**PAR Laboratory Assignment** Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Marc Duran Lopez: par1311

Jesús Pérez Bermejo: par1317

Spring 2022-23

May 19, 2023

ÍNDICE

[**1. Task decomposition analysis for MergeSort 3**](#_kg8jjcj8lxrb)

[Task decomposition analysis with Tareador 3](#_l3kv95p44tmi)

[Leaf strategy in OpenOMP 3](#_ctkdhep108y6)

[Tree strategy in OpenOMP 4](#_z2f9wb4p5riq)

[**2. Shared-memory parallelism with OpenMP tasks 8**](#_sfcb5qvuotib)

[Leaf strategy in OpenMP 8](#_hhagy8yuos5j)

[Tree strategy in OpenMP 13](#_58hhmx8t97sd)

[Task granularity control: the cut-off mechanism 18](#_by3fmymozxfq)

[**3. Shared-memory parallelisation with OpenMP task using dependencies 23**](#_sho6iqn9dcek)

[**Conclusion 27**](#_h4icffgbqovb)

# 1. Task decomposition analysis for MergeSort

In this first session, we will study the multisort-tareador.c file and check the code for potential concurrency issues using the Tareador tool. These issues will be dealt using OMP directives that we must consider in the following sessions.

## Task decomposition analysis with Tareador

Now, we will investigate, using the Tareador tool, potential task decomposition strategies and their implications in terms of parallelism and task interaction required of multisort-tareador.c.

Specifically, we will be using two strategies for this code: Leaf strategy and Tree strategy.

### Leaf strategy in OpenOMP

To implement this strategy, we incorporate the task creation within the code section where the functions "basicsort" and "basicmerge" are invoked. The modified code, which includes the creation of tasks, can be observed in the next figure. (Fig. 1)

**void merge(long n, T left[n], T right[n], T result[n\*2], long start, long length) {**

**if (length < MIN\_MERGE\_SIZE\*2L) {**

**// Base case**

**tareador\_start\_task("basicmerge");**

**basicmerge(n, left, right, result, start, length);**

**tareador\_end\_task("basicmerge");**

**} else {**

**// Recursive decomposition**

**merge(n, left, right, result, start, length/2);**

**merge(n, left, right, result, start + length/2, length/2);**

**}**

**}**

**void multisort(long n, T data[n], T tmp[n]) {**

**if (n >= MIN\_SORT\_SIZE\*4L) {**

**// Recursive decomposition**

**multisort(n/4L, &data[0], &tmp[0]);**

**multisort(n/4L, &data[n/4L], &tmp[n/4L]);**

**multisort(n/4L, &data[n/2L], &tmp[n/2L]);**

**multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L]);**

**merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);**

**merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L);**

**merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);**

**} else {**

**// Base case**

**tareador\_start\_task("basicsort");**

**basicsort(n, data);**

**tareador\_end\_task("basicsort");**

**}**

**}**

*Fig.1. Code modified for Leaf strategy*

### Tree strategy in OpenOMP

In this approach, we designate the task within the code segment that calls each function involved in the recursive decomposition phase, namely "merge" and "multisort". The updated code, which includes the task creation, is depicted in the following figure (Fig. 2).

**void merge(long n, T left[n], T right[n], T result[n\*2], long start, long length) {**

**if (length < MIN\_MERGE\_SIZE\*2L) {**

**// Base case**

**basicmerge(n, left, right, result, start, length);**

**} else {**

**// Recursive decomposition**

**tareador\_start\_task("merge\_start");**

**merge(n, left, right, result, start, length/2);**

**tareador\_end\_task("merge\_end");**

**tareador\_start\_task("merge\_start");**

**merge(n, left, right, result, start + length/2, length/2);**

**tareador\_end\_task("merge\_end");**

**}**

**}**

**void multisort(long n, T data[n], T tmp[n]) {**

**if (n >= MIN\_SORT\_SIZE\*4L) {**

**// Recursive decomposition**

**tareador\_start\_task("multisort\_start");**

**multisort(n/4L, &data[0], &tmp[0]);**

**tareador\_end\_task("multisort\_end");**

**tareador\_start\_task("multisort\_start");**

**multisort(n/4L, &data[n/4L], &tmp[n/4L]);**

**tareador\_end\_task("multisort\_end");**

**tareador\_start\_task("multisort\_start");**

**multisort(n/4L, &data[n/2L], &tmp[n/2L]);**

**tareador\_end\_task("multisort\_end");**

**tareador\_start\_task("multisort\_start");**

**multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L]);**

**tareador\_end\_task("multisort\_end");**

**tareador\_start\_task("merge\_start");**

**merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);**

**tareador\_end\_task("merge\_end");**

**tareador\_start\_task("merge\_start");**

**merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L);**

**tareador\_end\_task("merge2\_end");**

**tareador\_start\_task("merge\_start");**

**merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);**

**tareador\_end\_task("merge\_end");**

**} else {**

**// Base case**

**basicsort(n, data);**

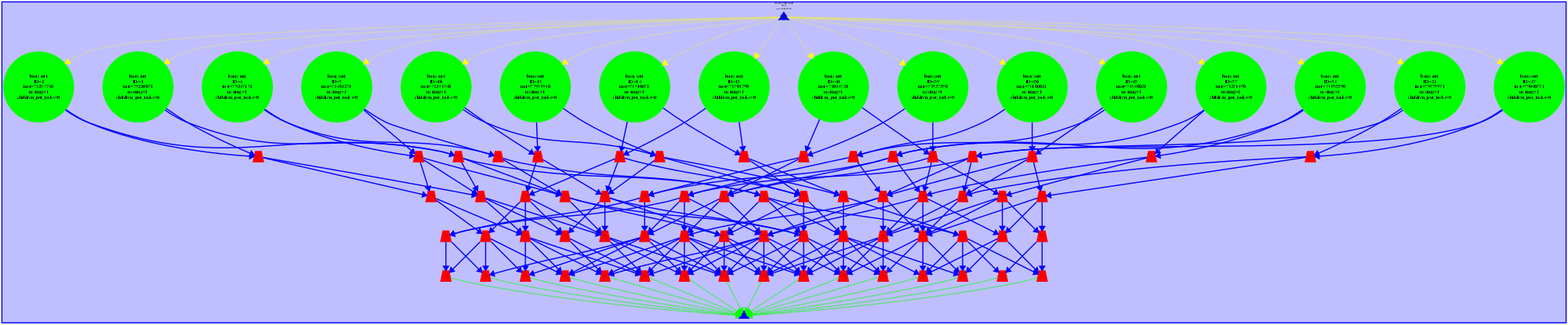
**}**

**}**

*Fig.2. Code modified for Tree strategy*

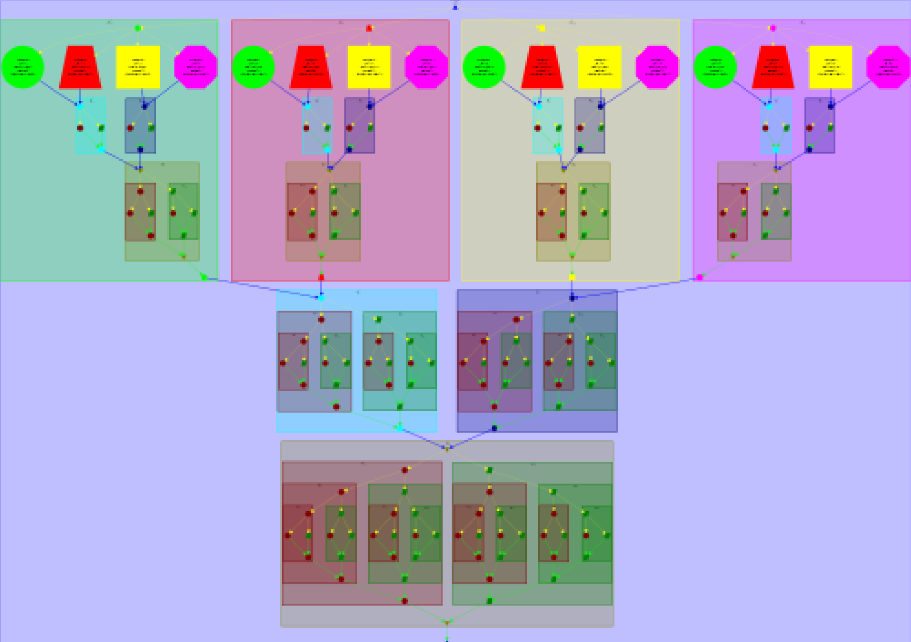
The next step involves executing the two strategies using Tareador to visualize the dependency graphs generated by the script for each strategy.

We obtain the following dependency graph for the Leaf Strategy (Fig. 3).



*Fig.3. TDG Leaf strategy*

For the Tree Strategy, we obtain the following task dependence graph (Fig. 4).



*Fig.4. TDG Tree strategy*

Upon examining the two task dependence graphs (TDGs), noticeable differences between each strategy become apparent. Firstly, it is evident that the Tree strategy has generated a greater number of tasks compared to the Leaf strategy. This discrepancy arises from the increased invocations of the "merge" and "multisort" functions in the Tree strategy, which occur more frequently due to the recursive nature of the algorithm.

Additionally, it is noticeable that the Tree strategy exhibits greater variation in the granularity of its tasks. The initial tasks in the Tree strategy tend to have a larger granularity compared to the later tasks, resulting in an imbalance among the tasks. A similar imbalance can also be observed in the Leaf strategy, where the basicsort tasks (represented by green tasks) are significantly larger in size than the basicmerge tasks (represented by red tasks).

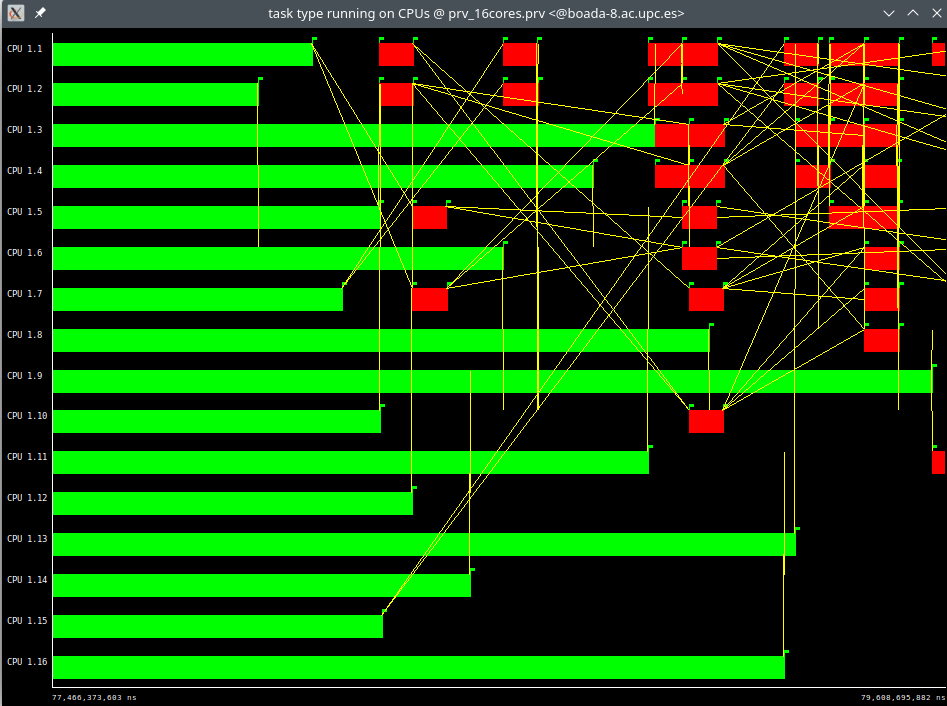
Furthermore, the structure of the dependency graphs also differs between the strategies. In the Leaf strategy, tasks are exclusively generated at the leaf level. This means that tasks are only created in the base cases of the "merge" and "sort" functions, rather than in the recursive cases. On the other hand, the Tree strategy generates tasks at multiple levels, including both the base cases and the recursive cases, resulting in a more extensive task hierarchy within the graph.

In the Leaf strategy, we observe that dependencies arise due to the "basicmerge" function, which relies on the two halves of the arrays to be pre-sorted by the "basicsort" function. Similarly, in the Tree strategy, we observe similar dependencies. Each recursive invocation of the "merge" function necessitates that its array parameter be sorted beforehand.

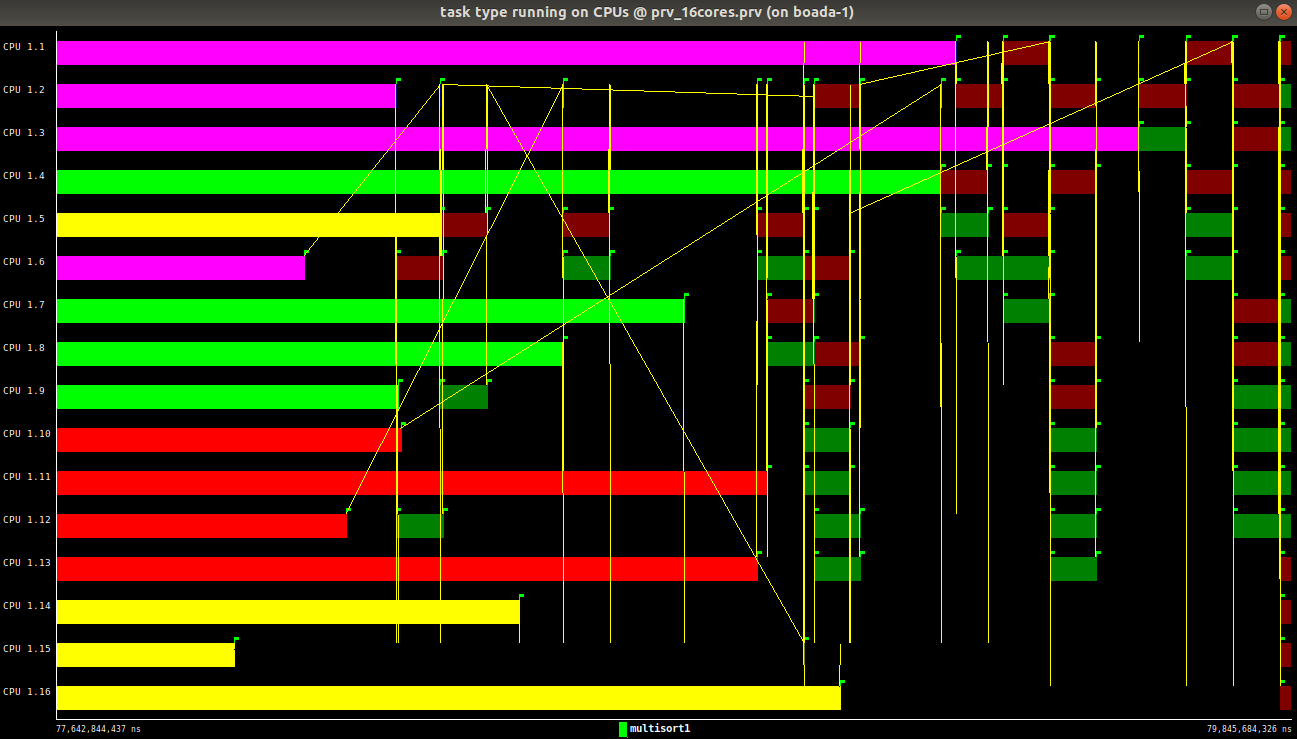
Upon analyzing the code, we can infer that in the Leaf decomposition strategy, the need for synchronization arises from the aforementioned dependencies. This issue can be addressed by incorporating taskgroup directives. One taskgroup directive can be added to encompass all recursive calls of the "multisort" function, while another taskgroup directive can encompass the initial two "merge" calls. By implementing these directives, synchronization among the tasks can be achieved effectively.

In the case of the Tree decomposition strategy, the need for synchronization also arises from the "merge" functions, which require the array parameters to be sorted by the "mergesort" functions. To address this, similar to the Leaf strategy, we can resolve the synchronization issue by incorporating taskgroup directives. By adding taskgroup directives, we can ensure proper synchronization and ordering of the tasks within the Tree strategy.

Finally, we simulate the execution of both strategies using Paraver with 16 processors. The subsequent images represent zoomed sections of the traces obtained from the executions of the leaf and tree strategies, respectively.



*Fig.5. Paraver trace leaf strategy*



*Fig.6. Paraver trace tree strategy*

# 2. Shared-memory parallelism with OpenMP tasks

In this section we are working with the multisort-omp.c file. We want to investigate two parallelization approaches.

## Leaf strategy in OpenMP

First of all, we start with the Leaf strategy.

To implement this strategy, we introduce the directives "#pragma omp task" to the base case of the "multisort" function and the base case of the "merge" function. These directives allow us to create the leaf tasks. To prevent data races, we incorporate a synchronization mechanism using the "#pragma omp taskwait" directive before the merge calls within the iterative part of the "multisort" function. This ensures that the tasks are synchronized appropriately before proceeding with the merge operations.

Finally, we include the directives "#pragma omp parallel" and "#pragma omp single" before the invocation of the "multisort" function in the main body. The "#pragma omp parallel" directive establishes a parallel region, allowing multiple threads to execute the code within it. The "#pragma omp single" directive ensures that only one thread creates the tasks, preventing redundant task creation by multiple threads. This combination of directives enables parallel execution of the code while maintaining control over task creation.

**void merge(long n, T left[n], T right[n], T result[n\*2], long start, long length) {**

**if (length < MIN\_MERGE\_SIZE\*2L) {**

**// Base case**

**#pragma omp task**

**basicmerge(n, left, right, result, start, length);**

**} else {**

**// Recursive decomposition**

**merge(n, left, right, result, start, length/2);**

**merge(n, left, right, result, start + length/2, length/2);**

**}**

**}**

**void multisort(long n, T data[n], T tmp[n]) {**

**if (n >= MIN\_SORT\_SIZE\*4L) {**

**// Recursive decomposition**

**multisort(n/4L, &data[0], &tmp[0]);**

**multisort(n/4L, &data[n/4L], &tmp[n/4L]);**

**multisort(n/4L, &data[n/2L], &tmp[n/2L]);**

**multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L]);**

**#pragma omp taskwait**

**merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);**

**merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L);**

**#pragma omp taskwait**

**merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);**

**} else {**

**// Base case**

**#pragma omp task**

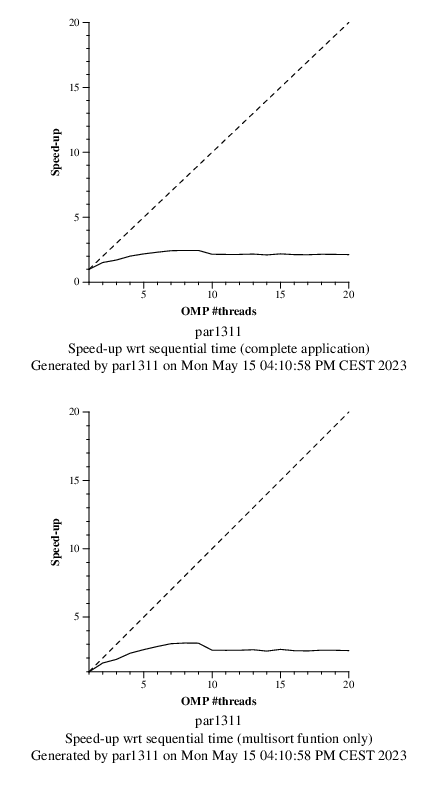
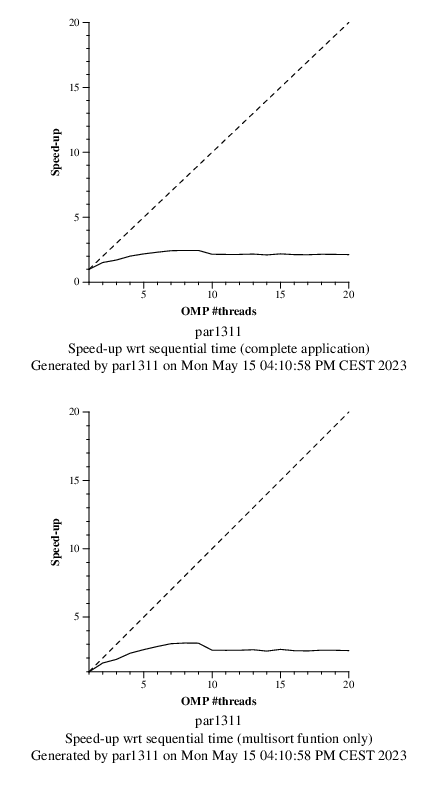
**basicsort(n, data);**

**}**

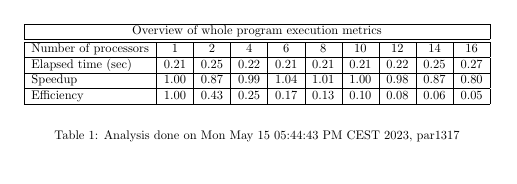
**}**

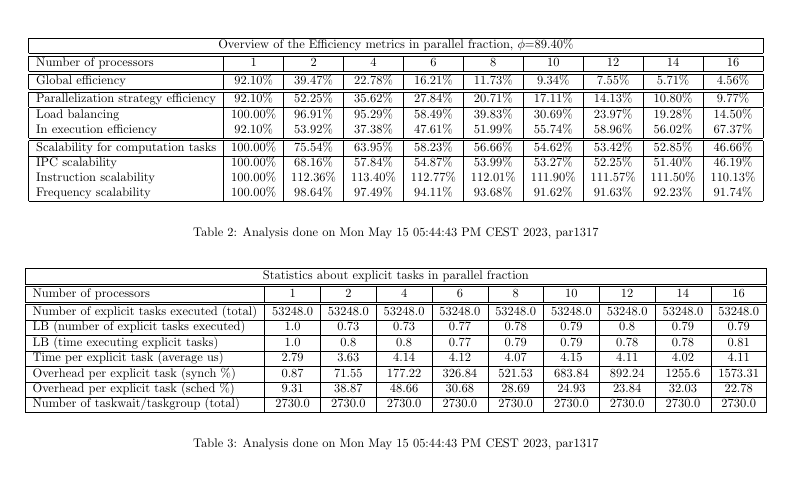
*Fig.7. Leaf strategy code modification*

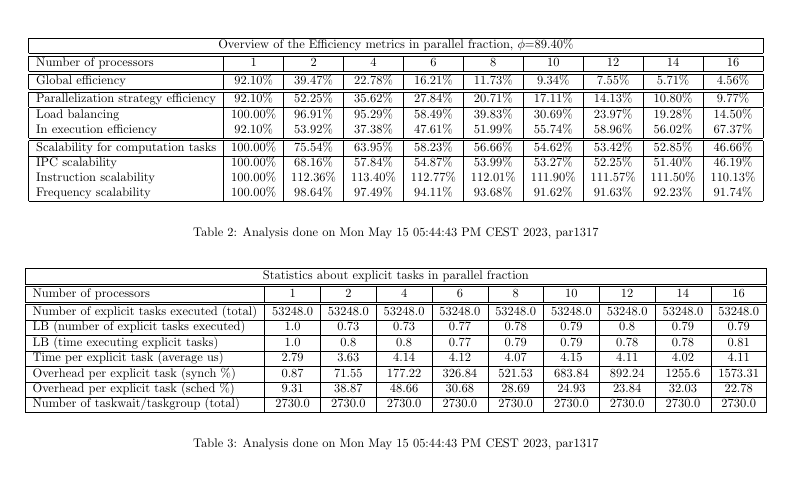
As we can see on the plots below, it is evident that the achieved speedups are not as significant as the anticipated speedups typically observed in parallel codes. This discrepancy can be attributed to the fact that the leaf strategy results in a substantial portion of tasks being executed sequentially, rather than in parallel.

*Fig.8. Leaf strategy plots of speed-up*

Following that, we conducted a modelfactors analysis for this strategy, and the obtained tables are as follows:



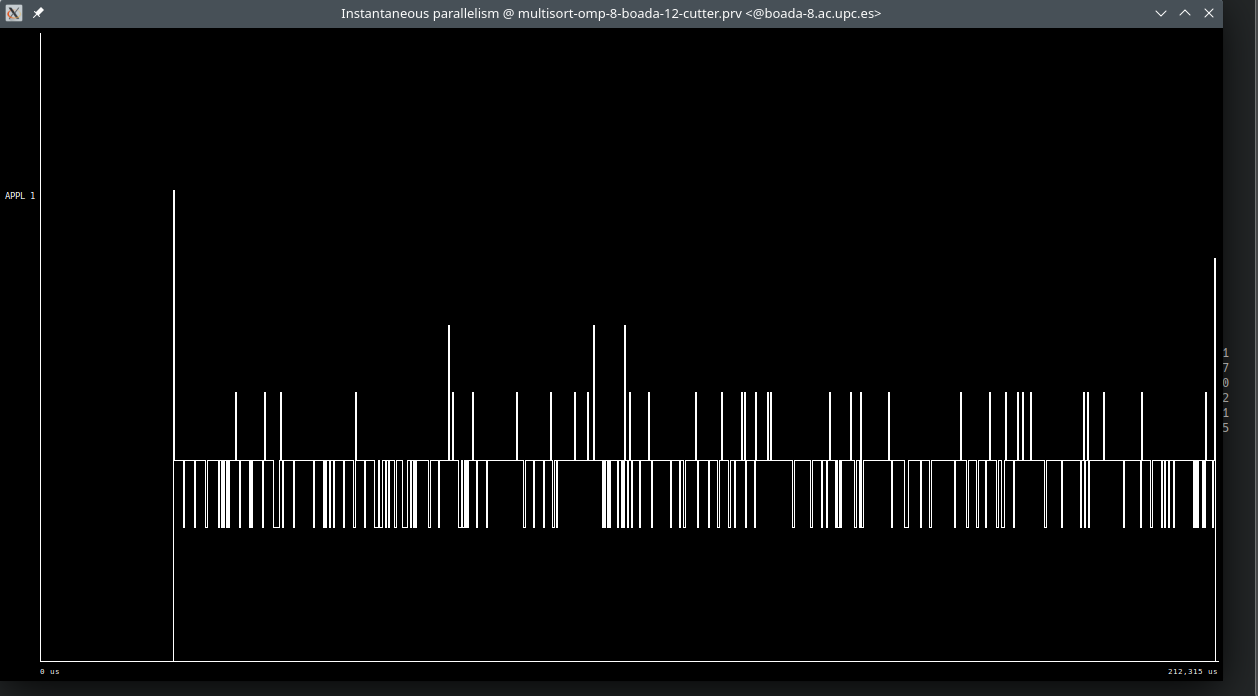




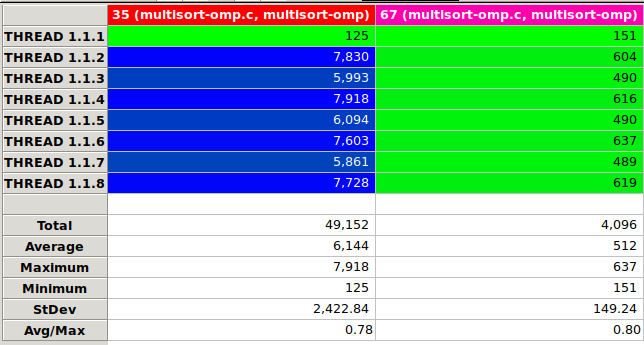
*Fig.9. Leaf strategy tables of the modelfactors analysis*

Observing the tables, it is apparent that both the global efficiency and the efficiency of the parallelization strategy decrease significantly as more threads are added to the execution. We believe that these low values are attributed to the increased overhead per explicit task due to thread synchronization. Contrary to efficiency, the overheads per task substantially increase as more threads are introduced, leading to diminished performance.

Lastly, we performed a Paraver analysis to delve deeper into the execution of the leaf strategy. The analysis obtaining the following timelines and histograms, which provide detailed perpectives into the execution process.



*Fig.10. Trace of instantaneous parallelisation of leaf strategy*



*Fig.11. Histogram of task execution of leaf strategy*

## 

## 

## 

*Fig.12. Trace of task creation and execution of leaf strategy*

## 

If we observe the histograms above, we can conclude that thread one is primarily responsible for the creation of tasks. This is evident from the observation that it executes fewer tasks compared to the other threads, which distribute the workload more evenly among themselves.

The program generates an ample number of tasks for every thread, except for the first thread which is specifically dedicated to task creation. On average, this first thread concurrently executes around 7-8 tasks throughout the program, excluding the time spent on synchronization. It is worth noting that the synchronization process accounts for a significant portion of the program's execution time.

## 

## Tree strategy in OpenMP

Now, let’s repeat the process with the Tree strategy, aiming for an execution time of less than half of that of the Leaf strategy. In this case, we will use the task group directive, ensuring that the tasks wait for their ancestors before executing.

void merge(long n, T left[n], T right[n], T result[n\*2], long start, long length) {

if (length < MIN\_MERGE\_SIZE\*2L) {

// Base case

basicmerge(n, left, right, result, start, length);

} else {

// Recursive decomposition

#pragma omp task

merge(n, left, right, result, start, length/2);

#pragma omp task

merge(n, left, right, result, start + length/2, length/2);

}

}

void multisort(long n, T data[n], T tmp[n]) {

if (n >= MIN\_SORT\_SIZE\*4L) {

// Recursive decomposition

#pragma omp taskgroup

{

#pragma omp task

multisort(n/4L, &data[0], &tmp[0]);

#pragma omp task

multisort(n/4L, &data[n/4L], &tmp[n/4L]);

#pragma omp task

multisort(n/4L, &data[n/2L], &tmp[n/2L]);

#pragma omp task

multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L]);

}

#pragma omp taskgroup

{

#pragma omp task

merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);

#pragma omp task

merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L);

}

#pragma omp task

merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);

} else {

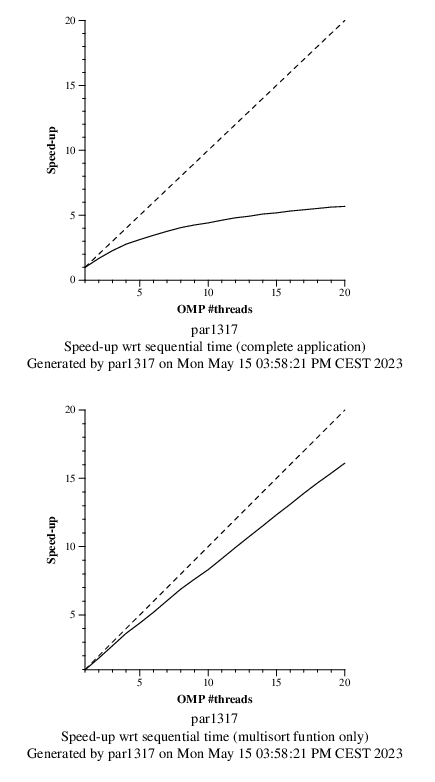
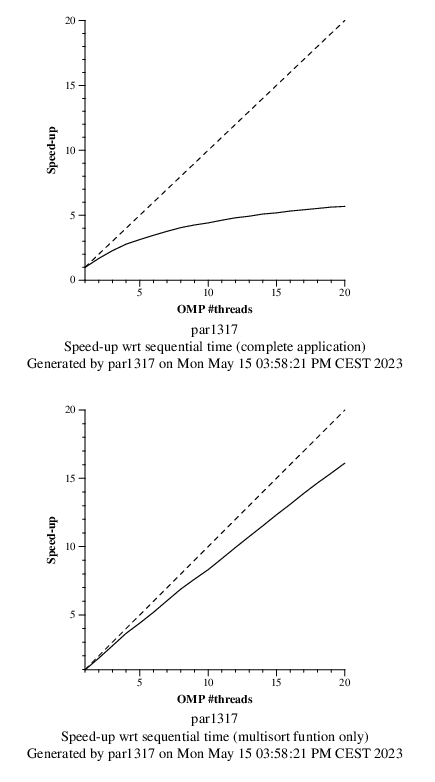
// Base case

basicsort(n, data);

}

}

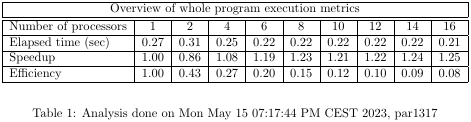
*Fig.13. Tree strategy code modification*

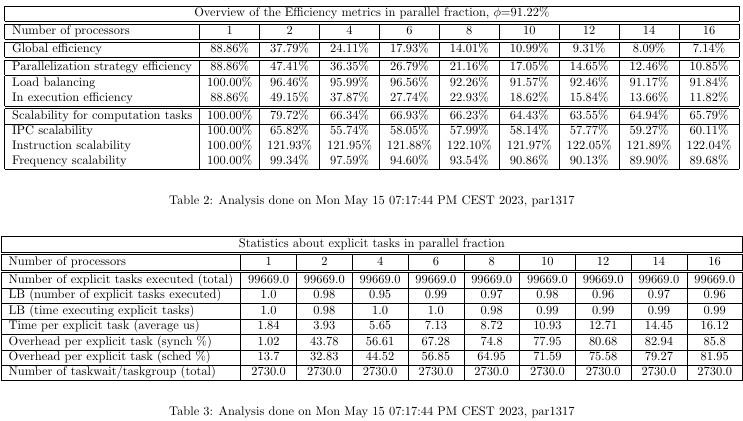


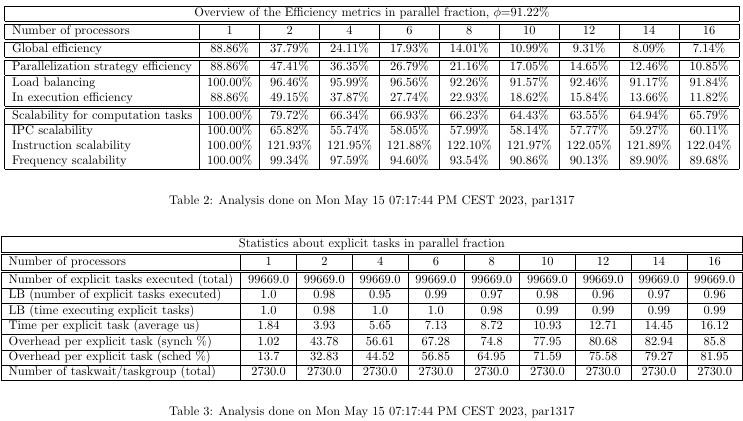
*Fig.14. Tree strategy plots of speed-up*

The plots generated for the Tree strategy indicate a promising level of parallelization performance when focusing especially on the multisort function, which represents the parallel region. However, it is worth nothing that the overall strong scalability of the entire application has not yet reached the desired level.

Following that, we conducted a modelfactors analysis for this strategy, as we did for leaf strategy, and the obtained tables are as follows:



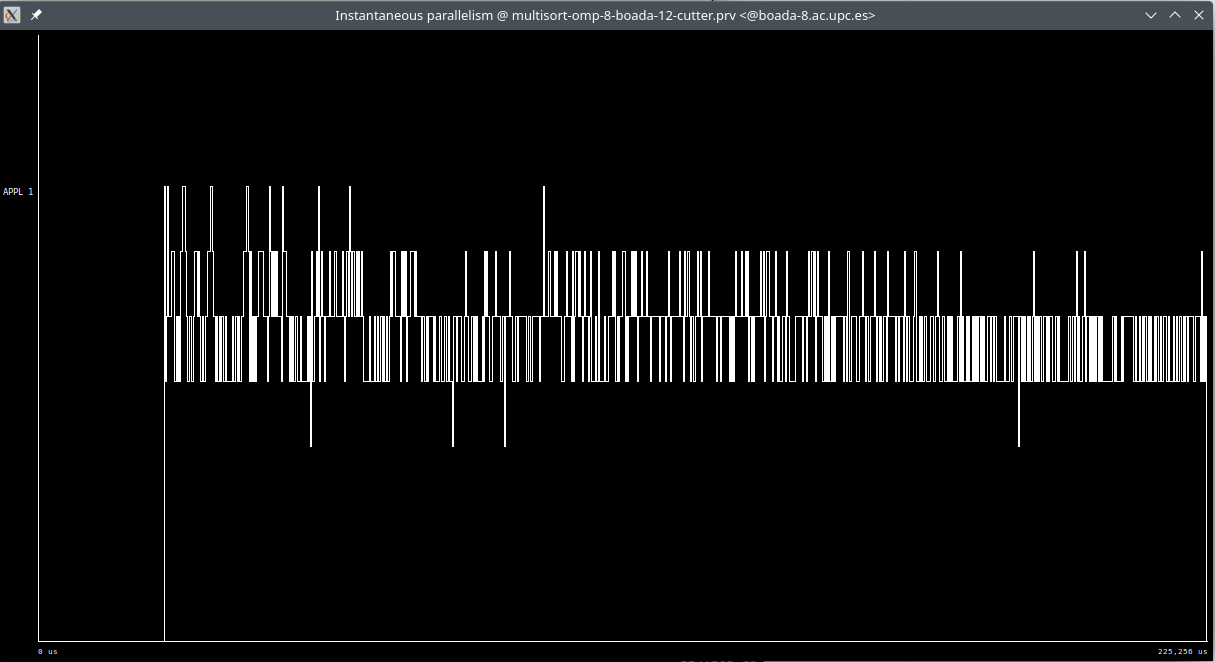




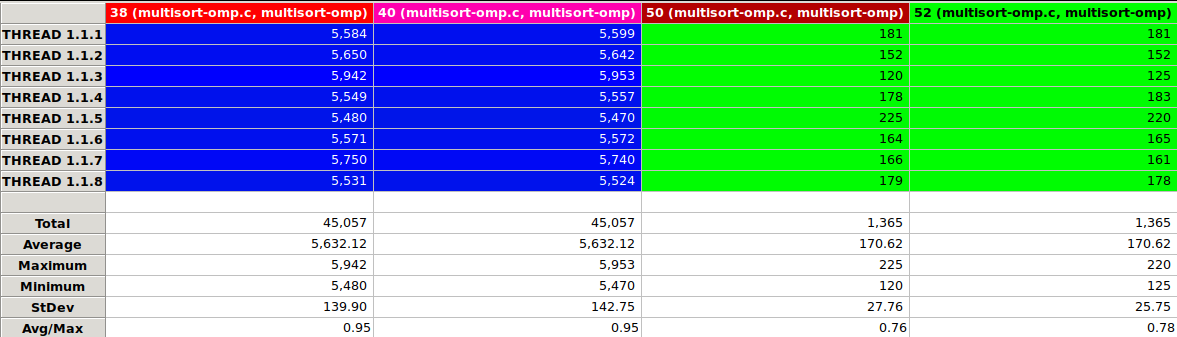
*Fig.15. Leaf strategy tables of the modelfactors analysis*

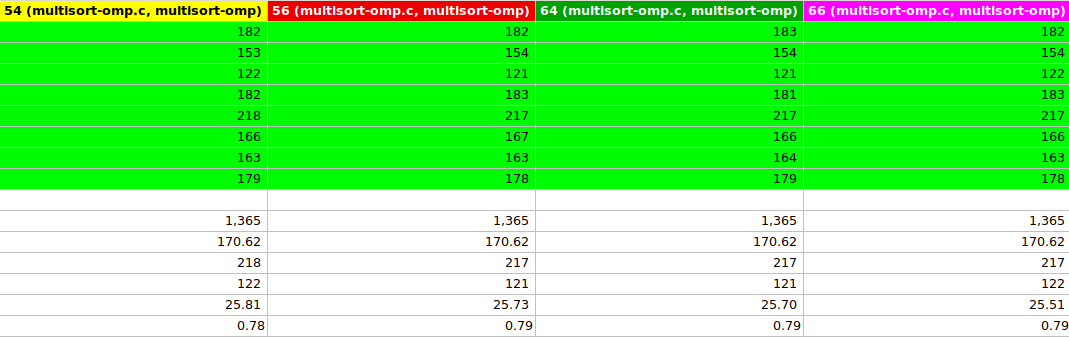
In this case, we can observe that the global efficiency and the parallelization strategy efficiency decreases a little bit less as we add more threads compared to the leaf strategy. Also, we can see that the overhead per task is lower than in the previous case. This helps to improve efficiency of this strategy.

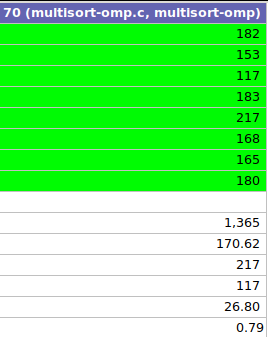
To end this part, we performed a Paraver analysis to delve deeper into the execution of the tree strategy. In the analysis we obtained the following timelines and histograms, which provide detailed perpectives into the execution process.



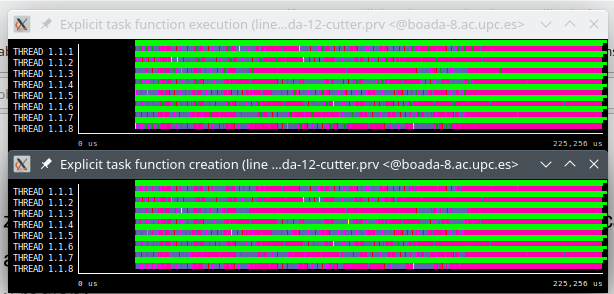
*Fig.16. Trace of instantaneous parallelisation of tree strategy*







*Fig.17. Histogram of task execution of leaf strategy*



*Fig.18. Trace of task creation and execution of leaf strategy*

If we observe the histograms above, we can conclude that all threads create tasks and execute them. We can see it perfectly watching Fig. 17 and 18

Version comparison

Indeed, the leaf strategy in parallelization creates tasks only at the base cases of recursive implementation. This approach generates tasks solely at the leaf level, where no further recursion is possible. In contrast, the tree strategy creates tasks before the calls to the merge and multisort functions, resulting in a higher quantity of explicitly created tasks.

Upon comparing the execution times, if the tree strategy consistently achieves better execution times than the leaf strategy, it indicates that the tree strategy is more efficient in terms of performance. The improved execution time suggests that the tree strategy successfully exploits parallelism and optimizes the utilization of available resources.

Now if we observe the scalability plots, it appears that both the leaf strategy and the tree strategy encounter strong scalability issues. However, the tree strategy demonstrates slightly better strong scalability compared to the leaf strategy. Additionally, the speed-up of the multisort region closely aligns with the line representing ideal strong scalability.

Based on the performance analysis using modelfactors and Paraver, it appears that both the leaf and the tree strategy encounter efficiency problems.

The leaf strategy executes fewer explicit tasks compared to the tree strategy, but it experiences larger overheads as more threads are added. This suggests that the additional threads created in the leaf strategy introduce significant overhead and may negatively impact its performance.

On the other hand, the tree strategy exhibits minor overheads, indicating that the parallelization and task distribution among threads are more efficient. However, the tree strategy executes a larger number of explicit tasks, which can potentially affect its overall performance due to increased task management and coordination overhead.

In both cases, it is noted that the tasks are equally distributed among threads, except for the thread responsible for creating the tasks.

## Task granularity control: the cut-off mechanism

In this section, we will introduce a cut-off mechanism to control the task granularity of the tree strategy. This mechanism allows us to set a threshold or cut-off point where the recursion stops and the remaining work is processed sequentially rather than creating additional tasks.

To implement the cut-off mechanism in the tree strategy, we introduced a variable “depth” (we call it ‘d’), to track the current level in the tree. This depth variable should be passed as an additional parameter to both the multisort and merge functions.

In each recursive call, you increment the depth variable by one, indicating that you are going deeper into the tree. By monitoring the depth level, you can determine the current tree level and decide whether to continue generating tasks or stop it if it exceeds the specified threshold passed as a parameter specified with the “‘-c” option.

In order to manage this, we incorporated the “#pragma omp task final(d >= CUTOFF)” directive into the multisort and merge functions, and included two additional else statements to handle the sequential execution of the cutoff level functions. Fig.19. illustrates the implementation of the aforementioned mechanism.

void merge(long n, T left[n], T right[n], T result[n\*2], long start, long length, int d) {

if (length < MIN\_MERGE\_SIZE\*2L) {

// Base case

basicmerge(n, left, right, result, start, length);

} else {

// Recursive decomposition

if (!omp\_in\_final()) {

#pragma omp task final (d >= CUTOFF)

merge(n, left, right, result, start, length/2, d+1);

#pragma omp task final (d >= CUTOFF)

merge(n, left, right, result, start + length/2, length/2, d+1);

#pragma omp taskwait

}

else {

merge(n, left, right, result, start, length/2, d+1);

merge(n, left, right, result, start + length/2, length/2, d+1);

}

}

}

void multisort(long n, T data[n], T tmp[n], int d) {

if (n >= MIN\_SORT\_SIZE\*4L) {

// Recursive decomposition

if (!omp\_in\_final()) {

#pragma omp task final (d >= CUTOFF)

multisort(n/4L, &data[0], &tmp[0], d+1);

#pragma omp task final (d >= CUTOFF)

multisort(n/4L, &data[n/4L], &tmp[n/4L], d+1);

#pragma omp task final (d >= CUTOFF)

multisort(n/4L, &data[n/2L], &tmp[n/2L], d+1);

#pragma omp task final (d >= CUTOFF)

multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L], d+1);

#pragma omp taskwait

#pragma omp task final (d >= CUTOFF)

merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, d+1);

#pragma omp task final (d >= CUTOFF)

merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L, d+1);

#pragma omp taskwait

#pragma omp task final (d >= CUTOFF)

merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, d+1);

#pragma omp taskwait

}

else {

multisort(n/4L, &data[0], &tmp[0], d+1);

multisort(n/4L, &data[n/4L], &tmp[n/4L], d+1);

multisort(n/4L, &data[n/2L], &tmp[n/2L], d+1);

multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L], d+1);

merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, d+1);

merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L, d+1);

merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, d+1);

}

} else {

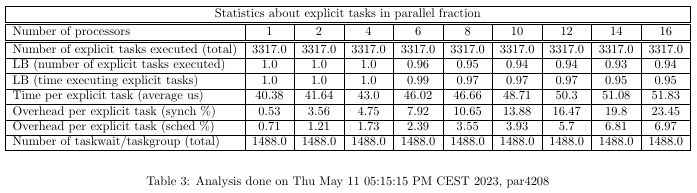
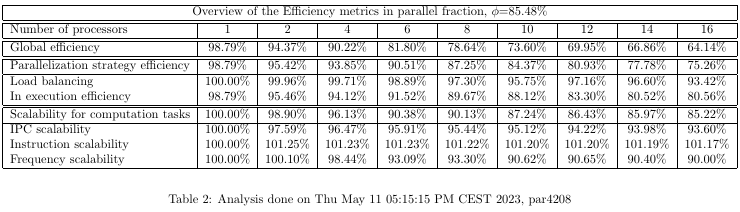
// Base case

basicsort(n, data);

}

}

*Fig.19. Tree strategy code modification with cut-off*



# 

# 

# 

# 

# 

# 

# 

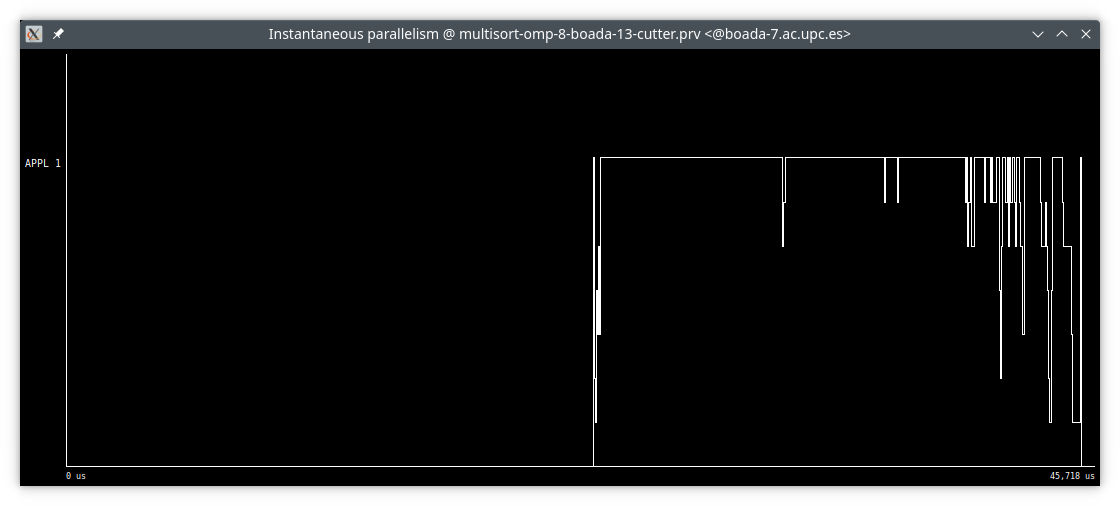
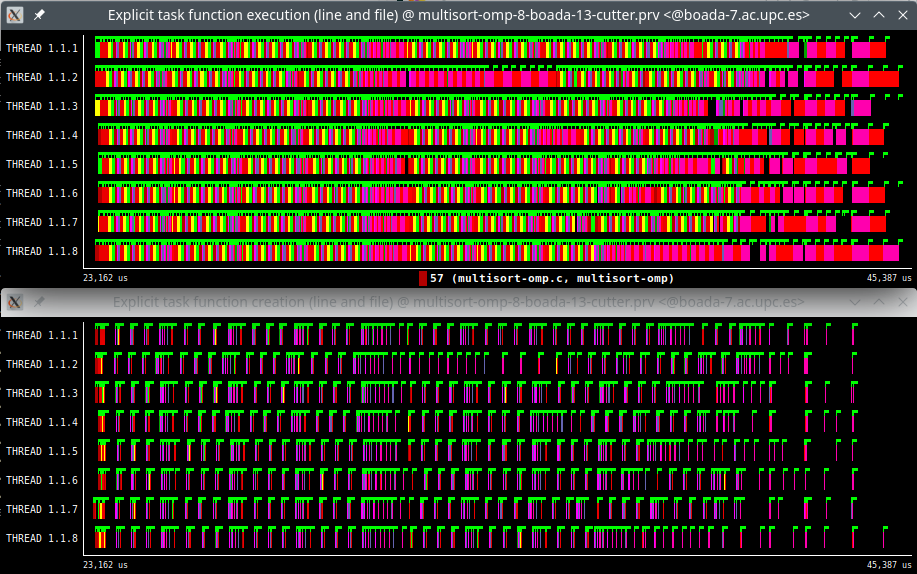
# 



*Fig.20. Tree strategy with cut-off tables of the modelfactors analysis*

Upon observing Fig. 20., it becomes apparent that the elapsed time and overheads have experienced a slight reduction. Additionally, an increase in the number of explicit tasks can be observed as a result of the cut-off level. This is primarily due to the tree strategy, where tasks are generated for each recursive call. Consequently, a higher cut-off value leads to a deeper level of the tree, resulting in an increased number of recursive calls and subsequently, more tasks being created.

Then, we did the analysis with paraver to observe the behavior of the tree strategy with cut-off using 8 threads:



# 

# 

# 

# 

# 

*Fig.21. Trace of task creation and execution of Tree strategy with cut-off*

# 

*Fig.22. Trace of instantaneous parallelisation of tree strategy with cut-off*

We can see that all threads create and execute tasks, as tree strategy did.

# 

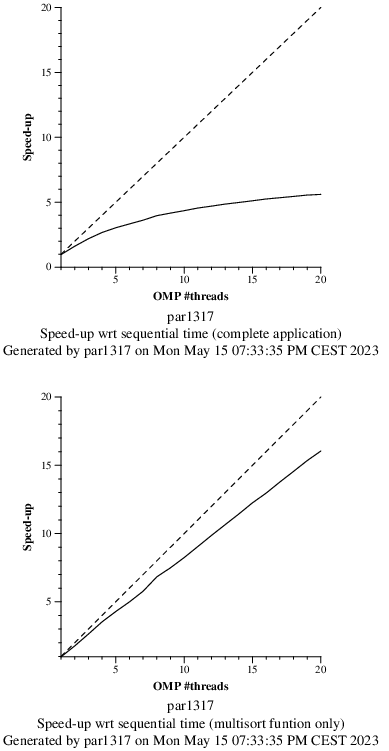
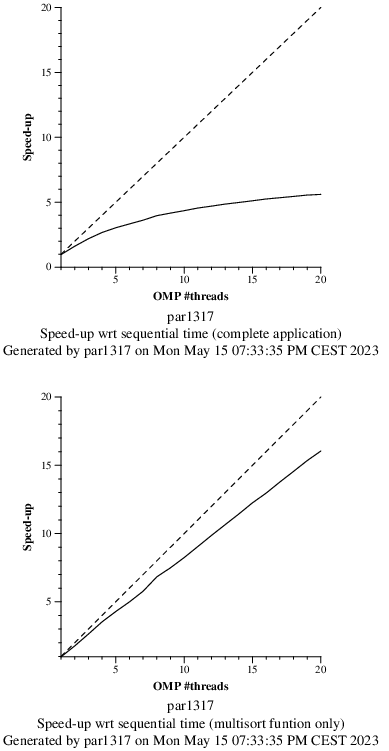
# 

# 

# 

# 

*Fig.23. Execution time with cut-off*



*Fig.24. Tree strategy with cut-off plots of speed-up*

# 3. Shared-memory parallelisation with OpenMP task using dependencies

We have now modified our tree version strategy with the cut-off mechanism by replacing the redundant taskwait synchronizations with point-to.point task dependencies. The following code fragment illustrates the changes made:

void merge(long n, T left[n], T right[n], T result[n\*2], long start, long length) {

if (length < MIN\_MERGE\_SIZE\*2L) {

// Base case

basicmerge(n, left, right, result, start, length);

} else {

// Recursive decomposition

#pragma omp task

merge(n, left, right, result, start, length/2);

#pragma omp task

merge(n, left, right, result, start + length/2, length/2);

#pragma omp taskwait

}

}

void multisort(long n, T data[n], T tmp[n], int depth) {

if (n >= MIN\_SORT\_SIZE\*4L) {

// Recursive decomposition

#pragma omp task depend(out: data[0])

multisort(n/4L, &data[0], &tmp[0], depth +1);

#pragma omp task depend(out: data[n/4L])

multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);

#pragma omp task depend(out: data[n/2L])

multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);

#pragma omp task depend(out: data[3L\*n/4L])

multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L], depth+1);

//#pragma omp taskwait

#pragma omp task depend(in: data[0],data[n/4L]) depend(out: tmp[0])

merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);

#pragma omp task depend(in: data[3L\*n/4L],data[n/2L]) depend(out: tmp[n/2L])

merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L);

//#pragma omp taskwait

#pragma omp task depend(in: tmp[0], tmp[n/2L])

merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);

#pragma omp taskwait

} else {

// Base case

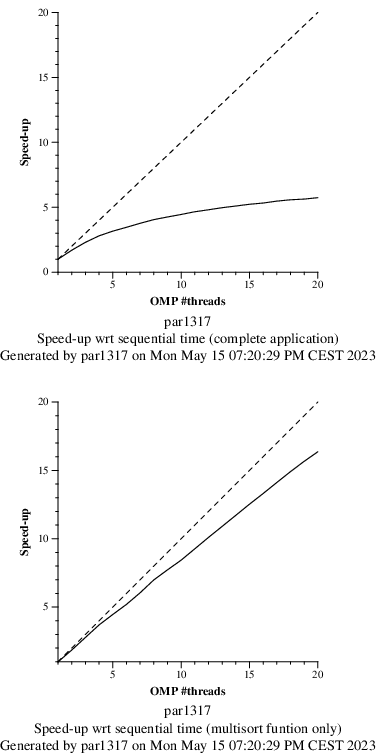
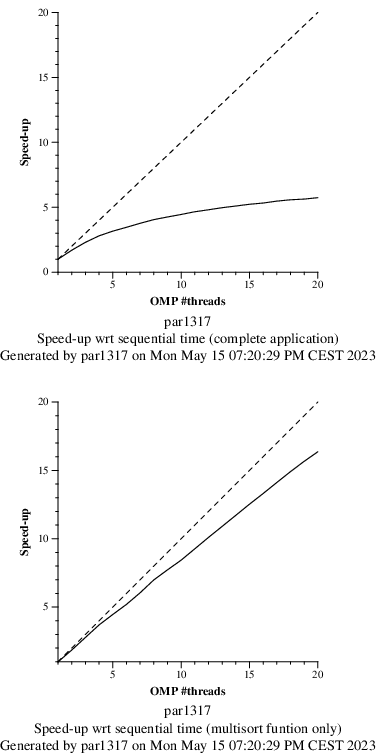
basicsort(n, data);

}

}

*Fig.25. Dependencies code modification of tree strategy*

By utilizing point-to-point task dependencies, we have enhanced the efficiency of the tree version strategy with the cut-off mechanism, reducing the need for excessive taskwait synchronizations.

*Fig.26. Dependencies tree strategy with cut-off plots of speed-up*

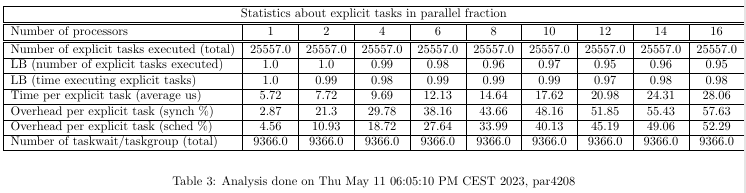
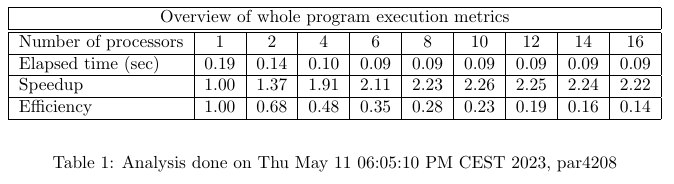
Upon examining the scalability plots in Fig. 26 and comparing them with the previous version, it is apparent that the scalability of the new version, implemented with tree strategy and the cut-off mechanism, is very similar. However, the new version appears to approach the ideal scalability line more closely.

This observation suggests that both the tree strategy and the cut-off strategy exhibit similar performance characteristics in terms of scalability when compared to the new version. Therefore, in terms of performance, it seems that both strategies are high comparable to the improved version of the application.

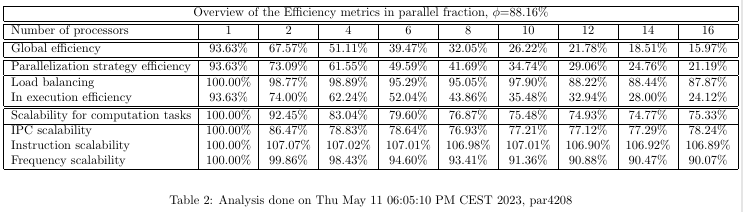
When considering the programmability aspect of the new version, implementing point-to-point dependencies can indeed contribute to a better understanding of the data flow within the execution and the specific points where tasks need to wait for others. This explicit control over task dependencies can provide clearer insights into the program’s behavior.

On the other hand, using task wait or task group versions can offer a higher level of abstraction and simplicity to the programmer. These approaches allow the programmer to abstract away the explicit management of task dependencies, enabling them to focus more on the overall logic of the program without having to worry about explicit wait points.

Ultimately, the choice between using point-to-point dependencies or taskwait/taskgroup versions depends on the specific needs of the application and the preferences of the programmer. Both approaches have their advantages and trade-offs, and it is important to strike a balance between programmability and explicit control to achieve the desired performance in the code.









*Fig.27. Dependencies tree strategy tables of the modelfactors analysis*

Upon comparing the outcomes derived from modelfactors examination with the other editions, we have noticed that the results are quite similar to the results obtained with the cut-off version, with insignificant discrepancies in their values. As anticipated, the number of taskwait/taskgroup in the new version is quite lower than in the previous version.

When contrasting it with the tree version, the results demonstrate a significant improvement as they closely resemble those of the cut-off version.

Then, we executed the Paraver analysis obtaining this different trace below:

# 

# 

# 

# 

*Fig.28. Trace of instantaneous parallelisation of tree strategy with dependencies*

We can observe that there are more tasks executed at the same time than in the tree strategy. This is easy to see comparing Fig 28 and Fig 16

# 

# Conclusion

During the initial phase of this practical task, besides acquiring knowledge on handling recursive functions, we encountered two distinct approaches to parallelize the provided code: the leaf strategy and the tree strategy.

After thoroughly examining these options and confirming their effectiveness, we proceeded to incorporate them into the code. Then, we compiled and executed the code utilizing the appropriate script, allowing us to observe the visual representation using Paraver.

During the second phase, our focus shifted towards exploring various strategies for parallelizing the script. We dedicated our efforts to testing and evaluating the different approaches to parallelize the code effectively.

In conclusion, throughout these laboratory sessions, we have acquired the knowledge and skills necessary to parallelize codes that involve recursive functions. Furthermore, we have successfully assessed the efficiency of the different implementations based on specific cases and scenarios.