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

Natalia Dai
Daniel Ruiz Jiménez

Username: par2306
par2317
Date: 14/05/2022
Spring 2021-22

INDEX

1. Introduction 3

2. Task decomposition analysis for Mergesort 4-8

3. Parallelisation with OpenMP tasks 9-18

4. Using OpenMP task dependencies 19-25

5. Conclusion 26

INTRODUCTION

In this lab (divided into 3 sessions), we're going to try 2 different strategies about task decomposition in recursive programs. The first strategy is Leaf strategy, where tasks are created once the program reaches the non-recursive case (leaves). On the other hand, the second strategy is Tree strategy, where a task is created each time a recursive call takes place. We will take these strategies into practice with a divide-and-conquer algorithm, Mergesort.

The second day, we will parallelise the original multisort.c program, but without the use of Tareador. We're going to still use both strategies from the previous session, but now with OpenMP clauses. On top of that, we're going to use a strategy that prevents the excessive creation of tasks causing overheads to the Tree strategy, the cutoff method.

The third and last day, we will implement task dependencies of the Tree version with OpenMP clauses. Basing on the TDG, we will add "depend" clauses to express those dependencies. Further on, we will analyse its scalability to check if the performance of the program has improved.

1. Task decomposition analysis for Mergesort

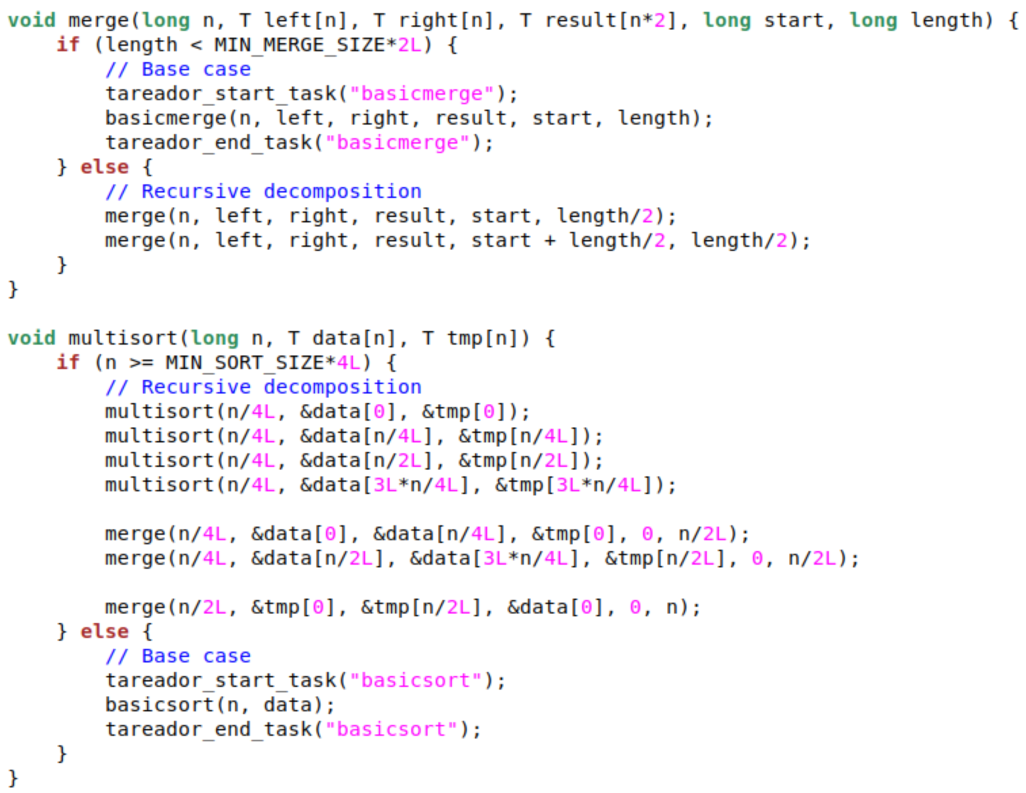
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024

```
*******************************************************************************
Initialization time in seconds: 0.857180
Multisort execution time: 6.308882
Check sorted data execution time: 0.015286
*******************************************************************************
```

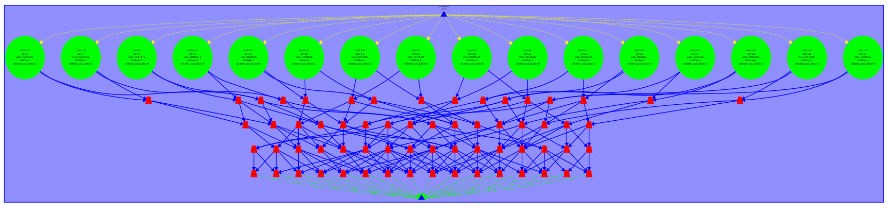
This is the output of the sequential version of multisort. We will leave it here so we can compare execution times with the following versions.

Leaf Strategy:

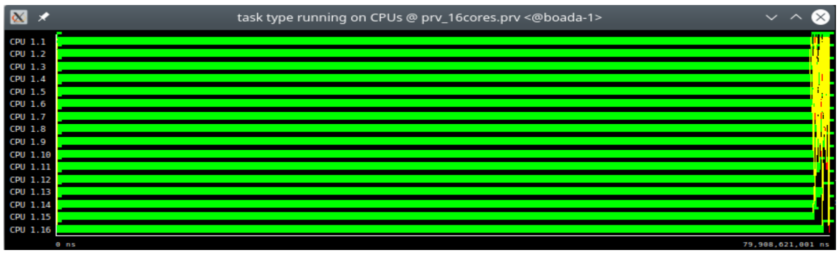
For this strategy, we'll add task creation and removal before and after each basicsort and basicmerge call. The code should look like this:

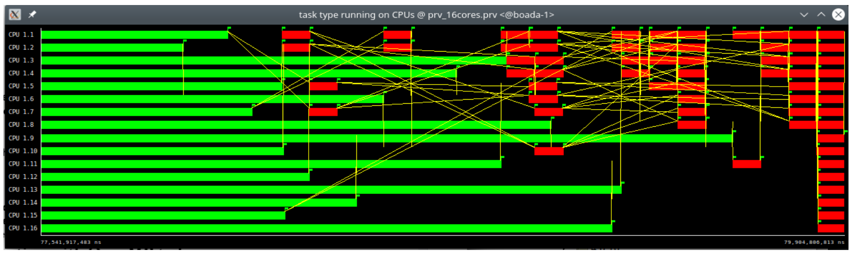


```c
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        tareador_start_task("basicmerge");
        basicmerge(n, left, right, result, start, length);
        tareador_end_task("basicmerge");
    } else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

void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        multisort(n/4L, &data[0], &tmp[0]);
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

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

        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
    } else {
        // Base case
        tareador_start_task("basicsort");
        basicsort(n, data);
        tareador_end_task("basicsort");
    }
}
```

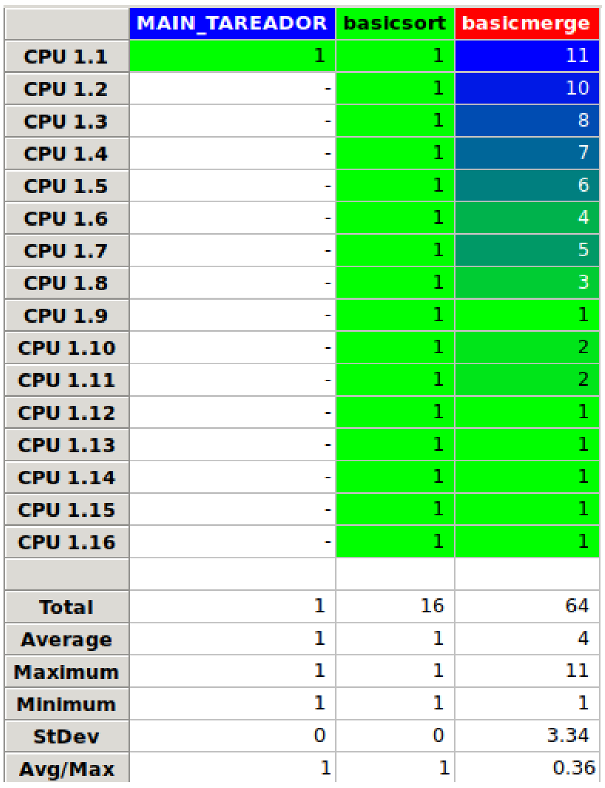


Figs 1 and 2: Leaf Strategy in Paraver

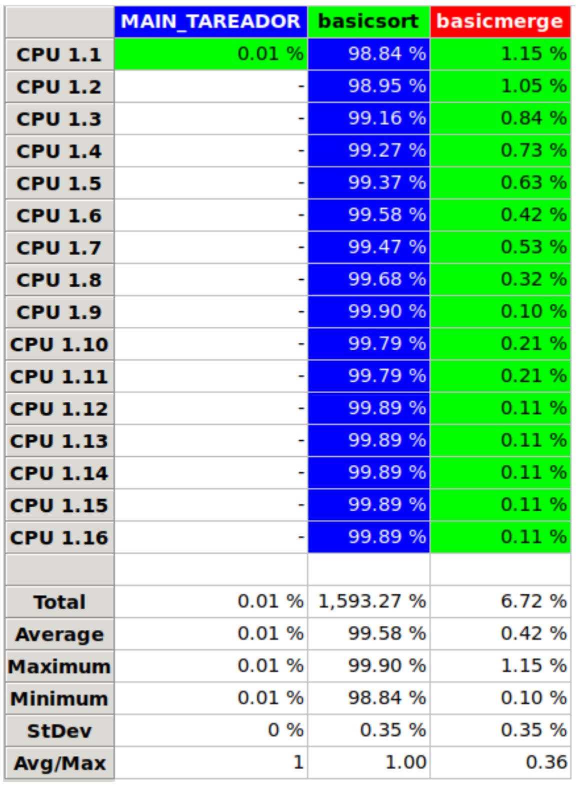




Figs 3 and 4: Leaf Strategy code in Tareador with 16 CPUs



	MAIN_TAREADOR	basicsort	basicmerge
CPU 1.1	1	1	11
CPU 1.2	-	1	10
CPU 1.3	-	1	8
CPU 1.4	-	1	7
CPU 1.5	-	1	6
CPU 1.6	-	1	4
CPU 1.7	-	1	5
CPU 1.8	-	1	3
CPU 1.9	-	1	1
CPU 1.10	-	1	2
CPU 1.11	-	1	2
CPU 1.12	-	1	1
CPU 1.13	-	1	1
CPU 1.14	-	1	1
CPU 1.15	-	1	1
CPU 1.16	-	1	1
Total	1	16	64
Average	1	1	4
Maximum	1	1	11
Minimum	1	1	1
StDev	0	0	3.34
Avg/Max	1	1	0.36



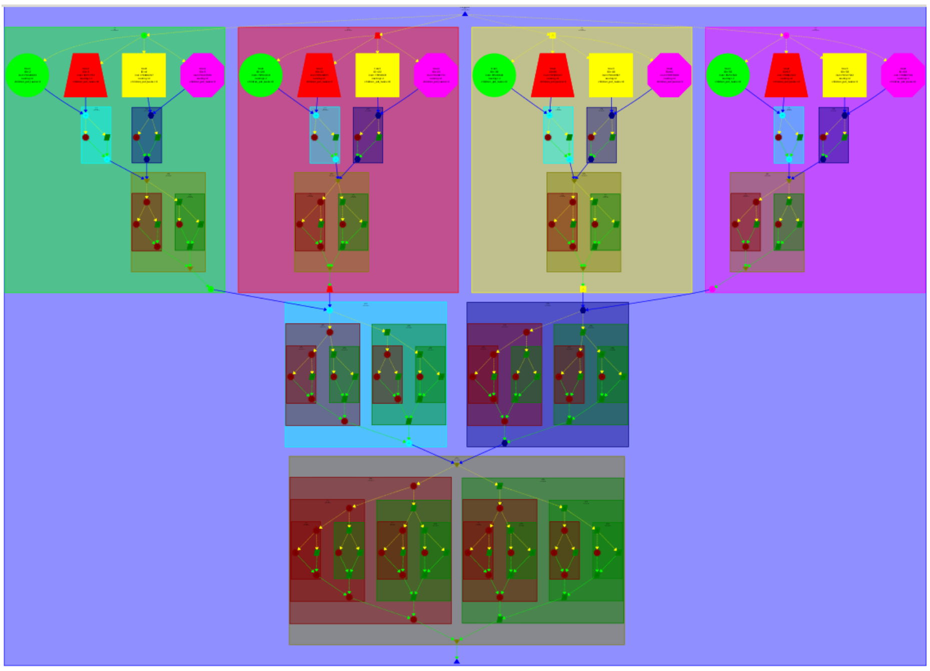
	MAIN_TAREADOR	basicsort	basicmerge
CPU 1.1	0.01 %	98.84 %	1.15 %
CPU 1.2	-	98.95 %	1.05 %
CPU 1.3	-	99.16 %	0.84 %
CPU 1.4	-	99.27 %	0.73 %
CPU 1.5	-	99.37 %	0.63 %
CPU 1.6	-	99.58 %	0.42 %
CPU 1.7	-	99.47 %	0.53 %
CPU 1.8	-	99.68 %	0.32 %
CPU 1.9	-	99.90 %	0.10 %
CPU 1.10	-	99.79 %	0.21 %
CPU 1.11	-	99.79 %	0.21 %
CPU 1.12	-	99.89 %	0.11 %
CPU 1.13	-	99.89 %	0.11 %
CPU 1.14	-	99.89 %	0.11 %
CPU 1.15	-	99.89 %	0.11 %
CPU 1.16	-	99.89 %	0.11 %
Total	0.01 %	1,593.27 %	6.72 %
Average	0.01 %	99.58 %	0.42 %
Maximum	0.01 %	99.90 %	1.15 %
Minimum	0.01 %	98.84 %	0.10 %
StDev	0 %	0.35 %	0.35 %
Avg/Max	1	1.00	0.36

Figs 5 and 6: Histograms #bursts and %time

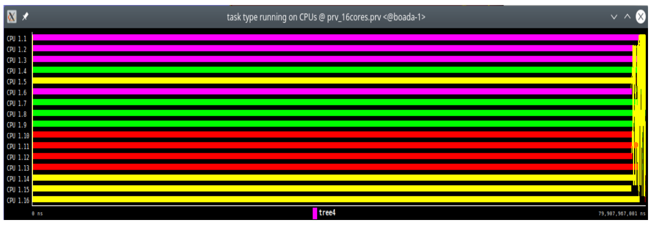
Tree strategy:

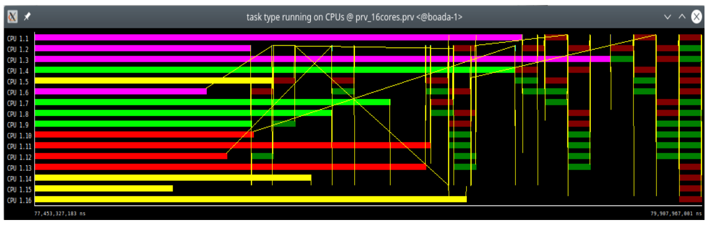
In the tree strategy, we will add a task start and end function before and after every multisort and merge call. The code looks like this afterwards:

```
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("tree8");
        merge(n, left, right, result, start, length/2);
        tareador_end_task("tree8");

        tareador_start_task("tree9");
        merge(n, left, right, result, start + length/2, length/2);
        tareador_end_task("tree9");
    }
}

void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        tareador_start_task("tree1");
        multisort(n/4L, &data[0], &tmp[0]);
        tareador_end_task("tree1");

        tareador_start_task("tree2");
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        tareador_end_task("tree2");

        tareador_start_task("tree3");
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        tareador_end_task("tree3");

        tareador_start_task("tree4");
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
        tareador_end_task("tree4");

        tareador_start_task("tree5");
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        tareador_end_task("tree5");

        tareador_start_task("tree6");
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        tareador_end_task("tree6");

        tareador_start_task("tree7");
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        tareador_end_task("tree7");
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

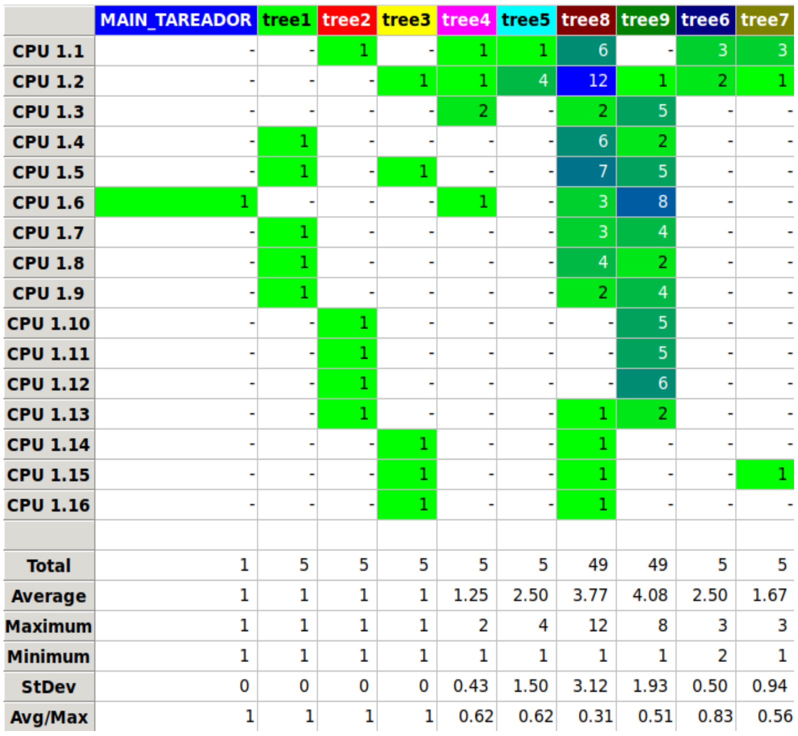


Figs 7 and 8: Tree Strategy in Tareador



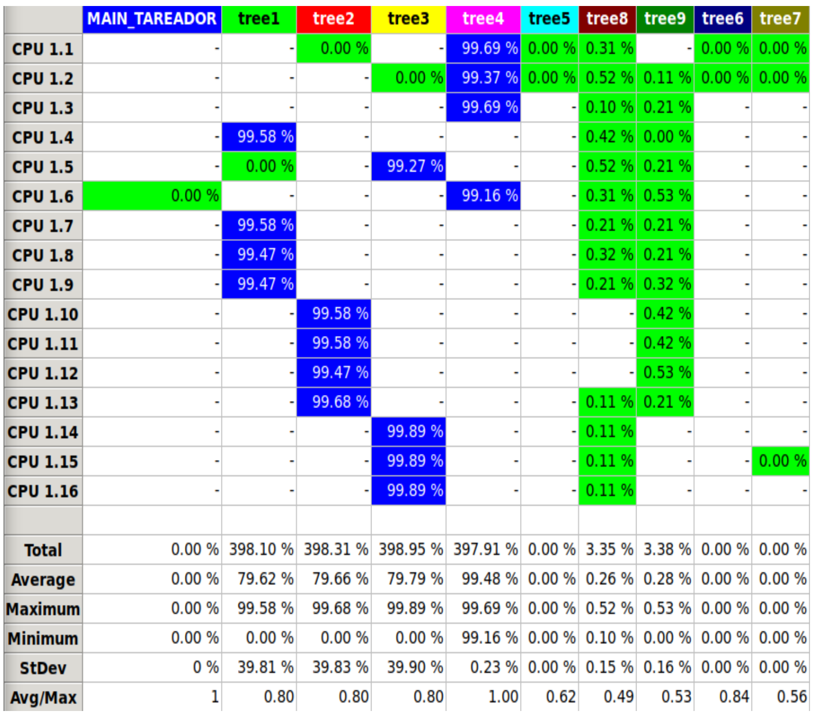


Figs 9 and 10: Tree Strategy code in Paraver with 16 CPUs



	MAIN_TAREADOR	tree1	tree2	tree3	tree4	tree5	tree8	tree9	tree6	tree7
CPU 1.1	-	-	1	-	1	1	6	-	3	3
CPU 1.2	-	-	-	1	1	4	12	1	2	1
CPU 1.3	-	-	-	-	2	-	2	5	-	-
CPU 1.4	-	1	-	-	-	-	6	2	-	-
CPU 1.5	-	1	-	1	-	-	7	5	-	-
CPU 1.6	1	-	-	-	1	-	3	8	-	-
CPU 1.7	-	1	-	-	-	-	3	4	-	-
CPU 1.8	-	1	-	-	-	-	4	2	-	-
CPU 1.9	-	1	-	-	-	-	2	4	-	-
CPU 1.10	-	-	1	-	-	-	-	5	-	-
CPU 1.11	-	-	1	-	-	-	-	5	-	-
CPU 1.12	-	-	1	-	-	-	-	6	-	-
CPU 1.13	-	-	1	-	-	-	1	2	-	-
CPU 1.14	-	-	-	1	-	-	1	-	-	-
CPU 1.15	-	-	-	1	-	-	1	-	-	1
CPU 1.16	-	-	-	1	-	-	1	-	-	-
Total	1	5	5	5	5	5	49	49	5	5
Average	1	1	1	1	1.25	2.50	3.77	4.08	2.50	1.67
Maximum	1	1	1	1	2	4	12	8	3	3
Minimum	1	1	1	1	1	1	1	1	2	1
StDev	0	0	0	0	0.43	1.50	3.12	1.93	0.50	0.94
Avg/Max	1	1	1	1	0.62	0.62	0.31	0.51	0.83	0.56

Fig 11: Histogram #bursts



	MAIN_TAREADOR	tree1	tree2	tree3	tree4	tree5	tree8	tree9	tree6	tree7
CPU 1.1	-	-	0.00 %	-	99.69 %	0.00 %	0.31 %	-	0.00 %	0.00 %
CPU 1.2	-	-	-	0.00 %	99.37 %	0.00 %	0.52 %	0.11 %	0.00 %	0.00 %
CPU 1.3	-	-	-	-	99.69 %	-	0.10 %	0.21 %	-	-
CPU 1.4	-	99.58 %	-	-	-	-	0.42 %	0.00 %	-	-
CPU 1.5	-	0.00 %	-	99.27 %	-	-	0.52 %	0.21 %	-	-
CPU 1.6	0.00 %	-	-	-	99.16 %	-	0.31 %	0.53 %	-	-
CPU 1.7	-	99.58 %	-	-	-	-	0.21 %	0.21 %	-	-
CPU 1.8	-	99.47 %	-	-	-	-	0.32 %	0.21 %	-	-
CPU 1.9	-	99.47 %	-	-	-	-	0.21 %	0.32 %	-	-
CPU 1.10	-	-	99.58 %	-	-	-	-	0.42 %	-	-
CPU 1.11	-	-	99.58 %	-	-	-	-	0.42 %	-	-
CPU 1.12	-	-	99.47 %	-	-	-	-	0.53 %	-	-
CPU 1.13	-	-	99.68 %	-	-	-	0.11 %	0.21 %	-	-
CPU 1.14	-	-	-	99.89 %	-	-	0.11 %	-	-	-
CPU 1.15	-	-	-	99.89 %	-	-	0.11 %	-	-	0.00 %
CPU 1.16	-	-	-	99.89 %	-	-	0.11 %	-	-	-
Total	0.00 %	398.10 %	398.31 %	398.95 %	397.91 %	0.00 %	3.35 %	3.38 %	0.00 %	0.00 %
Average	0.00 %	79.62 %	79.66 %	79.79 %	99.48 %	0.00 %	0.26 %	0.28 %	0.00 %	0.00 %
Maximum	0.00 %	99.58 %	99.68 %	99.89 %	99.69 %	0.00 %	0.52 %	0.53 %	0.00 %	0.00 %
Minimum	0.00 %	0.00 %	0.00 %	0.00 %	99.16 %	0.00 %	0.10 %	0.00 %	0.00 %	0.00 %
StDev	0 %	39.81 %	39.83 %	39.90 %	0.23 %	0.00 %	0.15 %	0.16 %	0.00 %	0.00 %
Avg/Max	1	0.80	0.80	0.80	1.00	0.62	0.49	0.53	0.84	0.56

Fig 12: Histogram %time

Major differences in terms of structure, types, number and granularity of tasks...

We can observe very appreciable differences between the two strategies. The first one is the number of tasks created. The <u>leaf version</u> is first divided into 16 tasks (basicsort function), where each thread executes it a single time and is fully parallelizable, and then divided again into 4 layers of 16 tasks (basicmerge function). However, the <u>tree version</u> is divided into a total of 134 tasks. In the figure 8 we can see that the program is divided into the first 4 tree tasks (multisort function). Inside of each task there are more tasks.

Identify the task ordering constraints that appear in each case and the causes for them, and the different kind of synchronisations that could be used to enforce them

<u>Leaf strategy:</u> Dependences are generated in basicmerge function. As we can see in figure 1, it needs to have two sorted arrays as parameters (left and right). The basicsort functions does this. In order to fix the dependencies, we could add synchronization clauses such as taskgroup or taskwait (it's better to use taskgroup because it has an implicit taskwait), so all the recursive calls' tasks are generated correctly.

<u>Tree strategy:</u> Both the cause of the constraint and the solution to it are the same as in the previous strategy. The recursive calls need sorted vectors in order to work.

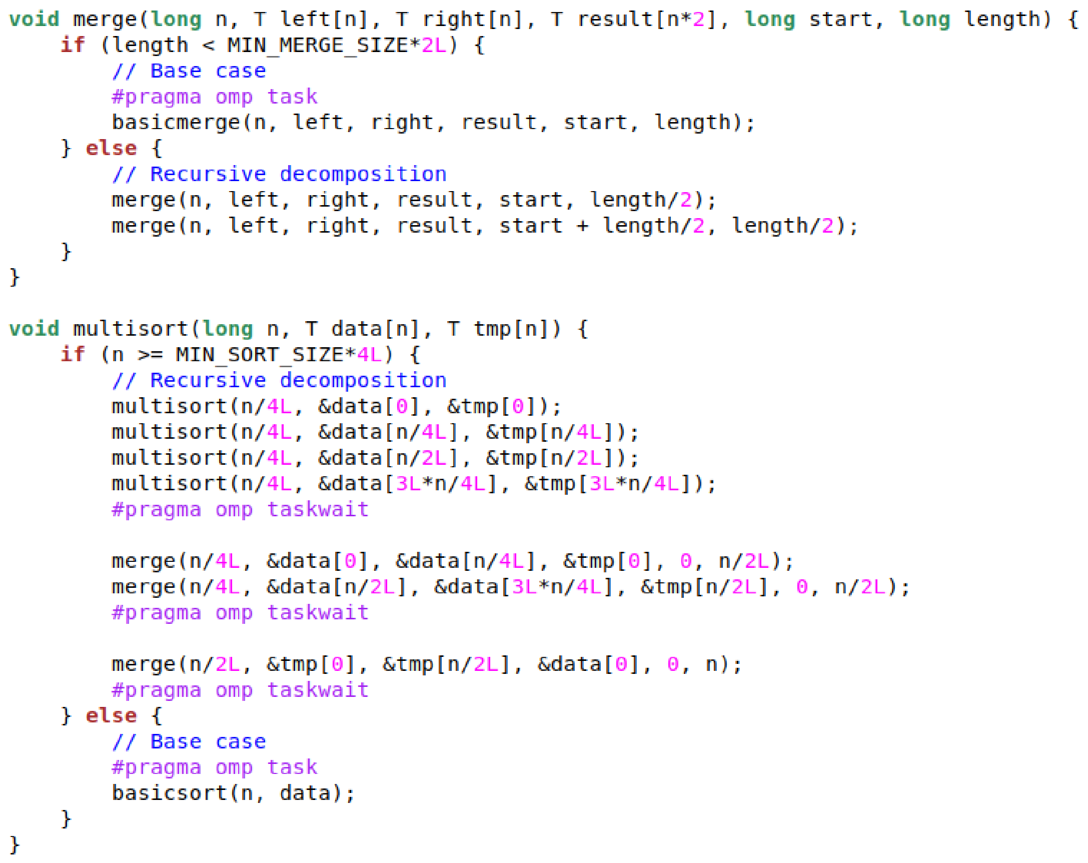
Do you notice any differences in terms of how and when tasks doing computation are generated? You will have to zoom at the beginning of the trace in order to observe these differences.

In the <u>leaf strategy</u>, in order for one task to be completed, all the recursive calls must have been made, because the tasks are generated in the leaves of the recursion. On the other hand, the <u>tree strategy</u> creates a task after each recursive call.
Now, the computing tasks, as the Paraver graphs show, are very different in execution time. In the <u>Leaf strategy</u>, basicsort functions (green) take the most time to execute (Figs 5 and 6). At the end, basicmerge functions (red) are executed and take a lot less time in comparisson to the first function (Fig 6).
In the <u>Tree strategy</u>, the Paraver graph shows a lot of colors (Figs 11 and 12), representing all the different tasks (from tree1 to tree9). The tasks that take the most time are the functions tree1 to tree4, corresponding to the multisort functions. If we keep zooming in, we can see the rest of the functions at the end, which take less time.

2. Parallelisation with OpenMP tasks

Leaf strategy

For this strategy, we will use #pragma omp task and #pragma omp taskwait clauses in the multisort and merge functions. It is also written both #pragma omp parallel and single right before the multisort call in the main.



```c
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);
        #pragma omp taskwait
    } else {
        // Base case
        #pragma omp task
        basicsort(n, data);
    }
}
```

Fig13: Leaf Strategy with OpenMP

The previous code is executed with 2 and 4 CPUs and the results are:

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=2**

****

Initialization time in seconds: 0.855893

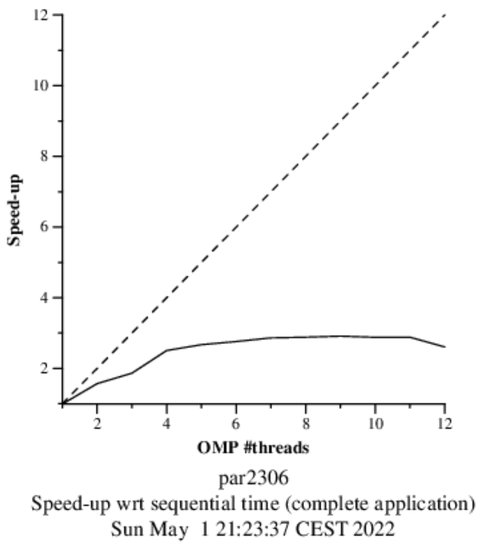
Multisort execution time: 3.682423

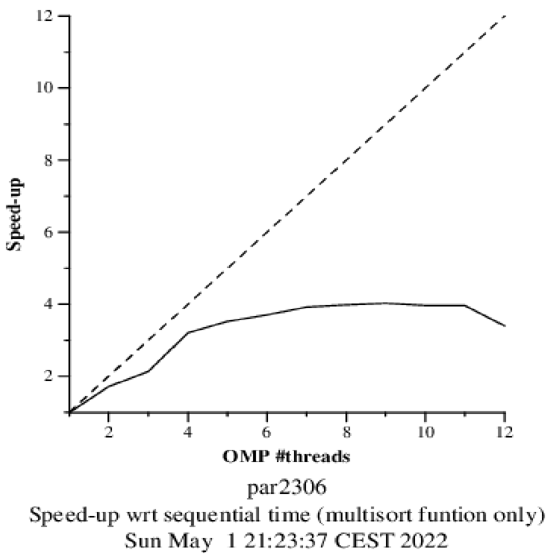
Check sorted data execution time: 0.016746

****

```
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=4
***************************************************************************************
Initialization time in seconds: 0.856895
Multisort execution time: 1.962076
Check sorted data execution time: 0.016359
***************************************************************************************
```

As we can see, the execution time with 2 CPUs is 3,68 s and with 4 CPUs is 1,96 s approximately, so we will assume that the parallelism is correct.
Now let's watch the Postscript file generated. It contains 2 speed-up plots that look like this:





Figs 14 and 15: Leaf Strategy speed-up plots

Referring to the previous speed-up plots, we can assure that the speed-up for this strategy is not good. Both plots start to deviate from the ideal speed-up line as soon as we get to 2 CPUs or more. This happens because this strategy tries to parallelise only the leaves of the recursion. The sequential calls leading to the leaves are kept sequential. This affects the performance heavily.

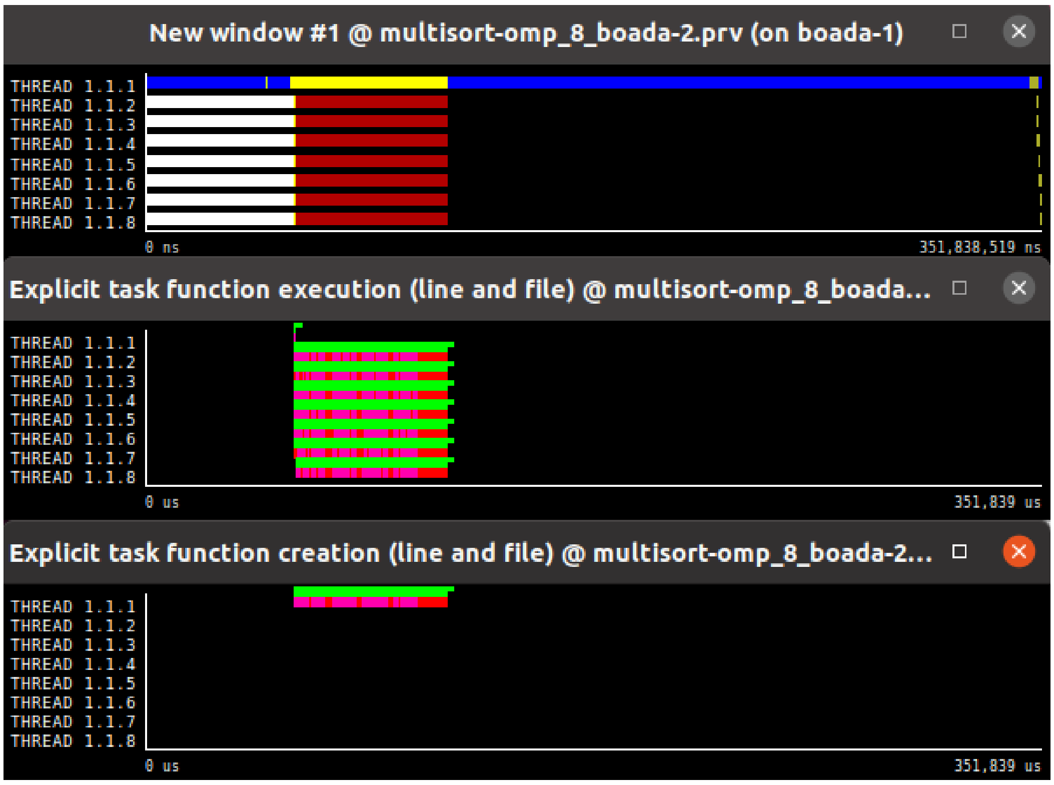
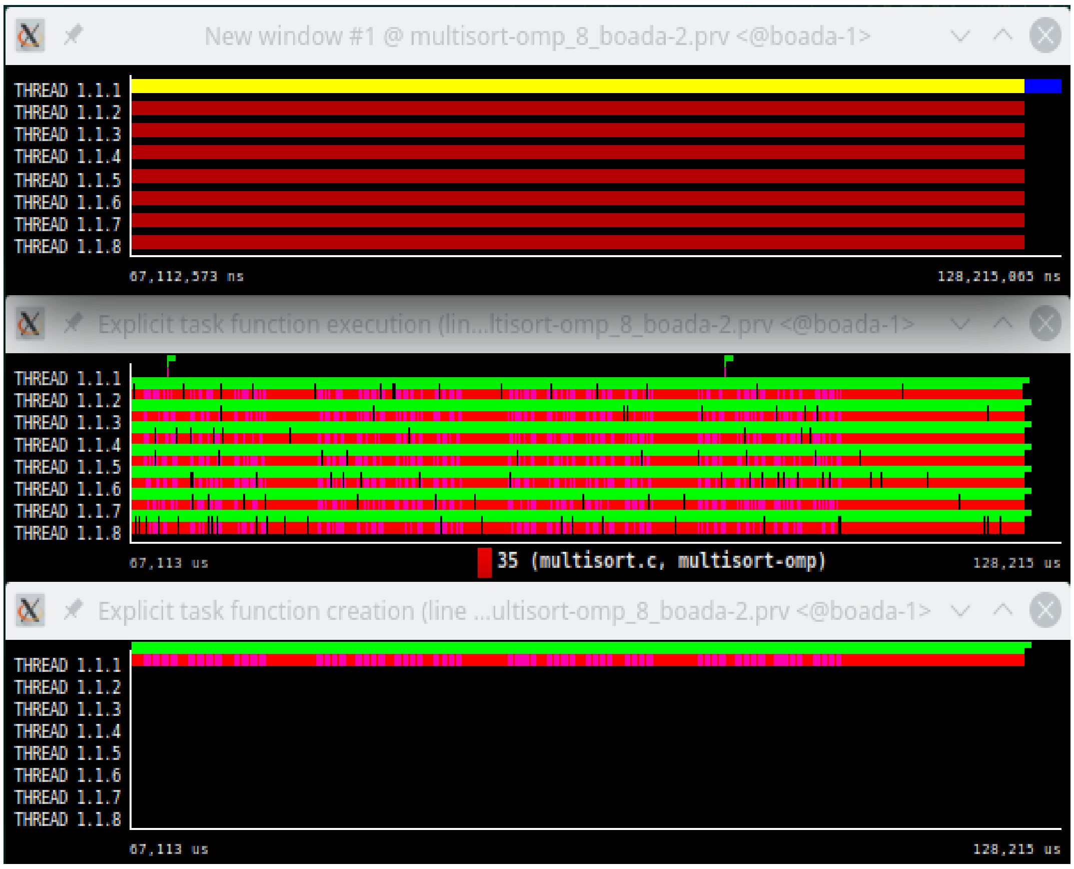
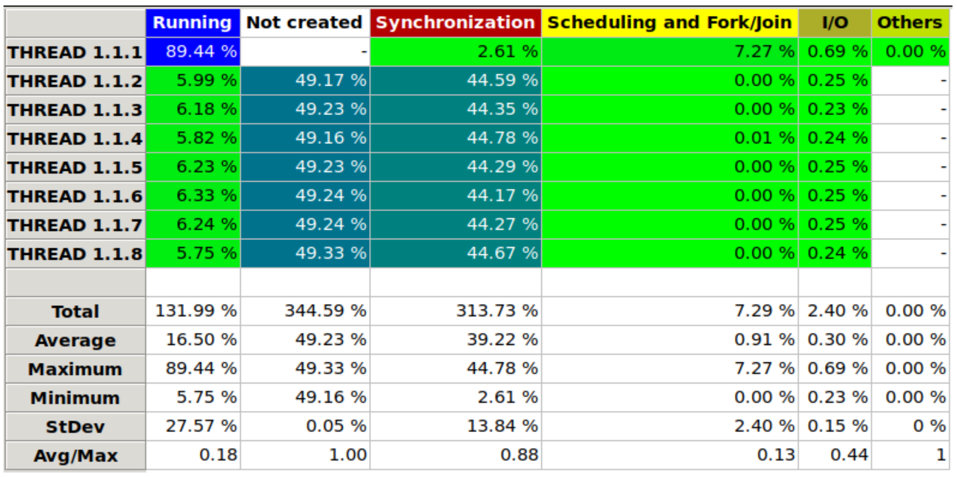

New window #1 @ multisort-omp_8_boada-2.prv (on boada-1)
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
351,838,519 ns
Explicit task function execution (line and file) @ multisort-omp_8_boada...
0 us
351,839 us
Explicit task function creation (line and file) @ multisort-omp_8_boada-2...
0 us
351,839 us

Fig 16: Thread state, Explicit task function creation & execution

if we zoom in we can see:


New window #1 @ multisort-omp_8_boada-2.prv <@boada-1>
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
67,112,573 ns
128,215,865 ns
Explicit task function execution (lin...ltisort-omp_8_boada-2.prv <@boada-1>
67,113 us
35 (multisort.c, multisort-omp)
128,215 us
Explicit task function creation (line ...ultisort-omp_8_boada-2.prv <@boada-1>
67,113 us
128,215 us



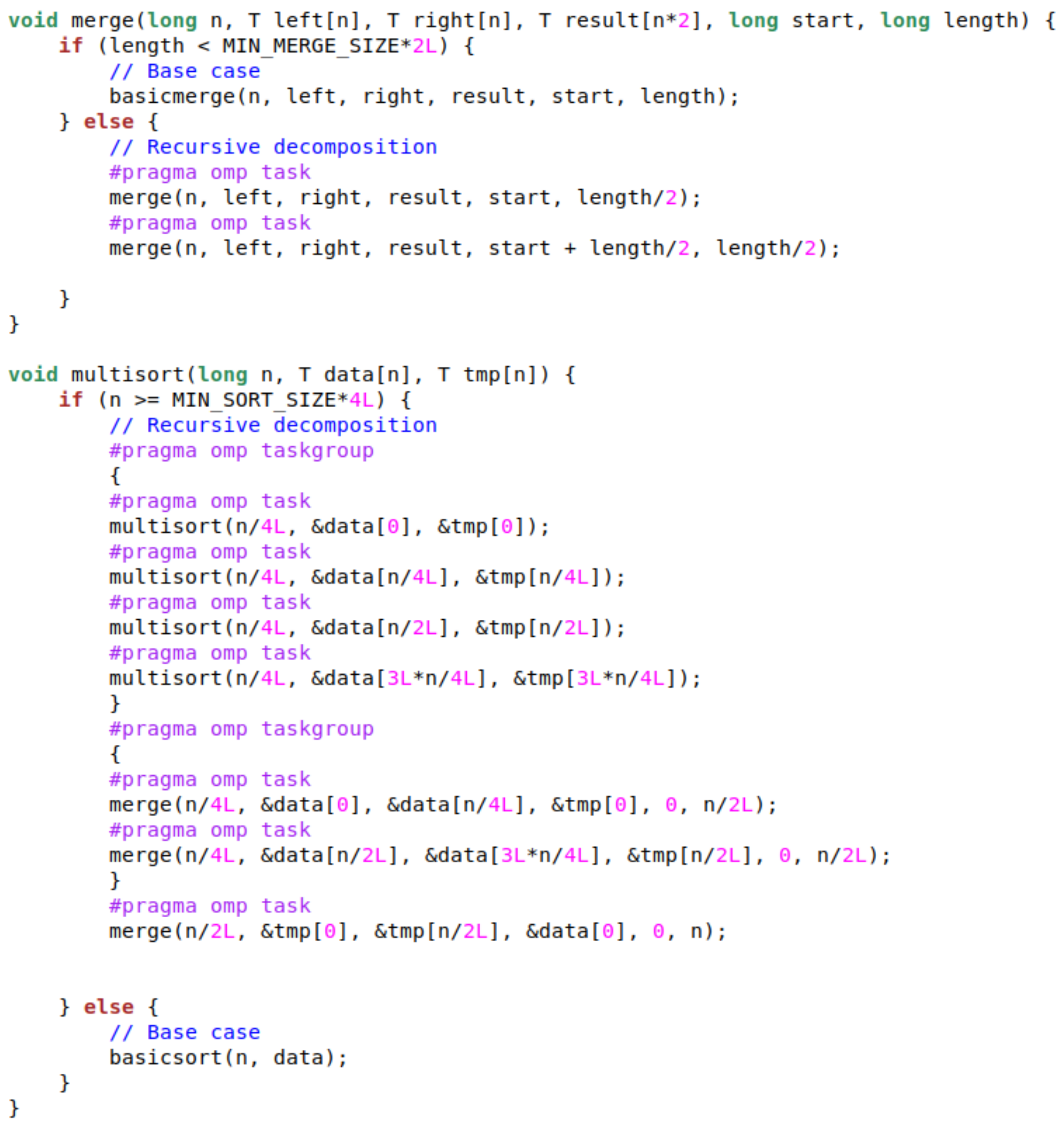
	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	89.44 %	-	2.61 %	7.27 %	0.69 %	0.00 %
THREAD 1.1.2	5.99 %	49.17 %	44.59 %	0.00 %	0.25 %	-
THREAD 1.1.3	6.18 %	49.23 %	44.35 %	0.00 %	0.23 %	-
THREAD 1.1.4	5.82 %	49.16 %	44.78 %	0.01 %	0.24 %	-
THREAD 1.1.5	6.23 %	49.23 %	44.29 %	0.00 %	0.25 %	-
THREAD 1.1.6	6.33 %	49.24 %	44.17 %	0.00 %	0.25 %	-
THREAD 1.1.7	6.24 %	49.24 %	44.27 %	0.00 %	0.25 %	-
THREAD 1.1.8	5.75 %	49.33 %	44.67 %	0.00 %	0.24 %	-
Total	131.99 %	344.59 %	313.73 %	7.29 %	2.40 %	0.00 %
Average	16.50 %	49.23 %	39.22 %	0.91 %	0.30 %	0.00 %
Maximum	89.44 %	49.33 %	44.78 %	7.27 %	0.69 %	0.00 %
Minimum	5.75 %	49.16 %	2.61 %	0.00 %	0.23 %	0.00 %
StDev	27.57 %	0.05 %	13.84 %	2.40 %	0.15 %	0 %
Avg/Max	0.18	1.00	0.88	0.13	0.44	1

Fig 17: Thread state Histogram

Figure 16 shows that only the first thread is in charge of creating tasks, since it is doing all the sequential work leading to the leaves. Once the main thread has executed all the previous recursive calls by itself, the leaves are created. They do their job in parallel, and once it's finished, the task dies. Then the main thread continues to carry on until the end of the execution. The synchronization overhead lies only onto the main thread. Finally, there is a big portion of the program that is executed sequentially from half of the execution until the end. This affects the performance negatively.

Tree strategy

For this strategy, we will use Task and Taskgroup clauses in the multisort and merge functions. It is written both #pragma omp parallel and single right before the multisort call in the main.



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

    }
}

void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        #pragma omp taskgroup
        {
        #pragma omp task
        multisort(n/4L, &data[0], &tmp[0]);
        #pragma omp task
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        #pragma omp task
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        #pragma omp task
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
        }
        #pragma omp taskgroup
        {
        #pragma omp task
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        #pragma omp task
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        }
        #pragma omp task
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);


    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Fig 18: Tree Strategy with OpenMP

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=2**

****

Initialization time in seconds: 0.855414

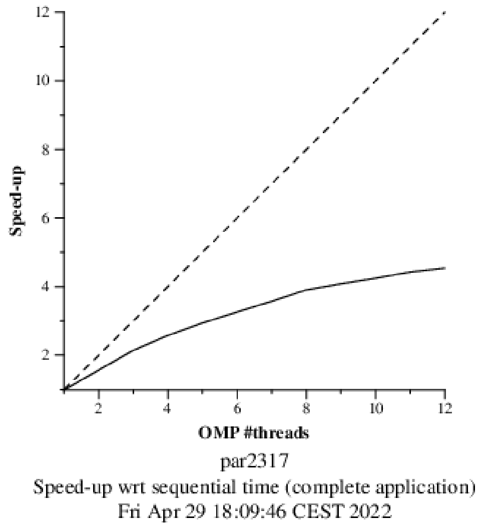
Multisort execution time: 3.677105

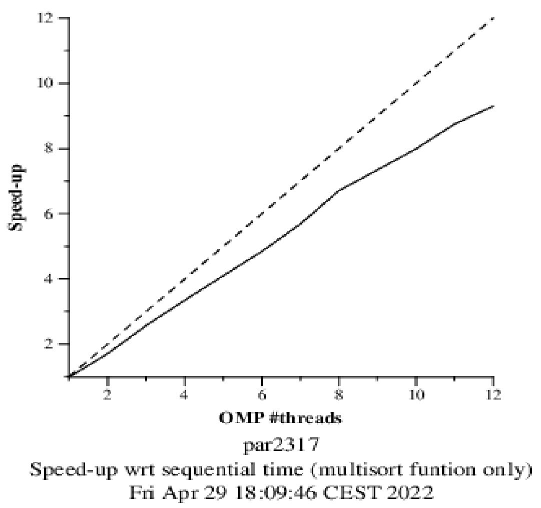
Check sorted data execution time: 0.017659

****

```
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=4
****************************************************************************************
Initialization time in seconds: 0.854644
Multisort execution time: 1.875355
Check sorted data execution time: 0.018054
****************************************************************************************
```

Both execution times seem to be inside the normal boundaries, so we will assume that the parallelism is correct. Now let's watch the Postscript file generated. It contains 2 speed-up plots that look like this:





Figs 19 and 20: Tree Strategy speed-up plots

As we can see, the overall speedup is not very good. This happens because the tree strategy generates a task in each recursive call, and this generates overheads. But it's better than the Leaf strategy, because more tasks are being executed in parallel. However, the speedup of the multisort function is near perfection, it's really, really good. As more threads are added, the performance increases.

Now we will check Paraver graphs, so we can see why the program isn't working as expected. We have zoomed in the Thread state and Tasks Creation and Execution graphs, and it looks like this:

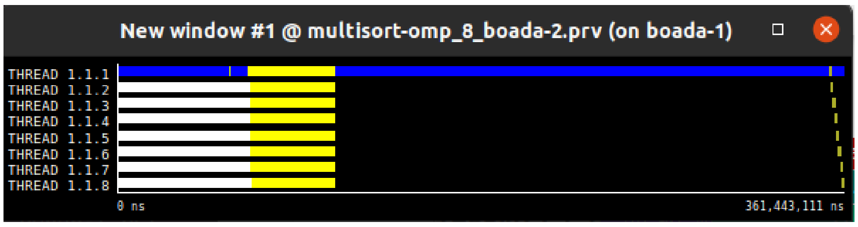


Fig 21: Thread state graph of the Tree Strategy

if we zoom in we can see:

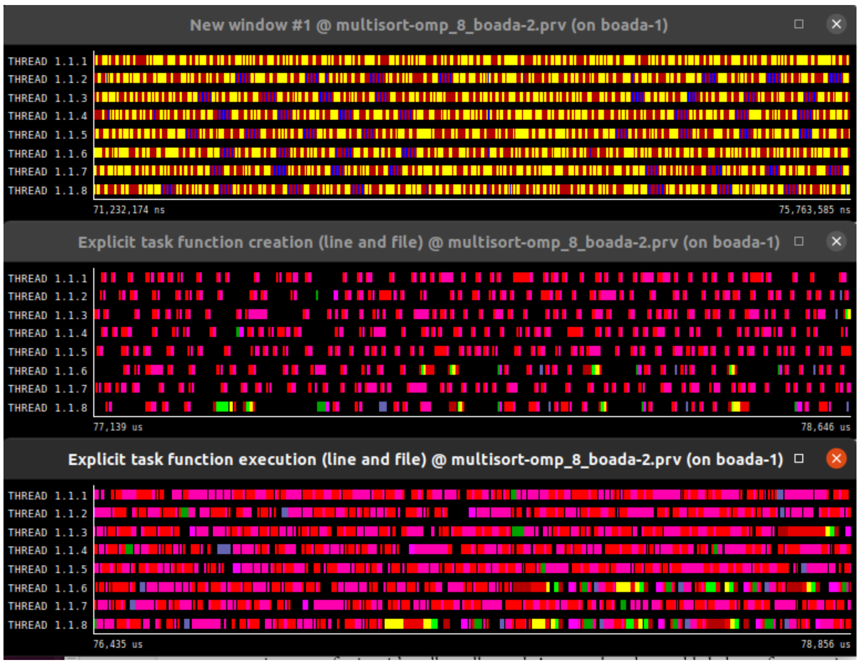
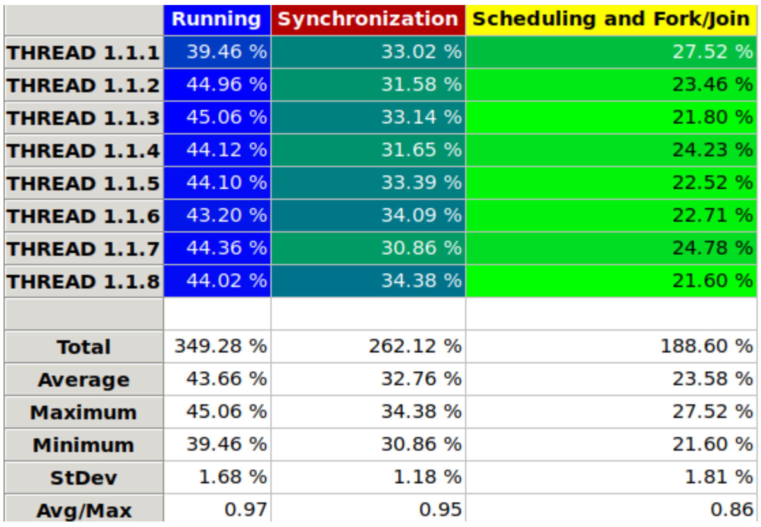

New window #1 @ multisort-omp_8_boada-2.prv (on boada-1)
71,232,174 ns
75,763,585 ns
Explicit task function creation (line and file) @ multisort-omp_8_boada-2.prv (on boada-1)
77,139 us
78,646 us
Explicit task function execution (line and file) @ multisort-omp_8_boada-2.prv (on boada-1)
76,435 us
78,856 us
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

Fig 22: Thread state, Explicit task function creation & execution



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	39.46 %	33.02 %	27.52 %
THREAD 1.1.2	44.96 %	31.58 %	23.46 %
THREAD 1.1.3	45.06 %	33.14 %	21.80 %
THREAD 1.1.4	44.12 %	31.65 %	24.23 %
THREAD 1.1.5	44.10 %	33.39 %	22.52 %
THREAD 1.1.6	43.20 %	34.09 %	22.71 %
THREAD 1.1.7	44.36 %	30.86 %	24.78 %
THREAD 1.1.8	44.02 %	34.38 %	21.60 %
Total	349.28 %	262.12 %	188.60 %
Average	43.66 %	32.76 %	23.58 %
Maximum	45.06 %	34.38 %	27.52 %
Minimum	39.46 %	30.86 %	21.60 %
StDev	1.68 %	1.18 %	1.81 %
Avg/Max	0.97	0.95	0.86

Fig 23: Thread state Histogram

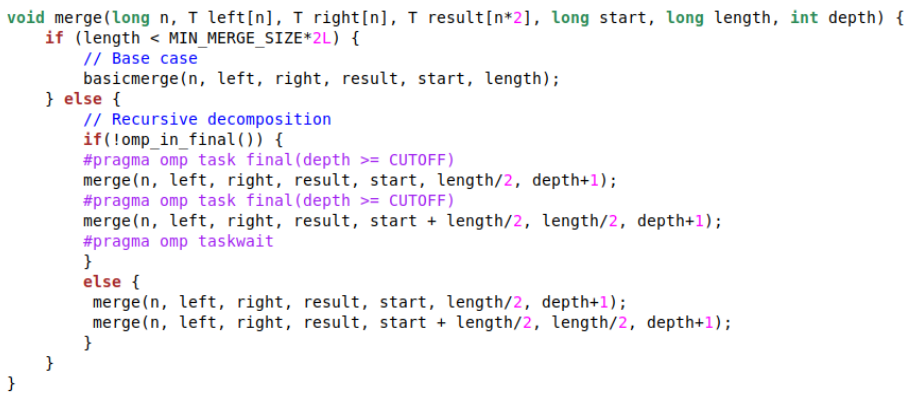
These graphs and the histograms show us that all threads are creating tasks, instead of just one, just like the Tree strategy tells us. However, we can see a lot of yellow in figure 22, which indicates that there is quite a lot of synchronization overhead. We need a way to increase the performance.There is a big sequential portion at the end of the execution. This will affect the scalability.

Compare the scalability results and traces generated for both strategies and draw the appropriate conclusions.

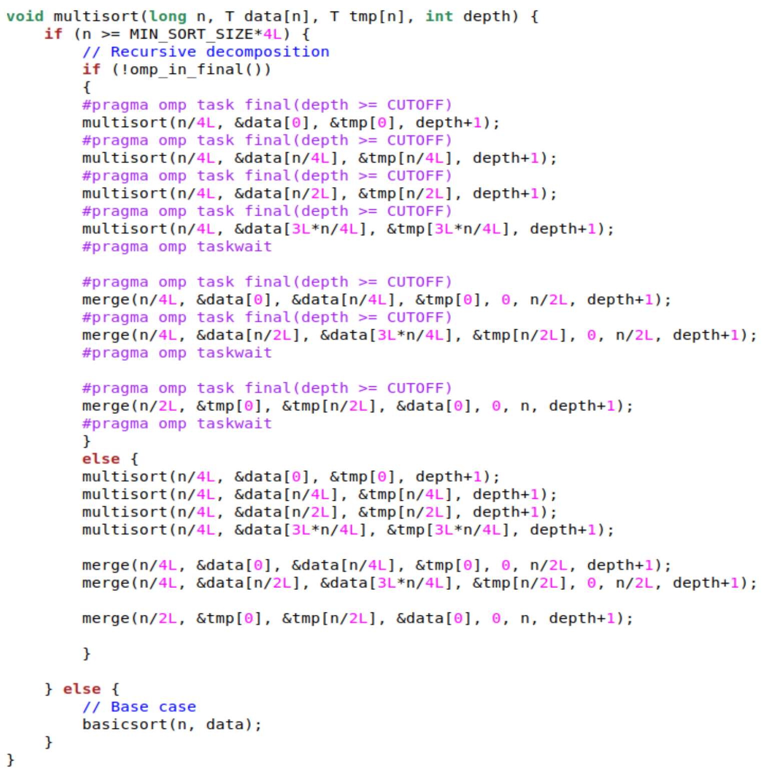
Both strategies don't have very good scalability, but for different reasons. The leaf strategy parallelises only the leaves, so the main recursive part is kept sequential. The tree strategy, however, has a lot of synchronization overhead because too many tasks are created, and there are tasks inside of tasks. If we had to choose between the 2 options, our best choice would be the tree strategy, because in terms of execution time and scalability, this strategy is superior. We would only recommend the leaf strategy for programs with a recursion tree that is not very deep. Otherwise the main thread would have to travel across a very big recursive portion that would be sequential.

Task granularity control: the cut-off mechanism

For both strategies, we have no way of controlling how many tasks are generated. But for the Tree strategy, there is a way: the cut-off mechanism. This will allow us to control the maximum recursion we can get for each level. We have made the following changes to the code:



```
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 {
         merge(n, left, right, result, start, length/2, depth+1);
         merge(n, left, right, result, start + length/2, length/2, depth+1);
        }
    }
}
```



```
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)
        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 taskwait

        #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 taskwait

        #pragma omp task final(depth >= CUTOFF)
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);
        #pragma omp taskwait
        }
        else {
        multisort(n/4L, &data[0], &tmp[0], depth+1);
        multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
        multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);

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

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

        }

    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Figs 24 and 25:
Tree Strategy after implementation of a cutoff method

We have added “final(depth >= CUTOFF)” right after every #pragma omp task clause, so that once the variable depth (new variable created by us) reaches the CUTOFF, it will stop generating more tasks and will start executing sequentially. We have also added the omp_in_final function, which returns true once we have arrived at the cutoff desired level, so once we’re there, the program will execute normally.

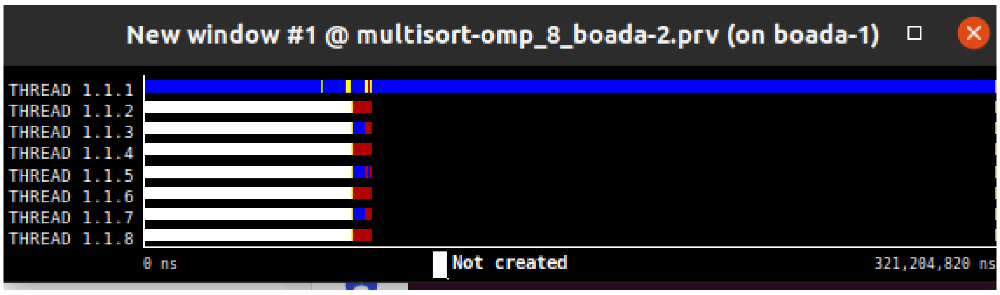


Fig 26: Thread state with CUTOFF = 0

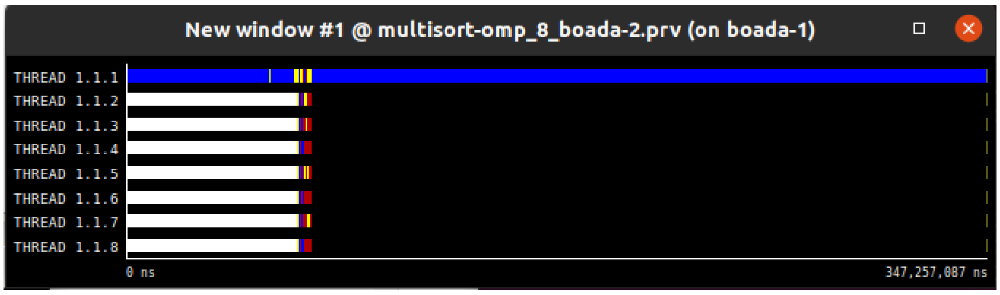


Fig 27: Thread state with CUTOFF = 1

What we can see in both images is that applying a cutoff method reduces the number of tasks created, and thus, the work done in parallel. The blue section in thread 1 has increased compared to the normal tree version graph.

Fig 28 shows us how the execution time of the tree strategy changes depending on the value of CUTOFF. When we have 16 threads, the value is around 7, and we achieve the optimal execution time.

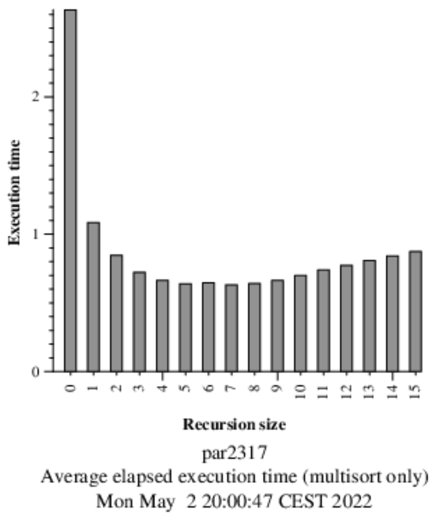
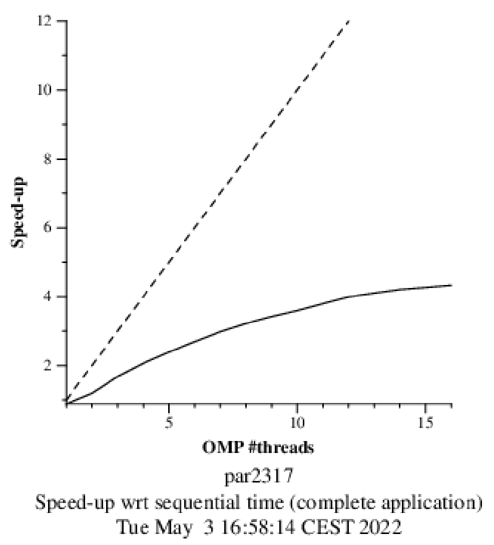
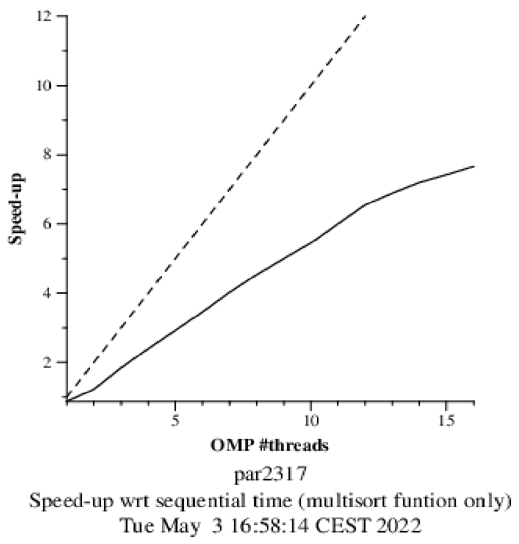


Fig 28: Postscript with CUTOFF variable

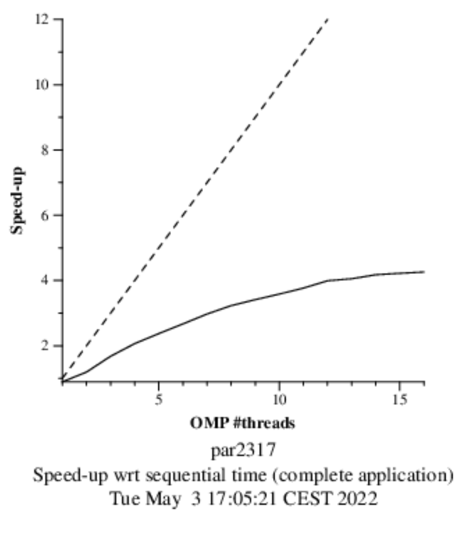
Compare the speed-up plots generated when submitting the submit-strong-omp.sh script with the values for sort size and merge size set to 128 and for two different values for cutoff: the original one in the script (the maximum) and the optimum value you have found in the previous exploration. Comment the results that you obtain.

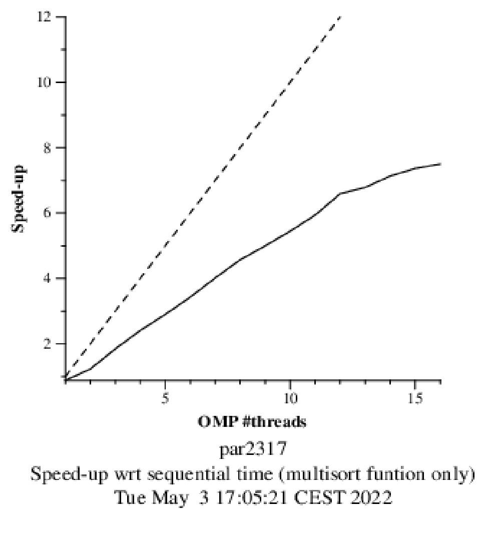
With this paragraph we're going to do the following: now that we know that our optimal cutoff is around 7, we will generate 3 scalability plots: the first one will consist of the program with our optimal CUTOFF value (7). The next one will be generated with twice the value of the cutoff (14). And the last one will be the regular tree strategy version with no cutoff. All of the plots will have merge_size and sort_size variables set to 128 in the submit-strong.sh file.



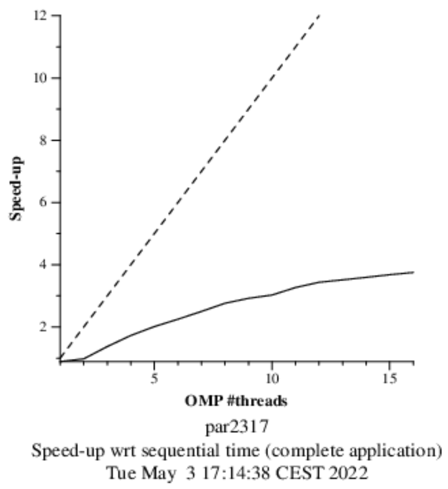


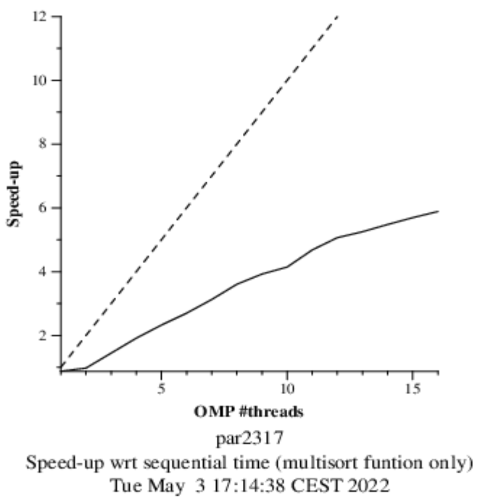
Figs 29 and 30: Speed-up plots with cutoff = 7





Figs 31 and 32: Speed-up plots with cutoff = 14





Figs 33 and 34: Speed-up plots without cutoff

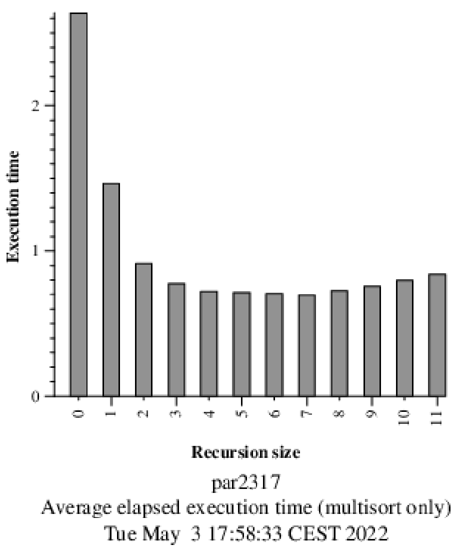
Having seen this scalability plots, we can confirm that:

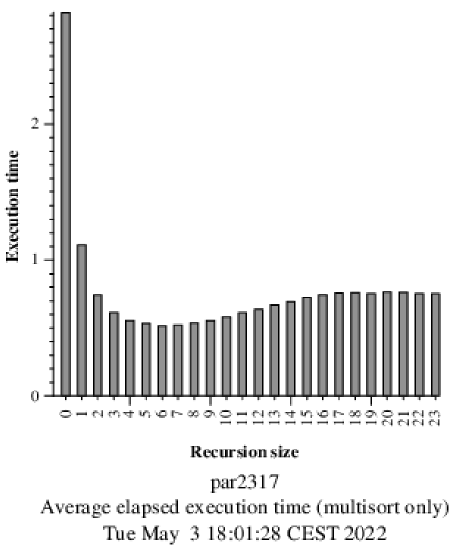
- Having an optimal cutoff instead of choosing a random one increases the speed-up of the program (Figs 29 and 30).
- Not having a cutoff strategy at all makes your program run badly (Figs 33 and 34).

In our case, the first plots are better than the seconds, which are better than the thirds.

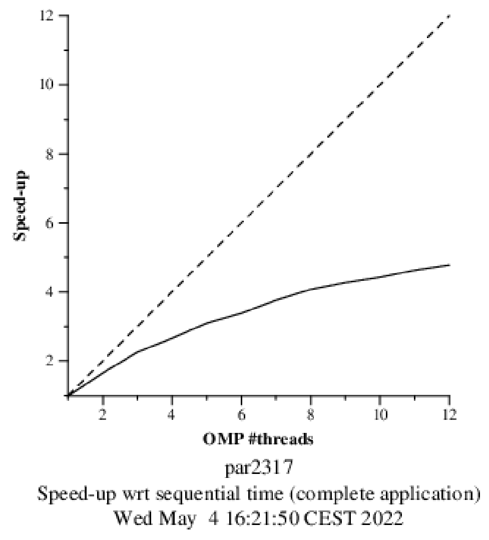
Optional 1: Have you explored the scalability of your tree implementation with cut–off when using up to 24 threads? Why is performance still growing when using more than the 12 physical cores available in boada-1 to 4? Set the maximum number of cores to be used (variable np_NMAX) by editing the submit-strong-omp.sh script in order to do the complete analysis.

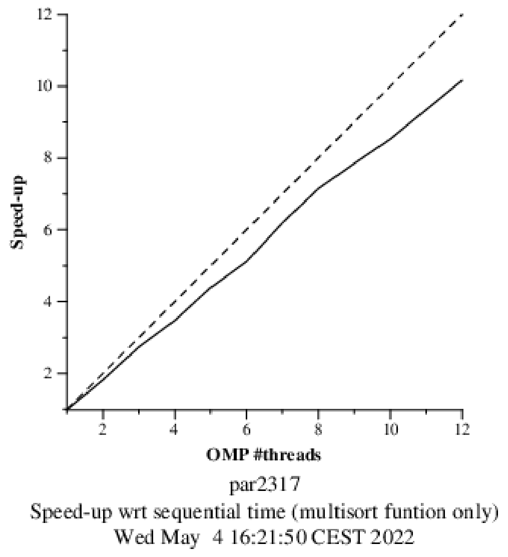
We have calculated the optimum cutoff value for 12 and 24 threads. The following graphs show us the values: for 12 threads, the optimum value is 7, and for 24 threads, it's 6.



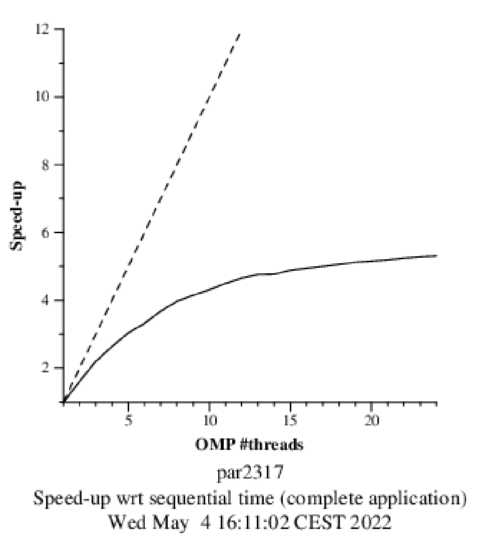


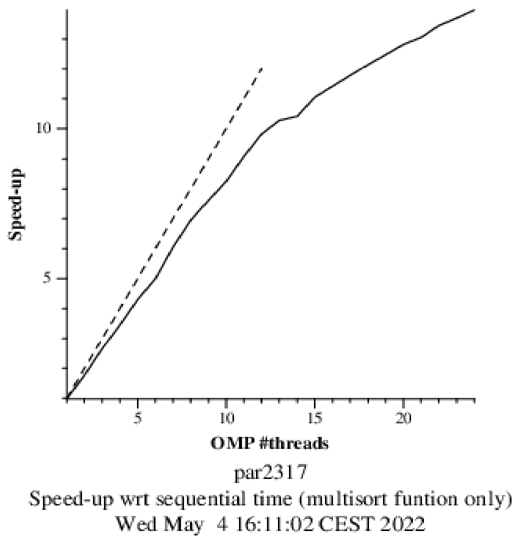
Figs 35 and 36: Postscript plot used to calculate the optimum cutoff with 12 and 24 threads





Figs 37 and 38: Speed-up plots with 12 threads and its optimum cutoff



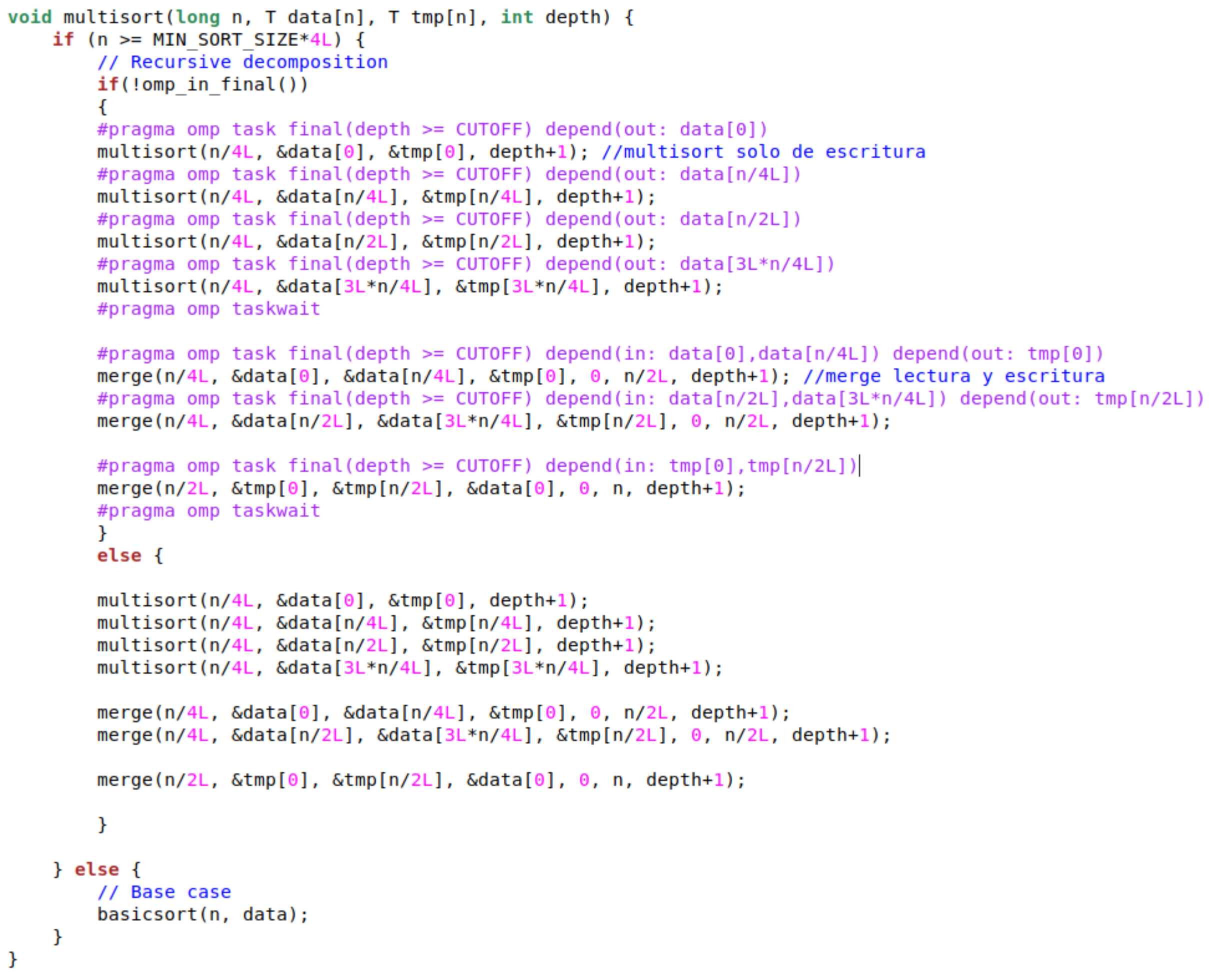


Figs 39 and 40: Speed-up plots with 24 threads and its optimum cutoff

The performance of the program keeps increasing because even though boada has a maximum of 12 physical threads, if we use 24, we are using "virtual threads", and that allows us to increase the performance of the program. The extra threads help execute tasks that with 12 threads we wouldn't be able to execute. It fills the time "holes" where nothing is being executed.

3. Using OpenMP task dependencies

In this session, we will modify our Tree Strategy code with a Cut-off mechanism to express dependencies between tasks, using OpenMP. The code after this change looks like this:



```
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); //multisort solo de escritura
        #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 taskwait

        #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); //merge lectura y escritura
        #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 {

        multisort(n/4L, &data[0], &tmp[0], depth+1);
        multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
        multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);

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

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

        }

    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Fig 41: Tree Strategy with cutoff and depend clauses

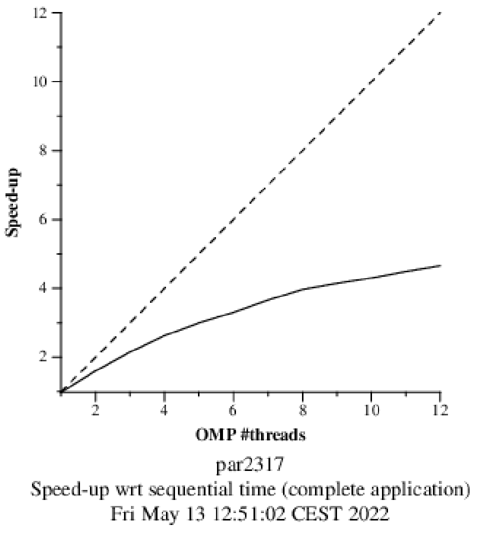
Only the multisort function is changed, adding depend clauses after each #pragma omp task.
In the multisort function calls of fig 41, depend clauses have been added out. And in the merge function calls the in and out clauses. The data vector is read and the tmp vector is written, so reads will be depend in and writes will be depend out.

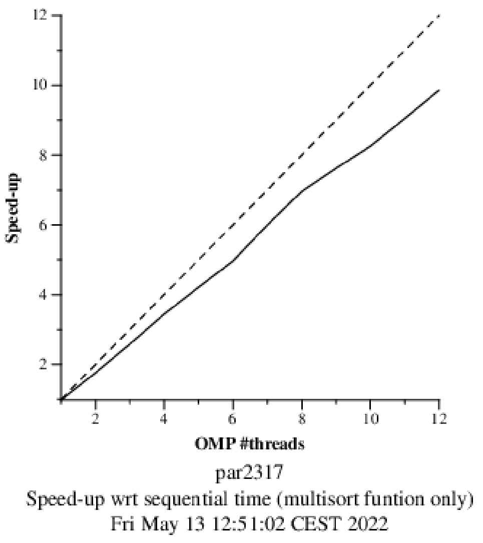
Compile and submit for parallel execution using 8 processors. Make sure that the program verifies the result of the sort process and does not throw errors about unordered positions.

After compiling and executing the program, we can see a noticeable improvement in the Multisort function execution time and the execution time of checking the sorted data. The program does not throw any kind of error regarding unordered positions.

Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level: **CUTOFF=7**
Number of threads in OpenMP: **OMP_NUM_THREADS=8**
****
Initialization time in seconds: 0.855643
Multisort execution time: **0.862321**
Check sorted data execution time: **0.015424**
****

Analyse its scalability by looking at the two strong scalability plots and compare the results with the ones obtained in the previous chapter. Are they better or worse in terms of performance? In terms of programmability, was this new version simpler to code?





Figs 42 and 42: Speed-up plots with depend clauses

Comparing them with the previous speed-up plots from the last session (figs 29, 30, 31, 32, 33, 34, 37 and 38), we observe that the performance of this strategy with dependence clauses is better than those with a bad cutoff strategy, regardless of not having dependence clauses. However, in the last 2 figures (37 and 38), the ones that were executed with 12 threads and optimal cutoff, we can see that the plots are almost identical.
This shows us that this improvement, despite being harder in terms of coding, doesn't improve the performance as much as we would have liked.

Trace the parallel execution with 8 processors and use the appropriate configuration files to visualise how the parallel execution was done and to understand the performance achieved.

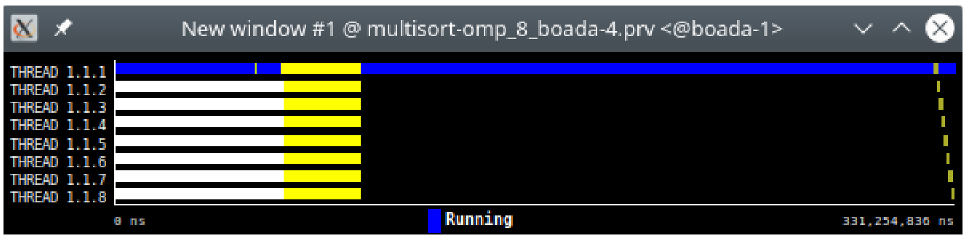


Fig 43: Thread state of Tree Strategy with cutoff and depend clauses

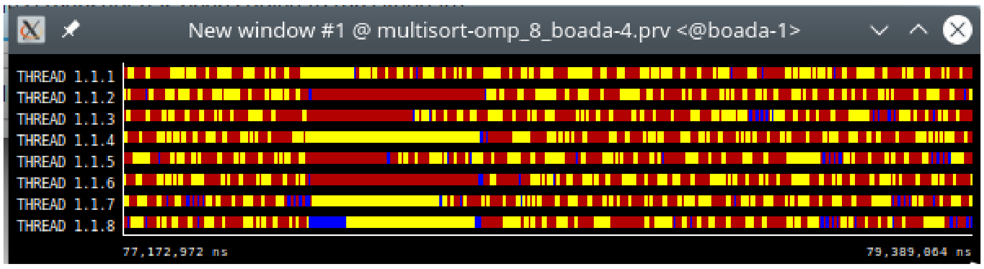
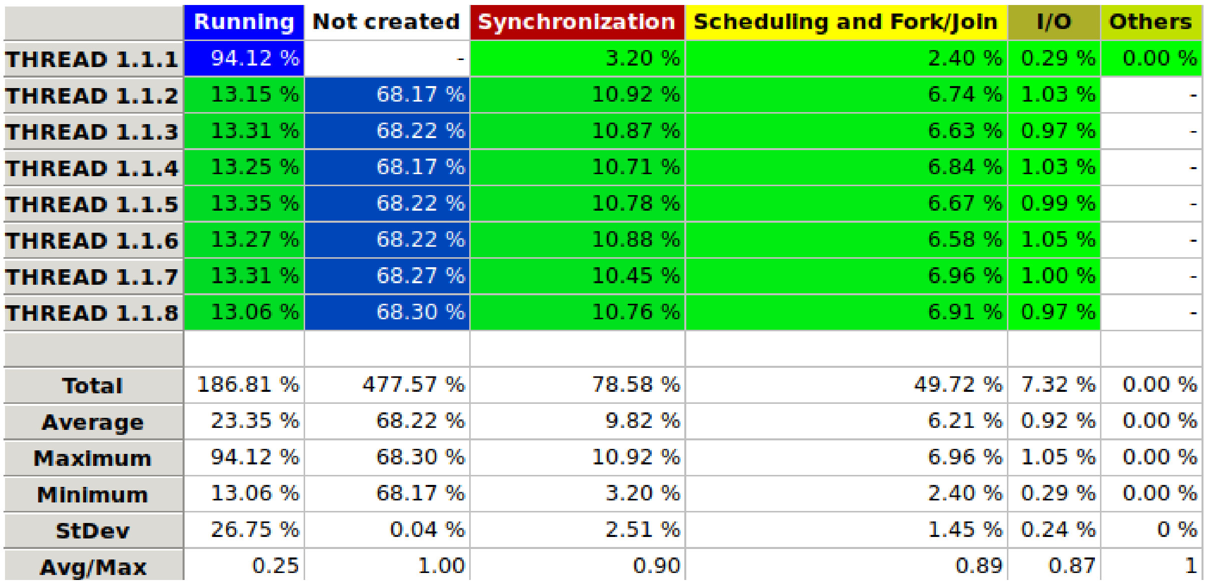


Fig 44: Zoom in of the Thread state graph



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	94.12 %	-	3.20 %	2.40 %	0.29 %	0.00 %
THREAD 1.1.2	13.15 %	68.17 %	10.92 %	6.74 %	1.03 %	-
THREAD 1.1.3	13.31 %	68.22 %	10.87 %	6.63 %	0.97 %	-
THREAD 1.1.4	13.25 %	68.17 %	10.71 %	6.84 %	1.03 %	-
THREAD 1.1.5	13.35 %	68.22 %	10.78 %	6.67 %	0.99 %	-
THREAD 1.1.6	13.27 %	68.22 %	10.88 %	6.58 %	1.05 %	-
THREAD 1.1.7	13.31 %	68.27 %	10.45 %	6.96 %	1.00 %	-
THREAD 1.1.8	13.06 %	68.30 %	10.76 %	6.91 %	0.97 %	-
Total	186.81 %	477.57 %	78.58 %	49.72 %	7.32 %	0.00 %
Average	23.35 %	68.22 %	9.82 %	6.21 %	0.92 %	0.00 %
Maximum	94.12 %	68.30 %	10.92 %	6.96 %	1.05 %	0.00 %
Minimum	13.06 %	68.17 %	3.20 %	2.40 %	0.29 %	0.00 %
StDev	26.75 %	0.04 %	2.51 %	1.45 %	0.24 %	0 %
Avg/Max	0.25	1.00	0.90	0.89	0.87	1

Fig 45: Histogram of Thread state

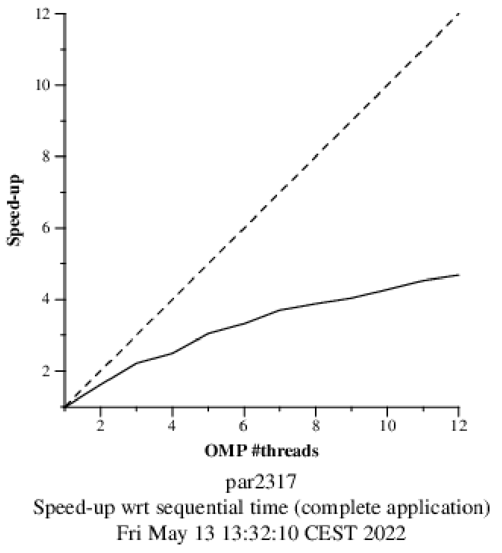
After executing the program and visualizing the graphs with Paraver, we see that there is a big yellow area in the middle of fig 43, and that tells us that task creation overhead takes place in all threads. However, if we zoom in, we see that all threads are doing work, some more than others (fig 45), but the synchronization overhead is bigger than the task creation overhead in %. If we compare it with figures 21, 26 and 27, we can see that fig 43 is very similar to fig 21.

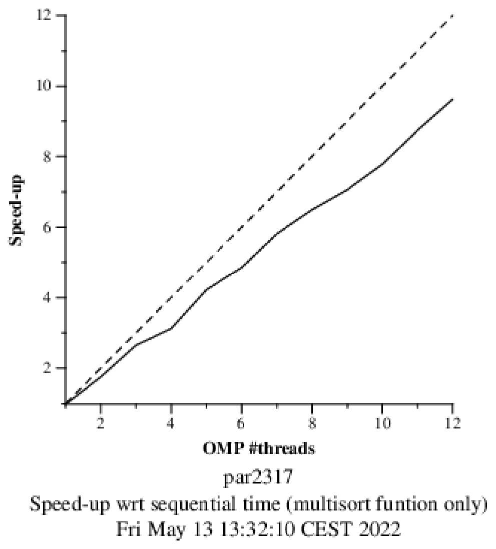
Optional 2: Complete your parallel implementation of the multisort.c by parallelising the two functions that initialise the data and tmp vectors. Analyse the scalability of the new parallel code by looking at the two speed–up plots generated when submitting the submit-strong-omp.sh script. Reason about the new performance obtained with support of Paraver timelines.

For this optional, we will parallelise the initialise function that initializes the data and tmp vectors. We have added a parallel region in the main with #pragma omp parallel and #pragma omp single right before the initialise and clear function calls.

```
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
Cut-off level:                 CUTOFF=7
Number of threads in OpenMP:             OMP_NUM_THREADS=8
*******************************************************************************************
Initialization time in seconds: 0.807420
Multisort execution time: 0.911678
Check sorted data execution time: 0.016645
*******************************************************************************************
```

In this output of the program, we can see that the Initalization time has decreased because the part of the program responsible for this has been paralelised. However, the multisort execution time has increased the same amount of time that the initialization has decreased. The total execution time keeps being the same, but the % of each part has changed.





Figs 46 and 47: Speed up plots with parallelised data and tmp vectors initialisation

Looking at figs 46 and 47, we see that the performance has not changed significantly, we could even say that it has not increased, compared with figs 42 and 43, especially with more threads.

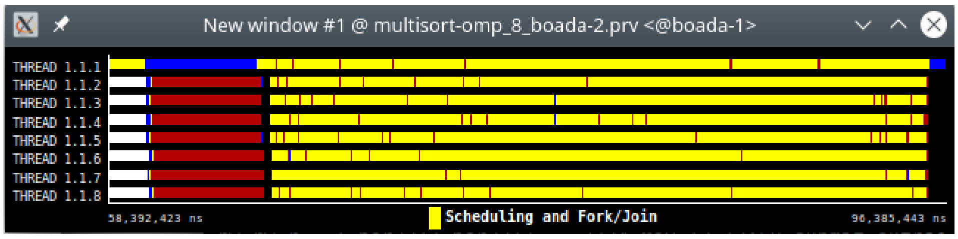
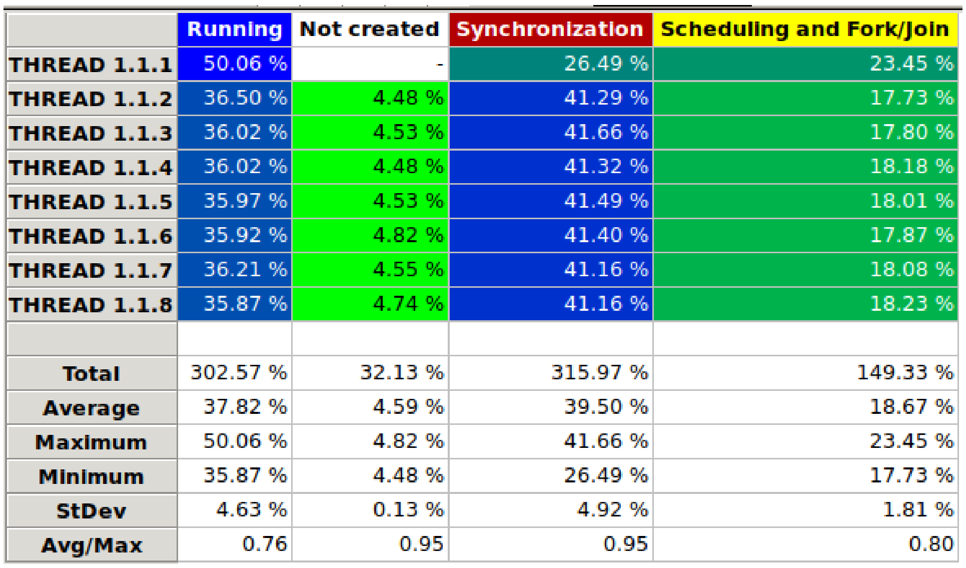
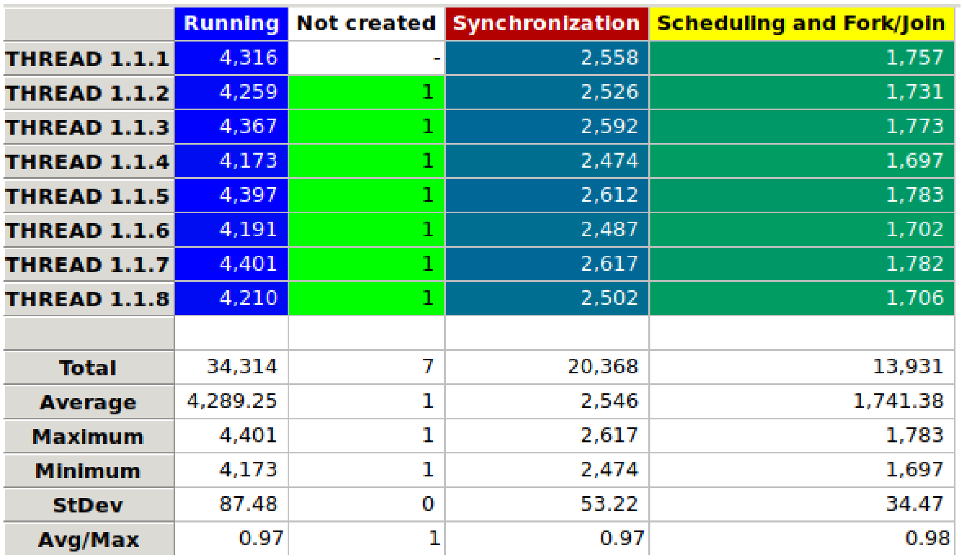


Fig 48: Thread state of the program with parallelised data and tmp vectors initialisation



	Running	Not created	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	50.06 %	-	26.49 %	23.45 %
THREAD 1.1.2	36.50 %	4.48 %	41.29 %	17.73 %
THREAD 1.1.3	36.02 %	4.53 %	41.66 %	17.80 %
THREAD 1.1.4	36.02 %	4.48 %	41.32 %	18.18 %
THREAD 1.1.5	35.97 %	4.53 %	41.49 %	18.01 %
THREAD 1.1.6	35.92 %	4.82 %	41.40 %	17.87 %
THREAD 1.1.7	36.21 %	4.55 %	41.16 %	18.08 %
THREAD 1.1.8	35.87 %	4.74 %	41.16 %	18.23 %
Total	302.57 %	32.13 %	315.97 %	149.33 %
Average	37.82 %	4.59 %	39.50 %	18.67 %
Maximum	50.06 %	4.82 %	41.66 %	23.45 %
Minimum	35.87 %	4.48 %	26.49 %	17.73 %
StDev	4.63 %	0.13 %	4.92 %	1.81 %
Avg/Max	0.76	0.95	0.95	0.80

Fig 49: Histogram of Thread state graph (%time)



	Running	Not created	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	4,316	-	2,558	1,757
THREAD 1.1.2	4,259	1	2,526	1,731
THREAD 1.1.3	4,367	1	2,592	1,773
THREAD 1.1.4	4,173	1	2,474	1,697
THREAD 1.1.5	4,397	1	2,612	1,783
THREAD 1.1.6	4,191	1	2,487	1,702
THREAD 1.1.7	4,401	1	2,617	1,782
THREAD 1.1.8	4,210	1	2,502	1,706
Total	34,314	7	20,368	13,931
Average	4,289.25	1	2,546	1,741.38
Maximum	4,401	1	2,617	1,783
Minimum	4,173	1	2,474	1,697
StDev	87.48	0	53.22	34.47
Avg/Max	0.97	1	0.97	0.98

Fig 50: Histogram of Thread state graph (#bursts)

Now onto the Paraver graphs. With this new version of the program, we see that a new red region has been created. This is because that's where the data and tmp vectors are initialised. Then, we have the typical yellow region, with very few red spots. In the previous version, the yellow region and red were alternated, so a task would be created and then synchronized (fig 44). But in this version, we create all the tasks first and then they do work (fig 48).

Conclusions

In these laboratory sessions we had to parallelise the mergesort algorithm applying two different strategies: the leaf strategy and the tree strategy. After having studied them with Paraver and Tareador (execution times, scalability plots and graphics), we have come to the conclusion that the latter one is much better than the former one, so we focused more on improving the tree strategy with different mechanisms and strategies: a cut-off mechanism and dependence clauses, which has brought us even greater results.