Assignment 2

Niccolò Tosato

Problem description

A *kd-tree* is a data structure to represent k-dimensional data as a tree in order to perform efficient search. The following implementation is specialized in building *2* dimensional tree. Then, at each split node there are 2 possible directions. Fixed the direction,the set of points are bisected in 2 chuncks respect one pivot point and the selected direction, preserving partial order respect the pivot.

Algorithm

Under the assumption of that data are homogeneously distributed among all direction, we can simplify the algorithm. At each split in order to keep the tree balanced we can find the median point and use it as pivot. Furthermore at each split round robin is a suitable solution tochoose split axis. Given a dataset and the initial direction, the first operation to be performed is to find the median. Is possible to find the median with a computational complexity of $O(n)$ in all cases (median of medians), but I used a quickselect that perform a search for kth element with $O(n)$ comparison in the average case ($O(n^2)$ in the worst) and end up with an array partially ordered respect the median element. The three way partition routine scans the entire array only one time performing again $O(n)$ comparison. Once the array is partially sorted, the node could be built selecting the median point as pivot and the dataset could be splitted recursively.

Implementation

I've followed an hybrid approach exploiting openMP to build the tree and MPI to deliver data to feed openMP routine.

OpenMP layer

OpenMP is used to build tree given a set of point. The implementation exploit the task construct, since that a serial solution could be implemented recursively. Each recursive call become a task and the function call is handled by the first available thread. In order to keep the tree contiguous in memory (this detail will simplify the message passing phase), all nodes are pre-allocated ($n_{knode} = n_{kpoint}$) and at each call when requested a node's address is captured from a global shared index.

```
    global_node_address = malloc(p_number * sizeof(knode));
```

This index is increased when a node is requested, this configures a data-race, avoided with an atomic capture directive.

```
#pragma omp atomic capture
    this_node_address = global_node_address++;
```

To find the median the routine implements 2 different methods. If the number of points to sort is bigger than a certain threshold it finds the mediand using quickselect (as side effect the array is partially sorted), otherwise it uses a set of sorting network to order datapoints and then selects the point in the middle. The sorting networks are specific for the array size, then in order to select the correct network two solutions are available: a big switch-case construct or a function pointer array. A big switch case expose the program to a big number of branch, but the counterpart function pointer array configure a memory jump. I've implemented the function pointer array to keep the code cleaner and tunable. Only changing one parameter is possible. It sets the maximum array size for which the sorting network can be selected. Sorting network of size 16 and 32 have been tested.

```
    if (size > 15) {
        TYPE median_value = find_kth(dati, median, myaxis, size);
    } else {
        (*sortin_network[size - 1])(dati, myaxis);
    }
```

The openMP implementation require an amount of serial and not fully parallel phase of "warmup". The warmup stops when the $2^{tree\_depth} > n_{threads}$.

Further improvement.

Quickselect and sorting network routine when order the data point array direct swap the values on two positions. A smarter and lighter solution could be order an array of pointer, this solution saves some memory writes. Move a pointer is lighter than move the kpoint structure. On the same track, an approach to keep the structure lighter and faster instead to save a point inside a node, it is sufficient to save a pointer. When saving a pointer only 8 bytes are moved and stored, instead 16 bytes. The gain could be greater with more than 2 dimension. The bigger improvement of this solution cold be in the search phase. Saving only the interesting coordinate (a "key") and a pointer to the entire pointer, traversing the tree is easier and more cache friendly, there are more possibilities that 2 key are near. A faster solution to implement knode structure:

```
struct knode {
    TYPE coordinate;
    kpoint* split;
    knode *left;
    knode *right;
    unsigned short int axis;
};
```

Position of **coordinate** is good if $sizeof(TYPE) = 8B$. Otherwise it could be moved in the last one position or second-last, due memory padding. The swap function is optimized suggesting to the compiler that the 2 pointer will never overlap, leaving possibility for more aggressive optimization using the mark **restrict**. The assembly code is slightly different, but doesn't bring any performance improvement.

```
void swap(kpoint *restrict a, kpoint *restrict b) {
    kpoint t = *a;
    a->coord[0] = b->coord[0];
    a->coord[1] = b->coord[1];

    b->coord[0] = t.coord[0];
    b->coord[1] = t.coord[1];
}
```

MPI layer

Master MPI process starts building the tree until the desired number of leaf nodes is reached (i.e number of MPI processes).Thus the number of mpi processes spawned determine the depth of tree reached from the master. Only power of 2 MPI process number can be selected.

When the master has reached the maximum selected depth it start sending data points to slave MPI process. One branch is kept local and threated again with the plain openMP routine. To speedup the first phase, master process use openMP routine to build initial level. Data are handled by different threads and they call MPI routine inside a parallel region explointing MPI multithread support. Those thread are kept waiting incoming message from slave once they finished building the tree. This ends up in an overbooking thread number (respect core number) for the master process, since that one branch is kept local and keep growing and consume thread in a nested parallel region. When slaves complete their work send back the node to master's thread (the ones that were waiting). This procedure need to enable the nested parallel region, or set the level at least to 1.

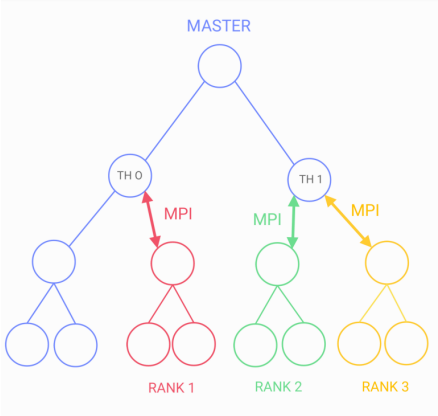


Figure 1: Hybrid approach scheme.

The plain openMP routine differs a bit from the hybrid one. Each node contains two pointer, that makes sense only on a shared memory approach. Then all pointers need to be mapped to an address owned by another process or machine to be meaningfull in the full tree. Then the master process pre-allocate the entire memory for nodes and send to the slave the pointer to that memory in order to offset correctly the local pointer. This is possible only to the contiguity of the tree.

Each mpi process can spawn some threads. Ideally given a total of N cores available, each process spawns $N_{core}/N_{processes}$, only the master MPI process has some additional threads, in total it spawns $N_{core}/N_{processes} + N_{processes} - 1$. Each thread is spawned close to father processes to keep the NUMA locality since when possible.

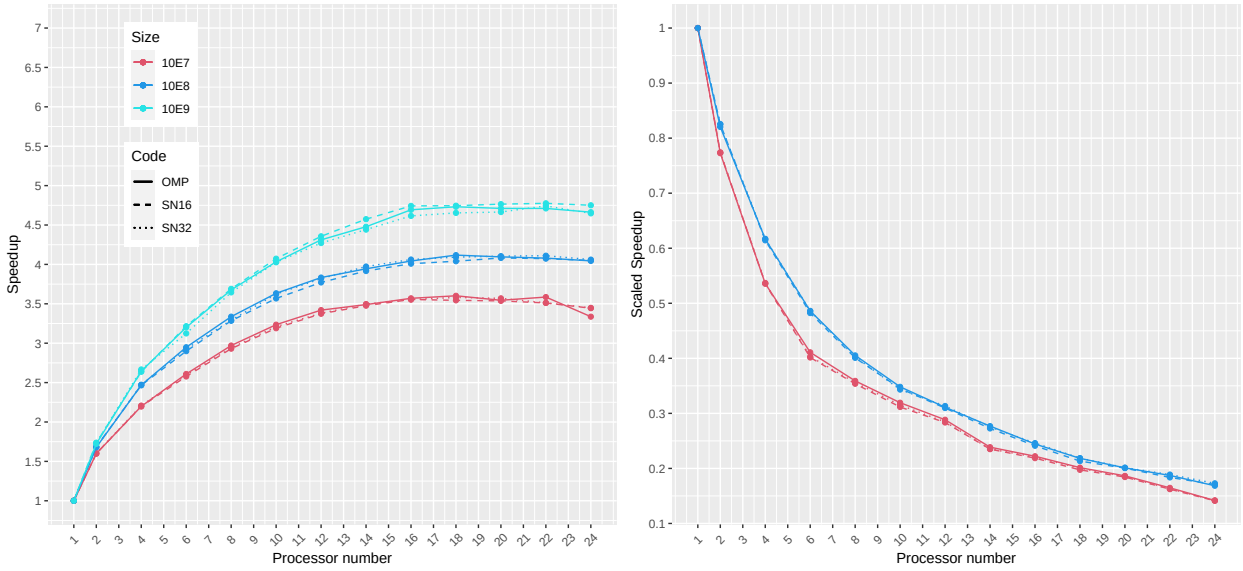
```
knode *current_address = (buildtree(r_slice, 2, myaxis, r_size, base_address, memory_offset));
        this_node->right = (current_address - base_address) + memory_offset;
```

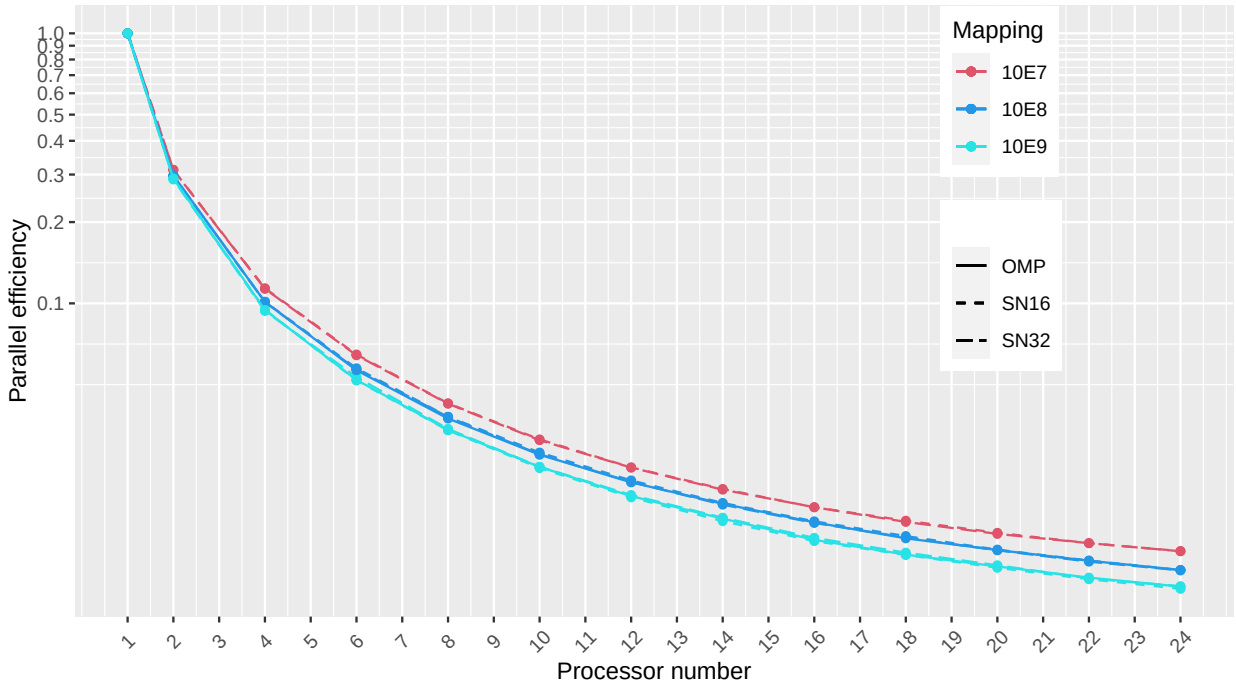
```
n_point=15
n_cpu=2
max_level=floor(log2(15))-1 #count from zero!
warmup_level=floor(log2(2))-1
```

Performance model and scaling

Since that different layer (plain OMP and hybrid) and solution (quickselect, quicksort and sorting network) have been implemented, several tests have been performed. Each run is repeated several times, at least 5 times for the longest, reserving in all case the entire node. Omp placement setting is managed by enviroment variable and number of spawned threads is set with function call inside the code.

Strong and Weak scalability openMP





	10^7		10^8		10^9	
cpu	T_{sn16}/T_{omp}	T_{sn32}/T_{omp}	T_{sn16}/T_{omp}	T_{sn32}/T_{omp}	T_{sn16}/T_{omp}	T_{sn32}/T_{omp}
1	1.022445	1.025856	1.005878	1.020550	0.9991113	1.011373
2	1.014666	1.018471	1.002036	1.017778	0.9980115	1.008237
4	1.025238	1.017431	1.006967	1.018829	0.9996097	1.007033
8	1.018831	1.034823	1.021949	1.034480	1.0126683	1.021641
16	1.011798	1.043072	1.014430	1.015845	1.0031713	1.015858
24	1.003586	1.029229	1.004812	1.016607	0.9679810	0.978654

[a] Sorting network vs partial ordering, strong scenario

	10^7		10^8	
cpu	T_{sn16}/T_{omp}	T_{sn32}/T_{omp}	T_{sn16}/T_{omp}	T_{sn32}/T_{omp}
1	1.0012823	1.020108	0.9984024	1.009586
2	0.9951471	1.013961	0.9965710	1.008229
4	1.0050497	1.019390	0.9984783	1.007418
8	1.0123782	1.026682	1.0132552	1.018162
16	1.0146405	1.013010	1.0142345	1.016127
24	0.9879942	0.993975	1.0017102	1.008652

[a] Sorting network vs partial ordering, weak scenario

From the previous graph we can see first that the speed up increase with the problem size. Second, the version with the sorting network seems to scale a little bit better than the normal version, however this is due to the lower performance in the serial version. The difference are minimal, also in terms of efficiency. The two versions seems be equivalent, even if with huge problem size, the number of nodes near the leaf is not neglegible and sorting network seems to be more efficient. Increasing the problem size, I expect a better performance using SN. Specializing the sorting network of size 3 and 2 is possible to delete two if statement and avoid useless branches.

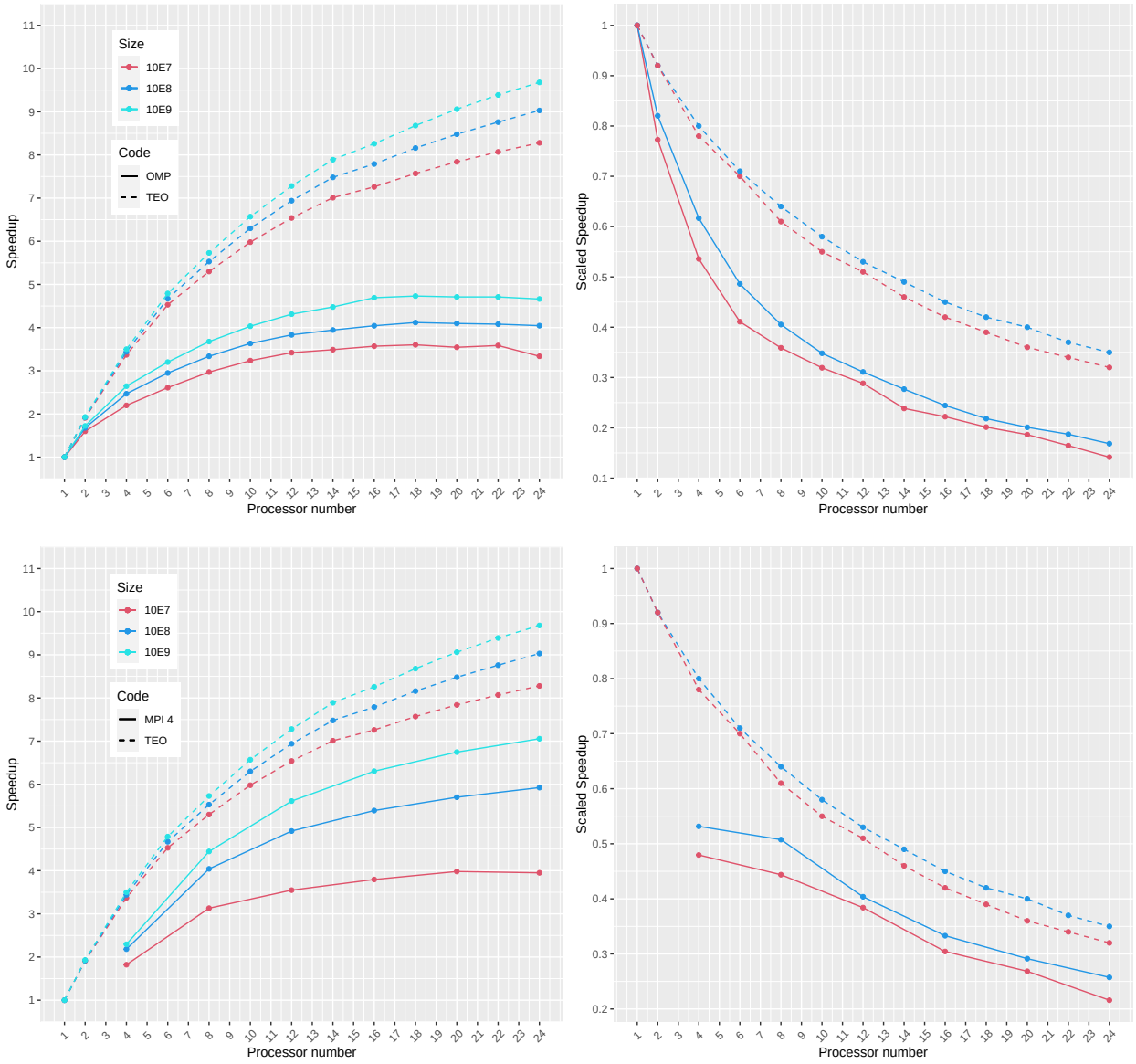
Performance model

A simple performance model could be:

$$operations = \sum_{i=0}^{log_2(N_{procs})} \frac{n_{points}}{2^i} - 2^i + \sum_{i=log_2(N_{procs})+1}^{log_2(n_{points})} \frac{n_{point}}{N_{procs}} - 2^i$$

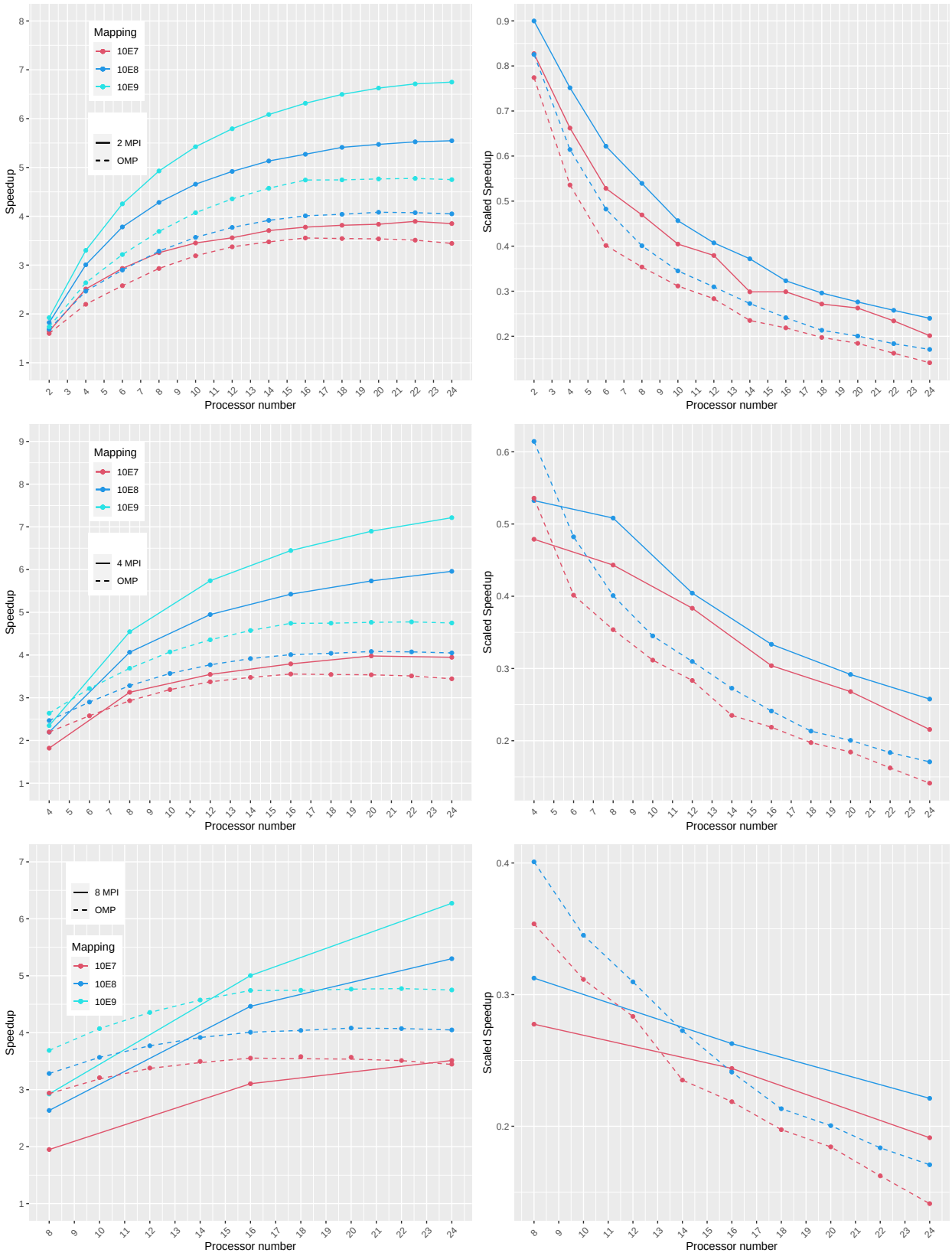
.

First sum term takes into account the serial fraction of code plus the "warmup" of the code, when there aren't enough tasks to go fully parallel. This is a performance killer by design, even if with a perfect code we can't achieve a perfect scaling.

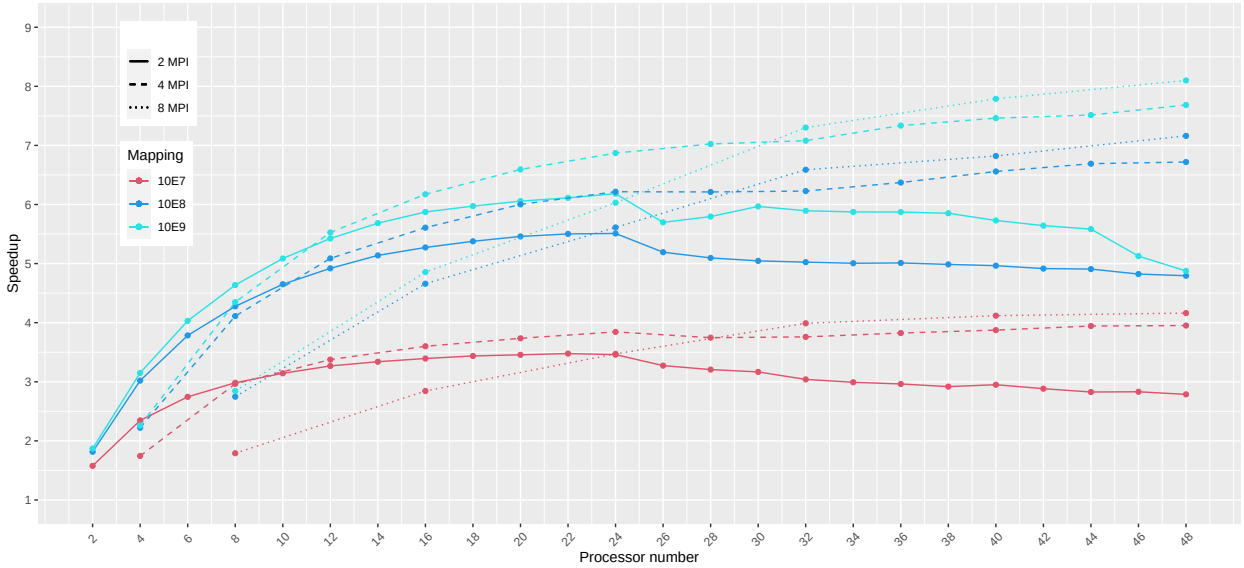


Performance model seems validated also for the MPI version, that goes closer (using the best MPI/OMP configuration) to the theoretical results.

Strong ad weak scalability Hybrid MPI + openMP (1 node)



Strong scalability Hybrid MPI + openMP (2 nodes)



Hybrid version shows always better performances respect openMP even if the overhead and communication is greather. The best performances in terms of speedup could be obtained using 4 MPI processes, 2 each socket. In the hybrid version the placement of the points in memory is forced to be in the nearest RAM location, then off course we can achieve a better memory access. OMP version, with more than 12 cores, on ORFEO access to data that belong to another numa region. Also there are less thread per process that perform some atomic capture execution. Then less overhead and sinchronization.

Since that with MPI paradigm is possible to have multiple nodes involved, I performed some scalability test across 2 nodes using up to 48 cores.

As the previous tests, speedup increases with the problem size. With only 2 processes mpi involved over 24 cores the speedup curve has negative slope. Since the variables OMP_PROC_BIND and OMP_PLACES are respectively **close** and **master**, when each process has more than 12 thread spawned, the other socket is used and the latency to access the memory is increased (for latter threads). This problem is not present in case of 4 MPI processes, since that they are mapped by socket and each process spawn his threads only on the "local" socket. Unfortunatly the L2 cache is not shared on Skylake, then there is no benefit from the cache running on the same socket.

The number of cores used is multiple of the number of processes, doesn't make any improvement have more core dedicated to a single MPI processes, this configure a workload unbalance, since that the program is slow as the slowest worker, even if some worker end their task before, there is no gain in the execution time.

Extra

This is the different assembly code syntentyzed by the compiler using **-O3 -march=native**, the first use restrict, the second one not. The two code are different and perform operation in different sequences.

```
swap_restrict:
.LFB21:
    .cfi_startproc
    vmovsd  (%rsi), %xmm2   # b_3(D)->coord, b_3(D)->coord
    vmovsd  (%rdi), %xmm1   # MEM[(struct kpoint *)a_2(D)], t$coord$0
    vmovsd  8(%rdi), %xmm0  # MEM[(struct kpoint *)a_2(D) + 8B], t$coord$1
    vmovsd  %xmm2, (%rdi)   # b_3(D)->coord, a_2(D)->coord
    vmovsd  8(%rsi), %xmm2  # b_3(D)->coord, b_3(D)->coord
    vmovsd  %xmm1, (%rsi)   # t$coord$0, b_3(D)->coord
    vmovsd  %xmm2, 8(%rdi)  # b_3(D)->coord, a_2(D)->coord
    vmovsd  %xmm0, 8(%rsi)  # t$coord$1, b_3(D)->coord
    ret
    .cfi_endproc
```

```
swap:
.LFB21:
    .cfi_startproc
    vmovsd  (%rsi), %xmm2   # b_3(D)->coord, D.4333
    vmovsd  (%rdi), %xmm1   # MEM[(struct kpoint *)a_2(D)], t$coord$0
    vmovsd  %xmm2, (%rdi)   # D.4333, a_2(D)->coord
    vmovsd  8(%rdi), %xmm0  # MEM[(struct kpoint *)a_2(D) + 8B], t$coord$1
    vmovsd  8(%rsi), %xmm2  # b_3(D)->coord, D.4333
    vmovsd  %xmm2, 8(%rdi)  # D.4333, a_2(D)->coord
    vmovsd  %xmm1, (%rsi)   # t$coord$0, b_3(D)->coord
    vmovsd  %xmm0, 8(%rsi)  # t$coord$1, b_3(D)->coord
    ret
    .cfi_endproc
```

Execution time:

cpu	10E9 (s)	10E8 (s)	10E7 (s)
1	498.018	42.870	3.751667
2	289.100	25.537	2.347000
4	188.310	17.367	1.706333
8	135.458	12.848	1.263000
16	106.118	10.603	1.051000
24	106.802	10.598	1.124333

Note:
Execution time