UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Izan Cordobilla Blanco (par1304)

Alejandro Salvat Navarro (par1120)

Grau en enginyeria informàtica

Quadrimestre de primavera 2021-2022

INDEX

1. INTRODUCTION

In this laboratory assignment we explore the use of parallelism in recursive programs. Recursive task decompositions are parallelisation strategies that try to exploit this parallelism and in this case, we will explore Mergesort.

Mergesort is a Divide and Conquer algorithm that consists in dividing the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

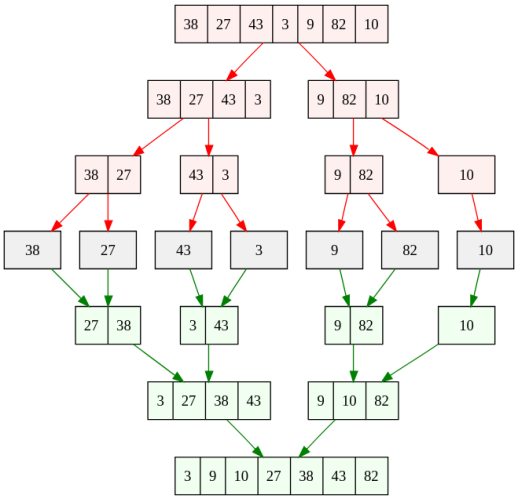

Figure 1: Mergesort algorythm

The strategies that we are going to explore are the Leaf and Tree Strategies. The first one consists of creating tasks for the leaves and the second one is based on generating a new task every time a recursive call is performed.

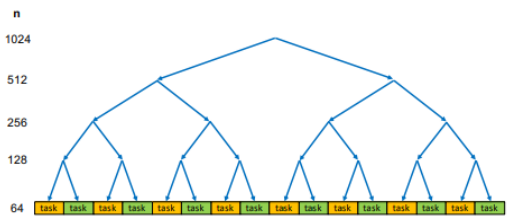

Figure 2: Leaf Strategy

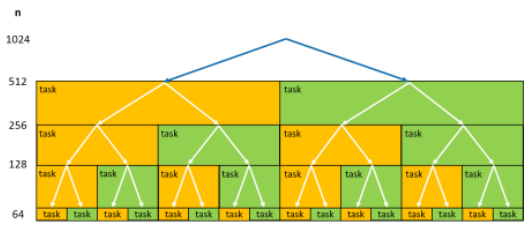

Figure 3: Tree Strategy

2. TASK DECOMPOSITION ANALYSIS FOR MERGE SORT

Using the Tareador tool we will explore potential task decomposition strategies and their implications in terms of parallelism and task interactions required. In order to do that, we will divide this section in two parts: one for the Leaf Strategy and another one for the Tree Strategy.

Leaf strategy

As said in the introduction, this strategy consists in creating tasks for the leaves on the tree. On account of this, we have modified the code of the *multisort-tareador.c* file including the calls related to Tareador in the base cases:

```
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");
    }
}
```

Figure 4: multisort-tareador.c with Leaf Strategy

Executing that code we have obtained the following Task Dependence Graph:

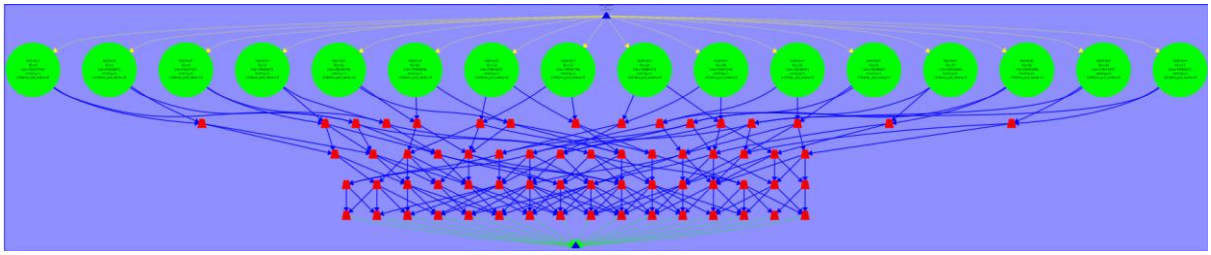

Figure 5: Leaf Strategy TDG

By observing it we can see the creation of the tasks in the leaves. The green circles correspond to the calls to function BasicSort() and there are no dependencies between them. Moreover, the red triangles are calls to function BasicMerge(). There are dependencies between them because this function depends on the previous call to it. That is because we need to have the subvectors that we want to merge sorted.

With the use of Paraver, executing the code with 16 processors we obtain the next timeline:

Figure 6: Leaf Strategy executed with 16 processors

As we can see, the most of the time is spent in the function BasicSort() (sorting the subvectors). But if we zoom into the last part, as we can see in the following image, BasicMerge() functions are called (merging the subvectors) and we can also see the dependencies between them.

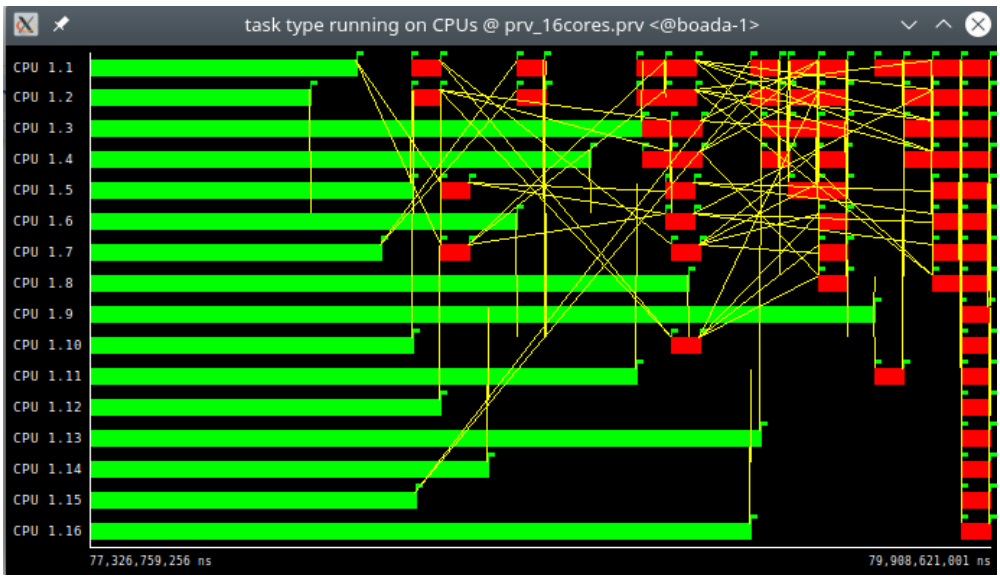

Figure 7: Zoom In into Leaf Strategy executed with 16 processors

Tree strategy

For this strategy, and as said in the introduction, tasks are generated every time a recursive call is performed. In order to achieve that, we have modified the multisort-tareador.c file introducing the tasks related to Tareador in every recursive call in functions merge and multisort as shown in the next figure:

```
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("merge4");
        merge(n, left, right, result, start, length/2);
        tareador_end_task("merge4");

        tareador_start_task("merge5");
        merge(n, left, right, result, start + length/2, length/2);
        tareador_end_task("merge5");
    }
}

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("merge1");
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        tareador_end_task("merge1");
        tareador_start_task("merge2");
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        tareador_end_task("merge2");

        tareador_start_task("merge3");
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        tareador_end_task("merge3");
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Figure 8: multisort-tareador.c with Tree Strategy

Executing that code we have obtained the following Task Dependence Graph:

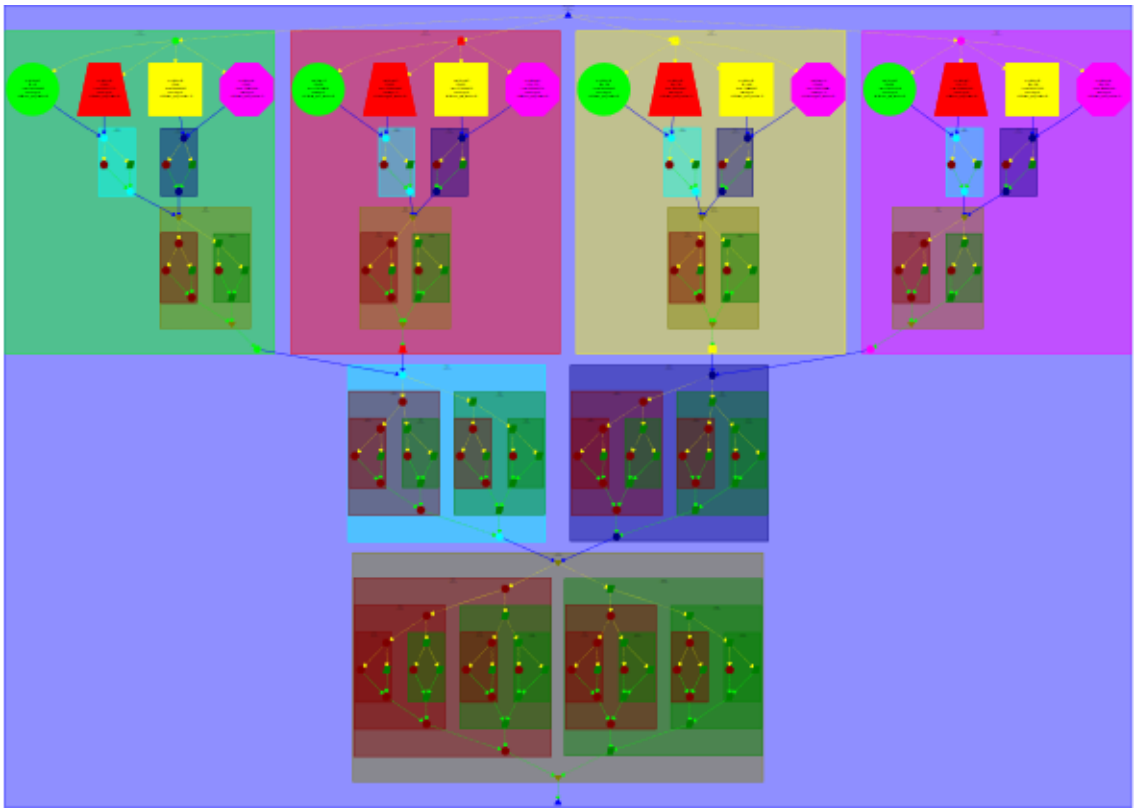

Figure 9: Tree Strategy TDG

As we can see in it, it follows a Tree structure. If we take a look at the code and at the TDG, we can see the 4 multisort recursive calls and then the 3 merge calls inside of them to achieve merging the different subvectors.

With the use of Paraver, executing the code with 16 processors we obtain the following timeline:

Figure 10: Tree Strategy executed with 16 processors

In the following image we can see all the synchronization that requires this strategy when it arrives at the end.

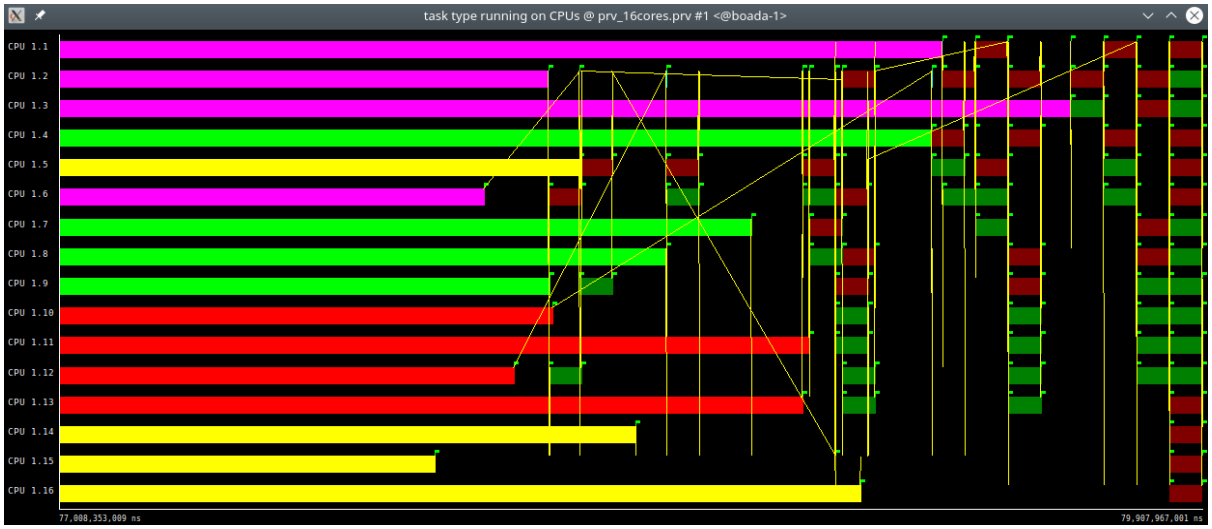

Figure 11: Zoom In into Tree Strategy executed with 16 processors

3. SHARED-MEMORY PARALLELLISATION WITH OpenMP TASKS

The objective of this sessions is to parallelize the code inside *multisort-omp.c* using three types of implementations:

- Leaf Strategy.
- Tree Strategy.
- Tree Strategy with cut-off.

Leaf strategy

In order to implement this strategy, we have modified the previous code for creating tasks in each leaf.

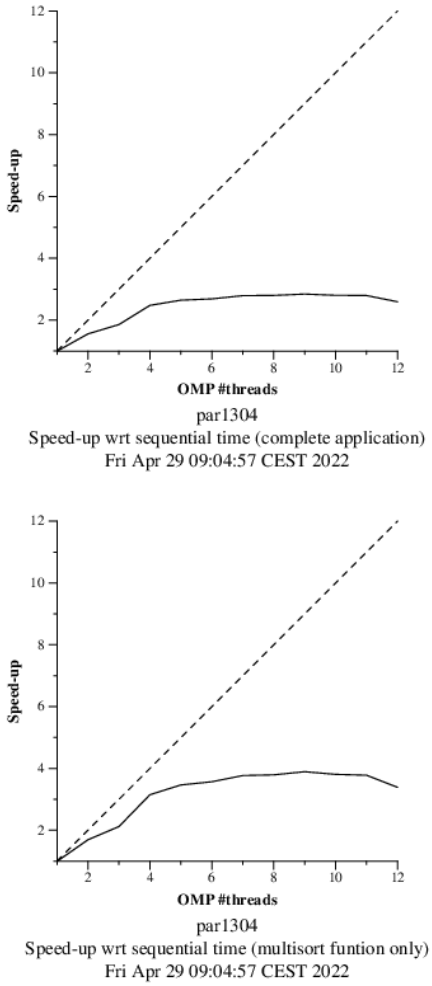

Figure 12: Scalability plots using Leaf Strategy

As we can observe from the previous image we can conclude that this parallelisation implementation is far from being successful. The speed-up isn't good for either of the two plots and if we take a closer look, the multisort function grows in a good way until it arrives at 4 threads. Due to that, the application's speedup suffers from the same problem.
This may be because the Leaf Strategy only uses parallelisation at the end of the program and due to that it cannot take advantage of the parallelisation.

In the following chart, we can see the execution metrics. We can observe that the main problem is that with the increment of processors, the overheads per explicit task grows a lot.

Overview of whole program execution metrics:

Number of processors	1	2	4	8
Elapsed time (sec)	0.55	0.40	0.30	0.31
Speedup	1.00	1.38	1.82	1.77
Efficiency	1.00	0.69	0.46	0.22

Overview of the Efficiency metrics in parallel fraction:

Number of processors	1	2	4	8
Parallel fraction	94.02%			
Global efficiency	82.23%	58.06%	39.38%	18.98%
-- Parallelization strategy efficiency	82.23%	54.25%	36.56%	18.59%
-- Load balancing	100.00%	97.25%	75.52%	43.50%
-- In execution efficiency	82.23%	55.79%	48.41%	42.74%
-- Scalability for computation tasks	100.00%	107.02%	107.71%	102.10%
-- IPC scalability	100.00%	87.27%	85.67%	85.47%
-- Instruction scalability	100.00%	110.94%	112.21%	112.21%
-- Frequency scalability	100.00%	110.54%	112.04%	106.46%

Statistics about explicit tasks in parallel fraction

Number of processors	1	2	4	8
Number of explicit tasks executed (total)	53248.0	53248.0	53248.0	53248.0
LB (number of explicit tasks executed)	1.0	0.66	0.7	0.81
LB (time executing explicit tasks)	1.0	0.72	0.76	0.83
Time per explicit task (average)	4.67	5.27	5.37	5.6
Overhead per explicit task (synch %)	2.92	77.79	200.18	575.81
Overhead per explicit task (sched %)	34.2	42.24	40.46	38.61
Number of taskwait/taskgroup (total)	4095.0	4095.0	4095.0	4095.0

Chart 1: modelfactors.out for Leaf Strategy

In the following images that represent the creation and execution of the explicit tasks we can observe that thread 1 is the only one that creates tasks, while the other threads are working on tasks.

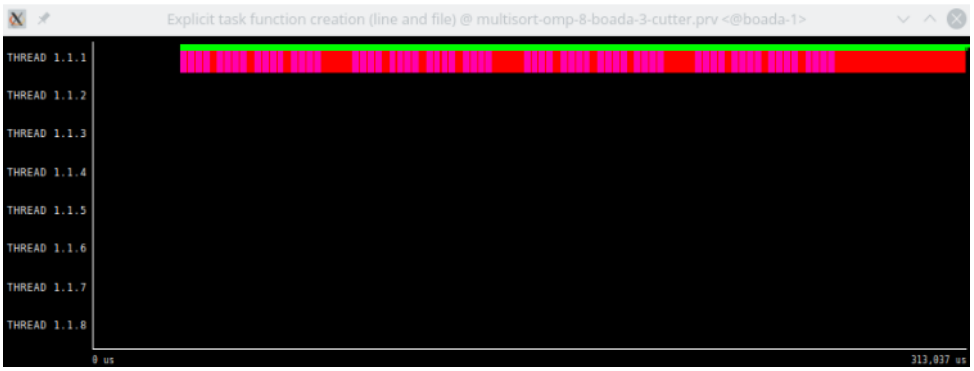

Figure 13: Explicit task function creation for Leaf Strategy

Figure 14: Explicit task function execution for Leaf Strategy

Tree strategy

In order to implement this strategy, we have modified the previous code for creating tasks in each node of the tree, except in the leaves.

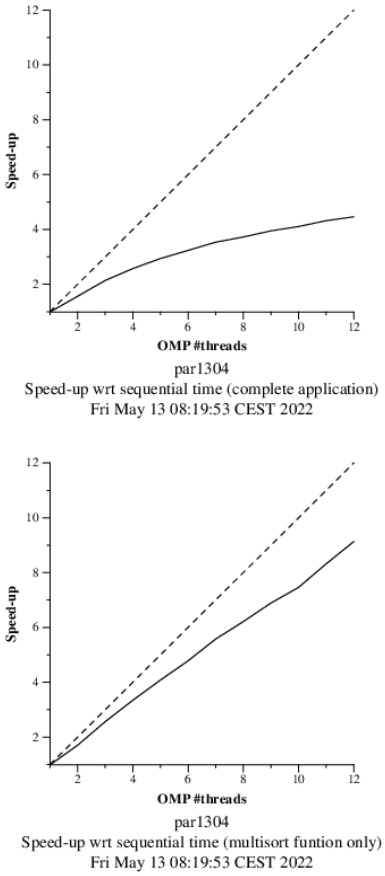

Figure 15: Scalability plots using Tree Strategy

By analyzing these two plots, we can conclude that this strategy is better than the Leaf one. Although it is better than the previous strategy, in the first plot we can see that the whole application's speed-up is still very weak. In contrast, the multisort function has experienced a very great improvement because the speed-up is nearly ideal. This happens because we are parallelizing each recursive call and that implies that more tasks are created and more threads can execute these tasks.

In addition, we will take a look at these Paraver windows:

Figure 16: Explicit task function creation for Tree Strategy

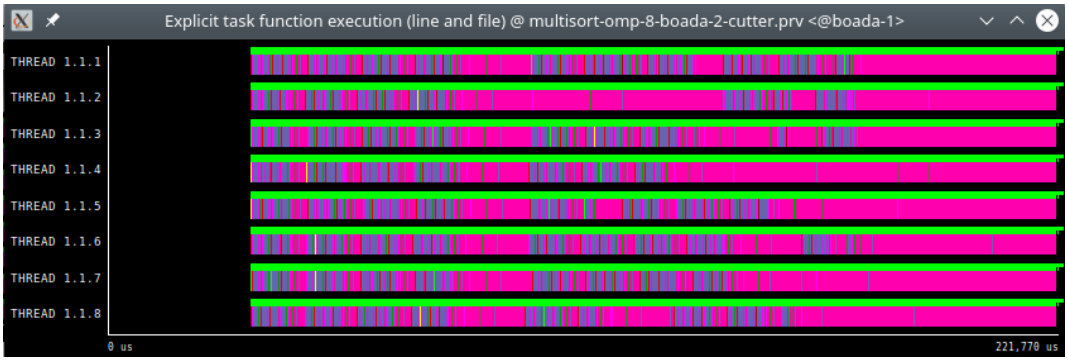

Figure 17: Explicit task function execution for Tree Strategy

In them we can see how all the threads collaborate in the creation and execution of the tasks. Lastly, we can see in the following table that a lot of small tasks are executed and the time of synchronization between them is low.

```
Overview of whole program execution metrics:
================================================================================
   Number of processors |          1 |          2 |          4 |          8
================================================================================
Elapsed time (sec)      |       0.84 |       0.57 |       0.32 |       0.22
Speedup                 |       1.00 |       1.49 |       2.61 |       3.80
Efficiency              |       1.00 |       0.74 |       0.65 |       0.47
================================================================================

Overview of the Efficiency metrics in parallel fraction:
================================================================================================
                  Number of processors |          1 |          2 |          4 |          8
================================================================================================
Parallel fraction                      |     96.12% |
------------------------------------------------------
Global efficiency                      |     79.91% |     60.64% |     55.49% |     42.99%
-- Parallelization strategy efficiency |     79.91% |     52.45% |     48.76% |     41.48%
   -- Load balancing                   |    100.00% |     99.70% |     98.88% |     98.86%
   -- In execution efficiency          |     79.91% |     52.61% |     49.31% |     41.96%
-- Scalability for computation tasks   |    100.00% |    115.63% |    113.80% |    103.64%
   -- IPC scalability                  |    100.00% |     85.47% |     86.28% |     83.39%
   -- Instruction scalability          |    100.00% |    118.64% |    118.77% |    118.82%
   -- Frequency scalability            |    100.00% |    114.03% |    111.06% |    104.59%
================================================================================================

Statistics about explicit tasks in parallel fraction
================================================================================================================================
                           Number of processors |              1 |              2 |              4 |              8
================================================================================================================================
Number of explicit tasks executed (total)       |        99669.0 |        99669.0 |        99669.0 |        99669.0
LB (number of explicit tasks executed)          |            1.0 |            0.99 |            0.98 |            0.97
LB (time executing explicit tasks)              |            1.0 |            1.0 |            0.99 |            0.99
Time per explicit task (average)                |            4.71 |            7.49 |            8.01 |            9.83
Overhead per explicit task (synch %)            |            2.01 |           40.36 |           43.54 |           50.97
Overhead per explicit task (sched %)            |           32.62 |           27.54 |           31.27 |           38.81
Number of taskwait/taskgroup (total)            |         2730.0 |         2730.0 |         2730.0 |         2730.0
================================================================================================================================
```

Chart 2: modelfactors.out for Tree Strategy

Tree Strategy with cut-off

To implement this strategy, we have modified the previous code in order to create a cut-off mechanism that controls the number of leaves in the recursion tree that are executed by computational tasks and to increase task granularity.

When we had the code implemented, we submitted the *submit-strong-extrae.sh* script to check the number of tasks generated when using different values for the cut–off level: 0, 1, 2, 4 and 8 (we only put the 0, 1, and 8 examples):

- CUTOFF = 0

In the following table generated by the script, we can see in the third chart that the number of explicit tasks executed is 7. This is what we expect because when the program arrives at the multisort function it generates 7 tasks and then, because of CUTOFF = 0, it doesn't generate any more.

Overview of whole program execution metrics:

Number of processors	1	2	4	8
Elapsed time (sec)	0.20	0.12	0.08	0.08
Speedup	1.00	1.65	2.39	2.33
Efficiency	1.00	0.82	0.60	0.29

Overview of the Efficiency metrics in parallel fraction:

Number of processors	1	2	4	8
Parallel fraction	83.40%			
Global efficiency	99.94%	94.54%	82.39%	39.58%
-- Parallelization strategy efficiency	99.94%	95.04%	83.21%	41.80%
-- Load balancing	100.00%	95.33%	84.58%	41.97%
-- In execution efficiency	99.94%	99.70%	98.37%	99.60%
-- Scalability for computation tasks	100.00%	99.47%	99.02%	94.69%
-- IPC scalability	100.00%	99.67%	99.48%	99.52%
-- Instruction scalability	100.00%	100.00%	99.99%	99.99%
-- Frequency scalability	100.00%	99.80%	99.54%	95.17%

Statistics about explicit tasks in parallel fraction

Number of processors	1	2	4	8
Number of explicit tasks executed (total)	7.0	7.0	7.0	7.0
LB (number of explicit tasks executed)	1.0	0.88	0.58	0.47
LB (time executing explicit tasks)	1.0	0.95	0.85	0.67
Time per explicit task (average)	23258.28	23386.94	23487.97	24554.42
Overhead per explicit task (synch %)	0.02	5.16	20.1	139.19
Overhead per explicit task (sched %)	0.03	0.03	0.04	0.05
Number of taskwait/taskgroup (total)	3.0	3.0	3.0	3.0

Chart 3: modelfactors.out for Tree Strategy with Cutoff = 0

We can see it better by taking a look at the explicit task function creation. The first thread creates all the tasks and the other threads execute them.

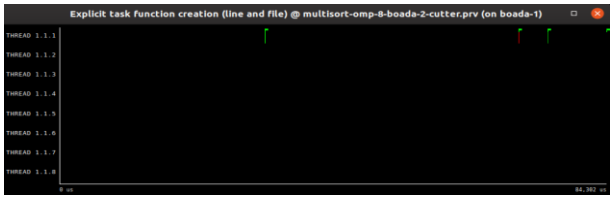

Figure 18: Explicit task function creation for Tree Strategy with Cutoff = 0

Figure 19: Explicit task function execution for Tree Strategy with Cutoff = 0

- CUTOFF = 1

Using this CUTOFF, we can see that the number of explicit tasks executed is 41. This value is also what we expected.

Overview of whole program execution metrics:

Number of processors	1	2	4	8
Elapsed time (sec)	0.19	0.12	0.08	0.06
Speedup	1.00	1.68	2.57	3.21
Efficiency	1.00	0.84	0.64	0.40

Overview of the Efficiency metrics in parallel fraction:

Number of processors	1	2	4	8
Parallel fraction	83.90%			
Global efficiency	99.89%	98.18%	93.93%	71.26%
-- Parallelization strategy efficiency	99.89%	99.46%	94.63%	76.03%
-- Load balancing	100.00%	99.81%	95.17%	83.60%
-- In execution efficiency	99.89%	99.65%	99.44%	90.94%
-- Scalability for computation tasks	100.00%	98.71%	99.25%	93.73%
-- IPC scalability	100.00%	98.67%	99.58%	98.99%
-- Instruction scalability	100.00%	100.01%	100.01%	100.00%
-- Frequency scalability	100.00%	100.04%	99.66%	94.69%

Statistics about explicit tasks in parallel fraction

Number of processors	1	2	4	8
Number of explicit tasks executed (total)	41.0	41.0	41.0	41.0
LB (number of explicit tasks executed)	1.0	0.98	0.93	0.73
LB (time executing explicit tasks)	1.0	1.0	0.98	0.84
Time per explicit task (average)	3974.38	4032.33	4109.2	4464.06
Overhead per explicit task (synch %)	0.03	0.43	5.37	29.7
Overhead per explicit task (sched %)	0.07	0.08	0.11	0.19
Number of taskwait/taskgroup (total)	18.0	18.0	18.0	18.0

Chart 4: modelfactors.out for Tree Strategy with Cutoff = 1

We can see it better by taking a look at the explicit task function creation and execution. In them we can observe how all the threads are now creating and executing the explicit tasks.

Figure 20: Explicit task function creation for Tree Strategy with Cutoff = 1

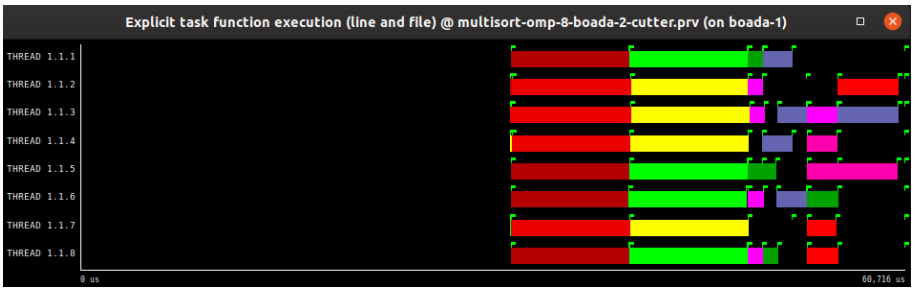

Figure 21: Explicit task function execution for Tree Strategy with Cutoff = 1

- CUTOFF = 8

Overview of whole program execution metrics:

Number of processors	1	2	4	8
Elapsed time (sec)	0.66	0.40	0.23	0.16
Speedup	1.00	1.63	2.80	4.23
Efficiency	1.00	0.82	0.70	0.53

Overview of the Efficiency metrics in parallel fraction:

Number of processors	1	2	4	8
Parallel fraction	95.05%			
Global efficiency	77.90%	65.68%	60.08%	49.79%
-- Parallelization strategy efficiency	77.90%	58.51%	56.23%	49.36%
-- Load balancing	100.00%	99.95%	99.26%	98.58%
-- In execution efficiency	77.90%	58.54%	56.65%	50.08%
-- Scalability for computation tasks	100.00%	112.26%	106.85%	100.86%
-- IPC scalability	100.00%	91.56%	86.81%	87.23%
-- Instruction scalability	100.00%	111.84%	111.92%	111.97%
-- Frequency scalability	100.00%	109.63%	109.98%	103.27%

Statistics about explicit tasks in parallel fraction

Number of processors	1	2	4	8
Number of explicit tasks executed (total)	57173.0	57173.0	57173.0	57173.0
LB (number of explicit tasks executed)	1.0	1.0	0.99	0.98
LB (time executing explicit tasks)	1.0	1.0	1.0	0.99
Time per explicit task (average)	5.84	9.24	10.15	11.95
Overhead per explicit task (synch %)	14.33	37.27	37.82	44.01
Overhead per explicit task (sched %)	26.97	20.84	23.19	28.32
Number of taskwait/taskgroup (total)	27904.0	27904.0	27904.0	27904.0

Chart 5: modelfactors.out for Tree Strategy with Cutoff = 8

Lastly, using this CUTOFF, we can see that the number of explicit tasks executed is 57173. Also, we can observe that the elapsed time is higher than the ones obtained with CUTOFF = 0 and CUTOFF = 1. Looking at the following image we can see that because of having a lot of tasks, there is a lot of time spent in overheads of synchronization.

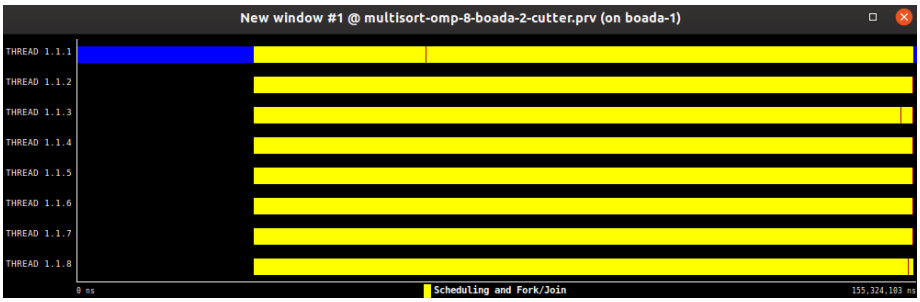

Figure 22: Paraver execution for Tree Strategy with Cutoff = 8

In order to analyze the best CUTOFF size, we have submitted the *submit-cutoff-omp.sh* and we have obtained the following graphic. We can see that the best size for CUTOFF is 7, so we will establish that number as the optimum value for the CUTOFF.

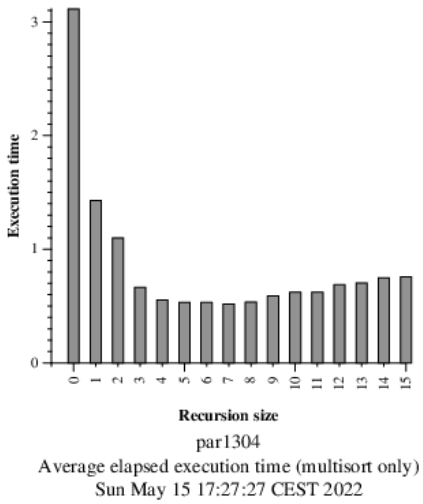

Figure 23: Cutoff plot showing Execution time per Cutoff

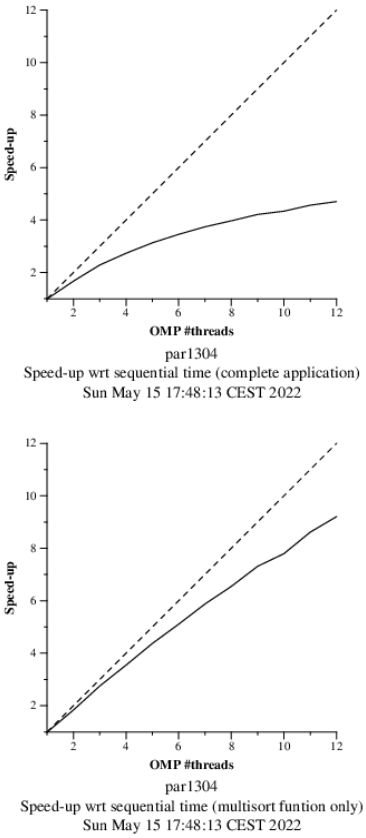

Figure 24: Scalability plot for Cutoff = 7

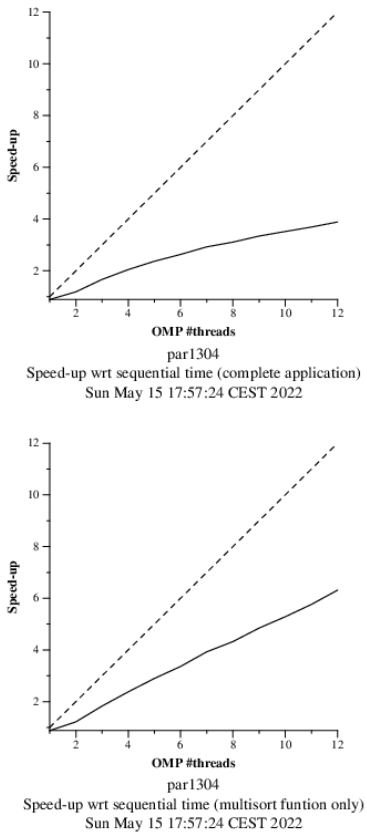

Figure 25: Scalability plot for Cutoff = 16

In the images above we have two plots that represent the scalability of the program when the values of **sortsize** and **mergesize** are set to 128.

On the one hand, in the left image we can see the plots obtained when the CUTOFF is set to the optimum value, the one obtained before, that is 7.

On the other hand, the right image shows us the plots obtained when the CUTOFF is set to maximum value, 16.

By taking a look and analyzing the plots we can observe that, although when we have the cut-off set to maximum we obtain a speed-up that is not so bad, using the optimal CUTOFF size gives us a better speed-up. In addition, when we are using the optimal CUTOFF size, we obtain a plot in the multisort function that is very close to the ideal case and, because of that, we can conclude that we have done a good job.

Optional 1

For this section, we will change the **NP_max** variable's value from 12 to 24 (it's double than the ones available on the Boada). We have obtained the following plots.

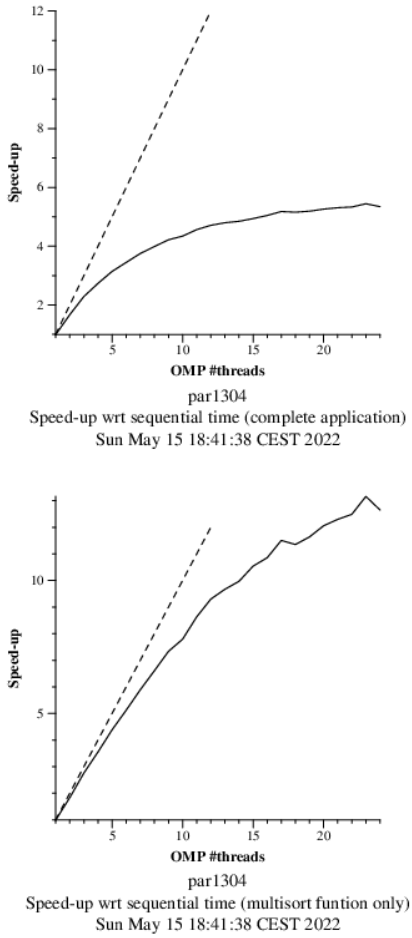

Figure 26: Scalability plot for Cutoff = 7 (NP_max = 24)

We can see that the speed-up doesn't get stuck at 12 threads (as it should because of physical constraints) and it continues growing. Moreover, we can add that although it continues growing, it's not as fast as before.

4. USING OpenMP TASK DEPENDENCIES

In this last session, we will make changes to the latest version of the multisort-omp.c code, so that we avoid some taskwait or taskgroup synchronizations using dependencies. For this, we have added the depend clause immediately after the cutoff clause (for which we are using the value 7, which we saw in the previous session was the optimal value for this variable).

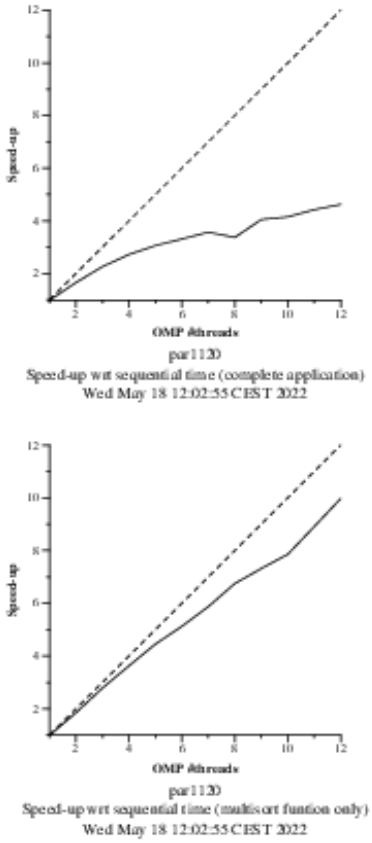

Figure 27: Scalability plot for Cutoff = 7

As we can see, the graphics are very similar in the 2 versions, although this new version with dependencies presents a slight improvement, since it spends less time in synchronizations. The reason why there is not a significant improvement is because with only the depend clause we do not ensure the correct operation of multisort. For it, we have to add a taskwait at the end of the same one, so that to make a multisort, we make sure that the previous one has finished. Therefore, as we will need a taskwait at the end, the improvement is minimal.

We can conclude that the version with the depend clause is not worth it, as it has little improvement over the complexity of parallelizing the code. The version with taskwaits works a little worse, but it is much simpler to program. Now we will use paraver to observe in detail how multisort works.

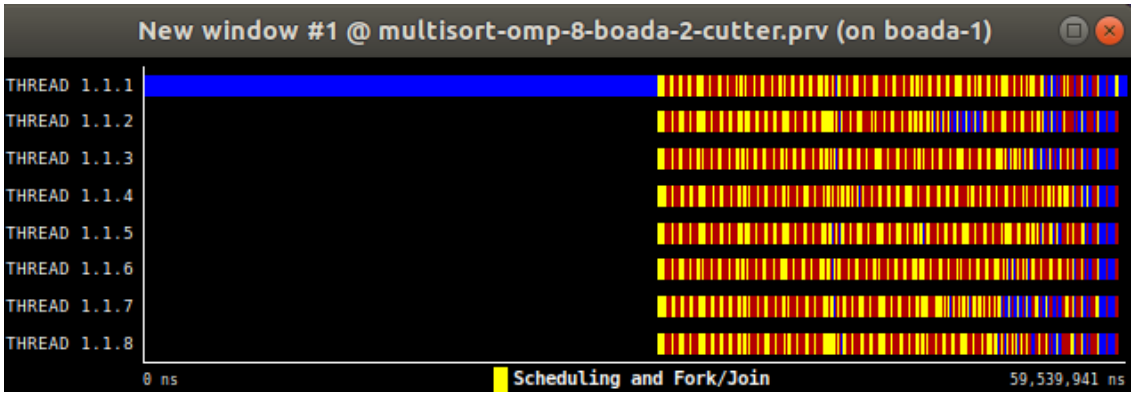

Figure 28: Paraver execution with 8 threads.

As we can see in the image, the operation is very similar to the version of the code without the depend clause. But, in this second version, we observe that, although the program performs a little better, it spends much more time in synchronization due to the dependencies created between functions.

Optional 2

In this optional section we are asked to parallelize the functions that initialize the data and tmp vectors, and compare the performance of the program with those versions seen so far.

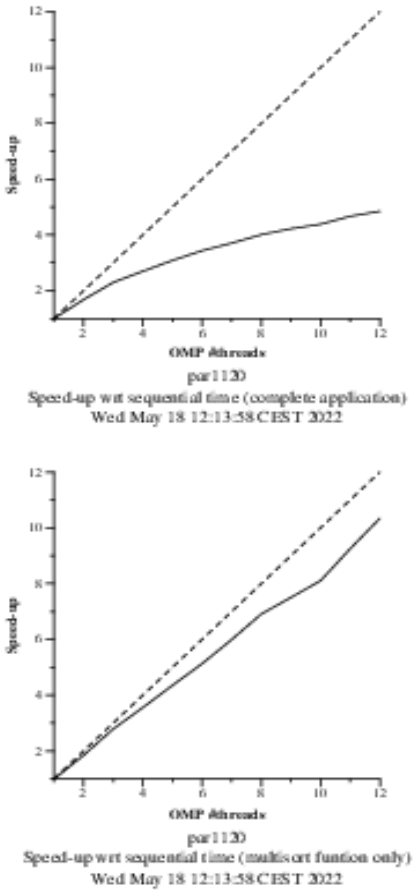

Figure 29: Scalability plot for Cutoff = 7

We can see that, by parallelizing the functions that initialize the data and tmp vectors, the scalability of our program improves. This can be seen in that the slope of the graph is slightly steeper. We can also see that its performance improves with 8 threads. In addition, the graph evolves more linearly. This improvement is due to the fact that the initialization of a vector can be a tedious task if it is done sequentially, so, by improving the performance of this part of the program, we get a higher overall speedup.

5. CONCLUSIONS

Throughout these lab sessions, we have learned 2 task generation techniques, the tree strategy and the leaf strategy.

As we have seen in session 2, the tree strategy presents fewer dependencies, as the tasks are declared in a transversal way. Therefore, this strategy is better for parallelizing processes.

Investigating more in depth about the tree strategy and its possible improvements, we have seen that there is a technique called cutoff, which can greatly improve the scalability of the program, by limiting the number of tasks that we create.

We have also made a version of the code using the depend clause just after the cutoff, but we have seen that the improvement is very little for how complex the parallelization of the code becomes.

As an optional part, we have seen that parallelizing the vector initialization process has a very positive impact on program scalability, since this task, if done sequentially, can be quite detrimental to program performance.

In summary, we have seen different task generation techniques and different ways to improve the performance of the tree strategy in particular. Within all the code versions tested, it is worth noting that sometimes it is not worth taking the code to too high levels of complexity if the improvement obtained is going to be small.