PAR Laboratory Assignment

Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Teacher:

Jordi Tubella

Students:

Eric Hurtado (par2114)

Jordi Pitarch (par2118)

Index:

1. Introduction

In this lab session we are going to learn two different generation strategies related to the parallelization of a Tree Task Structure, these two strategies are leaf recursive task decomposition and tree recursive task decomposition.
Firstly, we will proceed to analyze the task dependencies using Tareador, in the second section we are going to analyze the two task strategies when there are no dependencies, and finally we will repeat the analysis with task dependencies.

2. Task decomposition analysis for Mergesort

2.1 “Divide and conquer”

Firstly, we have executed using the commands:

- make multisort-seq
- sbatch submit-seq.sh multisort-seq

In order to see the execution time sequentially of the program. We got the following results:

```
************************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
************************************************************************************************
Initialization time in seconds: 0.681402
Multisort execution time: 5.187507
Check sorted data execution time: 0.011800
Multisort program finished
************************************************************************************************
```

This execution has been done considering a N=32768, MIN_SORT_SIZE=1024 and MIN_MERGE_SIZE=1024.

2.2 Task decomposition analysis with Tareador

Now we are going to implement two different strategies, the leaf and the tree strategies.

In the leaf strategy, you should define a task for the invocations of **basicsort** and **basicmerge** once the recursive divide–and–conquer decomposition stops. We have do it like the following image:

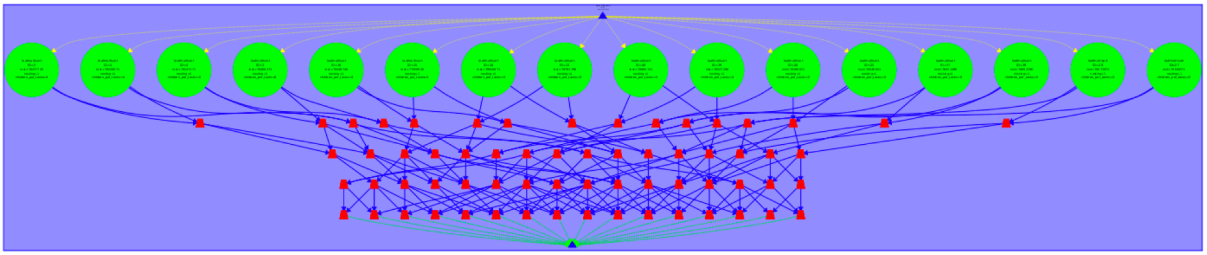
```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        tareador_start_task("leafmerge");
        basicmerge(n, left, right, result, start, length);
        tareador_end_task("leafmerge");
    } else {
        // 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("leafmultisort");
        basicsort(n, data);
        tareador_end_task("leafmultisort");
    }
}
```

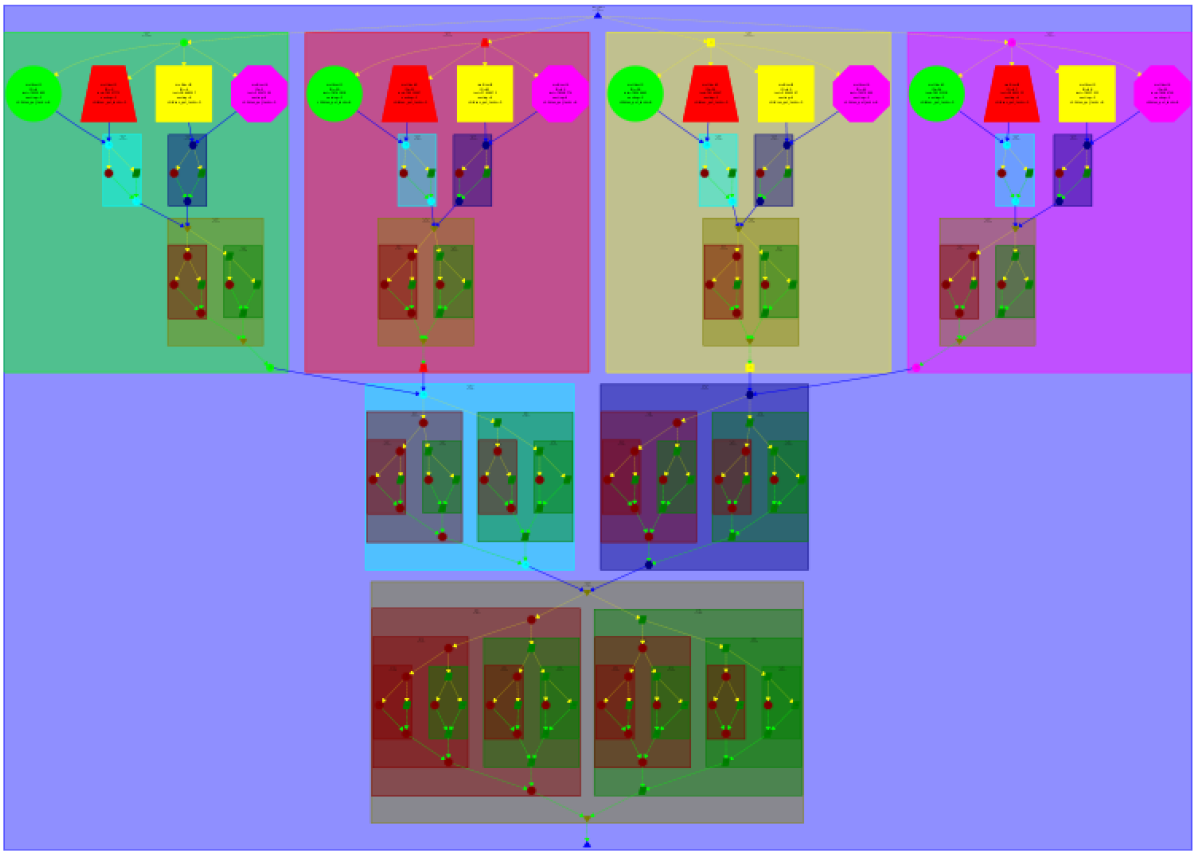
In the tree strategy, you should define tasks during the recursive decomposition, i.e. when invoking multisort and merge. We have implemented the code as the following image:

```
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("merge1");
        merge(n, left, right, result, start, length/2);
        tareador_end_task("merge1");
        tareador_start_task("merge2");
        merge(n, left, right, result, start + length/2, length/2);
        tareador_end_task("merge2");
    }
}

void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        tareador_start_task("multisort1");
        multisort(n/4L, &data[0], &tmp[0]);
        tareador_end_task("multisort1");
        tareador_start_task("multisort2");
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        tareador_end_task("multisort2");
        tareador_start_task("multisort3");
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        tareador_end_task("multisort3");
        tareador_start_task("multisort4");
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
        tareador_end_task("multisort4");
        tareador_start_task("multisort5");
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        tareador_end_task("multisort5");
        tareador_start_task("multisort6");
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        tareador_end_task("multisort6");
        tareador_start_task("multisort7");
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        tareador_end_task("multisort7");
    } else {
        // Base case
        basicsort(n, data);
    }
```

After modifying the code with both strategies, we have decided to execute the paraver because we wanted to see if the decisions implemented in the code are correct or not.

Firstly, we can see the leaf strategy and after the tree strategy:



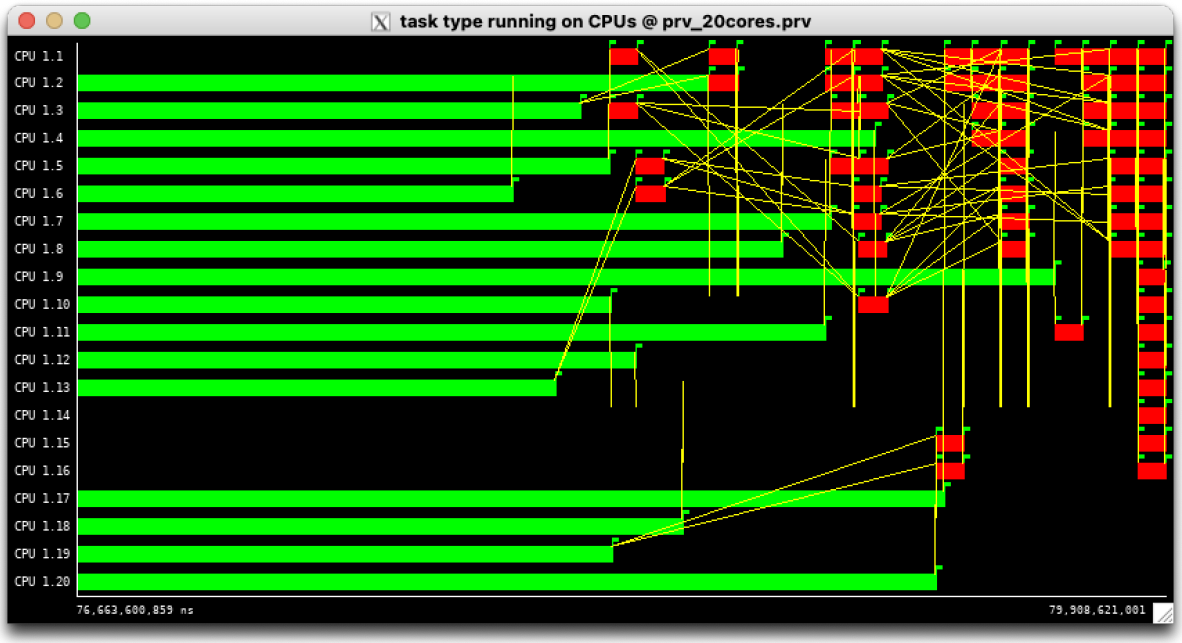


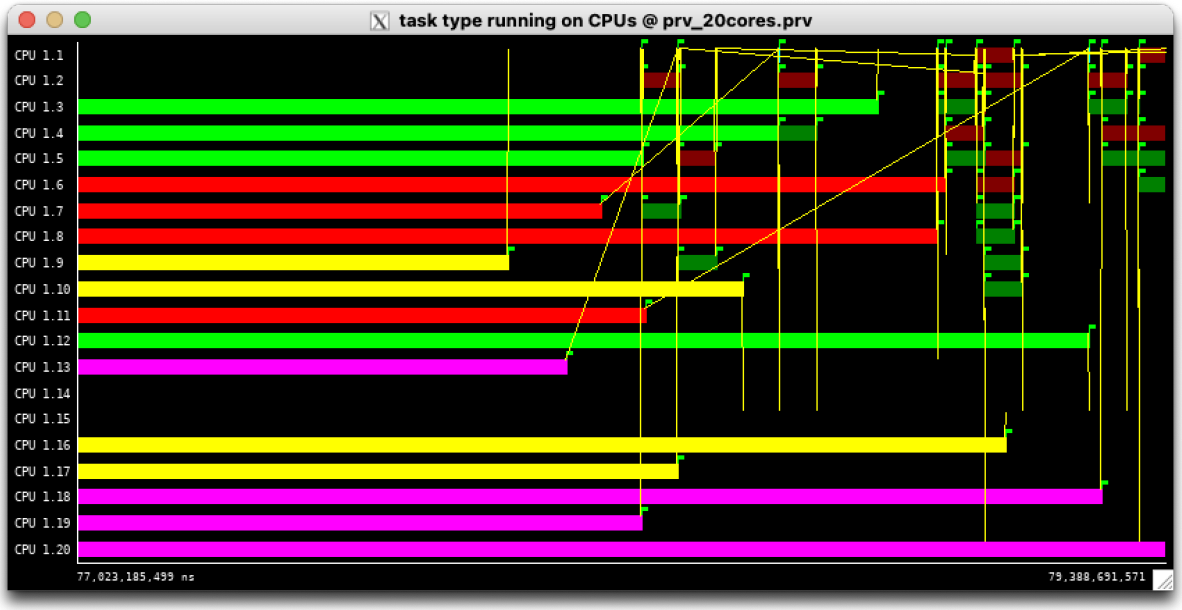
As we can see in the 2 task dependence graphs, there are a lot of things different between one strategy and another strategy. In the tree strategy there are more tasks compared with the leaf strategy, this is due to the fact that there are more invocations for the **merge** and **multisort** functions rather than the **basicsort** and **basicmerge**. On the other hand, the structure of the graphs generated are different, because both strategies generate tasks in different places of the code. If we consider the granularity, in the leaf strategy there is a big imbalance in the tasks generated by the **basicsort** and **basicmerge**, due to the fact that they are not executing the same code, in the tree strategy, we also can see a big imbalance between the first multisort tasks, the four top tasks, and the other tasks generated inside them and the tasks generated after these four big tasks. The four big initial tasks are created

by the multisort initial, between each brand there are no dependencies, these dependencies appear when they do the merge, and then they need to be synchronized.

If we want a synchronization between tasks, we should create explicit tasks with “pragma omp task” and synchronize the tasks with “pragma omp taskwait” in order to wait for the other tasks that are required for the execution of the code. In our case we should add tree taskwait, one before the multisort5, waiting the multisort1 and multisort2, another before the multisort6, waiting the multisort3, and multisort4, and the last before the multisort7, waiting multisort5 and multisort6.

Finally, we have executed the code with 20 threads using Paraver, and we had got the following images, the first from the leaf strategy and the second from the tree strategy:





As we can see in both images, the merge is the function that creates dependencies between tasks, because this has to wait until the end of the tasks that are needed. On the other hand the multisort does not generate any type of dependencies due to the fact that only takes a part of the original vector.

Also, we can see that despite executing with 20 threads, this could have been executed with 16 threads because this does not use all threads.

3. Shared-memory parallelisation with OpenMP tasks

3.1 Leaf strategy in OpenMP

Firstly, we have implemented the leaf strategy using “taskwait”. We have changed the code in order to make a task in each base case, and we have used the taskwait in the function **multisort** in the different parts of the recursive decomposition. In the following image you can see the code:

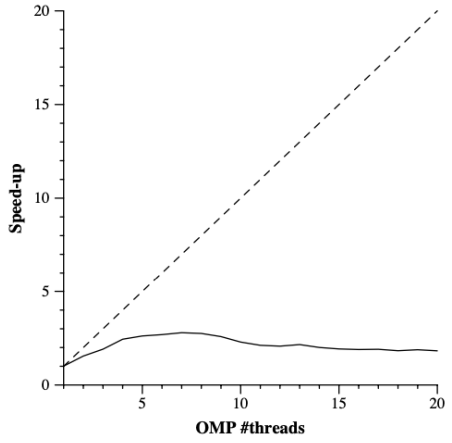
```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        #pragma omp task
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        merge(n, left, 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);
    }
}
```

In order to see if the result is correct, we have executed the following command:

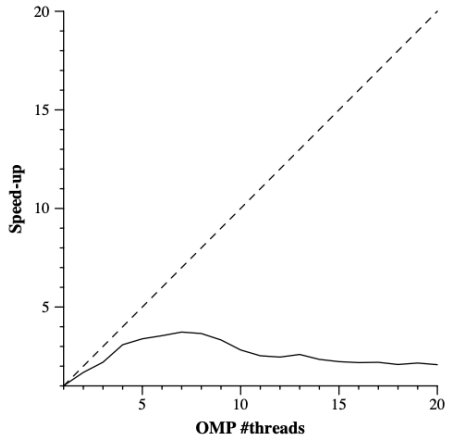
sbatch submit-omp.sh multisort-omp 2

We have done it in 40 threads because we wanted to see the result with a lot of threads, and after several executions we always get the same result, which can be seen in the following image:

```
************************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=50
Number of threads in OpenMP:          OMP_NUM_THREADS=2
************************************************************************************************
Initialization time in seconds: 0.684361
Multisort execution time: 3.144940
Check sorted data execution time: 0.013939
Multisort program finished
************************************************************************************************
```



par2118
Speed-up wrt sequential time (complete application)
Generated by par2118 on Tue Nov 15 10:54:06 AM CET 2022



par2118
Speed-up wrt sequential time (multisort funtion only)
Generated by par2118 on Tue Nov 15 10:54:06 AM CET 2022

As we can see in the plots, the speedup that we can see is far from being the ideal we are expecting. After the execution with 8 threads, we can see that the speedup decrease instead of increasing it.

As a result, this is not scalable, in order to see what is happening, we are going to execute the modelfactors.

If we execute modelfactors, we get the following tables:

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.21	0.24	0.20	0.20	0.20	0.21	0.23	0.26	0.25
Speedup	1.00	0.88	1.04	1.03	1.03	0.98	0.92	0.81	0.82
Efficiency	1.00	0.44	0.26	0.17	0.13	0.10	0.08	0.06	0.05

Table 1: Analysis done on Tue Nov 15 11:14:02 AM CET 2022, par2118

Overview of the Efficiency metrics in parallel fraction, ϕ=89.37%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	91.79%	39.84%	24.15%	15.90%	12.00%	9.14%	7.01%	5.23%	4.68%
Parallelization strategy efficiency	91.79%	53.33%	38.50%	27.34%	21.27%	16.86%	12.91%	10.55%	9.37%
Load balancing	100.00%	98.01%	91.83%	62.62%	40.66%	30.71%	24.24%	17.71%	15.58%
In execution efficiency	91.79%	54.41%	41.93%	43.67%	52.30%	54.90%	53.25%	59.53%	60.12%
Scalability for computation tasks	100.00%	74.71%	62.72%	58.16%	56.43%	54.22%	54.29%	49.55%	49.96%
IPC scalability	100.00%	67.92%	57.13%	54.82%	53.81%	53.04%	52.57%	48.36%	48.72%
Instruction scalability	100.00%	111.80%	113.02%	112.76%	111.94%	111.81%	111.83%	111.13%	110.92%
Frequency scalability	100.00%	98.39%	97.14%	94.09%	93.69%	91.43%	92.35%	92.19%	92.45%

Table 2: Analysis done on Tue Nov 15 11:14:02 AM CET 2022, par2118

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0
LB (number of explicit tasks executed)	1.0	0.72	0.75	0.77	0.78	0.81	0.79	0.75	0.79
LB (time executing explicit tasks)	1.0	0.8	0.81	0.78	0.78	0.81	0.81	0.78	0.82
Time per explicit task (average us)	2.73	3.55	4.1	4.15	4.03	4.1	3.99	3.97	3.96
Overhead per explicit task (synch %)	0.89	69.51	159.74	322.67	499.7	692.05	978.09	1365.59	1555.14
Overhead per explicit task (sched %)	9.72	37.29	41.62	33.74	27.02	26.36	31.79	31.35	30.0
Number of taskwait/taskgroup (total)	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0

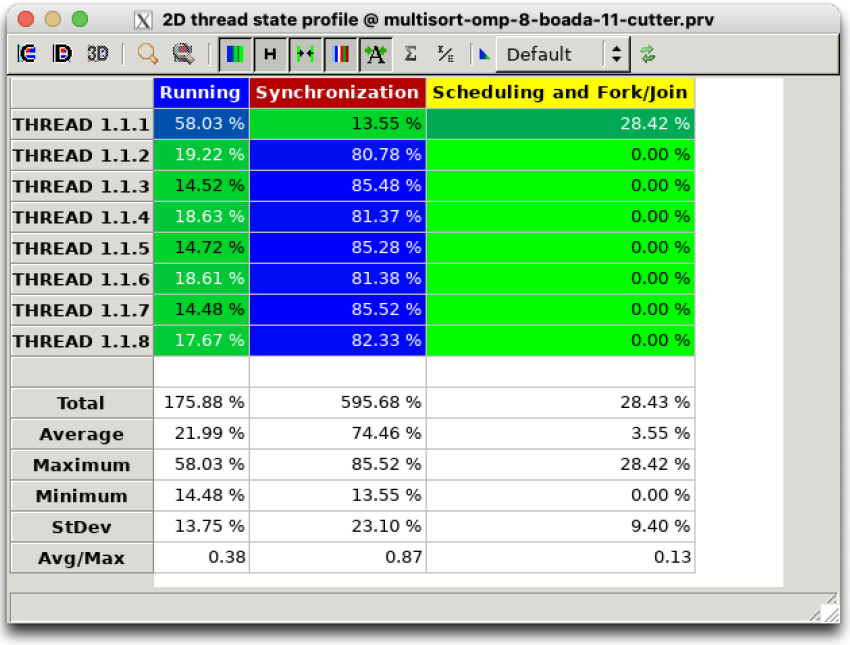
Table 3: Analysis done on Tue Nov 15 11:14:02 AM CET 2022, par2118

If we analyze the result, we can see that the global efficiency decreases a lot in relation to the threads, also with the parallelization strategy efficiency happens the same. Despite being an 89.37% of parallel fraction, we do not get the wished result.

If we see the statics about explicit tasks, we can see that the overhead per explicit task related with the synchronization is so high that provokes the decrease of efficiency.

Finally, in wxparaver we can see with 8 threads the following image:

In the image we can see that the thread one is who is creating the other tasks, and is who is executing the code most of the time, while the others are on synchronization most of the execution time.



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	58.03 %	13.55 %	28.42 %
THREAD 1.1.2	19.22 %	80.78 %	0.00 %
THREAD 1.1.3	14.52 %	85.48 %	0.00 %
THREAD 1.1.4	18.63 %	81.37 %	0.00 %
THREAD 1.1.5	14.72 %	85.28 %	0.00 %
THREAD 1.1.6	18.61 %	81.38 %	0.00 %
THREAD 1.1.7	14.48 %	85.52 %	0.00 %
THREAD 1.1.8	17.67 %	82.33 %	0.00 %
Total	175.88 %	595.68 %	28.43 %
Average	21.99 %	74.46 %	3.55 %
Maximum	58.03 %	85.52 %	28.42 %
Minimum	14.48 %	13.55 %	0.00 %
StDev	13.75 %	23.10 %	9.40 %
Avg/Max	0.38	0.87	0.13

In this image we can see in numbers what we have said before about synchronizations, and the fork/join times and the running time of each thread.

In order to compare this strategy, we are going to implement the tree strategy and see the differences.

3.2 Tree strategy in OpenMP

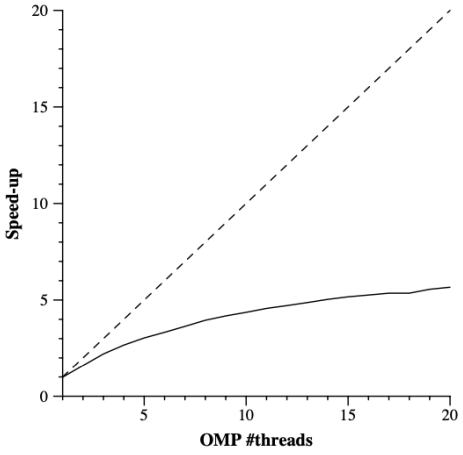
In this new strategy we have changed the point where the tasks are created in order to do not create these tasks in the base tasks. We have implemented the code using the taskgroup clause like we can see in the following image:

```
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);
    }
}
```

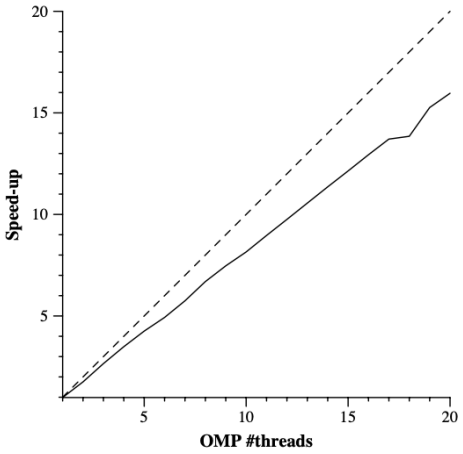
If we execute several times the sbatch command with two threads, as in the previous strategy, we can see the following result:

```
************************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=50
Number of threads in OpenMP:          OMP_NUM_THREADS=2
************************************************************************************************
Initialization time in seconds: 0.683089
Multisort execution time: 2.956374
Check sorted data execution time: 0.014830
Multisort program finished
************************************************************************************************
```

In the other hand, we have executed with the submit-omp.sh in order to see the plots that we get with this new code.



par2118
Speed-up wrt sequential time (complete application)
Generated by par2118 on Tue Nov 15 11:42:48 AM CET 2022



par2118
Speed-up wrt sequential time (multisort funtion only)
Generated by par2118 on Tue Nov 15 11:42:48 AM CET 2022

If we compare the graphs with the tree strategy with the leaf strategy, we can see that now the speedup increases with more threads. Although the speedup increase is not the same when we have few threads or when we have more threads.

By the other hand in the multisort function only, we get a speedup similar to the ideal as a consequence that the tasks now are created among the task instead of only one like we had before.

In order to see more details, we are going to execute the model factors in order to see more differences.

With the modelfactors we got the following tables:

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.26	0.31	0.25	0.23	0.23	0.22	0.22	0.22	0.22
Speedup	1.00	0.87	1.06	1.15	1.17	1.20	1.19	1.21	1.20
Efficiency	1.00	0.43	0.26	0.19	0.15	0.12	0.10	0.09	0.08

Table 1: Analysis done on Tue Nov 15 11:57:16 AM CET 2022, par2118

Overview of the Efficiency metrics in parallel fraction, ϕ=91.21%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	89.07%	38.12%	23.73%	17.32%	13.26%	10.98%	9.07%	7.89%	6.89%
Parallelization strategy efficiency	89.07%	49.19%	34.72%	27.54%	20.68%	17.09%	14.23%	12.26%	10.57%
Load balancing	100.00%	95.21%	96.29%	94.63%	91.72%	93.32%	91.56%	89.33%	93.81%
In execution efficiency	89.07%	51.67%	36.06%	29.10%	22.54%	18.31%	15.55%	13.72%	11.27%
Scalability for computation tasks	100.00%	77.49%	68.35%	62.89%	64.13%	64.29%	63.74%	64.36%	65.18%
IPC scalability	100.00%	63.64%	57.86%	55.92%	56.35%	58.48%	58.86%	59.73%	60.25%
Instruction scalability	100.00%	121.64%	121.36%	121.30%	121.62%	121.51%	121.54%	121.37%	121.70%
Frequency scalability	100.00%	100.10%	97.33%	92.71%	93.59%	90.48%	89.11%	88.77%	88.91%

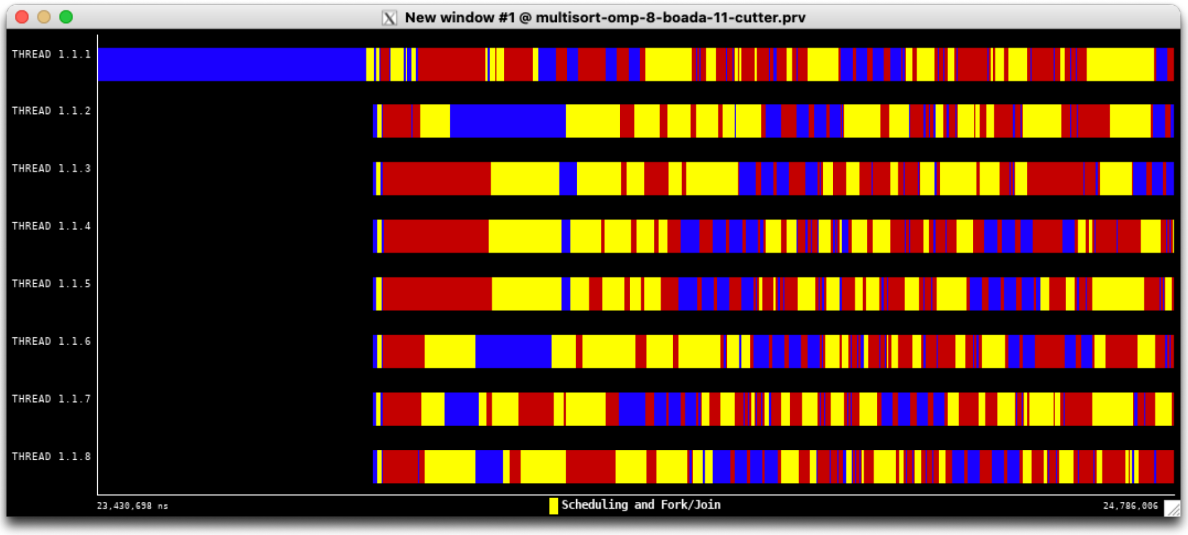
Table 2: Analysis done on Tue Nov 15 11:57:16 AM CET 2022, par2118

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0
LB (number of explicit tasks executed)	1.0	1.0	0.97	0.97	0.98	0.98	0.97	0.96	0.98
LB (time executing explicit tasks)	1.0	0.99	1.0	0.99	0.99	1.0	0.99	0.99	0.99
Time per explicit task (average us)	1.85	3.96	5.7	7.38	9.19	10.82	12.93	14.64	16.53
Overhead per explicit task (synch %)	1.02	41.23	57.18	66.45	74.64	79.22	81.65	84.4	86.99
Overhead per explicit task (sched %)	13.27	31.43	46.92	55.91	65.81	71.35	76.15	79.65	82.49
Number of taskwait/taskgroup (total)	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0

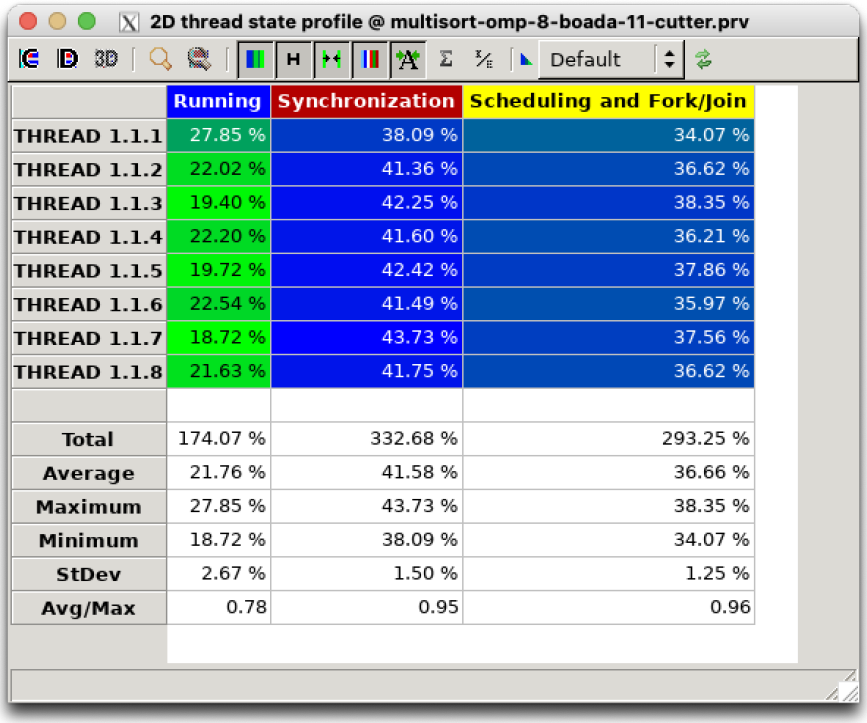
Table 3: Analysis done on Tue Nov 15 11:57:16 AM CET 2022, par2118

In the tables, we can see that now the overhead per explicit tasks in synchronization is fewer tan before, thanks to this fact, we get an increase in the speedup in the function multisort.

Finally we have executed the wxparaver.



As we can see in the image, now all threads are creating tasks instead of in the other strategy, as a result the execution time is done by all threads.



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	27.85 %	38.09 %	34.07 %
THREAD 1.1.2	22.02 %	41.36 %	36.62 %
THREAD 1.1.3	19.40 %	42.25 %	38.35 %
THREAD 1.1.4	22.20 %	41.60 %	36.21 %
THREAD 1.1.5	19.72 %	42.42 %	37.86 %
THREAD 1.1.6	22.54 %	41.49 %	35.97 %
THREAD 1.1.7	18.72 %	43.73 %	37.56 %
THREAD 1.1.8	21.63 %	41.75 %	36.62 %
Total	174.07 %	332.68 %	293.25 %
Average	21.76 %	41.58 %	36.66 %
Maximum	27.85 %	43.73 %	38.35 %
Minimum	18.72 %	38.09 %	34.07 %
StDev	2.67 %	1.50 %	1.25 %
Avg/Max	0.78	0.95	0.96

In this second image, we can see that the running time is more equally done by all threads and all threads are scheduling and fork and join, and the time spent in synchronization is half the value we got in the previous strategy.

In conclusion, we can say that the second strategy, the tree strategy, is more efficient for scalability as a result of being who spend less time synchronizing and more time executing the tasks in more threads. This reduction in synchronization time is caused by the tasks thread created, this is because of the granularity of each implementation. If we see the number of tasks created, we can see that in the first strategy we generate fewer tasks than in the second, but as a consequence of the overheads, the time is less in the second strategy.

3.3 Optimization: Task granularity control: the cut–off mechanism

In order to control the number of tasks generated by the program, we have added a cut-off mechanism to control better the program. To do this implementation, we have added the parameter “d” in the multisort and merge functions, and we increment it in each recursive call. As a result, we can control this cutoff when -c parameter is specified.

```
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);
    }
}
```

If we see the code that we have implemented, we added the directive #pragma omp task final (d >= CUTOFF) to create a final task when the condition is done, as a result, this can be specified by the user using the new parameter CUTOFF.

In order to validate our code, we have executed this program with 1 and 8 threads to see the results and time required to execute the program in this new version.

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

```
********************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=50
Number of threads in OpenMP:          OMP_NUM_THREADS=1
********************************************************************************
Initialization time in seconds: 0.681650
Multisort execution time: 5.276805
Check sorted data execution time: 0.011379
Multisort program finished
********************************************************************************
```

sbatch ./submit-omp.sh multisort-omp 8

```
********************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=50
Number of threads in OpenMP:          OMP_NUM_THREADS=8
********************************************************************************
Initialization time in seconds: 0.681637
Multisort execution time: 0.752580
Check sorted data execution time: 0.012272
Multisort program finished
********************************************************************************
```

If we analyze the results, we can see that the program is correct and is using by default a CUTOFF = 50.

If we try to reduce the CUTOFF to 0, and execute the submit-strong-extrae.sh scrip, we obtain the following tables:

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.16	0.09	0.06	0.07	0.07	0.07	0.07	0.07	0.07
Speedup	1.00	1.69	2.45	2.33	2.29	2.26	2.26	2.27	2.26
Efficiency	1.00	0.84	0.61	0.39	0.29	0.23	0.19	0.16	0.14

Table 1: Analysis done on Wed Nov 16 09:20:04 AM CET 2022, par2118

Overview of the Efficiency metrics in parallel fraction, ϕ=85.43%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	99.96%	94.28%	81.59%	51.65%	38.02%	30.23%	25.14%	21.39%	18.78%
Parallelization strategy efficiency	99.96%	95.14%	83.86%	56.00%	41.82%	33.65%	28.00%	23.87%	20.95%
Load balancing	100.00%	95.25%	93.18%	56.72%	46.37%	36.97%	33.04%	31.50%	24.80%
In execution efficiency	99.96%	99.89%	90.00%	98.73%	90.18%	91.02%	84.74%	75.76%	84.47%
Scalability for computation tasks	100.00%	99.09%	97.29%	92.24%	90.91%	89.85%	89.76%	89.62%	89.65%
IPC scalability	100.00%	99.23%	99.03%	99.17%	98.71%	99.23%	99.11%	98.95%	99.00%
Instruction scalability	100.00%	100.00%	99.99%	99.99%	99.98%	99.98%	99.98%	99.97%	99.97%
Frequency scalability	100.00%	99.86%	98.25%	93.02%	92.12%	90.57%	90.58%	90.59%	90.59%

Table 2: Analysis done on Wed Nov 16 09:20:04 AM CET 2022, par2118

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0
LB (number of explicit tasks executed)	1.0	0.88	0.88	0.47	0.58	0.7	0.7	1.0	0.58
LB (time executing explicit tasks)	1.0	0.95	0.93	0.68	0.62	0.74	0.79	0.63	0.66
Time per explicit task (average us)	18905.59	19078.26	19430.42	20494.39	20792.16	21032.21	21050.09	21080.23	21063.36
Overhead per explicit task (synch %)	0.01	5.06	19.19	78.53	139.1	197.21	257.19	319.18	377.63
Overhead per explicit task (sched %)	0.02	0.02	0.03	0.04	0.04	0.04	0.05	0.06	0.08
Number of taskwait/taskgroup (total)	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0

Table 3: Analysis done on Wed Nov 16 09:20:04 AM CET 2022, par2118

By the other hand, we also executed the program with a CUTOFF of 1, to see the differences in the number of tasks generated.

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.16	0.09	0.06	0.05	0.05	0.05	0.05	0.05	0.05
Speedup	1.00	1.75	2.57	2.89	2.89	3.23	3.23	3.27	3.23
Efficiency	1.00	0.87	0.64	0.48	0.36	0.32	0.27	0.23	0.20

Table 1: Analysis done on Wed Nov 16 09:27:40 AM CET 2022, par2118

Overview of the Efficiency metrics in parallel fraction, ϕ=85.15%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	99.94%	99.17%	87.22%	73.92%	55.46%	56.55%	46.83%	40.45%	34.93%
Parallelization strategy efficiency	99.94%	99.66%	90.03%	80.14%	60.78%	63.32%	52.50%	45.37%	39.31%
Load balancing	100.00%	99.89%	90.67%	92.68%	76.98%	81.61%	70.24%	53.08%	62.99%
In execution efficiency	99.94%	99.77%	99.30%	86.48%	78.95%	77.59%	74.74%	85.49%	62.41%
Scalability for computation tasks	100.00%	99.52%	96.87%	92.23%	91.25%	89.31%	89.20%	89.15%	88.86%
IPC scalability	100.00%	99.49%	98.73%	98.68%	98.37%	98.57%	98.40%	98.35%	98.02%
Instruction scalability	100.00%	100.01%	100.00%	100.00%	100.00%	99.99%	99.99%	99.99%	99.98%
Frequency scalability	100.00%	100.01%	98.11%	93.47%	92.77%	90.61%	90.66%	90.66%	90.67%

Table 2: Analysis done on Wed Nov 16 09:27:40 AM CET 2022, par2118

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	41.0	41.0	41.0	41.0	41.0	41.0	41.0	41.0	41.0
LB (number of explicit tasks executed)	1.0	0.98	0.85	0.68	0.85	0.68	0.68	0.59	0.51
LB (time executing explicit tasks)	1.0	1.0	0.94	0.91	0.75	0.78	0.64	0.58	0.57
Time per explicit task (average us)	3233.05	3251.12	3465.07	3684.93	3908.13	3760.97	3813.68	3990.27	4080.08
Overhead per explicit task (synch %)	0.02	0.25	10.46	23.35	58.18	55.41	85.67	109.16	137.39
Overhead per explicit task (sched %)	0.04	0.08	0.18	0.19	0.3	0.3	0.28	0.23	0.23
Number of taskwait/taskgroup (total)	18.0	18.0	18.0	18.0	18.0	18.0	18.0	18.0	18.0

Table 3: Analysis done on Wed Nov 16 09:27:40 AM CET 2022, par2118

If we compare the number of tasks created in both cases, we can see that if we have a CUTOFF = 0, we generate 7 tasks, but if we have a CUTOFF =1, we generate 41 tasks. This is the result that we were expecting because if we have a bigger CUTOFF, we will have more tasks, because it will take more levels to arrive to the last part of the program where new tasks are generated.

Now, we have executed the program with a CUTOFF = 8 and we obtained the following tables:

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.24	0.22	0.17	0.15	0.14	0.14	0.14	0.14	0.14
Speedup	1.00	1.11	1.45	1.59	1.73	1.68	1.72	1.69	1.72
Efficiency	1.00	0.56	0.36	0.27	0.22	0.17	0.14	0.12	0.11

Table 1: Analysis done on Wed Nov 16 09:36:57 AM CET 2022, par2118

Overview of the Efficiency metrics in parallel fraction, ϕ=90.73%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	87.91%	49.81%	33.48%	25.14%	20.93%	16.21%	13.90%	11.63%	10.44%
Parallelization strategy efficiency	87.91%	58.97%	46.33%	39.15%	31.04%	24.71%	20.72%	17.53%	15.52%
Load balancing	100.00%	96.52%	98.70%	96.16%	93.91%	96.28%	91.10%	90.73%	93.90%
In execution efficiency	87.91%	61.10%	46.94%	40.71%	33.05%	25.66%	22.75%	19.32%	16.53%
Scalability for computation tasks	100.00%	84.46%	72.25%	64.21%	67.43%	65.60%	67.05%	66.38%	67.24%
IPC scalability	100.00%	72.57%	62.95%	59.16%	61.70%	61.97%	63.96%	63.49%	64.43%
Instruction scalability	100.00%	114.00%	113.85%	113.90%	113.81%	113.80%	113.83%	113.80%	113.77%
Frequency scalability	100.00%	102.10%	100.81%	95.30%	96.03%	93.01%	92.10%	91.87%	91.73%

Table 2: Analysis done on Wed Nov 16 09:36:57 AM CET 2022, par2118

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	57173.0	57173.0	57173.0	57173.0	57173.0	57173.0	57173.0	57173.0	57173.0
LB (number of explicit tasks executed)	1.0	1.0	0.98	0.96	0.98	0.96	0.98	0.98	0.98
LB (time executing explicit tasks)	1.0	0.98	0.98	0.98	0.98	0.98	0.99	0.99	0.98
Time per explicit task (average us)	2.86	5.11	7.21	9.47	11.32	14.62	17.04	20.23	22.5
Overhead per explicit task (synch %)	7.27	33.82	44.41	49.63	54.82	58.01	59.99	62.11	63.61
Overhead per explicit task (sched %)	8.98	20.64	30.76	36.79	43.55	49.37	53.11	56.31	57.99
Number of taskwait/taskgroup (total)	27904.0	27904.0	27904.0	27904.0	27904.0	27904.0	27904.0	27904.0	27904.0

Table 3: Analysis done on Wed Nov 16 09:36:57 AM CET 2022, par2118

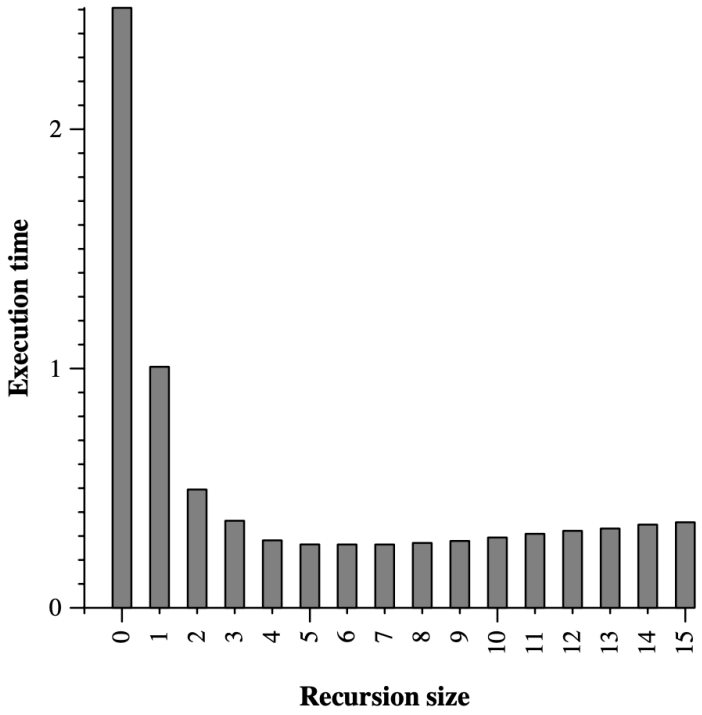
If we take a look at the tables, we can see that the execution time with 8 threads and more is always the same, as a result in this case is not necessary to have more threads executing the program because no better time will be got.

In table 3, we can see that the number of tasks generated do not depend on the number of threads that are executing the program, but with more threads we have an increment of the overhead produced per explicit tasks with more threads. Despite this fact, this number when arrives to 8 threads increment slower than in the firsts columns.

Finally, in the second table we can see that the load balancing is very good with all number of processors because is always above 90%, but the in execution efficiency is reduced when

more threads are added to the execution. This result was what we expected due to if we have more threads and the time is not reduced, the efficiency will be worst in each thread.

If we execute the script submit-cutoff-omp.sh and open the “ps” archive that generates, we can see the following result:



par2118
Average elapsed execution time (multisort only)
Wed Nov 16 09:53:28 AM CET 2022

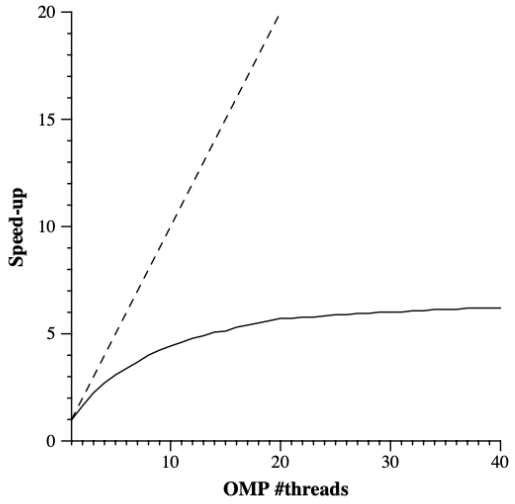
In this graph, we can see that the execution time is reduced a lot when the number of threads start to increase, but when we arrive to 6 threads it starts to increment slowly, this can be caused by the overheads of synchronization of all threads.

After all the tests that we have done during the previous executions, we can conclude that the best number of threads is 8, because we do not have a big problem with the overheads, and the tasks can be equally distributed among the nodes. In relation to the cutoff we have observed that with CUTOFF = 1, was the lowest execution time as a result of the low overheads of the among of tasks generated. The tables can be seen in the respective point of the laboratory timeline.

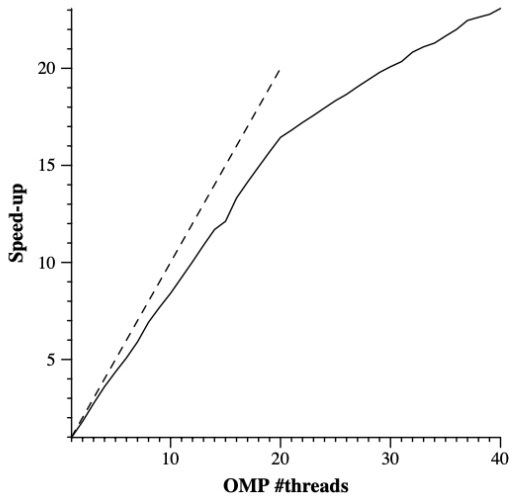
The number of tasks generated by the CUTOFF = 0, and the CUTOFF = 1 is due to in the first case because in the multisort function we have 7 pragma statement, and as a result of being CUTOFF = 0 and “d” starts with the value 0, we only have one iteration where the tasks are generated. In the other hand, with a CUTOFF = 1, we have two levels generating tasks, as a result we have 41 tasks.

3.4 Optional 1

Now, we are going to change the maximum number of threads to 40 threads and the cutoff value with the scrip “submit-strong-omp.sh” in order to see what happens with the execution time and all the other factors.



par2118
Speed-up wrt sequential time (complete application)
Generated by par2118 on Wed Nov 16 10:09:30 AM CET 2022



par2118
Speed-up wrt sequential time (multisort funtion only)
Generated by par2118 on Wed Nov 16 10:09:30 AM CET 2022

If we take a look at the graphs, we can see that despite the boada having 20 cores, each one has 2 threads, as a result can execute with a maximum of 40 tasks simultaneously, as a conclusion the time executing the program can be better with more threads, but is not incremented as fast as when the threads were less than twenty due to the number of cores.

4. Shared-memory parallelisation with OpenMP task using dependencies

In this last session, we have implemented a modified code of the cutoff strategy developed before in order to experiment with the dependencies. We have avoided using taskwait or taskgroup as a result we used the depend(in:) and depend(out:) to replace the previous statements.

```
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
        #pragma omp task
        merge(n, left, right, result, start, length/2, d+1);
        #pragma omp task
        merge(n, left, right, result, start + length/2, length/2, d+1);
        #pragma omp taskwait
    }
}

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

        #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, d+1);
        #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, d+1);

        #pragma omp task depend(in: tmp[0], tmp[n/2L])
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, d+1);
        #pragma omp taskwait
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

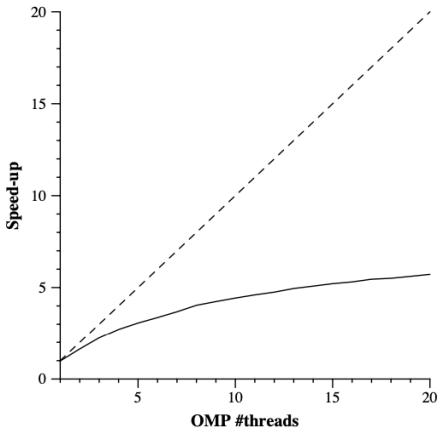
In the following image we can see the code that we have modified.

In order to see if the code is correct, we have executed this program, and we got the following result:

sbatch ./submit-omp.sh multisort-omp 8 16

```
multisort-omp_8_16_boada-12.times.txt
::::::::::::::
********************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=16
Number of threads in OpenMP:          OMP_NUM_THREADS=8
********************************************************************************************
Initialization time in seconds: 0.683138
Multisort execution time: 0.764642
Check sorted data execution time: 0.014862
Multisort program finished
********************************************************************************************
```

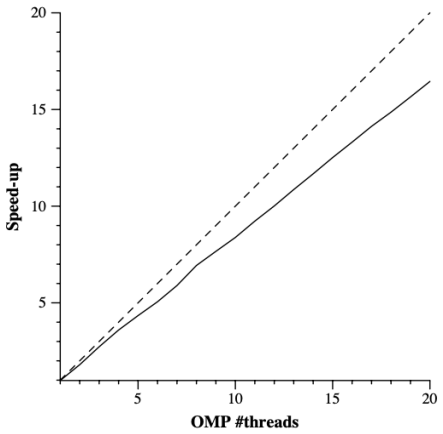
Next, we have executed the submit-strong-omp.sh scrip to compare the execution time of this version compared with the previous versions.



par2118
Speed-up wrt sequential time (complete application)
Generated by par2118 on Tue Nov 22 11:47:06 AM CET 2022

As we can see in the plots, they are similar to the tree version despite being so different in the execution. These differences came out because when we add a task depend clause, tree execution only has to wait to the indicated value and not all the code as before.

In relation to the programmability, this code is harder to write, because it is longer and more detailed, but it helps to know what is exactly waiting each time.



par2118
Speed-up wrt sequential time (multisort funtion only)
Generated by par2118 on Tue Nov 22 11:47:06 AM CET 2022

If we execute the submit-strong-extrae.sh we get the following tables:

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.32	0.32	0.25	0.24	0.23	0.23	0.24	0.23	0.24
Speedup	1.00	1.00	1.28	1.33	1.39	1.39	1.34	1.40	1.36
Efficiency	1.00	0.50	0.32	0.22	0.17	0.14	0.11	0.10	0.08

Table 1: Analysis done on Tue Nov 22 04:00:56 PM CET 2022, par2118

Overview of the Efficiency metrics in parallel fraction, ϕ=93.03%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	76.66%	38.12%	25.04%	17.59%	13.89%	11.11%	8.91%	8.00%	6.74%
Parallelization strategy efficiency	76.66%	50.92%	38.84%	30.76%	23.48%	19.00%	15.05%	13.31%	11.35%
Load balancing	100.00%	96.04%	96.32%	95.88%	92.32%	92.78%	91.96%	92.91%	91.97%
In execution efficiency	76.66%	53.02%	40.33%	32.09%	25.43%	20.48%	16.37%	14.33%	12.34%
Scalability for computation tasks	100.00%	74.87%	64.47%	57.18%	59.17%	58.47%	59.18%	60.09%	59.40%
IPC scalability	100.00%	63.20%	55.93%	51.95%	53.22%	54.52%	55.99%	57.12%	56.38%
Instruction scalability	100.00%	118.46%	118.55%	118.50%	118.58%	118.51%	118.55%	118.56%	118.61%
Frequency scalability	100.00%	100.01%	97.24%	92.88%	93.76%	90.49%	89.17%	88.73%	88.83%

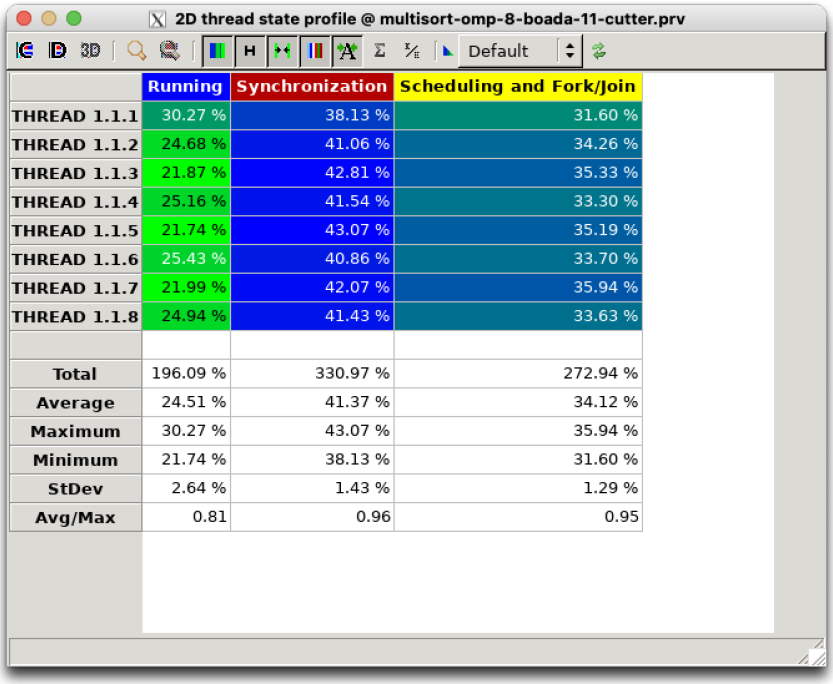
Table 2: Analysis done on Tue Nov 22 04:00:56 PM CET 2022, par2118

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0
LB (number of explicit tasks executed)	1.0	0.96	0.96	0.96	0.99	0.97	0.97	0.97	0.98
LB (time executing explicit tasks)	1.0	0.98	0.99	1.0	0.99	0.99	0.99	0.99	0.99
Time per explicit task (average us)	1.85	4.47	6.62	9.4	11.98	15.07	18.9	21.08	24.97
Overhead per explicit task (synch %)	9.24	40.81	49.48	54.81	58.08	59.84	60.88	61.48	62.85
Overhead per explicit task (sched %)	13.04	25.57	35.55	41.76	47.9	51.7	55.42	57.06	58.54
Number of taskwait/taskgroup (total)	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0

Table 3: Analysis done on Tue Nov 22 04:00:56 PM CET 2022, par2118

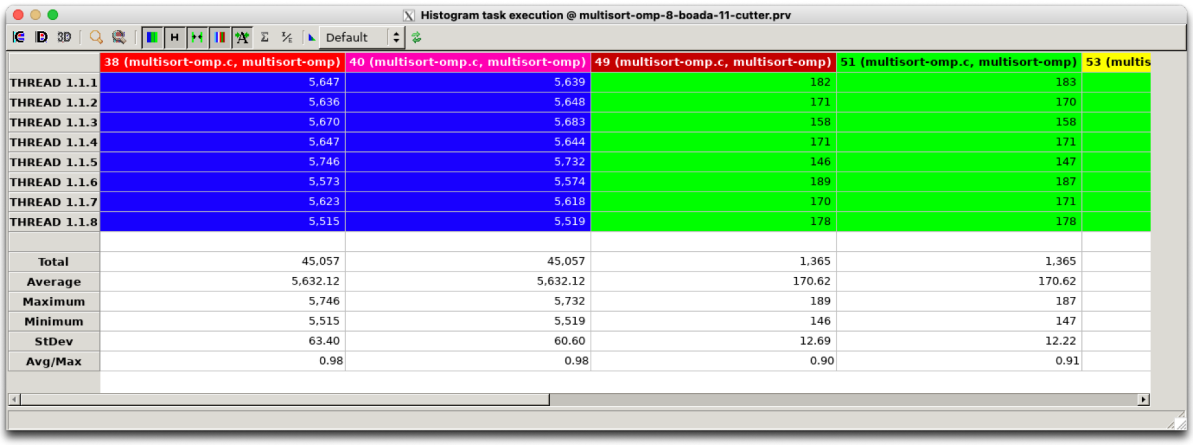
If we compare the execution time, we can see that now we have a slower program compared with the version with cutoff. This could be caused as a result of the rise of the number of explicit tasks executed and the number of taskwait/taskgroup. With more tasks, the execution time is higher.

If we execute the wxparaver, with 8 threads, we can see the following results:

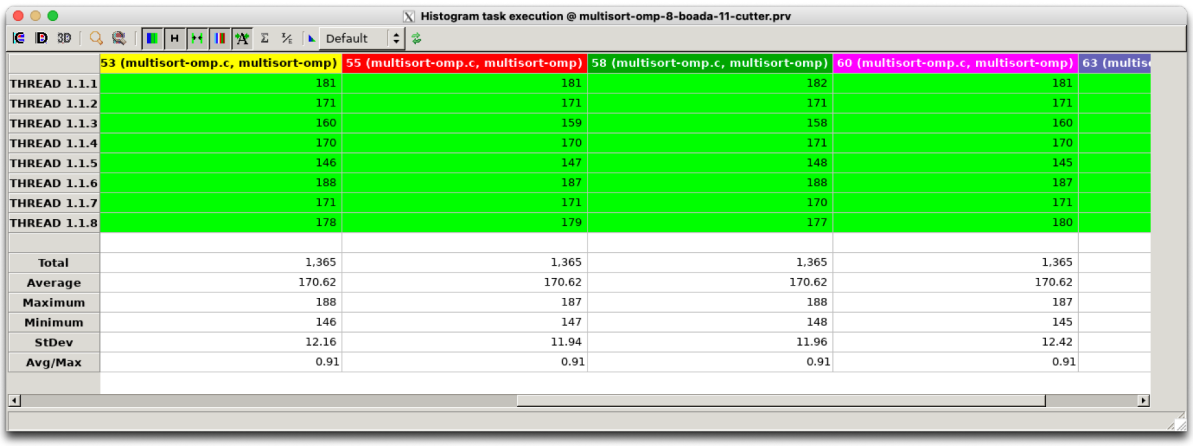


	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	30.27 %	38.13 %	31.60 %
THREAD 1.1.2	24.68 %	41.06 %	34.26 %
THREAD 1.1.3	21.87 %	42.81 %	35.33 %
THREAD 1.1.4	25.16 %	41.54 %	33.30 %
THREAD 1.1.5	21.74 %	43.07 %	35.19 %
THREAD 1.1.6	25.43 %	40.86 %	33.70 %
THREAD 1.1.7	21.99 %	42.07 %	35.94 %
THREAD 1.1.8	24.94 %	41.43 %	33.63 %
Total	196.09 %	330.97 %	272.94 %
Average	24.51 %	41.37 %	34.12 %
Maximum	30.27 %	43.07 %	35.94 %
Minimum	21.74 %	38.13 %	31.60 %
StDev	2.64 %	1.43 %	1.29 %
Avg/Max	0.81	0.96	0.95

If we compare these images, with the other versions of the program, we can see that we have similar characteristics as the initial execution of this laboratory, in terms of time spend running, synchronization and scheduling.



	38 (multisort-omp.c, multisort-omp)	40 (multisort-omp.c, multisort-omp)	49 (multisort-omp.c, multisort-omp)	51 (multisort-omp.c, multisort-omp)	53 (multis
THREAD 1.1.1	5,647	5,639	182	183	
THREAD 1.1.2	5,636	5,648	171	170	
THREAD 1.1.3	5,670	5,683	158	158	
THREAD 1.1.4	5,647	5,644	171	171	
THREAD 1.1.5	5,746	5,732	146	147	
THREAD 1.1.6	5,573	5,574	189	187	
THREAD 1.1.7	5,623	5,618	170	171	
THREAD 1.1.8	5,515	5,519	178	178	
Total	45,057	45,057	1,365	1,365	
Average	5,632.12	5,632.12	170.62	170.62	
Maximum	5,746	5,732	189	187	
Minimum	5,515	5,519	146	147	
StDev	63.40	60.60	12.69	12.22	
Avg/Max	0.98	0.98	0.90	0.91	



	53 (multisort-omp.c, multisort-omp)	55 (multisort-omp.c, multisort-omp)	58 (multisort-omp.c, multisort-omp)	60 (multisort-omp.c, multisort-omp)	63 (multis
THREAD 1.1.1	181	181	182	181	
THREAD 1.1.2	171	171	171	171	
THREAD 1.1.3	160	159	158	160	
THREAD 1.1.4	170	170	171	170	
THREAD 1.1.5	146	147	148	145	
THREAD 1.1.6	188	187	188	187	
THREAD 1.1.7	171	171	170	171	
THREAD 1.1.8	178	179	177	180	
Total	1,365	1,365	1,365	1,365	
Average	170.62	170.62	170.62	170.62	
Maximum	188	187	188	187	
Minimum	146	147	148	145	
StDev	12.16	11.94	11.96	12.42	
Avg/Max	0.91	0.91	0.91	0.91	

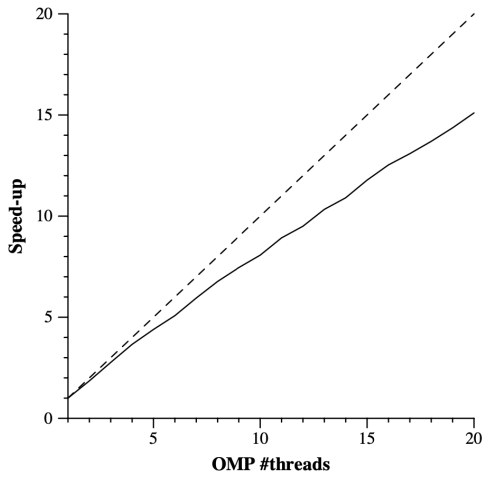
As we can see in the previous tables, the program is working correctly parallelized because each thread is executing the same amount of tasks as we can see in each column.

4.1 Optional 2

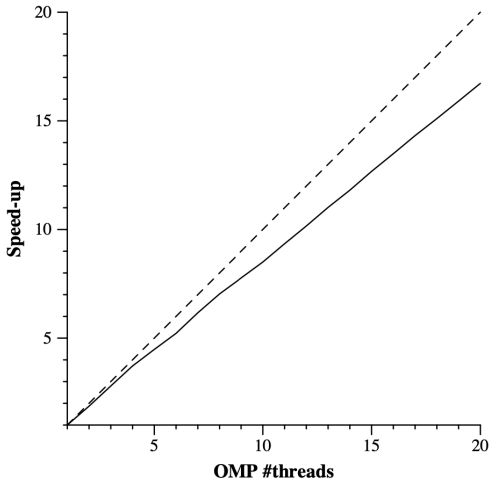
In this final optional, we are going to parallelize the functions where data and tmp vectors are initialized. This code has been modified like the following image:

```
static void initialize(long length, T data[length]) {
    long i;
        #pragma omp parallel for
    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 parallel for
    for (i = 0; i < length; i++) {
        data[i] = 0;
    }
}
```

If we execute the script, submit-strong-omp.sh script, we obtain the following plots:



par2118
Speed-up wrt sequential time (complete application)
Generated by par2118 on Tue Nov 22 04:37:50 PM CET 2022



par2118
Speed-up wrt sequential time (multisort funtion only)
Generated by par2118 on Tue Nov 22 04:37:50 PM CET 2022

If we see the plots and compare with the previous versions, we can see that the performance is better. This is because we have more functions parallelized.

If we execute the submit-strong-extrae.sh script, we can see the following results:

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.33	0.31	0.25	0.22	0.22	0.21	0.22	0.21	0.22
Speedup	1.00	1.05	1.31	1.48	1.47	1.51	1.47	1.54	1.46
Efficiency	1.00	0.53	0.33	0.25	0.18	0.15	0.12	0.11	0.09

Table 1: Analysis done on Tue Nov 22 04:46:56 PM CET 2022, par2118

Overview of the Efficiency metrics in parallel fraction, ϕ=99.87%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	77.58%	40.87%	25.45%	19.14%	14.23%	11.76%	9.50%	8.54%	7.11%
Parallelization strategy efficiency	77.58%	52.67%	38.84%	31.83%	23.74%	19.87%	16.07%	13.89%	11.76%
Load balancing	100.00%	96.72%	95.98%	96.69%	94.24%	93.31%	92.94%	92.81%	93.33%
In execution efficiency	77.58%	54.45%	40.47%	32.92%	25.19%	21.29%	17.29%	14.97%	12.61%
Scalability for computation tasks	100.00%	77.61%	65.52%	60.14%	59.94%	59.20%	59.10%	61.46%	60.43%
IPC scalability	100.00%	64.41%	55.26%	53.93%	53.36%	54.75%	55.22%	57.80%	57.03%
Instruction scalability	100.00%	118.28%	118.43%	118.38%	118.31%	118.25%	117.85%	118.22%	118.17%
Frequency scalability	100.00%	101.87%	100.11%	94.20%	94.96%	91.44%	90.83%	89.93%	89.67%

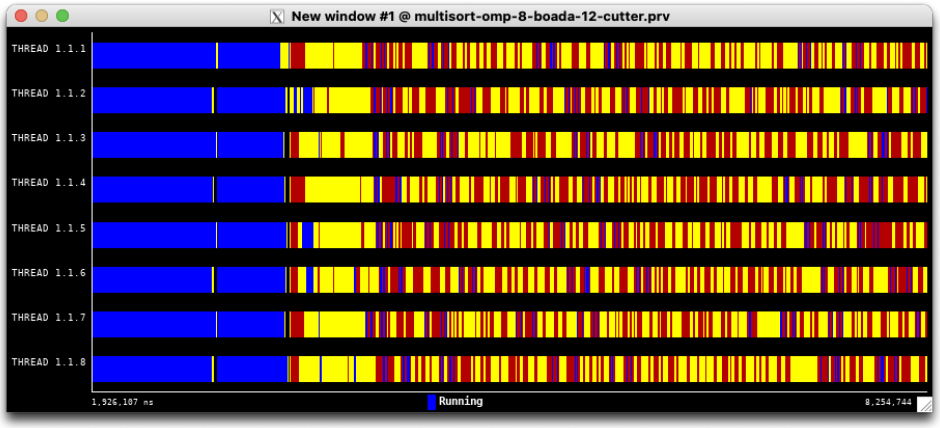
Table 2: Analysis done on Tue Nov 22 04:46:56 PM CET 2022, par2118

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0
LB (number of explicit tasks executed)	1.0	0.97	0.97	0.96	0.97	0.98	0.96	0.96	0.98
LB (time executing explicit tasks)	1.0	0.98	0.99	0.99	0.99	1.0	0.99	0.99	0.99
Time per explicit task (average us)	2.14	4.39	6.96	9.31	12.67	15.4	19.19	21.35	25.68
Overhead per explicit task (synch %)	8.34	41.26	50.67	54.39	58.08	59.59	60.99	62.25	63.2
Overhead per explicit task (sched %)	11.42	25.28	36.58	42.26	48.79	52.07	55.33	57.11	58.9
Number of taskwait/taskgroup (total)	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0

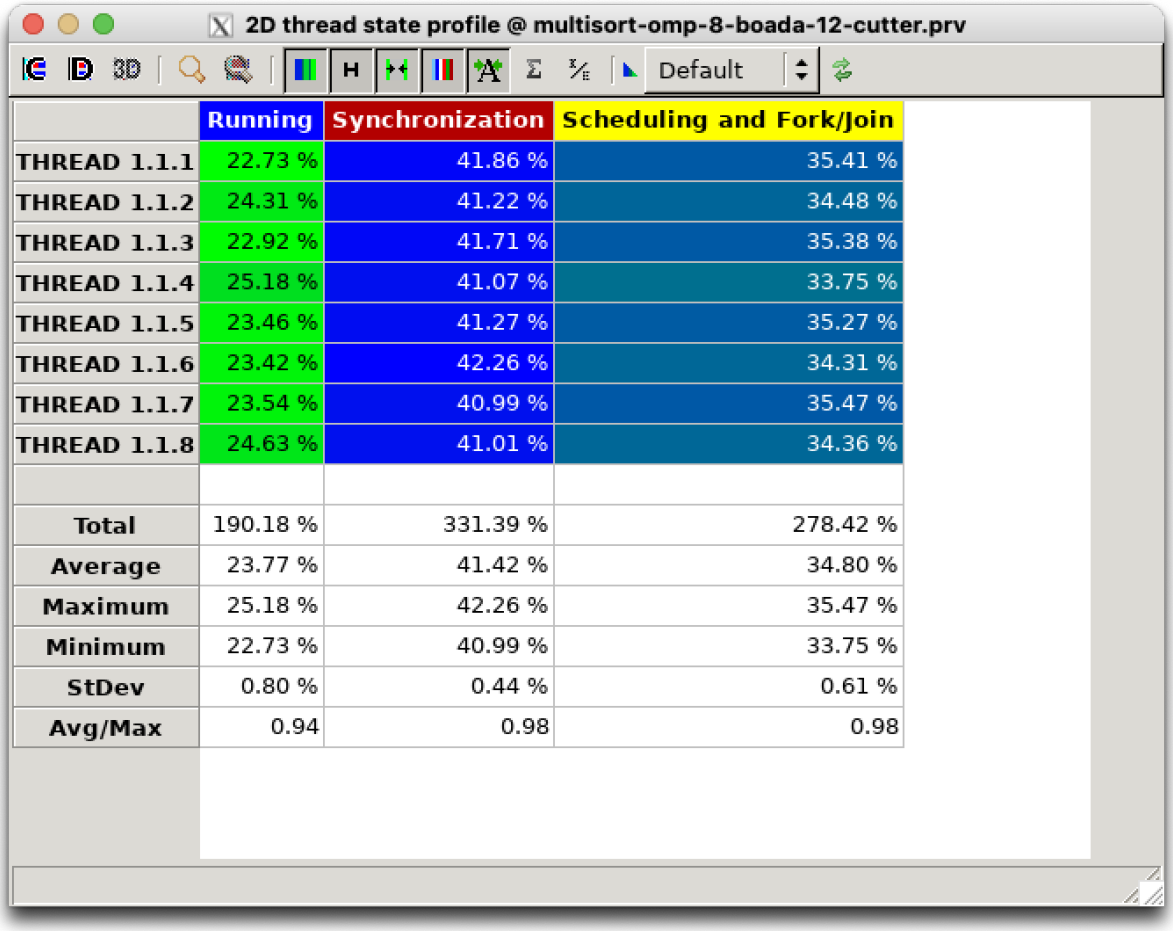
Table 3: Analysis done on Tue Nov 22 04:46:56 PM CET 2022, par2118

In the tables, we can see that the parallel fraction has been increased in comparison with the other versions. In the other parts, we get a similar results compared with the previous version.

If we execute the wxparaver, we get the following image:



In the image we can see that now at the beginning all threads are running, instead of waiting like was doing in the other versions, as a result we get a better performance in the initialization of the vectors and a better optimization of the threads.



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	22.73 %	41.86 %	35.41 %
THREAD 1.1.2	24.31 %	41.22 %	34.48 %
THREAD 1.1.3	22.92 %	41.71 %	35.38 %
THREAD 1.1.4	25.18 %	41.07 %	33.75 %
THREAD 1.1.5	23.46 %	41.27 %	35.27 %
THREAD 1.1.6	23.42 %	42.26 %	34.31 %
THREAD 1.1.7	23.54 %	40.99 %	35.47 %
THREAD 1.1.8	24.63 %	41.01 %	34.36 %
Total	190.18 %	331.39 %	278.42 %
Average	23.77 %	41.42 %	34.80 %
Maximum	25.18 %	42.26 %	35.47 %
Minimum	22.73 %	40.99 %	33.75 %
StDev	0.80 %	0.44 %	0.61 %
Avg/Max	0.94	0.98	0.98

Now the running time is better distributed among the threads.

5. Conclusion

During this laboratory, we have understood several ways to do task decomposition, using taskwait/taskgroup, depend… We have learned to implement several strategies like tree or leaf or using a cut-off mechanism. And finally, we have used task dependencies to try to get a better way to implement the code.

In conclusion, in this lab sessions we have learned how to parallelize code with recursive functions, and check and change the efficiency in different implementations depending on each situation.