PAR Laboratory Assignment

Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Group 13-03

Carlos Bárcenas Holguera

Miquel Gómez i Esteve

Facultat d'Informàtica de Barcelona - UPC

March 2018

Spring 2017-2018

Index

Group 13-03: Carlos Barcenas Holguera
Miquel Gómez i Esteve

Part I: ”Divide and conquer”

We will try to analyse the program with tareador. The first thing is to do is to add the the tareador API’s calls to the code as shown in figure 1.

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

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

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

		tareador_start_task("multisort");
		multisort(n/4L, &data[n/4L], &tmp[n/4L]);
		tareador_end_task("multisort");

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

		tareador_start_task("multisort");
		multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
		tareador_end_task("multisort");

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

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

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

Figure 1: multisort-tareador.c with the tareador’s calls added.

Once we have modified the code we can execute the program and we can obtain the dependence graph from figure 2.

Group 13-03: Carlos Barcenas Holguera
Miquel Gómez i Esteve

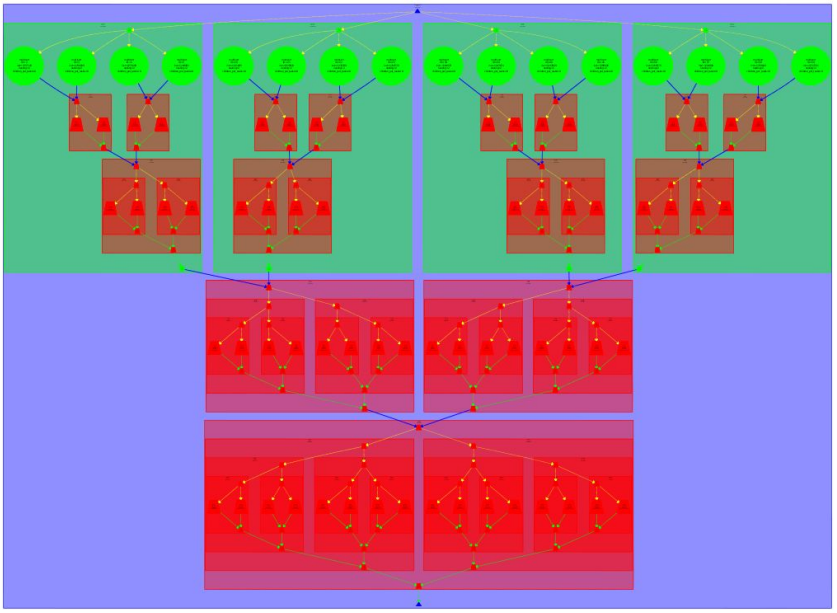


Figure 2:
Task dependence graph

We can see that in every case the vector is divided in four segments, until the segments obtained are smaller than the preset minimum size, which is MIN_SORT_SIZE *4L. Once the segments are smaller, they are ordered on the base case, with the basicsort function.
After that we enter the merge part of the code where we have a minimum size for us to execute the merge function, which is MIN_MERGE_SIZE * 2L. If the segments are smaller we merge them with the basicmerge function.
All the dependencies shown in the graph are caused by the fact that we need the segments to be ordered before we merge them.
We will now use the Tareador's predictions to see how our program would behave. Figure 3 are the results obtained with different number of threads.

# threads	Execution Time	SpeedUp
1	20.334.421	-
2	10.173.740	1,998
4	5.087.595	3,997
8	2.547.419	7,983
16	1.289.941	15,775
32	1.289.920	15,779
64	1.289.909	15,779

Figure 3: Times obtained with Tareador predictions

From the results obtained, we can prove that once we have 16 threads, we have reached the maximum parallelism of the program. This means that we cannot have a higher speedup and any other threads we want to add will not be used in the execution.

Group 13-03: Carlos Barcenas Holguera
Miquel Gómez i Esteve

Part II: Shared-memory parallelization with OpenMP tasks

We will now analyse the program parallelization and performance with tasks. First of al we must modify the code to implement the two versions: leaf and tree. The figures 4 and 5 show the modified codes for the leaf and tree versions respectively.

```
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);
		}
}
```

Figure 4: Leaf Strategy Code

```
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);
		}
}
```

Figure 5: Tree Strategy Code

In the leaf strategy, all we add is a openmp task construct every time we use the functions basicsort and basicmerge. However, in the tree strategy we need to add the task construct every time we use the function multisort or merge, and for the program to work properly, some taskwait constructs are needed as well.

We will now see how the leaf version behaves when checking its strong scalability.

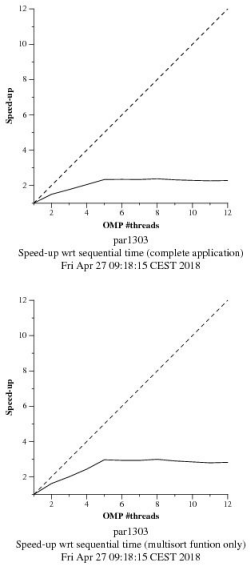


Figure 6:
Plots for Leaf Strategy

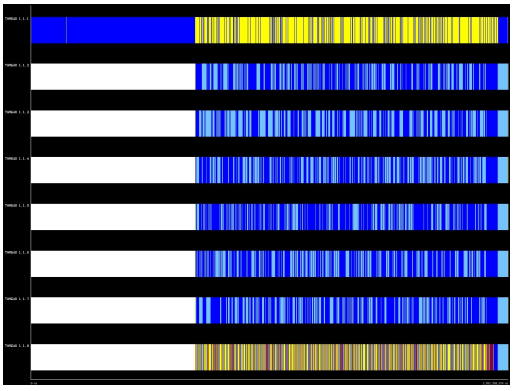


Figure 7:
Leaf Strategy Trace
with 8 threads

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O
THREAD 1.1.1	1,900,632,297 ns	-	-	1,051,065,678 ns	657,969 ns
THREAD 1.1.2	734,808,130 ns	1,015,305,440 ns	-	918 ns	768,394 ns
THREAD 1.1.3	652,533,255 ns	1,015,306,012 ns	-	985 ns	727,494 ns
THREAD 1.1.4	730,750,578 ns	1,015,304,287 ns	-	923 ns	726,560 ns
THREAD 1.1.5	853,195,884 ns	1,015,308,445 ns	-	870 ns	757,465 ns
THREAD 1.1.6	719,048,426 ns	1,015,210,682 ns	-	5,195 ns	72,268 ns
THREAD 1.1.7	774,297,852 ns	1,015,322,475 ns	-	793 ns	53,651 ns
THREAD 1.1.8	1,660,986,151 ns	1,015,210,674 ns	200,724,093 ns	9,036,168 ns	875,662 ns
Total	8,026,252,573 ns	7,106,968,015 ns	200,724,093 ns	1,060,111,530 ns	4,639,463 ns
Average	1,003,281,571.62 ns	1,015,281,145 ns	200,724,093 ns	132,513,941.25 ns	579,932.88 ns
Maximum	1,900,632,297 ns	1,015,322,475 ns	200,724,093 ns	1,051,065,678 ns	875,662 ns
Minimum	652,533,255 ns	1,015,210,674 ns	200,724,093 ns	793 ns	53,651 ns

Figure 8:
Leaf Strategy Statistics
with 8 threads

As we can see in figures 6, 7 and 8, the leaf strategy does not behave very well so its scalability is not strong at all. The reason must be that we must wait until the recursive part finishes so that we can create the different tasks and distribute them between the rest of the threads. At the end we also have a lot of synchronizations that will affect negatively on the execution time.

Group 13-03: Carlos Barcenas Holguera
Miquel Gómez i Esteve

It’s time to now check the behavior of the tree version.

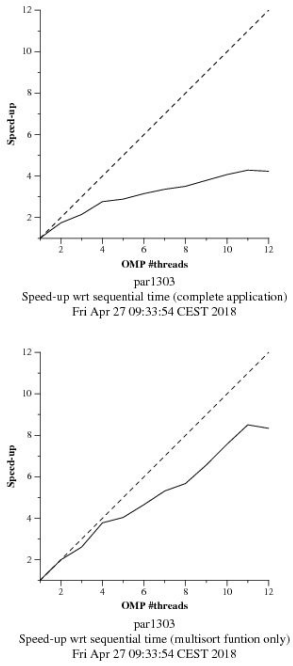


Figure 9:
Plots for Tree Strategy

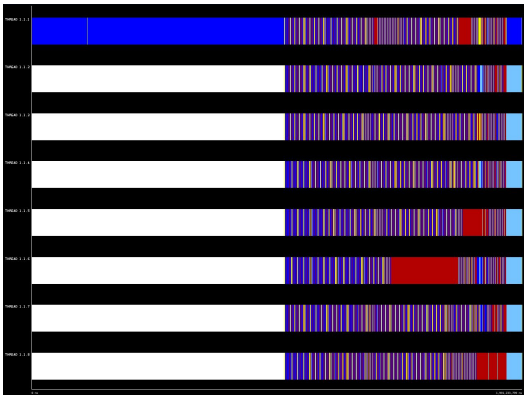


Figure 10:
Tree Strategy Trace
with 8 threads

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O
THREAD 1.1.1	1,890,107,176 ns	-	89,979,365 ns	10,789,387 ns	355,488 ns
THREAD 1.1.2	857,461,434 ns	1,026,059,469 ns	25,182,480 ns	1,040,825 ns	279,916 ns
THREAD 1.1.3	880,420,941 ns	1,026,058,727 ns	15,439,666 ns	922,426 ns	310,645 ns
THREAD 1.1.4	862,892,409 ns	1,026,059,409 ns	19,277,139 ns	891,938 ns	277,739 ns
THREAD 1.1.5	794,327,567 ns	1,026,054,429 ns	103,044,033 ns	751,521 ns	258,748 ns
THREAD 1.1.6	591,043,723 ns	1,025,964,133 ns	291,624,947 ns	712,513 ns	187,508 ns
THREAD 1.1.7	858,660,048 ns	1,026,069,950 ns	32,660,945 ns	782,323 ns	283,092 ns
THREAD 1.1.8	782,780,246 ns	1,025,963,355 ns	115,967,577 ns	614,486 ns	279,297 ns
Total	7,517,693,544 ns	7,182,229,472 ns	693,176,152 ns	16,505,419 ns	2,232,433 ns
Average	939,711,693 ns	1,026,032,781.71 ns	86,647,019 ns	2,063,177.38 ns	279,054.12 ns
Maximum	1,890,107,176 ns	1,026,069,950 ns	291,624,947 ns	10,789,387 ns	355,488 ns
Minimum	591,043,723 ns	1,025,963,355 ns	15,439,666 ns	614,486 ns	187,508 ns

Figure 11:
Tree Strategy Statistics
with 8 threads

The behaviour and scalability of the tree version is a much better as we can see in figures 9, 10 and 11. This is due to the fact that we create tasks during the recursive part of the program, so that its execution is much faster, while in the leaf version tasks where only created in the base cases. We do not have that many synchronizations because of the taskwait construct we added in several places, and we can assure that the data is already ordered before getting to the merge function for example.
Comparing the speed-up plot for the multisort execution with the ideal speed-up values obtained with tareador, we can assure that this is a very good parallelization since they are very similar in most of the values for #threads, and also because the tareador values are ideal and do not check any of the overheads.
However there is an exception on the last execution with 12 threads.

Group 13-03: Carlos Barcenas Holguera
Miquel Gómez i Esteve

Optional:

When submitting for execution with 8 threads we see that the initialization part without any kind of parallelization of the program takes 0.797846 seconds and the trace resulting would be the same shown in figure 10 where is clearly represented the initialization part, which uses only one thread.

Next, we parallelize the functions used to initialize the vectors used in the program as seen the the next figure (12).

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task
  initialize(N, data);
  #pragma omp task
  clear(N, tmp);
  #pragma omp taskwait
}
```

Figure 12: Modified code to parallelize program initialization

When submitting for execution with 8 threads we see a very little improvement since the initialization time is 0.753048 seconds. The total speed-up is 0.753048/0.797846 = 1,06.
The reason behind this poor speed-up would be the disbalance between the two tasks, as we can see in figure 14, where the thread 1 executes the function initialize while the thread 4 executes the clear function, which takes very little time compared to the initialize function.

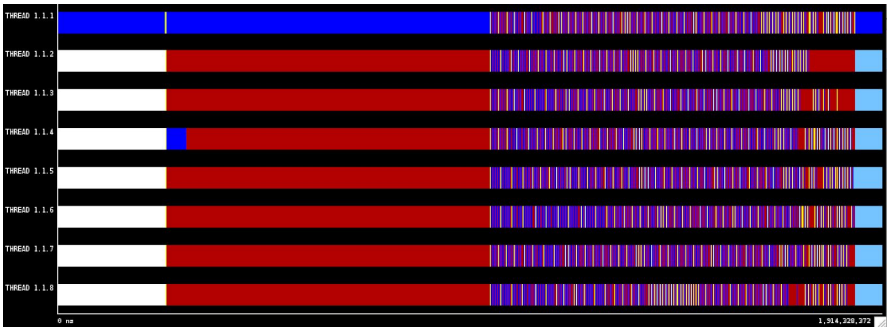


Figure 13:
Tree Strategy Trace and parallelized initialization with 8 threads.

After analyzing the code we can see that the function initialize has a clear dependency (to calculate data[i] we need the value of data[i-1]) which forces the execution to be sequential.

Because of that we think it wouldn't make sense to try to parallelize the two functions more efficiently (with a parallel for) since the improvement margin is very small.

Group 13-03: Carlos Barcenas Holguera
Miquel Gómez i Esteve

Part III: Using OpenMP task dependencies

First of all we need to change the code adding all the in, out and inout dependencies of the different tasks of the code. The changed code is shown in figure 15

```
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 depend(inout: data[0])
        multisort(n/4L, &data[0], &tmp[0]);

        #pragma omp task depend(inout: data[n/4L])
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);

        #pragma omp task depend(inout: data[n/2L])
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);

        #pragma omp task depend(inout: 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]) 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);
  }
}
```

Figure 14: Modified code to control dependencies between tasks

Then we submit the program for execution with 8 threads and the execution time obtained for the multisort is 0.996871 seconds.

It's now time to check the strong scalability of our program, so we submit for execution the code with the submit-strong script. The results obtained as plots are shown in figure 16, and we can see that it is very similar to the last version analyzed (tree version with taskwait to ensure dependencies are satisfied)

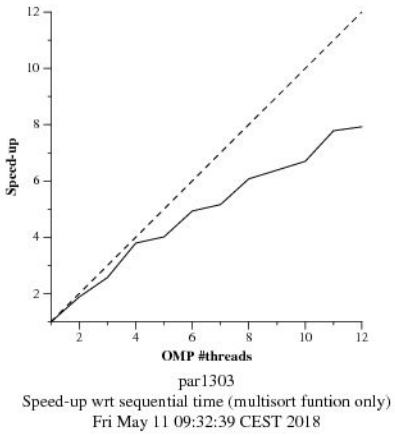


Figure 15: Speed Up plot for the task dependencies version

The speed-up obtained in the different cases is pretty close to the ideal one given by the tareador. We can see that it has a very similar scalability than the one shown in the tree task version, mostly because even though the dependencies declarations are different, the dependencies graph would be nearly the same except for some minor differences (We now know that all the dependencies generated are necessary).

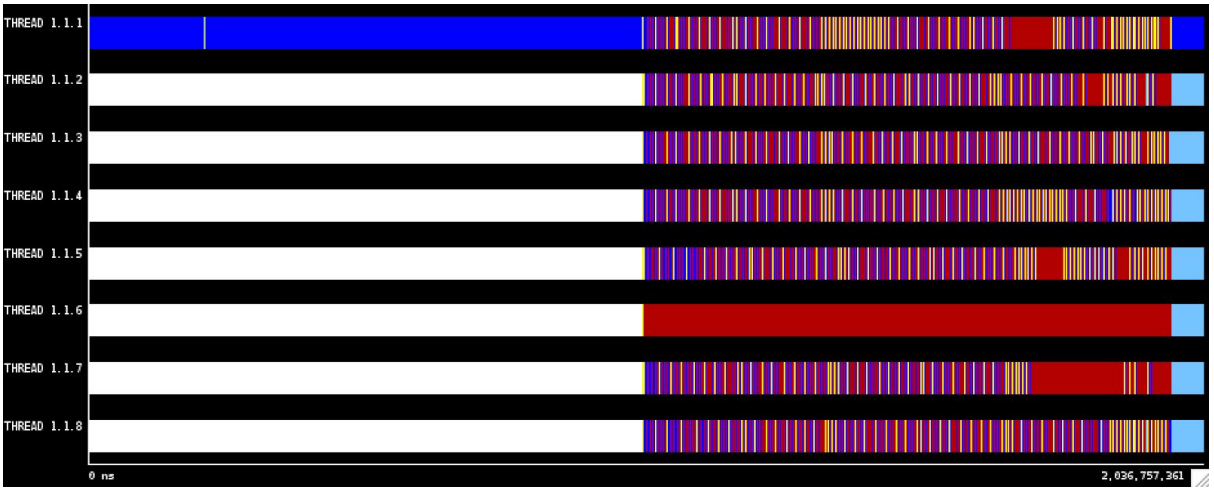


Figure 16: Trace for task dependencies version

Group 13-03: Carlos Barcenas Holguera
Miquel Gómez i Esteve

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	94.97 %	-	4.75 %	0.27 %	0.01 %	0.00 %
THREAD 1.1.2	44.36 %	51.33 %	4.25 %	0.06 %	0.01 %	-
THREAD 1.1.3	47.70 %	51.32 %	0.92 %	0.05 %	0.01 %	-
THREAD 1.1.4	48.63 %	51.25 %	0.05 %	0.06 %	0.01 %	-
THREAD 1.1.5	44.91 %	51.27 %	3.75 %	0.07 %	0.01 %	-
THREAD 1.1.6	0.00 %	51.21 %	48.78 %	0.01 %	0.00 %	-
THREAD 1.1.7	37.12 %	51.21 %	11.60 %	0.06 %	0.01 %	-
THREAD 1.1.8	48.21 %	51.32 %	0.40 %	0.06 %	0.01 %	-
Total	365.89 %	358.91 %	74.49 %	0.63 %	0.08 %	0.00 %
Average	45.74 %	51.27 %	9.31 %	0.08 %	0.01 %	0.00 %
Maximum	94.97 %	51.33 %	48.78 %	0.27 %	0.01 %	0.00 %
Minimum	0.00 %	51.21 %	0.05 %	0.01 %	0.00 %	0.00 %
StDev	24.01 %	0.05 %	15.31 %	0.07 %	0.00 %	0 %
Avg/Max	0.48	1.00	0.19	0.29	0.75	1

Figure 17: Histogram for task dependencies version

In figure 17 we can see that in some threads a lot of time is lost in some threads in synchronization state. We think this could be due to the tasks queue access.

We make the problem size 10 times bigger and check how it behaves. It is clear that we have the same issue here as represented in figure 18.

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	97.96 %	-	1.98 %	0.05 %	0.01 %	0.00 %
THREAD 1.1.2	53.46 %	44.19 %	2.30 %	0.03 %	0.01 %	-
THREAD 1.1.3	54.40 %	44.19 %	1.35 %	0.04 %	0.01 %	-
THREAD 1.1.4	54.53 %	44.19 %	1.23 %	0.04 %	0.01 %	-
THREAD 1.1.5	52.96 %	44.20 %	2.80 %	0.03 %	0.01 %	-
THREAD 1.1.6	43.23 %	44.19 %	12.54 %	0.03 %	0.01 %	-
THREAD 1.1.7	4.09 %	44.19 %	51.71 %	0.00 %	0.01 %	-
THREAD 1.1.8	55.24 %	44.19 %	0.52 %	0.04 %	0.01 %	-
Total	415.87 %	309.36 %	74.44 %	0.25 %	0.08 %	0.00 %
Average	51.98 %	44.19 %	9.30 %	0.03 %	0.01 %	0.00 %
Maximum	97.96 %	44.20 %	51.71 %	0.05 %	0.01 %	0.00 %
Minimum	4.09 %	44.19 %	0.52 %	0.00 %	0.01 %	0.00 %
StDev	23.75 %	0.00 %	16.43 %	0.01 %	0.00 %	0 %
Avg/Max	0.53	1.00	0.18	0.70	0.78	1

Figure 18: Histogram for task dependencies version (10 times bigger problem size)

Now, we try to make the problem 100 times the size of the original, however when submitting, it seems like the program doesn't end and we can't obtain the corresponding trace and histogram. So we can't see whether or not it behaves the same way as with the original size or the 10 times the original.

Optional:

Now we want to find the optimal values for sort_size and merge_size. First, we execute the script to find the optimal value for sort_size:

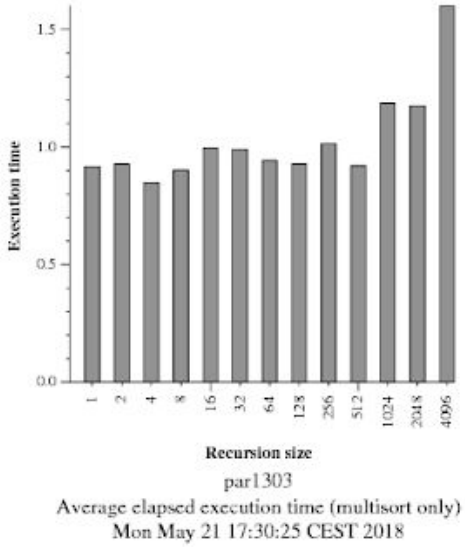


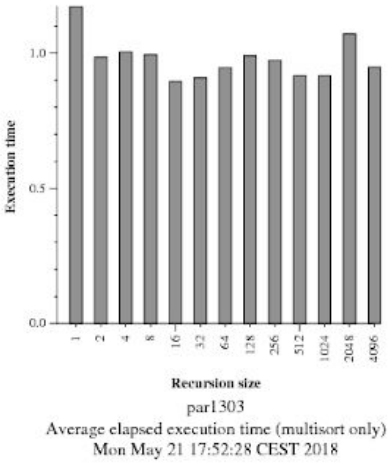
Figure 19: Plot of the execution times with different values of sort_size.

In the plot we can appreciate that seems there's no optimal number for sort_size. Almost all the execution times are the same, the difference is minimum. So maybe we can appoint that the sort_size doesn't affect in great measure the execution time, at least for this code.

To find the possible best or optimal merge_size we are going to try with sort_size = 4 and sort_size = 512. So we can appreciate the differences between a big number and a little one.

Now we want to find the optimal values for merge_size:

Figure 20:
Plot of the execution times with sort_size = 4
for different values of merge-size



Group 13-03: Carlos Barcenas Holguera
Miquel Gómez i Esteve

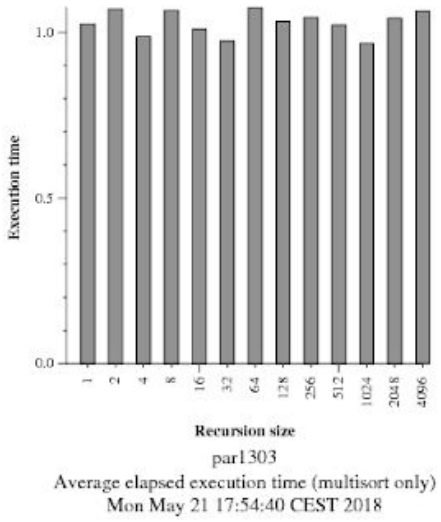


Figure 21:
Plot of the execution times with sort-size = 512 for different values of merge-size

With the results obtained we can see that there is not a remarkable difference that would allow us to assure that one of the options would be better that the rest of them. The execution time is very similar in all the executions. We would need a much more intense test of the program behaviour with the different values of merge-size and sort-size to see which are the most efficient combinations.
In this particular case, when comparing the execution times we can see that the best option seems to be sort-size = 4 and merge-size = 16, but, as said the difference is not remarkable enough.