Kd-Tree Parallelization

Alessandro Serra

March 2022

Contents

1 Introduction

1.1

K-dimensional tree is a data structure widely used for partitioning and organizing points in a k-dimensional space, they 're involved in many different applications such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches). In the following report we will analyse two parallel implementations of a kd-tree, one employing the MPI interface and the other one the OpenMP framework. Firstly we will discuss their structure from an algorithmic perspective and then we will proceed with a comparison of their overall performances. In addition for clarity we wrote a small routine for printing a graph representation of the tree.

2 Algorithm

2.1 Serial Algorithm

The essential structure of the algorithm consists in choosing an axis in the k-dimensional space and selecting as a node a point from the data set, then reiterate the procedure on the subsets to the left and to the right of the selected point with respect to the chosen axis. The resulting tree is supposed to be balanced and in order to simplify the problem we assume the points to be homogeneously distributed in each dimension, so at each iteration we can cycle the axis and pick as a new node the median point over the selected dimension.

Algorithm 1: Build Kd-tree

Data: list of points pointList,int axis
$x \leftarrow$ axis
$x \leftarrow$ axis (mod number-of-dimension)
select median by axis from pointList
node.leftChild $\leftarrow$ kdtree(points in pointList before median-1, axis+1)
node.rightChild $\leftarrow$ kdtree(points in pointList after median+1, axis+1)
return node

In order to find the median we employed a variation of the *Quicksort* algorithm which orders the vector of points up to the element in position $\frac{\text{size of array}}{2}$ corresponding to the median. The algorithm, called *Kth-smallest*, require all the elements of the array to be different from one another so there cannot be repeated elements. In specific we're using the utility **std::nth_element** which employ the *Introselect* algorithm, an hybrid of *Quickselect* and *Median of Medians* which has fast average performances and optimal worst-case performances.

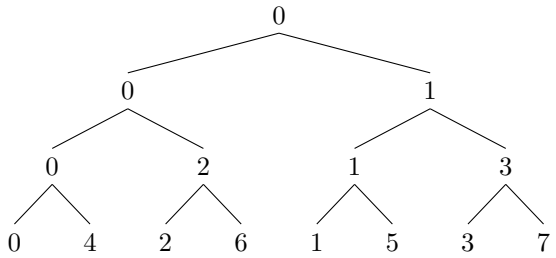


Figure 1: Communication tree

2.2 MPI Algorithm

The parallelization of the serial algorithm through the MPI interface consists in distributing the recursive calls among the processes provided, which for simplicity are assumed to be powers of two. The communication scheme among the parallel worker can be represented through the binary tree at figure 1.

Starting from the master process - rank 0 in our implementation - each worker keep half of the data set received and the send the other half to his son, once all the processors has received their input data they proceed the computation serially. At the end of this section each processor, with exception of the master, starting from the bottom of the tree, start sending to his parent the tree computed on the received portion of the data set. The parent, on his behalf, will merge the received buffer with his tree so that eventually the master process will have the complete tree.

2.3 OMP Algorithm

The OpenMP implementation is similar to the MPI one although it follows a much simpler scheme. The first call of the function is made by master thread and then each recursive call is assigned to a new task.

Algorithm 2: Build Kd-tree

Data: list of points pointList,int axis
$x \leftarrow$ axis (mod number-of-dimension)
select median by axis from pointList
Task: node.leftChild $\leftarrow$ kdtree(points in pointList before median-1, axis+1)
Task: node.rightChild $\leftarrow$ kdtree(points in pointList after median+1, axis+1)
return node

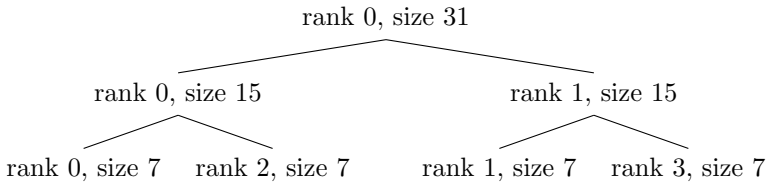


Figure 2: Dimension Tree - at each position the receiving rank and the receive buffer size is displayed

3 Implementation

3.1 MPI

As mention before we can divide the structure of the code in two sections: first the one in which the data set is split and send to each processor following the scheme at figure 1 which we will refer to as *send section*, then a second one in which each processor receive the tree computed by its child and merge it to his tree, which we will refer to as *send-back section*.
The first issue that need to be address regard the fact that memory is not shared among the processes in the MPI interface. For this reason in order to have an implementation of the tree independent from pointers and specific memory addresses, we allocated the data structure in a vector such as that each node store its children as the indexes of the vector at which they're stored.

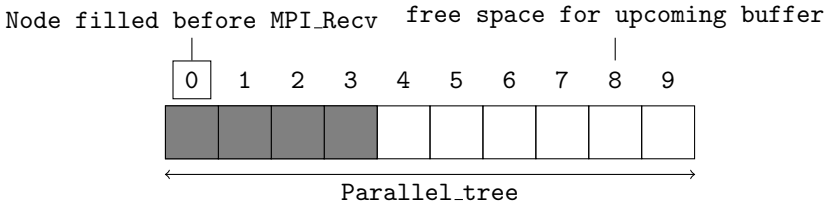
3.1.1 Dimension Tree

A critical information required many times in program regard the size of the buffer exchanged during the communication of two processes.An obvious solution to the problem could be requiring the sender process to send preemptively a message with the size of the buffer that is going to send but this would mean doubling the communications among workers affecting the overall performances due to their overhead. For this reason before the actual computation and message-passages there's a preparatory phase in which we compute a binary tree associated to the communication scheme displayed earlier, in which each node stores the dimension of the message received by the processor associated to the corresponding node in the communication tree. As we can see at figure 2

3.1.2 Receiving Buffer

The dimension tree become very handy also in the *send-back section* in which each processor has to store the tree received from the child. At the beginning of the program each processor will instantiate a vector *parallel_tree* of size equal to the size of the buffer that it is going to receive in the *send section*, and then it

will store there the tree that it have computed serially. The *parallel_tree* will be used also to allocate directly the receiving buffer during the *send-back section* so that each time a process will receive a vector it will append in *parallel_tree* filling the free spaces, in this way we will avoid the employment of temporary vectors and redundant copy assignements.



In order to perform this operation we will need to know the **offset** from the beginning of the vector to pass to the MPI_Recv call in which we can add the received tree but this is equal to the size of the buffer sent during *send section* i.e. the value stored in the dimension tree at position of the child of the receiving rank.

3.1.3 Merging

The third issue regards updating indexes of left and right children of the tree nodes. As I said before the pointers were substituted with indexes which depend though on the size of the array and as during the merge the received tree is placed next to the previously computed tree all the indexes of left and right children of each nodes need to be shifted by value equal to the previously mentioned **offset**. Obviously this shifting operation can be performed each time a processor receive a tree since at that point we already know both the offset and the dimension of the received array. However this would lead to update some nodes multiple times (more specifically each time that node happens to be sent to another process). In order to avoid this redundant operations we implemented an updating routine which follows this idea: during the construction of the dimension tree we create a vector of "strides" in which each position at the ith-element contains the the nodes computed by ith rank added to the final tree, then we use these information for updating the indexes just once when the all tree has been received by rank 0. A more detailed description:

Algorithm 3: Updating Indexes

```
Data: list of nodes nodesList, list of stride strideList,int
      number_of_processes
stride ← 0
for i ← 0 to number_of_processes do
    stride + = strideList[i]
    r ← strideList[i + 1]
    for k ← 0 to stride +r do
        nodesList[k].right + = stride
    end
end
```

4 Performance Model and Scaling

In this section we will present and analyse the benchmarks performed with the MPI and OpenMP implementations. All the tests have been runned on

- THIN node: Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz
- GPU: NVIDIA® Tesla® V100 Tensor Core

THIN nodes and GPU nodes of Orfeo Data center while the distribution employed for parallelization was OpenMPI 4.0.3. As regards the input we wrote for the purpose a routine which generate a toy data set drawn from a uniform distribution of points in a 2-dimensional space.

4.1 Performance

4.1.1 MPI

In this section we will try to give a coarse estimation of the MPI program, further and deeper analysis is definitely required to compute tighter bounds. A possible formalization of the complexity could be:

$$\begin{aligned} T(P,N) = \text{dimension\_tree}(P) + \text{parallel\_comp}(P,N) \\ +\text{serial}(P,N) + \text{updating\_ind}(P,N) + \text{communication}(P,N) \\ +searching\_dim(P) \end{aligned} \tag{1}$$

In the following formulas we will use $log()$ instead of $log_2()$

- dimension_tree(P): it regards the complexity of the building of the dimension tree. The function iterates $2 \cdot P$ times and in each iteration it performs a constant number of operations that we will define α. So the overall complexity is: $\alpha \cdot 2P$
- parallel_comp(P,N): The work performed by each workers during the *send* phase. The complexity of std::nth_element routine is $nlog(n)$, then it follows:

$$\sum_{i=0}^{log(P)-1} log(\frac{N}{2^i})\frac{N}{2^i} = \sum_{i=0}^{log(P)-1} (log(N) - i)\frac{N}{2^i} \tag{2}$$

Being $P < N$ and $N < 2$ (trivial case) we can show that $\exists \alpha \in \mathbb{R}$ so that:

$$log(N) - i \leq \alpha \leq log(N) \forall i \in [0, log(P)] \tag{3}$$

so we can conclude:

$$\sum_{i=0}^{log(P)-1} log(\frac{N}{2^i})\frac{N}{2^i} \leq \alpha 2Nlog(N)(1 - \frac{1}{2^{log(P)}}) \tag{4}$$

- serial(P,N): Represent the complexity of compute the remaining part of the dataset once the *send* phase is completed. Since at that point each processor will have to compute $\frac{N}{2^{log(P)}}$ elements the complexity will be: $O(\frac{N}{2^{log(P)}} log(\frac{N}{2^{log(P)}}))$.
- updating_ind(P,N): cost of updating indexes, it's simply consist in scanning the all tree an adding an offset so the complexity will be: $O(N)$
- communication(P,N): time for sending and receiving a buffer. If we define as $\psi(N)$ the cost of exchanging a buffer of size N between two processors then cost of the communication instances both in the send and in the receiving part will be:

$$2\psi(\frac{N}{2})log(P) \tag{5}$$

- searching_dim(P): The cost of searching the send and receive size of the buffer. Each time a process need to retrieve this information it will simply scan the level of the tree corresponding to the respective layer in the communication tree so in the worst case the cost of the search is: $O(P)$

4.2 OpenMP

The OpenMP complexity is pretty much similar to the MPI one with the difference that there's no communication stages and that the number of processes is not bounded to be a power of two: A possible formulation of the computation could be: Being $K = \lfloor log(N) \rfloor$ and $M = P - K$

$$\begin{aligned} T(P,N) = &\text{parallel\_comp}(K,N) \\ &+ omp(P,N) + \text{serial}(P,N) \end{aligned} \tag{6}$$

- serial(P,N): In this case we can employ the remaining M workers to other M tasks. The cost of compute serially the data set further divided with the free nodes is: $O(\frac{N}{2^{log(K+1)}} log(\frac{N}{2^{log(K+1)}}))$ and the cost work performed by the remaining nodes is $\frac{N}{2^{log(K)}} log(\frac{N}{2^{log(K)}})$. Thus a possible way of expressing the cost could be:

$$\beta(P/M)\frac{N}{2^{log(K+1)}} log(\frac{N}{2^{log(K+1)}})) + \gamma(P/K)\frac{N}{2^{log(K)}} log(\frac{N}{2^{log(K)}}) \qquad (7)$$

  with $\beta, \gamma \in \mathbb{R}$

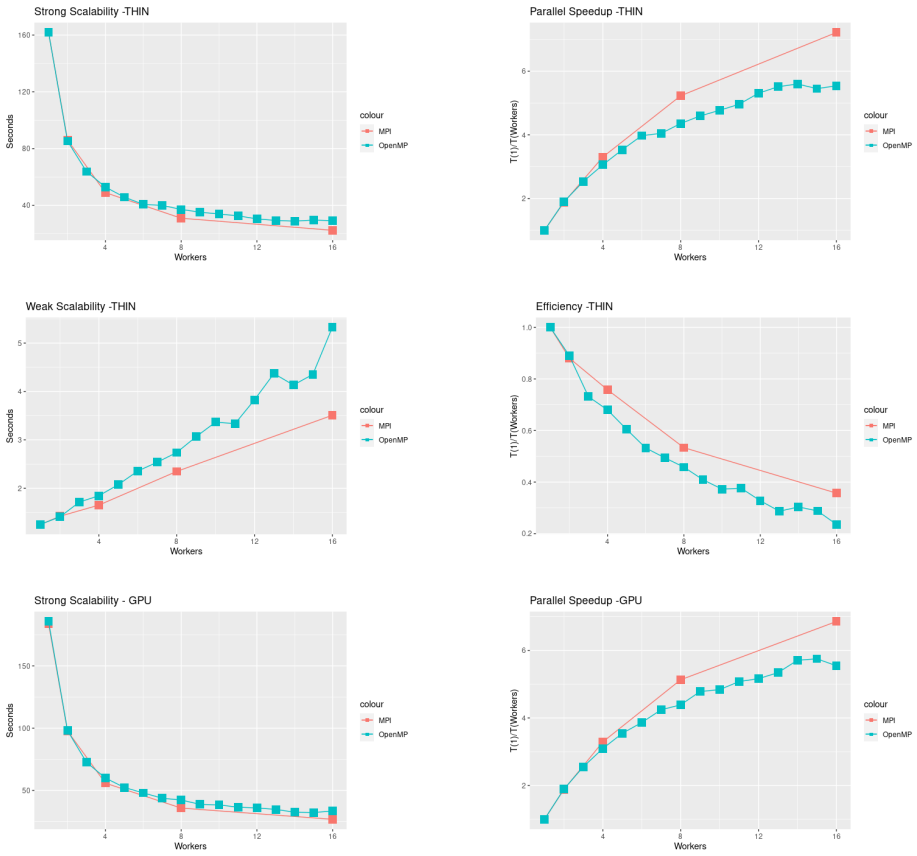
- omp(P,N): finally as we will see in the performances plot the omp implementation complexity it's not just equal as the MPI one minus the cost of the communication but there's an additional cost due to overhead.

4.3 Strong Scalability

The benchmarks for testing the strong scalability of the program were performed using $N = 10^8$ points over $k = 2$ dimension and using 2,4,8,16 processes for MPI and varying from 1 to 16 the number of OpenMP threads. As we can see in figure 4.4 the excution times falls quickly up to 4 processor, after that, the decrease slows down and adding more parallel workers doesn't significantly improve the performances. Most likely 4 processors reduce the amount of parallel work to a negligible amount so that the serial part of the job dominates and we're close to the *Amdahl's law* bound. Computations that may affect the serial part of the total job could be the overhead caused by instantiating processes and their communication, besides before all of the process has received a chunck of the dataset there's a significant work imbalances as many processes are idle waiting for input data. From the plot we can see that MPI is slightly faster than the OpenMP implementation

4.4 Weak Scaling

As regards weak scaling we fixed an amount of $N = 10^6$ points for each processor varying the number of workers as we did in the strong scaling. As we can see from the plot at figure 4.5 the problem is far from perfect scaling and this suggests us that the there's a significant increase of serial work caused by an increase of the overall problem. One of the cause of this behaviour could be the work imbalance mentioned before, as matter of fact the serial amount of work that each processor has to perform during the communication phase increase quickly with the input size. At any rate the results aren't surprising as poor weak scaling often occurs in context of memory bound problem and indeed the program need to access multiple time the data set vector for reading and sorting operation. We then see again how MPI achieve a slightly better efficiency than the OpenMP implementation.

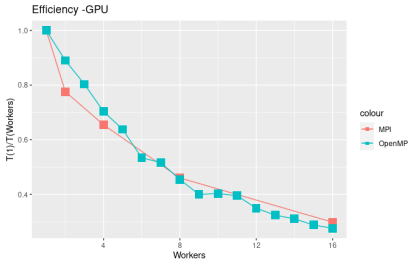


4.5 GPU

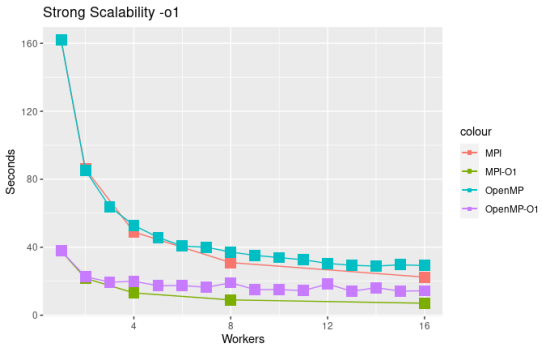
We repeated the previous evaluation on GPU nodes in which hyperthreading has been enabled. We noticed that the timings with the serial implementation were actually faster however as we added more workers the performances of THIN nodes were slightly better. It's also interesting to point out that scaling curve on GPU node was less steep than the one on THIN nodes. In addition an other significant difference with the measurements provided earlier regards the gap between MPI and OpenMP performances. On thin node MPI dominated over OpenMP but on GPU the behaviour of these two framework are almost identical.

4.6 Compiler Optimization

The compiler employed for the previous test is g++ (GCC) 4.8.5, which provide a series of levels of optimization exploiting different techniques such as Loop Optimization, Peephole optimizations etc. The level 1 optimization obtained with flag -o1 yielded much better performances but any higher level didn't



achieve any further improvement so we omit plots regarding those tests.



5 Discussion

5.0.1 OpenMP

The OpenMP framework employ *threads* as parallel workers which can access to a shared memory, so this allowed us two different way to store the kd-tree: one is the vector implementation presented in the MPI section, and the second is building the tree through pointers allocated by each thread in the heap. The former has some drawbacks as can potentially lead to a false-sharing situation. Since the vector tree is shared among the workers this implies that each thread will have to write into a shared container and simultaneous updates of individual elements in the same cache line coming from different processors will invalidates entire cache lines causing a constant re-flushing operations. We tried to fix the problem employing the second tree implementation, in this way don't enforce data locality as we would with a normal std::vector of nodes, avoiding to write in concurrent memory locations. Unfortunately this implementation didn't yield any significant speed-up in the performances so eventually we ditched this implementation. From the plots we can see that OpenMP performed slightly worse than MPI, it's hard to detect the real cause but still it's safe to guess that some false-sharing actually happened and neither of those two implementation described above were able to fix the issue.

5.0.2 MPI

We tried to reduce some of the communication cost through the employment of auxiliary data structure such as dimension_tree we then implemented also the program in which program exchange the sizes of the buffers in order to test whether there was some actual gain or not. As a matter of fact we didn't detect any significant improvement and perhaps a significantly large number of processes should be employed to detect an improvement.

6 Conclusion and Further Improvement

Probably one of the biggest limitations of the MPI program regards the constraints over the number of processes that can be used. A possible solution could keep the same structure already implemented and grow a further level in the communication tree using the remaining nodes. This will obviously lead to an unbalanced tree i.e. unbalanced workload and it's likely that we would still have peak performances for a number of workers which is a multiple of 2.
The structure of the MPI program is fairly complicated and perhaps we could achieve satisfying results simply assigning to each processor a partition of the initial space to build a tree upon and then merging all the trees in final tree. In this way we would not obtained exactly the same kd-tree built by our program but maybe the results could be still satisfying for some specific applications.
Finally as OpenMP is concerned the main issue to solve now regards assessing whether there's false-sharing or not and how to fix it. Probably we should design better memory access patterns so that different threads doesn't access to often contiguous memory addressed.