UPC: Facultat de Informàtica de Barcelona

Lab 4 - Divide and Conquer parallelism with OpenMP: Sorting

PAR Laboratory Assignment

Oriol Moyano Núñez
Martí Ferret Vivó

ID: par2108

22/05/2018

Index

Introduction

The focus of this document is the study of the parallelization of a divide and conquer problem; observing the possible opportunities it presents (timewise), analyzing the overheads it produces and extracting conclusions about the whole process. To this end, first we need to understand some basic concepts, the problem we will be working on and the parallelization strategies we will study.

Divide and conquer is a technique used in many fields, among them politics, sociology, economics and computer science. The basis of this strategy is dividing a greater problem into smaller ones, easier to handle by the one implementing this approach. In the case of this document, focused on the implementation of said technique in the computer science field, the strategy basically translates to algorithms that recursively break down a problem until the sub-problems can be solved directly. Afterwards, those partial solutions are merged to give the solution to the original problem. We can easily see, how this kind of problems could benefit from recursive parallelization strategies such as *Tree* or *Leaf*, reducing the time needed to create all the subproblems or making the smaller problem computation faster. Those strategies will be the ones explored in this document but first, a look to the specific problem is in order.

Multisort algorithm

In this assignment we have been tasked with the study of a sorting strategy that somewhat blends the well-known quicksort and mergesort algorithms. A brief explanation extracted from one of the documents referenced in the *bibliography* is as follows:

> *"Multisort is a sort algorithm which combines a "divide and conquer" mergesort strategy that divides the initial list into multiple sublists recursively, a sequential quicksort that is applied when the size of these sublists is sufficiently small, and a merge of the sublists back into a single sorted list"*

We can deduce that the amount of parallelization code in this algorithm is quite substantial due to the recursive nature of the merge and quick-sort algorithms, allowing us to implement the strategies presented in the next section (every recursive call is a potentially parallelizable structure).

Parallelization strategies

To understand the considered parallelization strategies, first we will briefly analyze the sequential code without any parallel constructs (the code sections irrelevant to the parallelization of the program will be overlooked, and only explained in words to avoid unnecessary information).

```
void merge(...) {
        if (length < MIN_MERGE_SIZE*2L) {  // Base case
                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(...) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition (1/4 of the vector for each)
                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 quarters to obtain the two halves of the vector
                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);

                // Final merge of the two halves into one
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
      } else {
            // Base case
            basicsort(n, data);
      }
}
```

The program can be easily understood and the structure reflects the explanation done in the previous section so we will not be delving deeper in this part of the code. There is also another chunk of the program relevant to the study of this problem, namely, the creation of the vectors that get sorted by the code shown above. Those methods will be explored and explained in another section of this document (see *Optional 1*) where we will analyze the potential parallelism opportunities they present.

As for the strategies proposed to parallelize the multisort and merge functions, we will use the following ones:

1. **Leaf:** there is a task for each “base case” call, meaning a task for each execution of the lowest recursive level. In our case, a task will be created every time the *basicsort* or *basicmerge* functions are called.
2. **Tree**: there is a task for each recursive call except for the base calls. In our code, we will create tasks each time a *multisort* or *merge* functions are called.

In the following sections we will study, among other things, the effects each strategy has on the performance of the program and the results we get using the different constructs offered by OpenMP to create parallel applications (and how each strategy behaves using those different techniques).

Expected results

Before starting with the performance evaluation of each strategy and OpenMP directive, we want to put some hypothesis forth based on our current knowledge. This way, at the end of this assignment, we can evaluate if our initial thoughts were on point and if we learnt something new during the realization of the exercises solved in this document.

To be able to state our expected results, first we need to explain the parallelization strategies used (which is done in the *previous section* so we will not repeat it here) and the OpenMP directives we will be using:

- Tasks: a basic construct to create parallelization using OpenMP, used in all the implementations in this document.
- Taskwaits: a directive that "*specifies a wait on the completion of child tasks generated since the beginning of the current task*" (*OpenMP reference guide*).
- Task dependencies: a directive that allows the specification of inter-task dependencies.

With all those explanation, we can finally propose our initial hypothesis regarding the expected results of the experimentation that we will carry on the next sections of this document:

1. There is a high probability of the tree strategy producing better results than the leaf one, due to the nature of each technique. That is because the base cases of a recursive program usually represent a small fraction of the execution time while the recursive call are often more relevant in terms of computation time. Because of this difference, and based on Amdahl's law, we can deduce that a program that uses a tree strategy will be able to parallelize a greater part of the code than one that relies on the leaf technique, thus being able to produce better results.
2. The codes that use task dependencies (directive *depends*) instead of *taskwaits* will have a better performance because with the former, only the needed tasks are halted while with taskwaits every child task is stopped, thing that will generate a bigger synchronization overhead.

At the *conclusions* section, we will confirm or deny these hypothesis based on the results we get, and in case we are wrong, we will try to explain and justify why.

Performance evaluation

Analyzing potential parallelism with Tareador

First we will analyze the potential parallelism the two proposed strategies have. In order to accomplish this, Tareador will be used (tool to study the time and load of different tasks before the program is truly parallel). These section will be divided in two different parts, one for each parallelization strategy proposed above.

Leaf

As explained, the leaf strategy will generate one task each time a base case function is called (*basicmerge* and *basicsort*). That means that the only calls to Tareador API will be on the leafs of the recursion tree (hence the name of the technique). The relevant code to allow Tareador to simulate the parallelism is shown below:

```
void merge(...) {
        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 ...
        }
}

void multisort(...) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition ...
        }
        else {
            // Base case
            tareador_start_task("basicsort");
            basicsort(n, data);
            tareador_end_task("basicsort");
        }
}
```

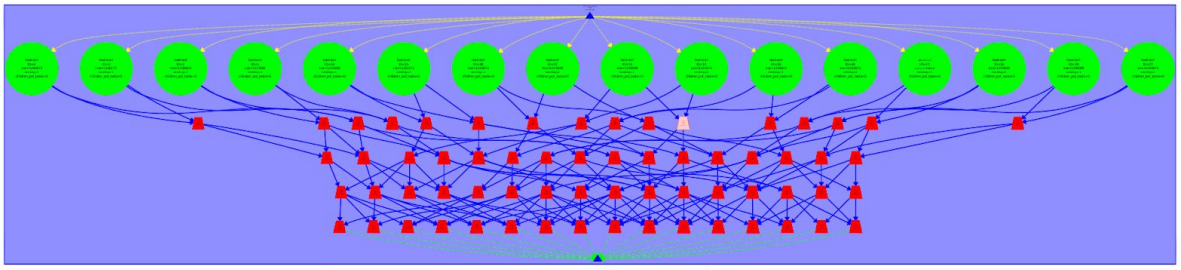

Figure 1. *Dependency graph generated by Tareador using the Leaf parallelization.*

In figure 1 we can observe the dependency graph generate by Tareador using the code previously shown. In this decomposition each task do not have more subtasks inside it. We can observe that all basic sort tasks are independent from each other and that all merge tasks depends from two previous tasks (each merge needs at least a couple sorted vectors to generate a bigger one). This graph can be useful to know where to place the taskwaits (each time a basicmerge call is done, we need the corresponding basicsort calls finished) and the task dependencies (similar to the other case), making easier the creation of an error-free implementation.

In the simulation of leaf strategy in Tareador we can observe the order of creation of tasks related to the total execution time. As we can see in the figure below, most tasks are created almost at the end of the execution of the program, while all threads except the first, remain idle during all this time. That is one of the problems of the leaf strategy; the tasks are created sequentially, losing potential parallelism in some programs. This fact, make us more sure that our first hypothesis was right (see *Expected results*).

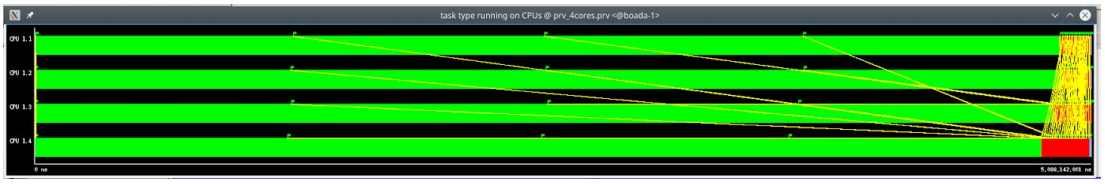

Figure 2. *Simulation of an execution of the Leaf strategy obtained with Tareador (yellow indicates the creation of tasks).*

Tree

As mentioned in the parallelization strategy section, the tree technique will generate a task each time a recursive call is done. In our program that is a task for each *merge* and *multisort*. The difference in concept with the leaf strategy is that the tree generates a task for each node of the recursive structure, making the creation of tasks also parallel. The relevant code that allows Tareador to simulate this strategy is shown below:

```
void merge(...) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                basicmerge(n, left, right, result, start, length);
        }
        else {
                // Recursive decomposition_
                tareador_start_task("merge_1");
                merge(n, left, right, result, start, length/2);
                tareador_end_task("merge_1");

                tareador_start_task("merge_2");
                merge(n, left, right, result, start + length/2, length/2);
                tareador_end_task("merge_2");
        }
}
```

```
void multisort(...) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                tareador_start_task("multisort_1");
                multisort(n/4L, &data[0], &tmp[0]);
                tareador_end_task("multisort_1");

                //Multisort is called three more times

                tareador_start_task("merge_in_sort_1");
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                tareador_end_task("merge_in_sort_1");

                tareador_start_task("merge_in_sort_2");
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
                tareador_end_task("merge_in_sort_2");

                tareador_start_task("merge_in_sort_3");
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
                tareador_end_task("merge_in_sort_3");
      } else {
            // Base case
            basicsort(n, data);
      }
}
```

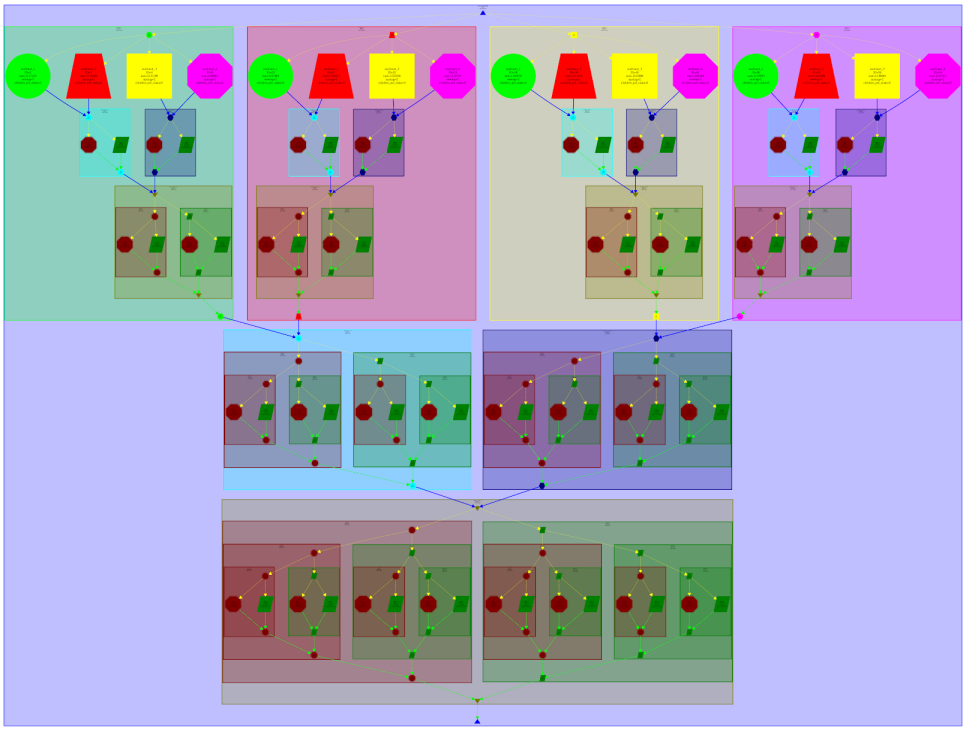

Figure 3. *Dependency graph generated by Tareador using the Tree parallelization.*

In the image above we can see how Tareador divides the different tasks of the tree strategy multisort (creating a dependency graph). Note that inside big tasks there are a number of smaller tasks, that happens because in tree strategy tasks are generated by recursion until they get to the base case. We can easily observe a pattern in the structure of the tasks, each "little square" (corresponds to a merge) has a couple of tasks that need to be done (merge inside of a greater merge). We can also see how all the multisort tasks are the first that are created (top layer). As before, this graph can help us implementate this strategy because now we know the dependencies between the different tasks. We can conclude that the tree strategy allows for a greater degree of parallelization which should make it better performance wise, so again, that agrees with the first hypothesis raised in this document.

Comparison of times and speedups between strategies (Tareador)

To finish with the Tareador based analysis of our strategies, we have made a simulation of each technique with the same parameters to see the execution time and the speedup in relation to the sequential execution using a different number of processors. The results of those experiments can be seen below:

Leaf strategy time and speedup table (varying number of CPUs)							
#CPUs	**1**	**2**	**4**	**8**	**16**	**32**	**64**
Time	20,3	10,14	5,09	2,55	1,29	1,29	1,29
Speed-up	1	2,002	3,988	7,961	15,736	15,736	15,736

Tree strategy time and speedup table (varying number of CPUs)							
#CPUs	**1**	**2**	**4**	**8**	**16**	**32**	**64**
Time	20,3	10,17	5,086	2,55	1,29	1,29	1,29
Speed-up	1	1,996	3,991	7,961	15,736	15,736	15,736

Unexpectedly, both tables are really similar, but we believe that is due to an error produced in Tareador or the size of the input. We know the results from those table are not right because in the following sections we have seen the differences between strategies when the real implemented code is tested. Knowing the high probability of those tables to be some kind of error, we can neither confirm nor deny the first hypothesis proposed initially.

Shared-memory parallelization with OpenMP tasks

Now that the dependencies between tasks have been established, we can start implementing each version using OpenMP directives. In this section the task and taskwaits will be used (although taskgroups could produce the same code) and as before, we will divide the content in two different parts, one for each strategy proposed.

Leaf

The code used to implement the leaf strategy using task constructs is 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);
      }
}
```

In the leaf strategy, the tasks are only created in the base case of the recursion, to ensure the correct execution of the program we need to do a taskwait before calling the two different merges, that is because merges can not start until all multisort calls are done. The second taskwait is needed because the last merge depends on the two previous ones. All this information can be extracted from the dependency graph obtained with Tareador.

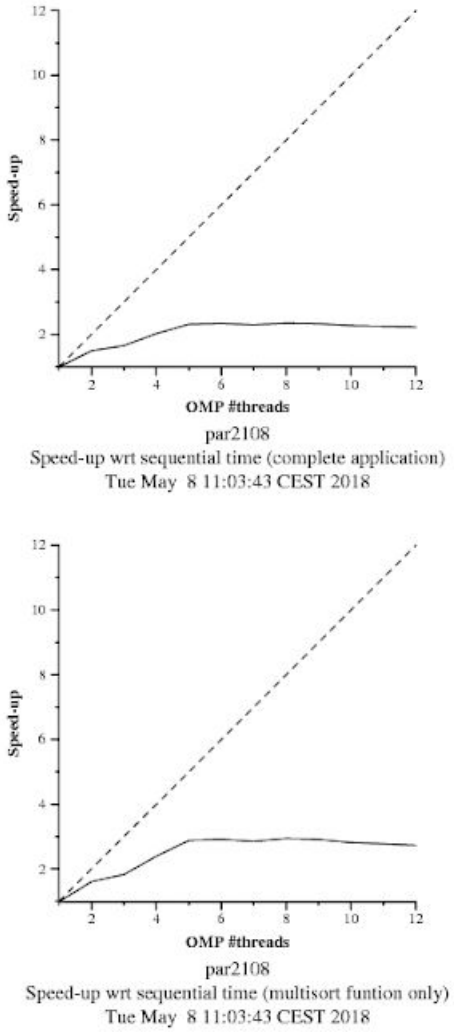

Figure 4. *Results of the submit-strong-omp.sh. Strong scalability plots of the Leaf parallelization strategy.*

This figure above depicts the strong scalability plot of the leaf strategy, showing the behaviour of our code when using a different number of processors. We can see that the speed-up is far below the theoretical line (that indicates perfect parallelism), thing that happens due to a couple of reasons. The first one is that in leaf strategy all the tasks are generated by one thread so there is a load unbalance between threads (the first one does a ton of work while the others wait for tasks to be created), and the second reason is that there are chunks of code that are sequential (the functions that initialize the vectors used in the multisort function), that limit the parallelization of the program. This sequential part of the code and the the impact it has on the overall performance can be seen in Figure 5 .

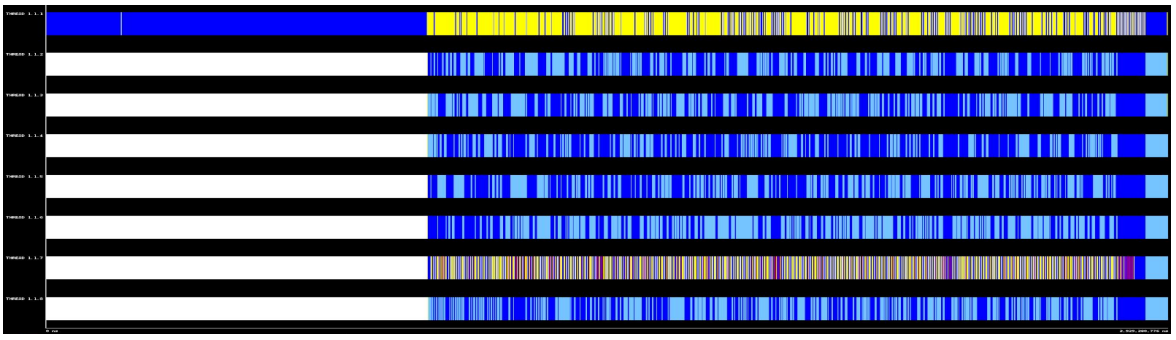

Figure 5. *Trace of the execution using Paraver of the Leaf strategy code.*

In this image (trace of an execution analyzed with Paraver) we can observe, aside from the sequential part of the code, the creation of tasks done by the first thread, and the amount of synchronization this strategy has (light blue depicts this overhead).

Tree

The piece of code relevant to the implementation of the tree strategy using tasks and taskwait directives can be seen 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
                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);
    }}
```

In this strategy the tasks are generated recursively so before any recursive call we create a new task. We need to do taskwait in the same way as we have done on the leaf strategy, but in this case we also need one task wait at the end of the last merge to preserve the dependency of different recursion levels (if a level of recursion above the one we are executing tries to take the results of our unfinished execution, data coherence and cohesion will be broken). As before, the positioning of taskwaits can be deduced from the Tareador dependency graph.

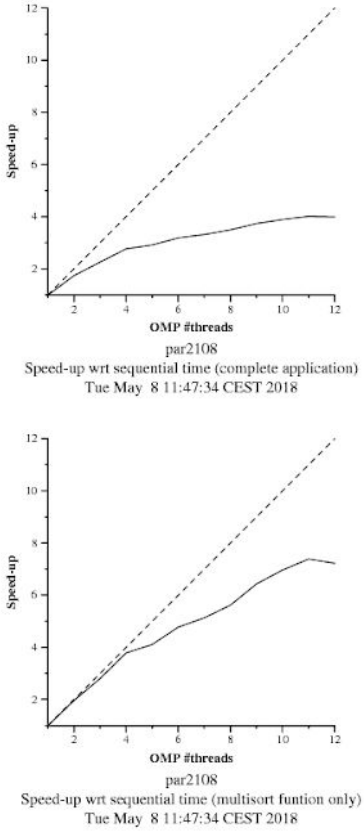

Figure 6. *Results of the submit-strong-omp.sh. Strong scalability plots of the Tree parallelization strategy.*

In figure 6 the plot that show the strong scalability of the Tree code presented can be seen. If we compare the multisort plot with the results obtained with the Leaf strategy (also the multisort), we can see that the speedup on Tree is substantially better than on Leaf as the number of processors grow (Tree strategy scales better than the Leaf one). The problem is that the complete application has a similar speedup on both strategies, which is the product of the sequential code on both cases (initialization of vectors mainly). As before, this sequential chunk of code can be seen in the figure below:

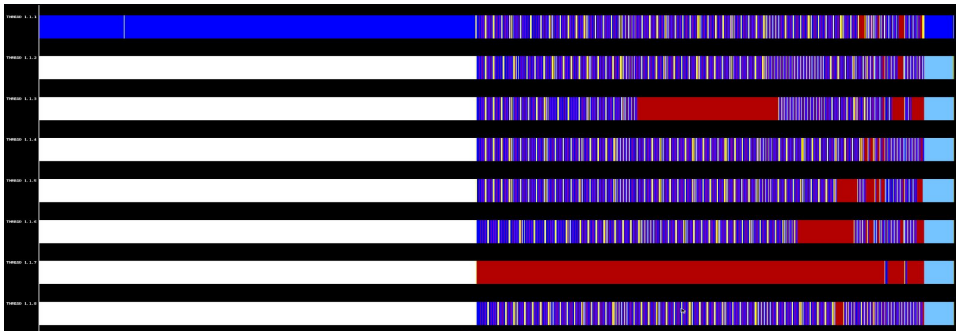

Figure 7. *Trace of the execution using Paraver of the Tree strategy code.*

In this figure, we can also see something strange happening with thread number 6, that is stuck in synchronization for most of the execution. This is not a problem of the program but one of visualization; if we zoom in, we can see the thread 6 is doing some amount of work but due to Paraver antics, if we take a full picture of the execution only the synchronization is shown.

As a conclusion of this section, we have seen that the tree strategy creates a code that scalates and performs slightly better than the leaf technique, fact that validates in part our initial hypothesis.

Optional 1: parallelizing the sequential sections

The objective of this section is to solve the problem that we have seen repeatedly before; there is a big part of sequential code at the start of the execution that hampers our parallelization efforts [Amsahl's law]). To solve this we need to parallelize the functions initialize and clear. The code to do so is shown below:

```
static void initialize(long length, T data[length]) {

   int chunk_size;
   #pragma omp parallel
   {
            chunk_size = length/omp_get_num_threads();

            int thid = omp_get_thread_num();
            int aux = thid*chunk_size;
            for (long i = aux; i < (thid+1)*chunk_size; ++i) {
                  if (i == aux) {
                        data[i] = rand();
                  }
                  else {
                        data[i] = ((data[i-1]+1) * i * 104723L) % N;
                  }
            }

      }

}

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

In the initialize function we start a parallel section and we divide the work in chunks of the same size for each thread (chunks meaning groups of iterations). We can see that each iteration sets a random value in the vector, and if we ignore the first one, each depends on the previous one. To make the parallelization possible, we need to ensure that the first iteration of each chunk has a value, that's why we set a random value at the start of each group.

To parallelize clear function, we have chosen to implement a for with static scheduling and a chunk size length of the vector divided by the number of threads. That is because in this function the only relevant work is to put all elements to 0, so we do not need a complex structure.

This is the scalability plot of the application with the new improvement:

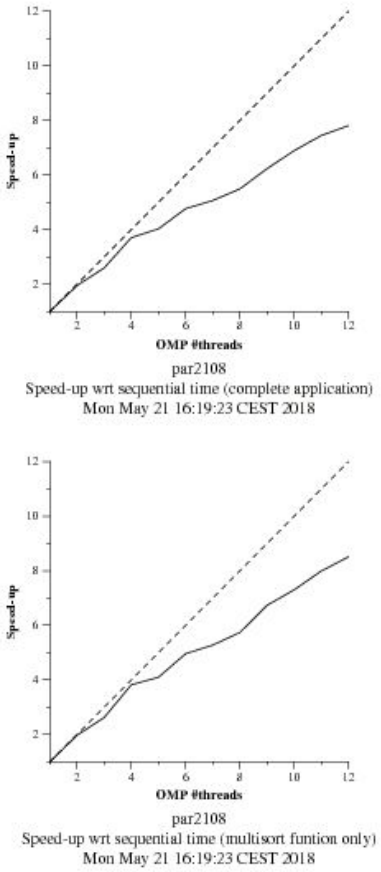

Figure 8. *Results of the submit-strong-omp.sh. Strong scalability plots of the Tree parallelization strategy with the initial functions parallelized.*

As stated before, we knew that the sequential part of the code was holding back the performance of our program, but with most of the code now parallelized the general speedup has improved a lot. Now it is difficult to appreciate the difference between the multisort function and the overall speedup, due to the sequential part of the program being reduced. We can see in figure 9 how much sequential code is still in our code after this last modification.

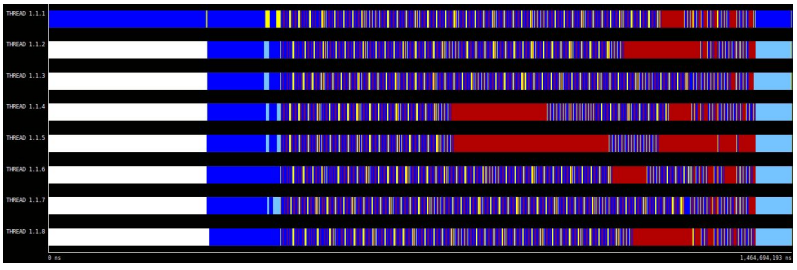

Figure 9. *Trace of the execution using Paraver of the Tree strategy code with the initial functions parallelized.*

These results emphasize the significance of having as less sequential code as possible when parallelizing a program as stated by Amdhal's Law.

Parallelization using task dependencies

In this section we will parallelize the same code using mostly task dependencies (through the OpenMP directive *depend*). This way we can reduce the synchronization overhead of the previous versions because each task will only wait for those that are needed specifically, not a general wait like the one produced by *taskwait.* We will also analyze if the second hypothesis proposed initially is true or not and the reason behind the answer. In this case the section will no be divided in two different parts because only the tree strategy has been implemented (task dependencies in the leaf technique would make little sense).

The relevant code implementing what has been explained is shown below:

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

Let's analyze the code briefly: the first merge call depends on the first and second multisort tasks (this can be deduced by the range of the vector each function is having an effect on), the second merge call depends on the third and fourth multisort tasks and the last merge depends only on the first and second merge tasks (it is a general merge that unifies all the work done before). At the end of the recursion decomposition we need a taskwait to ensure the correctness of the whole application (avoiding problems between levels of recursion as explained before).

The following figure depicts the scalability plot of this code:

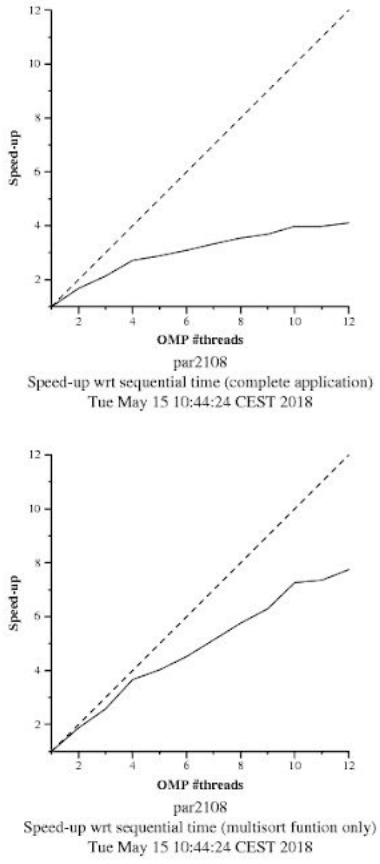

Figure 10. *Results of the submit-strong-omp.sh. Strong scalability plots of the Tree parallelization strategy using task dependencies.*

Leaving aside the plot of the complete application (it does not show a substantial improvement in relation to the task and taskwait plot), the speed-up plot of the multisort function is slightly better

than using the tree strategy with taskwaits. That happens because, as we initially believed, with depend we do not have to wait for all tasks to finish, just some of them. That validates in part the second hypothesis we proposed initially, if in an underwhelming manner (we thought that the difference would be greater if the number of task dependencies was big enough).

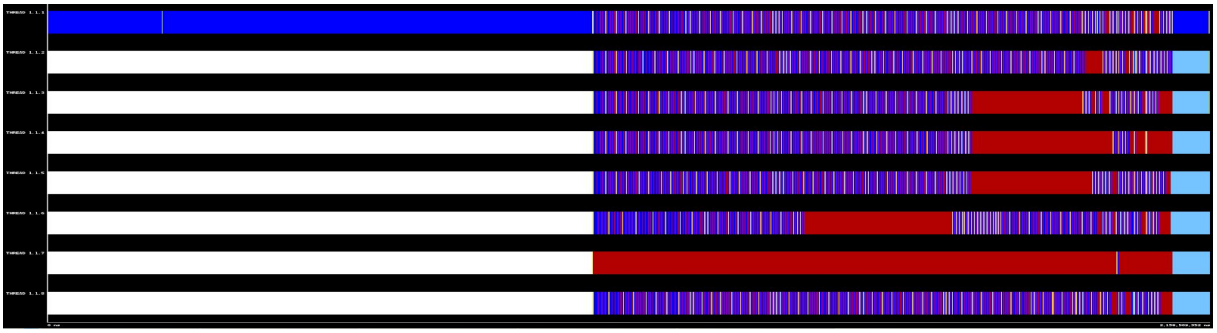

Figure 11. *Trace of the execution using Paraver of the Tree strategy code parallelized with task depends.*

Optional 2: best values for size parameters

In this section we are tasked with finding the best possible values for the `sort_size` and `merge_size`, in order to do so we have used the script **submit-depth-omp.sh**, that tries a set of different values (for the parameters we wish to investigate) and prints a plot with the execution time of the program in each case. First we have run this script to obtain the best possible value for the `sort_size` and we have gotten these plot result:

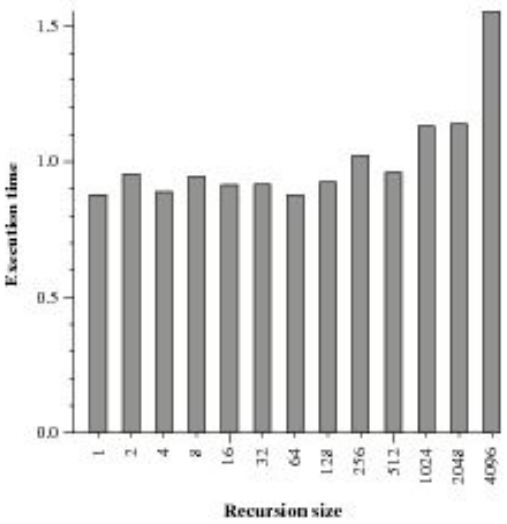

Figure 12. *Plot obtained by using the submit-depth-omp.sh, that uses different values for the sort_size parameter and plots the execution time of the resulting program.*

So we assume that the best value for `sort_size` is **64** (we could have chosen other values as some differences are minimal). Then we do the same to know the best possible value for `merge_size`, to do that we fix the sort_sieze value and iterate for all possible values for `merge_size.` The relevant code of the modified script is shown below:

```
sort_size=64
set depth_list = "1 2 4 8 16 32 64 128 256 512 1024 2048 4096"
#setenv merge_size 512
set i = 1

set out = $PROG-$OMP_NUM_THREADS-depth.txt
...
foreach depth ( $depth_list )
      echo $depth >> $out
      ./$PROG $size $sort_size $depth >> $out
```

We execute this new script with the variable `sort_size = 64` and we get these plot result:

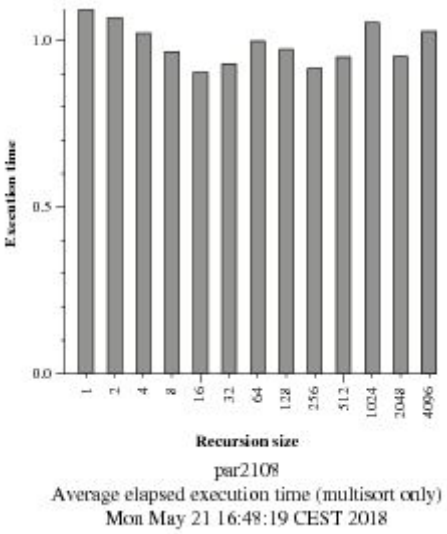

Figure 13. *Plot obtained by using the submit-depth-omp.sh, that uses different values for the merge_size parameter and plots the execution time of the resulting program.*

So we assume that the best value for `merge_size` is **16** (256 could also be a valid choice). Finally to compare the results of the best values and the default ones we have made a plot to study the speedup of both programs; the left plot below corresponds to the scalability plot of multisort with task calling it with default values (`sort_size = 32` and `merge_size = 512`) and the right one corresponds to the same program calling it with our optimal values (`sort_size = 64` and `merge_size = 16`).

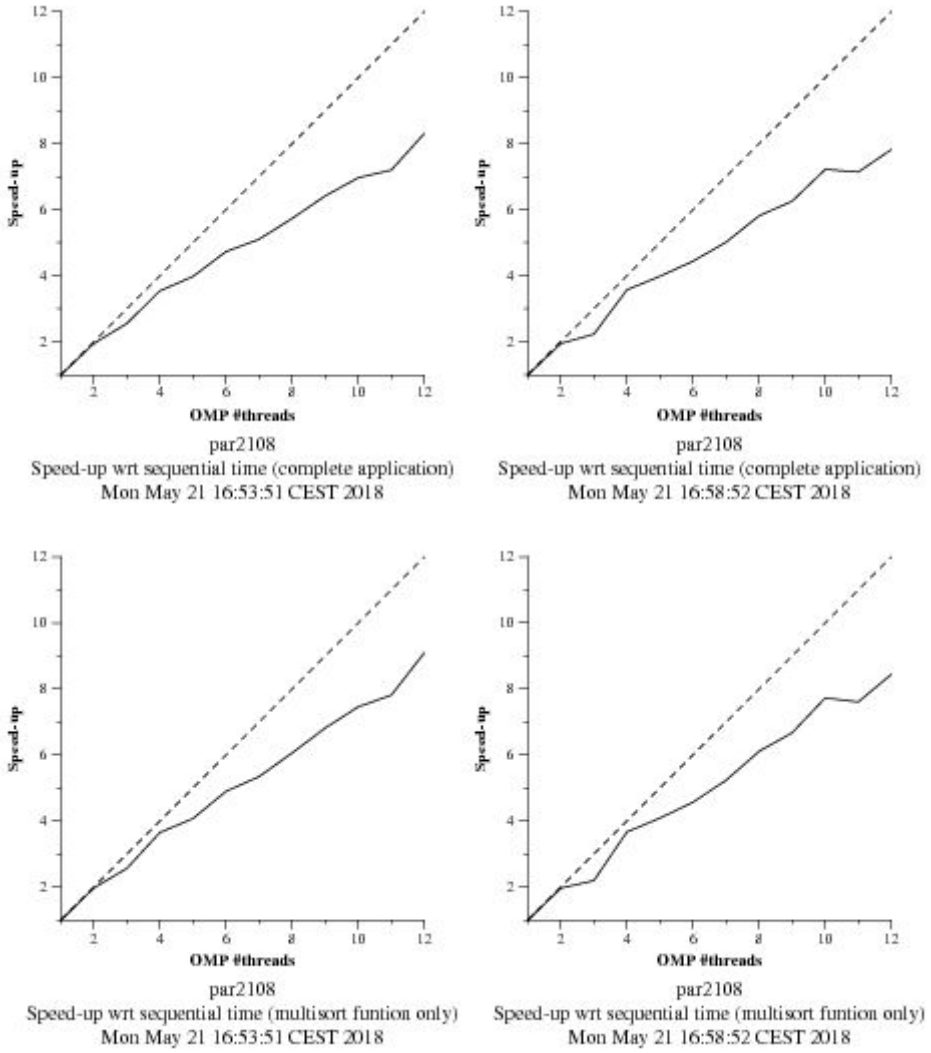

Figure 14. *Results of the submit-strong-omp.sh. Strong scalability plots of the Tree parallelization strategy. On the left, program with default values, on the right, with optimal values.*

We can not see a lot of improvement but theoretically these values are the ones that give the best speed-up (according to the tests). We can conclude that the default values used by the program are close (performance wise) to the optimal ones, but the difference with the worst possible values is significant.

Conclusions

In this Laboratory Assignment, we have learnt about different recursive parallelization strategies using the multisort algorithm and we have analyzed, among other things, the scalability, performance and differences between them. We have also tested the some of the directives offered by OpenMP to create inter tasks dependencies such as taskwait or depend.

On one hand, we have studied the difference in performance when using a Leaf or Tree strategy. As can be seen in our results and the comments we have done, using a the tree technique allows more code to be parallelized (task creation process parallelize) thus making it better performance wise. That does not mean that the leaf strategy should not be used, but in general, will get worse results than the tree one.

In the other hand, we have analyzed the differences in execution time using two different OpenMP directives: a taskwait-based approach or a depends one. We have concluded that the results are fairly similar in both cases with a slight improvement for the depends code, probably due to less synchronization overhead as a result of task having to wait just for the ones that they need, instead of all of them.

We have also seen the significance of having as much code as possible parallelized and the impact it can have in the performance of our programs and also the, somewhat less relevant choice of sizes when it comes to determining chunks of data to be parallelized (though we know that some values could force an almost sequential program, fine tuning those parameters is not as important as using as less sequential code as possible).

Finally, we can confirm our initial hypothesis presented on the *Expected results* section. We have seen that the tree strategy yields better results and the depends approach works better if not as much as expected.

Bibliography

The links below have been consulted to shape this document and provide more information about the topics discussed:

- E. Ayguadé, J. Corbalán, J. Morillo, J. Tubella and G. Utrera (2017). *Lab 4: Divide and Conquer parallelism with OpenMP: Sorting*. Spain: UPC. [Date of reference: 20/05/2018]. Available on: <http://atenea.upc.edu/pluginfile.php/2247200/mod_resource/content/10/lab4-PAR.pdf>
- Wikipedia (2018). *Divide and conquer algorithm*. Estados Unidos: Wikipedia. [Date of reference: 20/05/2018]. Available on: <https://en.wikipedia.org/wiki/Divide_and_conquer_algorithm>
- Wikipedia (2018). *Sorting algorithms*. Estados Unidos: Wikipedia. [Date of reference: 21/05/2018]. Available on: <https://en.wikipedia.org/wiki/Sorting_algorithm>
- Intel (2015). *OpenMP user and reference guide*. Estados Unidos: Intel. [Date of reference: 21/05/2018]. Available on: <https://software.intel.com/en-us/node/524542>