

PARAL·LELISME

Divide and Conquer parallelism with OpenMP: Sorting

Sergio Utrero Preciado (**par1122**)

Jordi Bru Carci (**par1104**)

11/05/2022

2021-2022.Q2



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona



TABLE OF CONTENTS

Introduction	2
Task decomposition analysis with Tareador	4
Parallelisation and performance analysis with tasks	9
Leaf strategy in OpenMP	9
Tree strategy in OpenMP	15
Controlling task granularities: cut-off mechanism	21
Optional 1	26
Parallelisation and performance analysis with task dependencies	27
Optional 2	31
Final Conclusions	36

Introduction

In this laboratory we will analyze the parallelization of codes that implement "Divide & Conquer" algorithms using various strategies such as leaf and tree. In order to study what we are proposing, we have been provided with a code called `multi_sort.c` that uses the sorting algorithm known as mergesort.

Mergesort is a sorting algorithm that consists of taking a vector and dividing it into different subvectors until they are of desirable size. Once this process is finished, we now start merging those by pairs in an orderly manner. And so on until we end up with the original vector sorted in ascending order.

```
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
        basicsort(n, data);
    }
}
```

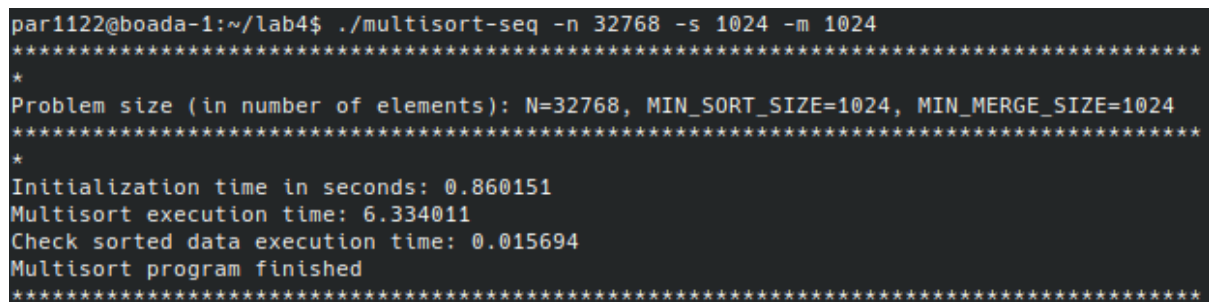
Figure 1. Screenshot of multisort

As we can see in the figure 1, the function `multisort` calls itself recursively with the vector divided in 4 segments. Then it merges the first two segments to the first half of a temporary buffer and then it does the same for the last two segments. Finally, it does a final merge of the two halves of the temporary buffer to the original vector.

It's important to highlight the fact that the aim of this code is to find a better parallelization of a sorting algorithm. We can say that, because the code divides the merging into halves until we reach a desired number of elements, then it uses a `basicmerge(n, data)` instead of the merge itself. All these tools are implemented to exploit the parallelization of the code.

To see the performance of the code without any modifications and prove that it works correctly, we will execute it with a number of elements equal to 32768, with a sort size and merge size of 1024.

```
>./multisort-seq -n 32768 -s 1024 -m 1024
```

A screenshot of a terminal window showing the execution of the program `multisort-seq`. The prompt is `par1122@boada-1:~/lab4$`. The command entered is `./multisort-seq -n 32768 -s 1024 -m 1024`. The output consists of several lines: a line of asterisks, a line with a single asterisk, a line stating the problem size and parameters, another line of asterisks, a line with a single asterisk, and then three lines of execution times: `Initialization time in seconds: 0.860151`, `Multisort execution time: 6.334011`, and `Check sorted data execution time: 0.015694`. The final line is `Multisort program finished`, followed by another line of asterisks.

```
par1122@boada-1:~/lab4$ ./multisort-seq -n 32768 -s 1024 -m 1024
*****
*
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
*****
*
Initialization time in seconds: 0.860151
Multisort execution time: 6.334011
Check sorted data execution time: 0.015694
Multisort program finished
*****
```

Figure 2. Screenshot of the execution of `multisort-seq.c`

The output of this execution (figure 2) will help us to compare the results of future strategy tests (which we will see below), with the current results and thus support future conclusions when measuring performance.

Task decomposition analysis with Tareador

As I have introduced, in this section we will analyze how code dependencies can affect parallelization. To do this, we will use the provided code called multisort-tareador.c, which is a variant of the original code to use the already known tool tareador.

To start analyzing what we want, we will employ two different strategies. The first one we will deal with the leaf strategy, that consists in declaring a task for each call of the basicsort and basic merge functions (figure 3). On the other hand, there is the tree strategy that consists of defining tasks for each call to the multisort and merge functions (figure 5).

```
//code ...
tareador_start_task("leaf merge strat");
basicmerge(n, left, right, result, start, length);
tareador_end_task("leaf merge strat");
//code ...

//code ...
tareador_start_task("leaf sort strat");
basicsort(n, data);
tareador_end_task("leaf sort strat");
//code ...
```

Figure 3. Screenshot of the relevant code of multisort-tareador.c in the mentioned leaf strategy

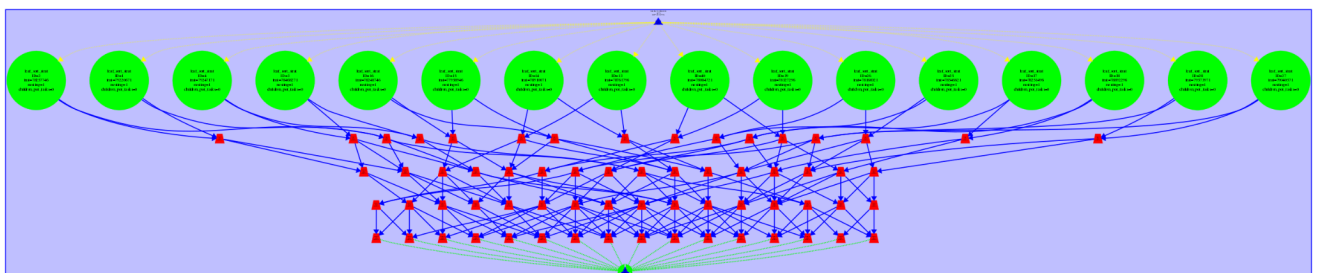


Figure 4. Dependency graph of multisort-tareador.c with the leaf parallelization strategy

```

//code ...
tareador_start_task("tree_4 merge strat");
merge(n, left, right, result, start, length/2);
tareador_end_task("tree_4 merge strat");
tareador_start_task("tree_5 merge strat");
merge(n, left, right, result, start + length/2, length/2);
tareador_end_task("tree_5 merge strat");
//code ...

//code ...
tareador_start_task("tree_1 multisort strat");
multisort(n/4L, &data[0], &tmp[0]);
tareador_end_task("tree_1 multisort strat");
tareador_start_task("tree_2 multisort strat");
multisort(n/4L, &data[n/4L], &tmp[n/4L]);
tareador_end_task("tree_2 multisort strat");
tareador_start_task("tree_3 multisort strat");
multisort(n/4L, &data[n/2L], &tmp[n/2L]);
tareador_end_task("tree_3 multisort strat");
tareador_start_task("tree_4 multisort strat");
multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
tareador_end_task("tree_4 multisort strat");
tareador_start_task("tree_1 merge strat");
merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
tareador_end_task("tree_1 merge strat");
tareador_start_task("tree_2 merge strat");
merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
tareador_end_task("tree_2 merge strat");

tareador_start_task("tree_3 merge strat");
merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
tareador_end_task("tree_3 merge strat");
//code ...

```

Figure 5. Screenshot of the relevant code of multisort-tareador.c in the mentioned tree strategy

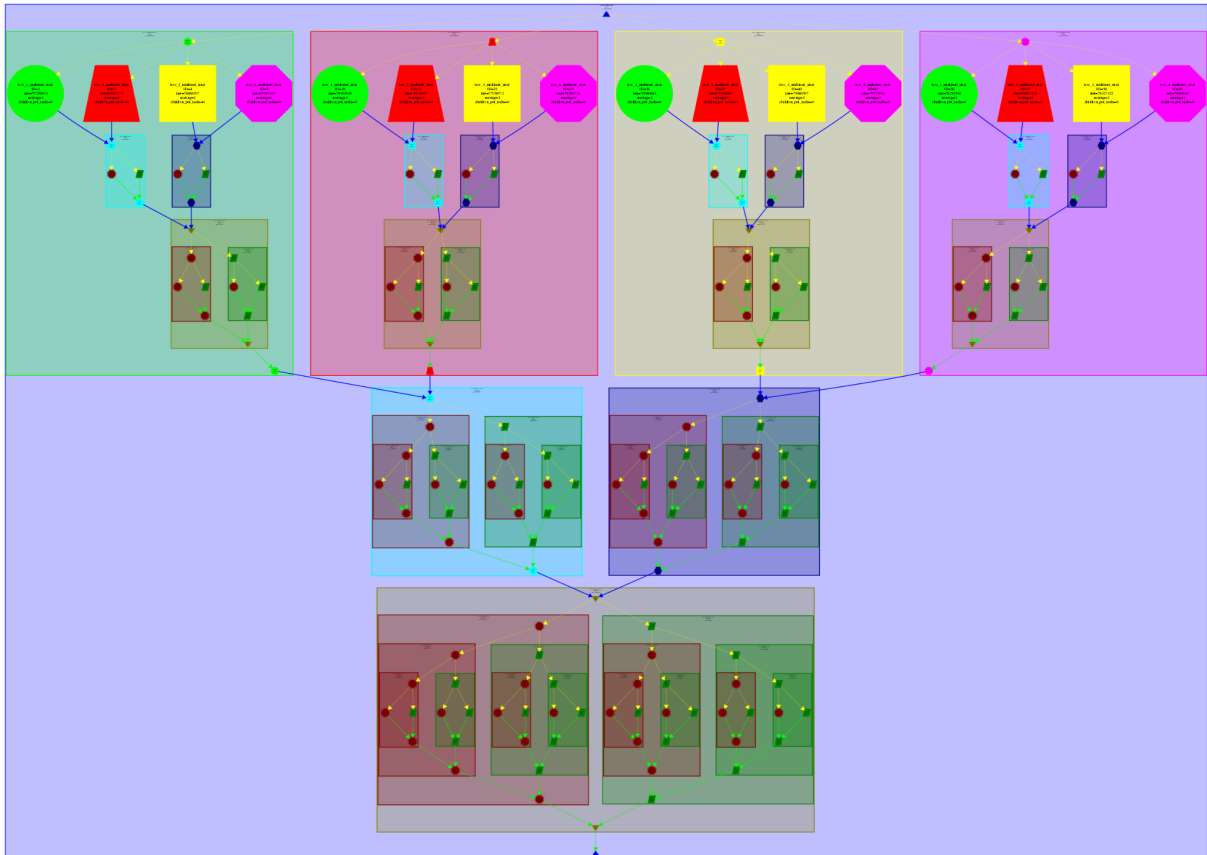


Figure 6. Dependency graph of multisort-tareador.c with the tree parallelization strategy

At first glance, the dependency graph from Figure 4, we can see how in the leaf strategy, 16 sort tasks and 64 merge tasks are generated. This is due to the fact that the number of elements are reduced and distributed by tasks and then merged between all of them, until we get to the final stage.

But if we then look at the dependency graph of the tree strategy (figure 6), we can see how the complexity of the dependencies has increased significantly. Now we see a much more tree-like structure because as we can see in the green, red, yellow and pink colored boxes, the multisort function is executed in each of them, thus creating 4 subdivisions of the vector. Once arrived at this point, every task executes recursively the procedure of the code explained above, thus achieving the desired tree figure.

It is important to highlight that as we go down the tree structure, the same diversity of colors of the boxes is maintained. This helps us to better understand the recursion performed and how the tasks are distributed in parallel.

	CPU 1.1	CPU 1.2	CPU 1.3	CPU 1.4	CPU 1.5	CPU 1.6	CPU 1.7	CPU 1.8	CPU 1.9	CPU 1.10	CPU 1.11	CPU 1.12	CPU 1.13	CPU 1.14	CPU 1.15	CPU 1.16
MAIN_TAREADOR	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
leaf_sort_strat	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
leaf_merge_strat	11	10	8	7	6	4	5	3	1	2	2	1	1	1	1	1
Total	13	11	9	8	7	5	6	4	2	3	3	2	2	2	2	2
Average	4.33	5.50	4.50	4	3.50	2.50	3	2	1	1.50	1.50	1	1	1	1	1
Maximum	11	10	8	7	6	4	5	3	1	2	2	1	1	1	1	1
Minimum	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
StDev	4.71	4.50	3.50	3	2.50	1.50	2	1	0	0.50	0.50	0	0	0	0	0
Avg/Max	0.39	0.55	0.56	0.57	0.58	0.62	0.60	0.67	1	0.75	0.75	1	1	1	1	1

Figure 7. Number of tasks in leaf strategy

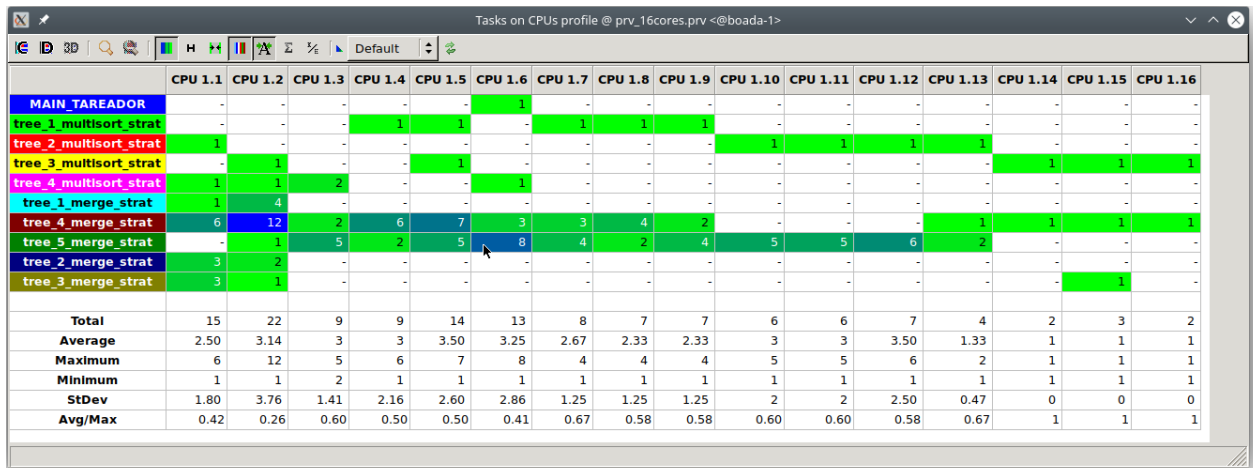


Figure 8. Number of tasks in tree strategy

To find out how many tasks we have and of what type we can consult figures 7 and 8, which correspond to the tasks carried out with the leaf and tree strategy respectively with sixteen cores.

On the leaf strategy we see an even distribution of work on the sort tasks and an uneven distribution of the merge ones. We have sixteen sort tasks and sixty-four merge ones. This is due to the fact that we have four stages of sixteen tasks.

On the tree strategy we see how the sort tasks are balanced, which is not the case for the merge tasks. The `tree_4_merge_strat` and `tree_5_merge_strat` tasks are evenly distributed.

In order to study how tasks are created and executed, we can take advantage of the Paraver tool to check the timelines of the two strategies: leaf and tree. These timelines correspond to figure 8 and 9 respectively.

In Figure 9 we can see how throughout practically the execution time the sorting tasks are executed (leaf sort strat in the code of Figure 3) while it is at the end where the merge tasks are executed (leaf merge strat in the code of Figure 3). On the other hand, in Figure 10, similarly to the above, the sorting tasks are executed first and then the merge tasks are executed almost at the end of the execution time.

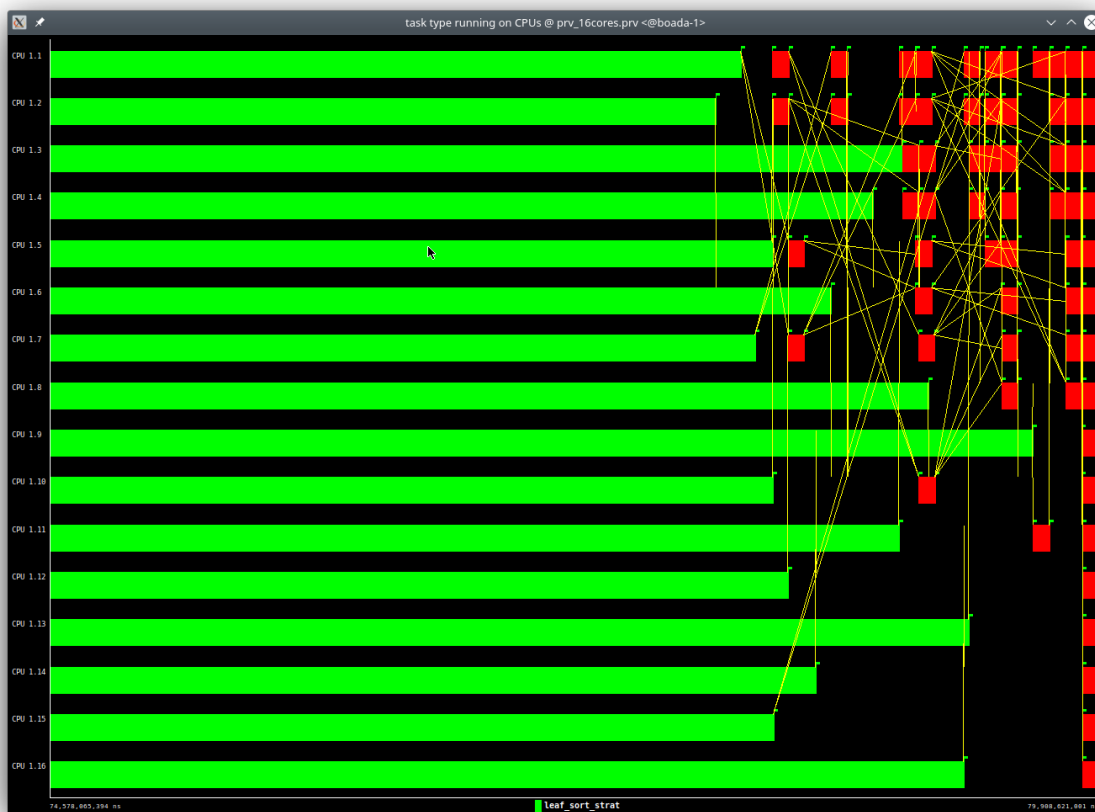


Figure 9. Chronogram of leaf strategy with 16 threads (Zoom-in at the end)

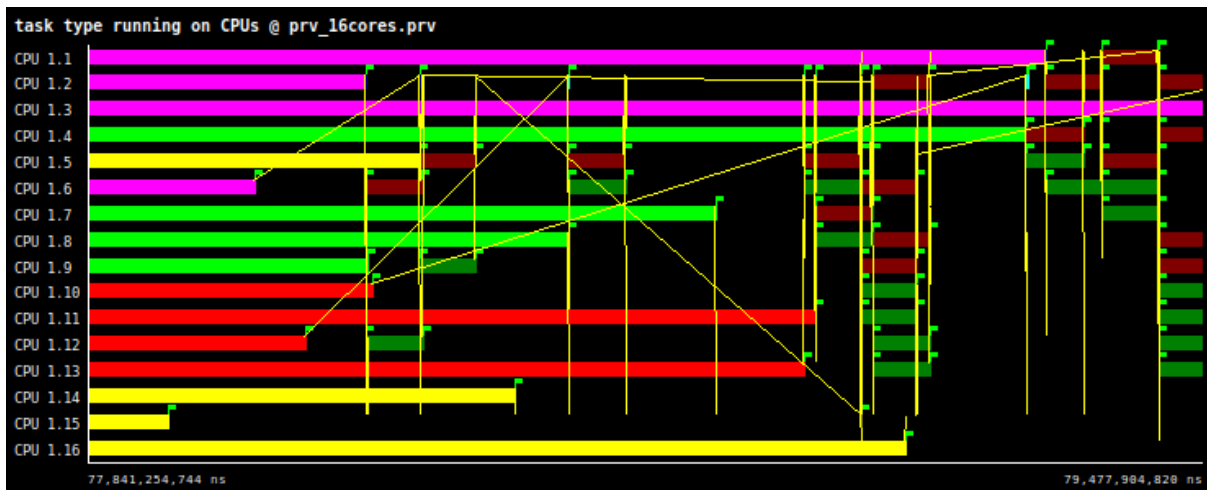


Figure 10. Chronogram of tree strategy with 16 threads (Zoom-in at the end)

Parallelisation and performance analysis with tasks

Leaf strategy in OpenMP:

After familiarizing ourselves with the two types of strategies in the previous section, we will discuss code parallelization in more depth while we focus on the mergesort function. To do this, we will add OpenMP clauses in the multisort.c code.

First of all, we will work with the leaf strategy in parallel. As we can see in Figure 11 we have declared parallel and single regions so we can ensure that only one thread has to be in charge of creating tasks. As the code is executed recursively, each time it reaches the base case, it will create a task for another thread. We have also added taskwait clauses to prevent any merge from being computed before any sort or other merge until its finished. That means we can avoid data races.

```
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);
    }
}

int main(int argc, char **argv) {
    #pragma omp parallel
    #pragma omp single
    multisort(N, data, tmp);
}
```

Figure 11. Screenshot of the key fragment of multisort-omp.c with the implemented leaf parallelization

To check that the code works and compiles correctly, we will execute it with different numbers of threads to compare its performance.

```
>sbatch ./submit-omp.sh multisort-omp 1
```

```
>sbatch ./submit-omp.sh multisort-omp 2
```

```
>sbatch ./submit-omp.sh multisort-omp 4
```

Rellenaremos la siguiente tabla con el output de las tres ejecuciones.

#Threads	Initialization	Multisort	Check sorted data
1	0.855455	6.312449	0.015243
2	0.855526	3.800366	0.038825
4	0.856959	2.015849	0.016894

Figure 12. Result table of the execution time with different number of threads of multisort-omp.c

As shown in figure 12, we can see that increasing the number of threads to 4 we can actually have a better performance on the multisort/sort section. It is clear that the check sorted data and initialization sections do not have any significant changes because those sections are not parallelized.

To have a deeper understanding of how parallelization scales, we will execute our version of the code with the script provided called submit-strong-omp.sh with a range of 1 to 12 threads. This way we can clearly see how it evolves.

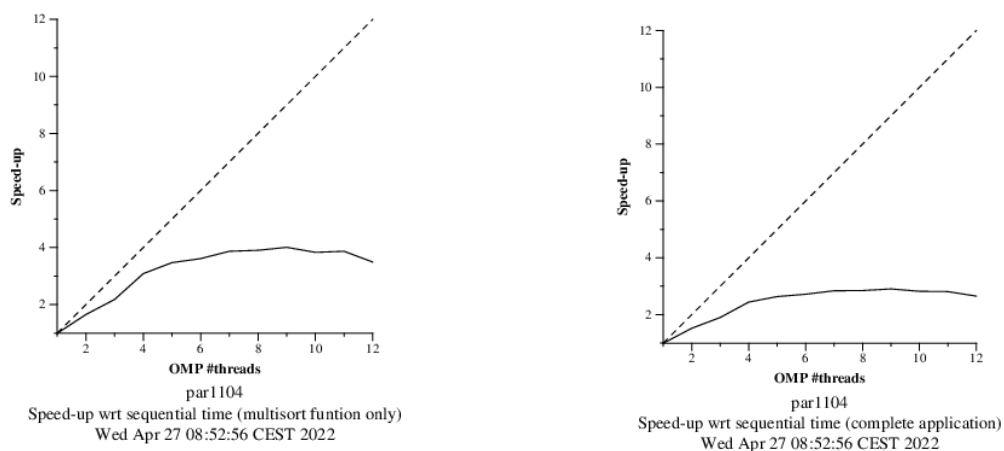


Figure 13. Speedup scalability plots of the leaf parallelization in multisort function only (left) and the code in general (right)

The output of the execution are those two plots (figure 13). The plot on the left side is the speedup scalability only of the parallel region (multisort) and the one on the right side is the speedup scalability of the code in general.

To talk about the scalability of the two graphs we will separate it into three parts: from 1 to 4 threads, from 5 to 11 threads and from 12 threads onwards. We can see how from 1-4 threads the speedup scalability grows considerably, especially in the parallelized region of the multisort, which grows in a fairly linear and fast way. On the other hand, if we focus on the range of 5-11 threads, we can see how in the two plots it barely grows, it fluctuates but remains the same. Finally, from 12 threads on, the speedup of the multisort section drops significantly due to synchronization overheads.

Obviously, these scalability changes are much more noticeable in the plot that emphasizes the multisort and not so much the one of the code in general, since in the second one, we do not use any parallelization. The little that the second plot is affected is because of the multisort.

Since we are not convinced of the results obtained, we will now submit the execution of the binary using `submit-strong-extrac.sh` script, which will trace the execution of the parallel execution with 1, 2, 4 and 8 processors with a much smaller input (`-n 1024 -s 256 -m 256`) and execute `modelfactors.py`. This way, we are going to be able to have a more in-depth analysis of the execution with a different amount of threads.

```
>sbatch ./submit-strong-extrae.sh multisort-omp
```

Overview of whole program execution metrics:

Number of processors	1	2	4	8
Elapsed time (sec)	0.54	0.39	0.28	0.31
Speedup	1.00	1.39	1.90	1.77
Efficiency	1.00	0.70	0.47	0.22

Overview of the Efficiency metrics in parallel fraction:

Number of processors	1	2	4	8
Parallel fraction	93.89%			
Global efficiency	82.89%	59.08%	41.69%	19.28%
-- Parallelization strategy efficiency	82.89%	55.23%	38.68%	19.37%
-- Load balancing	100.00%	98.62%	76.83%	44.43%
-- In execution efficiency	82.89%	56.01%	50.34%	43.60%
-- Scalability for computation tasks	100.00%	106.97%	107.79%	99.52%
-- IPC scalability	100.00%	87.13%	85.83%	84.81%
-- Instruction scalability	100.00%	110.98%	112.30%	112.20%
-- Frequency scalability	100.00%	110.63%	111.82%	104.59%

Statistics about explicit tasks in parallel fraction

Number of processors	1	2	4	8
Number of explicit tasks executed (total)	53248.0	53248.0	53248.0	53248.0
LB (number of explicit tasks executed)	1.0	0.64	0.7	0.94
LB (time executing explicit tasks)	1.0	0.72	0.76	0.95
Time per explicit task (average)	4.67	5.22	5.33	5.7
Overhead per explicit task (synch %)	1.9	71.44	179.45	541.4
Overhead per explicit task (sched %)	32.94	43.14	38.41	37.86
Number of taskwait/taskgroup (total)	2730.0	2730.0	2730.0	2730.0

Figure 14. Screenshot of modelfactors.out of leaf parallelization

Figure 14 is the result of executing the script plus modelfactors.py, which contains three tables with valuable information to understand how the leaf parallelization performs. The first table details some parameters from the executed program with a different amount of threads, from 1 to 8. First table clearly shows how the efficiency falls when the amount of threads rises, so we can, at first sight, see how poor the performance is.

If we look into the second table, it will give us information about the efficiency of the code in general. A point to be highlighted is that the parameter called “parallelization strategy efficiency” also decreases brutally when the amount of threads increases. One of the reasons behind this is load balancing because of how tasks are distributed. Figure 15 and 19 shows how tasks are distributed. The balance between the computation threads is fairly balanced, exceptuating the thread number one, because it is the one in charge of creating unique tasks (Figure 16 & 20). This parallelization is not as load balanced as we feared, causing its efficiency level to drop.

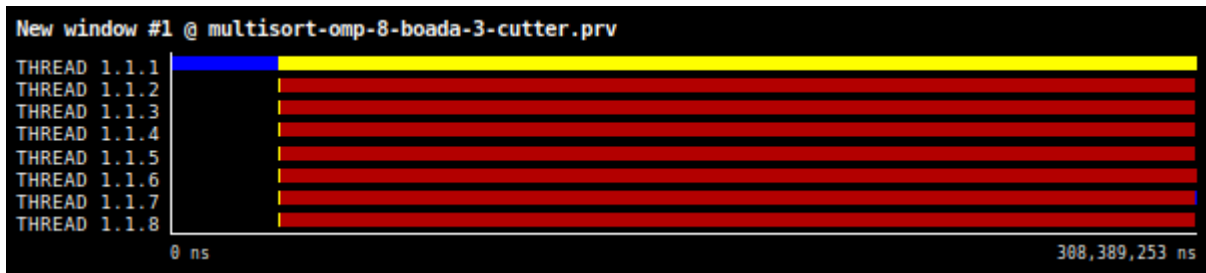


Figure 15. Chronogram of leaf parallelization with eight threads

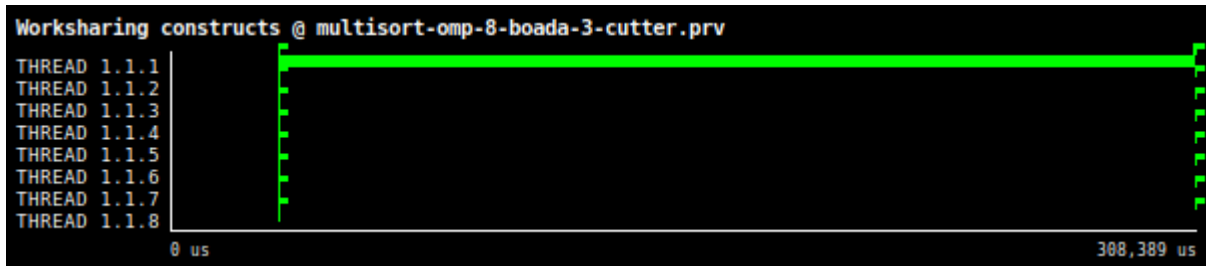


Figure 16. Worksharing constructs of leaf parallelization with eight threads

Figures 17 and 18 illustrate how many tasks are created and executed by each thread. Both chronograms confirm that the thread number 1 is in charge of task creation.

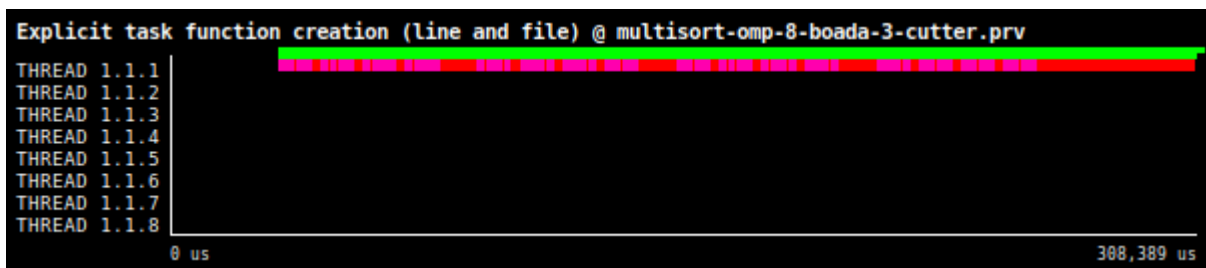


Figure 17. Task creation chronogram of leaf parallelization with eight threads



Figure 18. Task execution chronogram of leaf parallelization with eight threads

	35 (multisort-omp.c, multisort-omp)	60 (multisort-omp.c, multisort-omp)
THREAD 1.1.1	1	1
THREAD 1.1.2	6,492	546
THREAD 1.1.3	7,228	594
THREAD 1.1.4	6,916	607
THREAD 1.1.5	7,409	619
THREAD 1.1.6	6,771	558
THREAD 1.1.7	7,547	611
THREAD 1.1.8	6,788	560
Total	49,152	4,096
Average	6,144	512
Maximum	7,547	619
Minimum	1	1
StDev	2,345.34	194.84
Avg/Max	0.81	0.83

Figure 19. Histogram task execution of leaf parallelization with eight threads

	35 (multisort-omp.c, multisort-omp)	60 (multisort-omp.c, multisort-omp)
THREAD 1.1.1	49,152	4,096
THREAD 1.1.2	-	-
THREAD 1.1.3	-	-
THREAD 1.1.4	-	-
THREAD 1.1.5	-	-
THREAD 1.1.6	-	-
THREAD 1.1.7	-	-
THREAD 1.1.8	-	-
Total	49,152	4,096
Average	49,152	4,096
Maximum	49,152	4,096
Minimum	49,152	4,096
StDev	0	0
Avg/Max	1	1

Figure 20. Histogram task instantiation of leaf parallelization with eight threads

The third table from figure 14 works the same way as the other two, it details the explicit tasks in the parallel section. If we want to talk about what causes the drop in efficiency we have to focus on these two parameters: overhead per explicit task (synch%) and (sched%). In fact, we can clearly see how the overhead per explicit task (synch%) increases significantly as we increase the number of threads. Our goal is to avoid this growth of synchronization overheads, therefore, in agreement with figure 21, we can say that the generated synchronization overheads are a real problem for the efficiency of our leaf strategy. On the other hand, the overhead per explicit task (sched%) does not increase like the other one so it does not affect the code the same way as the synch% does.

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	49.71 %	12.71 %	37.59 %
THREAD 1.1.2	15.33 %	84.67 %	0.00 %
THREAD 1.1.3	15.82 %	84.18 %	0.00 %
THREAD 1.1.4	15.68 %	84.32 %	0.00 %
THREAD 1.1.5	16.63 %	83.37 %	0.00 %
THREAD 1.1.6	15.44 %	84.55 %	0.00 %
THREAD 1.1.7	16.74 %	83.25 %	0.00 %
THREAD 1.1.8	15.78 %	84.22 %	-
Total	161.13 %	601.27 %	37.60 %
Average	20.14 %	75.16 %	5.37 %
Maximum	49.71 %	84.67 %	37.59 %

Figure 21. Thread state histogram of the leaf parallelization

Tree strategy in OpenMP:

In this section, we will start to deeply analyze tree parallelization to see if it can really exploit parallelism better. Strategy which we have already discussed previously.

As we can see in figure 22, we implemented taskgroups clauses to make sure every task created inside the group is finished before moving on, so we can avoid data races.


```

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);
    }
}

int main(int argc, char **argv) {
    //code ...
    #pragma omp parallel
    #pragma omp single
    multisort(N, data, tmp);
    //code ...
}

```

Figure 22. Screenshot of the key fragment of multisort-omp.c with the implemented tree parallelization

To ensure that our code is correct we are going to execute it with the script provided so we can compare our results with different amounts of threads and also, the execution time comparison between the leaf and tree parallelization.

#Threads	Initialization	Multisort	Check sorted data
1	0.856213	6.373156	0.015316
2	0.856095	3.682495	0.017755
4	0.854830	1.882833	0.017913
4 (Leaf)	0.856959	2.015849	0.016894

Figure 23. Result table of the execution time with different number of threads of multisort-omp.c

After executing the code we have seen that it runs correctly, so we are about to compare the results obtained. In figure 23 we can see that the execution time is reduced but there is not much improvement if we compare it with the execution time table of the leaf parallelization. This way, we will run the code with the submit-strong-omp.sh script to generate the graphs we got in the previous section, so that we can actually compare the results obtained.

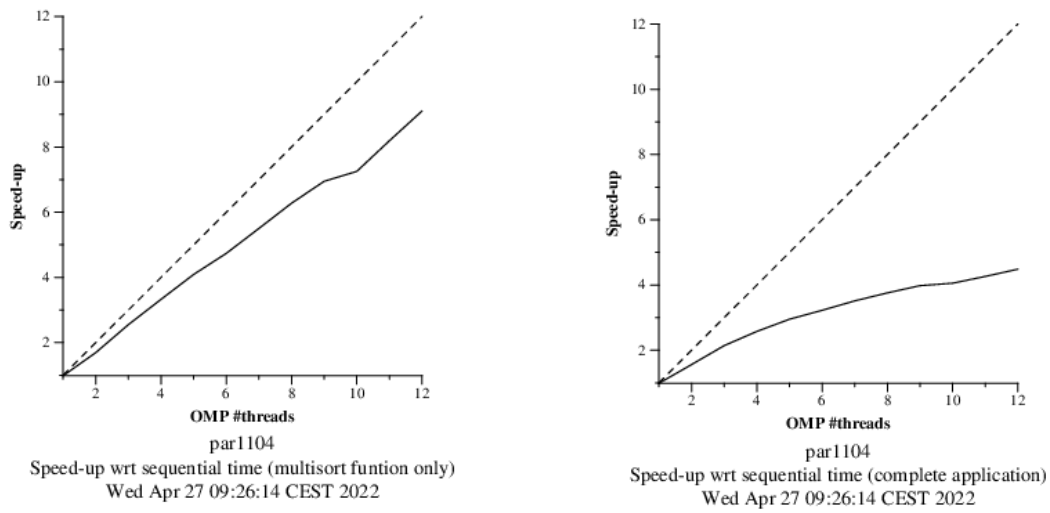


Figure 24. Speedup scalability plots of the tree parallelization in multisort function only (left) and the code in general (right)

At first glance, in figure 24, we can now tell that the speedup improves in the tree parallelization. We can actually say that the scalability is almost linear with the new clauses that we have implemented. It is important to highlight the fact that the vast improvement on the scalability in the multisort region, did affect the complete application which the other parallelization strategy does not. We can see how the right side plot starts to rise a bit when the amount of threads increases.

As we did in the previous section, we executed our tree parallelization with modelfactors, so we can have a more in-depth view of our strategy.

Overview of whole program execution metrics:				
Number of processors	1	2	4	8
Elapsed time (sec)	0.83	0.58	0.33	0.22
Speedup	1.00	1.43	2.55	3.77
Efficiency	1.00	0.72	0.64	0.47
Overview of the Efficiency metrics in parallel fraction:				
Number of processors	1	2	4	8
Parallel fraction	96.10%			
Global efficiency	79.99%	58.36%	54.36%	42.56%
-- Parallelization strategy efficiency	79.99%	52.23%	48.73%	41.23%
-- Load balancing	100.00%	99.86%	99.27%	97.36%
-- In execution efficiency	79.99%	52.31%	49.09%	42.35%
-- Scalability for computation tasks	100.00%	111.73%	111.56%	103.22%
-- IPC scalability	100.00%	84.05%	85.12%	82.85%
-- Instruction scalability	100.00%	118.64%	118.77%	118.82%
-- Frequency scalability	100.00%	112.05%	110.35%	104.85%
Statistics about explicit tasks in parallel fraction				
Number of processors	1	2	4	8
Number of explicit tasks executed (total)	99669.0	99669.0	99669.0	99669.0
LB (number of explicit tasks executed)	1.0	1.0	0.99	0.98
LB (time executing explicit tasks)	1.0	1.0	1.0	0.99
Time per explicit task (average)	4.67	7.72	8.15	9.85
Overhead per explicit task (synch %)	1.97	40.37	42.72	51.08
Overhead per explicit task (sched %)	32.54	27.86	31.75	39.18
Number of taskwait/taskgroup (total)	2730.0	2730.0	2730.0	2730.0

Figure 25. Screenshot of modelfactors.out of tree parallelization

The first table of figure 25 shows how the efficiency drops when the amount of processors increases. Not as much as the leaf parallelization but it still has a lot to improve.

In the second table we can see that the load balancing is fixed with this strategy due the fact that tree parallelization has a better distribution of the generated tasks.

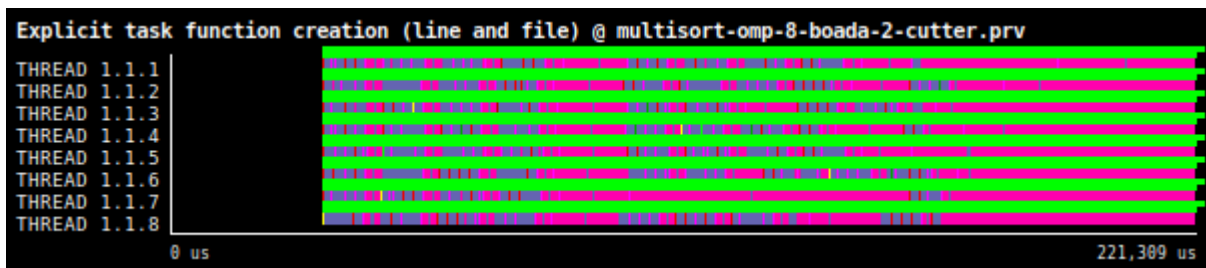


Figure 26. Task creation chronogram of tree parallelization with eight threads

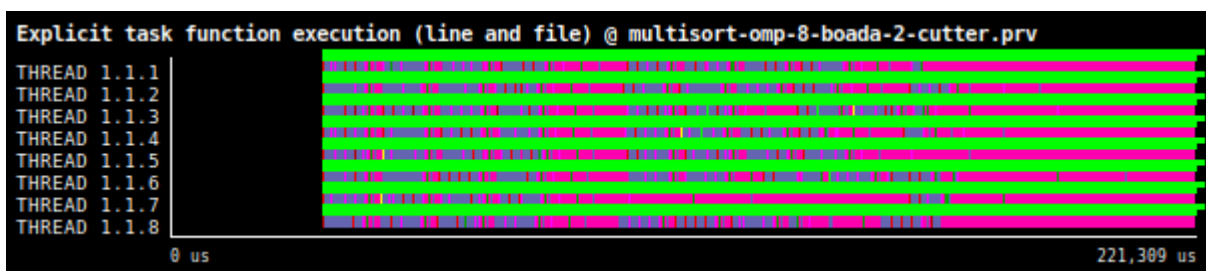


Figure 27. Task execution chronogram of tree parallelization with eight threads

The chronograms from the paraver tool (figure 26 and 27) clearly shows how the task creation and execution have a different form compared to the leaf one. Now not only one thread creates tasks. In this case, one thread creates tasks every time it enters the parallel and single region recursively. Then every other thread starts computing the tasks and whenever they enter the region mentioned before, they create more tasks. We can say that the share of work is now more distributed and in a better shape. Figure 28 shows the total number of executed tasks by every thread complementanting what we have just explained.

	THREAD 1.1.1	THREAD 1.1.2	THREAD 1.1.3	THREAD 1.1.4	THREAD 1.1.5	THREAD 1.1.6	THREAD 1.1.7	THREAD 1.1.8
38 (multisort-omp.c, multisort-omp)	5,469	5,571	5,534	5,646	5,590	5,504	5,998	5,745
40 (multisort-omp.c, multisort-omp)	5,484	5,574	5,525	5,648	5,591	5,510	5,997	5,728
50 (multisort-omp.c, multisort-omp)	178	192	175	169	170	197	101	183
52 (multisort-omp.c, multisort-omp)	175	191	177	170	171	198	102	181
54 (multisort-omp.c, multisort-omp)	175	193	176	172	171	197	101	180
56 (multisort-omp.c, multisort-omp)	175	192	176	173	171	196	102	180
61 (multisort-omp.c, multisort-omp)	176	191	177	171	169	199	103	179
63 (multisort-omp.c, multisort-omp)	175	192	177	172	171	196	102	180
66 (multisort-omp.c, multisort-omp)	175	192	175	173	171	197	102	180
Total	12,182	12,488	12,292	12,494	12,375	12,394	12,708	12,736
Average	1,353.56	1,387.56	1,365.78	1,388.22	1,375	1,377.11	1,412	1,415.11
Maximum	5,484	5,574	5,534	5,648	5,591	5,510	5,998	5,745
Minimum	175	191	175	169	169	196	101	179
StDev	2,203.81	2,236.95	2,225.60	2,276.41	2,253.28	2,207.52	2,451.05	2,309.88
Avg/Max	0.25	0.25	0.25	0.25	0.25	0.25	0.24	0.25

Figure 28. Histogram task execution of tree parallelization with eight threads

The parameter that shows a bad performance is a “in execution efficiency” where it drops considerably. To have a better understanding of the cause of it we are going to sum it up with the third table. It shows that the overhead per explicit task in synchronization has a huge improvement compared to the leaf strategy meaning that this strategy has a better performance, but we can not forget the fact that there is still an increasing amount of overheads but there is still room for improvement. Figure 29 shows what we have already said about the performance of this strategy parallelization but in a more visual way.

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	49.73 %	28.79 %	21.48 %
THREAD 1.1.2	41.70 %	32.69 %	25.61 %
THREAD 1.1.3	41.34 %	33.12 %	25.54 %
THREAD 1.1.4	41.22 %	33.17 %	25.61 %
THREAD 1.1.5	41.16 %	33.22 %	25.62 %
THREAD 1.1.6	41.73 %	32.70 %	25.57 %
THREAD 1.1.7	40.10 %	33.62 %	26.28 %
THREAD 1.1.8	42.11 %	32.30 %	25.60 %
Total	339.07 %	259.62 %	201.31 %
Average	42.38 %	32.45 %	25.16 %
Maximum	49.73 %	33.62 %	26.28 %
Minimum	40.10 %	28.79 %	21.48 %
StDev	2.83 %	1.44 %	1.41 %
Avg/Max	0.85	0.97	0.96

Figure 29. Thread state histogram of the tree parallelization

Controlling task granularities: cut-off mechanism:

After having tested these two types of parallelism strategies, we can say that the tree strat performs better than the leaf strat. So, what we will do now is to continue focusing on the tree strategy by adding a mechanism to control the granularity of the parallelism in order to further improve the code.

This mechanism is called "cut-off", which consists of implementing some comparisons in the code to check up to a certain point, whether to let it continue creating tasks or simply to let it continue with the computation of the task that was previously running.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int depth) {
    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(depth >= CUTOFF)
            merge(n, left, right, result, start, length/2, depth+1);
            #pragma omp task final(depth >= CUTOFF)
            merge(n, left, right, result, start + length/2, length/2, depth+1);
        }
        else{
            //same code as the if section but without task declaration
        }
    }
}

void multisort(long n, T data[n], T tmp[n], int depth) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        if(!omp_in_final()){
            #pragma omp taskgroup
            {
                #pragma omp task final(depth >= CUTOFF)
                multisort(n/4L, &data[0], &tmp[0], depth+1);
                #pragma omp task final(depth >= CUTOFF)
                multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
                #pragma omp task final(depth >= CUTOFF)
                multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
                #pragma omp task final(depth >= CUTOFF)
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);
            }
            #pragma omp taskgroup
            {
                #pragma omp task final(depth >= CUTOFF)
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);
                #pragma omp task final(depth >= CUTOFF)
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, depth+1);
            }
            #pragma omp task final(depth >= CUTOFF)
            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);
        }
        else {
            //same code as the if section but without task declaration
        }
    }
    else {
        // Base case
        basicsort(n, data);
    }
}
```

Figure 30. Screenshot of the key fragment of multisort-omp.c with the implemented tree parallelization and the cutoff mechanism

In this way we can control even better the creation of tasks in a more efficient way. For this purpose we have put the clause `final(depth >= CUTOFF)` which, if the condition is fulfilled, activates a flag so later, with the function `omp_in_final()`, it decides whether to enter or not in the region that keeps creating tasks or in the one that does not.

As we have done with the previous codes, we proceed to execute the code to check its correct operation.

#Threads	Initialization	Multisort	Check sorted data
1	0.855816	6.403373	0.015209
2	0.858475	3.685908	0.017763
4	0.854579	1.885917	0.018066
4 (Leaf)	0.856959	2.015849	0.016894
4 (Tree)	0.854830	1.882833	0.017913

Figure 31. Result table of the execution time with different number of threads of multisort-omp.c

As we can see in Figure 31, no improvement is noticed. This is due to the fact that the script has a predefined cutoff of 16, which does not affect the program since it does not reach that depth level.

To know which cutoff level we should set, we have to study how different cutoff levels affect the execution time.

```
>sbatch ./submit-cutoff-omp.sh 8
```

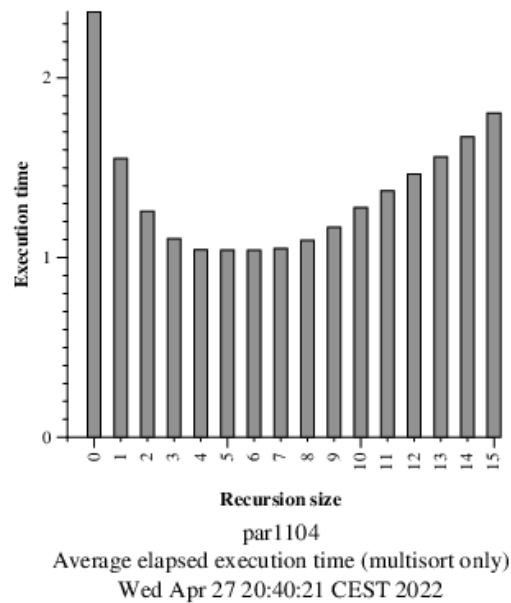


Figure 32. Execution time of multisort-omp with different level of cut-off

Through Figure 32 we can see how different cutoff levels (0-15) affect the execution time. As shown on the graph, a valley shape is created that explains which cutoff levels are better or worse. Thus we can demonstrate that the cutoff really works and ends up improving the program. To better analyze its performance, we will run the code again but with a cutoff level of 4 or 5 or 6 as these are the levels with the best possible performance.

#Threads	Initialization	Multisort	Check sorted data
4 (Cut-off=6)	0.854933	1.713140	0.017934
4 (Cut-off=16)	0.854579	1.885917	0.018066
4 (Leaf)	0.856959	2.015849	0.016894
4 (Tree)	0.854830	1.882833	0.017913

Figure 33. Result table of the execution time with different number of threads of multisort-omp.c

Figure 33 gives us a comparison of how it performs in different parallelization strategies. The cutoff level of 6, as we can see, has reduced its execution time but not that much. For this reason, we proceed to analyze our program in a more in-depth way.

Now that we have understood the details of the cut-off strategy we will look at scalability.

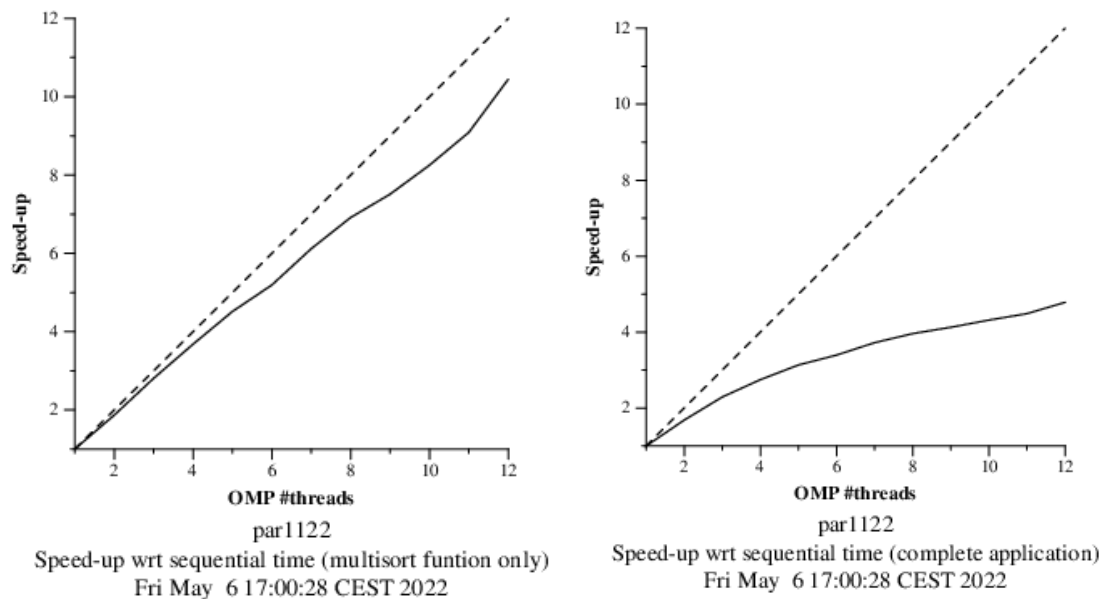


Figure 34. Speedup scalability plots of the tree cut-off level of 6 parallelization in multisort function only (left) and the code in general (right) from 0 to 12 threads.

Note that with a level 6 cutoff we can achieve, as we expected, a very close to the ideal speedup with an increasing number of threads.

To better understand the cutoff strategy, we have used the model factors tool with cutoff level 0 and 1. We can see in figures 35 and 36 how when going from level 0 to 1 there is a substantial increase of tasks performed, this is due to the clause `#pragma omp task final(depth >= CUTOFF)` that we can see in figure 30.

Overview of whole program execution metrics:

Number of processors	1	2	4	8
Elapsed time (sec)	0.20	0.12	0.08	0.09
Speedup	1.00	1.66	2.39	2.30
Efficiency	1.00	0.83	0.60	0.29

Overview of the Efficiency metrics in parallel fraction:

Number of processors	1	2	4	8
Parallel fraction	83.20%			
Global efficiency	99.95%	94.68%	83.14%	39.01%
-- Parallelization strategy efficiency	99.95%	94.99%	83.58%	41.65%
-- Load balancing	100.00%	95.14%	83.83%	45.60%
-- In execution efficiency	99.95%	99.84%	99.70%	91.35%
-- Scalability for computation tasks	100.00%	99.67%	99.47%	93.66%
-- IPC scalability	100.00%	99.76%	99.78%	99.66%
-- Instruction scalability	100.00%	100.00%	99.99%	99.99%
-- Frequency scalability	100.00%	99.92%	99.69%	93.99%

Statistics about explicit tasks in parallel fraction

Number of processors	1	2	4	8
Number of explicit tasks executed (total)	7.0	7.0	7.0	7.0
LB (number of explicit tasks executed)	1.0	0.88	0.58	0.7
LB (time executing explicit tasks)	1.0	0.95	0.84	0.73
Time per explicit task (average)	23376.16	23456.37	23499.5	24951.06
Overhead per explicit task (synch %)	0.02	5.22	19.56	140.04
Overhead per explicit task (sched %)	0.03	0.03	0.04	0.04
Number of taskwait/taskgroup (total)	2.0	2.0	2.0	2.0

Figure 35. Screenshot of modelfactors.out with a cutoff level of 0

Overview of whole program execution metrics:

Number of processors	1	2	4	8
Elapsed time (sec)	0.20	0.11	0.08	0.06
Speedup	1.00	1.71	2.57	3.04
Efficiency	1.00	0.86	0.64	0.38

Overview of the Efficiency metrics in parallel fraction:

Number of processors	1	2	4	8
Parallel fraction	83.81%			
Global efficiency	99.88%	99.52%	93.99%	65.84%
-- Parallelization strategy efficiency	99.88%	99.40%	94.60%	70.61%
-- Load balancing	100.00%	99.77%	95.16%	84.04%
-- In execution efficiency	99.88%	99.63%	99.41%	84.02%
-- Scalability for computation tasks	100.00%	100.13%	99.36%	93.24%
-- IPC scalability	100.00%	100.10%	99.70%	99.04%
-- Instruction scalability	100.00%	100.01%	100.00%	100.00%
-- Frequency scalability	100.00%	100.01%	99.65%	94.15%

Statistics about explicit tasks in parallel fraction

Number of processors	1	2	4	8
Number of explicit tasks executed (total)	41.0	41.0	41.0	41.0
LB (number of explicit tasks executed)	1.0	0.98	0.93	0.73
LB (time executing explicit tasks)	1.0	1.0	0.95	0.84
Time per explicit task (average)	3994.83	3995.58	4027.86	4292.26
Overhead per explicit task (synch %)	0.04	0.5	5.53	41.33
Overhead per explicit task (sched %)	0.07	0.08	0.13	0.17
Number of taskwait/taskgroup (total)	10.0	10.0	10.0	10.0

Figure 36. Screenshot of modelfactors.out with a cutoff level of 1

We can see how in the second table the load balancing is better in cutoff level 1 compared to level 0, this could be due to the fact that there are more tasks (third table) and therefore there is more balance between them. As there are more tasks at cutoff level 1, the time taken for each task is lower as we can see in figures 35 and 36.

Optional 1:

As we have seen in the previous results, we have improved the performance of our program considerably. But now we want to see how far we can get with our program if we execute it with more threads than physical cores Boada has available. We will increase the possible range of threads of the provided submit-strong-omp.sh script to 24 to get the following plots.

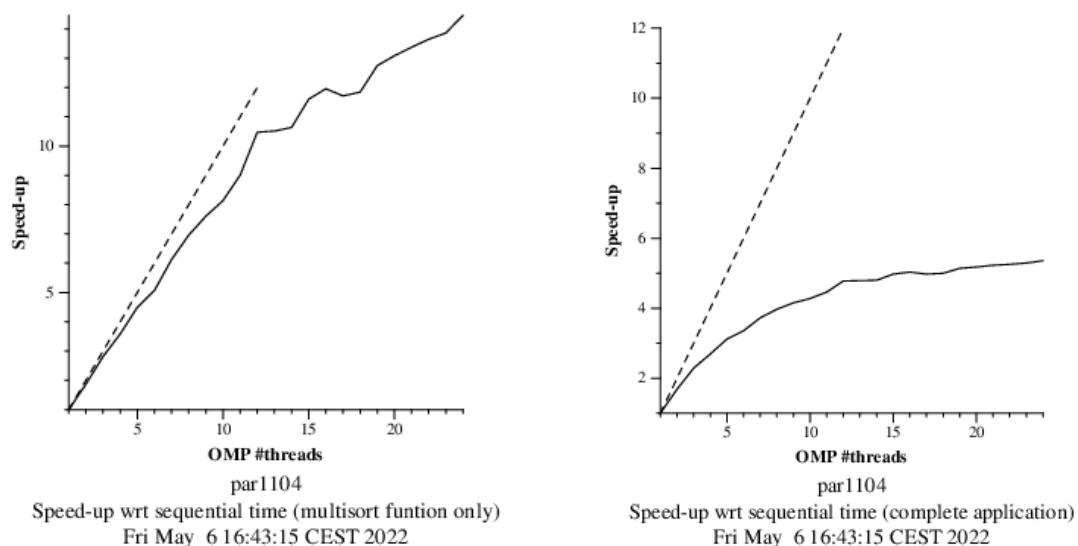


Figure 37. Speedup scalability plots of the tree cut-off level of 6 parallelization in multisort function only (left) and the code in general (right) from 0 to 24 threads.

Figure 37 shows the scalability of our program up to 24 threads. Even if we only have 12 physical cores, we can achieve even better performance. It is due to the fact that enough tasks can be created to be able to feed the 12 cores or physical threads simultaneously. This is possible thanks to the Boada architecture that we already discussed in our first lab sessions.

We have found it convenient not to run the program with the modelfactors since with the two plots obtained previously we can already reach the conclusions we needed and it would be redundant.

Parallelisation and performance analysis with task dependencies

In this last session, we will use all the knowledge obtained in the previous sections to finally find an even better parallelism. In order to do so, we will keep using tree parallelization with the cut-off mechanism at level 6, since it is the one we get the best results from.

In the previous section, we had to create taskgroup clauses to maintain the correctness of the code since we were interested in having tasks waiting for the rest of the group before moving on to execute the next task and thus avoid data races and possible sorting failures. This has the problem of applying the restrictions to all the tasks inside the clause instead of just restringing each task depending on the inputs and outputs it has. To solve this, we will use depend() clauses that will allow us to declare dependencies, in and out, for each declared task.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int depth) {
    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(depth >= CUTOFF)
            merge(n, left, right, result, start, length/2, depth+1);
            #pragma omp task final(depth >= CUTOFF)
            merge(n, left, right, result, start + length/2, length/2, depth+1);
            #pragma omp taskwait
        }
        else{
            //same code as the if section but without task declaration
        }
    }
}

void multisort(long n, T data[n], T tmp[n], int depth) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        if(!omp_in_final()){
            #pragma omp task final(depth >= CUTOFF) depend(out: data[0])
            multisort(n/4L, &data[0], &tmp[0], depth+1);
            #pragma omp task final(depth >= CUTOFF) depend(out: data[n/4L])
            multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
            #pragma omp task final(depth >= CUTOFF) depend(out: data[n/2L])
            multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
            #pragma omp task final(depth >= CUTOFF) depend(out: data[3L*n/4L])
            multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);

            #pragma omp task final(depth >= CUTOFF) depend(in: data[0], data[n/4L]) depend(out: tmp[0])
            merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);
            #pragma omp task final(depth >= CUTOFF) depend(in: data[n/2L], data[3L*n/4L]) depend(out: tmp[n/2L])
            merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, depth+1);

            #pragma omp task final(depth >= CUTOFF) depend(in: tmp[0], tmp[n/2L])
            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);
            #pragma omp taskwait
        }
        else {
            //same code as the if section but without task declaration
        }
    }
    else {
        // Base case
        basicsort(n, data);
    }
}
```

Figure 38. Screenshot of the key fragment of multisort-omp.c with the implemented tree parallelization, cutoff mechanism and task dependencies declared

In each declared task, as we can see in figure 38, along with the cut-off clauses, we have instantiated data-in and data-out dependencies to ensure that it executes correctly. We have also added a taskwait at the end of the region to prevent it from running without waiting for the rest and to be a barrier for the last declared dependencies.

To check that our code works as expected, we will execute it and save the results.

#Threads	Initialization	Multisort	Check sorted data
8 (Cut-off=6 + depend)	0.857004	0.919051	0.017618
4 (Cut-off=6 + depend)	0.854463	1.718007	0.017980
4 (Cut-off=6)	0.854933	1.713140	0.017934
4 (Cut-off=16)	0.854579	1.885917	0.018066
4 (Leaf)	0.856959	2.015849	0.016894
4 (Tree)	0.854830	1.882833	0.017913

Figure 39. Result table of the execution time with different number of threads of multisort-omp.c

Figure 39 shows a comparison of all the executions of all the strategies performed with 4 threads in order to compare them on a large scale. As we can see, there is no improvement between using or not using the dependency clauses. But in order to better understand their performance, we will try to create scalability graphs to see if there really is the improvement we are looking for (figure 40). We have also added the execution with 8 threads to see at a glance its performance and to finally check that it does not pop up any error at the time of sorting.

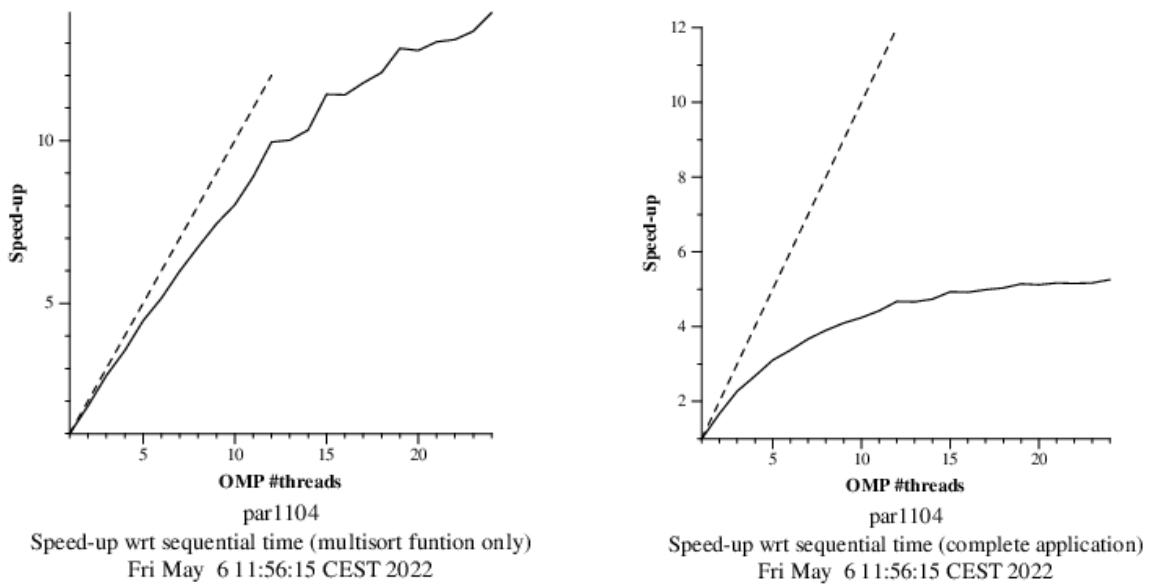


Figure 40. Speedup scalability plots of the tree cut-off depend parallelization in multisort function only (left) and the code in general (right) [cut-off = 6]

If we look at the two graphs generated (figure 40), we can say that this strategy, in addition to the declared dependencies, achieves a better scalability than the original one studied. We can even go to the point of saying that it is close to ideal in the first phases since it grows in a fairly linear growth.

If we compare them with the strategy without dependency clauses but with taskgroups, we can see that they are very similar. This explains a lot why the tables show very similar results. We really expected to find results in favor of the latter strategy, but in the end we can say that the two are very similar and have close results.

If we have to choose one of the two, personally, we will choose the strategy that uses taskgroups instead of `depend()` because it ends up being more comfortable to code by not having to look at all its possible dependencies. But it can be more practical with `depend()` because we can have more control of those dependencies, if that is what we are interested in.

Overview of whole program execution metrics:

Overview of the Efficiency metrics in parallel fraction:

Statistics about explicit tasks in parallel fraction

Figure 40, 41 and 42 shows us in detail how it performs, just as we did in the previous section. With this output we can still say the same thing: there is no big difference or significant improvement with the previous taskgroup strategy. But we can highlight how these last two strategies end up giving a better shape to the original code.

```
Explicit task function creation (line and file) @ multisort-omp-8-boada-3-cutter.prv
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
```

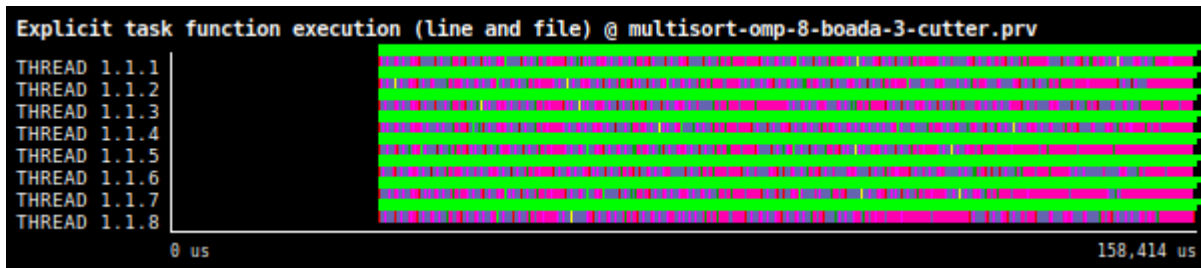


Figure 43. Task execution chronogram of tree cut-off depend parallelization with eight threads

Optional 2:

In the previous sections we have been working with different versions of the mergesort and multisort functions to find better levels of parallelization. But now we will try to parallelize the code without modifying these two functions. We will also parallelize the initialization of the randomized data buffer and the zero-initialization of the tmp vector.

```
static void initialize(long length, T data[length]) {
    long i;
    #pragma omp taskloop private(i)
    for (i = 0; i < length; i++) {
        if (i==0) {
            data[i] = rand();
        } else {
            data[i] = ((data[i-1]+1) * i * 104723L) % N;
        }
    }
}

static void clear(long length, T data[length]) {
    long i;
    #pragma omp taskloop private(i)
    for (i = 0; i < length; i++) {
        data[i] = 0;
    }
}

int main(int argc, char **argv) {
    // code ...
    #pragma omp parallel
    #pragma omp single
    {
        initialize(N, data);
        clear(N, tmp);
        multisort(N, data, tmp, 0);
    }
    // code ...
}
```

Figure 44. Screenshot of the key fragment of multisort-omp.c of the parallelization of initialization of data and tmp vectors

As we can see in figure 44, we have added a taskloop pragma in both initializations (initialize() and clean()) together with the private(i) clause to avoid problems in the content of that variable. As we have done before, we have included the two new functions in the parallel and single region together with the multisort function.

Now we will proceed to execute the code to check its correctness and to see if it really improves its performance. First of all we will create scalability plots and then we will attach a table with the execution times obtained in order to compare them with the rest of the strategies.

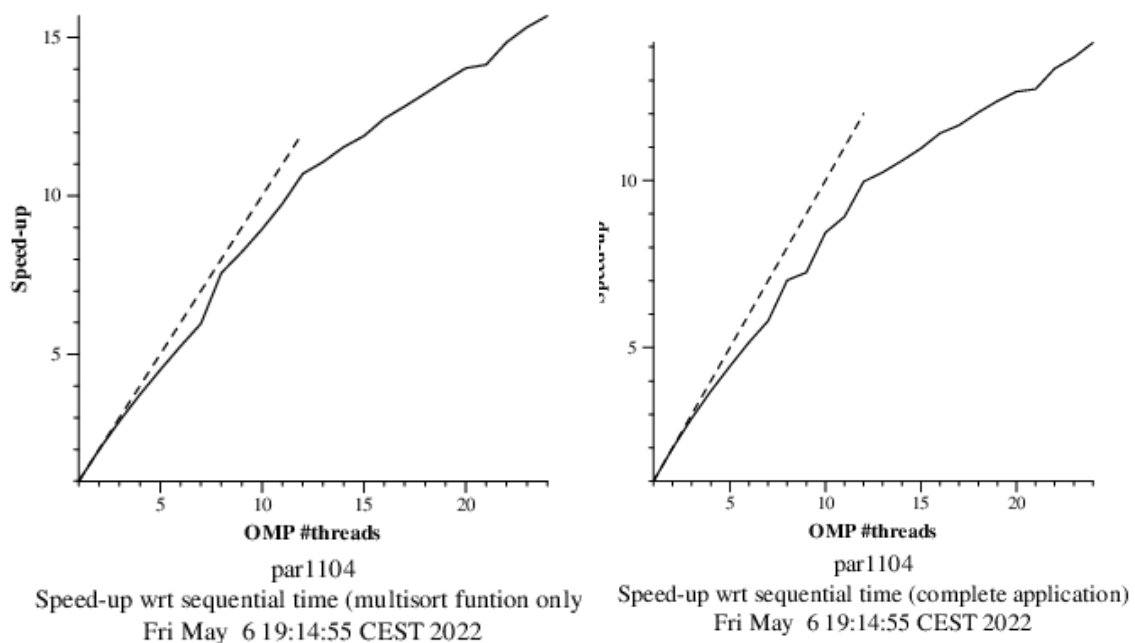


Figure 45. Speedup scalability plots of the tree cut-off depend parallelization with initialization parallelization in multisort function only (left) and the code in general (right) [cut-off = 6]

#Threads	Initialization	Multisort	Check sorted data
4 (Cut-off=6 + depend + init parallelization)	0.208361	1.795938	0.017892
4 (Cut-off=6 + depend)	0.854463	1.718007	0.017980
4 (Cut-off=6)	0.854933	1.713140	0.017934
4 (Cut-off=16)	0.854579	1.885917	0.018066
4 (Leaf)	0.856959	2.015849	0.016894
4 (Tree)	0.854830	1.882833	0.017913

Figure 46. Result table of the execution time with different number of threads of multisort-omp.c

Thanks to figure 45 we can see how this version still has a very linear and almost ideal scalability in the multisort part. But the most remarkable of this figure is the complete application plot since we have finally managed to modify the growth of it and also meaning a great improvement of its performance. The two graphs grow linearly and represent a good performance. This can also be seen in the table in figure 46, where we can see how in the current version of the code, we have managed to significantly reduce the execution time of the initialization section.

For a deeper analysis of what we have explained, we will proceed to execute our code version with the modelfactors and thus obtain much more accurate results.

Overview of whole program execution metrics:

Number of processors	1	2	4	8
Elapsed time (sec)	0.41	0.23	0.12	0.08
Speedup	1.00	1.79	3.30	5.41
Efficiency	1.00	0.90	0.82	0.68

Overview of the Efficiency metrics in parallel fraction:

Number of processors	1	2	4	8
Parallel fraction	99.42%			
Global efficiency	80.09%	72.27%	67.21%	56.36%
-- Parallelization strategy efficiency	80.09%	69.79%	67.31%	60.41%
-- Load balancing	100.00%	99.58%	98.60%	97.61%
-- In execution efficiency	80.09%	70.08%	68.26%	61.89%
-- Scalability for computation tasks	100.00%	103.55%	99.85%	93.30%
-- IPC scalability	100.00%	94.30%	93.80%	92.71%
-- Instruction scalability	100.00%	105.48%	105.49%	105.46%
-- Frequency scalability	100.00%	104.10%	100.91%	95.42%

Statistics about explicit tasks in parallel fraction

Number of processors	1	2	4	8
Number of explicit tasks executed (total)	25577.0	25587.0	25607.0	25647.0
LB (number of explicit tasks executed)	1.0	1.0	0.97	0.97
LB (time executing explicit tasks)	1.0	0.99	0.99	0.96
Time per explicit task (average)	10.29	14.45	15.4	17.76
Overhead per explicit task (synch %)	6.1	23.68	25.42	31.25
Overhead per explicit task (sched %)	24.82	13.34	14.96	19.2
Number of taskwait/taskgroup (total)	9368.0	9368.0	9368.0	9368.0

Figure 47. Screenshot of modelfactors.out of the tree cut-off depend parallelization in multisort function in addition of the initialization parallelization

Figure 47 shows more detailed information about the execution of our code. We can see how in the first table the speedup grows considerably, which corroborates what we had mentioned previously in figure 45.

Furthermore, if we look at the second table and compare it with Figure 41 (execution without parallelization of the initialization phase), the efficiency is approximately 10%/20% better. This is a remarkable point since it is one of the factors that directly affect the performance of our program.

Finally, if we focus on the third table and compare it also with Figure 41, less overheads are generated per explicit task in synchronization as we can see in Figure 48.

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	60.87 %	23.88 %	15.26 %
THREAD 1.1.2	59.76 %	24.61 %	15.63 %
THREAD 1.1.3	59.05 %	25.59 %	15.36 %
THREAD 1.1.4	60.31 %	24.14 %	15.55 %
THREAD 1.1.5	60.17 %	24.59 %	15.24 %
THREAD 1.1.6	62.32 %	22.96 %	14.72 %
THREAD 1.1.7	59.22 %	24.87 %	15.91 %
THREAD 1.1.8	59.15 %	25.19 %	15.66 %

Figure 48. Thread state histogram of the tree parallelization

After having analyzed the performance of this version of the code, we can affirm that parallelizing the initialization region is an improvement of our program.

Final conclusions

In this lab we have seen how rigorous the task of parallelizing a program, such as ours, can be for a programmer. We started from a base code and analyzed two possible strategies. From there, we were able to quickly discard the leaf strategy because of its poorer performance. Then, our task was to try different levels of parallelization of the program until we reached a desired level based on the tree strategy.

Behind this task, there is a thorough work to maintain all possible dependencies without losing the correctness of our program. We have seen how the cut-off mechanism worked with our code, seeing how, depending on the level of depth we treat, we can improve its performance. Specifically, the range of cut-off levels where a considerable improvement could be observed was between 4 and 6 because we better controlled the creation of tasks. To make sure this strategy worked, we implemented taskgroup clauses.

After analyzing this parallelization, we opted to study the same strategy but, instead of taskgroup clauses, to use dependencies and thus have even tighter control. We thought that it would optimize the program more, when in fact, its performance was almost the same as using taskgroup clauses.

Finally, we can say that we have successfully achieved our goal of reducing the execution time of our program by working with different levels of parallelization. We have gone from a time of around 6 seconds to about 1 second. We, as programmers, if we have to choose one of the previously analyzed strategies to parallelize our program, we would opt for the tree cut-off level of 6 with taskgroup clauses parallelization.