aci
avances en ciencias e ingenierías

Artículo/Article
Vol. 17, nro. 1
e3701

Empirical evaluation of parallel implementations of MergeSort

José E. Solsona[1]*, Sergio Nesmachnow[2]

[1]Universidad ORT, Montevideo, Uruguay.

[2]Universidad de la República, Montevideo, Uruguay.

*Corresponding author, email: solsona@ort.edu.uy

Evaluación empírica de implementaciones paralelas de MergeSort

Abstract

Sorting algorithms are a fundamental piece in the development of computer systems. MergeSort is a well-known sorting algorithm, much appreciated due to its efficiency, relative simplicity, and other features. This article presents an empirical evaluation of parallel versions of MergeSort, applying shared and distributed memory, on a high-performance computing infrastructure. The main result indicates that parallelization of recursive invocations combined with a parallel merge operation offers better speedup than just parallelization of recursive invocations. Moreover, better speedup was achieved in a shared memory environment.

Keywords: High Performance Computing, Parallel sorting

Resumen

Los algoritmos de ordenamiento son una pieza fundamental en el desarrollo de sistemas informáticos. MergeSort es un algoritmo de ordenamiento muy conocido, muy apreciado por su eficiencia, relativa simplicidad y otras características. Este artículo presenta una evaluación empírica de versiones paralelas de MergeSort, aplicando memoria compartida y distribuida, en una infraestructura computacional de alto rendimiento. El resultado principal indica que la paralelización de invocaciones recursivas combinada con una operación de mezcla paralela ofrece una mejor aceleración que la paralelización de invocaciones recursivas por sí sola. Además, se logró una mejor aceleración en un entorno de memoria compartida.

Palabras clave: Computación de alto rendimiento, Ordenamiento paralelo

Licencia Creative Commons Atribución-NoComercial 4.0

Editado por / Edited by: Dennis Cazar

Recibido / Received: 20/11/2024

Aceptado / Accepted: 09/12/2024

Publicado en línea / Published online: 08/05/2025

INTRODUCTION

Sorting algorithms are a fundamental piece in the development of computer systems and information processing. In particular, MergeSort is a general-purpose sorting algorithm based on element comparison [1]. Although it is reasonable to assume that the basic sorting operation of performing a comparison has low computational cost, sorting large volumes of data in the Big Data era would require long execution times using sequential

 DOI: https://doi.org/10.18272/aci.v17i1.3701

sorting implementations. Therefore, it is natural to explore adapting existing algorithms to leverage the parallel processing capabilities provided by modern hardware.

In this line of work, this article presents an empirical evaluation of parallel versions of the MergeSort algorithm in high-performance computing environments. The developed parallel versions of MergeSort are based on shared and distributed memory, following different parallelization schemes, using the OpenMP interface and the Message Passing Interface (MPI) library. The evaluation was carried out on a problem instance of size elements, considering up to 39 computing resources.

The main results of the empirical evaluation indicate that in the shared memory implementation parallelization of recursive invocations combined with a parallel merge operation offers better speedup than just parallelization of recursive invocations, as suggested by theoretical analysis. For the distributed memory implementation, a functional decomposition approach performed better than a straightforward master-slave approach. However, the functional decomposition approach did not result in better performance results than a hybrid implementation combining distributed and shared memory. Overall, the best speedup was achieved in the shared memory implementations.

PROBLEM DESCRIPTION

Classical MergeSort

MergeSort is a general-purpose sorting algorithm based on element comparison and the divide-and-conquer strategy [2]. On a conceptual level, MergeSort operates as follows on a vector of n elements:

1. Divide the vector into n parts, each with a single element, which are considered ordered.

2. Merge the parts producing new ordered vectors until only one ordered vector remains as the final result.

MergeSort is presented in Algorithm 1. Sequential MergeSort., following the version by Cormen et al. [3]. Being an algorithm based on the divide-and-conquer strategy, its most natural description is recursive. Assuming an initial vector $A=(a_1,\dots,a_n)$, the result is obtained by invoking $MergeSort\ (A,1,A.len)$, where $A.len = n$, l (*left*) stands for the initial position, and r (*right*) stands for the final position.

```
procedure MERGESORT(A, l, r)
    if l < r then
        m := ⌊(l + r)/2⌋
        MERGESORT(A, l, m)                    ▷ Sort first half
        MERGESORT(A, m + 1, r)                ▷ Sort second half
        MERGE(A, l, m, r)                     ▷ Merge both halves
```

Algorithm 1. Sequential MergeSort.

DOI: https://doi.org/10.18272/aci.v17i1.3701

The division of the problem is performed by two recursive invocations of the procedure MergeSort on the left part $A[l,...,m]$ and the right part $A[m+1,...,r]$ of the vector, respectively. The division is performed while there are at least two elements, since otherwise the vector is already sorted. After the division, the conquer action is carried out through the Merge operation to merge the parts $A[l,...,m]$ and $A[m+1,...,r]$ already ordered. The usual way to implement this operation follows the intuitive idea of how to mix two decks of cards already sorted where the smallest cards appear on top. The basic step consists of comparing two cards taken from each deck where we choose and remove the smallest one to build a new deck. We repeat the step until one of the decks runs out of cards in which case the remaining cards will go to the end of the new deck. The sequential merge operation is presented in Algorithm 2. Sequential merge operation.. The presented version differs slightly from the one by Cormen et al. [3] since it does not use sentinels, just to simplify the presentation.

```
procedure MERGE(A, l, m, r)
    i, j, k := l, m + 1, 1
    Let T be a new vector of size r − l + 1.
    while i ≤ m ∧ j ≤ r do                          ▷ Both halfs not empty
        if A[i] ≤ A[j] then
            T[k] := A[i]
            i := i + 1
        else
            T[k] := A[j]
            j := j + 1
        k := k + 1
    while i ≤ m do                                  ▷ Remaining elements on the left
        T[k] := A[i]
        i, k := i + 1, k + 1
    while j ≤ r do                                  ▷ Remaining elements on the right
        T[k] := A[j]
        j, k := j + 1, k + 1
    for k ∈ [1..T.len] do
        A[l + k] := T[k]
```

Algorithm 2. Sequential merge operation.

The computational complexity of MergeSort considers the number of comparisons performed when shuffling as the representative operation of the execution time cost. In general, mixing in MergeSort requires no more than $n-1$ comparisons for a vector of elements. The total cost is the sum of the cost of ordering two halves plus the cost of merging them. Therefore, in the worst case (and also in the average case) the cost of MergeSort is $T(n) = 2\ T(n/2)+\Theta(n) = \Theta(n\ log\ n)$.

Regarding memory cost, the typical implementation of MergeSort is not considered an in-place algorithm since the merge operation requires space $\Theta(n)$ for an auxiliary vector (vector T in Algorithm 2. Sequential merge operation.).

Since the cost of MergeSort coincides with the lower bound, MergeSort is said to be optimal. There are other optimal sorting algorithms (e.g., HeapSort), but

 DOI: https://doi.org/10.18272/aci.v17i1.3701

implementations of MergeSort have the property of stability, as equal elements maintain their position. This property, together with its relative simplicity, makes MergeSort attractive for several applications.

RELATED WORK

Several parallel implementations have been proposed for MergeSort. Rolfe [4] proposed an approach for implementing a parallel MergeSort in a distributed memory environment using MPI. The proposed algorithm did not consider a strategy for parallelization of the mixing operation. The experimental performance evaluation was not systematic. Few executions were reported, showing that a better speedup is obtained when the processes are executed within the same machine. This finding emphasizes the importance of considering communication overhead when assessing the performance of parallel implementations. Radensky [5] presented MergeSort implementations using OpenMP and MPI. However, the parallelization of the merge operation was not considered. The experimental evaluation showed that better performance was computed by the shared-memory implementation using OpenMP. Duvanenko [6] implemented a parallel MergeSort using the Microsoft PPL and Intel TBB libraries. Performance gains were reported for problem instances larger than 10^5 on a quad core machine. Axtmann et al. [7] studied how to scale MergeSort generalizations efficiently across a significant number of resources. The research aimed at addressing the challenges associated with handling large-scale applications by optimizing the communication aspects of the sorting algorithms.

The analysis of related works indicates that there is a better understanding of the sorting problem in theory than in practice. In this article, the distributed memory implementations follow a similar approach to Radensky [5], while the hybrid implementation incorporates parallel merging at the shared-memory level.

Parallelization of MergeSort

This section describes strategies for the developed parallel MergeSort versions.

Overall parallel approaches

MergeSort follows a divide-and-conquer strategy, and the natural technique for parallelization is domain decomposition. All parallel versions of MergeSort follow the strategy of executing in parallel on different partitions of the input vector.

Specific instructions are applied for creating threads (`fork`) or processes (`spawn`), assuming a dynamic multithreading environment following the style presented by Cormen et al. [3]. Communications and synchronizations are performed using sync and join instructions in the shared memory implementations, or via explicit message passing (`Send` and `Receive`) in the distributed memory implementations.

The intuitive semantics of `spawn` $f(x_1,\ldots,x_n)$ indicate that the execution of $f(x_1,\ldots,x_n)$ can take place in a new thread of execution, and there is no need to wait for its completion

DOI: https://doi.org/10.18272/aci.v17i1.3701

to proceed. Similarly, the intuitive semantics of `sync` instructs the current thread of execution to wait until all previously created threads have finished their execution. Some parallel programming platforms offer similar instructions to express nested parallelism, but the implementations use them as a means to convey the expected solution at an abstract level.

From a theoretical perspective, there is a lower bound for the comparison ordering problem in the parallel paradigm. Considering n processing units to sort n elements, no algorithm using the available n processors can achieve a sorting time less than a constant multiple of n comparisons in the worst case.

First version: Parallelizing recursive invocations

The first parallel version of MergeSort (MergeSortPv1) is based on parallelizing only recursive invocations. This version uses the sequential merge algorithm described in Algorithm 2. Sequential merge operation.. Algorithm 3. Parallel MergeSort (parallel recursive invocations). presents the pseudocode of MergeSortPv1.

```
procedure MERGESORTPV1(A, l, r)
    if l < r then
        m := ⌊(l + r)/2⌋
        spawn MERGESORTPV1(A, l, m)
        spawn MERGESORTPV1(A, m + 1, r)
        sync
        MERGE(A, l, m, r)
```

Algorithm 3. Parallel MergeSort (parallel recursive invocations).

The execution cost of MergeSortPv1 on a single processor matches the execution cost of MergeSort. Ideally, for an unlimited number of processing units, recursive invocations can execute in parallel, but the merge operation imposes a linear cost. The ideal parallelizability of this approach is given by Eq. i (1), where TP_i is the execution time on a parallel machine with processing units.

$$\frac{TP_1}{TP_\infty} = \frac{\Theta(n \log n)}{\Theta(n)} = \Theta(\log n) \quad (1)$$

Second version: Parallelizing the merge operation

The second version of parallel MergeSort (MergeSortPv2) is based on parallelizing the merge operation to improve the degree of parallelism of the first version of parallel MergeSort. In the previous version, the merge operation is the main limitation for parallelization, since it is inherently sequential. Thus, a new merging algorithm is used (MergeP, presented in Algorithm 4. Parallel merge operation.), which also applies the divide-and-conquer strategy and is, therefore, also naturally parallelizable.

DOI: https://doi.org/10.18272/aci.v17i1.3701

At a conceptual level, MergeP operates on two ordered vectors A and B (where $A.len \geq B.len$ is assumed, without loss of generality) as follows:

1. Divide A into two parts A_1 and A_2.

2. Let x be the first element of A_2. A binary search is performed on B to find the position h where x should be inserted into B. Then B is partitioned over h, generating two parts B_1 and B_2.

3. A_1 is merged with B_1 resulting in the first sorted half, and A_2 is merged with B_2 resulting in the second sorted half.

```
procedure MergeP(A[1..i], B[1..j], C[1..k])
    if j > i then                                        ▷ Ensure A.len ≥ B.len
        MergeP(B[1..j], A[1..i], C[1..k])
        return
    if k = 1 then
        C[1] := A[1]
        return
    if i = 1 then
        if A[1] ≤ B[1] then
            C[1] := A[1]
        else
            C[1] := B[1]
        return
    Search h such that B[h] ≤ A[i/2] ≤ B[h + 1].
    spawn MergeP(A[1..(i/2)], B[1..h], C[1..(i/2 + h)])
    spawn MergeP(A[(i/2 + 1)..i], B[(h + 1)..j], C[(i/2 + h + 1)..k])
    sync
    return C
```

Algorithm 4. Parallel merge operation.

Figure 1. Description of the logic in MergeP. shows an intuitive scheme of how the result C is constructed from the input vectors A and B in Algorithm 4. Parallel merge operation..

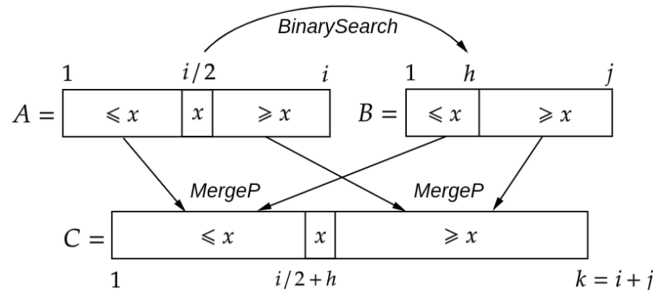

Figure 1. Description of the logic in MergeP.

 DOI: https://doi.org/10.18272/aci.v17i1.3701

The cost of executing MergeP on a single processor is asymptotically equal to the sequential version. However, assuming an unlimited number of process units, the cost is logarithmic rather than linear, i.e., $\Theta(\log^2 n)$, assuming that the search for position is performed in a binary way.

Thus, in the parallel version of MergeSort using MergeP instead of Merge, the cost on a single processor matches the cost of MergeSortPv1, but on an ideal machine the cost is $\Theta(\log^3 n)$. The ideal parallelizability of this approach is given by Eq. In comparison, the theory suggests that the second proposed version has the property of scaling better over additional computing resources, ignoring any overhead associated with parallelism..

$$\frac{TP_1}{TP_\infty} = \frac{\Theta(n \log n)}{\Theta(\log^3 n)} = \Theta\left(\frac{n}{\log^2 n}\right) \quad (2)$$

In comparison, the theory suggests that the second proposed version has the property of scaling better over additional computing resources, ignoring any overhead associated with parallelism.

Implementation details

This section describes the implementation details of the developed sequential and parallel versions of MergeSort. The code is publicly available on GitHub, https://github.com/josedusol/parallel-merge-sort.

Sequential implementation

`MergeSort.c` is the classic implementation of MergeSort in the C language, following Algorithm 1. Sequential MergeSort.. The best implementations of MergeSort typically apply another stable sorting algorithm (e.g. InsertionSort) for small inputs, taking advantage of improved efficiency on those inputs. This hybrid approach is not applied in the developed implementation, to guarantee a fair comparison with the developed parallel implementations of MergeSort.

Shared memory implementations

The shared memory implementations of MergeSort were developed using the C language and the OpenMP application programming interface (API). Two approaches provided by the OpenMP API were studied for the execution of code segments in parallel:

1. `sections` directive. Each recursive invocation is placed in a different section. Because there is an implicit synchronization barrier at the end of this directive, the merge operation is inserted after. This alternative is only applicable if the nested parallelism option `OMP_NESTED` is explicitly activated.

2. `task` directive, available from OpenMP v3.0 onward. This is the best alternative to support unstructured parallelism, such as the one arising from parallel recursion. In this case, each recursive invocation is a different task. For the merge operation,

 DOI: https://doi.org/10.18272/aci.v17i1.3701

it is necessary to also use `taskwait` which explicitly indicates the precedence between the split and merge tasks.

Preliminary experiments showed that `task`/`taskwait` allowed for computing better performance results. Thus, this approach was adopted for deployments.

`MergeSortPv1_omp.c` implements the MergeSortPv1 algorithm. The merge operation in the algorithm assumes that a new temporal vector is used for each invocation. However, a more efficient approach is to create a single vector before starting the sort and pass its address as a parameter to the involved routines. By doing so, the overhead associated with the initializing and releasing memory in each invocation is avoided. The overhead reduction implies that the same vector is used concurrently by different threads, accessing different portions during the merge process, without any risk of data races.

`MergeSortPv2_omp.c` implements the MergeSortPv2 algorithm, following the version presented by Cormen et al. [3]. The algorithm receives the original array A and an additional output array B to store the resulting values. However, it assumes that a new temporal vector T is used in each invocation. This vector is passed as the extra output array to the recursive calls, and the parallel merge operation is performed on T, storing the result in B. This approach has overhead due to initializing and releasing memory in each invocation, similar to the merge phase in the previous algorithm MergeSortPv1. To avoid this overhead and eliminate the need for additional vectors, a solution is to swap the roles of vectors A and B at each level of the recursion. In fact, without this optimization, the speedup achieved by this implementation is only marginally higher than that of the `MergeSortPv1_omp.c` implementation.

In both implementations, creating OpenMP tasks at each level of the recursion introduces an overhead that degrades the expected performance gains, even though creating a task does not mean creating a new operating system thread. To mitigate this negative factor, one approach is to limit the application of parallelism based on either the size of the input vector or the depth of the recursion. The developed implementations apply the restriction based on the size of the input vector.

On the one hand, in the implementation of `MergeSortPv1_omp.c`, MergeSort is invoked sequentially for sizes smaller than a threshold value `CUT_OFF`. On the other hand, there are two different levels of recursion to control in `MergeSortPv2_omp.c`: initially, MergeSort is executed sequentially for sizes less than a threshold value `CUT_OFF_1`, and the sequential Merge is called for the total sizes (i.e., the sum of the sizes of two vectors) less than a threshold value `CUT_OFF_2`.

The optimal values for thresholds `CUT_OFF`, `CUT_OFF_1`, and `CUT_OFF_2` are platform-dependent and are determined through empirical testing and experimentation. These values are chosen in an ad-hoc manner to achieve the best performance on the specific computing platform.

Distributed memory implementations

The distributed memory implementations of MergeSort were developed using the C language and the MPI library. Three implementations were developed, applying a

 DOI: https://doi.org/10.18272/aci.v17i1.3701

straightforward master-slave approach, a functional decomposition approach, and a hybrid approach combining distributed and shared memory for efficient computation. The developed implementations are described next.

`MergeSortPv0_mpi.c` is a distributed memory implementation of MergeSort based on worksharing. A master-slave hierarchical architecture is applied. The master evenly divides the input vector and distributes chunks using `MPI_Scatter` among the available processes so that each process performs a local sequential sorting in the assigned chunk. The processes return the sorted parts to the master process using `MPI_Gather`, and the master performs a final sequential sort. Although this implementation does not correspond exactly to the theoretical algorithms MergeSortPv1 and MergeSortPv2, it is the most direct and elementary parallelization of the problem using MPI, and therefore it is interesting to consider it for simplicity as a baseline for the comparison of efficiency results.

`MergeSortPv1_mpi.c` is an implementation of the MergeSortPv1 algorithm, applying a functional decomposition technique [5].

A virtual tree of processes that represents the tree of recursive calls is built. At each level, different processes work in parallel either to divide or to merge. The depth of the process tree is limited by the number of available processes. The base case in the construction of the tree is to invoke MergeSort sequentially for the leave nodes of the tree. Figure 2. Virtual tree of four processes to sort vector . presents a scheme of the functional decomposition technique when using four processes on a vector of eight elements. The dashed lines indicate that the process retains the first half of the vector, and the bi-directional arrows indicate that the process should send the second half of the vector to an auxiliary process. The four lists of size 2 on the leaves of the tree are ordered sequentially. Asynchronous communications are performed, using `MPI_Isend`, to properly overlap data transmission and computation. Then, the process receives the ordered half from the auxiliary process using a standard `MPI_Recv` routine.

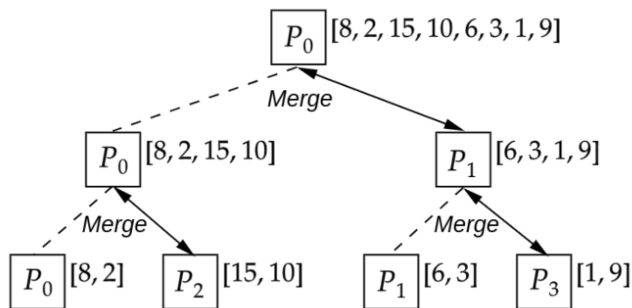

Figure 2. Virtual tree of four processes to sort vector .

`MergeSortPv2_hyb.c` is a hybrid implementation of the MergeSortPv2 algorithm using MPI and OpenMP. The data splitting phase is implemented using MPI, as performed in `MergeSortPv1_mpi.c`, and the merge operation is implemented using OpenMP, as

DOI: https://doi.org/10.18272/aci.v17i1.3701

performed in `MergeSortPv2_omp.c`. Additionally, OpenMP is also used in the base case of the virtual process tree. Parallelism restrictions with the values `CUT_OFF_1` and `CUT_OFF_2` are also applied to mitigate the overhead associated with threads creation and management in OpenMP.

Experimental evaluation

This section describes the empirical analysis of the developed parallel MergeSort implementations.

METHODOLOGY

The experimental evaluation was performed over a large vector of size with random integers. The memory required to store the vector was roughly 3 GB. Every experiment was repeated five times to avoid fluctuating execution times. If applicable, outliers were detected and discarded. The overall execution time was the average of non-discarded execution times.

Evaluation of shared memory implementations. For shared memory implementations, the main configurable parameter is the number of threads to use, set by the environment variable `OMP_NUM_THREADS`. The experiments studied the performance using a number of threads in the range of 1 to 39. The number of threads is always matched to the number of CPU cores available. The execution time (walltime) is measured using the `omp_get_wtime` function.

Evaluation of distributed memory implementations. The distributed memory implementations are executed through `mpiexec`/`mpirun`. The main configurable parameter is `np`, indicating the number of processes for execution. The experiments studied the performance using a number of processes in the range of 1 to 16. In particular, for the `MergeSortPv0_imp.c` implementation, only those numbers of processes that divide the size of the vector were used (1, 2, 4, 5, 8, 10, and 16). On the other hand, for the hybrid implementation `MergeSortPv2_hyb.c`, four CPU cores are always requested per process, and OpenMP is configured to use four threads. Additionally, the hybrid implementation was executed in a distributed manner in two different nodes. The walltime is measured using the `MPI_Wtime` function. To make this measurement meaningful, a barrier (`MPI_Barrier`) was included before and after performing the sort, taking the initial time after the first barrier and the final time after the second barrier.

Performance evaluation metrics

The execution time is used as the main metric for performance evaluation. The performance evaluation of parallelism is focused on strong scalability, i.e., how the execution time varies when using a different number of processing units for a fixed input size. The goal is to reduce the execution time.

The traditional approach applying the algorithmic speedup is applied to evaluate performance. Speedup computes the ratio of the execution time of a baseline sequential

 DOI: https://doi.org/10.18272/aci.v17i1.3701

implementation T_S and the execution time of the parallel algorithm using N compute resources TP_N (Eq. (3)) In this case, the sequential algorithm is `MergeSort.c`, described before. Computational efficiency is the normalized version of the speedup (Eq. Development and execution platform).

$$S_N = \frac{T_S}{TP_N} \tag{3}$$

$$E_N = \frac{S_N}{N} \tag{4}$$

Development and execution platform

The proposed algorithms were developed in the C language, using the GCC v8 compiler. Initial local developments were performed using the MPICH v3.2 implementation of MPI, and the final development was performed using the OpenMPI implementation of MPI.

The experimental evaluation was performed on the high-performance computing infrastructure of National Supercomputing Center, Uruguay (ClusterUY) [8]. Each computing node has two Intel Xeon Gold 6138 CPUs with 20 cores each and 128 GB of RAM. Nodes are connected by 10 Gbps Ethernet. Executions were performed without using hyperthreading.

Efficiency results

This subsection reports on the efficiency results of the developed parallel versions of MergeSort. For OpenMP-based implementations, the following values were determined in preliminary experiments to mitigate overheads: CUT_OFF = 800 in `MergeSortPv1_omp.c` and the hybrid implementation `MergeSortPv2_hyb.c`, and CUT_OFF_1=800, CUT_OFF_2=80000 in `MergeSortPv2_omp.c` and the hybrid implementation `MergeSortPv2_hyb.c`.

Shared-memory implementations. Table 1 reports the average execution times of the shared memory implementations using OpenMP. For reference, the average execution time of the sequential implementation `MergeSort.c` for n=10^9 w was 318.67s. Figure 3 graphically presents the evolution of execution times.

 DOI: https://doi.org/10.18272/aci.v17i1.3701

Table 1. Execution time (s) for different numbers of computing resources (threads) for shared memory MergeSort implementations using OpenMP.

implementation	computing resources					
	1	2	4	8	16	32
`MergeSortPv1_omp`	274.96	153.68	115.14	83.49	65.66	47.42
`MergeSortPv2_omp`	231.69	124.86	83.48	49.43	27.85	17.59

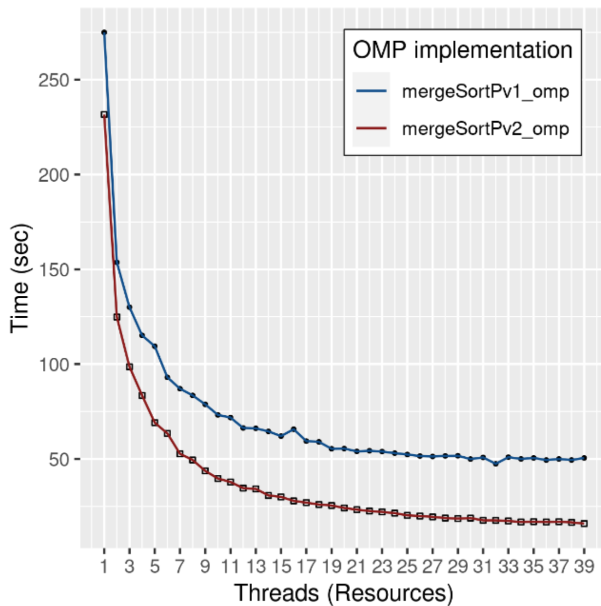

Figure 3. Execution time of shared memory OpenMP MergeSort implementations.

Tables 2 and 3 report the speedup and efficiency values computed for the shared memory implementations using OpenMP. Figure 4 graphically presents the graphs of speedup and efficiency as a function of the number of threads.

Table 2. Speedup for different number of computing resources (threads) for shared memory MergeSort implementations using OpenMP.

implementation	speedup					
	S_1	S_2	S_4	S_8	S_{16}	S_{32}
`MergeSortPv1_omp`	1.16	2.07	2.77	3.82	4.85	6.72
`MergeSortPv2_omp`	1.38	2.55	3.82	6.45	11.44	18.11

DOI: https://doi.org/10.18272/aci.v17i1.3701

Table 3. Efficiency for different number of computing resources (threads) for shared memory MergeSort implementations using OpenMP.

implementation	efficiency					
	E_1	E_2	E_4	E_8	E_{16}	E_{32}
`MergeSortPv1_omp`	1.16	1.04	0.69	0.48	0.30	0.21
`MergeSortPv2_omp`	1.38	1.28	0.95	0.81	0.72	0.57

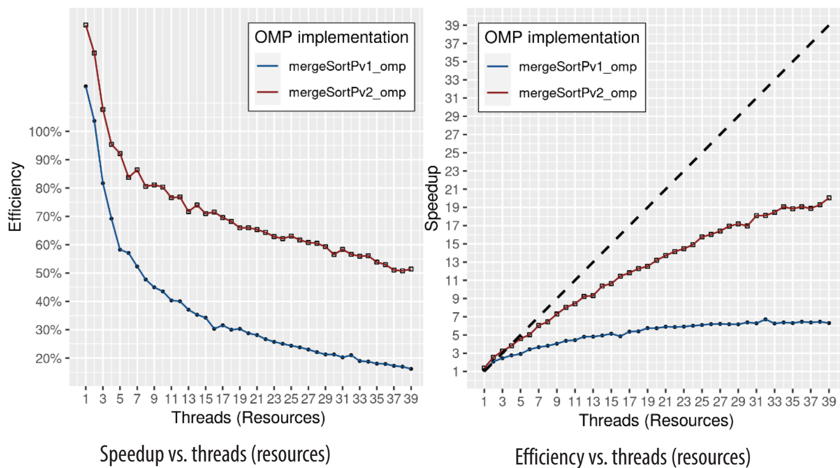

Figure 4. Speedup and efficiency for the shared memory MergeSort implementations using OpenMP. The dotted line is the linear speedup.

The results of the performance analysis indicate that the `MergeSortPv1_omp.c` implementation showed a high speedup when using a small number of threads (≤3). However, as the number of threads increased, the implementation deviated from the theoretically optimal performance given by the linear speedup. Using four threads, the efficiency dropped below 0.7. Moreover, when using more than 20 threads, the speedup tended to stagnate. This result aligns with Amdahl's Law, which suggests that sequential work becomes a limiting factor despite the availability of additional processing units. For the considered implementation, the limiting factor is the linear-cost merge operation, which acts as the bottleneck of the implementation.

In contrast, the `MergeSortPv2_omp.c` implementation demonstrated a remarkable speedup with consistent growth. Unlike the previous implementation, it showed no signs of stagnation and achieved the maximum speedup value when using the largest number of threads (39). However, it is anticipated that this upward trend will diminish or even reverse as the number of threads further increases. The efficiency of this implementation only dropped below 0.7 when using more than 20 threads.

Distributed-memory implementations. Table 4 reports the average execution times of the distributed memory implementations of MergeSort using MPI. Figure 5 graphically presents the evolution of execution times.

DOI: https://doi.org/10.18272/aci.v17i1.3701

Table 4. Execution time (s) for different number of computing resources (processes) for distributed memory MPI MergeSort implementations.

implementation	computing resources				
	1	2	4	8	16
MergeSortPv0_mpi	236.06	286.17	222.13	201.62	211.97
MergeSortPv1_mpi	281.88	287.91	221.54	159.27	117.59
MergeSortPv2_hyb	254.05	264.01	73.97	54.68	41.29

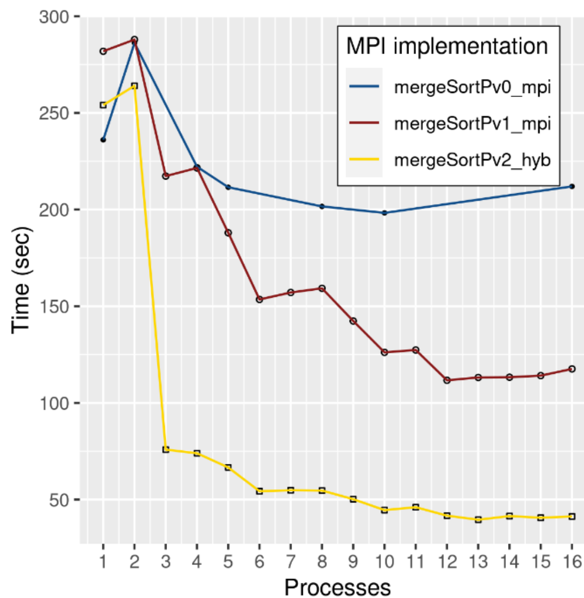

Figure 5. Execution times for distributed memory MPI MergeSort implementations.

Tables 5 and 6 report the speedup and efficiency of the distributed memory MergeSort implementations using MPI. Figure 6 graphically presents the speedup and efficiency as a function of the number of computing resources/processes.

Table 5. Speedup of distributed memory MPI MergeSort implementations.

implementation	speedup				
	S_1	S_2	S_4	S_8	S_{16}
MergeSortPv0_mpi	1.35	1.11	1.43	1.58	1.50
MergeSortPv1_mpi	1.13	1.11	1.44	2.00	2.71
MergeSortPv2_hyb	1.25	1.21	4.31	5.83	7.72

 DOI: https://doi.org/10.18272/aci.v17i1.3701

Table 6. Efficiency of distributed memory MPI MergeSort implementations.

implementation	efficiency				
	E_1	E_2	E_4	E_8	E_{16}
`MergeSortPv0_mpi`	1.35	0.56	0.36	0.10	0.09
`MergeSortPv1_mpi`	1.13	0.55	0.36	0.25	0.17
`MergeSortPv2_hyb`	1.25	0.60	1.08	0.73	0.48

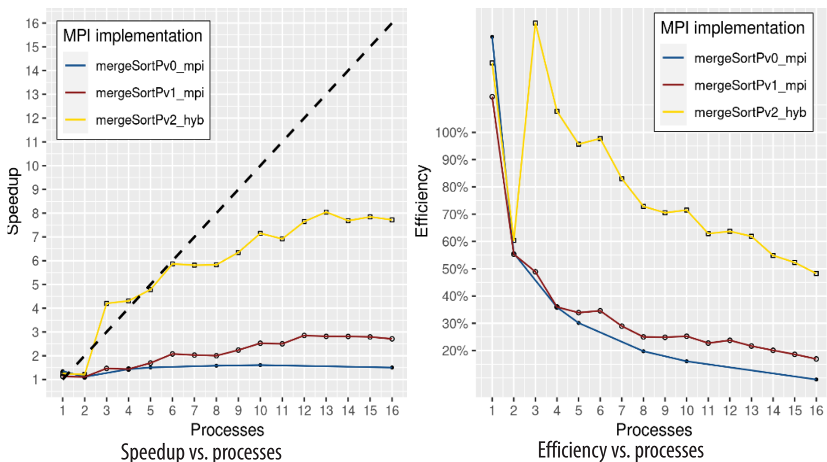

Figure 6. Speedup and efficiency for distributed memory MergeSort implementations using MPI. The dotted line is the linear speedup.

The results of the performance analysis indicate that the speedup achieved by the `MergeSortPv0_mpi.c` implementation was notably poor. Comparatively, the `MergeSortPv1_mpi.c` implementation demonstrated a slightly improved speedup, but its growth rate remained quite slow. On the other hand, the hybrid implementation `MergeSortPv2_hyb.c`, despite initially exhibiting unexpectedly poor performance when using only two processes, showed a better performance overall as the number of processes increased. This is a relevant result since the hybrid implementation was executed on different computing nodes, implying that the other developed implementations had not experienced significant inter-node communication costs.

When comparing the two groups of implementations, it is evident that superior results were achieved using OpenMP. In fact, the best-performing MPI-based implementation was the hybrid approach that combined MPI with OpenMP. Both groups had specific cases where superlinear speedup was achieved, i.e., the performance improvement exceeded the expected linear scaling. Overall, the utilization of OpenMP has proven to be more effective in terms of performance, and the presence of almost linear speedup values highlights the potential efficiency gains achievable through parallelization.

 DOI: https://doi.org/10.18272/aci.v17i1.3701

CONCLUSIONS AND FUTURE WORK

This article presented an empirical evaluation of parallel versions of MergeSort, applying shared memory and distributed memory approaches.

The standard MergeSort algorithm has inherent parallelizability, primarily because of the independence of its recursive calls. Both theoretical analysis and experimental evidence suggest that even greater advantages are achievable by recognizing the parallel nature of the merging operation, besides the recursive calls in MergeSort.

The empirical analysis of the developed parallel implementations demonstrated that the most favorable performance results were obtained when using a shared memory implementation using OpenMP, in line with previous results from related works. Regarding distributed memory implementations, the best performance results were achieved by a hybrid implementation executing on different computing nodes.

The main lines for future work are related to extending the empirical evaluation and analyzing the scalability for larger and different types of inputs, and analyzing methods for finding the best values of `CUT_OFF_1` and `CUT_OFF_2` parameters. In turn, a specific parallelization technique can be applied for distributed memory implementations to process the virtual process tree, adapted to parallelize the merge operation.

AUTHOR CONTRIBUTIONS

José Solsona conceived the research and conducted the experiment. Sergio Nesmachnow contributed to the manuscript.

CONFLICT OF INTEREST

The authors declare no conflict of interest related to the publication of this article. All authors have read and approved the final manuscript and consent to its publication.

DOI: https://doi.org/10.18272/aci.v17i1.3701

aci avances en ciencias e ingenierías

REFERENCES

[1] Knuth, D. E. (1997). *The art of computer programming, volume 3: Sorting and searching* (2nd ed.). Addison–Wesley.

[2] Katajainen, J., & Träff, J. L. (1997). Algorithms and complexity. In *Proceedings of the 3rd Italian Conference on Algorithms and Complexity.*

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd. ed.). The MIT Press.

[4] Rolfe, T. J. (2010). A specimen of parallel programming: Parallel merge sort implementation. *ACM Inroads, 1*(4), 72–79. http://doi.org/10.1145/1869746.1869767

[5] Radenski, A. (2011). Shared memory, message passing, and hybrid merge sorts for standalone and clustered SMPs. In *Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11)* (pp. 367–373).

[6] Duvanenko, V. (2011). Parallel merge sort. https://duvanenko.tech.blog/2018/01/13/parallel-merge-sort/

[7] Axtmann, M., Bingmann, T., Sanders, P., & Schulz, C. (2015). Practical massively parallel sorting. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures* (pp. 13–23). https://doi.org/10.1145/2755573.2755595

[8] Nesmachnow, S., & Iturriaga, S. (2019). Cluster-UY: Collaborative scientific high performance computing in Uruguay. In M. Torres & J. Klapp (Eds.), *Supercomputing* (pp. 188–202). Springer. https://doi.org/10.1007/978-3-030-38043-4_16

 DOI: https://doi.org/10.18272/aci.v17i1.3701