Assignment 2 - Report

Andrea Barbieri

Foundations of High Performance Computing - A.Y. 2021-2022

1 Introduction

The main objective of this assignment is the parallel implementation of a k-d tree builder, using both UMA and NUMA approaches to memory (implemented using, accordingly, MPI and OpenMP). A k-d tree is a particular type of binary tree, developed by Friedman, Bentley and Finkel [2] as a solution for the nearest neighbor problem, where each node of the tree represents a partition of the input dataset. The main goal of the algorithm to be implemented is to create such a tree from a dataset of N k-dimensional points, focusing on parallel implementations exploiting shared and distributed memory approaches.

1.1 Assumptions and definitions

The main assumptions for this assignment are:

1. The points coordinates are fixed to $k = 2$;
2. The dataset is assumed to be immutable, hence insertions/deletion operations can be ignored;
3. The data points are assumed to be homogeneously distributed in all the dimensions.

Before moving on, a more rigorous definition of "point" and "node" is needed. A "point" is an entity defined by k coordinates on a k-dimensional space and the dataset is composed by a number n of points. As stated in the first assumption, only two-dimensional points will be taken into consideration. A "node" is a custom structure containing a split point, a split axis and two pointers to the left and right child. As any other binary tree, the tree that is going to be built will have a certain number of internal nodes (nodes with at least one child) and leaf nodes (nodes with no children), the latter will be identified by the absence of pointers to their children. In the following sections a C-like syntax will be used to specify an attribute of a certain node (e.g. ThisNode.split means the splitting points of the node "ThisNode").

2 Algorithm

2.1 Tree-building function

The core of all the three implementations is the `BuildTree()` function, used to construct the k-d tree from a starting dataset. This is a recursive function, whose pseudocode can be found below:

Algorithm 1: Tree-building function

```
Input: a set of points and the axis of the father.
Output: A pointer to the computed node.
Function BuildTree(dataset,axis):
    1. Allocate LocalNode
    2. LocalNode.axis ← $(axis + 1) \mod 2$
    if SIZE(dataset)==1 then
        3. LocalNode.split ← dataset
        4. LocalNode.left ← NULL
           LocalNode.right ← NULL
    else
        3. LocalNode.split ← BestSplit (dataset,axis)
        4. LeftPoints, RightPoints ← AssignPoints (dataset, axis, split)
        5. LocalNode.left ← BuildTree (LeftPoints, LocalNode.axis)
           LocalNode.right ← BuildTree (RightPoints, LocalNode.axis)
    return *LocalNode
```

At first, the function allocates the needed space for the node and computes the axis along which the dataset will be split, according to the formula shown (specifically designed for $k = 2$). Then, according to the size of the dataset (i.e. the number of observations) two path are possible: the simplest one, the case $SIZE(dataset) = 1$, will return a leaf node, containing the dataset itself as splitting point and null pointers to the children. In the other case, the median point along the computed axis will be found and saved inside the node (by means of the `BestSplit()` function), then the dataset will be split into "left" points (points with a value smaller than the median along the splitting axis, assigned to the left child) and the "right" ones (points with value greater than the median along the splitting axis, assigned to the right child), using the `AssignPoints()` function. The `TreeBuilding()` function will be then recursively called for the two partitions created, building the two children nodes. Either way, a pointer to the computed node will be returned. Note that the median as splitting point was chosen due to assumption 3. Further information about the `BestSplit()` and `AssignPoints()` functions and the actual implementations of `BuildTree()` will be addressed in section 3.

2.2 Serial and parallel implementations overview

As for the overall implementations of the tree builder, we start by briefly talking about the serial implementation. This has a straight-forward approach, heavily relying on the aforementioned `TreeBuild()` function and the only notable detail is the final tree, for which we only now (at start) the address of root node: from there we can access other nodes using the pointers to the children contained in each node.

On the other hand, the parallel implementations differ from the serial one and from each other, the only common feature is the main approach to parallel programming, i.e. domain decomposition: the workload (i.e. data points to be processed) is shared among workers and each worker will build part of the tree. However, similarities stops here, due to the radically different approach to memory adopted by the two implementations that led to two very different implementations.

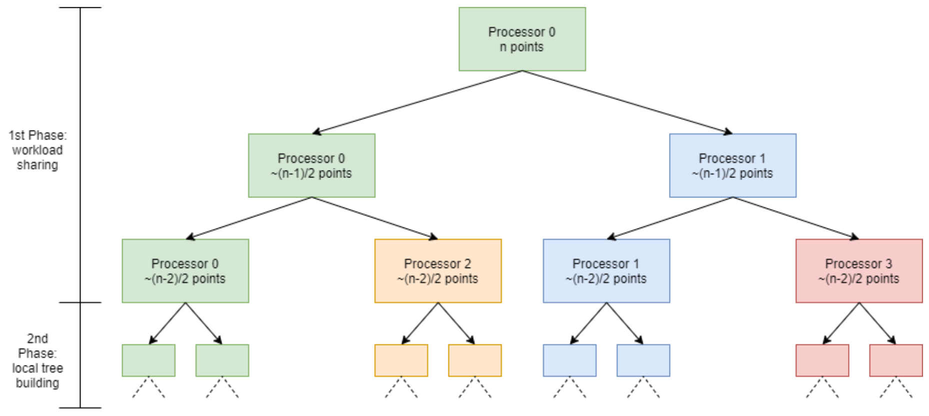
In the shared memory one the workload is managed by the task queue and more node-oriented: the elementary unit of work is indeed a tree node and its building is managed by the use of tasks. Each worker will dynamically build a tree node, according to the task queue state, hence we can not predict what node will be computed by which worker, however the dynamic assignment of work will hopefully avoid idle threads. As for the final result, the shared memory implementation is similar to the serial one, returning a pointer to the root node, with all the nodes linked one another due to the father-child relationship intrinsic in the node.

Conversely, the distributed memory approach is more points-oriented, with non-dynamic workload sharing (in the sense that each process receives a fixed amount of points to use). The entire tree-building process can be summarized into three phases, the first two are shown in the figure. Giving a high level idea, the phases are the following:

1. **Workload sharing**: here processor 0 searches for the best splitting point and splits in half the dataset. Then, the second half of the dataset is sent to processor 1, while processor 0 retains the first part. This procedure is then reiterated by the two active processors (0 and 1), sharing in the same way the dataset with processors 2 and 3, and so on. The stop condition is determined by a certain depth level of the tree, past that level we enter the second phase.

2. **Local tree building**: once the previous stop condition is reached, every processor builds the local tree, i.e. the nodes involving the local points. Each node will be saved into an array following a post-order fashion.

3. **Computed nodes gathering**: at last, once every processor has completed the local tree starts sending/receiving the computed array of nodes, following the same ordering as the first phase backwards until processor 0 is reached.



1st Phase: workload sharing
2nd Phase: local tree building
Processor 0 n points
Processor 0 ~(n-1)/2 points
Processor 1 ~(n-1)/2 points
Processor 0 ~(n-2)/2 points
Processor 2 ~(n-2)/2 points
Processor 1 ~(n-2)/2 points
Processor 3 ~(n-2)/2 points

Figure 1: Distributed memory implementation scheme. The figure refers to a hypothetical dataset of size n and 4 processors.

The approximation in the number of points shown in Fig. 1 is due to the fact that when a father node has an odd number of point to split, the left node will receive one more point with respect to the right node. The reason behind that is addressed in subsection 3.1. The final result is an array containing all the nodes of the tree, indexed in a post-order fashion and with the internal pointers to the children nodes replaced by their location inside the array, expressed by the index.

3 Implementation

The three different codes are implemented in the C programming language, using also OpenMP for the shared memory implementation and MPI for the distributed one. The serial and OMP sources can be found in the "*kd-tree.c*" file, varying accordingly the compilation flags, while the MPI source can be found in the "*kd-tree_MPI.c*" file. Both files use a library called "*util_functions*" where common functions are stored, which is statically linked in the Makefile.

3.1 `BestSplit()` and `AssignPoints()`

The first technical aspect discussed is the `BestSplit()` function implementation to find the median point in a given partition. The simplest solution is to sort the entire partition and return the $n/2$-th element and in a first implementation the algorithm implemented were exactly that. The solution exploited the merge sort algorithm with an average time complexity of $O(n \log(n))$. However, this algorithm, even though efficient, does an unnecessary job since we only care about how points are related to the median point (i.e. if they're greater or smaller than the median) for the successive steps, and not the actual complete sorting. An alternative solution was hence chosen, replacing the quicksort algorithm with the quick-select [1], due to its lower time complexity ($\Theta(n)$ on average) and its ability to partially sort the points: given the particular functioning method, the final array used by the quick-select function will contain points lower than the median to the left side of the array (and actually sorted), while the second half will contain the non-sorted points greater than the median. The partial sorting of the points makes the latter splitting part faster, since we don't have to compare the points to the median to assign the left and right points, but we directly take the first half for the left child node and the second one for the right child.

Another issue encountered was the high cost of the swap in the quick-select algorithm: since we're dealing with k-dimensional points, for each swap we have to read and write $2k$ different values, which for large k may become very expensive computationally. Furthermore, as we will see in the dedicated OpenMP section, sorting points may lead to false sharing issues. To solve this problem, the quick-select algorithm was converted to use the indexes of the array for sorting and searching for the median point. In this way, no matter the number of dimension, the cost of swapping will remain constant since we only have $2k$ integer values to manage.

Before moving on, a detail regarding the father-children relationship must be addressed: given how the `BestSplit()` function was implemented, whenever an odd number of points needs to be split in left and right, the left node will always receive one point more than the right node. A consequence of this is that a node could have both child or only the left one (in particular, when $n = 2$), hence the right-node creation was made conditional to the number of points. As we will see, this situation will come in handy when talking about OpenMP optimization.

3.2 OpenMP details

The OpenMP implementation, as said before, heavily relies on tasks, linking each node to be built to a new task. At first, `BuildTree()` is called inside a single directive, then the function will create two tasks for the left and right nodes, which will go on recursively creating other tasks. Tasks will be managed using a queue, where each free thread will take and compute one task at a time. Given the default implementation of tasks, an explicit shared flag for the dataset has been added: in this way each thread can access to the dataset in the shared space, without having to create a private copy for each task.

As cited before, a possible issue in the OpenMP implementation is false sharing: in a general scenario, this happens when each thread explicitly access a memory location that is different from any other thread but at least some of those memory locations reside in the same cache line. In our setting, this may happen when two threads are building a node with values located in the same cache line: whenever a thread start to swap values, the other thread must re-flush the cache to maintain the coherence, resolving in a potentially huge synchronization overhead. To prevent this issue, whenever we have to sort some points we sort a separate array containing the indexes of the points: in this way, the original dataset is left untouched and used only to read the values, while each thread, which has its own copy of the array's indexes to sort, can proceed with the computations without false sharing issues.

Another issue is the huge number of tasks: since we create a task for each node to compute, the total number will be n and tasks will be created even for nodes with just a couple of points to sort, leading to more time spent in "bureaucratic" overhead (picking up the task, creating the private copies, and so on) than in actual computations. To prevent this problem, two solutions were designed:

- **Tasking only the right children**: when a thread picks up a task, it will compute all the left descending nodes, "tasking" only the right ones. The total number of tasks will then be equal to the number of right children, which, in the worst case (a balanced tree), will be $n/2$. Even if with this approach threads may be tasked to compute more than one node, the issue of tasks with low number of points remains and in the worst case $n/4$ tasks will be created to compute nodes with only one observation.

- **Serialization**: when a certain condition is reached, threads stop creating new tasks and compute all the descending nodes, similarly to the MPI implementation. As stop condition, both the depth of the node and the number of points in the local partition were tested. This approach removes the low number of points issue, however it may lead to idle threads as well as load unbalance.

To choose the best solution, each approach was tested using a 10^7 points dataset and 4 threads inside a VirtualBox machine with 4 processors. Each program was ran eight times on the same dataset and the resulting boxplots can be seen in Fig. 2.

As we can see, the serialization approach (here with the condition `n_points < 20`, the best parameter tested) yields results similar to the standard one, with an average time slightly higher. This may be due to the fact that, for the most part, the two programs are identical since the condition introduced applies only in a limited number of cases. Summarizing, no one of the parameters tested showed an improvement in the code, making it slower or, in the best case, equal to the standard implementation. However, the right-tasking approach seems to work instead: the times obtained were (on average) lower than the standard ones, even though with a higher variance. In the end, the right-tasking method was chosen and implemented in the final code.

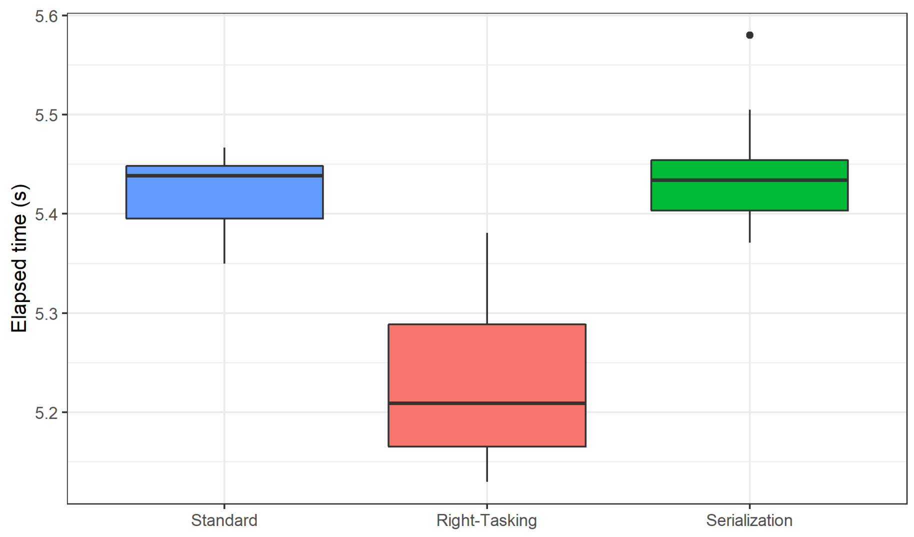


Figure 2: Boxplots of the times elapsed to build the trees.

3.3 MPI details

Among the three implementations, the MPI one was for sure the most tedious one: until now, each implementation was designed exploiting a shared space, resulting in very similar programs. The MPI implementation could not rely on such feature, leading to a completely different code and overall approach. Focusing on the implementation details, at first two custom data types for MPI were defined, one for the points and one for the nodes. Also, the node struct for the MPI implementation was changed, replacing the left and right pointers to plain integer values, that will be used to locate the children via the array index.

Another issue emerged when dealing with workload sharing: the first solution implemented made the master processor build the tree nodes until a certain number of nodes (equal to the total number of processes) was reached, only then the data points were shared. This solution however was poor particularly in terms of parallelism: before all the processes started working on the tree, the master processor had to compute the first $p-1$ nodes, which are also the most expensive nodes to compute due to the high number of points. An alternative solution was hence designed, ending up with the one presented in section 2. In detail, the communication flow was designed to get the processes receive points just a single time and, if needed, to send points to other processes. The condition used to verify whether a process needs to send points or continue with computations was the following:

$$\begin{cases} \text{send to ID} + 2^d & \text{if } 2^d < p \\ \text{continue with computations} & \text{if } 2^d \geq p \end{cases}$$

Where ID is the rank of the sender, d is the depth of the node and p is the total number of processors. Note that, because of the presented formula the program will only work if the number of processes can be expressed as $2^k, k \in \mathbb{N}$ and if that condition is not satisfied, processes will try to send data to a non-existing process rank, causing the entire program to crash.

A similar formula is used in the opposite direction from the processors waiting to receive a message. Indeed, by default each processor but the master one starts in a `MPI_Recv` state, waiting for the points to arrive and to identify the sender the following formula is used:

$$S = \mathrm{ID} - 2^{\lfloor \log_2(\mathrm{ID}) \rfloor}, \tag{1}$$

where S identifies the sender's rank.

Each communication consists in a first part, where the number of points that will be transferred is shared, and a second one where the points are actually transferred. Then, the receiver allocates a number of nodes equal to the number of points: in case the processor does not have to send any other point this is exactly the space needed for the computations, in the opposite case the empty space allocated will be later filled with the nodes computed by other processors. Jumping to the gathering part, this follows the same route as the workload sharing but backwards, joining the computed nodes in following the post-order as soon as they arrive. Each process, depending on its own rank, will receive a number R of different messages from other processes, containing the computed nodes. This number is defined as:

$$R = \begin{cases} 0 & \text{if } \mathrm{ID} \geq p/2 \\ \log_2(p) - \lfloor \log_2(\mathrm{ID}) \rfloor - 1 & \text{if } \mathrm{ID} \neq 0 \text{ and } \mathrm{ID} < p/2 \\ \log_2(p) & \text{if } \mathrm{ID} = 0 \end{cases} \tag{2}$$

Note that processes with rank $\geq p$ will not receive any message, starting immediately to send the computed nodes. Once the nodes are received and saved in a temporary array, the process joins them with the local computed ones by simply putting them in the first empty space inside the local node array: given the way the nodes are gathered, this ensures that the post-order is followed at each step, ending up with the full tree sorted using the same sorting. Whenever a node is copied from the temporary array to the local array its internal indexes to the children nodes are updated to maintain coherence: since the indexes refer to the original array, the update consists in adding the number of nodes computed in the local array. In such way that the index will point to the position of the children in the new array. After R iterations, each process will send the array containing all the computed nodes to the next process, using (1). A scheme of the gathering phase is presented in Algorithm 2.

All the message passing part is done entirely using blocking operations (i.e. `MPI_Send` and `MPI_Recv`), letting MPI to choose the best strategy to communicate: this may be a potential bottleneck in the workload sharing part since

Algorithm 2: Gathering procedure

```
for $i$ in {0, ..., R-1} do
    1. $n\_arriving \leftarrow 0$
       $sender \leftarrow \text{ID} + 2^{\log_2(p)-i-1}$
    2. Receive $n\_arriving$ from $sender$
       Allocate $n\_arriving$ temporary nodes
    3. Receive nodes from $sender$
    4. Join the received nodes with the local ones
$receiver \leftarrow \text{ID} - 2^{\lfloor \log_2(\text{ID}) \rfloor}$
Send the local nodes to $receiver$
```

processes must wait until the communication operation is complete before going on, however it was considered the best option in terms of safeness of the data sharing operation.

4 Performance model and scaling

Once the codes were completed they were tested to assess the scalability in both a weak and strong scenario. All the test were carried on the ORFEO data center, using thin nodes equipped with two Intel Xeon Gold 6126 processors (base clock 2.6 GHz), counting 48 physical cores in total, without hyperthreading. All the three implementations were compiled using `gcc` compiler, adding also the `-O1` flag for optimization, the best one found after some repeated tests with various flags. Once the programs were compiled, a synthetic dataset of 1.6×10^8 points was created, randomly generating the point's coordinates from a uniform random variable with range $[0, 1]$. This was achieved by means of an external program, called *data_sample*. Lastly, a note about the timings: the definition of "elapsed time" is the time taken by the entire program only to build the tree, expressed as time taken by a single worker (the master one, identified with rank 0).

4.1 Strong scalability

We first look at the strong scaling results. The main index to assess this kind of scaling is the speedup, which is defined as:

$$S = \frac{T(1)}{T(p)}, \tag{3}$$

where $T(i)$ identifies the time taken by i workers to complete a fixed size job. For these tests, the first 10^8 points of the dataset were used, and the program was executed using an increasing number of cores, following the power of 2 up to 32 cores. The obtained results can be seen in the figures 3 and 4.

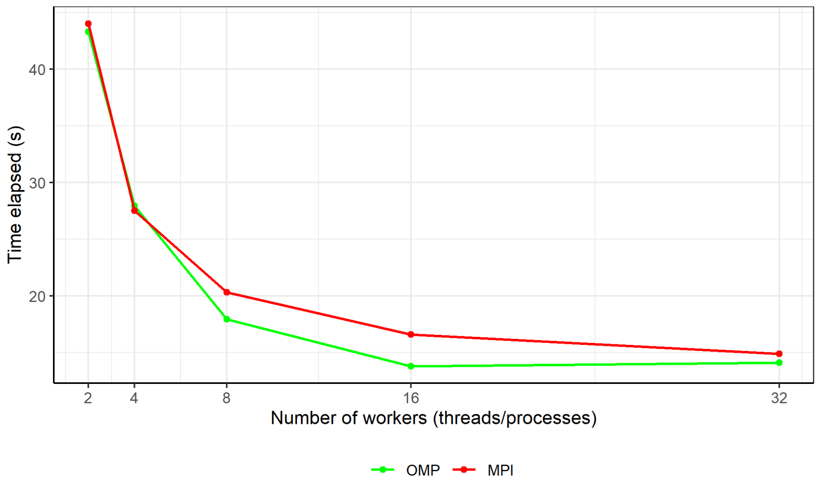


Figure 3: Elapsed time to build the trees as function of the workers in a strong scaling scenario.

We can see that, for a low number of workers, the two programs behave almost in the same way, however as the same number increases the MPI implementations slows down, the worst case being with $p = 16$ where the distributed-approach program is 20% slower than the OMP counterpart. Lastly, a strange behavior can be seen with $p = 32$: the OMP implementation is still the fastest one, however it is 2.32% slower than the previous run with $p = 16$. This may be due to the fact that the program ran on two different nodes and could not rely on a common shared space anymore, causing the entire execution to slow down. In terms of speedup, we can say that the program does not scale very well for a high number of workers: the best achieved speedup is 5.56 with $p = 16$ and the OMP implementation, resulting in a 34.77% efficiency (defined as S/p), which is particularly low if compared with the 88.71% obtained with $p = 2$.

4.2 Weak scalability

To assess weak scalability, the main index is efficiency, defined as:

$$\text{Eff} = \frac{T(1)}{T(p)}, \tag{4}$$

where $T(i)$ is the time taken by i workers to compute an amount of work proportional to a baseline, often defined as $p * \text{base}$. In our case, the chosen baseline was 5×10^6 points, resulting in the following (workers, points) pairs:

$$\{(2, 10^7); (4, 2 \times 10^7); (8, 4 \times 10^7); (16, 8 \times 10^7); (32, 1.6 \times 10^8)\},$$

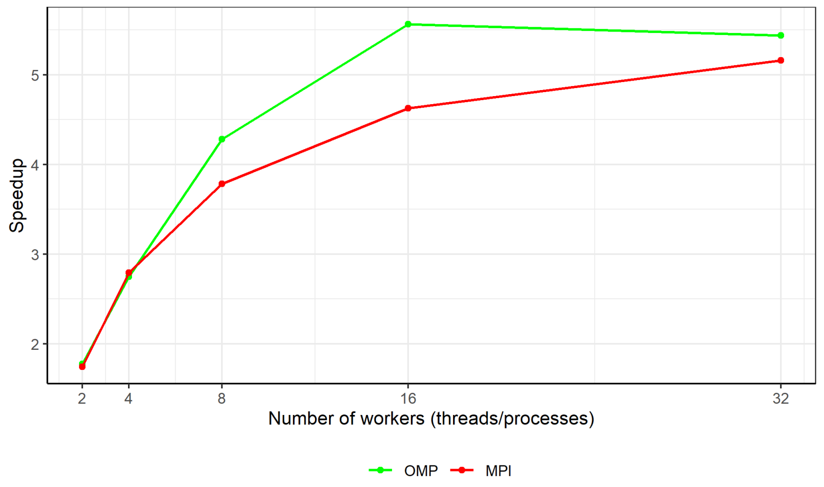


Figure 4: Resulting speedup from computed times in a strong scaling scenario.

in order to keep constant the amount of work as both the number of workers and the number of points increases. The obtained results can be seen in the figures 5 and 6.

The main idea we get from the graphs is that neither of the implementations reaches the perfect scaling: the elapsed time, which is constant in a perfectly scalable application, increases following a linear trend as the size of the workers increases, consequently we can see a linear drop in efficiency terms, going from more than 70% when $p = 2$ to less than 10% when $p = 32$. Comparing the two performances, we can conclude that they are very close one to the other, however the MPI implementation seems to scale a little better, with a maximum gain in efficiency of 4.6% with respect to the OMP implementation, when $p = 4$.

5 Discussion

In conclusion, we can say that the two implementations are particularly effective when paired with a low number of workers, however as that number increases the implementations show some slowdowns and the gain in terms of elapsed time is no longer "sustainable" in terms of number of processors. In particular, we can see that the OMP implementation scales well when the program is run inside a single node where it can effectively use the shared space, while it show some drawbacks when used on multiple nodes. The MPI implementation on the other hand seems to scale better in the long run: while it does not reach the same efficiency levels in a strong scaling scenario, even when the computations are done on multiple nodes it shows at least some gains in terms of efficiency and elapsed time (the OMP one as we have seen is even slower in the strong scaling when multiple nodes are used).

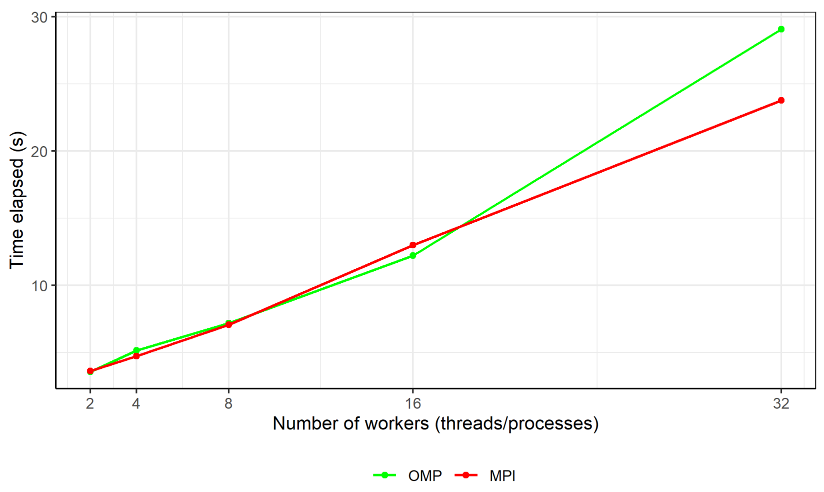


Figure 5: Elapsed time to build the trees as function of the workers in a weak scaling scenario.

5.1 Possible improvements

Besides the considerations in the previous section, all the three implementations are far from perfect, and could be improved in several ways. Here is a list of the most notable features that could be improved:

- **`BestSplit` function**: some possible refinements of the function used to search for the median could be the implementation of a randomly selected pivot or pivot selection via median of medians. Besides that, another possible improvement could be replacing the quick-select algorithm with a linear scan of the array searching for the median, as suggested in [2].
- **Hybrid approach for OMP tasks**: as we have seen, the right tasking solution was the best one to lower the overall number of tasks, however it does not remove entirely the problem of useless tasks (i.e. tasks with low number of points). It could be interesting to try then a mixed approach between the two, right-tasking only and serializing the computation as the number of points is below a certain threshold.
- **2^k processes constraint on MPI**: due to the design of the MPI implementation, we have seen that there is a constraint regarding the number of processes that can be used to run the program. This can be particularly expensive in terms of cores that can be used: in the tests carried on, five were done inside a single done, while only one could be carried on two nodes. Hence, a different communication flow could be designed in order to fully exploit the available resources.

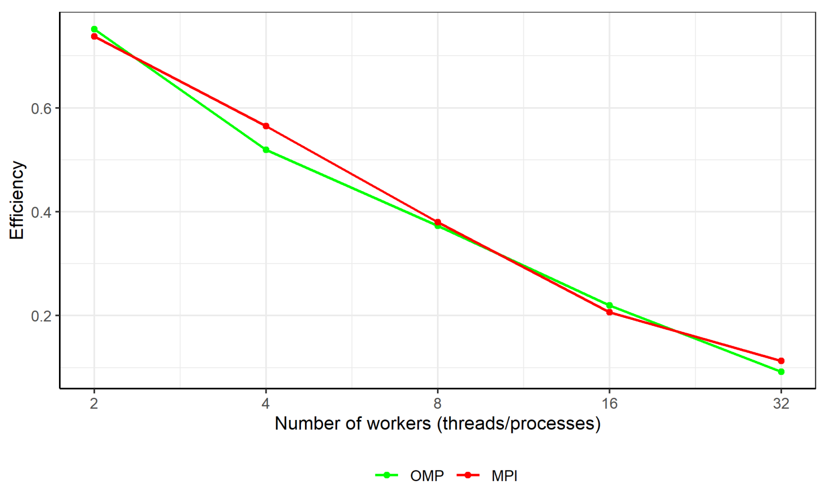


Figure 6: Resulting efficiency from computed times in a weak scaling scenario.

- **Non-blocking communication on MPI**: we said that the blocking communication were the safest ones, however it may be possible to increase performances using non-blocking communications in the workload sharing phase: a processor sends the data and carries on with the computations, avoiding being idle until the dataset partition is reached from the receiving process. With careful management of the `MPI_Status`, this could resolve in a solution both safe and performant.

- **Other non-efficient choices on MPI**: inside the MPI code (as well as the others) there may be many non-efficient choices in terms of algorithms and codes in general. Some of the most notable ones are the high usage of the `log2()` to compute the rank of the processes, which is a particularly expensive operation to compute with respect to sums or multiplications. Implementing a different operation to get the ranks may resolve in a (slightly) faster code. Another expensive operation is the one done when the nodes are gathered and the arrays joined: to see where the first empty space is the array is scanned linearly until the spot is found, however this could be avoided by a direct computation of the first empty array cell, which is possible since the number of already computed nodes is known.

References

[1] Hoare A.C.R. Algorithm 65: Find. *Communications of the ACM*, 7(4):321–322, 1961.

[2] Finkel R. Friedman J., Bentley J. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.