PAR Laboratory Assignment

Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Semester 2 - 2018

Group par2203

23/04/2018

Bernat Gené Škrabec

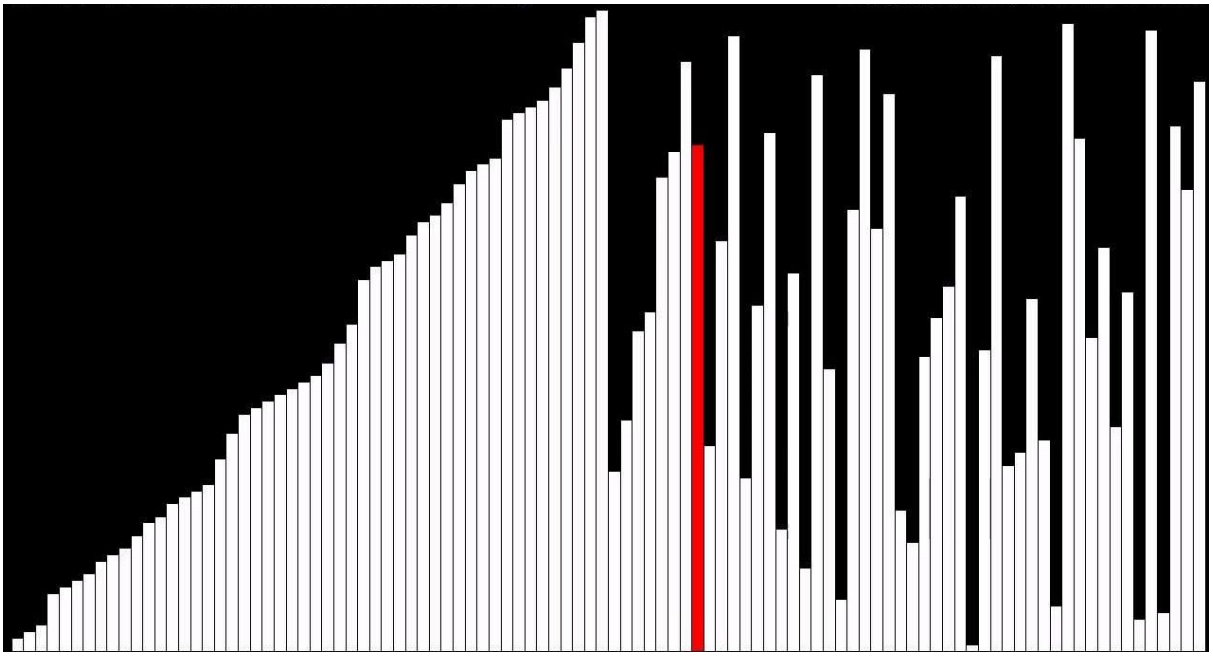
Ferran Ramírez Navajón

Index

Introduction

In this session we will further deepen our knowledge of parallelization techniques by analyzing a program and experimenting with different strategies and tools.

The program we are going to work on is an implementation of Multisort. Multisort is a sorting algorithm which combines a “divide and conquer” mergesort strategy that divides the initial list into multiple sublist recursively, a sequential quicksort that is applied when the size of the sublists is sufficiently small, and a megre of the sublists back to a single sorted list.



We will focus on discovering and understanding the best parallelization strategies for recursive programs using the Tareador tool to analyze potential task decomposition strategies and the OpenMP API to implement them. Given the recursive nature of the code we are going to analyze, there are going to be many complicated data dependencies. We will investigate which are the best techniques to deal with those.

Analysis with Tareador

First of all, we want to have a better understanding of the inner workings of our code. To better understand the tasks that can be defined and the dependencies between them, we will use the Tareador API. With Tareador we can define different tasks in our code and then visualize their dependencies in a neatly organized graph. The following section of code shows the relevant parts of it that were modified in order to define the distinct tasks.

```
 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("bm");
                basicmerge(n, left, right, result, start, length);
                tareador_end_task("bm");
        } else {
                // Recursive decomposition
                tareador_start_task("mm1");
                merge(n, left, right, result, start, length/2);
                tareador_end_task("mm1");
                tareador_start_task("mm2");
                merge(n, left, right, result, start + length/2, length/2);
                tareador_end_task("mm2");
        }
 }
void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                tareador_start_task("m1");
                multisort(n/4L, &data[0], &tmp[0]);
                tareador_end_task("m1");
                tareador_start_task("m2");
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                tareador_end_task("m2");
                tareador_start_task("m3");
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                tareador_end_task("m3");
                tareador_start_task("m4");
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                tareador_end_task("m4");

                tareador_start_task("me1");
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                tareador_end_task("me1");
                tareador_start_task("me2");
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
                tareador_end_task("me2");
                tareador_start_task("me3");
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
                tareador_end_task("me3");
        } else {
                // Base case
                tareador_start_task("b");
                basicsort(n, data);
                tareador_end_task("b");
        }
}
```

Fig1. Region of code with the calls to the Tareador API

We have placed a call to the Tareador directive “start_task” and “end_task” before and after every recursive call to multisort, and named this tasks m1-m4. Then, for each call to the merge function, we also defined a task. Inside this function, merge, a task is defined for each recursive call to itself.

Finally, tasks are defined for each call to the basicsort and basicmerge functions. This is the maximum number of tasks that make any sense to define; we have basically defined a task for each statement in the relevant part of the code. Defining less tasks however, would not make more sense, as it would waste away serious parallelisation potential. Therefore we think this is the optimal number of tasks to be defined.

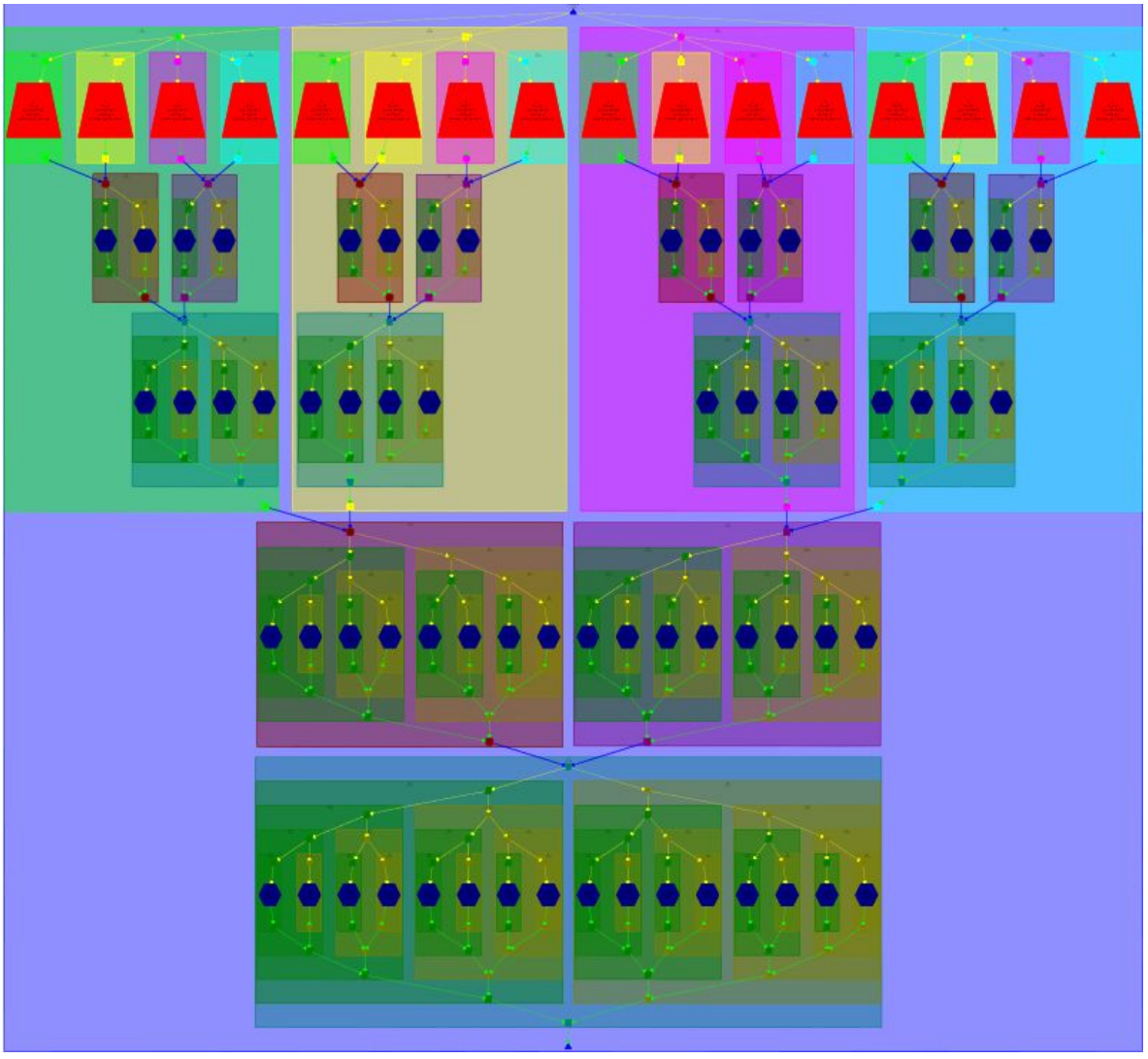


Fig2. Tareator graph of all the tasks generated in our *multisort* program

At first glance, the graph may seem too complicated, but upon further inspection we discover that there is a structure to it. First of all, the recursive nature of our program produces this embedded shapes included inside other figures. This means that every outermost shape is a task which is constituted by the smaller tasks inside them.

After understanding this, it can be seen how the first rectangles (in green, yellow, purple and blue) represent the recursive calls to multisort, which given the size use the basicsort (red trapezoid). After that, the size of the subproblems is small enough that we have the calls to merge, which given its recursive nature has an intricate shape, but we can describe it as the tasks constituted by a given number of blue hexagons.

Tareador provides us with tools that let us simulate the execution of our program with an arbitrary number of processors. We used that functionality to analyse the scalability of our program. In the following table we show the execution time of the simulation executed with different number of processors, and how their speed-up changed in respect to the sequential version.

# processors	**Execution time**	**Speed-up**
1	20,334,421 µs	-
2	10,173,712 µs	1.9987
4	5,085,801 µs	3.9983
8	2,550,377 µs	7.9731
16	1,289,885 µs	15.7645
32	1,289,850 µs	15.7650
64	1,289,850 µs	15.7650

Fig3. Table with the execution time and the speed-up values obtained using paraver

As we can see in the table above, the execution time decreased considerably until we have used 16 processors. In that point the speed-up stopped improving, and that it is due to the fact that the program we have used is not executing more than 16 tasks concurrently.

In the following image, the figX, we can see how all tasks have been distributed between the threads, and graphically is still easier to see how in the last 2 images there are a lot of threads marked with blue, what means there are not more tasks to execute in that thread. That is why we know we are not obtaining the maximum advantage of the parallelisation, therefore we can conclude that using more than 16 processors to execute our *multisort* algorithm is useless. However, this is dependent on the problem size. For a bigger problem size, there would be more tasks defined and therefore it would make sense to use more

threads. There is a proportional relation to the problem size and the number of threads that should be used.

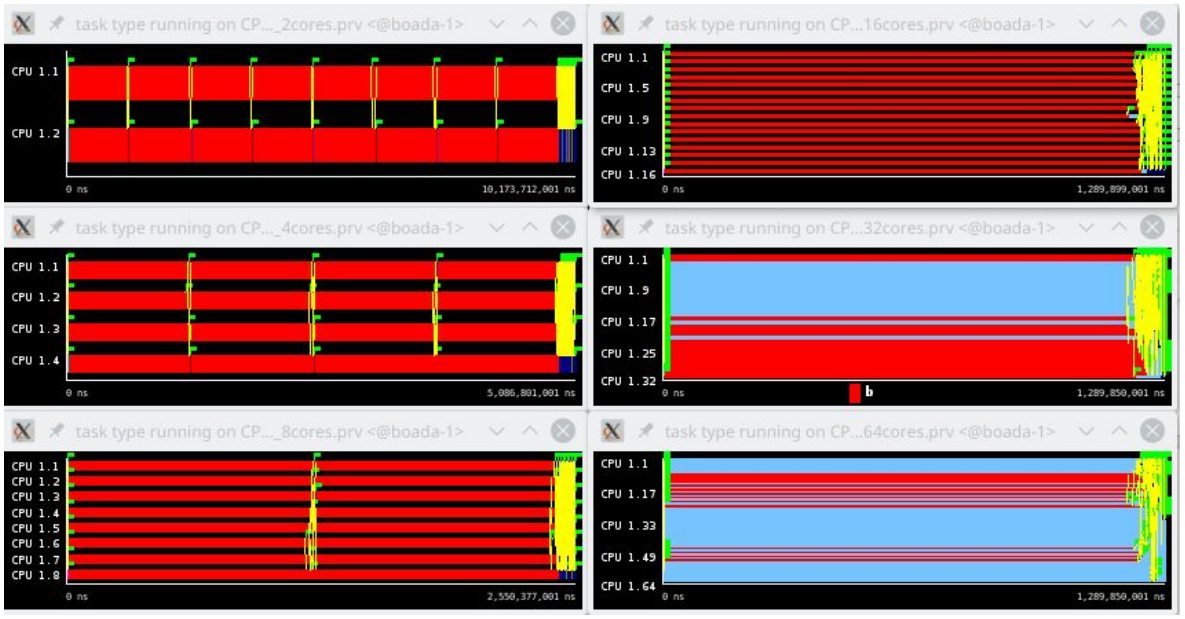


Fig4. Paraver traces of the tasks distribution among 2, 4, 8, 16, 32 and 64 processors

Parallelization and performance analysis with tasks

Now that we have analyzed the potential parallelism with Tareador we will try to implement it using the OpenMP API. We will explore two different parallel versions: *Leaf* and *Tree.*

- In *Leaf*, we define a task for the invocations of **basicsort** and **basicmerge** once the recursive divide-and-conquer decomposition stops.
- In *Tree*, we define tasks during the recursive decomposition, i.e. when invoking **multisort** and **merge**.

We implemented this two versions starting from the code found in “multisort.c”. We have name them “multisort_leaf.c” and “multisort_tree.c” respectively, they can be found in the source folder that was delivered together with this document. The relevant sections of the code are shown below.

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

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

                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
                #pragma omp taskwait
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        } else {
                // Base case
                #pragma omp task
                basicsort(n, data);
        }
}
```

Fig5. Part of the code which implements Leaf parallel version

```
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);
                #pragma omp taskwait
        }
}

void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                #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 taskwait

                #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 taskwait

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

Fig6. Part of the code which implements Tree parallel version

To check that our implementations are correct, we ran the “submit-omp.sh” script to check that the execution and the result of the sort process is correct and no errors about unordered positions are thrown.

Now that we implemented both parallel versions and made sure both work correctly we are going to analyze their behaviour. We have two scripts that will help us in this endeavour, “submit-strong-omp.sh” and “submit-omp-i.sh” . The first one mesures its strong scalability by executing the program with an increasing number of threads and plotting the speed-up respective to the sequential version in each run. The second one generates a trace of the execution which can be analyzed using Paraver. We have included the results of this analysis below.

Leaf

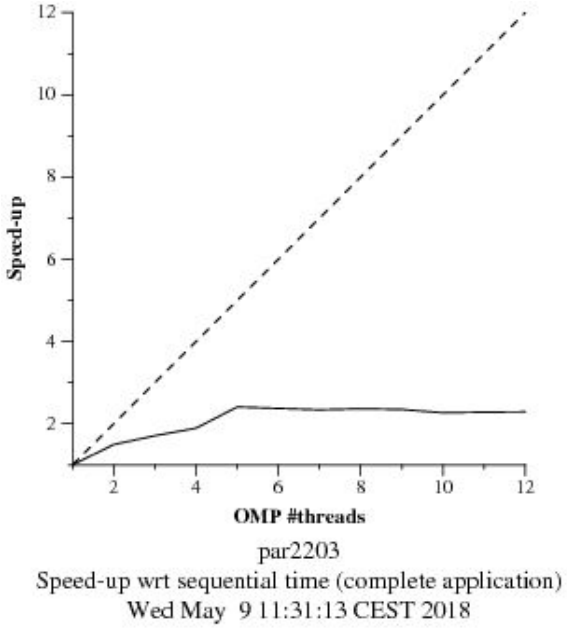


Fig7. Speed-up plot of the complete code execution (Leaf version)

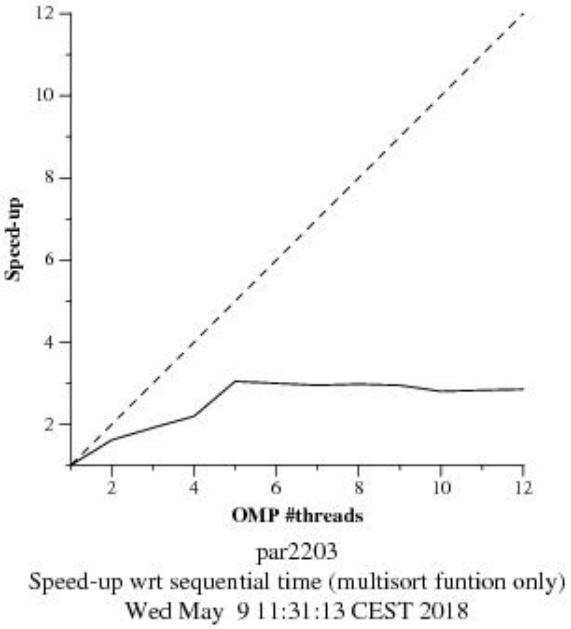


Fig8. Speed-up plot of the multisort function only (Leaf version)

Observing the speed-up plot of the leaf version, we can see that it does not scale very good and does not scale at all after 5 threads are used. Let us try to reason as to why this might

be. When decomposing our program in tasks using this strategy we are defining a task for each invocation of the basic functions once the recursive divide-and-conquer decomposition stops. The problem is that this way, only five tasks can be defined that can run concurrently and there is no use to using any more than five threads, just as the plot we are focusing on shows.

We can further prove that point by analyzing the trace of the execution with Paraver.

Next we have included two captures of the timeline.

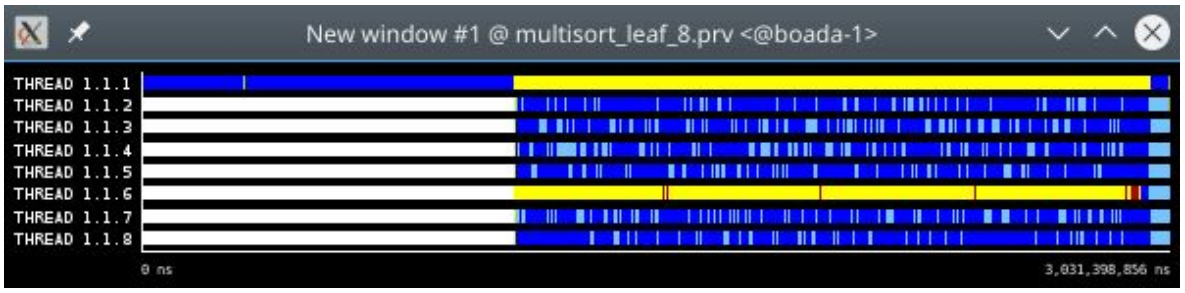


Fig9. Paraver trace of Leaf parallelisation

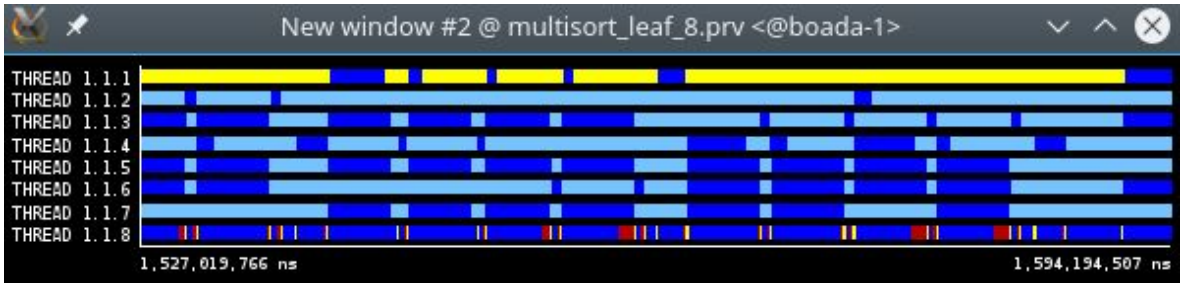


Fig10. Zoom-in paraver trace of Leaf parallelisation

Let us look at these two images, Fig9 and Fig10. The first one show the timeline of the execution from start to finish. From that perspective it seems we might be wrong about our assumption, because it appears that there are more than five tasks running concurrently. However, if we zoom in at any part of the timeline, as shown in Fig10, we see that at any given time, only five threads are actually running (dark blue). Therefore we can conclude that this is not a very good parallelization strategy because of the nature of the program and the way we defined the tasks, only five threads can work at a time.

Tree

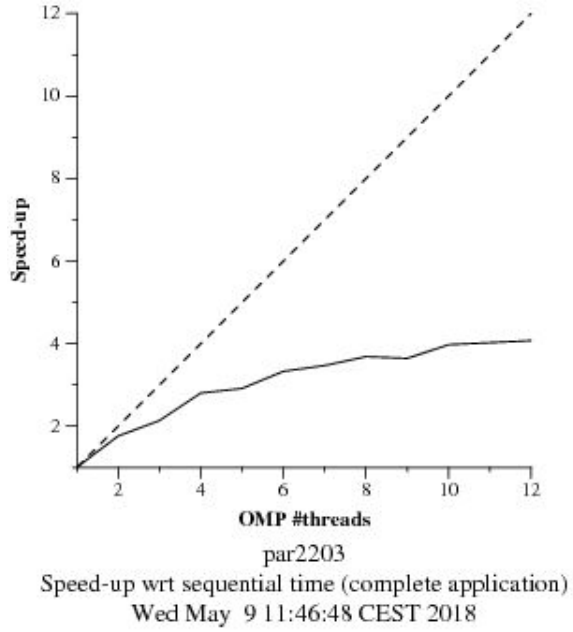


Fig11. Speed-up plot of the complete code execution (Tree version)

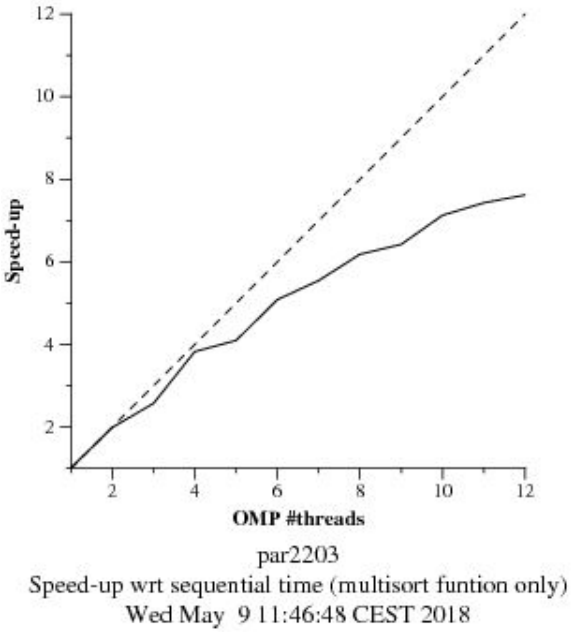


Fig12. Speed-up plot of the multisort function only (Tree version)

Now, looking at the speed up plot (Fig11) of the tree version we see that it scales better than the previous version. The scalability is not perfect and this has much to do with the nature of the program, but if we focus on just the speed-up plot of the multisort function it is obvious that this strategy is far better than the other. This is because now we are not limited in the dependencies between tasks and far more of them can be run concurrently. We will further prove this point but looking at the paraver timelines we included next.

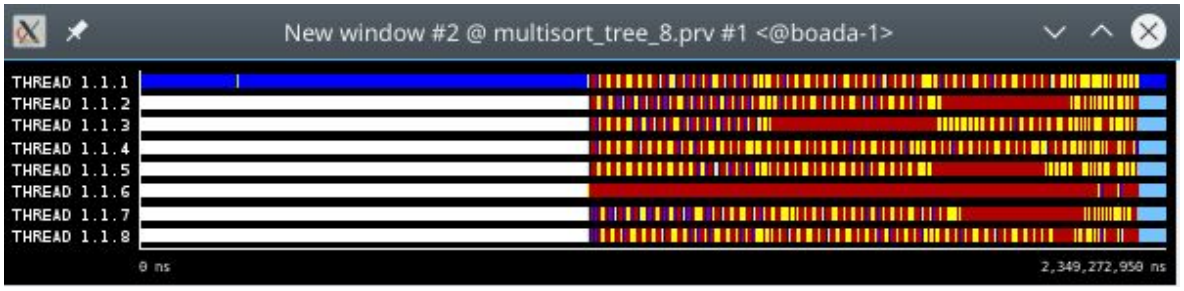


Fig13. Paraver trace of Tree parallelisation

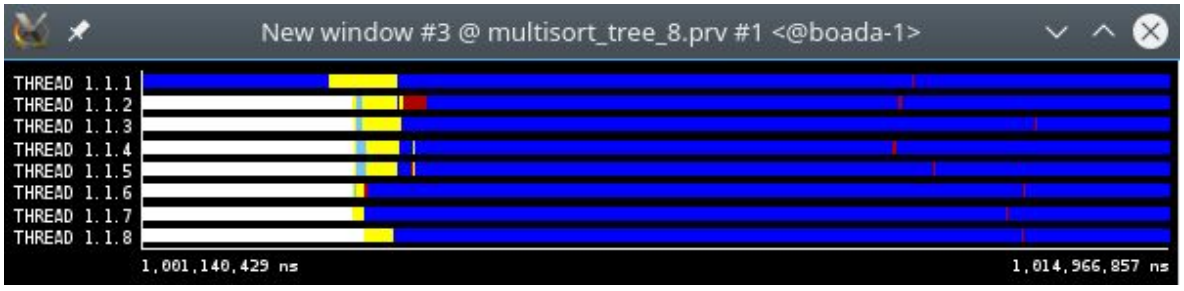


Fig14.Zoom-in paraver trace of Tree parallelisation

Fig 13 shows the timeline of the tree version execution. At first glance we might be bothered by the amount of red and yellow, that indicate time lost in synchronization and scheduling fork-join, respectively. This is because of the vast amount of tasks created and the dependencies they have between them, as they are being created continually due to the recursive nature of the decomposition and the need to combine the results of the work done by each task. However, this does not have a severe enough impact in the performance of the program as we saw in the plots. From the perspective of that trace it is difficult to assert if all threads are working concurrently because the amount of red and yellow clutters the vision. However, if we zoom in at any region of the timeline, as done by Fig14, we can see that indeed all eight threads are running concurrently most of the time.

Parallelization and performance analysis with dependent tasks

Now we will try to improve the previous version, Tree, using task dependencies. This makes sense given the nature of the program, given some of the taskwait and taskgroup synchronizations that had to be implemented to enforce the correct dependencies can be avoided by using the correct task dependencies directives. We have included a screenshot of the relevant portion of the code that show how we introduced those directives.

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

                #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);
                #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);
                #pragma omp taskwait
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        } else {
                // Base case
                basicsort(n, data);
        }
}
```

Fig15. Part of the code which implements Tree parallel version with dependencies

Fig15 shows how we have implemented the decomposition. For example, in the following task definition:

```
#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);
```

We are specifying that the task can not be executed until de sibling task (i.e. a task at its same level) that generates both data[0] and data[n/4L] finishes. Also, when the task finishes, it will signal other tasks waiting for tmp[0].

We can observe a slightly better performance and scalability when compared to the version without task depend (Fig16, 17). This is what we expected and the explanation is the one we gave before. In the previous version we had to force waits to ensure a correct concurrent execution because of the data dependencies. With this directive, wait, we can explicitly state

this dependencies and therefore optimize the time lost waiting with a better execution schedule because we only wait for actual dependencies.

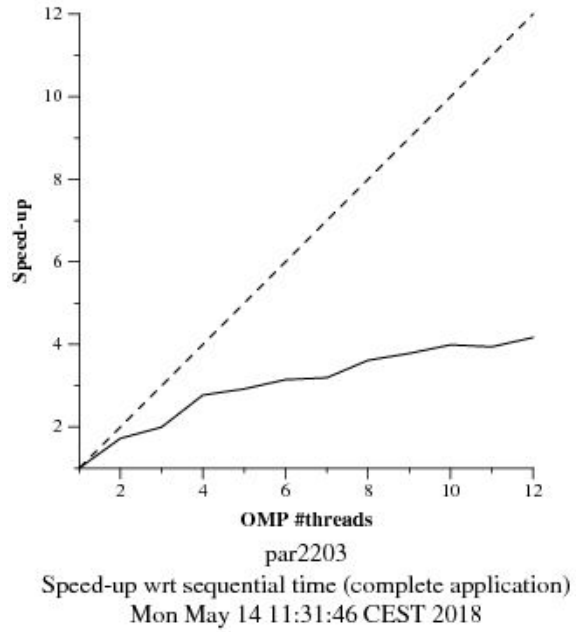


Fig16. Speed-up plot of the complete code execution (Tree version with dependencies)

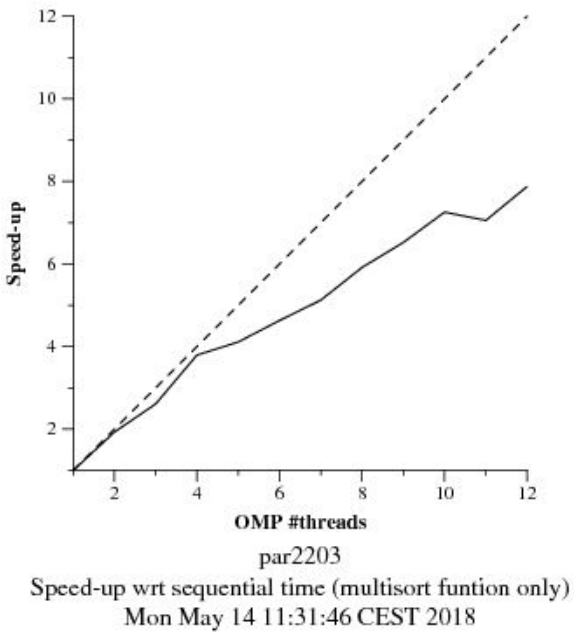


Fig17. Speed-up plot of the multisort function only (Tree version with dependencies))

Fig18. Paraver trace of Tree parallelisation with dependencies

Fig19. Zoom-in paraver trace of Tree parallelisation with dependencies

Similarly to what happened with the traces of the previous tree version, the timeline gets clustered with red and yellow, but this is due to their frequency of appearance and not their actual proportion of time dedication (Fig18). As we did before, when zooming in we can see the actual proportion of red is far less. However all the work one of the threads does, the 1.1.7, is to synchronize the other ones. This is one of the reason to the limit of performance we are observing.

Optional

Optional 1

In this section, we want to attempt to parallelize the two functions that initialize the data and tmp vectors1 (initialize and clear). We have included the changes that we deemed appropriate.

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

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

Fig20. Part of the code which implements parallelisation even in the initialization of vectors

Let us comment about the changes we have made (Fig5). The second function is easily parallelized by the pragma omp for. However there is no much we can do with the first one. We have tried a doacross strategy but given the dependencies of each iteration (every iteration depends from the previous one) it is still going to execute sequentially. In fact, due to the cost of defining those tasks, the performance is probably going to be even worse. Next we include the speed-up plots that back up our statements.

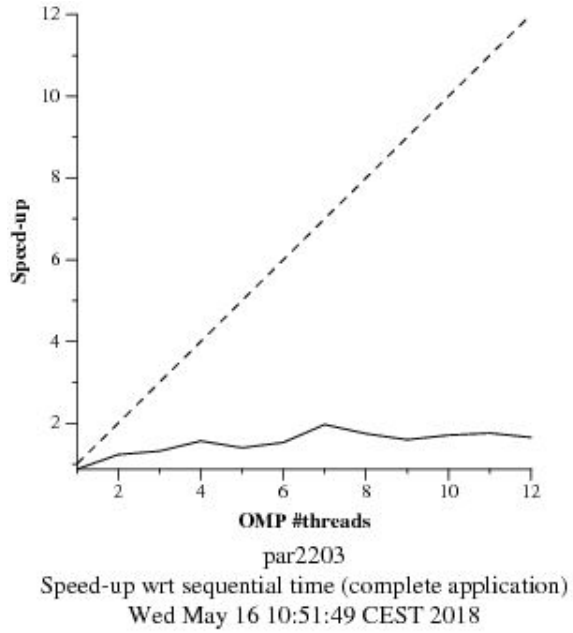


Fig21. Speed-up plot of the complete code execution (Tree version and parallelisation of initial vectors)

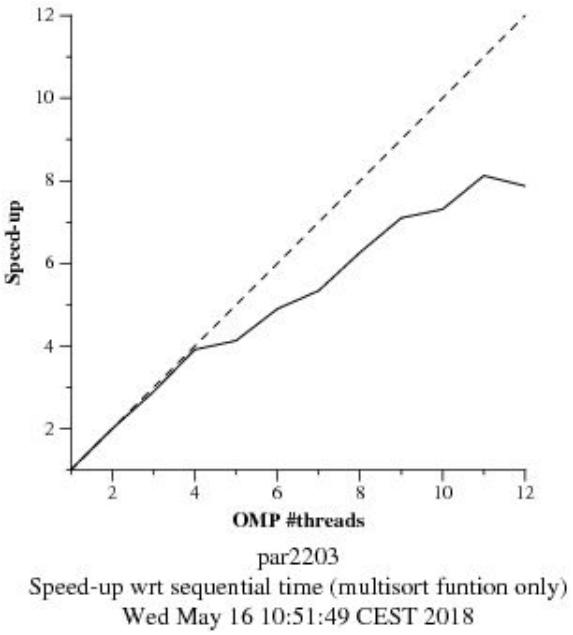


Fig22. Speed-up plot of the multisort function only (Tree version and parallelisation of initial vectors)

Fig 21 shows that the performance of the program is far worse than just the tree version(Fig16). Fig 22 only shows the speed-up of the multisort function and that is why it stays the same as it was.

We can conclude that given the structure of the code and the way of initializing the vectors, there is no option to parallelize given that each iteration depends form the previous one, and any attempt at defining useless parallelization strategies will in fact worsen the performance. Therefore, we can either leave it as is or maybe talk to the people that wrote the code and ask them if there is any other way of initializing the vectors.

Optional 2

In this section, we will try to reason what are the best possible values for the sort size and merge size arguments used in the execution of the program.

First we will execute the “submit-depth-omp.sh” script which performs a number of executions changing the value for sort size. From this we will try to reason what is the influence of the sort size argument is and select the best value for it.

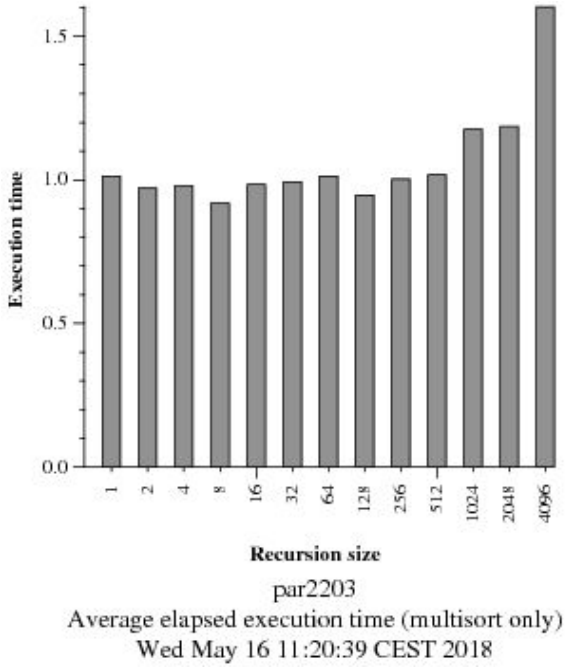


Fig23. Execution time plot of the multisort function with different sort sizes (Tree version with dependencies)

Fig 23 plots the execution time as a function of the sort size. As it can be observed, it is quite stable for smaller sizes and grows exponentially for bigger ones. The minimum execution time is with 8 as a sort size, but the difference is quite minimal and can be caused by other factors. However, as it is certainly not worse, we will take 8 as a sort size and now try to determine the best merge size.

To determine the best merge size, we will modify the script “submit-depth-omp-sh” so that it maintains a stable sort size, the one we determined previously, and goes about changing the merge size with the same possible values as the aforementioned variable. This will generate a plot similar to Fig23, that will hopefully tell us what should be the perfect merge size.

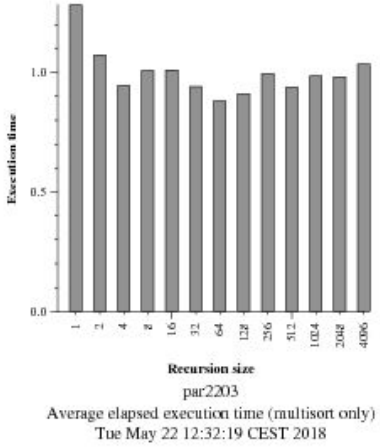
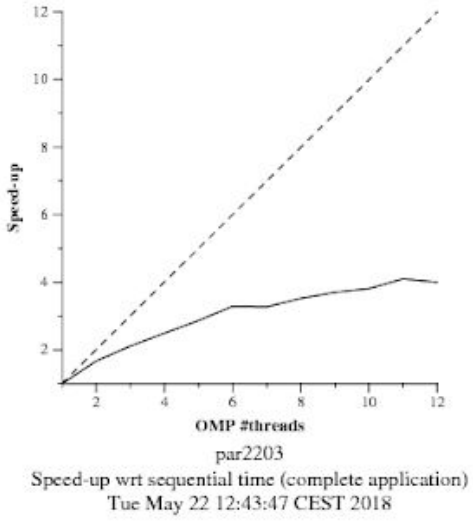


Fig24. Execution time plot of the multisort function with different merge sizes (Tree version with dependencies)

Fig24 shows the different execution times obtained by modifying the original merge size. The exact best size seems to be unclear, but if we draw an imaginary tendency line (parabolic) it looks like the extreme values (too small or too big) perform worse. Therefore, we are going to choose the certermost value, 64, as our merge size.

Now that we have determined those values, we will modify the original strong scalability script to obtain a new plot and compare it to the original one.



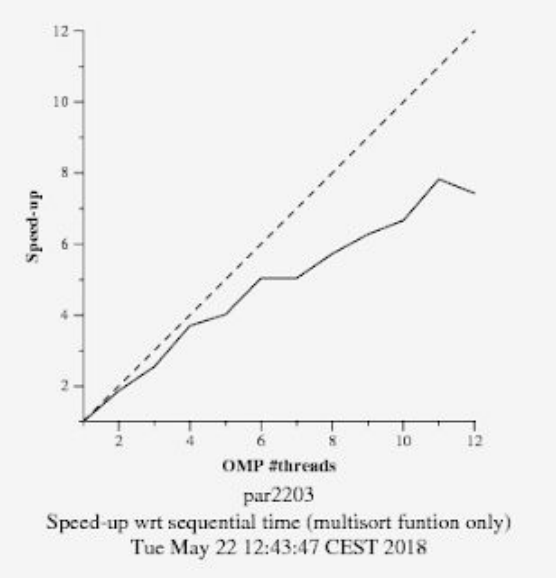


Fig24. Speed-up plot of the complete code execution (Tree version and parallelisation of initial vectors)

Fig25. Speed-up plot of the multisort function only (Tree version and parallelisation of initial vectors)

As it can be observed in the previous plots (Fig24 and Fig25) compared to the original ones (Fig16 and Fig17), the speed-up is very similar. From that we can conclude that the changes made in the recursive depth sizes had very little impact in the performance of the program, probably because the original one already had its values close to the best possible ones.

Conclusions

After concluding this session we can say that our knowledge of parallelization strategies and techniques has increased and we have improved our ability to tackle new parallelization challenges that may present to us in the future.

We learned how to use Tareador to analyze the complex recursive nature of the multisort program, because it gives a comprehensible graph of the tasks defined and their dependencies (in particular, recursive tasks are shown as embedded into their parent tasks, which makes the graph somewhat complicated but useful nonetheless).

With the information we gathered with it, we experimented with different strategies using OpenMP directives, much like in the previous session in which we tried to decompose a program that computed the Mandelbrot set. We also used a directive we had not seen before, depend, which is useful to synchronize tasks when the dependencies between them become complicated.