High Performance Computing final project
Exercise 2B

Giulio Fantuzzi [SM3800012]

21th February 2024

1 Abstract

This project focuses on optimizing the serial implementation of the vanilla Quicksort algorithm by introducing a hybrid parallel approach using `MPI`[1] and `openMP`[2]. The primary goal is to distribute the computational workload among processes, while leveraging multithreading within each process. Despite the limitations in scaling the Quicksort algorithm, characterized by its dependency on recursive calls and the potential load imbalance, my parallelized version demonstrates improved performance and mitigates some of the issues inherent in the algorithm's theoretical nature.

2 About serial Quicksort

Quicksort is a widely used sorting algorithm that follows the divide-and-conquer paradigm. Let X be an array of interest and denote its size with n. Such algorithm first selects an element x_{i^*} (with $i^* \in \{0, ..., n-1\}$), called pivot, and then divides the array into the following two partitions:

$$X_< = \{x_i \in X : x_i < x_{i^*}, i \in \{0, ..., n-1\}\}$$
$$X_\geq = \{x_i \in X : x_i \geq x_{i^*}, i \in \{0, ..., n-1\}\}$$

Quicksort finally calls recursively itself in each partition, and returns the sorted array by concatenating the sorted subarrays. Figure 1 briefly illustrates the procedure's scheme.

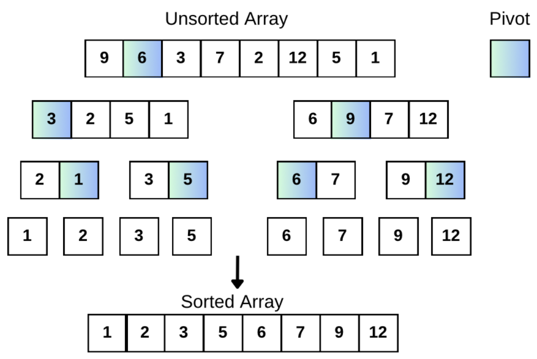


Figure 1: Serial Quicksort algorithm

The algorithm's recursion depth and overall complexity are intricately tied to the pivot choice. Mathematical analyses reveal that opting for a random pivot choice results in an average time complexity of $O(n \log n)$, while the worst-case scenario necessitates $O(n)^2$ comparisons. This pivot-related variability, which will be discussed in next sections, poses significant challenges when parallelizing the algorithm, as it directly influences workload distribution and coordination among parallel processes.

3 Parallelization strategies

The thought process behind my hybrid implementation of the Quicksort algorithm followed a bottom-up perspective. I concentrated first on how to improve sorting performance at a local level, exploiting multi-threading in a shared memory environment. Then, I progressed to the more complex task of orchestrating efficient workload distribution across multiple processes in distributed memory.

3.1 OpenMP

My `openMP` implementation introduces parallelism primarily through the use of `omp tasks`. More in details, tasks facilitate the parallel execution of recursive calls, allowing the left and the right partitions of the array ($X_<$ and $X_\geq$, respectively) to be sorted concurrently, enhancing the overall efficiency of the algorithm. The decision to opt for `omp tasks` over `omp sections` was determined by the dynamic scheduling capabilities of tasks. Unlike sections, tasks allow for a dynamic scheduling, enabling the runtime system to intelligently distribute tasks to the available threads, as they become accessible. This flexibility is particularly valuable in scenarios where the workload of each task is unpredictable at compile-time, a characteristic particularly relevant to my case: the size of each partition is, in fact, strictly dependent on the pivot selected at each stage.

3.1.1 Improvement with L1 cache

In addressing the optimization of the algorithm through `openMP`, a key consideration arises when dealing with smaller array dimensions. In such cases parallelism might not be fully exploited, and the serial version could even outperform the `openMP` one. In particular,the additional overhead introduced by parallelization mechanisms, such as thread creation and synchronization, could outweigh the computational benefits. Taking these observations into account, I updated the omp function in order to dynamically select between the serial and omp configuration, basing on a size threshold.

Algorithm 1 Pseudocode of the omp (L1 optimized) function

```
procedure OMP_QUICKSORT_L1($data, start, end, cmp\_ge$)
    $size \leftarrow end - start$
    if $size > 2$ then
        $mid \leftarrow$ PARTITIONING($data, start, end, cmp\_ge$)
        if $size > SIZE\_L1$ then
            #pragma omp task
            OMP_QUICKSORT($data, start, mid, cmp\_ge$)
            #pragma omp task
            OMP_QUICKSORT($data, mid + 1, end, cmp\_ge$)
        else
            SERIAL_QUICKSORT($data, start, mid, cmp\_ge$)
            SERIAL_QUICKSORT($data, mid + 1, end, cmp\_ge$)
        end if
    else
        if ($size == 2$) and ($data[start] \geq data[end - 1]$) then
            SWAP($data[start], data[end - 1]$)
        end if
    end if
end procedure
```

As part of the assignment, the data intended for sorting encompasses a structure designed to simulate scenarios where sorting involves non-basic types. This structure, denoted as `data_t`, encapsulates an array of `DATA_SIZE` doubles, with the default setting for `DATA_SIZE` being 8. Consequently, each instance of `data_t` necessitates 64 bytes of memory. In the context of EPYC nodes, equipped with an L1 cache of 4 MiB (4194304 bytes), I have reverse-engineered the maximum number of `data_t` entries that can fit within the L1 cache, obtaining a suitable threshold value of `L1_SIZE` = 65536.

The obtained gain in performance was notable, as illustrated in Figure 2:

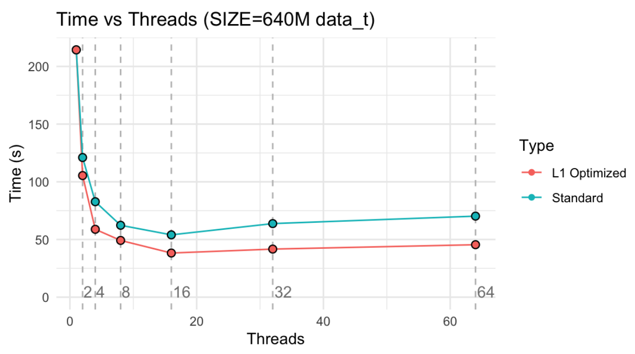


Figure 2: L1 optimized Quicksort algorithm

3.2 MPI

In the next phase of code optimization, I explored the distribution of computational tasks among different MPI processes. Initially, I scrutinized the constraints of the simplistic divide-and-merge approach, wherein the data is partitioned into smaller segments assigned to each process. Subsequently, I implemented an alternative algorithm [3] involving processes handling their data locally and exchanging partitions. It is noteworthy that in this alternative approach each process directly generates its own data chunk, as opposed to generating data in one single process and scattering it.

3.2.1 *Divide-and-merge* approach and its limitations

Initially, I considered a simplistic approach by partitioning the overall array into smaller chunks and assigning each segment to a distinct MPI process. This design aimed to fully exploit parallelism, enabling each process to autonomously sort its segment. However, to ensure ordering not only within each segment but also across different chunks, the implementation of a merging function was required.

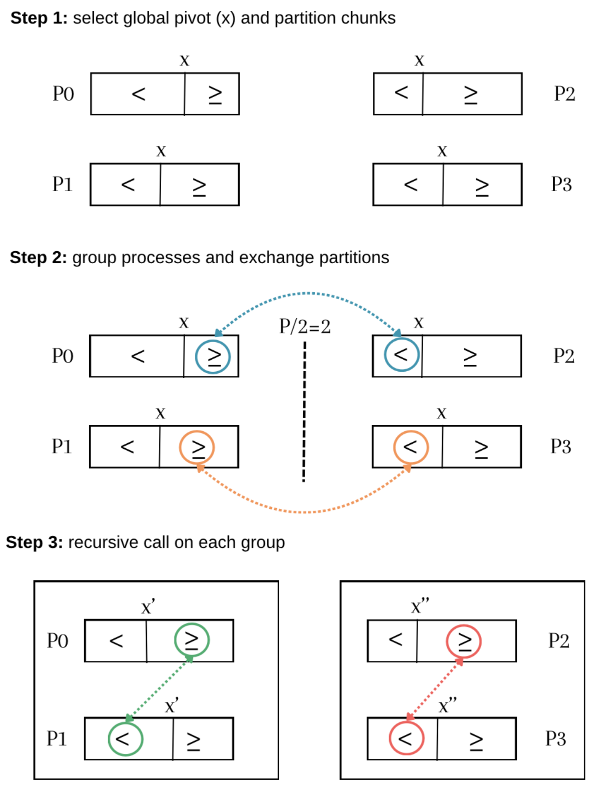
This initial approach presented two main limitations. Firstly, the merging function had the potential to become a heavy bottleneck, impeding overall efficiency. Additionally, the project specifications explicitly stated that we could not assume the data to fit into the memory available on a single node. Merging the entire array into a single process was hence deemed impractical, and I opted not to invest time in developing such an approach.

3.2.2 MPI Quicksort

Let P be the number of MPI processes of interest and $X_0, ..., X_{P-1}$ be the data chunks generated by processes $P_0, ..., P_{P-1}$, respectively. Initially, all the segments $X_0, ..., X_{P-1}$ are unordered. We seek to implement a procedure that partitions data on processes and exchanges messages (which essentially involves swapping data portions) so that at the end of the procedure we obtain $X_0 \leq ... \leq X_{P-1}$, with $X_i \leq X_j \iff \forall x_i \in X_i, \forall x_j \in X_j, x_i \leq x_j$. It then remains to locally sort data within each process, using the sequential version of Quicksort or, even better, the `openMP` configuration of Section 3.1.1.

The first step of this procedure consists in selecting a global pivot and broadcasting it to all the processes. For the sake of simplicity, let's assume the pivot to be selected randomly (we will discuss the impact of the pivot choice on the load imbalance more in details in Section 3.2.3). Basing on such pivot, each process p partitions its chunk into 2 sub-arrays: $X_<^{(p)}$ and $X_\geq^{(p)}$. Processes are then divided into 2 groups basing on their rank: the upper processes (with rank $p > \frac{P-1}{2}$) and the lower processes (with rank $p \leq \frac{P-1}{2}$). At this stage, each process p of the upper group sends its lower partition $X_<^{(p)}$ to the process $p' = p - \left(\frac{P-1}{2} + 1\right)$ in the lower group, receiving its upper list $X_\geq^{(p')}$, and vice-versa. This procedure is then repeated recursively in each of the 2 groups of processes.

Figure 3 illustrates the procedure explained above:



Step 1: select global pivot (x) and partition chunks
P0
P1
P2
P3
x
<
≥
Step 2: group processes and exchange partitions
P/2=2
Step 3: recursive call on each group
x'
x"

Figure 3: MPI quicksort procedure

After $\log_2(P)$ recursive calls, each process will have a list of values disjoint from all the other processes, and the largest element of process P_i will be less or equal than the smallest element of process P_{i+1}. In other terms, we managed to reach the condition $X_i \leq X_j \ \forall i, j \in \{0, ..., P-1\}$, which ensures the *order between* different processes to be respected. Notice that the recursion terminates when a group of processes is reduced to one process only. When this blocking condition is met, just one final step is required: sorting the local arrays using either the serial or, preferably, the `openMP` version of Quicksort, in order to ensure the elements also to be *sorted within* each process.

3.2.3 Pivot selection and load imbalance

As suggested in the previous sections, one of the major drawbacks of the Quicksort algorithm is that processes might potentially have a very different workload. Notice that this aspect regards both the serial version of the algorithm, both the MPI procedure we introduced in Section 3.2.2.

In the serial implementation, the pivot choice was originally determined within the `partitioning()` function utilizing the Lomuto partition scheme, where the pivot is selected as the rightmost element of the array. However, this approach revealed a weakness, particularly in worst-case scenarios when applied to already sorted (or "pseudo-sorted") arrays —a situation more common than expected. To address this issue I modified the `partitioning()` function choosing the pivot as the median among the first, middle, and last elements of the array. This adjustment, known as the "median-of-three" rule [4], provides a more accurate estimation of the optimal pivot (the true median), especially in cases where no prior information about the input's ordering is available.

Additionally, despite the expected balance achieved by the "median-of-three" rule in creating two roughly equal partitions, a practical adjustment was made in the placement of the pivot itself. Since the left partition contains elements $<$ pivot and the right partition contains elements $\geq$ the pivot, it is still reasonable to expect $X_{\geq}$ slightly larger than $X_{<}$. Hence, I positioned the pivot as the first element of the array (instead of the last, like in the provided code) and then shifted all $<$ elements to its left. On average, this should result into a small reduction of the `SWAP` operations between elements.

Algorithm 2 Pseudocode of "median-of-three"

```
$mid \leftarrow \frac{\text{start}+(\text{end}-1)}{2}$
if cmp_ge(data[start], data[mid]) then
    SWAP(data[start], data[mid])
end if
if cmp_ge(data[mid], data[end-1]) then
    SWAP(data[mid], data[end-1])
end if
if cmp_ge(data[mid], data[start]) then
    SWAP(data[start], data[mid])
end if
pivot ← data[start]
```

Regarding the MPI implementation, instead, the size of the process sub-arrays depends on the global pivot which is selected (and broadcasted to all processes) at the beginning of the procedure. In Section 3.2.2 I illustrated it by picking a random pivot, even though such a choice introduces potential pitfalls in terms of load imbalance. To address this concern, a more stable and robust methodology was developed. Specifically, the key idea was to make each process select a random element from its chunk, utilizing a seed based on its rank (just to enhance the degree of randomness). Subsequently, each process broadcasts its selected value to rank 0, populating an array of pivots. This array is then sorted [1] , and its median is selected as the global pivot to start the partitioning phase. While this approach incurs additional time compared to a completely random selection, the resulting improvement in load balance justifies the choice. Another consideration involved exploring the "median of three" instead of random selection within each chunk. However, empirical testing using `gdb` revealed that the impact on load balance was not so different from the adopted approach.

The pivot selection strategy discussed earlier did contribute to a slight alleviation of load imbalance, but it did not fully resolve the issue. Across numerous runs, the procedure consistently exhibited uneven work distribution among processes, adversely affecting overall time performance. Extensive literature exploration revealed that this challenge is deemed 'unavoidable' due to the inherent nature of the algorithm, prompting numerous studies to focus on potential improvements. Among these, *Hyper-Quicksort* and *Parallel Sort Regular Sampling (PSRS)* have emerged as particularly convincing proposals, but I deliberately omitted them from my analyses.

3.2.4 Generalization of the algorithm

The procedure described in Section 3.2.2 exhibits limitations when the number of processes (P) is not a power of 2. Notably, challenges arise when dealing with an odd number of processes, particularly in the case of the central process with rank $\frac{P-1}{2}$. More precisely, it fails to engage in partition exchanges with other processes, and since it encapsulates elements both less than and greater than the pivot, it cannot be included in any of the two groups of processes for the recursive calls. To address this challenge, I refined the algorithm's design, with the key observation that the division of processes

[1]The size of the pivots array is equal to the number of processes. Hence, this sorting step was assumed to be irrelevant in the duration of the overall procedure

into two sub-groups for recursive calls was actually independent of both the pivot selection and the partitioning within each process. The new generalized algorithm can be summarized as follows:

1. Select the global pivot and broadcast it to all the processes;
2. Partition the central process only (the one with rank $\frac{(P-1)}{2}$);
3. Compare the size of the two partitions $X_<^{\frac{(P-1)}{2}}$ and $X_\geq^{\frac{(P-1)}{2}}$. Scatter the elements of the minor partition to the other processes, while keeping the elements of the major partition;
4. Once concluded the scatter phase, proceed with the usual *partitioning and data exchange* procedure for all the other processes (with rank $p \neq \frac{(P-1)}{2}$);
5. Finally, assign the central process to the correct group for the recursive calls:
   - if the major partition was $X_<^{\frac{(P-1)}{2}}$, assign it to the group of processes with rank $p < \frac{(P-1)}{2}$
   - if the major partition was $X_\geq^{\frac{(P-1)}{2}}$, assign it to the group of processes with rank $p > \frac{(P-1)}{2}$

4 Scalability

The scalability of the code was evaluated on an EPYC node within the ORFEO[5] cluster. This assessment encompassed both strong and weak scalability and, in line with the primary goal of implementing a hybrid version of the Quicksort algorithm, my analyses will cover exclusively the hybrid code. To improve the reliability of results and reduce experimental variability, I conducted five repetitions of each measurement and considered their mean. Given the structure of the algorithm, I set the `--map-by` option of the `mpirun` command to `socket`, in order to minimize the time for MPI communications. Ultimately, I fixed `OMP_NUM_THREADS`=4, as it resulted the optimal omp configuration.

4.1 Strong scalability

To evaluate the strong scalability of the algorithm I fixed the size of the overall array to 160.000.000 `size_t` [2], and I measured the average sorting time by varying the number of MPI processes.

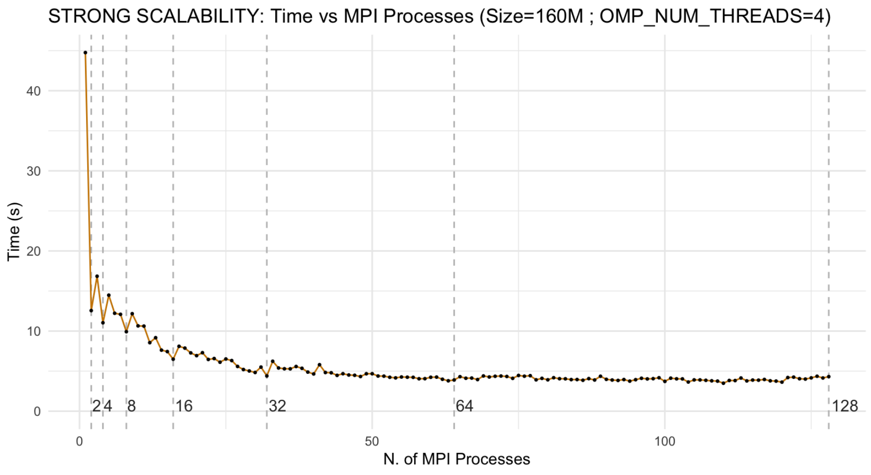


Figure 4: Strong scalability: Time vs MPI processes

Strong scalability was considered also in terms of Speed-up: $Sp(n, P) = \frac{T_s(n)}{T_P(n)}$, where $T_s(n)$ indicates the time to sort an array of size n with the serial version of the algorithm, while $T_P(n)$ indicates the time to do the same with the parallelized version, run with P processes.

[2]The chosen value does not carry inherent significance; it has a suitably large size to produce satisfactory results.

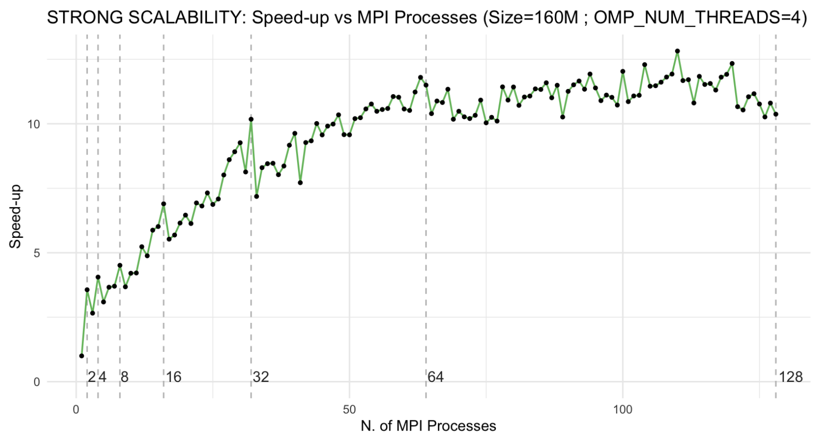


Figure 5: Strong scalability: Speed-up vs MPI processes

4.2 Weak scalability

To evaluate the weak scalability of the algorithm, instead, I let the overall size n vary, keeping the workload-per-process as a constant. In my case, I opted for a workload of 2.500.000 `size_t`.

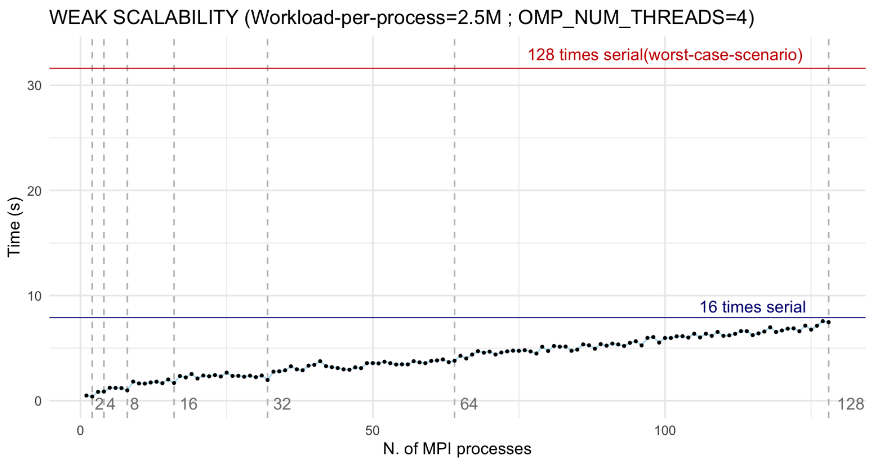


Figure 6: Weak scalability

5 Final considerations

While both strong and weak scalability deviate from the ideal scenarios, noteworthy results have emerged. In terms of strong scalability the ideal linear speed-up was not achieved: the trend displayed in Figure 5 is in fact more towards logarithmic. Nonetheless, Figure 4 still demonstrates a substantial reduction in sorting time attributed to my implementation. Turning to weak scalability, where the ideal case involves a constant horizontal line, Figure 6 shows that my code is upper-bounded by only 16 times the serial sorting time: a promising outcome considering that the worst-case scenario would involve a 128 times increase over the serial time. A worth highlighting achievement was the generalization of the algorithm. In contrast with the existing literature, which assumes to work with powers of 2 only, my implementation is robust across all input sizes. In summary, I identified clear limitations in scaling the Quicksort algorithm, attributed to its recursive calls, load imbalance issues, and the exponential growth of MPI communications when increasing the number of processes. Despite these inherent challenges, my parallelized version not only demonstrates improved performance, but successfully mitigates some of the theoretical limitations associated with the algorithm's structure.

References

[1] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[2] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[3] Frank Nielsen. *Introduction to HPC with MPI for Data Science.* Springer, 09 2016.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 3rd edition, 2009.

[5] areasciencepark. Orfeo-doc. https://orfeo-doc.areasciencepark.it.