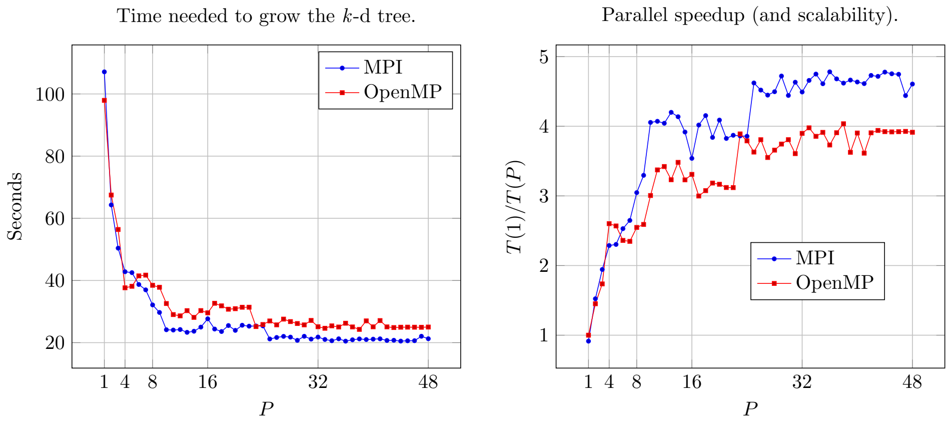


Figure 9: Strong scalability of the growth of a k-d tree on a single GPU node (N = 100'000'000, $k = 3$).

node. From this point the MPI performance becomes approximately a plateau, most likely due to the fact that adding new MPI processes does not guarantee any particular gain due to the additional communication overhead. In order to clarify the situation we present the plot in Figure 10, where we summarized the behavior of the MPI version for different usages of the parameter `--map-by` of `mpirun`. In this experiment we incrementally add two processors from two different GPU nodes for each measurement; then we measure the performance in the three cases (`core`, `socket`, `node`). As expected the worst performance overall is the one measured for `node`, since in this case we have the maximum distance for the communication between MPI processes.

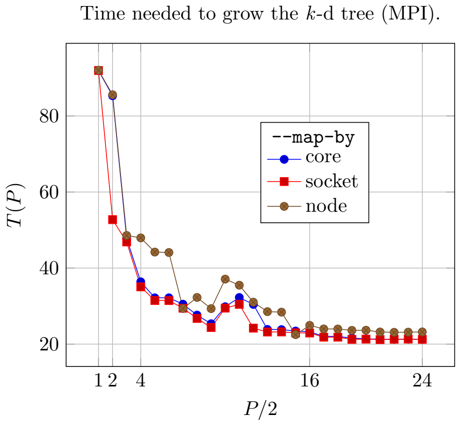


Figure 10: Time mesured adding incrementally pairs of processors on 2 GPU nodes, i.e. 2,4,...,46,48 (N = 100'000'000, $k = 3$).

3.4 Brief discussion of OpenMP results

Looking at Figure 3 and the other plots we showed in this section it is clear that MPI outperforms OpenMP in our experiments. Even the improvement discussed in Section 4, which allows us to improve slightly the performance of the OpenMP version as the number of threads increases, does not guarantee performance matching those of the MPI version. We now discuss a little bit some possible motivations.

First of all, as we mentioned earlier, the 1D representation of the tree we conceived implies the concurrent modification in different OMP threads of memory addresses which could reside on the same cache line, which causes *false sharing* and therefore worse performance. The fix we discussed may solve the problem, but with an additional cost given by the additional scan and re-arrangement of the array (which is done serially).

Another factor is most likely NUMA topology [3]. The node we borrowed to run our experiments contains multiple sockets, each of which contains multiple cores. One of those cores corresponds to `thread0`, i.e. the main thread, which allocates the memory needed to represent the k-d tree. Due to the *first touch* policy the memory is allocated on `thread0` which is the first one which uses the addresses. Cores which reside on other sockets incur in additional reading and writing costs due to the Non Uniform Memory Access. This problem may be solved allocating memory on each child thread to be used to grow the tree. Afterwards that memory should be copied to the parent buffer in an efficient way.

4 Conclusions and further developments

The full code that this report is based on is available at this link (GitHub repository), along with a `README` file and several code examples. I'm more than happy to provide more details or clarifications about parts of the code which have not been touched in this brief report.

In Section 2 we proposed several technical improvements to overcome problems in the frameworks we used to parallelize our code, and to improve the usage of the cache hierarchy. Even though those improvements look quite reasonable, it seems that they do not enable any particular speedup. We identified two main reasons to explain this fact:

- We overreacted to some issue which in our case is not awfully problematic;
- The improvement introduced an additional cost in another part of the code which nullifies the performance gain.

For instance, *false-sharing* might not be a huge problem in our case, since we write in the 1D array which represents the k-d tree exactly once for each position. Also, the improvement introduced to reduce *false-sharing* requires a new scan of the array (not present in the original code) with additional cache problems.

As we mentioned in the previous sections we plan to use a profile in order to identify which problems damage the most our performance results. There are still some unanswered questions, like for instance what happened in Section 2.5, i.e. why are the results we measure so bad with respect to what we expected. A profiler may definitely give the answer, or at least point us in the right direction.

We also tried to join the efforts of MPI and OpenMP in order to deliver the maximum possible parallelization. Unfortunately the increased complexity in the generation of PBS jobfiles needed to run the experiments on *Orfeo* made us desist, therefore there are no "hybryd" results in this report. However the code we wrote is actually available in in the branch `openmp-plus-mpi` of the GitHub repository at this link.

A possible way to improve performance when using MPI is replacing the multiple `MPI_Send`, `MPI_Recv` spawned by parent/children processes pairs with one single collective call to `MPI_Gather`, using the root MPI process as the receiver. Then the root process should merge the partial results produced by all the children processes, therefore an efficient algorithm to reconstruct the k-d tree is needed. Also, currently we have multiple `merge` operations which work in parallel bottom-up up to the root level. The improvement proposed in this paragraph would remove this kind of parallelism, and we should evaluate if the communication saved is worth it in order to check if this approach gives any kind of speedup in the end.

References

[1] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 2009.

[3] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers.* CRC Press, 2010.

[4] Martin Skrodzki. The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time. *arXiv preprint arXiv:1903.04936*, 2019.