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

GUILLEM GRÀCIA ANDREU, GERARD MADRID MIRÓ

TARDOR 2021-22

PAR3109

1. Task decomposition analysis for Mergesort

1.1 Divide and conquer

```
************************************************************
Problem size (in number of elements): N=32768,
MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
************************************************************
*
Initialization time in seconds: 0.859802
Multisort execution time: 6.324475
Check sorted data execution time: 0.015344
Multisort program finished
************************************************************
*
```

Figure 1: Sequential execution output

After compiling the sequential version of the program given and executing the binary received with the correct arguments to determine the size, we got the time values shown in *Figure 1*. Although these values might not seem relevant at the moment, they will be used as the base for comparison, as they are the plain times of a sequential execution.

1.2 Task decomposition analysis with Tareador

In this section of the lab assignment, we had to use the Tareador tool to investigate different decomposition strategies. We will be mainly focusing on two different strategies: **Leaf** and **Tree**.

1.2.1 Leaf strategy

```
void merge(long n, T left[n], T right[n], T result[n*2],
           long start, long length) {
   if (length < MIN_MERGE_SIZE*2L) {
       tareador_start_task("BasicMergeLeaf");
       basicmerge(n, left, right, result, start, length);
       tareador_end_task("BasicMergeLeaf");
   }
   ...
}

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

```
    ...
    else {
        tareador_start_task("BasicSortLeaf");
        basicsort(n, data);
        tareador_end_task("BasicSortLeaf");
    }
}
```

Figure 2: Leaf strategy Tareador code

In order to properly use the Tareador tool, we had to insert into our code starting and ending functions to tasks. We can see in *Figure* 2 how that is done to both `basic merge` and `basic sort` and how the tasks are started and ended before and after the call to action of the function.

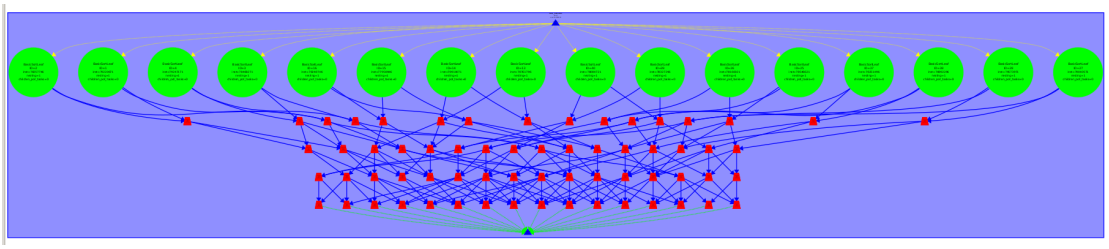


Figure 3: Leaf strategy decomposition task

Once the setup is done, we can now compile and execute the code to obtain a task dependence graph (TDG) that can be used to analyze the code. The case used in this execution is quite small so as to display a comprehensible TDG.

In this new image (*Figure* 3), we can see how tasks are created. First, we see how the problem size is divided into BasicSorts. Then, these are merged with other sorted pairs in order to start building the sorted array. As we progress lower into the graph, we can see how the merges depend on one another, and finally are all built into one array.

1.2.2 Tree strategy

```
void merge(long n, T left[n], T right[n], T result[n*2], long start,
long length) {
   ...
   else {
       // Recursive decomposition
       tareador_start_task("MergeTree0.M");
       merge(...);
       tareador_end_task("MergeTree0.M");
       ...
   }
}
void multisort(long n, T data[n], T tmp[n]) {
   if (n >= MIN_SORT_SIZE*4L) {
       // Recursive decomposition
       tareador_start_task("MultisortN");
       multisort(...);
       tareador_end_task("MultisortN");
       ...

       tareador_start_task("MergeTreeM");
       merge(...);
       tareador_end_task("MergeTreeM");
       ...
   }
       ...
}
```

Figure 4: Tree strategy tareador code

To adapt the code in order to implement the Tree Strategy we had to set a few more instructions, now in the recursive calls of the functions instead of the simple cases as we can see in the *Figure 4*.

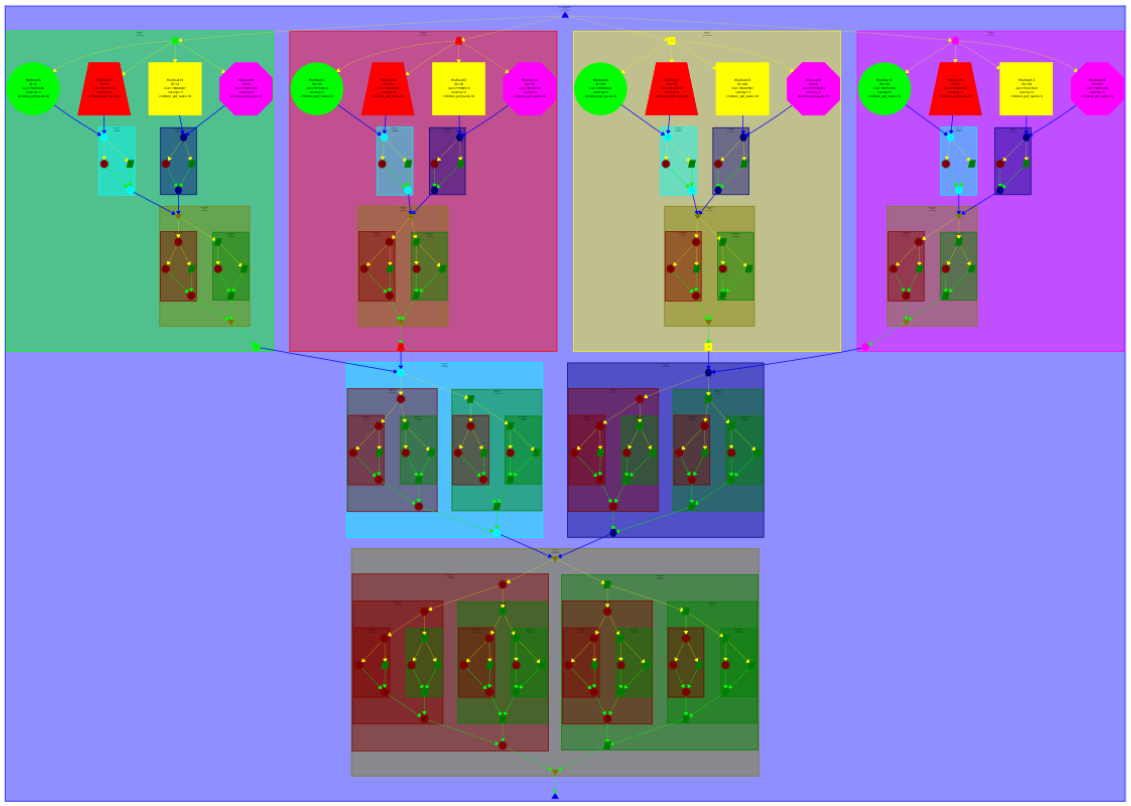


Figure 5: Tree strategy decomposition task

Once the code was fully adapted, we proceeded to compile it and we obtained the dependence graph seen in *Figure 5*. We've observed that first of all the multisort is generating smaller packets of data to be sorted and later merged. Precisely, each multisort is divided into 4 packets. When the size is small enough it begins to merge the first two packets, then the two last packets, and finally the result of the other two merges (recursively).

1.2.3 Leaf Strategy VS Tree Strategy

To properly find how the two strategies differ, we had to move to a different program. This time we used Paraver to see the task creation in a clearer way.

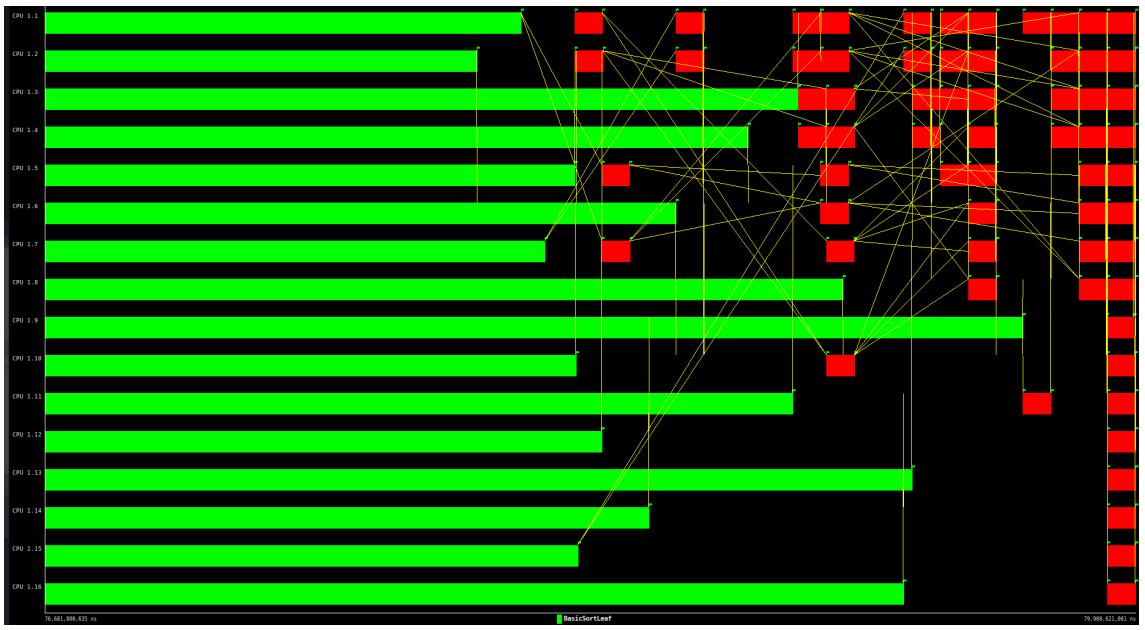


Figure 6: Leaf strategy simulation

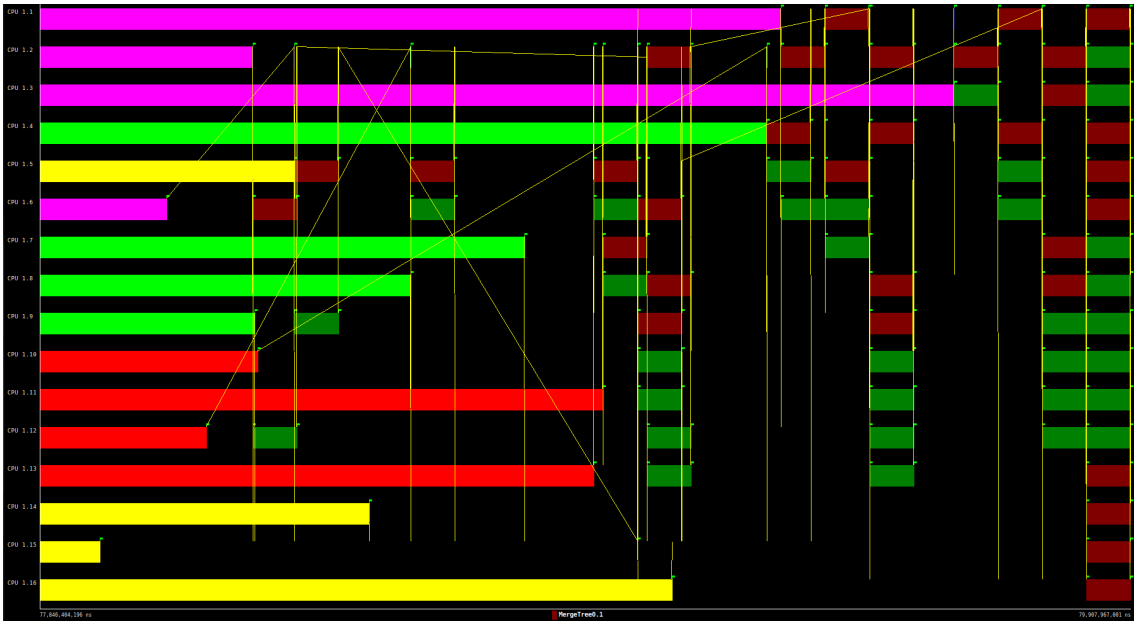


Figure 7: Tree strategy simulation

In these two simulations, after zooming in to specific parts of the program execution, we can see the difference between both simulations. The rectangles are execution lines whereas the yellow lines are dependencies between them.

2. Shared-memory parallelisation with OpenMP tasks

2.1 Leaf strategy in OpenMP

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

Figure 8: Leaf strategy OpenMP code

In order to apply leaf strategy to our code, we added different tasks and taskwaits in the appropriate locations (as seen in *Figure* 8). The taskwaits and their barriers are there to wait for all tasks to end before actually merging, since we don't want to merge an unsorted part of the array.

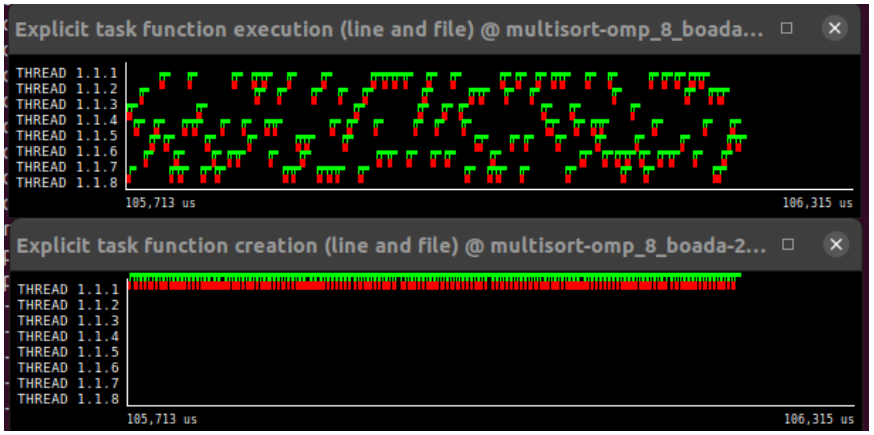


Figure 9: Leaf strategy explicit tasks

The image shown above (*Figure* 9), provides a clear explanation as to why this program is not performing as well as expected. First of all, the thread creating all tasks is creating them sequentially and too slow. This alone is the recipe for disaster, since threads executing tasks are way faster and can end the task before a new one is created.

2D thread state profile @ multisort-omp_8_boada-2.prv (on boada-1)

Default

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	89.89 %	-	2.11 %	7.39 %	0.61 %	0.00 %
THREAD 1.1.2	6.48 %	47.04 %	46.21 %	0.01 %	0.26 %	-
THREAD 1.1.3	6.66 %	47.04 %	46.07 %	0.01 %	0.23 %	-
THREAD 1.1.4	6.50 %	47.05 %	46.20 %	0.01 %	0.24 %	-
THREAD 1.1.5	6.67 %	47.03 %	46.04 %	0.01 %	0.24 %	-
THREAD 1.1.6	6.70 %	47.11 %	45.93 %	0.00 %	0.26 %	-
THREAD 1.1.7	6.65 %	47.10 %	46.00 %	0.00 %	0.24 %	-
THREAD 1.1.8	6.56 %	47.14 %	46.05 %	0.00 %	0.25 %	-
Total	136.10 %	329.50 %	324.62 %	7.44 %	2.34 %	0.00 %
Average	17.01 %	47.07 %	40.58 %	0.93 %	0.29 %	0.00 %
Maximum	89.89 %	47.14 %	46.21 %	7.39 %	0.61 %	0.00 %
Minimum	6.48 %	47.03 %	2.11 %	0.00 %	0.23 %	0.00 %
StDev	27.55 %	0.04 %	14.54 %	2.44 %	0.12 %	0 %
Avg/Max	0.19	1.00	0.88	0.13	0.48	1

Figure 10: Leaf strategy state profile

The state profile shown in *Figure* 10 provides a similar explanation as before, seeing how half of the time is spent in synchronization.

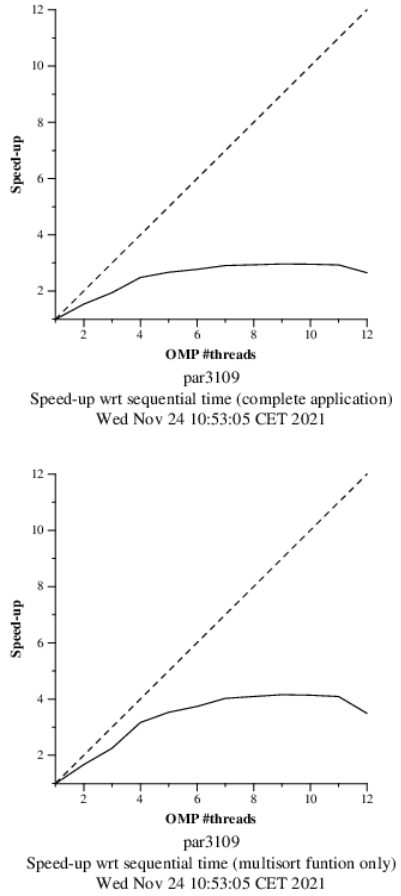


Figure 11: Leaf strategy strong scalability

As we can see in *Figure* 11, the scalability obtained is nowhere near what we expect when parallelizing a code. We can see that it doesn't follow the expected x=y line and it even goes down after surpassing 11 threads.

2.2 Tree strategy in OpenMP

```
void merge(long n, T left[n], T right[n], T result[n*2], long start,
           long length) {
   if (length < MIN_MERGE_SIZE*2L) {
       basicmerge(n, left, right, result, start, length);
   } else {
   #pragma omp task
       merge(n, left, right, result, start, length/2);
       ...
   }
}
void multisort(long n, T data[n], T tmp[n]) {
   if (n >= MIN_SORT_SIZE*4L) {
   #pragma omp taskgroup
   {
   #pragma omp task
       multisort(n/4L, &data[0], &tmp[0]);
   ...
   }
   #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 taskgroup
   {
   #pragma omp task
       merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
   }
   } else {
       basicsort(n, data);
   }
}
```

Figure 12: Tree strategy OpenMP Code

To apply tree strategy to our code, we now added task groups with an implicit barrier in order to manage the needed waiting of tasks (shown in *Figure* 12). These taskgroups grouped all multisorts and two of them were used to group the two sections of merges.

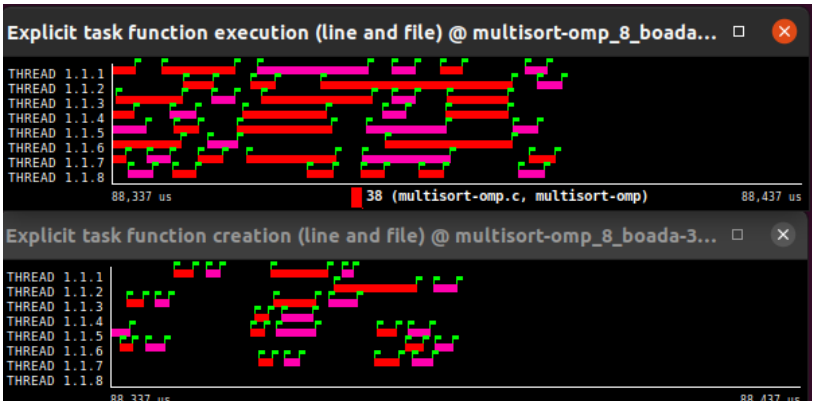


Figure 13: Tree strategy explicit tasks

The image shown above (*Figure* 13), provides a clear explanation as to why this program is performing better than leaf strategy. We can see that there are now more threads creating tasks and the tasks are longer than before, so it doesn't take that much time synchronizing data.

2D thread state profile @ multisort-omp_8_boada-3.prv (on boada-1)

Default

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	91.09 %	-	4.68 %	3.93 %	0.29 %	0.00 %
THREAD 1.1.2	17.11 %	53.77 %	16.24 %	11.80 %	1.08 %	-
THREAD 1.1.3	17.09 %	53.81 %	16.12 %	11.96 %	1.02 %	-
THREAD 1.1.4	16.69 %	53.74 %	15.86 %	12.63 %	1.08 %	-
THREAD 1.1.5	17.07 %	53.81 %	16.15 %	11.91 %	1.07 %	-
THREAD 1.1.6	17.03 %	53.86 %	15.95 %	12.05 %	1.11 %	-
THREAD 1.1.7	17.25 %	53.88 %	15.99 %	11.84 %	1.04 %	-
THREAD 1.1.8	17.11 %	53.93 %	15.77 %	12.10 %	1.09 %	-
Total	210.44 %	376.81 %	116.76 %	88.22 %	7.77 %	0.00 %
Average	26.31 %	53.83 %	14.59 %	11.03 %	0.97 %	0.00 %
Maximum	91.09 %	53.93 %	16.24 %	12.63 %	1.11 %	0.00 %
Minimum	16.69 %	53.74 %	4.68 %	3.93 %	0.29 %	0.00 %
StDev	24.49 %	0.06 %	3.75 %	2.69 %	0.26 %	0 %
Avg/Max	0.29	1.00	0.90	0.87	0.88	1

Figure 14: Tree strategy state profile

We can see in *Figure* 14 that now all threads spend more time running and don't take much time synchronizing but we think it could be improved further with some more optimizations.

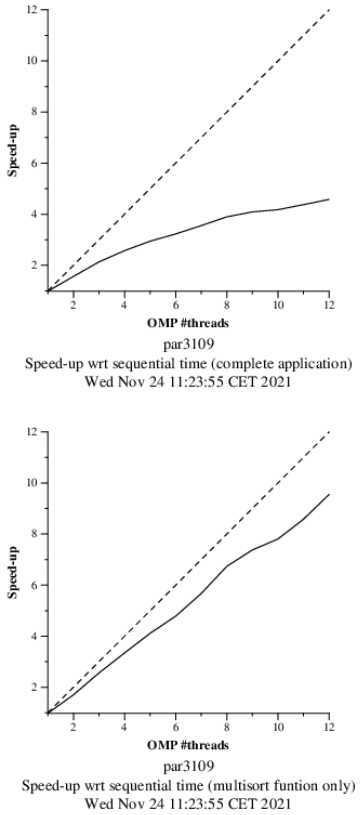


Figure 15: Tree strategy strong scalability

As we can see in *Figure* 15, the scalability obtained is much better than before. Now the data obtained is very close to the x=y line expected.

2.3 Task granularity control: the cut–off mechanism

```
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) {
       basicmerge(n, left, right, result, start, length);
   } else {
   if(!omp_in_final()) {

       #pragma omp task final(depth >= CUTOFF)
       merge(n, left, right, result, start, length/2, depth+1);
       ...
   }
   else {
       merge(n, left, right, result, start, length/2, depth+1);
       ...
   }
   }
}
void multisort(long n, T data[n], T tmp[n], int depth) {
   if (n >= MIN_SORT_SIZE*4L) {
   if(!omp_in_final()){
       #pragma omp taskgroup{
       #pragma omp task final(depth >= CUTOFF)
       multisort(n/4L, &data[0], &tmp[0], depth+1);
       ...
       }
       #pragma omp taskgroup {
       #pragma omp task final(depth >= CUTOFF)
       merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);
       ...
       }
       #pragma omp taskgroup {
       #pragma omp task final(depth >= CUTOFF)
       merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);
       }
   } else{
       multisort(n/4L, &data[0], &tmp[0], 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);
    }
    }
    ...
}
```

Figure 16: Tree strategy cut-off

In this section, we are presented with a new mechanism to maintain control in the number of leaves. In order to apply this in the tree strategy, we had to add a new variable to the recursion in order to count how deep we are (as can be seen in *Figure* 16). This variable is called depth and we add one to it every time we call a recursive function.

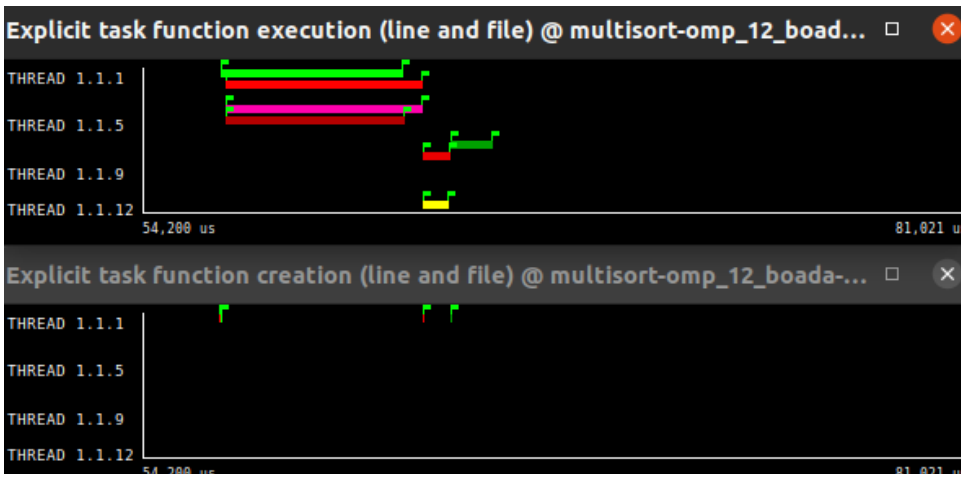


Figure 17: Tree strategy explicit tasks (cut-off = 0)

Figure 18: Tree strategy explicit tasks (cut-off = 1)

In these two figures (17 and 18), we can see the difference between cutoff set to 0 and it set to 1. The main difference is the number of tasks created and executed which is clearly higher in *Figure* 18 opposed to *Figure* 17.

2D thread state profile @ multisort-omp_12_boada-4.prv (on boada-1)

Default

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	98.51 %	-	0.84 %	0.64 %	0.01 %	0.00 %
THREAD 1.1.2	3.43 %	93.15 %	3.37 %	0.04 %	0.01 %	-
THREAD 1.1.3	4.82 %	93.12 %	1.97 %	0.07 %	0.01 %	-
THREAD 1.1.4	3.99 %	93.15 %	2.85 %	0.01 %	0.01 %	-
THREAD 1.1.5	4.59 %	93.15 %	2.25 %	0.00 %	0.01 %	-
THREAD 1.1.6	2.90 %	93.27 %	3.82 %	0.01 %	0.01 %	-
THREAD 1.1.7	2.79 %	93.30 %	3.79 %	0.11 %	0.01 %	-
THREAD 1.1.8	3.22 %	93.28 %	3.41 %	0.08 %	0.02 %	-
THREAD 1.1.9	3.79 %	93.14 %	3.06 %	0.00 %	0.01 %	-
THREAD 1.1.10	2.91 %	93.27 %	3.75 %	0.05 %	0.01 %	-
THREAD 1.1.11	4.95 %	93.22 %	1.82 %	0.00 %	0.01 %	-
THREAD 1.1.12	3.50 %	93.37 %	3.11 %	0.01 %	0.01 %	-
Total	139.38 %	1,025.40 %	34.04 %	1.02 %	0.15 %	0.00 %
Average	11.62 %	93.22 %	2.84 %	0.09 %	0.01 %	0.00 %
Maximum	98.51 %	93.37 %	3.82 %	0.64 %	0.02 %	0.00 %
Minimum	2.79 %	93.12 %	0.84 %	0.00 %	0.01 %	0.00 %
StDev	26.21 %	0.08 %	0.89 %	0.17 %	0.00 %	0 %
Avg/Max	0.12	1.00	0.74	0.13	0.80	1

Figure 19: Tree strategy state profile (cut-off = 1)

In order to see how well our new code was performing, we checked the paraver hint “thread state profiles”. We can see that in comparison with *Figure* 12, the current one (*Figure* 19) spends from four to ten times less in synchronization.

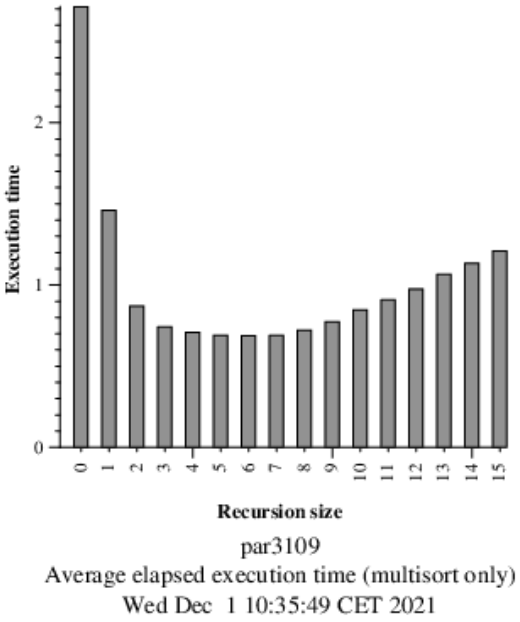


Figure 20: Tree strategy comparison different cut-off values execution times

From this image (*Figure* 20), we can extract the most efficient values to which we should set up our cutoff in order to reduce the execution time. We concluded that the most optimal would be 6.

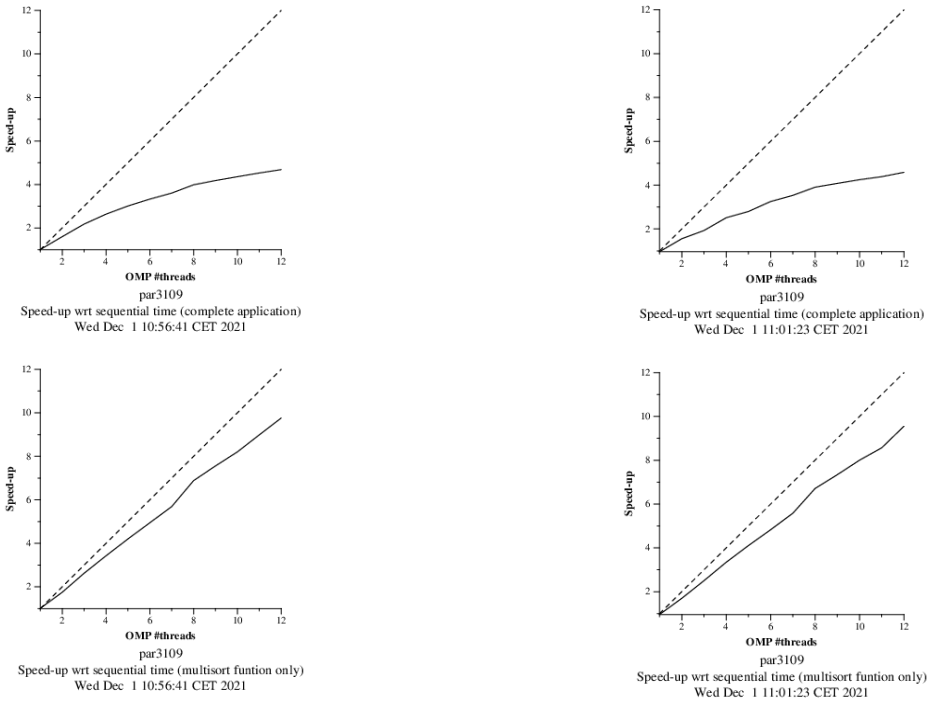


Figure 21: Comparison between default (left) and optimal (right) cut-off values

As we can see in *Figure* 21 above, there is no much difference between the cut-off with the optimal value and the default one but the optimal one shows a spike at the end of the graphic (around 11-12 threads) that shows us that it's better for a big number of threads, but it's basically the same otherwise.

Optional 1

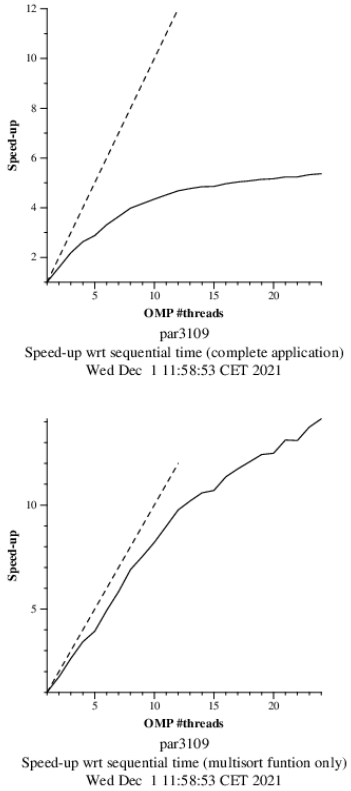


Figure 22: Strong scalability up to 24 threads

After testing the tree strategy and assigning 24 threads to the execution by changing the np_MAX variable in the submit-strong-omp script, we are presented with a constantly growing speed-up (shown in *Figure* 22). We can see how it doesn't stop after 12 threads and it even gets a better performance. We think that the possibility to obtain these results with 12 physical threads lays on the fact that threads can spend less time in synchronization.

3. Using OpenMP task dependencies

```
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) {
       basicmerge(n, left, right, result, start, length);
   } else {
   if(!omp_in_final()) {
       #pragma omp task final(depth >= CUTOFF)
       merge(n, left, right, result, start, length/2, depth+1);
       #pragma omp task final(depth >= CUTOFF)
       merge(n, left, right, result, start + length/2, length/2, depth+1);
   }
   else {
       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) {
   if(!omp_in_final()){
       #pragma omp taskgroup
       {
           #pragma omp task final(depth >= CUTOFF) depend(out: data[0])
           multisort(n/4L, &data[0], &tmp[0], depth+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, depth+1);
           #pragma omp task 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 taskgroup
       {
           #pragma omp task depend(in: tmp[0], tmp[n/2L]) depend(out: data[0])
```

```
            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);
        }
    }
    else{
        multisort(n/4L, &data[0], &tmp[0], 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 {
        basicsort(n, data);
    }
}
```

Figure 23: Task dependencies OpenMP code

In order to avoid some taskgroups we can add dependencies between tasks. We can do that by adding the directive depend(in: input variable dependencies, out: output variables dependencies) in every task that we want to generate (as can be seen in *Figure* 23).
By doing this, we can merge the taskgroup of the multisort calls and the 2 first merge calls in one big taskgroup instead of two.

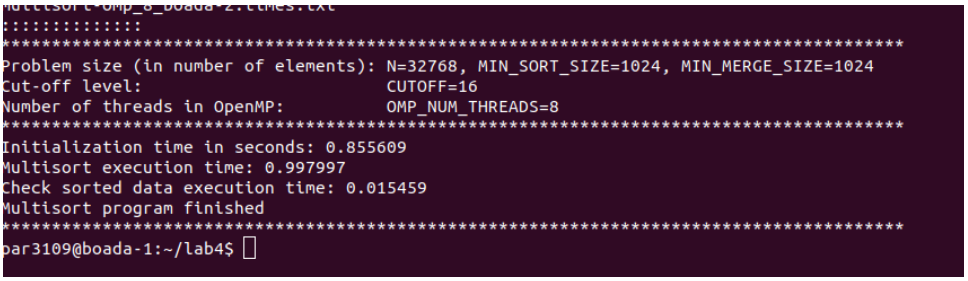


Figure 24: Task dependencies execution works

The picture above (*Figure* 24) shows how the execution is working and doesn't throw any error whatsoever. We can take a moment to realize how much our program has improved by looking at *Figure* 1, and seeing how the multisort time there was over 6 seconds and it doesn't get to 1 second in this one.

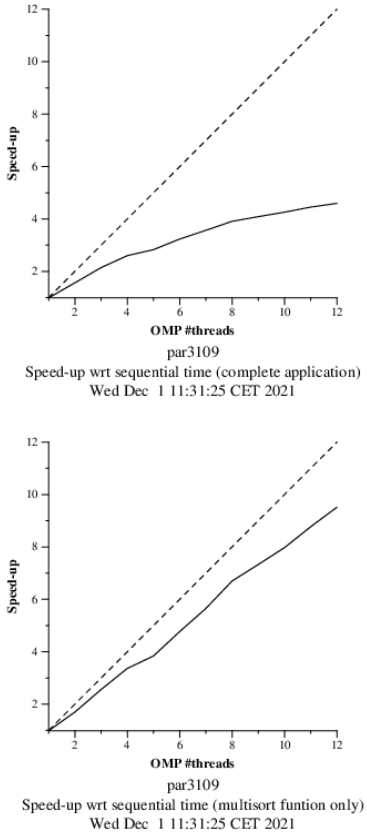


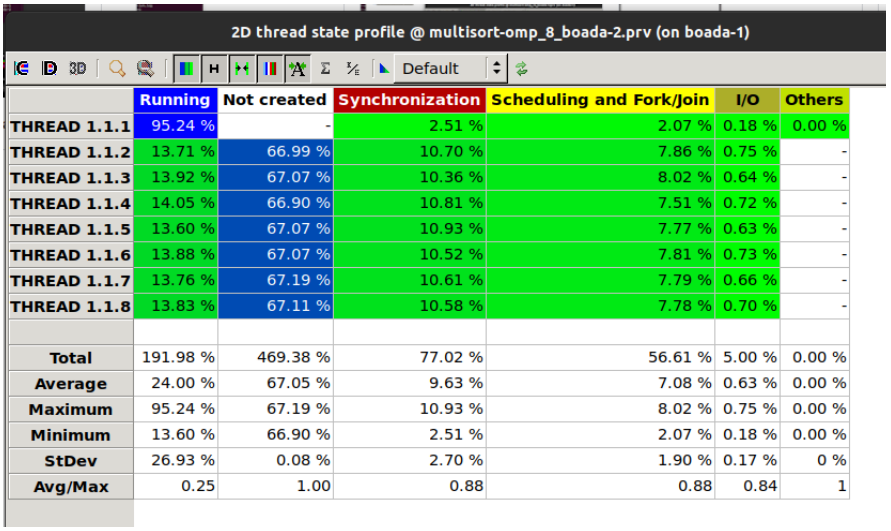
Figure 25: Scalability with dependencies

As we can see in *Figure* 25 above, the strong scalability it's a bit better with the current setup rather than the cutoff with 3 taskgroups.

Figure 26: Explicit tasks with dependencies (8 threads)

The explicit tasks of the new code are very different. We now have a lot of small tasks (as we can see in *Figure* 26) instead of a few bigger tasks. Doing this, we

optimize the time computing because tasks are more distributed and the work is much more balanced.



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	95.24 %	-	2.51 %	2.07 %	0.18 %	0.00 %
THREAD 1.1.2	13.71 %	66.99 %	10.70 %	7.86 %	0.75 %	-
THREAD 1.1.3	13.92 %	67.07 %	10.36 %	8.02 %	0.64 %	-
THREAD 1.1.4	14.05 %	66.90 %	10.81 %	7.51 %	0.72 %	-
THREAD 1.1.5	13.60 %	67.07 %	10.93 %	7.77 %	0.63 %	-
THREAD 1.1.6	13.88 %	67.07 %	10.52 %	7.81 %	0.73 %	-
THREAD 1.1.7	13.76 %	67.19 %	10.61 %	7.79 %	0.66 %	-
THREAD 1.1.8	13.83 %	67.11 %	10.58 %	7.78 %	0.70 %	-
Total	191.98 %	469.38 %	77.02 %	56.61 %	5.00 %	0.00 %
Average	24.00 %	67.05 %	9.63 %	7.08 %	0.63 %	0.00 %
Maximum	95.24 %	67.19 %	10.93 %	8.02 %	0.75 %	0.00 %
Minimum	13.60 %	66.90 %	2.51 %	2.07 %	0.18 %	0.00 %
StDev	26.93 %	0.08 %	2.70 %	1.90 %	0.17 %	0 %
Avg/Max	0.25	1.00	0.88	0.88	0.84	1

Figure 27: State profile with task dependencies

We can see (in *Figure* 27) that the state profile has somewhat changed in relation with past strategies. As we have lots of tasks now, more time is spent in synchronization than before, but it still leaves a lot of room for improvement when tasks are not synchronizing.

4. Conclusions

This fourth laboratory assignment has been the hardest so far. We have found ourselves battling with the scripts and the code to try to get the best parallelization out of our code. However, we think that we have been able to understand parallelism to a new level by completing all assignments due on this document. Even though there are still things that we cannot quite grasp, we hope to find all the answers and more challenges in the next and last lab assignment.