Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Pablo Cebollada Hernández
Iván Díaz Ortega
Par1104
Q1 2019/2020
1/12/19

Index

Introduction

In this assignment we are going to study the potential performance improvement caused by the parallelization of the code of the mergesort algorithm. First we are going to look at the potential parallelism using tareador with a tree task decomposition. Afterwards we are going to study the two possible task decomposition strategies: tree and leaf. For that purpose we are using paraver and strong scalability plots. We are also implementing a cut-off mechanism to see if this can grant us some performance improvement.
Finally the dependency granting is going to be made using the depends clause and, as done before, try to observe if there is some kind of performance impact.

1. Analysis with tareador

To achieve the Tareador execution, we've added a tareador instruction before every call to the function "multisort" and "merge". That provides us a specific view of all the dependencies and tasks created. In order to generate a more visually appealing dependency graph we named every task with a different name, so every kind of task generated would have a different colour.

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

```
void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                tareador_start_task("a");
                multisort(n/4L, &data[0], &tmp[0]);
                tareador_end_task("a");
                tareador_start_task("b");
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                tareador_end_task("b");
                tareador_start_task("c");
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                tareador_end_task("c");
                tareador_start_task("d");
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                tareador_end_task("d");

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

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

Fig 1.1 Code modified

We obtained the task dependency graph shown in Fig 1.2. As we can see the execution of the first level of recursion creates four tasks (green, red, yellow and pink squares) that represents the "multisort" function, once every task is created the first task and the second create a dependency for the "merge" task below, the same for the third and fourth. Afterwards the next merges create dependencies for the contiguous merge calls, thus creating a chain of dependencies between the original calls and the merges. You have to keep in mind that every task generated makes, again, every step made before, so these dependencies happen again recursively.

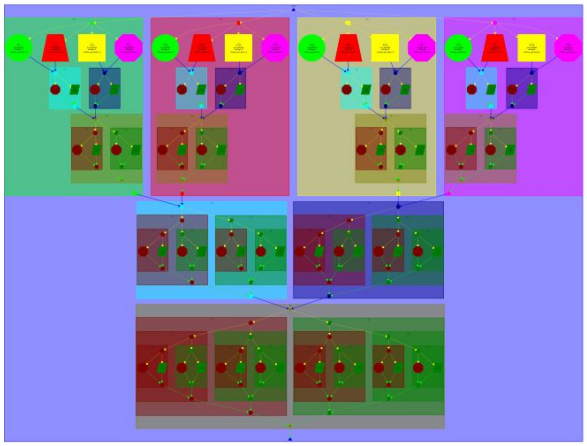

Fig 1.2 Dependencies graph

If we take a look at the graph above we can see the execution time and the speed-up plots for every core count. Focusing first on the execution time we can see that its improvement is almost proportional to the core count, since it is reduced by the same factor that the core count is augmented. The same could be said for the speed-up progression, since it also evolves proportionally to the core count. Also, if take into account the task dependency graph in Fig 1.2 and look for the maximum number of possible tasks that can be run simultaneously we can see that this situation occurs when the second recursive iteration is happening. At this moment there are 16 tasks running simultaneously so the maximum amount of simultaneous number of tasks is 16. Said the above, it's pretty obvious why both plots stabilize at 16 cores, since this is the maximum number of tasks that we can run in parallel, and it is the maximum parallelism possible.

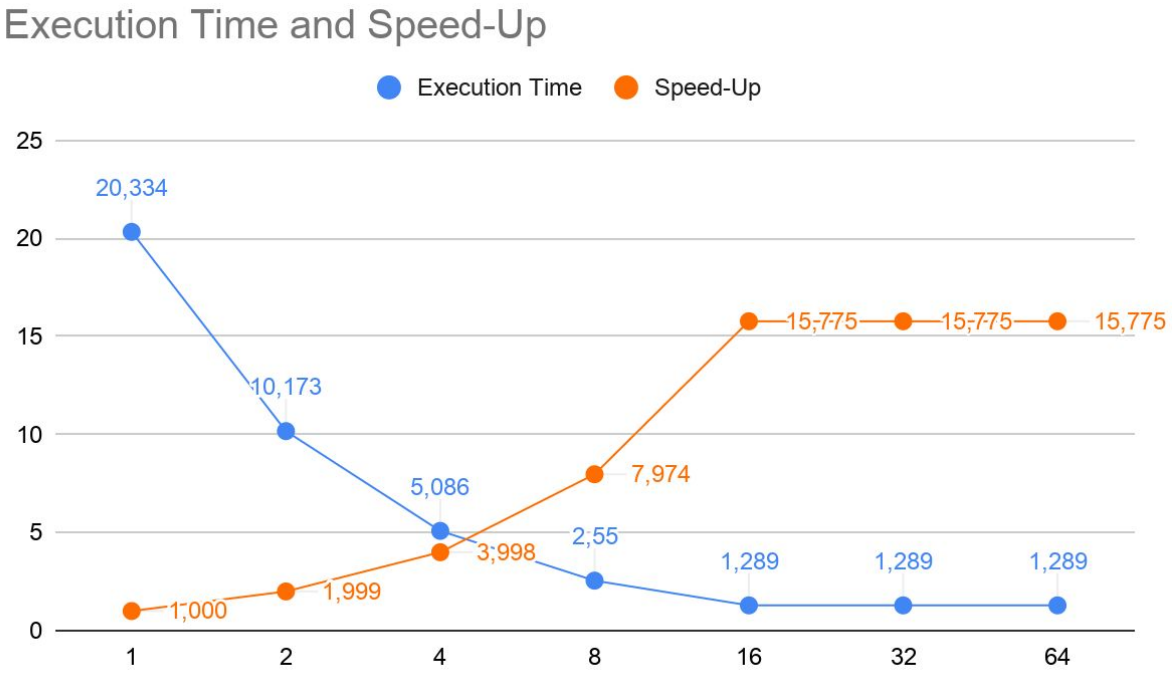

Fig 1.3 Execution Time and Speed-Up Graph

2. Parallelization and performance analysis with tasks

On one hand, to make the leaf version, all we did, was to create a task for every base case. These base cases are the calls to basicsort and the calls to basicmerge. This causes that only one thread executes the recursion and generates all the tasks. These tasks are generated when the recursion gets to the deepest level, a leaf, and are mostly executed by the other threads.

```
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);
        }
}
```

Fig 2.1 Leaf version code

On the other hand, the tree tree version was created by putting these task generation clauses before every recursive call. This way every thread creates and executes tasks all the time, hence every node is a new task except for the leafs.

```
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
                basicsort(n, data);
        }
}
```

Fig 2.2 Tree version code

In both cases we need some kind of synchronization mechanism to maintain the dependencies observed in the task dependency graph. These mechanisms are the taskwait clauses located after all the calls to multisort and after the calls to merge in the multisort method, and also the one before the recursive calls in the merge method in the tree version. These clauses keep the update order of the array to order done in the correct order.

Looking now at the leaf strategy we can see how the speed-up plot is not very promising (Fig 2.3). Since this strategy only makes tasks for the non-recursive part of the execution it is best suitable for a program that this part has more impact in the execution time than the recursive part, but in this program that is not the case. Due to this, one of the most important drawbacks of the tree parallelization is that if that doesn't happen there would be threads doing nothing since the thread that generates the tasks would be doing the recursion.

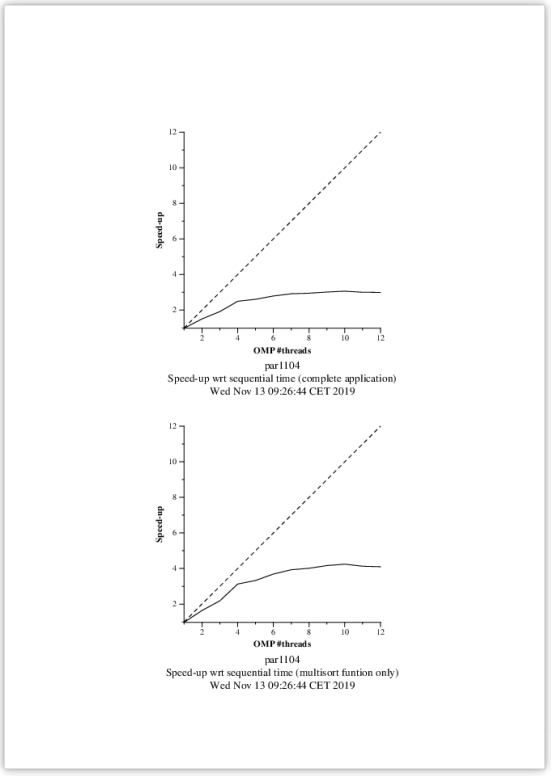

Fig 2.3 Strong scalability plot leaf strategy

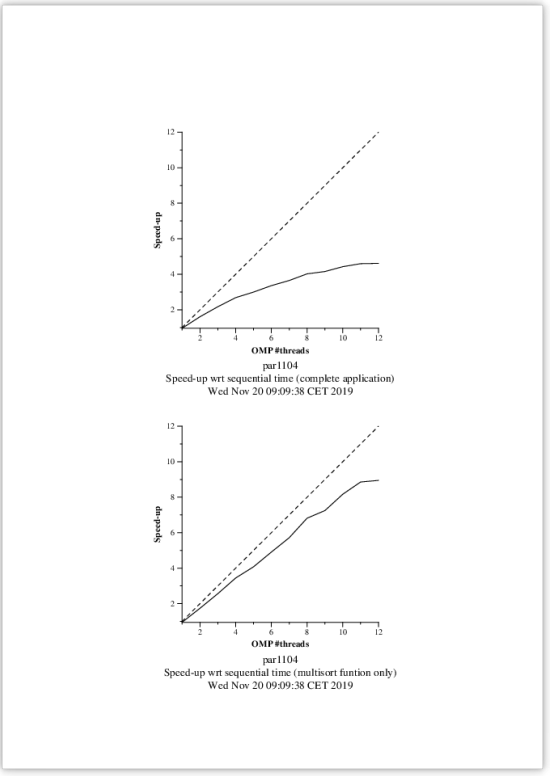

Fig 2.4 Strong scalability plot tree strategy

This situation can be seen in these two images below. Fig 2.5 shows the tasks that are executed and which thread executes them. Looking at this, it is clear the previous situation, since between almost every task executed there is a blue space, which indicates that the thread is doing nothing. The first thread has less tasks executing because it's the thread that creates them and does all the recursive execution. The creation of the tasks also shows us the situation explained, since there is a time space between some tasks instantiations and these spaces show a pattern, due to these instantiations are made for the non-recursive part of the execution and are always created in the same way

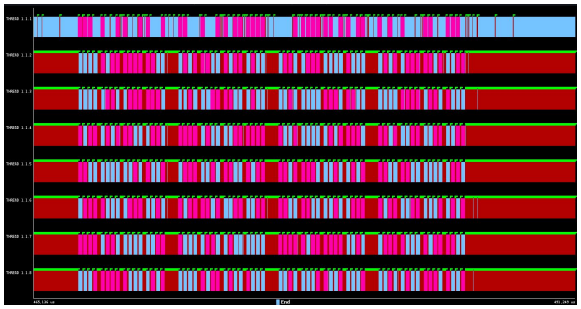

Fig 2.5 Tasks executed leaf

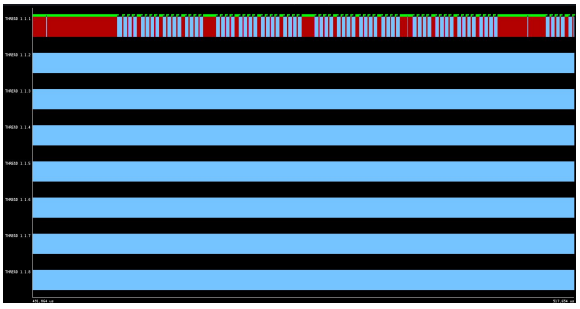

Fig 2.6 Tasks instantiated leaf

Now let's look at the strong scalability plot for the tree strategy shown in Fig 2.4. This one presents a better looking speed-up plot, meaning that its potential parallelism is bigger than the one for the leaf strategy. This is, as happened with the leaf strategy, due to the weight that both parts of the programs, the recursive and the non-recursive, have on the execution time. In this case there isn't only one thread creating tasks, but the other threads also creates tasks, avoiding the idle times that had the leaf strategy.

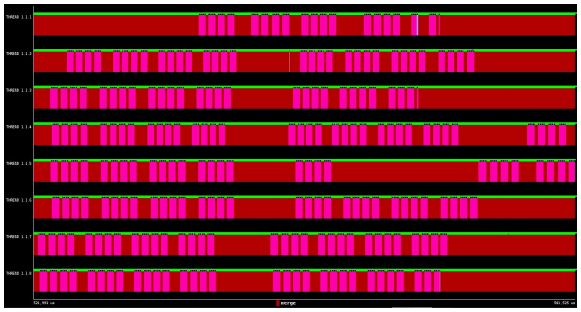

Fig 2.7 Tasks executed tree

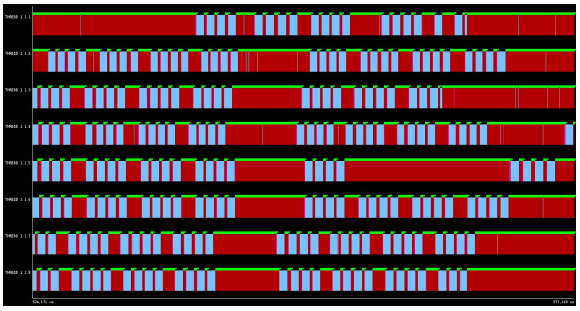

Fig 2.8 Tasks instantiated tree

This situation can be seen in the pictures above. In Fig 2.7 we have the tasks that are being executed and in Fig 2.8 the tasks that are being instantiated. By looking at this the improvement in the density of tasks is quite big, since there are almost no gap between tasks being executed. Because of this the potential speed-up is bigger, since there is less overhead generated.

Now we want to implement some kind of cut-off mechanism and see if this gives us some kind of performance improvement.
The two pictures below show the different changes made in the code of the original program to implement this cut-off mechanism. The most important change is the use of the final clause to limit the task creation under a certain recursion deepness. The condition for this clause to take effect is that a variable that is passed from the calling method is greater or equal to a certain value. This variable is a parameter that is passed from parent to child, and it increases by one every time that a recursive call is made, so, in this way, we can know the level of recursion that we are at the moment.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int p) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                basicmerge(n, left, right, result, start, length);
        } else {
                if(!omp_in_final()){
                        #pragma omp task final(p >= CUTOFF)
                        merge(n, left, right, result, start, length/2, p+1);
                        #pragma omp task final(p >= CUTOFF)
                        merge(n, left, right, result, start + length/2, length/2, p+1);
                        #pragma oap taskwait
                }
                else{
                        merge(n, left, right, result, start, length/2, p+1);
                        merge(n, left, right, result, start + length/2, length/2, p+1);
                }
        }
}
```

Fig 2.8 Changes for cutoff in merge

```
void multisort(long n, T data[n], T tmp[n], int p) {
        if (n >= MIN_SORT_SIZE*4L) {
                if(!omp_in_final()){
                        // Recursive decomposition
                        #pragma omp task final(p >= CUTOFF)
                        multisort(n/4L, &data[0], &tmp[0], p+1);
                        #pragma omp task final(p >= CUTOFF)
                        multisort(n/4L, &data[n/4L], &tmp[n/4L], p+1);
                        #pragma omp task final(p >= CUTOFF)
                        multisort(n/4L, &data[n/2L], &tmp[n/2L], p+1);
                        #pragma omp task final(p >= CUTOFF)
                        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], p+1);
                        #pragma omp taskwait

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

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

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

Fig 2.9 Changes for cutoff in multisort

To take a look at the possible performance improvement, first we had to decide what value would we use for the cut-off, since having a lower value or a higher than the adequate could give us no performance improvement or, even, worse performance. Having this in mind we generated the plot in Fig 2.9, which shows the different execution times for different cut-off values, and we saw that the most appropriate value is 4, since, in the plot, it is the value with the lowest execution time. When we decided what value would we use we proceeded to evaluate the scalability of this strategy. If we compare the plots in Fig 2.4, that shows the scalability plot for the tree strategy without cut-off, and in Fig 2.10, which shows the scalability plot for a tree strategy with cut-off, we can see that the difference is not really substantial since both plots are very similar. This can be due to having a very good task distribution which produces a very balanced work distribution for the different cores in both cases

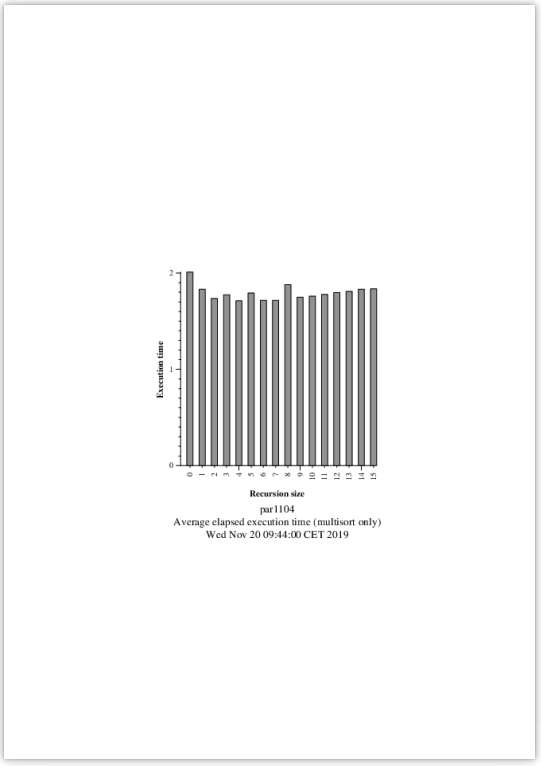

Fig 2.9 Execution time for cut-off size

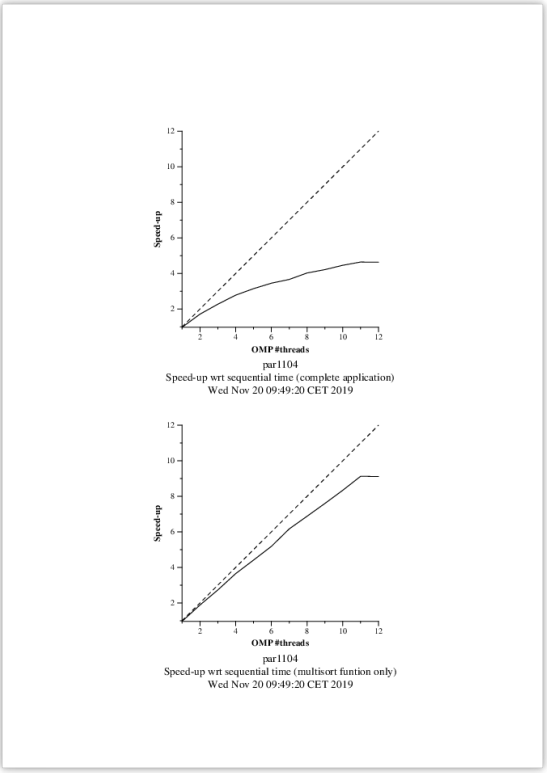

Fig 2.10 Speed-up plot for cut-off

3. Parallelization and performance analysis with dependent tasks

To create dependencies between functions we focused in the dependency graph that we previously made. It is important to know that these dependencies are originated because it is necessary that the part that we are going to use is previously ordered.

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

Fig 3.1 Dependencies code

In some way the dependencies that we created in each function are a substitute to the taskwait in the previous version. For this reason we can not observe a significant improvement between the depends version and the taskwait version.

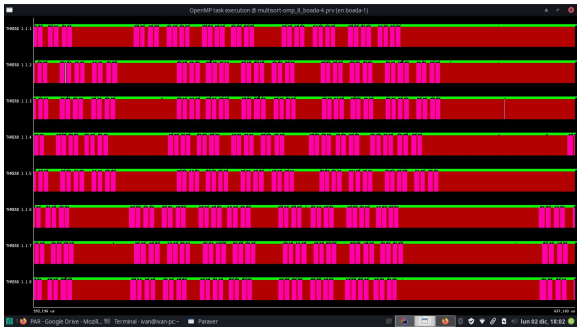

Fig 3.2 Task execution

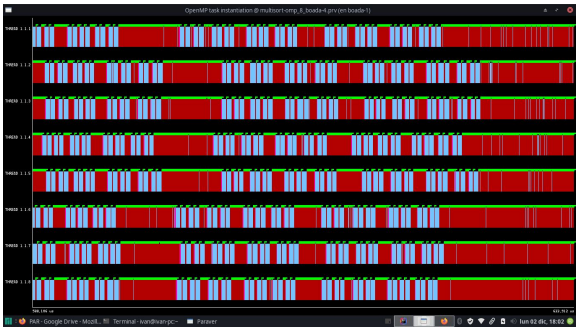

Fig 3.3 Task instantiation

We can also see these similarities with the taskwait version by taking a look at the paraver captures in Fig 3.2 and Fig 3.3. We can see how the tasks are instantiated in a very similar way, with very similar spaces between instantiations and they are executed in a very similar way too.

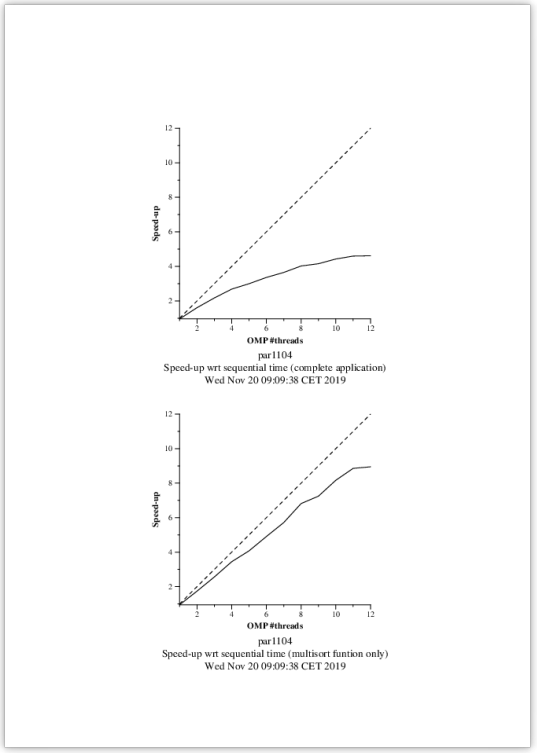

Fig 3.2 Strong scalability plot tree strategy

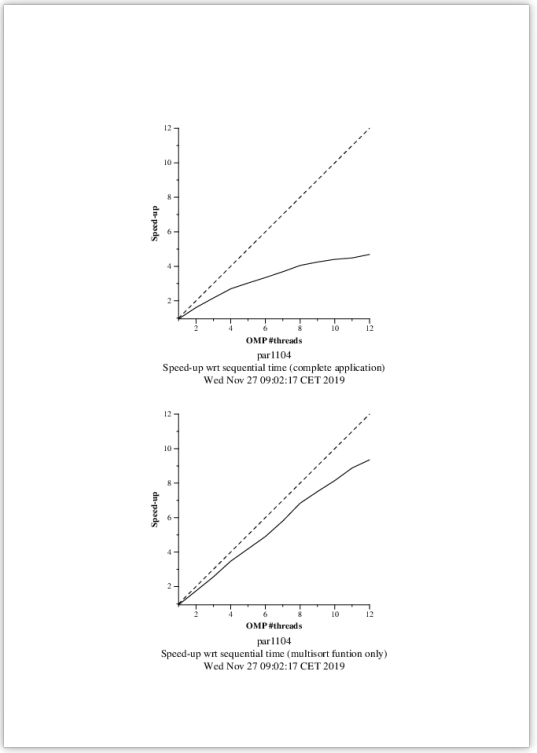

Fig 3.3 Strong scalability plot tree-dependencies strategy

4. Optional 1

After we send the tree verison into all nodes in boada (execution, cuda, execution2) we obtain the following plots:

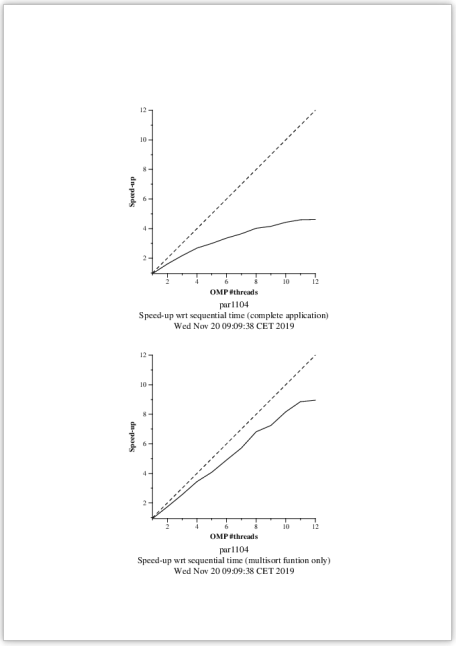

Fig 4.1 Execution

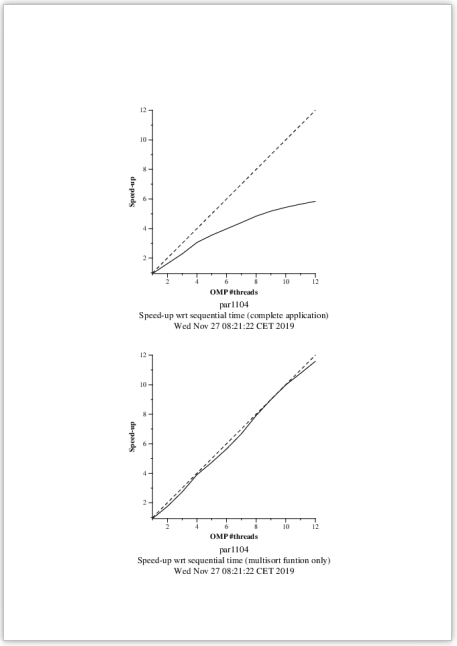

Fig 4.2 Cuda

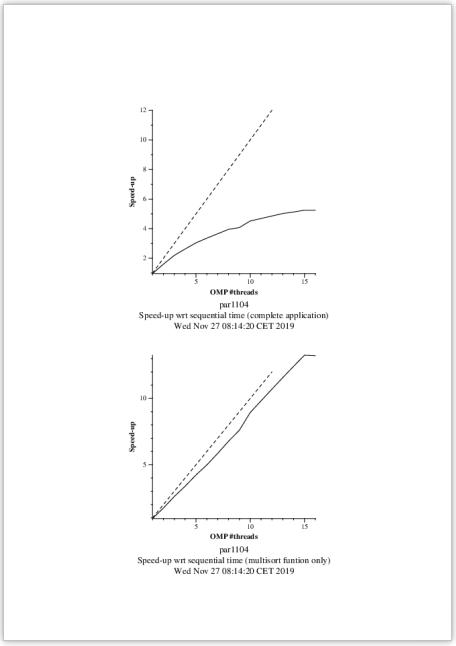

Fig 4.3 Execution 2

As we can appreciate there is not much difference between the top of the plot, but if we focus at the bottom, there are significative differences, these are due to the differences between nodes. For example Cuda is the node with the maximum value of the core frequency, thanks to that, Cuda execute the part with overhead faster than the others and execution2 have more cores that's the reason why exection2 is much better than the execution version.

5. Optional 2

In this part, the objective was to make the initialization part, previously sequential, into a parallel part. We achieve that by including before the loop of the clear function of the code a #pragma omp taskloop clause. As the iterations in this loop are completely independent we can parallelize it completely by just adding this clause.

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

Fig 5.1 Clear code

Another step to fully parallelize the startup code is to do it for the initialize method. Initially, this method creates only one random number with the call rand(), puts it in the first position of the array and then initializes the other positions by doing some mathematical operations to the previous position and assign the result to that position. If we want the code to do exactly the same then it's impossible to parallelize, because the dependencies in this code would make it to be completely sequential anyway, so if we want to parallelize this code, some modifications have to be made. This modifications consisted in dividing the original array in smaller chunks and do the same initialization that was made for the full array to these smaller chunks. This way we can initialize different chunks in different threads without having to worry for the dependencies that can exists, since there are none.

```
static void initialize(long length, T data[length]) {
   long i;
   byte randomed = 0;
   for (i = 0; i < length; i++) {
      if (!randomed) {
         data[i] = rand();
         randomed = 1;
      } else {
         data[i] = ((data[i-1]+1) * i * 104723L) % N;
      }
   }
}
```

Fig 5.2 Initialize code

After implementing all this changes, we send the code again with the submit-strong script to obtain the final plot.

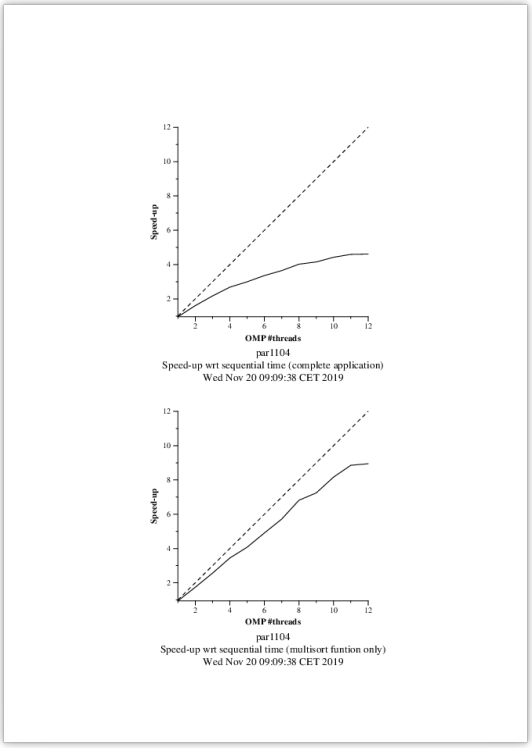

Fig 5.3 Normal strong scalability plot tree strategy

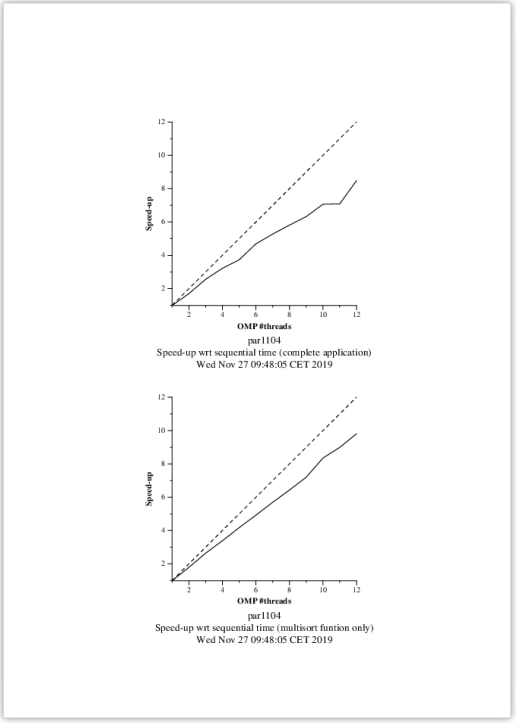

Fig 5.4 Strong scalability plot tree strategy with the optional changes

In this case the difference between the two plots can be observed at the top plot, which indicates the speed-up for the whole program. With the optional changes that speed-up is much better than the normal case, since the part that we have just parallelized is the one that wasn't parallelized before and therefore worsened the general speed-up plot by adding that extra execution time doing it sequentially.

When we obtain the instrumentalization of Paraver, we can see in the purple "circle" on the Fig 5.4 how the initialization of data and tmp vector is executed in parallel.

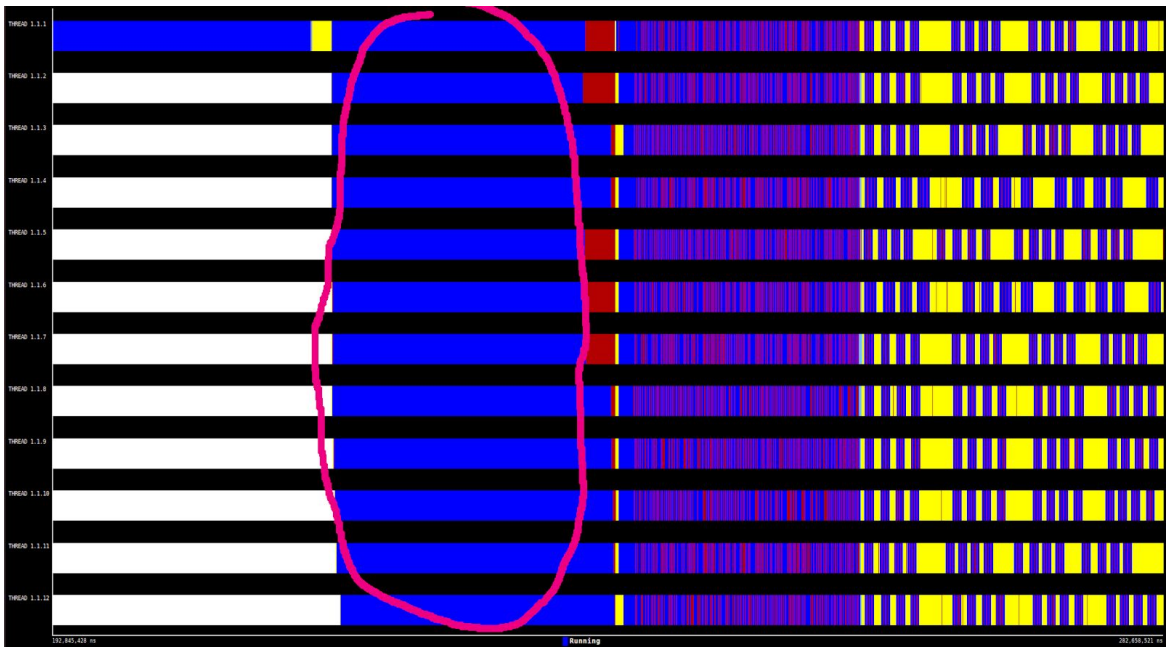

Fig 5.4 Paraver optional code

Finally we can see how the Time in the improvement (Fig 5.6) is lower than the original version (Fig 5.5).

Fig 5.5 Time with secuencial initialization

Fig 5.6 Time with the parallel initialization

Conclusions

To sum up all the work done above, we can say that, in this specific case, the best strategy to implement is the tree task distribution strategy, since, as said before, is better when the weight of the execution time is on the recursive part. We've applied several mechanisms as a cut-off mechanism, that showed practically no benefit in terms of performance, and different synchronization mechanisms, as taskwait and depend, that also showed almost no difference in performance between them. A difference in performance could be seen when parallelizing the initialization part, that showed a big boost for the speed-up plot for the whole program, since we had a gain in execution time, and we could also see how the different hardware configurations behaved different, depending on the core count and on the core freqüency.