Report of Assignment 02 - Foundations of High-Performance Computing

Author: Debarshi Banerjee
Course: MHPC
Affiliation: ICTP and SISSA
Session: 2021/22

Introduction

The goal of this assignment is to build a k-dimensional tree, for `k=2`, which is basically a binary tree. We must do this task in a parallel approach, using MPI and/or OpenMP, either separately (as I have done), or in a hybrid code. We assume, for the sake of simplicity that the dataset is immutable, and that data points are homogenously distributed across all k-dimensions. Also, we can assume the number of processes being used in the parallel approach to be a power of 2.

Running the program and explanation of directory structure

We have 2 folders, one for MPI and the other for OpenMP. They have a `kdtree_mpi.c` and `kdtree_omp.c` files respectively. There are associated Makefiles, some `gnuplot` scripts for plotting relevant data, and the `scaling.sh` script which was used to compile, and run the program for all the relevant runs. The OpenMP version also has an interactive `run_interactive.sh` script which explains various relevant flags. The `data` folder has all the relevant data from all the runs. The `plots` folder has some `*.dat` files for strong scaling, weak scaling, and the speedup comparisons, time comparisons, splitting point choice comparisons etc., as an average of all the data. There are also some generated graphs and their correspondng `*.gp` scripts, which are used here in this report as well.
To compile either program, load the relevant modules (`gnu`, `openmpi`), and type `make`. To run them, do the following (asking for the desired number of points instead of 100 and for desired number of processes/threads instead of 4 {`--oversubscribe` may be needed for MPI}):

```
#MPI
mpirun -np 4 ./kdtree_mpi.x 100

#OpenMP
export OMP_NUM_THREADS = 4
./kdtree_omp.x 100
```

Algorithm

OpenMP

- The basic idea is a recursive approach.
- At the beginning, a dataset is generated by random values.
- At each recursion, since the data is balanced along all dimensions, the choice for axis of split was chosen via round-robin method.
- The splitting point is chosen via the 'QuickSelect' algorithm. This chooses the k-th smallest element in an unordered list. It also partially sorts the data. There is further explanation later on.
- As we see, in practice, quickselect is much faster than quicksort, and clearly the better choice.
- We take the splitting point as the new node of the tree. The first half, prior to the splitting point is used to build the left branch of the tree from the new node onwards, and the second half is used to build the right branch from the new node.
- Since these 2 tasks are independent of each other, they are done using separate OpenMP `tasks`, so that idle threads take them up respectively.
- This process is recursive, so after a few rounds, we generate many more `tasks` than we have threads, and actually create a very unfortunate bottleneck.
- To resolve this partially, we use a pre-specified maximum allowed depth of OpenMP recursion (5, since, 2^5 = 32 = realistic maximum free threads on offer at any given time). Once this depth is reached, we call the serial versions of the corresponding recursive functions, instead of generating more `tasks`. Moreover, this whole process is wrapped up in a `#pragma omp taskgroup` directive, since `taskgroup` guarantees the completion of child tasks.
- In fact, we found that using `taskgroup` is slightly faster, than without it, and also, using the serial versions of the recursive functions explicitly instead of letting a very large number of threads be generated, can be upto 20% faster for our 10^8 size dataset.

MPI

- At the beginning, a dataset is generated by random values only on the ROOT/MASTER (rank = 0) processor.
- Then, the ROOT processor starts building the tree, involving a distributed recursive approach. What this means is that it will both start building a section of the tree by itself, while it also prepares to send data to other processors, who are waiting to start their workload.
- The waiting processors receive data from the ROOT processor, and call the main `build_kdtree()` function, and then proceed with building the tree as well.
- As described in the 'Implementation' section later, there is a choice of the splitting axis in a round robin fashion, and also the splitting point is chosen using the Quick Select algorithm. The virtues of this choice are explained.
- We take the splitting point as the new node of the tree.
- There is a depth parameter here, which is basically designed to ensure that as long as we have further available processors, that is the following condition is fully satisfied - $depth\_of\_tree < log_2(number\_of\_processors)$, we send the right branch of the dataset (i.e., the section AFTER the splitting point), along with corresponding information about the new depth (

$new\_depth = depth + 1$), the axis of splitting, and the number of points, to an available processor that is waiting to start building its own section of the tree.

- The left branch (or the section BEFORE the splitting point), is continued to be built by the existing processor.
- Once we reach the maximum depth limitation, and the process effectively becomes serialized since now there are no more freely available processors, and each processor continues recursively building its own tree section, both the left and the right branches.
- Further details are given in the implementation section below.

Implementation

Data Structure

The data structure for the kd-tree is defined as follows:

```
#define NDIM 2

#if !defined(DOUBLE_PRECISION)
    #define float_t float
#else
    #define float_t double
#endif

typedef struct kdnode kdnode;
typedef struct kpoint kpoint;

struct kpoint {
    float_t coord[NDIM];
};

struct kdnode {
    int axis;
    kpoint split;
    kdnode *left, *right;
};
```

- Every point in the kd-tree is called as a `kpoint`. They have an array of coordinates, and here their dimensionality is pre-defined = 2.
- Every node of the kd-tree is called as `kdnode` and has an axis of splitting (integer value), a splitting point (which must be of type `kpoint` for obvious reasons), and 2 children - pointers to left and right branches of the kd-tree from the current node onwards (of type `kdnode*` for obvious reasons).

Choice of splitting axis

The axis is chosen in a round-robin fashion, i.e., at each recursion the axis must be different from the previous one.

```
int choose_splitting_dimension(int axis) {
    return (axis + 1) % NDIM;
}
```

Selection of splitting point

The splitting point is chosen via the 'QuickSelect' algorithm. This chooses the k-th smallest element in an unordered list. It also partially sorts the data. Quickselect uses the same overall approach as quicksort, choosing one element as a pivot and partitioning the data in two based on the pivot, accordingly as less than or greater than the pivot. However, instead of recursing into both sides, as in quicksort, quickselect only recurses into one side – the side with the element it is searching for. This reduces the average complexity from $O(nlogn)$ to $O(n)$ with a worst case of $O(n^2)$.

```
kpoint* quickselect(kpoint* x, int N, int k, int axis) {
    int i, st;
    for (st = i = 0; i < N - 1; i++) {
        if (x[i].coord[axis] > x[N - 1].coord[axis])
            continue;
        swap(x + i, x + st);
        st++;
    }
    swap(x + N - 1, x + st);
    return k == st
               ? x + st
               : (st > k ? quickselect(x, st, k, axis) : quickselect(x + st,
N - st, k - st, axis));
}
```

OpenMP overview

The general approach is explained in the algorithms section. We call the `build_kdtree()` function recursively, and each corresponding call builds the left and the right branches of the tree in separate OpenMP tasks recursively. This is done till a certain maximum depth is reached. For better performance, since it guarantees completion of child tasks, `taskgroup` directive is also used.

```
int main(int argc, char** argv) {
...
    kpoint* set = (kpoint*)malloc(sizeof(kpoint) * n);
    kpoint_initialize(set, n);
    kdnode* final_nodes;
#pragma omp parallel
    {
```

```
#pragma omp single nowait
        {
#pragma omp taskgroup
            { final_nodes = build_kdtree(set, n, 0, 0); }
        }
#pragma omp taskwait
    }
    double tend = omp_get_wtime();
...
    return 0;
}
```

```
kdnode* build_kdtree(kpoint* x, int N, int axis, int depth) {
...
...
...

    if (depth >= MAX_DEPTH) {
        node->left = build_kdtree_serial(left_points, left_size, myaxis,
depth);
        node->right = build_kdtree_serial(right_points, right_size, myaxis,
depth);
        return node;
    } else {
        depth++;
#pragma omp task untied
        node->left = build_kdtree(left_points, left_size, myaxis, depth);
#pragma omp task untied
        node->right = build_kdtree(right_points, right_size, myaxis, depth);

    return node;
}
```

This process is recursive, so after a few rounds, we generate many more `tasks` than we have threads, and actually create a very unfortunate bottleneck. To resolve this partially, we use a pre-specified maximum allowed depth of OpenMP recursion (5, since, 2^5 = 32 = realistic maximum free threads on offer at any given time). Once this depth is reached, we call the serial versions of the corresponding recursive functions, instead of generating more `tasks`.

MPI overview

We have a helpful assumption that was mentioned earlier that is particularly relevant here - we can assume the number of processes being used in the parallel approach to be a power of 2.

The ROOT processor initially starts building the tree, while the others wait for their first batch of data.

```
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
...
    kpoint* set;
    kdnode* final_nodes;

    t_start = MPI_Wtime();

    if (rank == ROOT) {
        set = (kpoint*)malloc(sizeof(kpoint) * n);
        kpoint_initialize(set, n);
        final_nodes = build_kdtree(set, n, -1, 0);
    } else {
        final_nodes = init_build_kdtree();
    }

    t_end = MPI_Wtime();
...
    MPI_Finalize();
    return 0;
}
```

The processors that are waiting, call the `init_build_kdtree()` function, and once they receive their data from the ROOT, they call `build_kdtree()` which builds the tree recursively.

Meanwhile, the ROOT processor, and all other processors on the first run call `build_kdtree()`.
There is a depth parameter here, which is basically designed to ensure that as long as we have further available processors, that is the following condition is fully satisfied - $depth\_of\_tree < log_2(number\_of\_processors)$, we send the right branch of the dataset (i.e., the section AFTER the splitting point), along with corresponding information about the new depth ($new\_depth = depth + 1$), the axis of splitting, and the number of points, to an available processor that is waiting to start building its own section of the tree.
The left branch (or the section BEFORE the splitting point), is continued to be built by the existing processor.
Once we reach the maximum depth limitation and $depth\_of\_tree >= log_2(number\_of\_processors)$, and the process effectively becomes serialized since now there are no more freely available processors, and each processor continues recursively building its own tree section, both the left and the right branches.

```
kdnode* build_kdtree(kpoint* x, int N, int axis, int depth) {
    if (N == 0)
        return NULL;

...
```

```
    kpoint* split_point = quickselect(x, N, N / 2, myaxis);

    node->split = *split_point;

    kpoint* left_points = x;
    kpoint* right_points = split_point + 1;

    int left_size = split_point - left_points;
    int right_size = x + N - right_points;

    if (depth < log2(nproc)) {
        int tmp_depth = depth + 1;
        int destination_proc = get_dest_proc(rank, nproc, depth);
        MPI_Send(&myaxis, 1, MPI_INT, destination_proc, 13 * rank,
MPI_COMM_WORLD);
        MPI_Send(&tmp_depth, 1, MPI_INT, destination_proc, 15 * rank,
MPI_COMM_WORLD);
        MPI_Send(&right_size, 1, MPI_INT, destination_proc, 17 * rank,
MPI_COMM_WORLD);
        MPI_Send(right_points, right_size, MPI_kpoint, destination_proc, 19
* rank, MPI_COMM_WORLD);
        node->left = build_kdtree(left_points, left_size, myaxis, depth +
1);
    ...
    } else {
        node->left = build_kdtree(left_points, left_size, myaxis,
log2(nproc));
        node->right = build_kdtree(right_points, right_size, myaxis,
log2(nproc));
    }

    return node;
}
```

The choice of an idle processor is key. To do that we implement the following function:

```
int get_dest_proc(int rank, int nproc, int depth) {
    return rank + (nproc / pow(2, depth + 1));
}
```

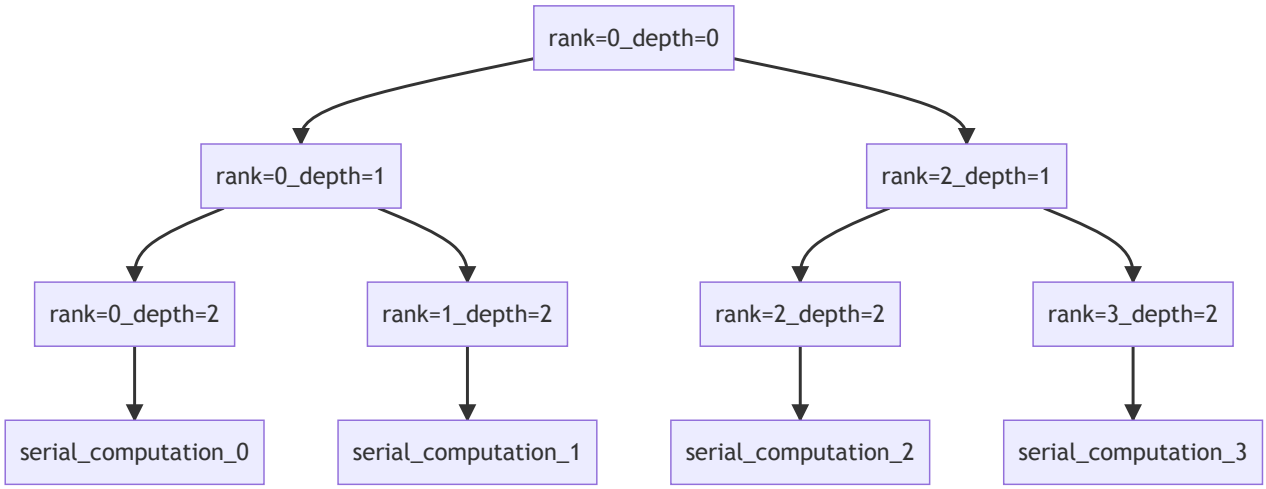
This basically follows the strategy that:

$$destination\_processor = current\_rank + \frac{total\_number\_of\_processors}{2^{depth+1}}$$

This allows us to identify existing idle processors for any given depth-level and distribute the workload as long as we have available processors. Once the depth limit is hit, we end up serializing the computation.

Visualization of MPI workload distribution

Let us see a visual explanation of the case of the MPI workload distribution using the above described schema. Let's consider the case of using 4 MPI processes. When $depth\_of\_tree >= log_2(number\_of\_processors)$, the process becomes serialized. Prior to that, as we can see, at each depth the left branch (dataset prior to splitting point) is processed by the existing rank, and the right branch (dataset post splitting point) is processed by the next determined rank.



Important note about `MPI_Send()` and `MPI_Recv()`

It is vital to remember that once processors apart from the ROOT start building the tree, once they start to receive their respective sections of data from the ROOT, they can all simultaneously send data to still idle processors, hence, it is important that the idle processors' `MPI_Recv()` calls use `MPI_ANY_TAG` and `MPI_ANY_SOURCE` to allow this transfer to happen without any issues.

Important note about `MPI_Datatype`

When we send the chunk of data containing the `kpoint` from the splitting point onward to a different MPI process, it is helpful to declare a new datatype so that the transfer is as smooth as possible. Below is a demonstration of how this was done and the respective `MPI_Send()` and `MPI_Recv()` calls where they were used:

```
    MPI_Datatype MPI_kpoint;
    MPI_Type_contiguous(sizeof(kpoint), MPI_BYTE, &MPI_kpoint);
    MPI_Type_commit(&MPI_kpoint);
    ...
    MPI_Send(right_points, right_size, MPI_kpoint, destination_proc, 19 *
```

```
rank, MPI_COMM_WORLD);
    ...
    MPI_Recv(local_points, local_right_size, MPI_kpoint, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, status);
```

Performance Model and Scaling

All calculations were run on a single GPU node on Orfeo cluster.
They use GCC-9.3.0 and OpenMPI-4.1.2.
All programs were written in C.
We assume that the total available number of processes/threads is a power of 2 upto $2^5 = 32$ for both OpenMP and MPI.
For strong scaling calculations, we use 10^8 points of data.
For weak scaling calculations, we keep the total workload per processor constant at 10^7 points of data per processor.

Strong Scaling Times

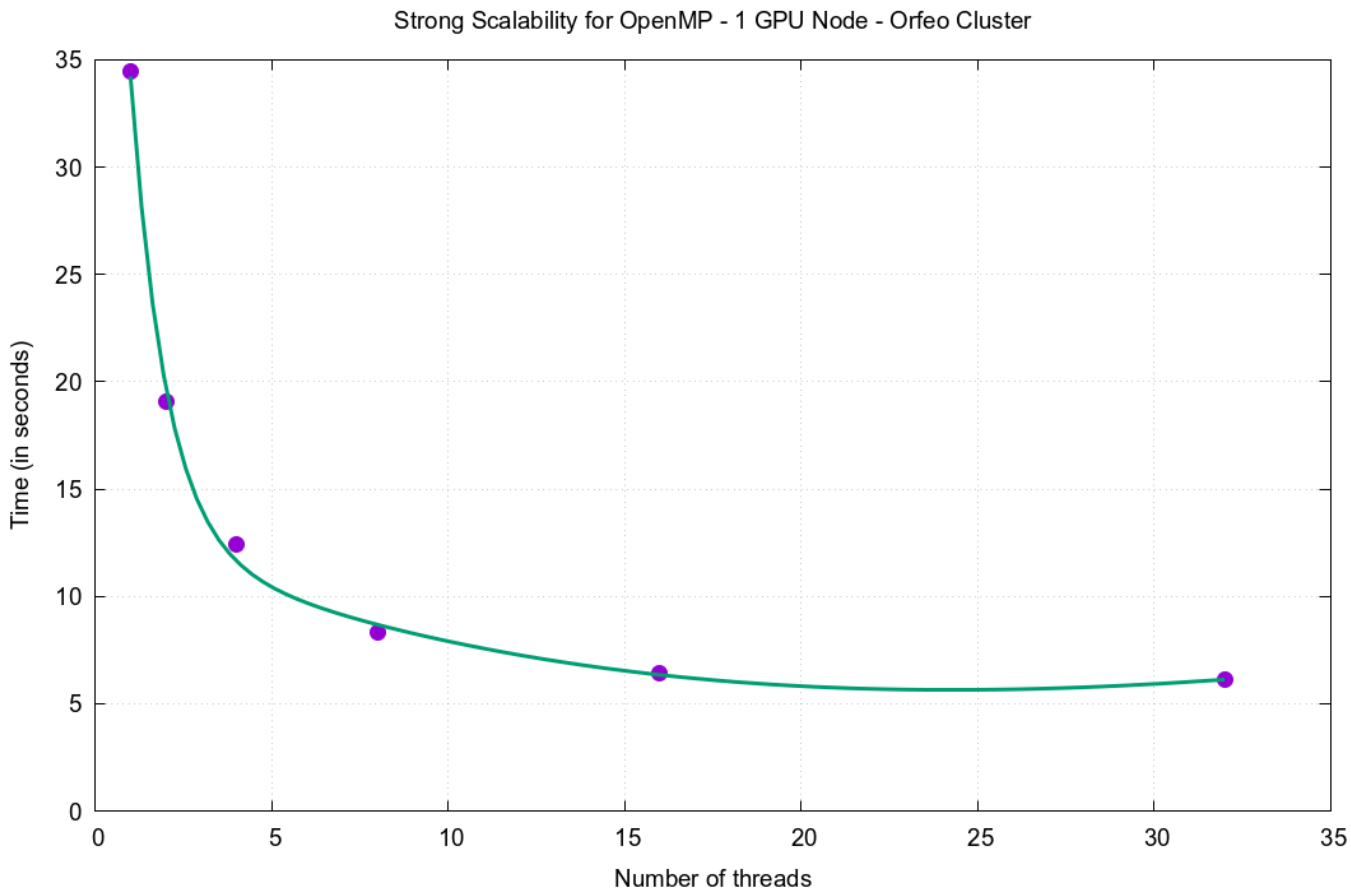
# of MPI Processors/OMP Threads	MPI times (sec)	OpenMP time (sec)	MPI Speedup - $S_P = \frac{T(1)}{T(P)}$	OpenMP Speedup - $S_P = \frac{T(1)}{T(P)}$
1	94.441	34.442	1.000	1.000
2	49.933	19.069	1.891	1.806
4	28.100	12.435	3.360	2.769
8	17.114	8.312	5.518	4.143
16	12.049	6.411	7.837	5.372
32	10.219	6.119	9.241	5.628

Weak Scaling Times

# of MPI Processors/OMP Threads	Dataset size	MPI times (sec)	OpenMP times (sec)
1	10^7	9.077	3.078
2	$2 * 10^7$	9.748	3.602
4	$4 * 10^7$	11.150	4.771
8	$8 * 10^7$	13.825	6.852
16	$16 * 10^7$	19.129	10.669
32	$32 * 10^7$	31.926	19.775

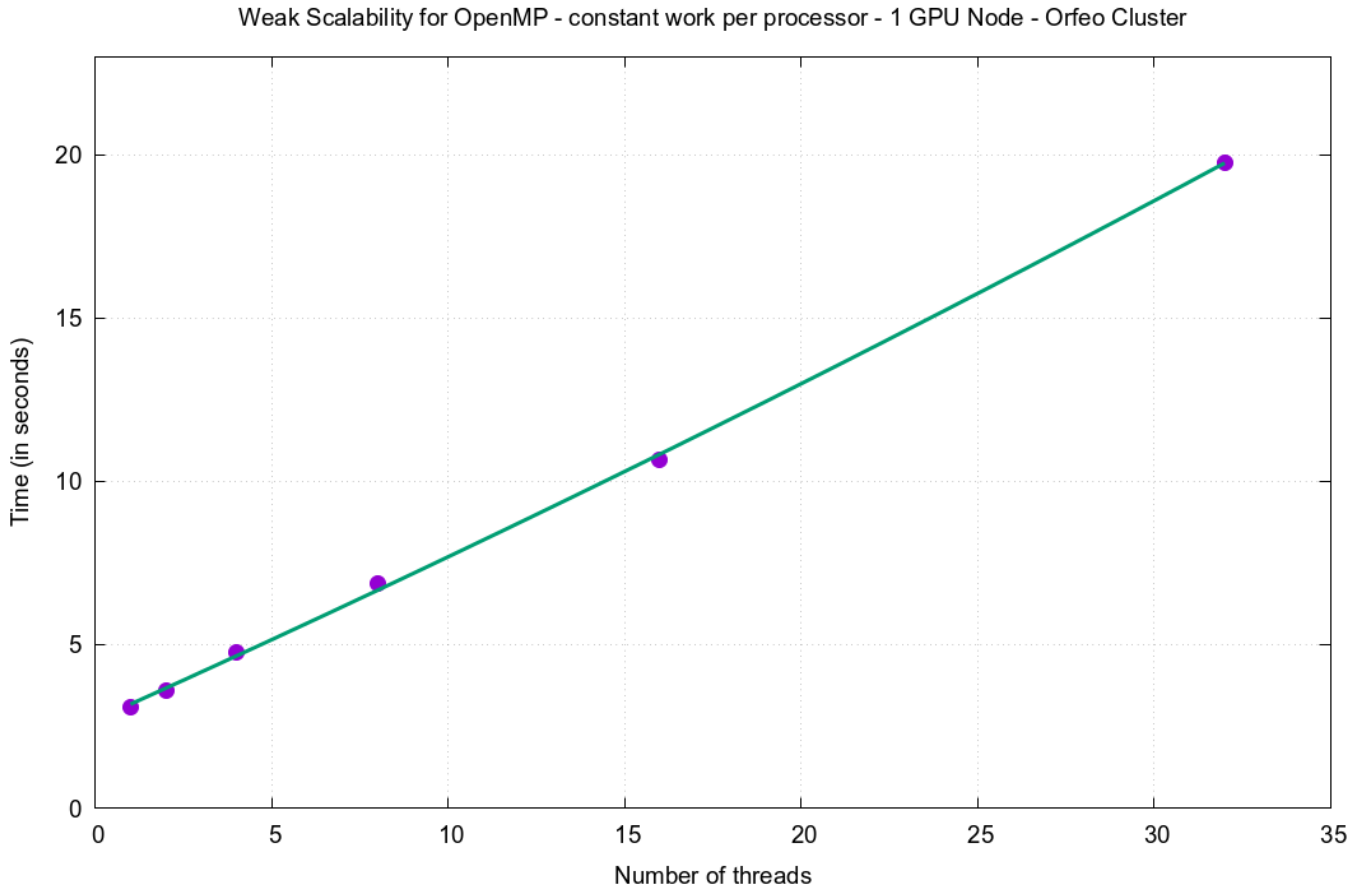
OpenMP strong scaling

Here, we see the graph of strong scaling for OpenMP. As we can observe, it becomes asymptotic after a certain point (around 8-16 threads) with only minimal decreases in time observed after that. The best time that we achieved was 6.119266 seconds for 32 threads.



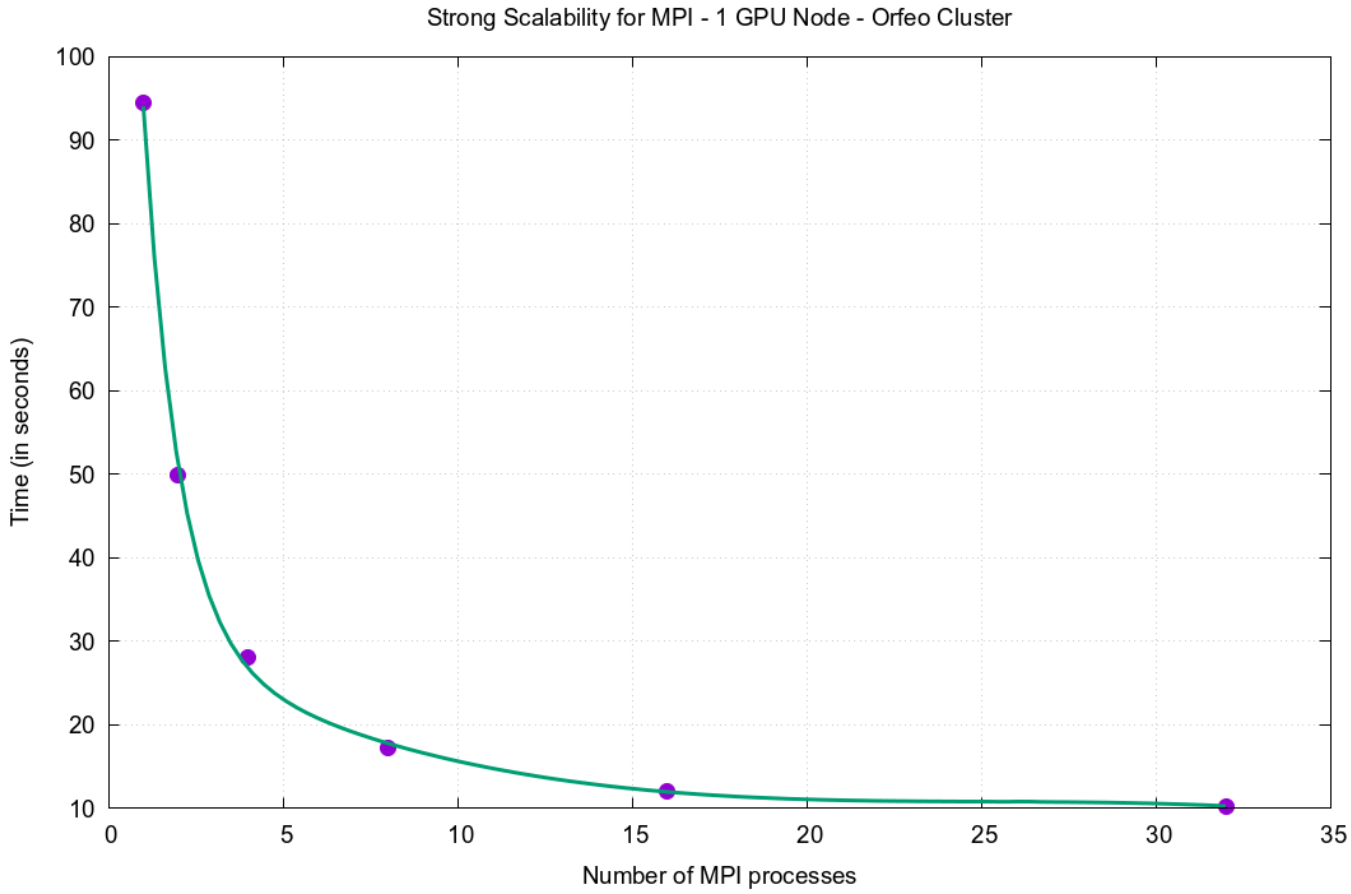
OpenMP weak scaling

Here, we see the graph of weak scaling for OpenMP.



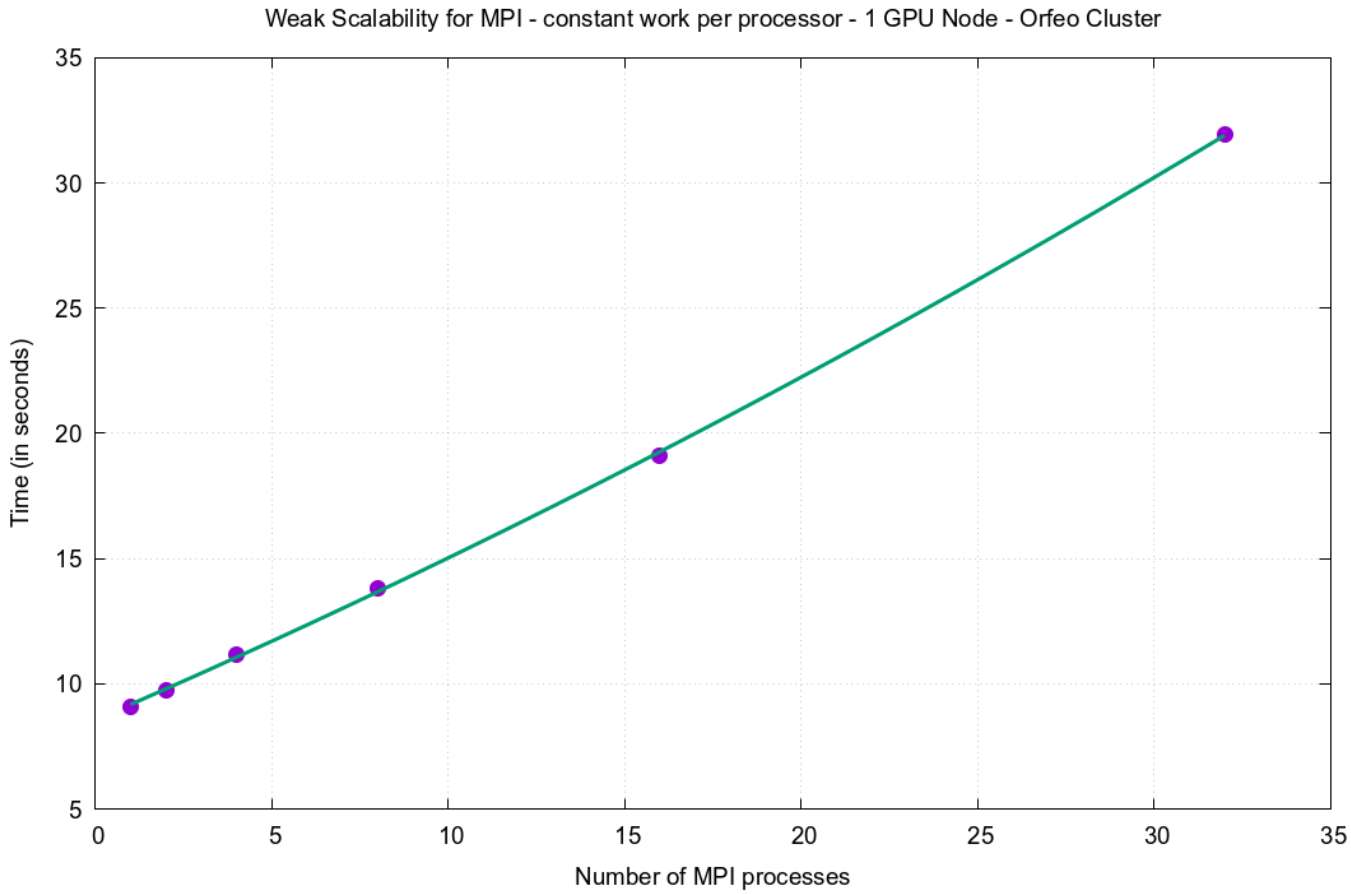
MPI strong scaling

Here, we see the graph of strong scaling for MPI. This shows a pretty significant improvement in performance initially, but like the OpenMP case, it becomes asymptotic at the end. The best obtained time is 10.219755 seconds for 32 MPI processes.



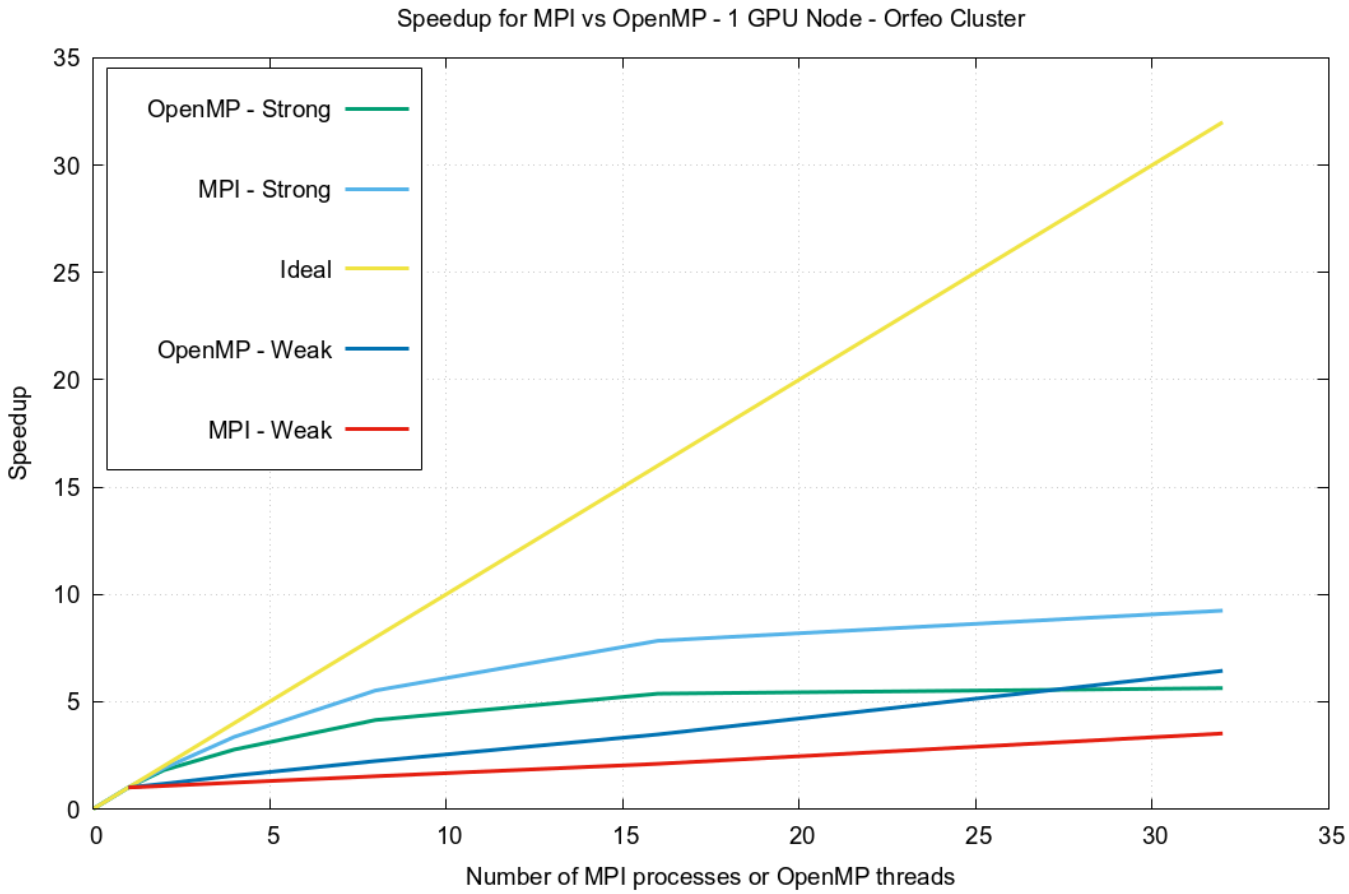
MPI weak scaling

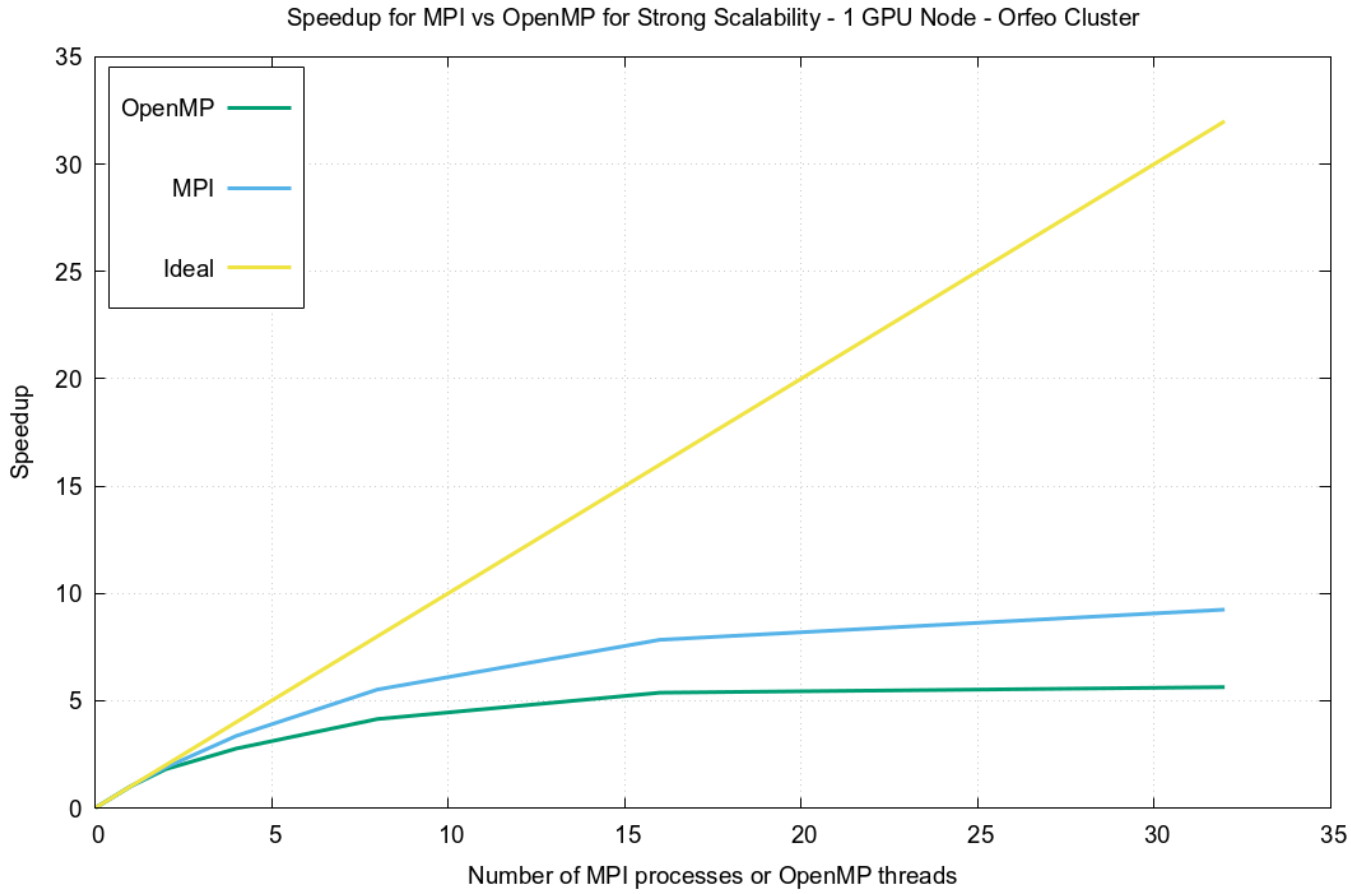
Here, we see the graph of weak scaling for MPI.

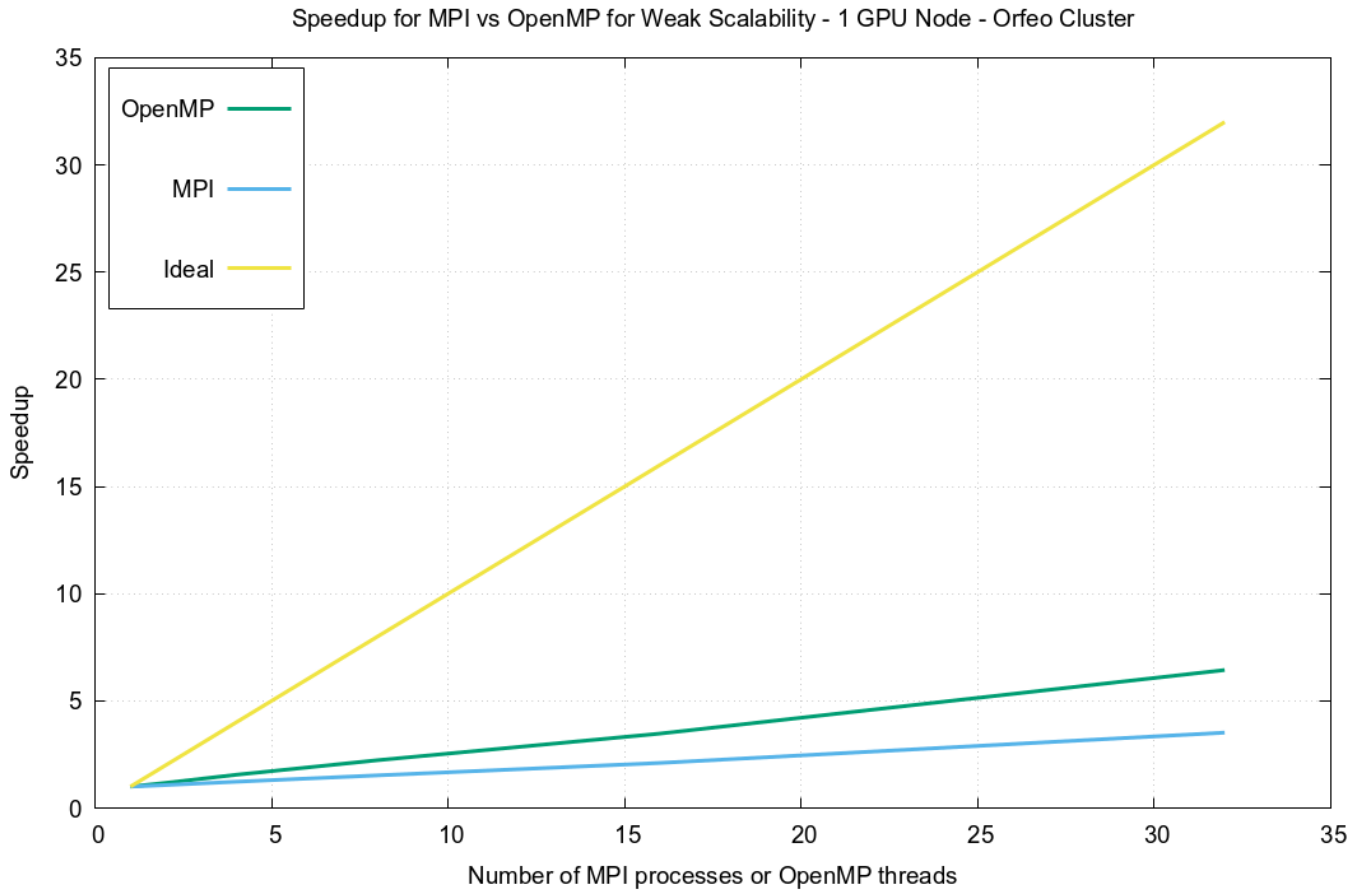


Speedup Comparison - MPI vs OpenMP

Here, we see the graph of relative speedup for the strong scaling and weak scaling condition between MPI and OpenMP. As we can observe, MPI is definitely more impacted by the increase in parallel processing power for strong scalability, however, this doesn't tell the full story. For weak scalability, OpenMP has better performance, however, the deviation from ideal conditions is incredibly high.



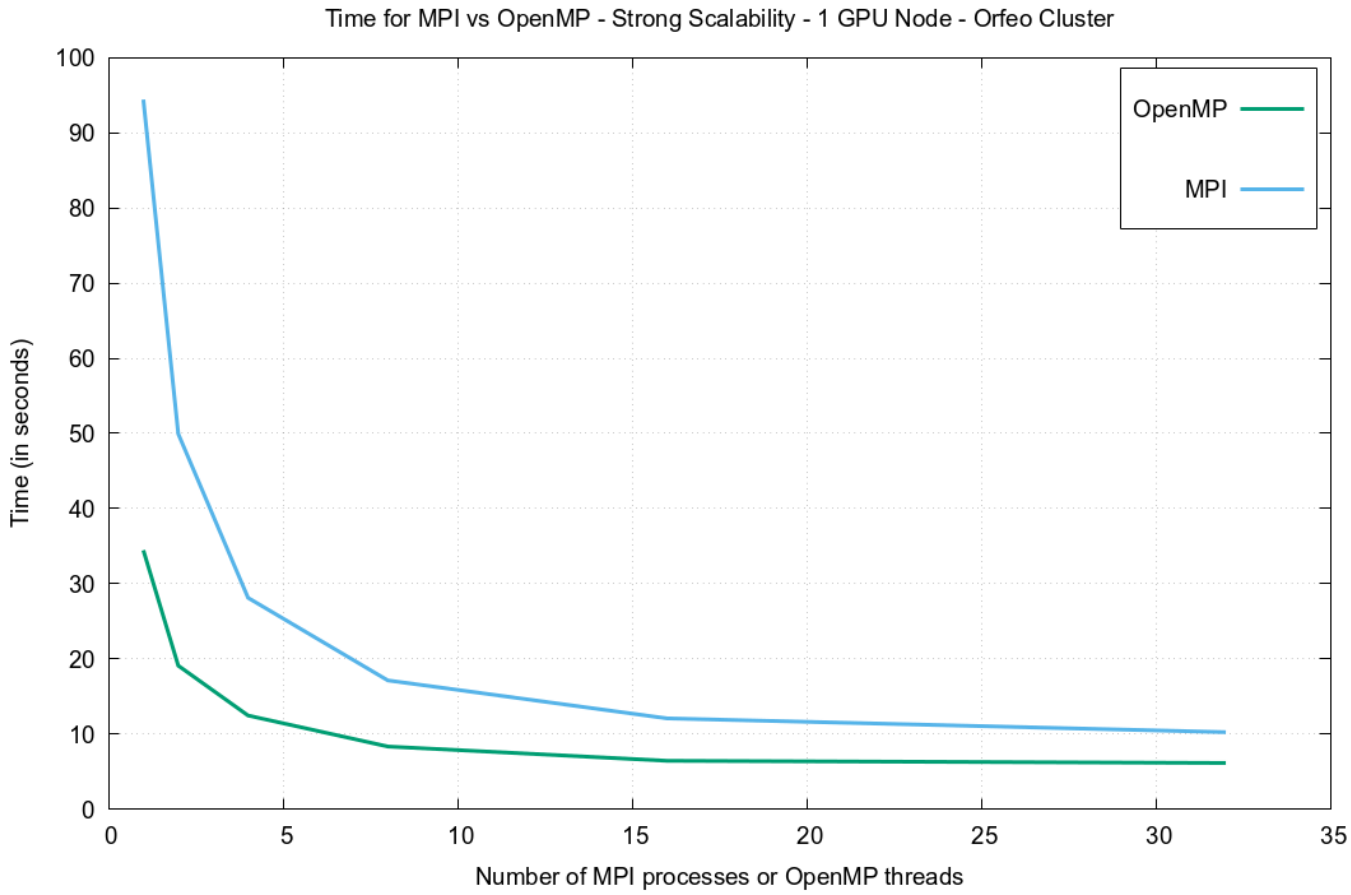


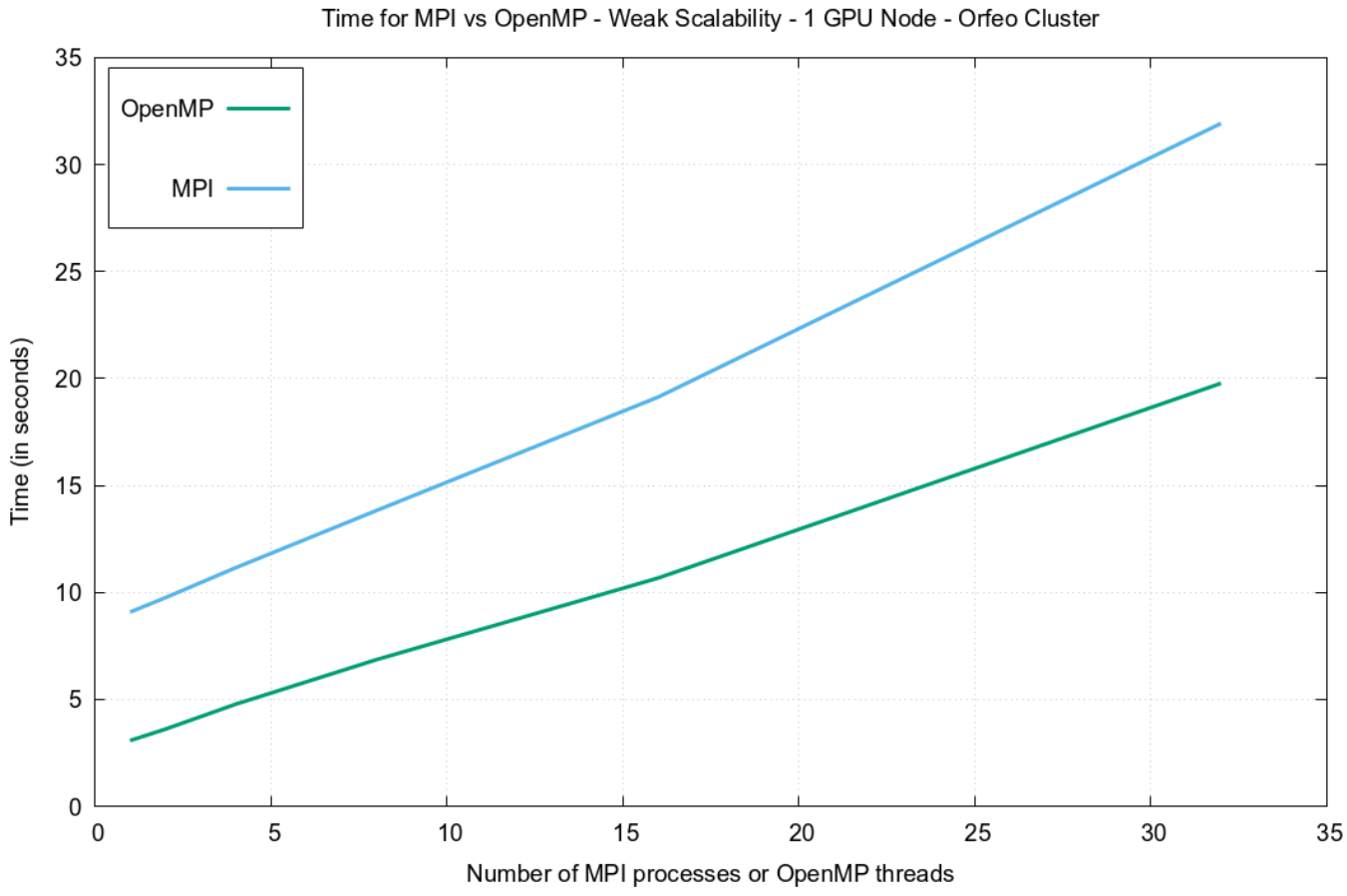


Time Comparison - MPI vs OpenMP

Here, we see the graph of time taken for both the strong and weak scaling scenario between MPI and OpenMP. The absolute times posted by OpenMP are better, probably due to the fact that the workload

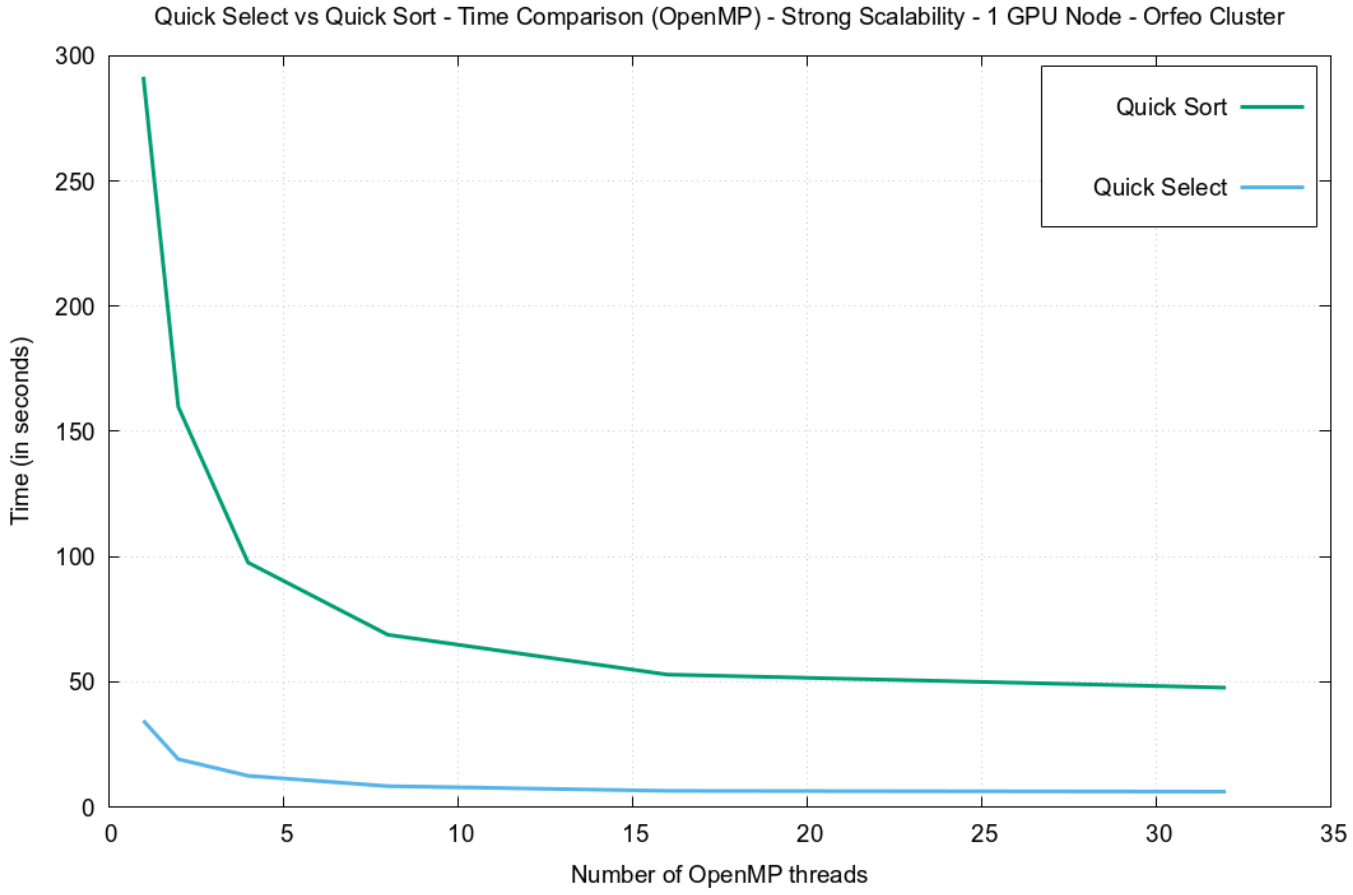
is more balanced there and due to lack of communication overhead.

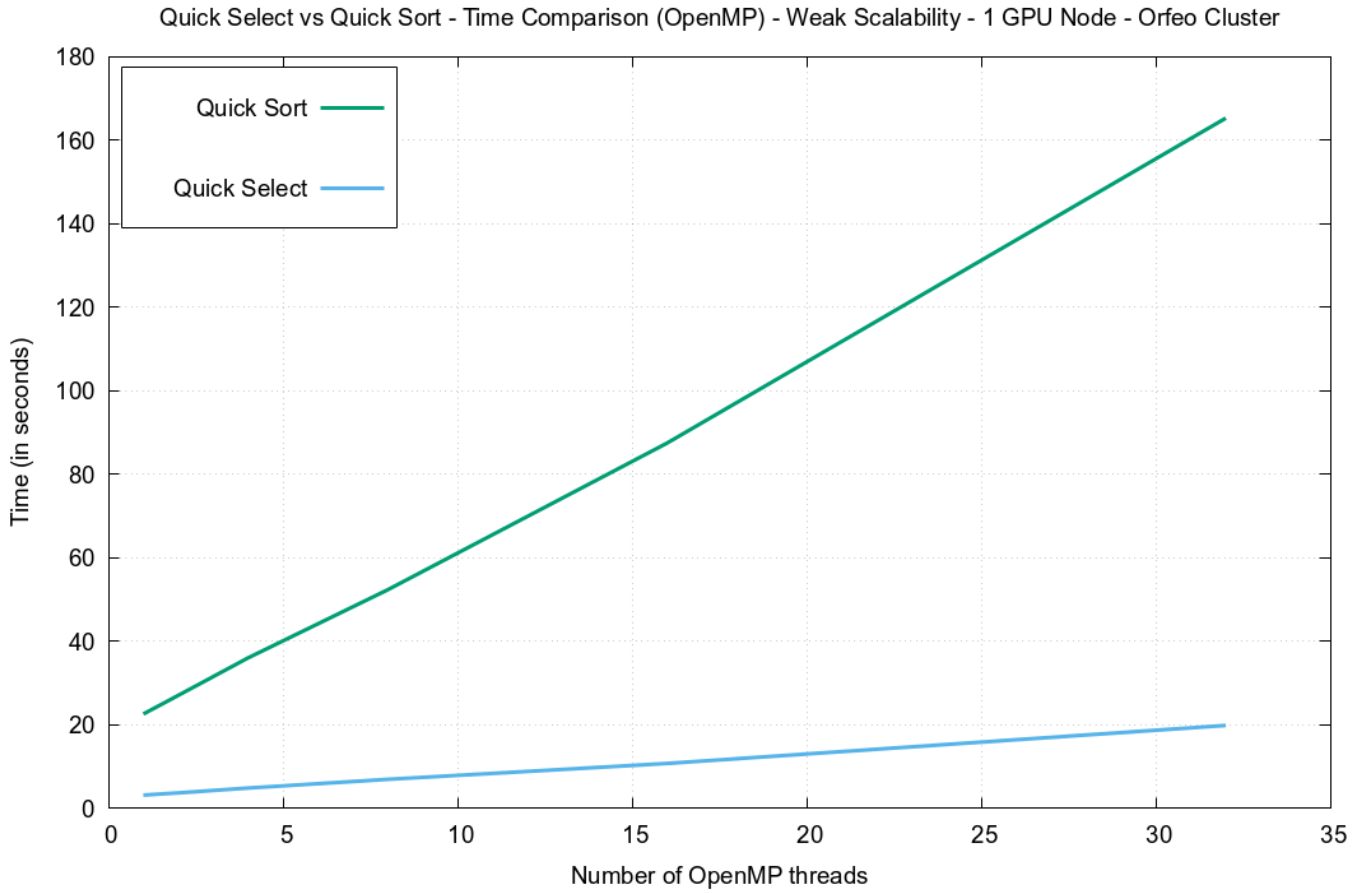




Quick Select vs Quick Sort

Earlier in the report, I mentioned that the choice of quick select is preferable over quick sort. An explanation courtesy of Wikipedia was also given there. Here, we see the data in the form of plots of both strong and weak scalability for the OpenMP code, that shows just how much faster Quick Select is compared to Quick Sort in the context of our code.





Discussion of Scaling Behaviour

Strong Scalability

If we compare the speedup for the strongly scalable test, we have $S_P = \frac{T(1)}{T(P)}$. Here, $T(1)$ and $T(P)$ are the times taken for serial and parallel execution with P processes/threads, respectively, and S_P is the corresponding speed-up. As we can see from the graphs and the table of data, the MPI program shows a much higher degree of speed-up than the OpenMP program. From Amdahl's law, which models strong scalability, we can infer that the parallel fraction of the MPI code is much larger than that of the OpenMP code. This could very well be because of how the MPI code distributes the tree-building across the different processors, and as a result, it shows a greater degree of parallelization. Both of these, however, also show quite poor performance with respect to the ideal scenario, where, $S_P = P$, where P is the total number of processors.

Absolute Performance

However, in terms of absolute performance, the OpenMP code performs much better. One possibility is that the MPI communication is too high, and that for sufficiently large datasets, the MPI version might outperform the OpenMP one. Another reason is probably that the MPI version has a less balanced workload in general. This is why, in the improvements suggested later on, I recommend trying to implement a hybrid version, since that would allow us to enjoy OpenMP within a node, and MPI across the nodes, allowing for large scalability, while cutting out unnecessary communication and having better balanced workload.

Weak Scalability

For weak scaling, as we can see in the graph as well, the deviation from ideal behaviour is very high for both OpenMP and MPI codes. Both have very poor weak scaling. In this case as well, the speedup is obtained by the same formula, and the ideal result should be the scenario where $S_P = P$, where P is the total number of processors. Reality is very far from this obviously. This is because Gustafson's law expects that serial part doesn't grow as problem size increases, however this is untrue in our case. For OpenMP, the quickselect section which is serial grows as $O(N)$, and also as we reach the maximum depth, at some point there is serialization of the tree building. For MPI, the same problem with quick select remains, and also the same issue of serialization is present. In addition, the data set must be partitioned by the ROOT processor in the very first iteration all by itself, and this is a serial task, and therefore, Gustafson's law's assumptions are not held true. Hence, this explains the very poor weak scalability of both OpenMP and MPI, but OpenMP's slight advantage over MPI.

Final Discussions

Defects

In the MPI program, we need to generate some `NULL` empty nodes whenever we distribute the data to an idle process to prevent some segmentation fault errors. This causes an increase or mis-match in the total size of the tree at the end. While the extra nodes are all `NULL`, I suspect there's a better way to solve this problem that unfortunately I couldn't come up with for now.

Quick Select vs Quick Sort

The choice of Quick Select is indeed justified due to much better performance (as explained earlier in greater detail), but I suspect that maybe Quick Sort can be made to perform a bit better than the implementation I have. For one, due to OpenMP nesting related concerns, and lack of available threads, the Quick Sort had to be serialized. I suspect there's a better way to approach this, even though, Quick Select remains the right choice. Merge Sort may also be a potentially interesting option, though.

Existing limitations and potential improvements

The MPI code is limited to be capable of running only on 2^n processors, which is a severe limitation. There has to be a better way to ensure that the extra processes on our architecture (for example, Ulysses `regular2` partition has 24 CPU cores), can take full advantage of the extra available cores. Using the extra processes for continued splitting maybe a potential solution to be explored.
Moreover, my approach only works for an immutable, homogenous dataset. Real life problem sets are rarely this simple and are more likely to cause much worse performance if they use my code. An appropriate restructuring to allow better performance for different types of datasets would be an interesting problem for the future.
A function to easily verify correct construction of the tree with the appropriate data would also be great. I tried to do such a thing, but didn't quite succeed in the end.
Much better work can also be done in terms of memory management and efficiency, cache access and so on. Analysis with a performance profiling tool and relevant optimizations remain a constant source of improvement.

It would be extraordinarily beneficial to be able to implement a parallel version of the Quick Select algorithm.
Finally, a hybrid program which would use MPI across nodes, and OpenMP within a node, would probably perform much better, especially in terms of scaling across multiple nodes of a cluster. This is also an interesting problem to be approached later on, if time permits.