MSc in Data Science and Artificial Intelligence

High Performance Computing A.A. 2023/2024

Parallel Sorting by Regular Sampling (PSRS)

A parallel implementation of Quicksort, based on MPI

Prepared by:

Gabriel Masella
University of Studies of Trieste, Italy
Escuela Politécnica Superior, Universidad de Alicante, Spain

February 17, 2025

Contents

1 Introduction

In the context of High-Performance computing (HPC), efficient sorting of large datasets poses a fundamental challenge that requires both innovative algorithmic solutions and meticulous attention to parallel architectures.
This report presents the implementation of a parallel version of Quicksort based on the technique of regular sampling, a method that capitalizes on the advantages of Quicksort's rapid average performance and the load-balancing properties of sampling. The implementation was developed for distributed memory systems using the Message Passing Interface (MPI).
A performance evaluation was conducted on ORFEO's HPC cluster, specifically utilizing its AMD EPYC nodes to evaluate the effectiveness of the proposed approach. These nodes, which are based on the "Rome" architecture, boast a high number of cores and advanced memory architectures designed for parallel applications, rendering them optimal for assessing the scalability and efficiency of the proposed sorting algorithm.
The following sections will provide a comprehensive description of the implementation, an analysis of its performance characteristics, and a discussion of the observed speedup compared to the sequential algorithm.

2 Algorithm Description

The Parallel Sorting by Regular Sampling (PSRS) method is a popular technique for implementing a parallel version of Quicksort on distributed memory systems. The fundamental principle of PSRS is to first let each processor sort its local segment of the data, then use a regular (or systematic) sampling approach to select pivots that nearly equally partition the entire dataset. Following the broadcast of these pivots, each processor partitions its local sorted list in accordance with the new set of pivots. Subsequently, each processor exchanges data with all other processors, ensuring that each processor receives a segment containing only elements within a specific global range. The final step involves the merging of these segments into a fully sorted subarray.The concatenation of the sorted subarrays from all processors results in the complete sorting of the data.
The advantages of this approach are several. Firstly, the selection of pivots from a regular sample of the locally sorted arrays ensures a balanced distribution of data, minimizing workload imbalance during the merging phase. This is a key benefit of PSRS, as it ensures that the processors are evenly loaded. The communication efficiency of PSRS is also a notable advantage, as it only exchanges a small subset of sample elements (on the order of the number of processors) to determine the pivots. Consequently, each processor is only responsible for transmitting and receiving the pertinent data blocks, thereby minimizing the overall communication overhead. The algorithm's scalability is evidenced by its ability to execute each primary phase—namely, local sorting, sampling, partitioning, and merging—in a concurrent manner, with minimal synchronization requirements. Furthermore, PSRS's design is characterized by its simplicity and adaptability. It builds upon established sequential techniques, such as quicksort, while incorporating a straightforward sampling step to address global ordering. This feature facilitates the implementation and adaptation of the algorithm to diverse architectures, including those with distributed memory or multicore nodes.

Algorithm 1: PSRS (Parallel Sorting by Regular Sampling)

```
Input: Array $A$ of size $n$, number of processors $p$
Output: Sorted array
/* Phase 1: Local Sorting */
$local\_n \leftarrow n/p$
foreach processor $i$ in parallel do
    $A_{\text{local}} \leftarrow A[i \times local\_n \text{ to } (i+1) \times local\_n - 1]$
    sort $A_{\text{local}}$                    // Using a sequential quicksort or any efficient sort

/* Phase 2: Sampling */
$sample[i] \leftarrow$ SelectRegularSamples($A_{\text{local}}$, $num\_samples = p - 1$)
/* Gather samples from all processors */
$global\_samples \leftarrow$ Gather($sample[0], sample[1], \ldots, sample[p-1]$)
sort $global\_samples$
/* Choose $p - 1$ pivots evenly from $global\_samples$ */
$pivots \leftarrow$ empty list
for $j \leftarrow 1$ to $p - 1$ do
    $pivots[j] \leftarrow global\_samples[j \times (p-1)]$
Broadcast $pivots$ to all processors
/* Phase 3: Partition Local Data Using Pivots */
foreach processor $i$ in parallel do
    Initialize $p$ empty buckets $B[1 \ldots p]$
    foreach element $x$ in $A_{local}$ do
        $bucket\_index \leftarrow$ BinarySearch($pivots$, $x$)
        Append $x$ to $B[bucket\_index]$

/* Phase 4: All-to-All Data Exchange and Merge */
foreach processor $i$ in parallel do
    for $j \leftarrow 1$ to $p$ do
        Send $B[j]$ to processor $j$
        Receive bucket $R[j]$ from processor $j$
    $A_{\text{sorted\_local}} \leftarrow$ Merge($R[1], R[2], \ldots, R[p]$)
/* Final global sorted array is the concatenation of $A_{\text{sorted\_local}}$ from
 processors 1 to $p$ */
return $A_{\text{sorted\_local}}$
```

The presented pseudocode illustrates the fundamental phases of the PSRS algorithm:

1. Local Sorting and Sampling: Each processor initially sorts its own chunk and selects a fixed number of equally spaced sample elements.

2. Global Pivot Selection: The samples are then gathered and sorted; after which, evenly spaced pivot values are chosen to partition the data.

3. Local Partitioning: Each processor partitions its sorted local array using the pivots.

4. Data Exchange and Merge: Processors exchange the appropriate partitions with one another so that each processor receives all elements belonging to a particular global interval, which are then merged.

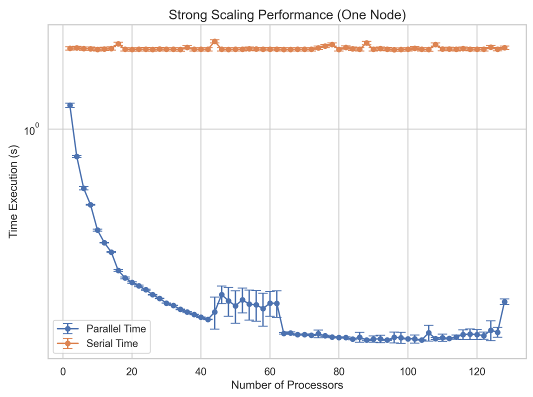
3 Experimental Results

3.1 Setup

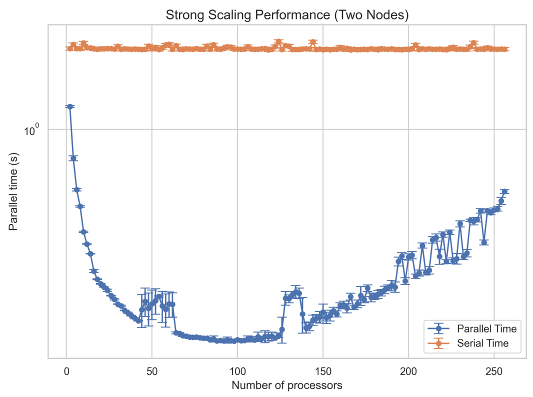
To evaluate the performance of the proposed parallel PSRS implementation, two types of scaling experiments were conducted on ORFEO's cluster using EPYC nodes: strong scaling and weak scaling. In the strong scaling study, the total problem size was maintained constant. In this study, the global

data size was fixed at 10,000,000 elements, and the number of MPI processes was varied (from as low as 2 up to 256). This approach tests the ability of our parallel PSRS implementation to reduce the overall execution time when the same fixed workload is divided among an increasing number of processes. In addition, similar strong scaling experiments were performed using both a single-node configuration and a multi-node (two-node) configuration to assess the impact of inter-node communication on performance.
For weak scaling, we maintained a constant workload per process—1,000,000 elements per MPI process. As the number of processes increased, the global data size grew proportionally (global size = number of processes × 1,000,000). These experiments, conducted on a single EPYC node, aim to evaluate whether the implementation can sustain efficiency as both the problem size and available computational resources grow in tandem. Both sets of experiments were automated via SLURM batch scripts.

3.2 Strong Scaling



(a) Time executions varying the number of processors in one EPYC node, in logarithmic scale



(b) Time executions varying the number of processors in two EPYC nodes, in logarithmic scale

Figure 1: Strong Scaling Analysis of PSRS approach

The parallel execution time (blue line in the figure 1a) on a single AMD EPYC node rapidly decreases with increasing process numbers, from 1 to approximately 32-64. Such strong-scaling speedups can be attributed to the shared workload among multiple MPI processes, leading to reduced sorting times. However, beyond a certain threshold, such as 64 or 128 processes, additional parallelism results in only marginal improvements or even slightly longer times. The overhead associated with interprocess communication and synchronization can, at a certain point, become more substantial than the gains derived from increased parallelism. This is because the additional overhead costs, such as those associated with partitioning, merging, and MPI calls, offset the benefits of further division of work. The orange line labeled "Serial Time" remains approximately constant at around 2 seconds. It functions as a baseline, indicating that if the parallel execution time dips below 2 seconds, the code is effectively utilizing more processes.
With two nodes, as shown in figure 1b, the parallel execution time exhibits an initial decrease in process count, similar to the behavior observed in a single-node scenario. However, as the process count continues to increase (particularly beyond a single node's total physical cores), greater dependency on cross-node communication becomes evident. When the workload is distributed across two separate physical nodes, the data must traverse the interconnect (e.g., Infiniband or Ethernet), resulting in significantly higher overhead compared to local shared-memory communication. Consequently, once the optimal range of processes is surpassed, the additional cross-node communication and synchronization begin to dominate, causing the overall parallel time to increase again.

3.3 Weak Scaling

In weak scaling, each process handles a fixed amount of data, so as the total data size grows, the number of processes also increases proportionally. Ideally, if the algorithm scaled perfectly, the total

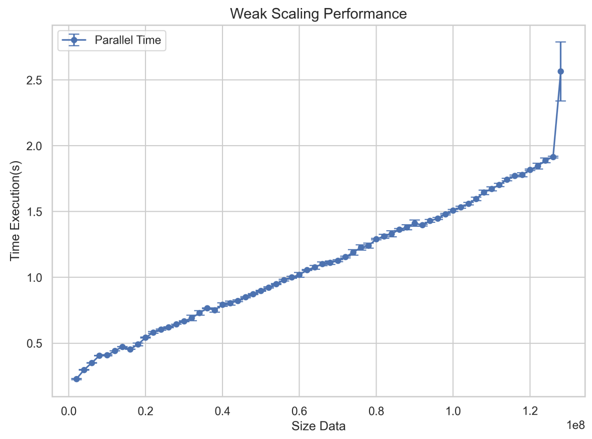


Figure 2: Time Execution varying the data size proportionally to the number of processes

execution time would remain nearly constant because each process is doing the same amount of work regardless of the total size. As illustrated by the plot 2, the total runtime experiences an increase from less than one second to more than two seconds when the scale of the data is expanded from a few tens of millions of elements to around 120 million. This upward trend signifies that the overhead associated with parallelization is not negligible and accumulates as additional processes—and the corresponding data—are introduced.
The potential sources of overhead could be as follows: Even though each process only sorts its local subset, phases such as pivot sampling, data exchange (partitioning), and final merging steps incur communication costs; the presence of more processes generally results in increased collective communication overhead, even if each process has roughly the same workload, global synchronization points—like barrier operations or collective calls—add to the total time. On a single node, many processes can compete for shared memory bandwidth; on multiple nodes, interconnect bandwidth can be limiting. As the system approaches saturation, the time required increases. While each process operates on a fixed portion of the data, the overall data structure, such as arrays used for storing pivot samples, can expand with the number of processes. This leads to increased overhead for steps like gathering pivots or final merges. In the context of weak scaling, it is ideal for the runtime to remain constant. However, the observed upward trend in the runtime is attributable to the cost of additional inter-process communication, synchronization, and potential resource contention, which outweighs the benefits of increased per-process speedup.

4 Conclusion

In this project, a parallel Quicksort algorithm was developed and evaluated using the PSRS (Parallel Sorting by Regular Sampling) approach. The results demonstrate that PSRS can effectively distribute the sorting workload among multiple MPI processes, achieving significant speedups over the serial baseline.
Strong scaling experiments on one node showed rapid reductions in execution time until communication and synchronization overheads began to dominate. Extending the setup to two nodes introduced additional interconnect costs, leading to a decline in the speedup curves. For weak scaling, each process handled a fixed subset of the data, and as the total data size increased (and thus the process count), communication and synchronization overheads naturally increased, resulting in gradually longer execution times. Nevertheless, the slope of the performance curve remained within an acceptable range, indicating that the PSRS-based parallel Quicksort maintains a reasonable level of scalability as both data size and process count increase.
In summary, these experiments demonstrate that PSRS provides a practical, scalable framework for sorting large datasets in HPC environments, with its performance being largely dependent on network bandwidth, node communication latencies, and the total degree of parallelism employed.