Matteo Scorcia

University of Trieste, Foundations of High-Performance Computing

All the code can be found at: https://github.com/MatteoScorcia/High-Performance-Computing-Assignments/tree/main/assignment2

ASSIGMENT 2

INTRODUCTION

Introduction of the problem to solve.

PARALLEL ALGORITHM

Abstract description of the algorithm that solves the problem.

IMPLEMENTATION

High-level description of the implementation of the algorithm.

PERFORMANCE MODEL AND SCALING

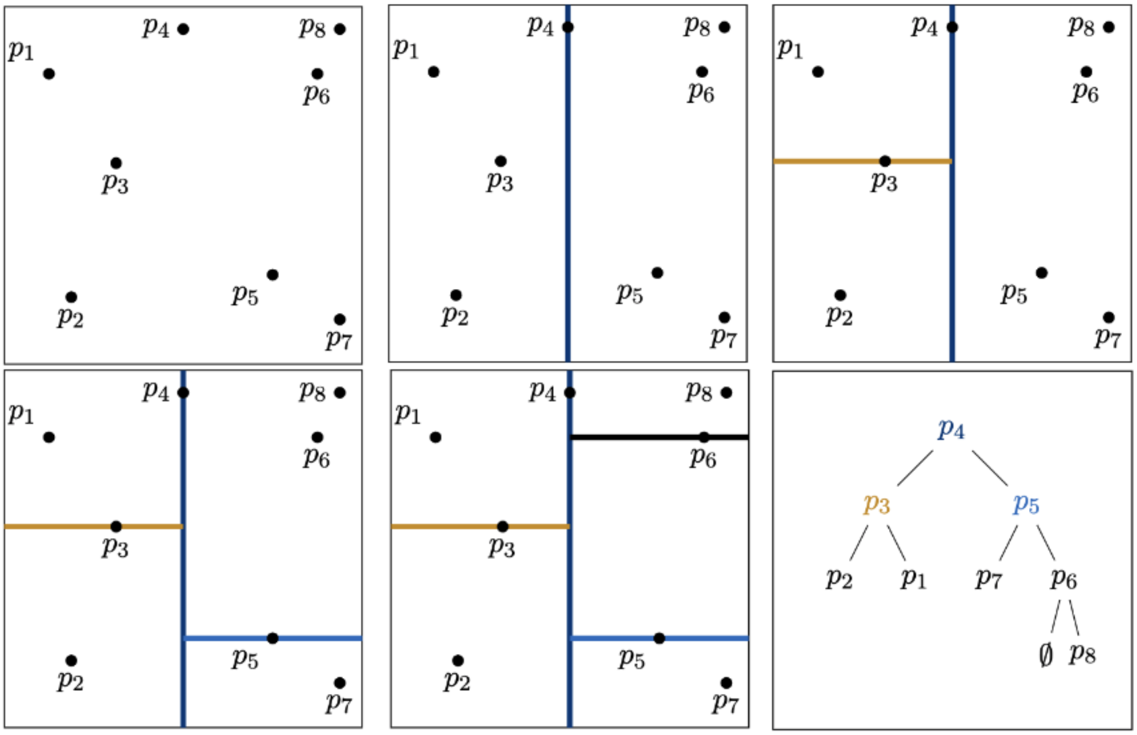
Discussion about the performance model and the strong/weak scaling measured.

CONCLUSIONS

Final discussion.

INTRODUCTION: BUILD A KD-TREE FOR 2-DIMENSIONAL DATA

Given a dataset of 2-dimensional arrays where each entry is a floating point; write a parallel code to efficiently build a kd-tree from the dataset provided.



PARALLEL ALGORITHM: ABSTRACT DESCRIPTION

The paradigm used to parallelize the code is the task paradigm of OpenMP and the send/receive paradigm of Open MPI.
We assume to have #threads and #processes that are a power of 2.

Given a number of nodes and a number of threads per node requested for the computation of the program on ORFEO:
Each node runs only 1 MPI process, which spawn a designated number of threads that will work under the openMP directives.

The sort used is the parallel quicksort presented during the lectures adjusted to be used with an array of pointers instead of the actual dataset.

First, there is a search of the most spread dimension in the dataset, followed by a sort of the dataset according to the dimension just found.

Then the "recursive" building of the KD-Tree starts on MPI process 0 until we reach the height in the KD-Tree needed to scatter the remaining dataset to every other MPI process that is waiting to start his own building of the KD-Tree.

The needed height obviously is $log_2(\#MPI\ processes)$

Every time that there is a call to the Build KD-Tree method (after the scattering), the call is queued in the task queue of the MPI process to be parallelized by OpenMP.

The only code that doesn't queue a task is the research of the most spread dimension in the dataset.
It's just a parallelized **for** that search for the min and max in a dimension.

IMPLEMENTATION: HIGH-LEVEL DESCRIPTION

```
- Retrieve dataset d
- Search the most spread dimension in the dataset d (parallel action)
- Sort the dataset according to the dimension d (parallel action)
- Final Height = $log_2(\#MPI\ processes)$
- height = 0
- if(MPI process == 0)
      Build KD-Tree until final height(d, height)
- else
    o Wait until receive dataset d
    o Build KD-Tree(d)
```

```
- Build KD-Tree until final height(dataset, height)
- If (height == Final Height)
      Scatter dataset to a free MPI process
- Search the most spread dimension in the dataset d
- Sort the dataset according to the dimension d (parallel action)
- Take the median point of the dataset m
- Define left-points as
      Datapoints p where p < m according to dimension d
- Define right-points as
      Datapoints p where p > m according to dimension d
- Define left-child as
      Build KD-Tree until final height(left-points, height+1) (send this
      action to a queue of tasks)
- Define right-child as
      Build KD-Tree until final height(right-points, height+1) (send
      this action to a queue of tasks)
- Return node containing m, left-child, right-child as children
-   Build KD-Tree(dataset)
-   If length of dataset == 1 then
            Return node containing the single point of the dataset
-   Search the most spread dimension in the dataset d
-   Sort the dataset according to the dimension d (parallel action)
-   Take the median point of the dataset m
-   Define left-points as
            Datapoints p where p < m according to dimension d
-   Define right-points as
            Datapoints p where p > m according to dimension d
-   Define left-child as
            Build KD-Tree(left-points) (send this action to a queue of tasks)
-   Define right-child as
            Build KD-Tree(right-points) (send this action to a queue of tasks)
-   Return node containing m, left-child, right-child as children
```

PERFORMANCE MODEL AND SCALING

Discussion on performance model

Single Thread, Single MPI process: performance modeling

$$N \;=\; \#points\ in\ the\ dataset$$

First, there is a call of a method that finds the extremes of the dataset for each dimension:

$$2N \times \#dimensions\ operations$$

To simplify a bit the formulas, let’s assume that we are in the case of building a perfect binary tree with N points.

We know that in a perfect binary tree with N leaves:

$$height = log_2(N+1)$$

$$\#leaves = 2^{height-1} = \frac{N+1}{2}$$

$$\#nodes = N$$

Each node comes from a call of the “build kd-tree” method.

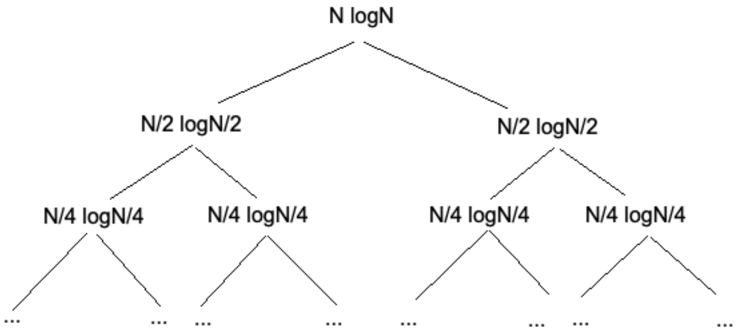
Each of these calls performs a sorting (if necessary), an allocation of a new node and 2 calls of the “build kd-tree” method.

Let’s suppose that every call needs to sort again the dataset (worst case).

In this case our execution time is bound to the Time Complexity of the quicksort! (All the other operations are negligible respect to the sorting).

$$Nlog_2N\ (average\ case\ for\ quicksort)$$

The quicksort is called N times, but the dataset size is variable depends on the current height of the kd-tree:



$$\sum_{h=0}^{h=height-1} \frac{N}{2^h} \times 2^h \times log_2\left(\frac{N}{2^h}\right) = \sum_{h=0}^{h=height-1} N log_2\left(\frac{N}{2^h}\right)$$

$$if \;\; N >> 1, \text{height - 1} = log_2(N+1) \; - \; 1 \;\approx log_2(N)$$

$$= \sum_{h=0}^{h=log_2(N)} N log_2\left(\frac{N}{2^h}\right)$$

$$= \mathrm{N}\, log_2\left(\frac{N^{log_2(N)}}{2^{\sum_{h=0}^{h=log2(N)} h}}\right) = \mathrm{N}\left[log_2\left(N^{log_2(N)}\right) - log_2\left(2^{\sum_{h=0}^{h=log2(N)} h}\right)\right]$$

$$= \mathrm{N}\left[log_2(N^2) - \sum_{h=0}^{h=log_2(N)} h\right] = N\left[log_2(N^2) - \left(\frac{log_2(N)\,(log_2(N)-1)}{2}\right)\right]$$

$$= N[log_2(N^2) - \frac{log_2(N^2) \; - \; log_2(N)}{2}] = \frac{N}{2} log_2(N^2) + \frac{N}{2} log_2(N)$$

$$= \frac{3}{2} N log_2(N)$$

So, our performance model is:

$$(2N \times \#dimensions) + \frac{3}{2} N log_2(N)$$

We can see that if we have a dataset that is distributed only on a single dimension, like a stripe, we are in the best case, because we need to sort the dataset only once!

The performance model in this case is:

$$(2N \times \#dimensions) + N log_2 N$$

Multi Thread, Multi process: performance modeling

$$N \;=\; \#points\ in\ the\ dataset$$

$$\#threads \;=\; \#threads\ per\ MPI\ process$$

First, there is a call int the root MPI process of a method that finds the extremes of the dataset for each dimension in a parallel manner:

$$\frac{N}{\#threads} \times \; \#dimensions$$

Before the scattering on the other MPI processes, there are $2^{\#MPI\ processes}$ -1 calls of the "build kd-tree" method performed on the root MPI process.

Each of these calls performs a sorting (if necessary), an allocation of a new node and 2 calls of the "build kd-tree" method.

Let's suppose that every call needs to sort again the dataset (worst case).

In this case our execution time is bound to the execution time of the parallel quicksort! (All the other operations are negligible respect to the sorting).

The parallel quicksort has the same performance behavior of the parallel merge sort:

$$2N + \frac{N}{\#threads} \times log \frac{N}{\#threads}$$

So, our performance model before the scatter to the MPI processes is:

$$\text{build\_root} = \sum_{h=0}^{h=\#MPI\ processes\ -\ 1} 2x + \frac{x}{\#threads} \times log_2 \frac{x}{\#threads}$$

$$where\ x = {}^{N}/_{2^h}$$

Then the scattering occurs:

Remember that every MPI process is bound to a different ORFEO node, hence we are using the InfiniBand network to send each chunk of the dataset.

$$chunk\ size\ =\ {}^{N}/_{\#MPI\ processes}$$

$$\lambda_{node}\ =\ infiniBand\ latency\ between\ 2\ nodes$$

$$b_{ucx,node}\ =\ infiniBand\ bandwidth$$

$$T_{\text{comm}} = \left[\lambda_{node} + \frac{chunk\ size}{b_{ucx,node}}\right] \times (\#MPI\ processes\ -\ 1)$$

After the scattering, for each MPI process:

$$chunk\ size\ =\ {}^{N}/_{\#MPI\ processes}$$

$$final\ height\ = log_2(N+1)\ -\ 1\ \approx log_2(N)\,, \qquad N >> 1$$

$$\text{build\_chunk} = \sum_{h=\#MPI\ processes}^{h=final\ height} 2x + \frac{x}{\#threads} \times log_2 \frac{x}{\#threads}$$

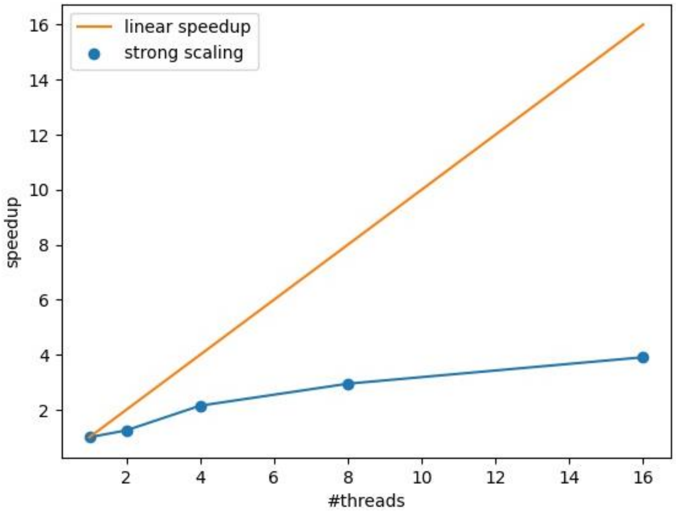
$$where\ x = {}^{chunk\ size}/_{2^h}$$

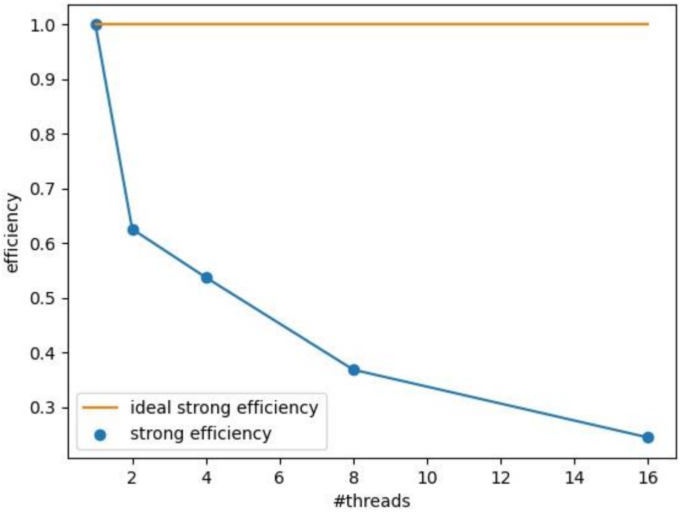
The performance model is:

$$\boldsymbol{build\_root + T_{\text{comm}}\ + (build\_chunk \times \#MPI\ processes)}$$

Discussion on openMP scalability

Disclaimer: all the measurements have been made with a fixed number of MPI processes.





As we can see, the efficiency drops drastically after a few threads.

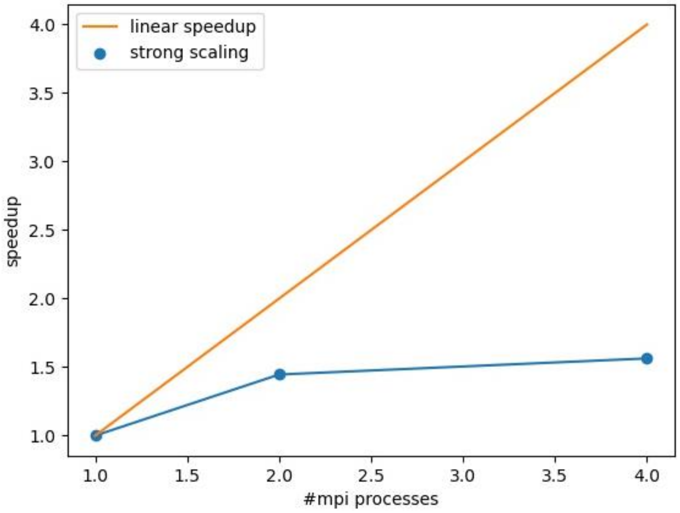
We can't fully exploit the threads due to the nature of the problem.

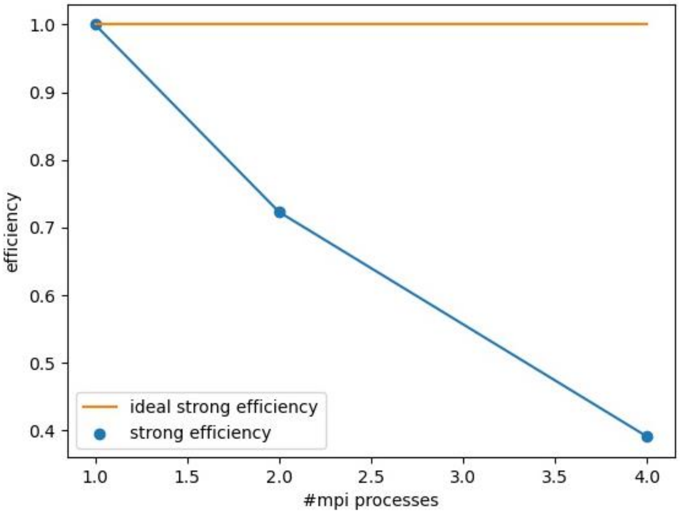
As we saw from the merge sort performance analysis, there are intrinsically serial operations that saturates the speed-up.

We can state that the algorithm does its job when we have less than 8 threads per MPI process, but beyond that point, is just a waste of resources.

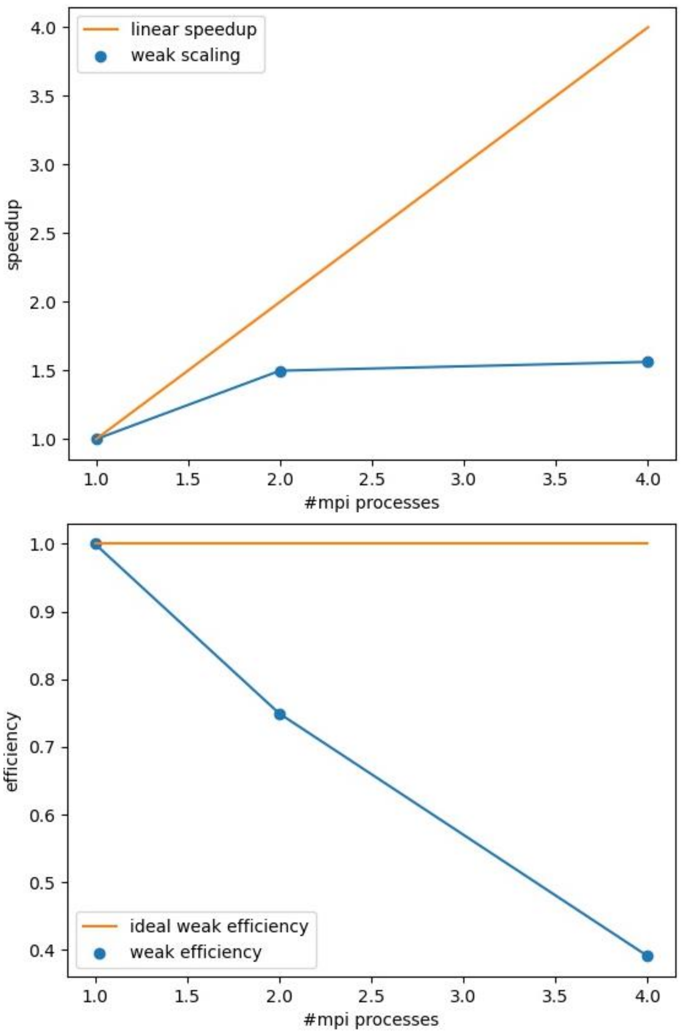
Discussion on strong/weak scalability on MPI

Disclaimer: all the measurements have been made with a fixed number of threads.





As we can see, with a fixed dataset size, there is a precise number of MPI processes scale well, after that number we are just wasting resources.



As we can see, varying the dataset size doesn't seem to make any improvement regarding the scalability.

CONCLUSIONS

We must always sort our dataset on a single MPI process before the scattering to other MPI processes.

This means that as the dataset size increases, there is an increasing idle time of the MPI processes!

This is bad news, because we can't work on a dataset "too big", all the burden is on the single MPI root process for the first (and obviously the heaviest) sort iterations.

The suggested approach with this code, is to scatter the dataset to the MPI process the earliest possible to minimize the idle time of the other MPI processes.

This led to the following question:

Do we really need the hybrid approach given the fact that we should use just a few MPI processes?

My answer is: it depends.

If the dataset isn't so big, then we should use just one node, because even if the scattering time is negligible, if we use a few nodes, we are instantiating a lot of resources when the actual runtime is always bound by the MPI root process.

I think that the sweet spot for this hybrid code is when we must use more than 8 threads on a single node to achieve a good runtime; reserving more threads is just not efficient:

It is better to reserve another node to work with half of the dataset.

In this case we have a better scaling, because we are always on the sweet spot of 8 threads per node without the dramatic drop in efficiency.