Laboratory Assignment 4

PAR - Q2 2019/2020

Marc Monfort Grau
Albert Mercadé Plasencia
Group par2302

April 16th 2020

Index

Introduction

In this report, we look at leaf and tree parallelization to evaluate their performance for the execution of the sorting program presented in to us in this assignment. We analyse the potential of both approaches using *Tareador* and *Paraver*. Then we implement them using OpenMP and introduce variations such as a cut-off mechanism and task dependencies to weigh if they bring any improvement. Furthermore, we explore the changes in strong scalability when executing our code in different architectures: boada-2 to 4, boada-5 and boada-6 to 8. We also attempt to parallelize the initialisation of the `data` and `tmp` vectors. Finally, we evaluate the performance of the different versions we have implemented to find the fastest executing one.

Tareador analysis

In this section, we will analyze the potential of both parallelization strategies, leaf and tree, with the help of *Tareador*. We start with leaf parallelization, Code 1 shows our *Tareador* instrumentation for it. We create tasks for every call to `basicmerge` and `basicsort`, as we aim to create a task for every leaf in the recursion tree.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                tareador_start_task("BasicMerge");
                basicmerge(n, left, right, result, start, length);
                tareador_end_task("BasicMerge");
        } else {
                // Recursive decomposition
                ...
        }
}

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

Code 1. Changes to `multisort-tareador.c` ***for leaf parallelization analysis***

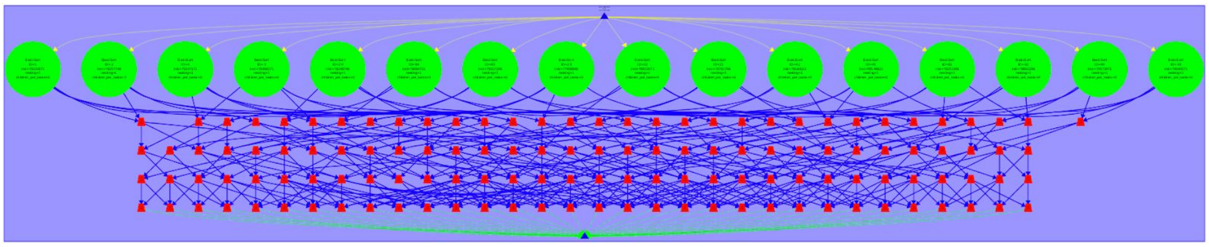


Figure 1. Tareador task dependency graph for leaf parallelization

Tareador provides us with the task dependence graph seen in Figure 1. It shows that while the calls to `basicsort` are fully parallelizable, the posterior calls to `basicmerge` aren't, as we find dependencies between them.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
                tareador_start_task("Merge4");
                merge(n, left, right, result, start, length/2);
                tareador_end_task("Merge4");
                tareador_start_task("Merge5");
                merge(n, left, right, result, start + length/2, length/2);
                tareador_end_task("Merge5");
        }
}

void multisort(long n, T data[n], T tmp[n]) {
        if (n >= MIN_SORT_SIZE*4L) {
                // Recursive decomposition
                tareador_start_task("MultiSort1");
                multisort(n/4L, &data[0], &tmp[0]);
                tareador_end_task("MultiSort1");
                tareador_start_task("MultiSort2");
                multisort(n/4L, &data[n/4L], &tmp[n/4L]);
                tareador_end_task("MultiSort2");
                tareador_start_task("MultiSort3");
                multisort(n/4L, &data[n/2L], &tmp[n/2L]);
                tareador_end_task("MultiSort3");
                tareador_start_task("MultiSort4");
                multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
                tareador_end_task("MultiSort4");

                tareador_start_task("Merge1");
                merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
                tareador_end_task("Merge1");
                tareador_start_task("Merge2");
                merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
                tareador_end_task("Merge2");

                tareador_start_task("Merge3");
                merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
                tareador_end_task("Merge3");
        } else {
                // Base case
                basicsort(n, data);
        }
}
```

Code 2. Changes to `multisort-tareador.c` ***for tree parallelization analysis***

In this second approach, using tree decomposition, we create a task at every step in the recursion, everytime we call `multisort` or `merge`. We can observe in Code 2, above, our implementation of this strategy in *Tareador*.

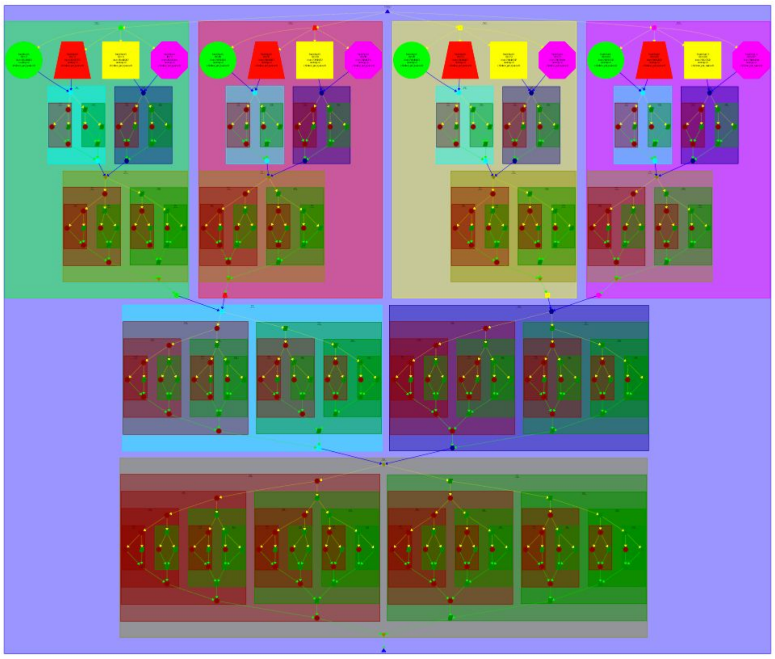


Figure 2. Tareador task dependency graph for tree parallelization

The output of *Tareador* to this approach is seen in Figure 2, which shows the task dependency graph for tree parallelization. As we can see, the code seems more parallelizable at first sight, but we'll see later, using *Tareador* simulations, how both strategies compare when looking at execution times.

Number of...	Leaf	Tree
Internal tasks	0	234
Computing tasks	144	144

Table 1. Number of internal tasks and tasks doing computation for leaf and tree parallelization

Table 1 shows us the number of tasks of the two types; internal meaning tasks that create new tasks and we call computing tasks those that do actual computations. It is important to note that for this execution of *Tareador* the input is `-n 32 -s 1024 -m 1024`, that is a list of size 32 Kiloelements with `MIN_SORT_SIZE` and `MIN_MERGE_SIZE` equal to 1024. As we would expect for leaf parallelization, we have 0 internal tasks, since we only create tasks for the leafs in the recursion tree, which are the tasks doing computations. Hence, all the tasks are computing tasks, 144 in total, as we only create tasks for the executions of `basicsort` and `basicmerge`. Furthermore, it doesn't come as a surprise that the number of computing tasks for tree parallelization is the same as for leaf, because the recursion tree is the same and the tasks doing the computations are still the leaves. The number of internal tasks, which allow us to traverse the tree in parallel and not sequentially like for leaf, is higher than that of computing tasks at 234, as we would expect.

Number of Threads	Leaf Execution Time	Leaf Speed-up	Tree Execution Time	Tree Speed-up
1	1,263,350,939,001 ns	1	1,263,350,939,001 ns	1
2	631,688,561,001 ns	1,9999	631,692,419,001 ns	1,9999
4	315,851,844,001 ns	3,9998	315,863,292,001 ns	3,9996
8	158,381,783,001 ns	7,9766	158,824,649,001 ns	7,9543
16	79,788,739,001 ns	15,8336	79,787,090,001 ns	15,8340
32	79,746,381,001 ns	15,8421	79,744,648,001 ns	15,8424
64	79,746,381,001 ns	15,8421	79,744,635,001 ns	15,8424

Table 2. Simulated execution time and speed-up by Tareador for leaf and tree parallelization

In Table 2 above, and more clearly below in Figures 3 and 4, we can observe the execution times and consequent speed-up simulated by Tareador for different numbers of threads.

Surprisingly, the executions times are almost identical, which we didn't expect as we thought tree parallelization would perform better. However, this might be explained by the overhead incurred in creating 234 more tasks with respect to leaf parallelization. Remarkable as well is the fact that beyond 16 threads we see no measurable improvement in execution times. Due to the size of the computation for this particular executions, at most we need 16 threads at the same time and hence having access to more doesn't provide better performance. Nonetheless, we do see up to that point an almost perfect speed up in both cases, which shows the great parallelization of this program.

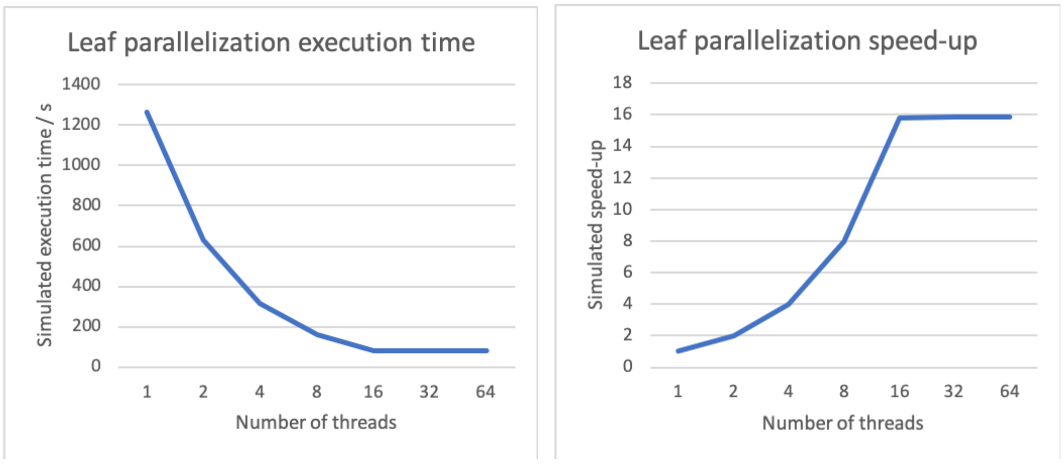


Figure 3. Simulated execution time and speed-up graphs for leaf parallelization

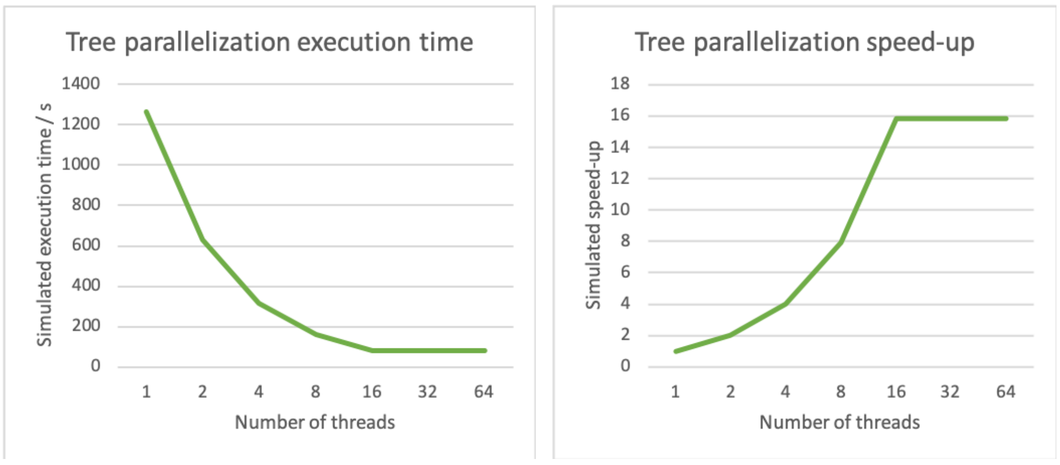


Figure 4. Simulated execution time and speed-up graphs for tree parallelization

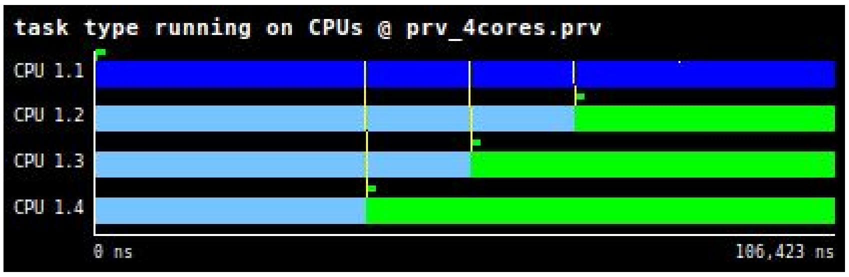


Figure 5. Start of the simulated leaf parallelization with 4 threads

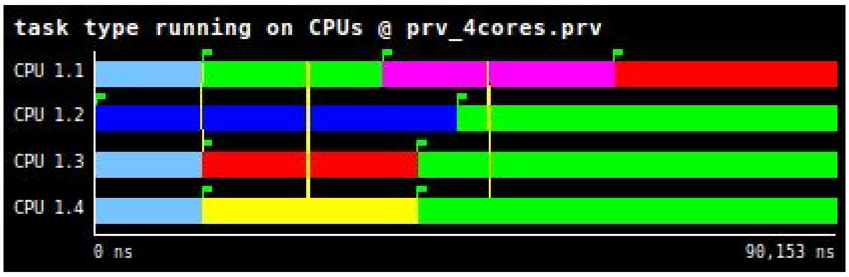


Figure 6. Start of the simulated tree parallelization with 4 threads

In Figures 5 and 6 we observe the start of the simulation with 4 threads for leaf and tree parallelization as shown in *Paraver*. The most observable difference is in the creation of tasks. When looking at leaf decomposition, tasks are not all created at the same time, they are created sequentially. On the other hand, the creation of tasks in tree decomposition is done in parallel as well and hence they are all created at the same time.

Parallelization strategies

First of all we have to create the parallel region before calling the `multisort` function. With `single` we limit to only one thread the creation of tasks. The rest of threads will be executing tasks that are being created and placed in the task pool.

```
#pragma omp parallel
#pragma omp single
multisort(N, data, tmp);
```

Code 3: Main function for both parallelizations

Leaf parallelization

As explained in a previous section, leaf parallelization consists in creating tasks for the leaves of the recursion tree. Thus, just as we did for *Tareador*, we create tasks for all calls to basicmerge and basicsort. Our OpenMP implementation can be found below in Code 4.

```
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);
	}
}
```

Code 4: OpenMP instrumentation for leaf parallelization

It is important to note the use of the `taskwait` constructs in order to add the necessary task synchronization to honour the dependencies on the calls to `merge`. This is the reason why we have added the `taskwait` after the four recursive `multisort` calls and again before the last `merge` call that unites the vectors resulting from the two previous calls to `merge`.

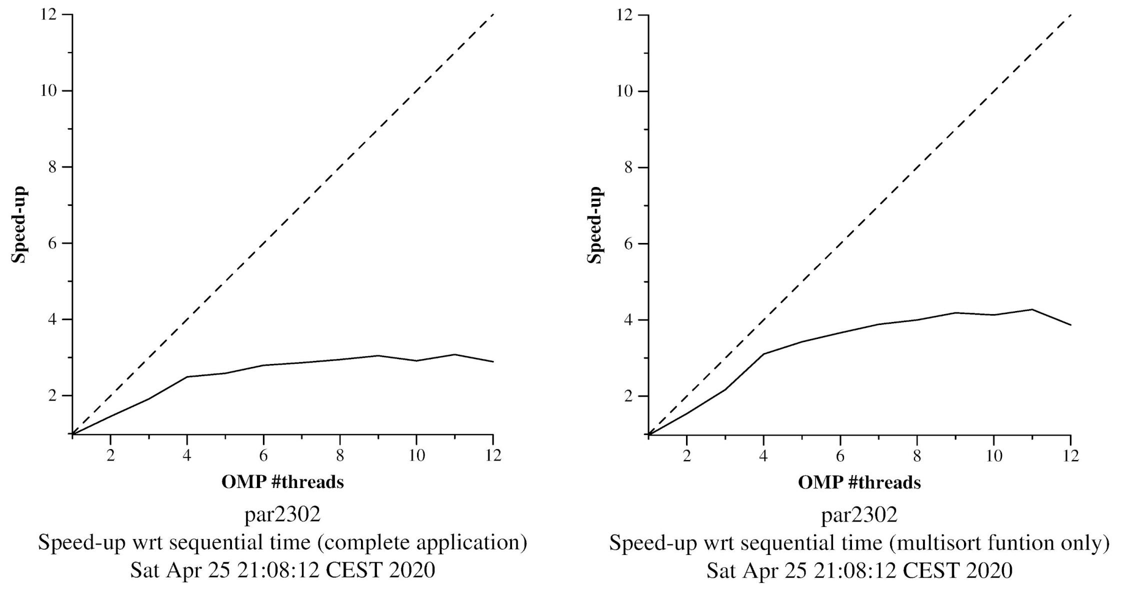


Figure 7. Strong scalability for leaf parallelization

As it's shown in Figure 7 above, this strategy have very poor scalability. The problem is the lack of parallelisation in the creation of tasks. A single thread has the job of creating all the tasks when the program reaches the base case. So all the tasks are created sequentially, and that implies a huge loss in performance. The slightly better scalability of the `multisort` function is explained by the fact that it doesn't take into account the rest of the code, which runs sequentially.

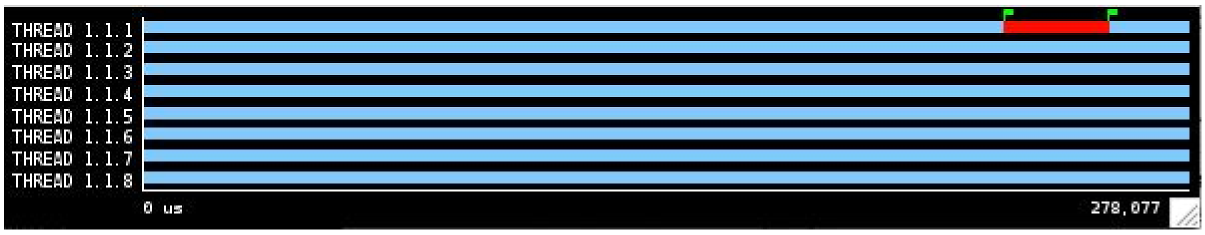
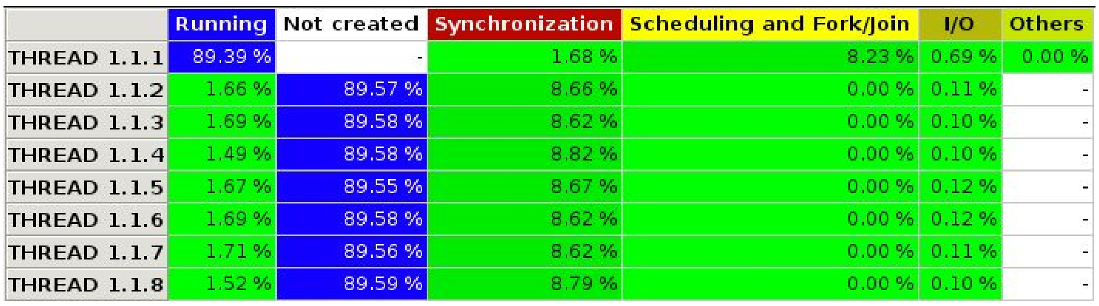


Figure 8. `omp_parallel_constructs.cfg` ***for leaf parallelization***



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	89.39 %	-	1.68 %	8.23 %	0.69 %	0.00 %
THREAD 1.1.2	1.66 %	89.57 %	8.66 %	0.00 %	0.11 %	-
THREAD 1.1.3	1.69 %	89.58 %	8.62 %	0.00 %	0.10 %	-
THREAD 1.1.4	1.49 %	89.58 %	8.82 %	0.00 %	0.10 %	-
THREAD 1.1.5	1.67 %	89.55 %	8.67 %	0.00 %	0.12 %	-
THREAD 1.1.6	1.69 %	89.58 %	8.62 %	0.00 %	0.12 %	-
THREAD 1.1.7	1.71 %	89.56 %	8.62 %	0.00 %	0.11 %	-
THREAD 1.1.8	1.52 %	89.59 %	8.79 %	0.00 %	0.10 %	-

Figure 9. `omp_state_profile.cfg` ***for leaf parallelization***

Above in Figures 8, we observe that only a very small region of the execution is done in parallel and when we look at Figure 9 we observe that for most of the time, besides the master thread, the threads aren't created and they are only actually running for around 1.60% of the time. This shows that we are under using the resources at our disposal. The reason behind this behaviour might be that since we are creating tasks sequentially, we are losing plenty of time traversing the recursion tree and thus the tasks creation isn't fast enough to feed all processors at the same time.

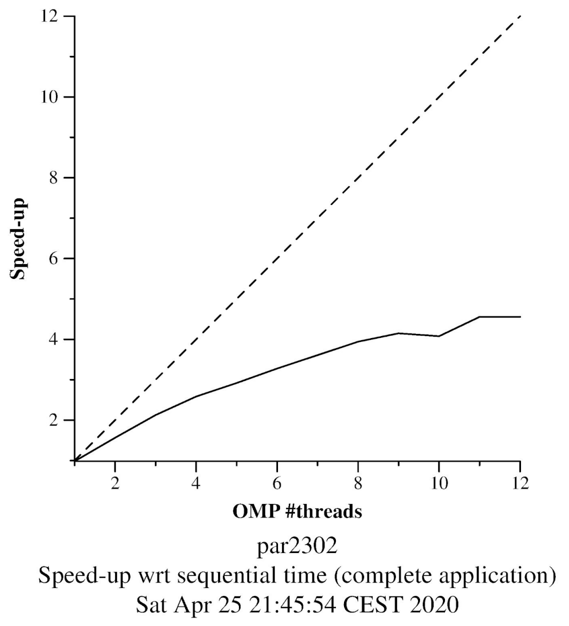
Tree parallelization

For this approach, we create a task for each recursive call we do, whether it is `multisort` or `merge`. Similarly to leaf parallelization, we have to introduce some sort of task synchronisation to guarantee the correct execution of the parallel computation. In order to acheive that, we have used the `taskgroup` clause twice. First to group all the recursive calls to `multisort` and secondly to group the two `merge` calls after that. This way we ensure that vectors to be merged have already been sorted. The resulting implementation can be observed below in Code 5.

```
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);
       }
}

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

```
            #pragma omp task
            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
      } else {
            // Base case
            basicsort(n, data);
      }
}
```
Code 5. OpenMP instrumentation for tree parallelization



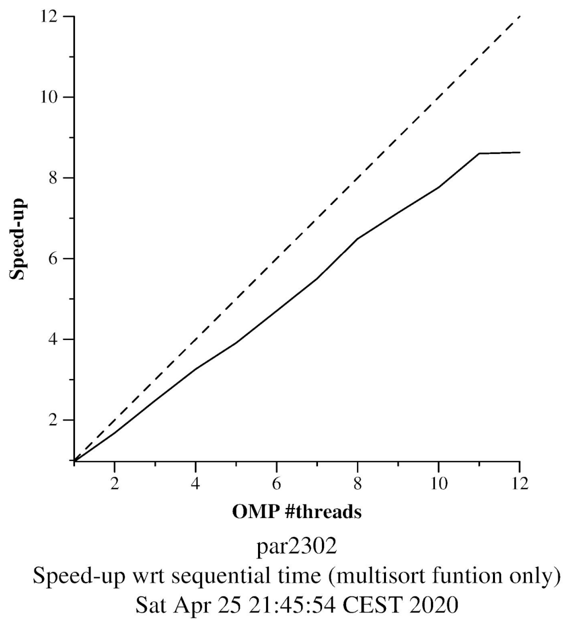


Figure 10. Strong scalability for tree decomposition

The strong scalability graph for the multisort function, shows a huge improvement over leaf decomposition. This is explained due to the fact that tasks are now created in parallel and no longer sequentially, which might introduce extra task creation overheads but is clearly more than offset by the performance gained.

The complete application graph, while it certainly has improved with respect to leaf parallelisation, still doesn't show very good scalability. That might be because of the sequential functions responsible for initializing the vector `data` and clearing the vector `tmp`.

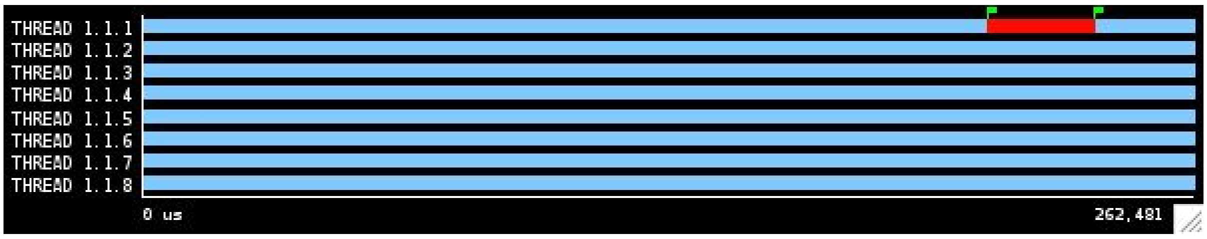
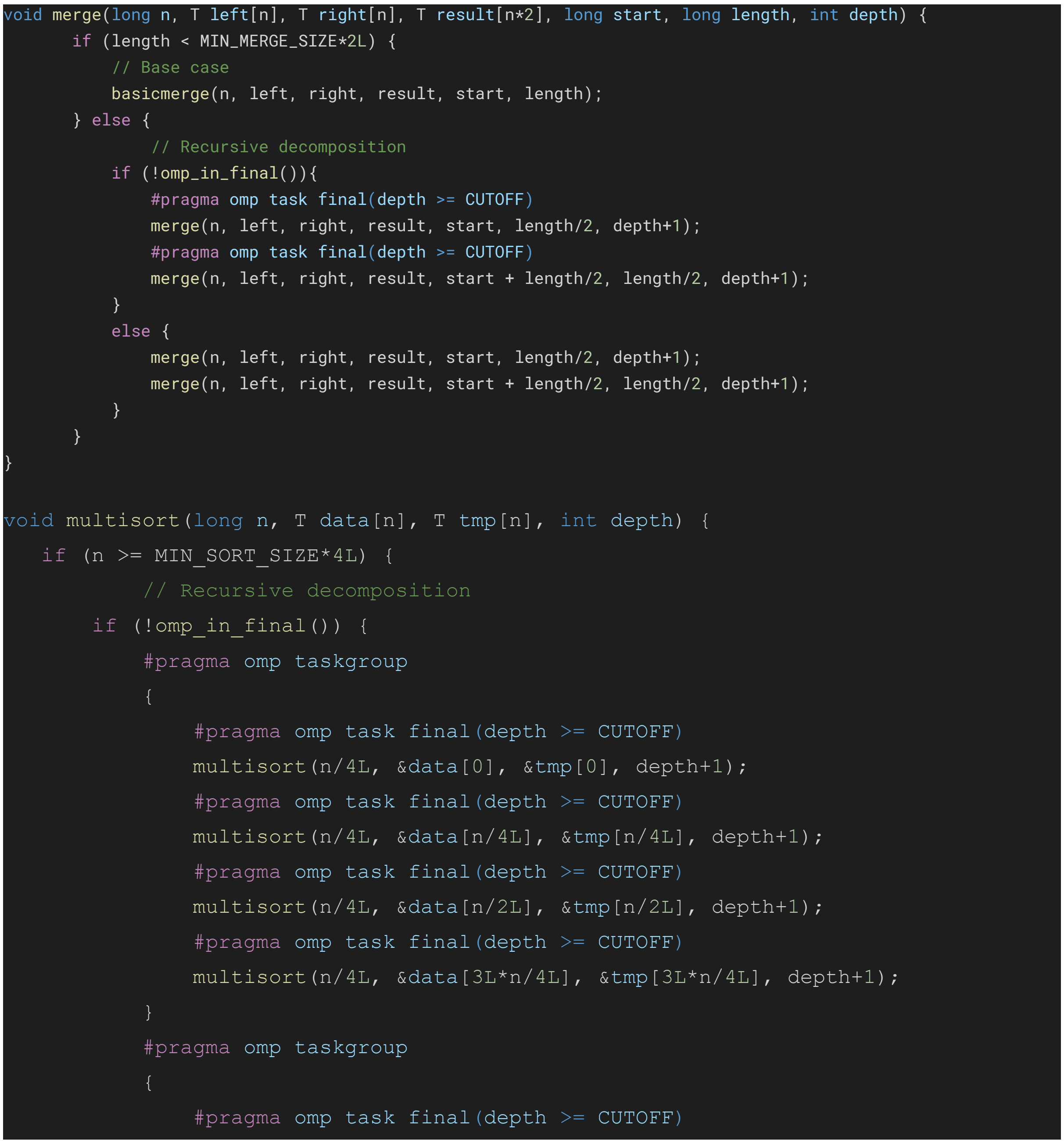


Figure 11. `omp_parallel_constructs.cfg` ***for tree parallelization***

Equivalently to leaf parallelization, in this case we also observe, above in Figure 11, that the parallel region of the execution is quite small and thus confirms our hypothesis that the bad scalability of the complete application is caused by the sequential fraction of the execution being very large.

Tree parallelization with cut–off mechanism

To implement a cut-off mechanism we have to add a parameter to both `merge` and `multisort` functions that represents the current depth level of the recursion tree. In each new call we will increment the value of depth by 1. We also add the `final` clause to the creation of tasks, which allows us to control whether the `CUTOFF` threshold has been reached, through the `omp_in_final` method, and thus we stop creating tasks.



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

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

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

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

            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);
        }
    } else {
        // Base case
        basicsort(n, data);
    }
}
```
Code 6. OpenMP instrumentation for tree parallelization with cut-off mechanism

The different cut-off levels allow us to control the number of tasks created. This can be observed very clearly if we look at Figures 12 and 13 below. We set `CUTOFF` equal to 0 and hence, in the first call to `multisort` the cut-off level is reached. As such, only 7 tasks are created: 4 calls to `multisort` and 3 to `merge`. If we look closely in Figure 12, we can count the 7 tasks, represented by the blue regions inside the parallel section of the execution. When we change the cut-off level to 1, we create the same 7 tasks in the first recursion level and then another 4x7 + 3x2 = 34 tasks, as for every call to `multisort` we create 7 tasks and to `merge` 2 tasks. In total that gives us 41 tasks created, which are the ones shown in Figure 13.

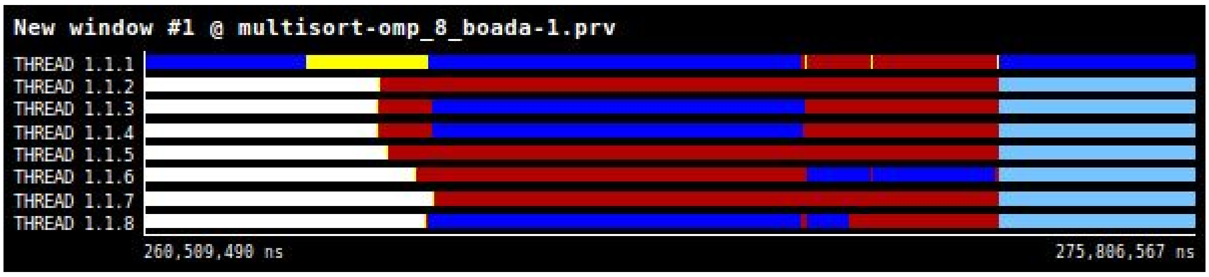


Figure 12. Paraver timeline for** `multisort-omp-tree-cut-off.c` **execution with** `-c 0` **and 8 processors (zoomed in where tasks are created)

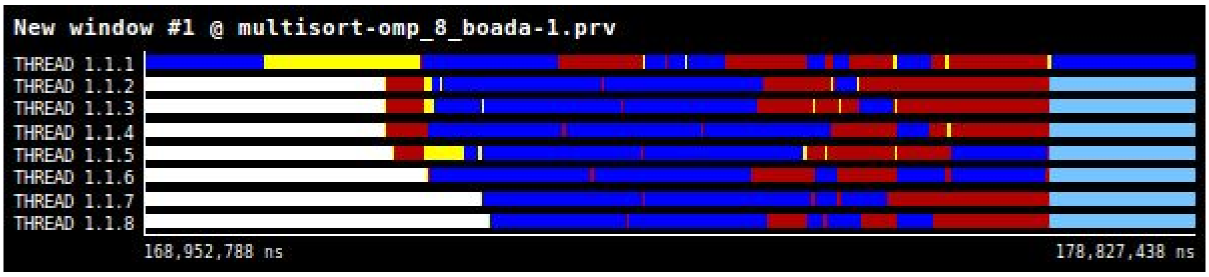
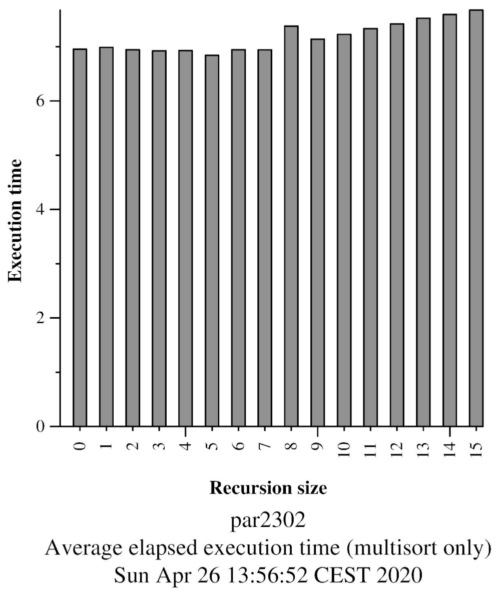
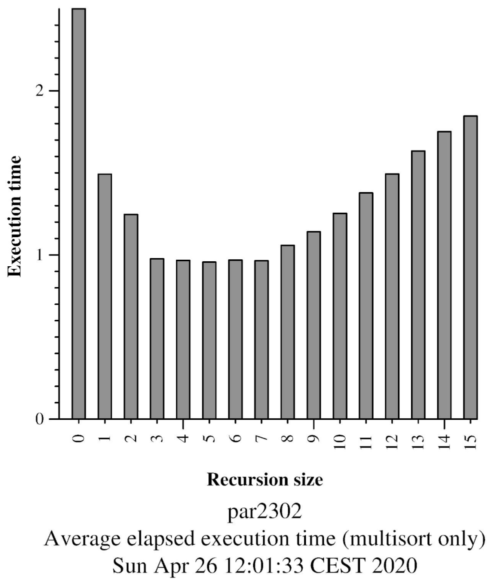
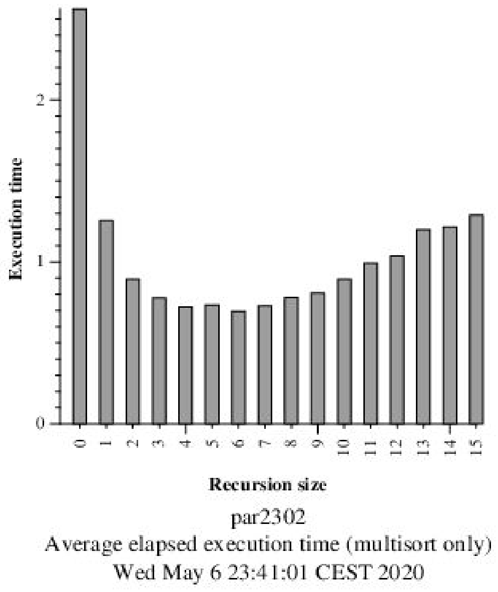


Figure 13. Paraver timeline for `multisort-omp-tree-cut-off.c` ***execution with*** `-c 1` ***and 8 processors (zoomed in where tasks are created)***







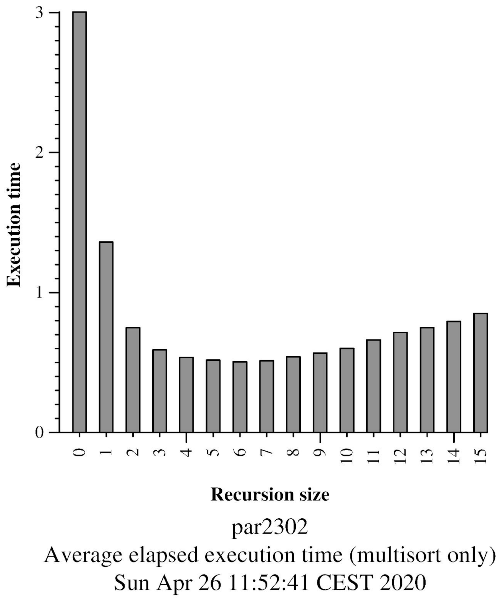
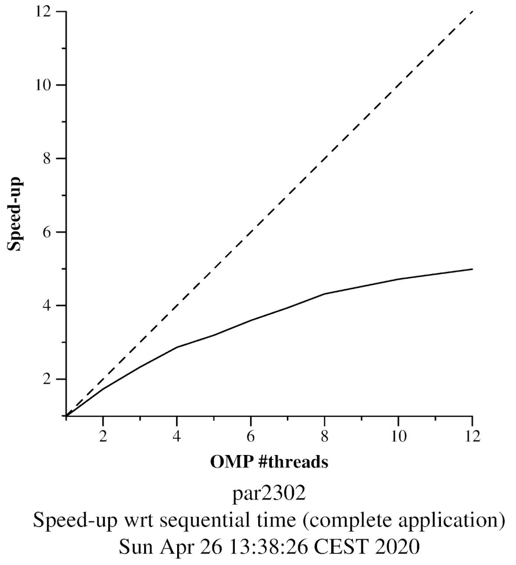


Figure 14. Execution time graphs for varying cut-off levels using 1 (top left), 8 (top right), 12 (bottom left) and 24 (bottom right) threads

The graphs in Figure 14 show the evolution of execution times when we vary the recursion size and using different number of threads. Looking closely at all graphs, we conclude that the best performance is obtained when using a cut-off level set to 5, and therefore that is the optimum value for `CUTOFF`.



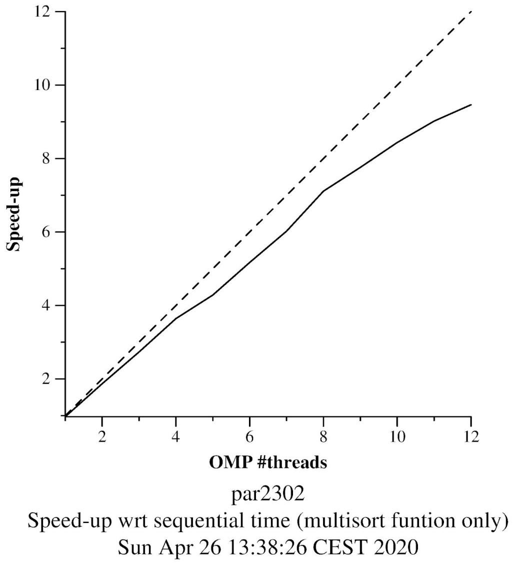


Figure 15. Strong scalability for tree decomposition with cut-off level set to 5

The speed-up plots in Figure 15, obtained executing `multisort-omp-tree-cut-off.c` with `-c 5`, show that even though there is some improvement over the plots for tree decomposition without the cut-off mechanism, the elevation in performance is limited and not very significant.

Optional 1:

As we saw before in Figure 14, the performance keeps increasing beyond using 12 threads. It appears quite clearly that the execution with 24 threads improves the performance of 12 threads. While it might seem inexplicable at first, as there are only 12 physical cores in boada-1 to 4, we have to remember that each physical core in these nodes has 2 threads and therefore they are able to execute programs using up to 24 threads.

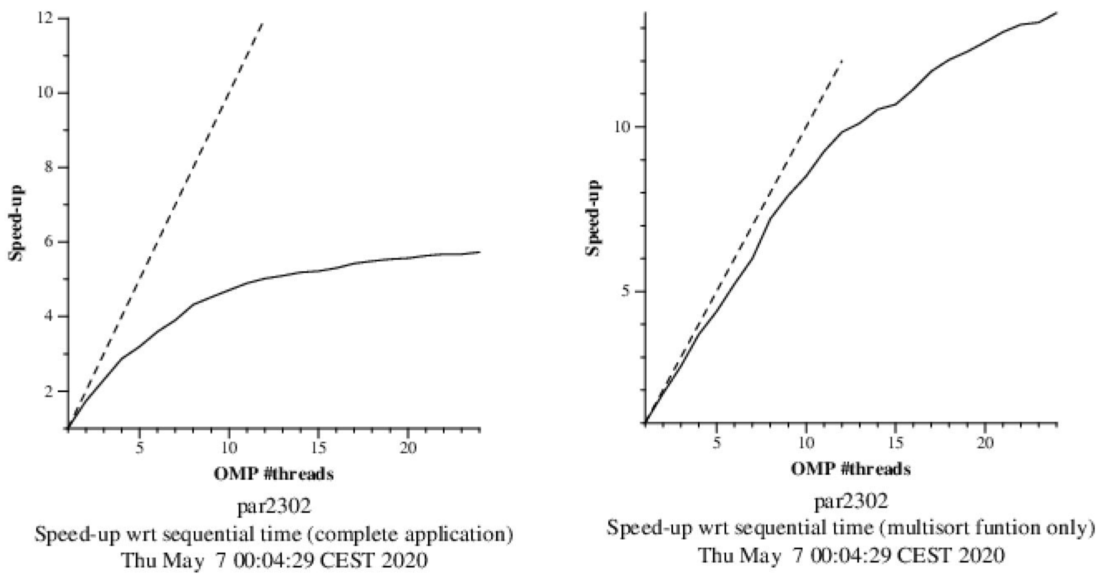


Figure 16. Strong scalability for tree decomposition with `CUTOFF = 5` ***using boada-3 (execution)***

The scalability of tree parallelization with a cut-off mechanism using up to 24 threads on boada-3, which corresponds to the architecture using the processor Intel Xeon E5645. This execution, as well as all the following ones for the rest of this section will be executed with the cut-off level set to 5, as it is the optimum threshold. Furthermore, it has been run, as will be run, with `-n 32768 -s 128 -m 128`.

We will also look at the scalability of the cut-off version of out code, Code 6, using boada-5 (cuda) and boada-6 to 8 (execution2). For this two different architectures, we will explore strong scalability by using up to 24 and 16 threads respectively, which are the maximum logical cores both are able to handle.

The plots below, in Figure 17, show us the strong scalability of `multisort-omp-tree-cut-off.c` when executed in the boada-5 architecture, which is based on an Intel Xeon E5-2620 v2 processor. We can observe that the performance is better than that shown in Figure 16 for an execution in boada-3. In fact, in the `multisort` function only plot we can even observe super linear speed-ups for some number of threads. Hence, we can note the difference in executing the same program in different architectures.

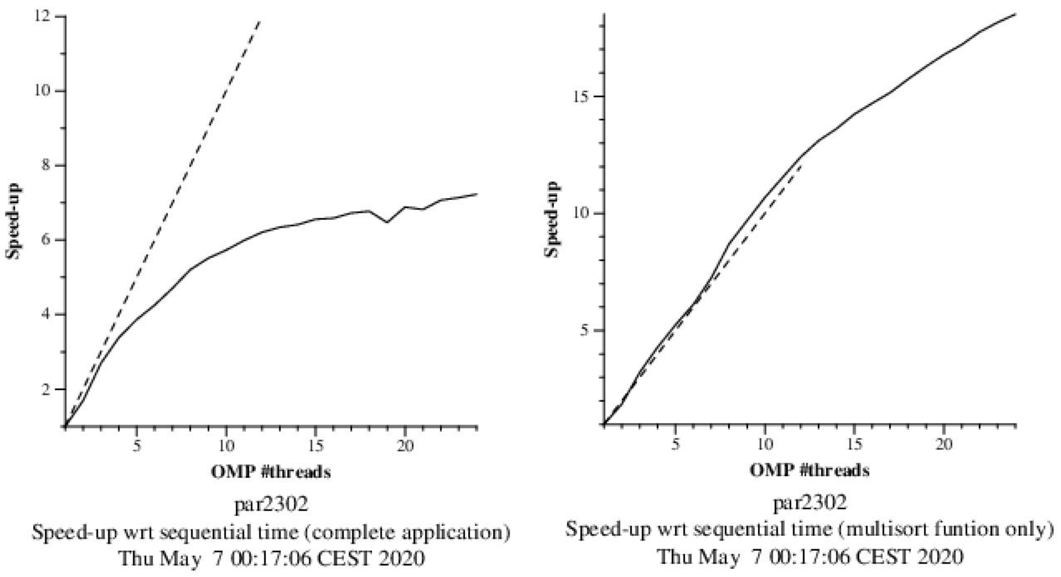


Figure 17. Strong scalability for tree decomposition with `CUTOFF = 5` ***using boada-5 (cuda)***

Lastly, in Figure 18 we can observe the execution in boada-6, which runs on an Intel Xeon E5-2609 v4 processor, as do boada-7 and 8. While we don't see super linear speed-ups in the `multisort` plot, we do observe that, unlike all the previous strong scalability analysis, the speed-up doesn't weaken as the number of threads increase and stays quite close to the 45º line.

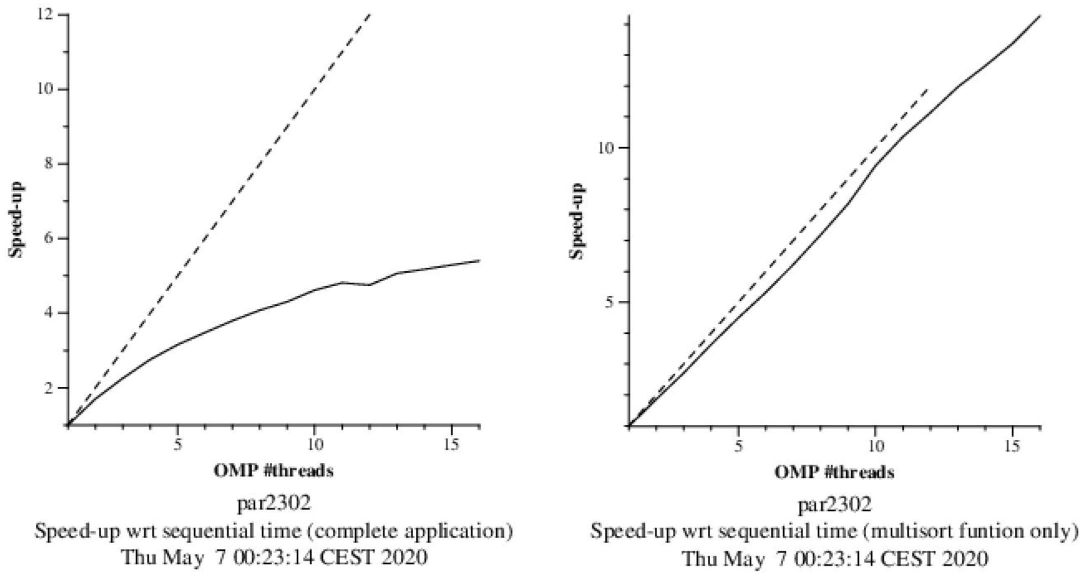


Figure 18. Strong scalability for tree decomposition with `CUTOFF = 5` ***using boada-6 (execution2)***

Task dependencies

In this section we will make use of explicit task dependencies to parallelize the `multisort` code with a tree strategy and cut-off. The implementation can be found below in Code 7.

In the merge function there is no dependencies between task, so it's not necessary to add any `depend` clause. But now we have to add a `taskwait` after the creation of both tasks. If we don't, we could run into some synchronization problems with the task created in the `multisort` function. In each task creation inside `multisort` function we have to add a `depend` clause with the data we are using (`in`) and the data we are generating (`out`). For example, the first call to multisort will apply recursively the `multisort` function to the first quarter of the input vector. So we add a `depend out` clause for this first section. The first merge call is merging the left side of the input vector, so it is using the data of the first two quarters of the vector. We mark it with the `depend in` clause. This merge will also generate the left side of the `tmp` vector, so we indicate it with the `depend out`.

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

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

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

           #pragma omp task final(depth >= CUTOFF) depend(in: tmp[0], tmp[n/2L])
           merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);

           #pragma omp taskwait
       }
```

```
        else {
            multisort(n/4L, &data[0], &tmp[0], depth+1);
            multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
            multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
            multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);

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

            merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);
        }
    } else {
        // Base case
        basicsort(n, data);
    }
}
```
Code 7. OpenMP instrumentation for tree parallelization with cut-off mechanism using task dependencies

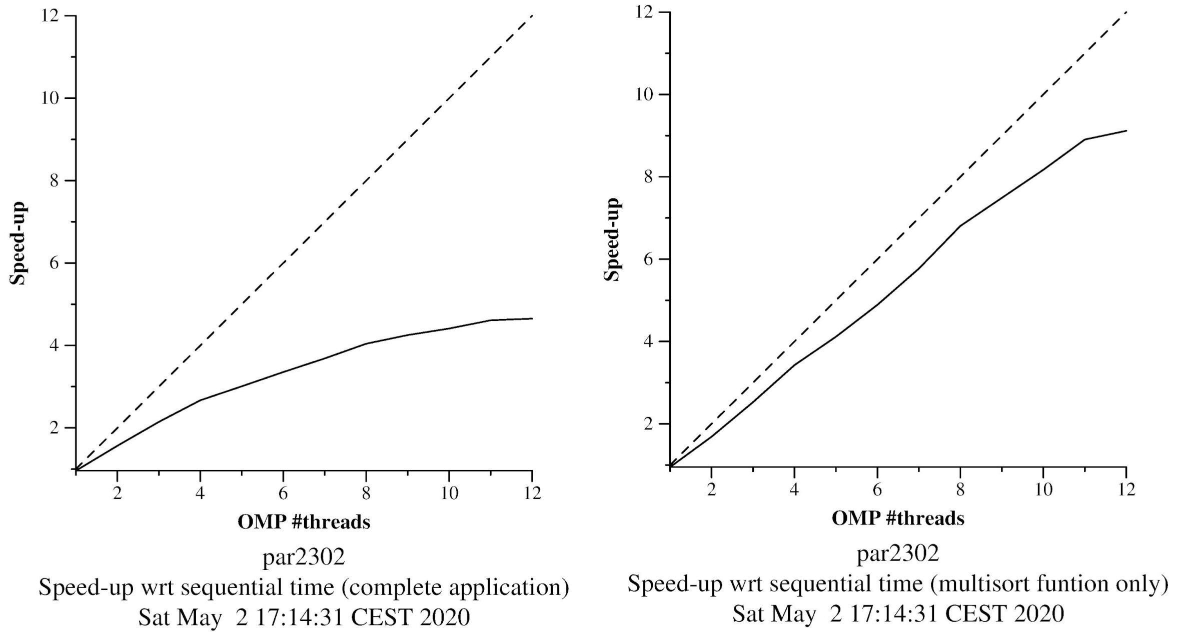


Figure 19. Strong scalability for tree parallelization with cut-off mechanism using task dependencies

The scalability is very similar to the Tree strategy without dependencies. The difference is that with task dependencies there is no need to wait for all the tasks inside the `taskgroup` to finish to be able to create and execute the next tasks. But in the case of mergesort this advantage doesn't produce any improvement because there are too many dependencies between the tasks, and the result is the similar.

Regarding to programmability, the code looks more complex than the previous versions and therefore less readable. We have to take in account all the dependencies and indicate them explicitly for each task, which is also more time consuming and messy.

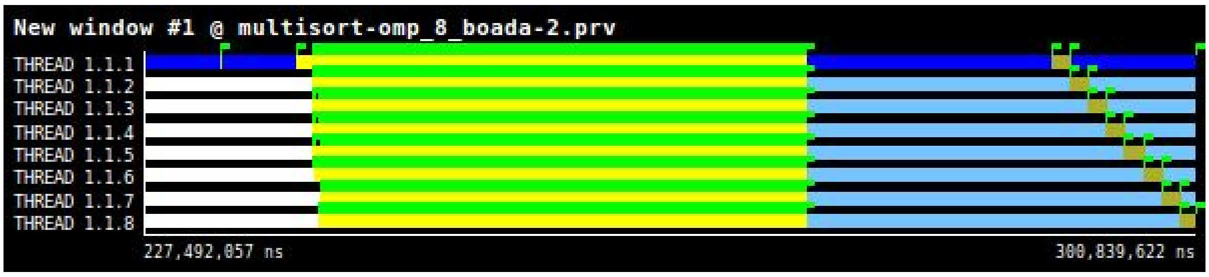


Figure 20. Task Dependency, 8 threads

Optional 2:

As we had seen in the strong scalability graphs from the previous versions, the complete application graph has very bad scalability compared with that of the `multisort` function only. The problem resides in the initialization of the vector `data` and clearing of vector `tmp`. We will try to parallelize the two function in charge of it to hopefully achieve a better overall performance of the program.

```
static void initialize(long length, T data[length]){
   long i;
   #pragma omp parallel
   {
       int nt = omp_get_num_threads();
       int who = omp_get_thread_num();
       data[who * length / nt] = rand();
       for (i = who * (length / nt) + 1; i < (who + 1) * (length / nt); i++) {
           data[i] = ((data[i-1]+1) * i * 104723L) % N;
       }
   }
}
```

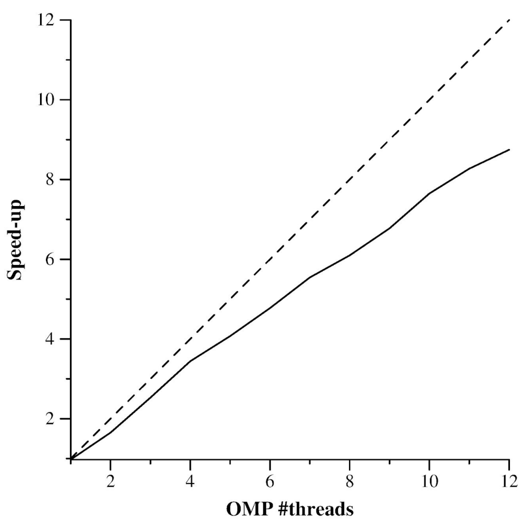
Code 8. OpenMP implementation of `initialize` ***function***

The `initialize` function, seen in Code 8, is in charge of initializing the vector `data` with random values. To parallelize the code we have to take into account the data dependency `data[i-1]` in each iteration. To do so, we will create a parallel region and manually distribute the work by chunks of `length/number_threads`**.** But before starting executing the for loop, we assign a random number to the first element of the chunk of each thread. Notice that if we create a random number for each element of the vector, the time of it will be huge.

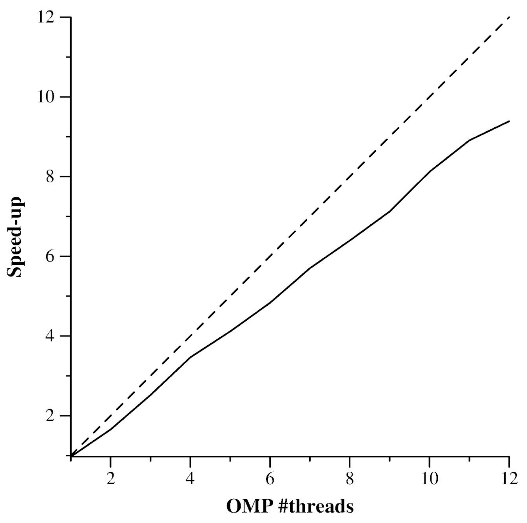
```
static void clear(long length, T data[length]) {
  long i;
  #pragma omp parallel for
  for (i = 0; i < length; i++) {
     data[i] = 0;
  }
}
```

Code 9. OpenMP implementation of `clear` ***function***

The `clear` function is in charge of clearing the vector `tmp` to 0. We parallelize it the same way with a `parallel for` construct.



par2302
Speed-up wrt sequential time (complete application)
Thu May 7 02:43:00 CEST 2020



par2302
Speed-up wrt sequential time (multisort funtion only)
Thu May 7 02:43:00 CEST 2020

Figure 21. Strong scalability for tree decomposition with data dependencies and parallel initialization of vectors

While in Figure 21 the scalability of the multisort function doesn't improve much, and it shouldn't as we haven't modified it, we do see a very clear increase in performance when looking at the speed-up plot for the overall execution of the program. In fact, for both plots Therefore, we conclude by parallelizing small functions that seem irrelevant, we can obtain significant gains in performance.

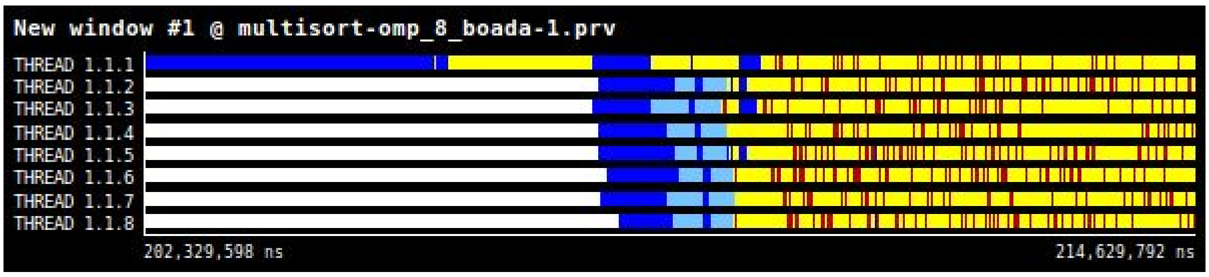


Figure 22. Paraver trace for tree decomposition with data dependencies and parallel initialization of vectors

The trace shown in Figure 22, allows us to observe how the initialization of the two vectors is now distributed between all threads before starting the execution of the `multisort` function. Those are the dark blue regions we see in all threads before the yellow region begins.

Performance evaluation

Now that we have a few OpenMP versions of the this multisort program, we will put them side by side and compare their execution times. We will execute them using the following parameters: problem size will be 32768, `MIN_SORT_SIZE` and `MIN_MERGE_SIZE` will be 1024 and the cut-off level 5 for the versions that use it. We will execute the tasks using 12 threads. Since we are not given the total execution time, we will consider it to be the addition of the initialization time, the multisort execution time and the time taken to check the sorted data.

Version	Execution time 1 / s	Execution time 2 / s	Execution time 3 / s	Average time / s
`multisort-omp-leaf.c`	2.369699	2.408717	2.378183	2.385533
`multisort-omp-tree.c`	1.599303	1.504748	1.504717	1.536256
`multisort-omp-tree-cut-off.c`	1.454018	1.441289	1.440481	1.445263
`multisort-omp-tree-cut-off-task-dependencies.c`	1.449508	1.467070	1.447357	1.454645
`multisort-omp-optional-2.c`	0.749425	0.744902	0.730171	0.741499

Table 3. Comparison of execution times for the different `multisort` ***versions***

Table 3 serves as a comparison between the different versions of this program we have created. By using a basic tree decomposition we already see a drastic improvement over the leaf strategy we first implemented, with the average execution time coming down to 1.536256s from 2.385533s. Furthermore, introducing the cut-off mechanism, when used with the optimum value (5 in this case), also provides a slight growth in performance lowering the average time from 1.536256s for `multisort-omp-tree.c` to 1.445263s for `multisort-omp-tree-cut-off.c`. The addition of explicit tasks dependencies hasn't resulted in an enhancement. However, our last version, `multisort-omp-optional-2.c`, which introduces the OpenMP instrumentation of the initialization of data and tmp, lowered the average execution time to 0.741499s. Therefore, it has proven to be the best performing out of all versions we have implemented. It doesn't come as a surprise as we knew this was the version that most exploited parallelism.

Conclusions

In this assignment we started understanding the behaviour of the mergesort algorithm. With *Tareador* we saw two ways to parallelize the algorithm; the leaf strategy and tree strategy. We also detected the dependencies that exist between the tasks generated in each of the strategies.

The next step was to implement both strategies with OpenMP, taking into account the dependencies. We started using the `taskwait` and `taskgroup` for the synchronization between task. Unsurprisingly we saw how the tree strategy had a much higher scalability than the leaf strategy. The creation of tasks in a sequential way in the leaf strategy meant a much slower traversal of the recursion tree. However, with the tree strategy the creation of tasks is parallelized, avoiding so much overhead and reaching before the execution of the base case.

We then implemented a cutting mechanism for the tree strategy. After several tests we concluded that the optimum cut-off value is 5. With this value we have obtained a slight improvement compared to the version without cut-off. That is because we found the sweet spot in the trade-off between traversing the tree in parallel and the overheads of task creation.

Using this last version, we explored the effect of running the program on different architectures. We observed how different processors allowed for different behaviours of the speed-up.

Next, we repeated the tree strategy, but this time using explicit dependencies instead of using `taskwait` and `taskgroup`. This implementation has had the same performance as the previous versions. However, we have found this version more complicated to implement as we need to mark explicitly all dependencies for each task.

For our final version, we have expanded on the previous one and implemented the parallelization of the initialize and clear functions, which has allowed for an improvement in performance over all the other versions, making this one the best performing one.