Parallel Quicksort Algorithms in MPI and OpenMP

Marco Tallone

March 2024

Abstract

Quicksort is a popular divide-and-conquer, recursive sorting algorithm. This report presents different ways to parallelize this algorithm using both the Message Passing Interface (MPI) library and OpenMP. Multiple versions are implemented both in distributed and shared memory to study performance differences in terms of both strong and weak scalability. This study shows that a clear distinction between the different parallelization strategies can be made in terms of performance and scalability.

1 Introduction

Quicksort is a highly efficient sorting algorithm that follows a divide-and-conquer approach in order to recursively sort large arrays of elements according to an order relation.
This report wants to explore the possibility of parallelizing this well known algorithm proposing different implementations both in distributed and shared memory paradigms. The objective is to compare both complexity and performance of these algorithm in order to understand the trade-offs and advantages of each of them.
With the purpose of avoiding ambiguity, the different parallel versions are labelled as follows.

- **Task Quicksort**: a shared memory parallel quicksort algrithm that takes advantage of OpenMP's `#pragma omp task` directive.
- **Simple Parallel Quicksort**: basic parallellization of the quicksort algorithm in which the initial array is split among processes.
- **Hyperquicksort**: improved version of the parallel quicksort where pivot selection is optimized to improve load balancing.
- **Parallel Sort by Regular Sampling (PSRS)**: regular sampling is used in this parallel quicksort to optimize load balancing and minimize communications.

In the following sections, each of these algorithm will be first described from a theoretical point of view and the core ideas behind the practical implementation in distributed memory will be presented in the form of pseudocodes[1]. A proposed shared memory counterpart will also be then presented for each of the algorithms.
Performance analysis and scalability results will be then presented and discussed in the final sections of this report, together with some final considerations.
All the implementations have been written in C and have been tested on the *ORFEO* cluster at the AREA Science Park in Trieste [6]. With the intention of maintaining clarity and conciseness in this report, detailed code implementations have been omitted, but are available in the author's GitHub repository [7].

[1] With the exception of the serial version and the **Task Quicksort** algorithm, only implemented in OpenMP.

2 Serial Quicksort

Before diving into the parallelization of the quicksort algorithm, it is first necessary to briefly describe its serial version in order to have a baseline for comparison.
Given an unsorted input array $\mathbf{X}$ of n elements, the serial quicksort algorithm sorts the values of the array according to the following procedure:

1. choose a pivot value τ from the array $\mathbf{X}$;
2. partition the array $\mathbf{X}$ into two sub-arrays, one containing only elements smaller than the pivot, $\mathbf{X}_{<\tau}$, the other containing elements greater or equal to the pivot, $\mathbf{X}_{\geq\tau}$;
3. recursively sort the two sub-arrays $\mathbf{X}_{<\tau}$ and $\mathbf{X}_{\geq\tau}$ until the base case of having a sub-array of size 2;

The final sorted array $\mathbf{X}$ is then obtained by concatenatenation of the sorted sub-arrays. This sorting algorithm offers an average time complexity of $\mathcal{O}(n \log n)$, with the worst case scenario complexity being $\mathcal{O}(n^2)$ [5].

3 Task Quicksort

OpenMP's `#pragma omp task` directive allows for the creation of unitary independent tasks that can be executed by any available thread inside a parallel region.
Due to the divide-and-conquer approach of the quicksort algorithm, this directive can be directly exploited to obtain a fairly simple parallelization of this algorithm. The main idea is to have, at each recursive call, one thread that, after having partitioned the array, creates two tasks: one for the $\mathbf{X}_{<\tau}$ sub-array and one for the $\mathbf{X}_{\geq\tau}$ sub-array. Since the two sub-array formed at each step do not have any intersection, any thread that enters a task can work independently on its sub-array without the risk of contention with other threads.
This simple implementation can be schematized by the following OpenMP pseudocode[2].

OpenMP Task Quicksort

```
function omp_task_qsort(X):
    if len(X) > 2:
        τ ← choose_pivot(X)
        X_<τ, X_≥τ ← partition(X, τ)

        #pragma omp task
        omp_task_qsort(X_<τ)

        #pragma omp task
        omp_task_qsort(X_≥τ)
    else:
        if len(X) == 2 and X[0] > X[1]:
            swap(X[0], X[1])
```

Code 1: Pseudocode for the OpenMP `omp_task_qsort` function. Recursion ends when the array has length less then 2, where a simple swap is performed if needed.

The time complexity of this algorithm is the same as the serial quicksort, i.e. $\mathcal{O}(n \log n)$, but the parallelization allows the program to execute faster at a large scale. The main advantages of this approach are the simplicity of its implementation and the fact that it does not require any explicit synchronization between threads. However, we might take into account that the number of tasks created at each step can be very large, especially for large arrays, and this can lead to a significant overhead in terms of memory and time.

[2]Pseudocodes here presented don't constitute implementation examples, but only aim to give a minimal and concise idea of the main steps performed in each function by omitting some of the actual code details.

4 Simple Parallel Quicksort

The **Simple Parallel Quicksort** is the first attempted method to parallelize the quicksort algorithm in a distributed memory environment. Given a number P of processes, the algorithms first requires to split the initial array $\mathbf{X}$ into P equally sized[3] chunks, $\mathbf{X}^i$, of size n/P and then distribute these sub-arrays among the processes.
One of the processes then selects a pivot value τ and broadcasts it to all the others. This can be easily achieved in MPI [3] thanks to the use of the `MPI_Bcast` function. Each process can then partition its chunk according to this value forming in this way two local partitions: the *low partition* $\mathbf{X}^i_{<\tau}$, with elements smaller than the pivot, and the *high partition* $\mathbf{X}^i_{\geq\tau}$, with elements greater or equal to the pivot.
Subsequently, any process having rank $i < P/2$ exchanges its high partition with the low partition of the process with rank $i + P/2$ in a send and receive operation. Such operation has indeed been implemented in MPI with the `MPI_Sendrecv` function. This results in processes with rank $i < P/2$ having only elements smaller than the pivot and processes with rank $i \geq P/2$ having elements greater or equal than the pivot.
The algorithm then proceeds recursively as in the serial version with the lower rank group of processes working on the *low* part of the array on one hand, while higher rank processes keep sorting the *high* section on the other. In the MPI implemented code this separation is acheved through the use of the `MPI_Comm_split` function, which allows to define custom communicators for each subsequent recursive calls.
The following pseudocode briefly summarizes the core steps in the MPI implementation:

MPI Simple Parallel Quicksort

```
1  function MPI_Parallel_qsort(X^i, P, comm):
2      if P > 1:
3          if rank == 0:
4              τ ← choose_pivot(X)
5              MPI_Bcast(τ, comm)
6
7          X^i_{<τ}, X^i_{≥τ} ← partition(X^i, τ)
8
9          if rank < P/2:
10             MPI_Sendrecv(X^i_{≥τ}, X^{i+P/2}_{<τ}, comm)
11         else:
12             MPI_Sendrecv(X^i_{<τ}, X^{i-P/2}_{≥τ}, comm)
13
14         MPI_Comm_split(comm, rank < P/2, low_comm)
15         MPI_Comm_split(comm, rank >= P/2, high_comm)
16
17         MPI_Parallel_qsort(X^i, P/2, low_comm)
18         MPI_Parallel_qsort(X^i, P/2, high_comm)
19     else:
20         X_i ← sort(X_i)
```

Code 2: Pseudocode for the MPI implemetation of the Simple Parallel Quicksort algorithm.

From pseudocode 2 it's possible to deduce two observations. First of all, the processes P must be a power of 2 for the algorithm to work properly, since at each recursive iteration the number of processes is halved. Secondly, the recursion stops when only one process is entering each recursvie call, at which point the local array is sorted using either the serial quicksort ot the **Task Parallel Quicksort** showed in code 1. After $\log_2(P)$ recursions, in fact, the largest value in the chunk of the i^{th} process will be smaller than the smallest value in the chunk of the $(i+1)^{th}$ process. The final sorted array can therefore be obtained by simply letting each process sort its partition locally.

[3] Plus any eventual remainder distributed in a *round-robin* fashion among the processes.

The average time complexity of this algorithm is $\tilde{\mathcal{O}}(n/P \log n/P)$, but its actual execution is strictly dependent on the choice of the pivot at each recursive call. A bad choice of the pivot might lead to the processes having to work on very differently sized chunks, resulting in a very unbalanced workload.
An attempt has also been made to implement a shared memory version of this algorithm using OpenMP [2]. This implementation is not as straightforward as its distributed memory counterpart and requires the more complex procedure here described.
At the beginning of the recursive call, each thread is assigned an independent portion of the array to be sorted, which, in this case, is a shared variable. One among the threads then picks a pivot and writes its value in a shared variable so that each thread can parallely partition its portion in place. Since in OpenMP there's no direct way to perform message passing, nor to exchange the partitions between threads, the next step assumes the form of a global rearrangement of the elements in the array according to a shared `indexes` array. Such operation requires to know in advance where the elements of each thread's partition will be placed in the final array. In order to find these positions, each thread first writes the count of elements in its low partition on a reserved location of a shared array **L**. The same happens for the high partitions in an array **H**. The prefix sum is then performed on these arrays so that

- the j^{th} element in the low partition of the i^{th} thread will end up in position $L[i] + j$ from the start of the initial array[4];
- the j^{th} element of the high partition of the i^{th} thread will end up in position $L[P] + H[i] + j$ from the start of original array[5].

Therefore, each thread can independently compute the final positions for the elements in its partitions and write these in the corresponding locations of the shared `indexes` array. This whole procedure is schematically depicted in figure 1.
Before entering the next recursive call, one among the threads takes the job of serially rearranging the elements of the array according to the so computed `indexes` array. Unfortunately, this last step requires a sequential execution in order to avoid possible data races or other types of undefined behaviors. This indeed constitutes the major bottleneck of this implementation.

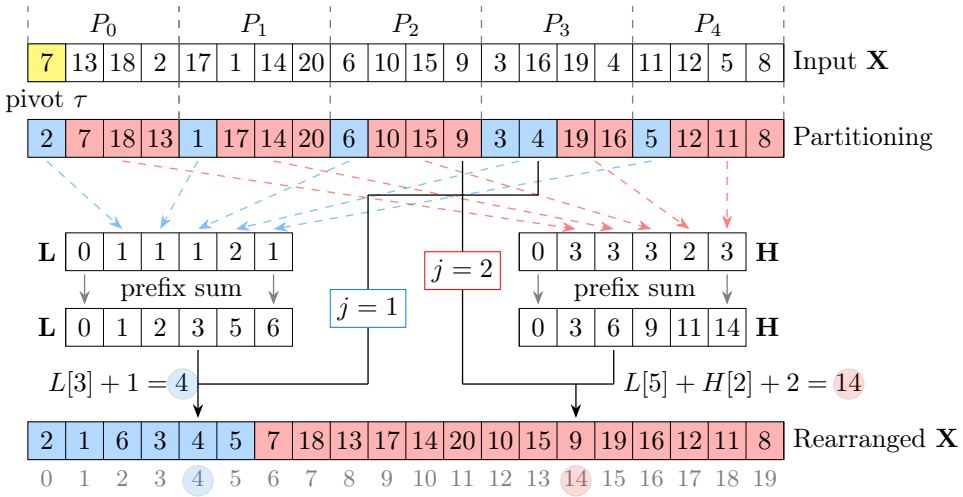

Figure 1: Rearrangement of partitions in shared memory using the prefix sum and $P = 5$ threads [4]. The pivot value is highligted in yellow, low partitions $\mathbf{X}^i_<$ are in blue and high partitions $\mathbf{X}^i_\geq$ are in red. Note that thread numbering is zero-based so the first thread will be identified by $i = 0$.

[4]The first element of both the **L** and **H** arrays is set to 0 to account for the final position of the first elements in each partition.
[5]Where P here is also the total number of threads.

5 Hyperquicksort

The **Hyperquicksort** algorithm tries to improve the load balancing among processes by having the P processes initially sorting their own chunks in $O(n/P \log n/P)$ time. The process that selects the pivot then chooses the median of its chunk and broadcasts it to all the other processes. The algorithm then proceeds in the same way as the **Simple Parallel Quicksort** described above.
Both the MPI and OpenMP implementations of this algorithm follow the same steps of the previous one with the only difference being the addition of the initial sorting of the chunks. Under the assumptions of uniform distribution of the elements in the processes' chunks and communication times dominated by transmission times (i.e. low latency), the overall complexity of the algorithm is $\tilde{O}(n/P \log(n+P))$. Overall, this algorithm provides a better alternative for load balancing but still relies on the assumption of finding uniformely distributed values for the partitions at each stage of the algorithm, which is not always guaranteed.

6 Parallel Sort by Regular Sampling (PSRS)

The **Parallel Sorting by Regular Sampling (PSRS)** prioritizes the pivot selection phase to improve load balancing among processes. Contairly to the prevoius algorithms, this one avoids the tipical quicksort recursive calls, hence the number of processes involved doesn't need to be a power of 2 and the all algorithm can be summarized in 4 stages.
In the first stage, each process P_i sorts its own chunk locally and chooses P samples at regular intervals from positions:

$$\frac{i \cdot n}{P^2} \quad \text{for} \quad i = 0, 1, \ldots, P-1$$

One process then gathers all the samples from the other processes, sorts the so obtained array of $P \times P$ samples and chooses $P-1$ pivots again at regular intervals. These pivots are then broadcasted to all the other processes. Each process then partitions its chunk in P sub-chunks using the received pivots and then participates in a *all-to-all* total echange in which, $\forall i \neq j$, each process P_i keeps its i^{th} partition and sends to process P_j the j^{th} partition. Finally, each process merges the P sub-chunks it received and sorts the resulting chunk. Pseudocode 3 describes the MPI implementation of this algorithm.
The overall complexity of the algorithm is $\tilde{\mathcal{O}}(n/P \log P)$, to which we have to account an overall communication cost of $\tilde{\mathcal{O}}(n/P)$ due to the *all-to-all* exchange.
The main advantange of the PSRS algorithm it that it prioritizes the pivot selection phase in order to reach optimal load balancing among processes. Repeated swappings of the same values are also avoided and communications among processes is minimized, making this algorithm the best candidate for large scale parallelization.

As for the previous algorithms, a shared memory implementation of this procedure has been attempted in OpenMP. Similarly to the distributed memory algorithm, the local chunk sorting, samples and pivot selection as well as the local partitioning phases are all done in parallel by each thread, with the only difference being that the *gather* and *broadcast* operations are simulated using shared variables.
The problem arises again in this algorithm when it comes to exchanging the different partitions among the threads. As in the shared memory versions of the **Simple Parallel Quicksort** and the **Hyperquicksort**, the *all-to-all* exchange assumes the form of a global rearrangement of the elements in the array according to a shared `indexes` array. The difference in the case of the PSRS algorithm is that the number of partitions to be exchanged is not fixed, but depends on the number P of threads. For this reason, the computation of the elements' final positions cannot be done with only two *prefix-sum* arrays, but requires an entire $(P+1) \times (P+1)$ *prefix* matrix[6] $\mathbf{M}$. This matrix is filled by all the threads and for all partitions in the following way. Initially, the i^{th} thread writes the count of elements in its k^{th} partition on the $(k+1)^{th}$ row and $(i+1)^{th}$ column of the matrix. After all the

[6]The additional dimension is needed since the 1^{st} row and the 1^{st} column of the matrix are zeroes for correct computation of the final positions.

MPI_PSRS

```
function MPI_PSRS(X^i):
    X^i ← sort(X^i)
    local_samples ← sample(X^i, P)

    MPI_Gather(local_samples, samples, P×P, 0, MPI_COMM_WORLD)

    if rank == 0:
        samples ← sort(samples)
        pivots ← pick_pivots(samples, P)
        MPI_Bcast(pivots, MPI_COMM_WORLD)

    (X^i_0, ... X^i_P) ← partition(X^i, pivots)

    MPI_Alltoall(X^i, P, X^i, P, MPI_COMM_WORLD)

    X^i ← sort(X^i)
```

Code 3: Pseudocode for the MPI implementation of the PSRS algorithm.

threads have finished writing, the i^{th} thread proceeds by computing the prefix sum of the $(i+1)^{th}$ row of the matrix. Once all the threads complete this operation, the prefix sum of the last column of the matrix is also computed and stored in an array $\mathbf{S}$. Having these quantities computed, finding the final positions of the elements in each thread's partition is straightforward: the j^{th} element in the k^{th} partition of the i^{th} thread will end up in position

$$\mathbf{S}[k] + \mathbf{M}[k+1][i] + j$$

since $\mathbf{S}[k]$ represents the total number of elements in the partitions preeceding the k^{th} one, while $\mathbf{M}[k+1][i]$ is the total number of elements in the same partition of the j^{th} element that the threads preceeding the i^{th} one have to place[7]. Once the `indexes` array has been filled with the so computed final positions, these are used by a single thread to serially reorder the elements in the shared input array. In appendix, the whole procedure is represented in figure 6. However, this algorithm presents a small advantage when compared to the other two shared memory versions. In fact, once the previous serial step is completed, no recursive call is needed and the final sorting of the array can actually happen in parallel by having thread i^{th} sorting the elements from position $\mathbf{S}[i]$ to position $\mathbf{S}[i]+\mathbf{M}[i+1][P]$. This possibility helps minimizing the penality of the serial portion of the algorithm and the difference when compared to the other two shared memory version is noticeable as reported in the results section.

7 Results

As previously anticipated, scalability has been assessed for all the implemented algorithms and the results are presented in this section. All the measurements have been conducted on the *ORFEO* cluster of the AREA Science Park in Trieste, using a single node of the *EPYC* partition. These nodes are equipped with 2 sockets, each of which has an AMD EPYC 7H12 (Rome) cpu installed, for a total of 128 cores [1].
Strong scalability of all the algorithms has been tested by feeding the different implementation with an input array of fixed dimension and incrementally increasing the number of processes involvend. The input array to be sorted contained randomly generated numbers that are homogeneously distributed in the range $[0, 1)$. For each possible value of processes and each algorithm, the sorting time has been collected multiple times on different input arrays[8] in order to estimate average time and standard deviation.

[7] Assuming zero-based indexing
[8] All having the same fixed size, but different randomly generated numbers.

On the other hand, weak scalability has been assessed by incrementally increasing the number of processes involved, but having an input array that guaranteed to have a fixed *workload-per-process*. This has of course been achieved by generating at each run a number of elements proportional to the processes. Also in this case multiple sorting times for different inputs have been collected in order to have a significant statistical sample.

Figure 2 summarizes the results obtained for the strong scalability test of the distributed memory algorithms implemented in MPI, for a fixed size array of 100 million elements. As expected, the algorithm showing the best scalability results is indeed the **PSRS** algorithm. It's possible to see from the speedup plot that the algorithm scales almost linearly up to 64 processes with great consistence and then it starts to slow down for higher numbers of processes by also increasing the variability of the results. Another noticeable aspect is that the **Hyperquicksort** algorithm shows improvements upon the **Simple Parallel Quicksort** only when the number of processes exceeds 16. This is expected and can be explained with the fact that the overhead of the initial local sort in the former algorithm is compensated only by a large number of processes, that depends on the input size of the array. On the other hand however, the **Simple Parallel Quicksort**, although running faster for small numbers of processes, shows the largest variability among all the algorithms as highligted by the high values measured for the standard deviations in the different runs of the test. This is in agreement to the theoretical expectations, since the algorithm is highly dependent on the choice of the pivot and the initial distribution of the elements among the processes. Similar results have been obtained for the weak scalability test where a fixed *workload-per-process* of 1 million elements has been used for all possible number of processes, as shown in figure 3. All the algorithms perform significantly better than the serial version, with the **PSRS** algorithm showing the best results among the three. In particular, in the worst-case-scenario with 128 processes, the **PSRS** performed 27 times better than the serial equivalent[9], while the other two implementations only performed 10 times better. In terms of efficiency, the **PSRS** algorithm maintains an average above 70% up to 64 processes, while the efficiency of the other two algoritms struggle to remain above 50% even for relatively small number of processes.

The strong scalability results obtained for the OpenMP algorithms are shown in figure 4. The input array used for this test contained 10 million elements and the number of threads has been increased up to 64 for each algorithm. The plots show that the best scaling algorithm among the different shared memory implementation is the task-based quicksort, which turns out to be the best alternative both in terms of perfomance and overall implmentation complexity. As highligted from the speedup plot, this algorithm show an initial linear speedup that then decreases after only 16 threads. Noticeably, the **PSRS** algorithm shows good results even with its shared memory implementation. Although its speedup does

[9]"Serial equivalent" means the time of the serial implementation with 1 process multiplied by 128, the highest number of processes used in the parallel versions.

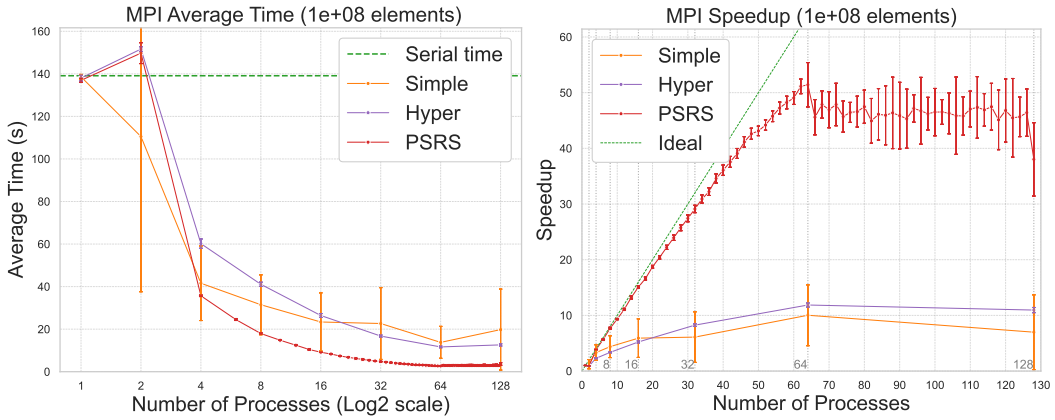

Figure 2: Strong scalability of the MPI implementations. On the left, the average sorting time against the number of processes for 100 million elements. On the right, the speedup against the number of processes. All results have been obtained with 2 threads per core.

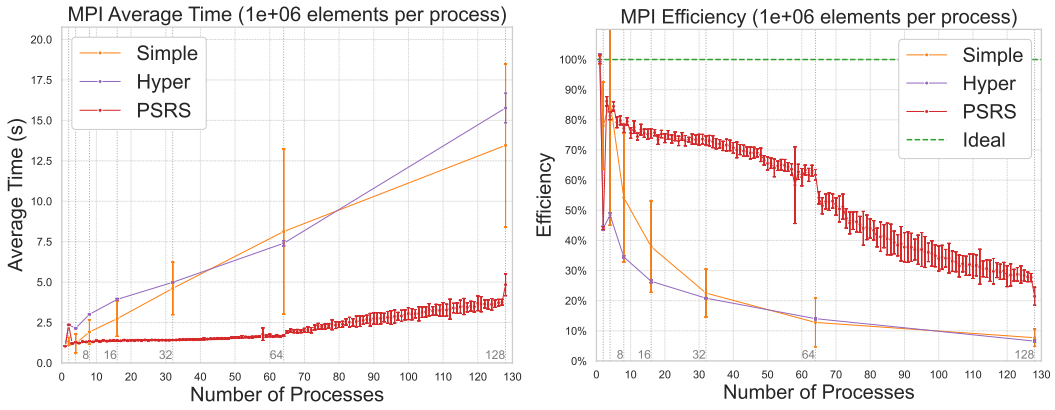

Figure 3: Weak scalability of the MPI implementations. On the left, the average time against the number of processes for 1 million elements per process. On the right, the efficiency vs. the number of processes. All results have been obtained with 2 threads per core.

not quite match the **Task Quicksort** nor the one of its distributed memory counterpart, it still constitutes a valid implementation up to 64 threads when compared to the serial version. The results obtained with the **Simple Parallel Quicksort** and the **Hyperquicksort** shared memory implementations, on the other hand, turned out to be even worse than the serial version for increasing number of threads. As anticipated, this was partially expected due to the serial sorting that need to be carried out by a single thread before advancing to the next recursive call. Therefore, in these cases, the addition of new threads only increases the parallelization overhead of the multiple parallel regions and the need of further synchronization among larger number of threads without bringing any real benefit.

The weak scaling analysis has therefore been conducted only on the **Task Quicksort** and the **PSRS** algorithms, as shown in figure 5. The input array used for this test contained 1 million elements per thread. The result of this test are similar to the ones previously presented, with the **Task Quicksort** having the most promising results among the two, showing an improvement on the serial version of a factor of 4, while the shared version of the **PSRS** performed about ~ 1.6 times better than the serial equivalent with 64 threads. Efficiency results in this case, however, are noticeably worse for both algorithms when compared to the distributed memory implementations: the **Task Quicksort** maintains an average efficiency of 50% only up to 8 cores, whereas the **PSRS** algorithm never shows efficiency above 50% for any number of threads in the tested range.

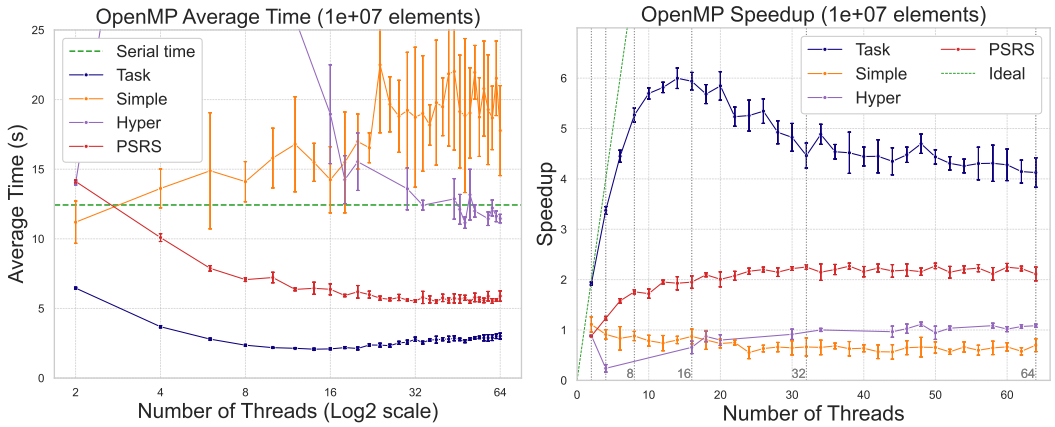

Figure 4: Strong scalability of the OpenMP implementations. On the left, the average sorting time against the number of threads for 10 million elements. On the right, the speedup vs. the number of threads.

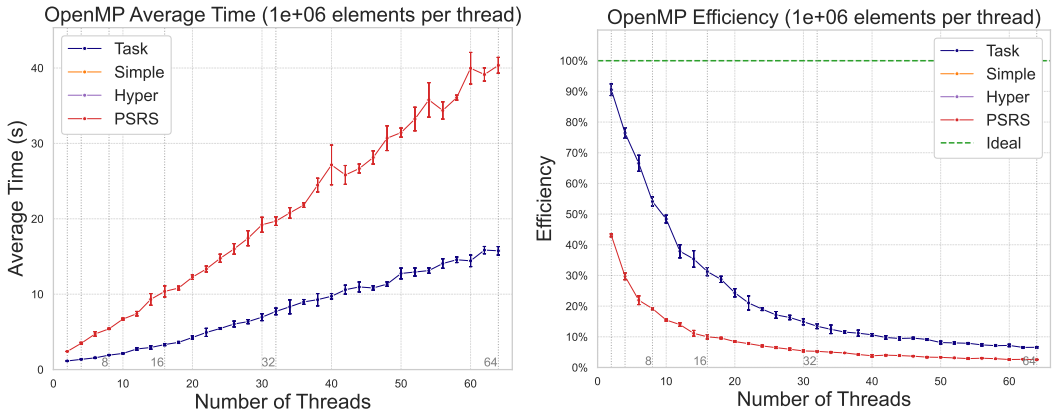

Figure 5: Weak scalability of the OpenMP implementations. On the left, the average sorting time against the number of threads for 1 million elements per thread. On the right, the efficiency vs. the number of threads.

8 Conclusions

In conclusion, this work presented multiple parallelization strategies for the quicksort algorithm and demonstrated that a significant difference exists, not only among the different parallel algorithm, but also due to the specific memory paradigm used for the concrete implementation.

The algorithms implemented in distributed memory using MPI showed a clear advantage in terms of scalability and performance when compared to the shared memory implementations. However, the results obtained for the OpenMP implementations showed that valid alternatives also exit for shared memory parallelization, with the **Task Quicksort** being the most promising among the different shared memory implementations. This can be of great importance in solving problems that might demand parallel sorting algorithms in a shared memory environment.

Additionally, as the results highlighted, the **PSRS** algorithm is the most scalable and efficient among all the implementations, both in terms of strong and weak scalability. This confirms the importance of load balancing and demonstrates that prioritizing the choice of pivots in this kind of algorithms greatly impacts its overall performance. Moreover, this algorithm also minimized the communication required between multiple processes which is another reason for its superior performance.

For what regards possible developments of this work, an improvements could be to find a way to further reduce the serial portion of work done in the recursive calls of the **Simple** and **Hyperquicksort** shared memeory algorithms, in order to make them more competitive with the other implementations and increasing their scalability. In fact, it might be interesting to check if solving that problem could yield better results with lower number of threads mirroring the results obtained in the distributed memory versions.

References

[1] AMD. Epyc 7h12 specifications, 2023. URL: `https://www.amd.com/en/processors/epyc`.

[2] OpenMP Architecture Review Board. Openmp application program interface, 2023. URL: `https://www.openmp.org/specifications/`.

[3] MPI Forum. Mpi: A message-passing interface standard, 2023. URL: `https://www.mpi-forum.org/docs/`.

[4] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2nd edition, 2003.

[5] Frank Nielsen. *Introduction to HPC with MPI for Data Science*. Springer, 2016. `doi:10.1007/978-3-319-21903-5`.

[6] AREA Science Park. Orfeo documentation, 2023. URL: `https://orfeo-doc.areasciencepark.it/`.

[7] Marco Tallone. Hpc exam repository, 2024. URL: `https://github.com/marcotallone/hpc-exam/tree/master/exercise2`.

Appendix

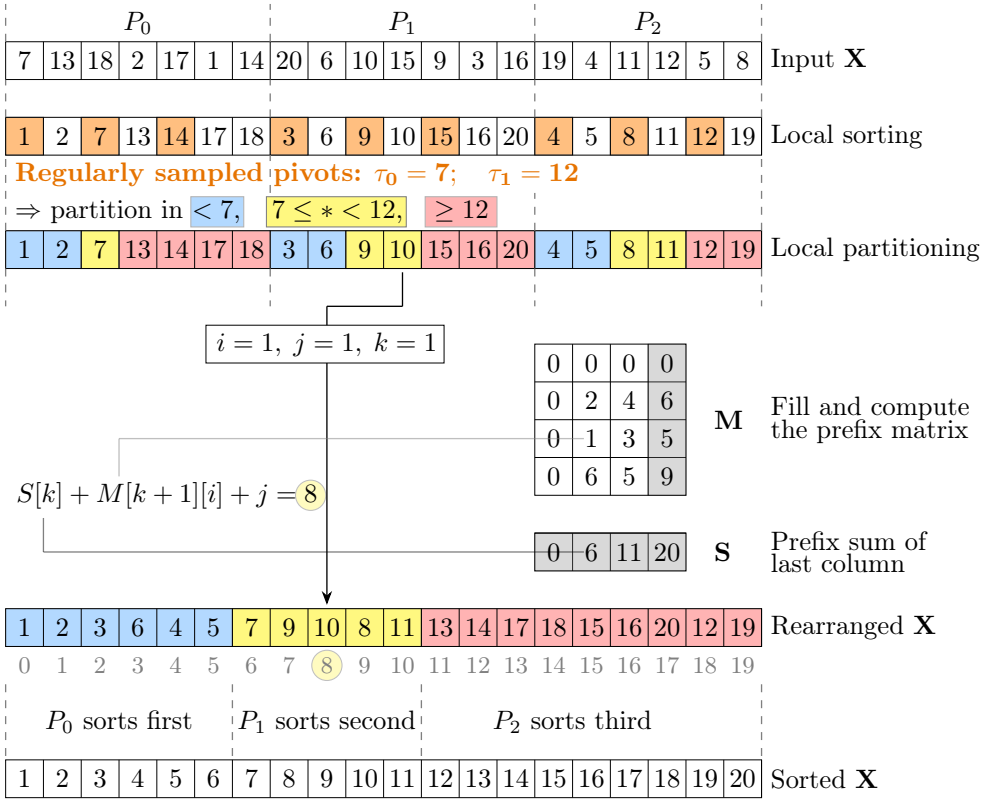

Figure 6: Sample execution of the **PSRS** algorithm implemented in shared memory with $P = 3$ threads and an array of $n = 20$ elements. Execution stages are labelled on the sides. Each thread sorts local chunk and select P samples (orange). $P - 1$ pivots are then selected by regular sampling from the sorted list of samples. Each thread therefore partitions its local chunks accordingly, then the prefix matric M and the prefix sum S of its last column are computed. Finally, the elements are rearranged in the final array according to the indexes computed. Each thread will then locally sort one of the P sub-chunks and the whole array will be sorted.