PAR Laboratory Assignment

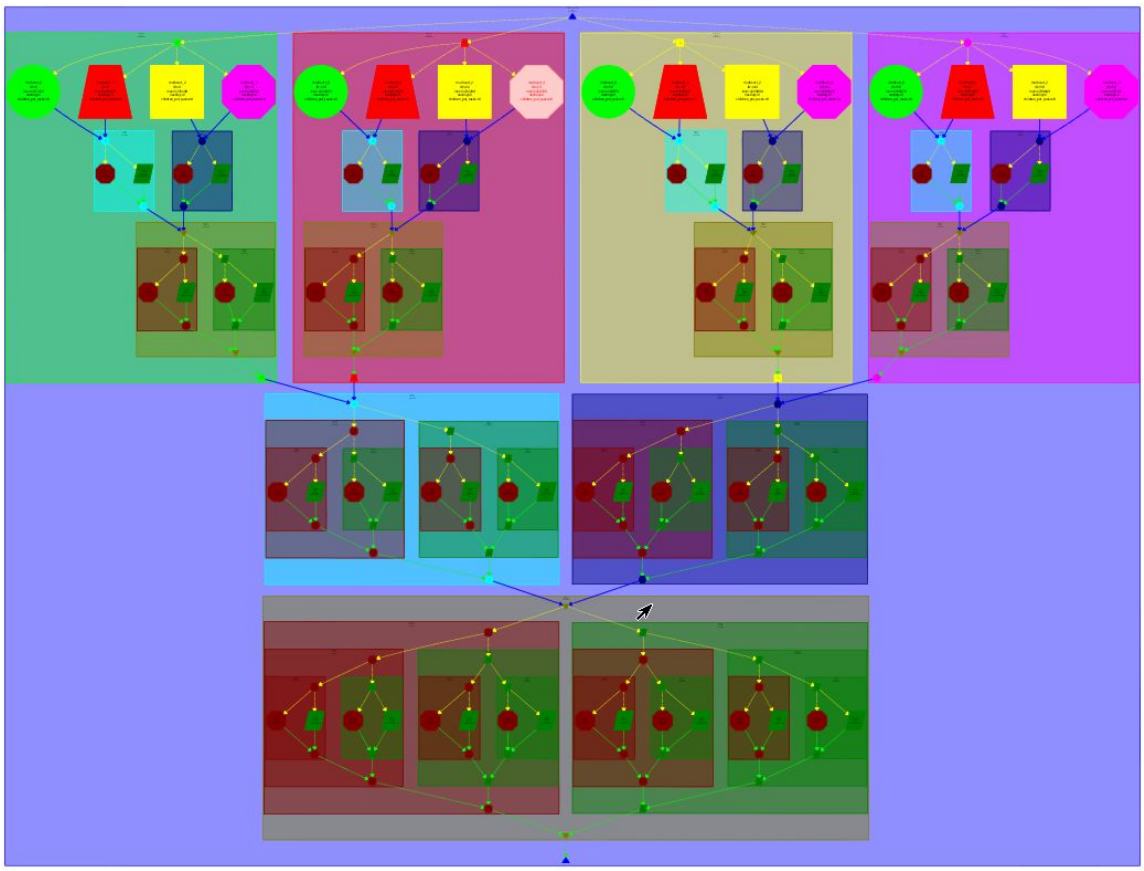
Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

PAR 2312
Fèlix Arribas Pardo
Ana Mestre Borges

1. Introduction

In this laboratory session we are going to practice and observe different parallelization strategies on a recursive algorithm: Multisort.

This algorithm can be splitted in a lot of little tasks with smaller problem size and then merged into one.



Here we can observe the Paraver scheme. However, we may have some overhead forking, and joining all different tasks.

Processors	Execution time (ns)	Speed-up
1	20.334.421.001	--
2	10.173.726.001	1,999
4	5.086.725.001	2,000
8	2.550.595.001	1,994
16	1.289.922.001	1,977
32	1.289.909.001	1,000
64	1.289.909.001	1,000

We can see that, from 1 to 16 processors, when we double the amount of processors we obtained a nice Speed-up, around 2.
Using more than 16 processors we do not have Speed-up. That is because there is no much work to do for the rest of processors, they are useless.

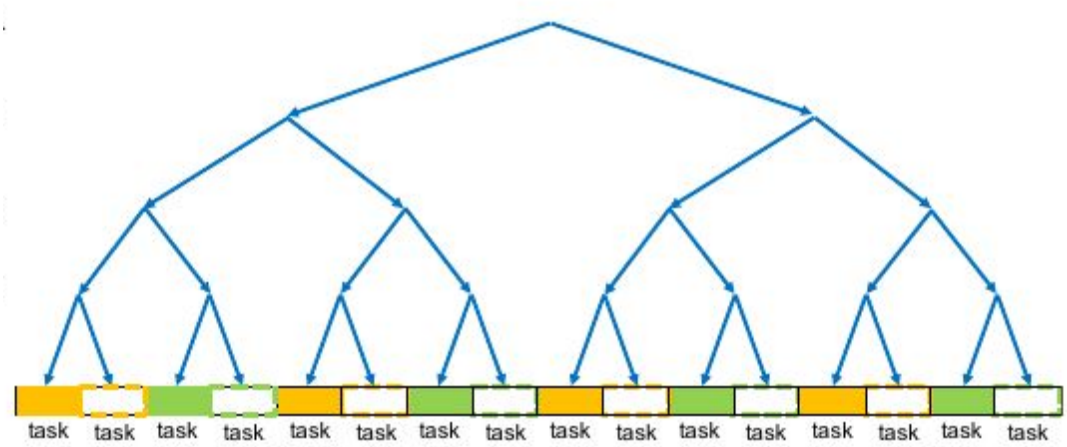
2. Parallelization strategies

Divide-and-conquer

A divide-and-conquer algorithm divides a problem into two or more subproblems of the same type, when each subproblem is solved, the solutions are combined to to give a solution to the original problem.

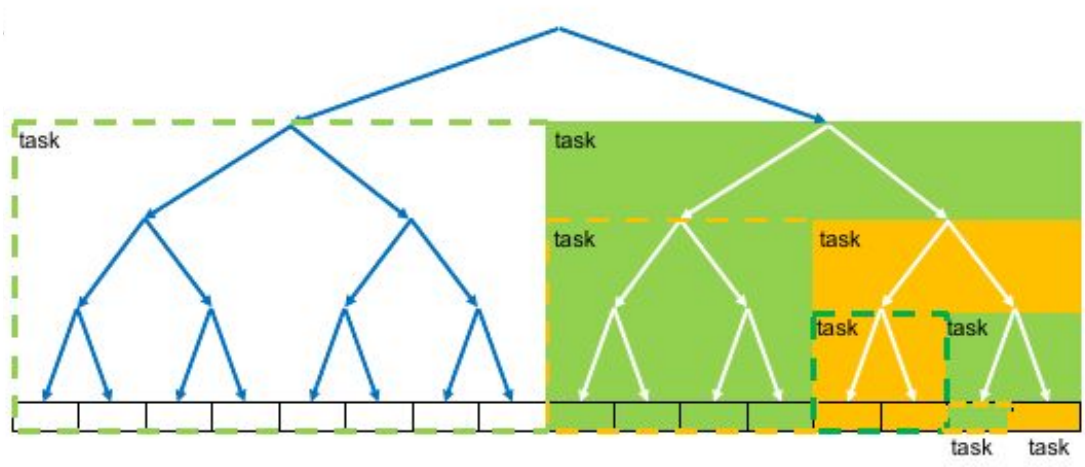
1. Leaf strategy (*Sequential generation of tasks)*

   A task corresponds with each invocation of the function once the recursive divide-and-conquer decomposition stops.



2. Tree strategy *(Parallel generation of tasks)*

   A task corresponds with each invocation of the function during the recursive decomposition.



We have different ways to write a OpenMP code for tasks to sync and merge properly.

When tasks are divided and then we need to merge the generated data like in multisort, we cannot start merging before all tasks finish. In this session we are waiting for them and creating dependencies between tasks: **`#pragma omp taskwait`** and **`#pragma omp task depend(...)`**.

1. Taskwait

```
// 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);
}

merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
```

2. Dependencies

```
// 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
```

3. Performance evaluation

Leaf strategy

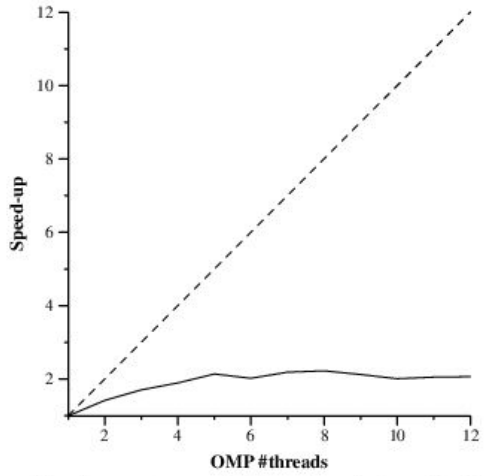
When using Leaf strategy we create task in the base cases:

- **Multisort**: Just before the *basicsort*. All sorting tasks are waited in the *taskwait* before the first merge.
- **Merge**: The function *basicmerge* is parallelized. Tasks are waited after the merge.

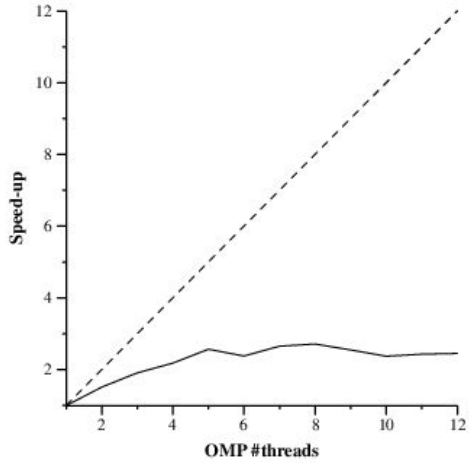
Leaf size depends on `4 * MIN_SORT_SIZE` for multisort and `2 * MIN_MERGE_SIZE` for merge.

```
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);
	}
}
```

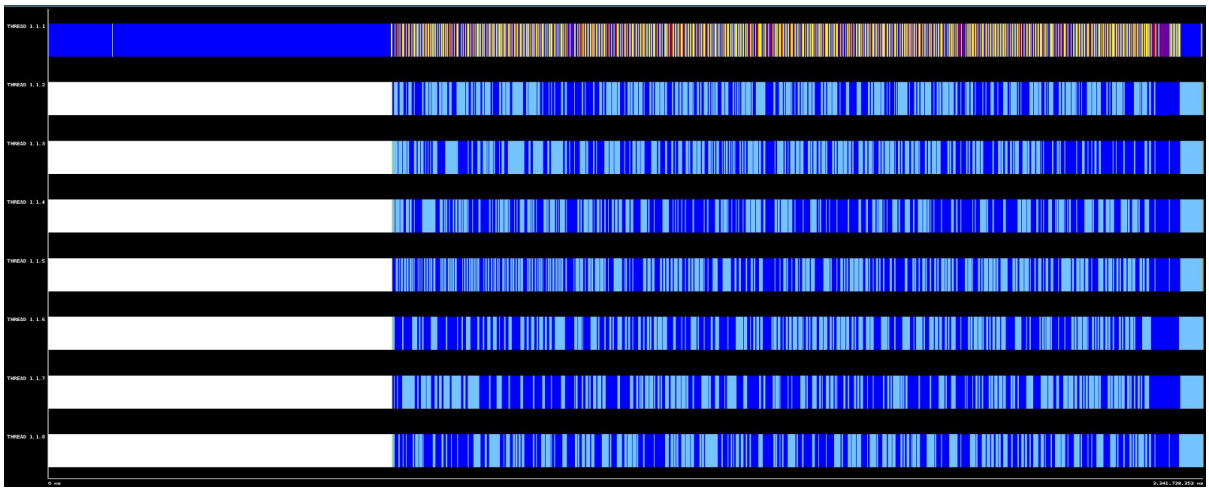
The results are not what we expected. **The Speed-up is not good enough**. With more than five processors the Speed-up stops increasing and it looks linear.



Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)



Thanks to Paraver we see that the processors are mostly waiting for a task to execute (soft blue) and not computing (dark blue).

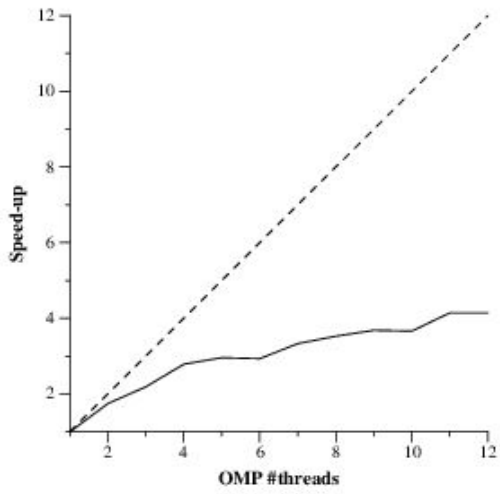
Tree strategy

When using the tree strategy for parallel generation of tasks, a task corresponds with the invocation of the function during the recursive decomposition. In this case we have used a **`#pragma omp taskgroup`** and **`#pragma omp task`** for both multisort and merge invocations in the multisort function and just **`#pragma omp task`** inside the implementation for the merge function.

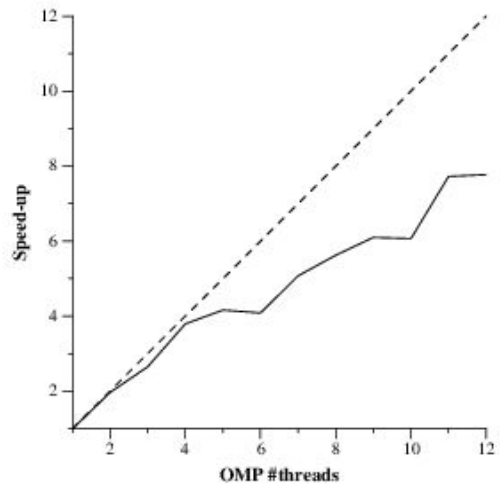
Instead of using **`#pragma omp taskwait`** we used **`#pragma omp group`**

```
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
```

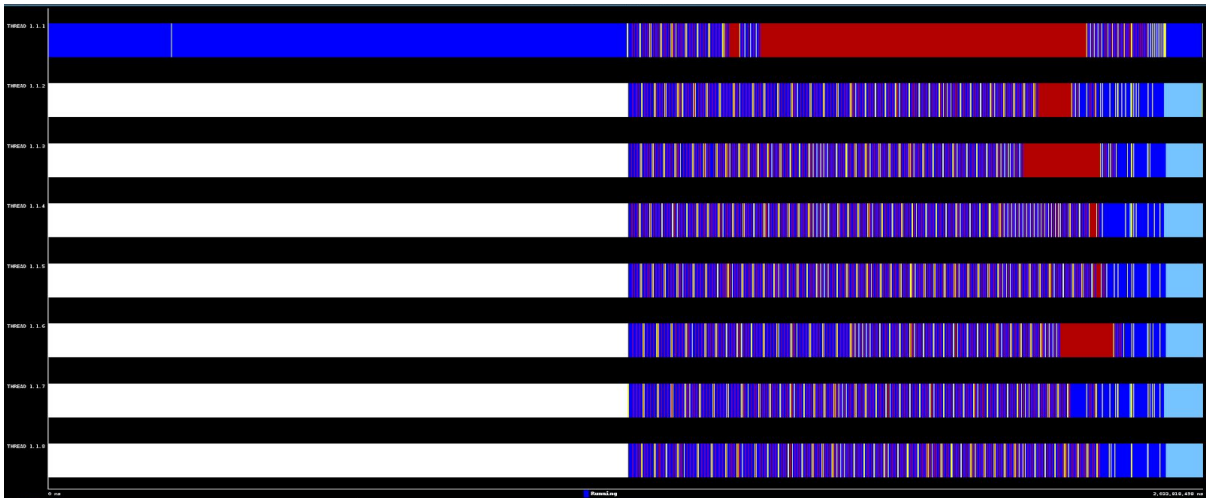
```
                #pragma omp task
                merge(n, left, right, result, start, length/2);
                #pragma omp task
                merge(n, left, right, result, start + length/2, length/2);
        }
}

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



Speed-up wrt sequential time (complete application)

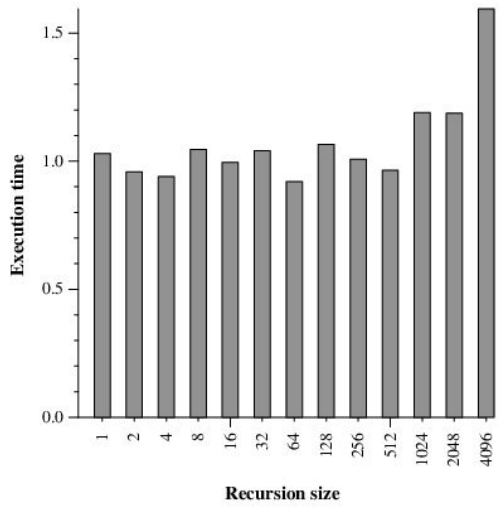


Speed-up wrt sequential time (multisort funtion only)



As we can see on the speed-up plots above, it is easy to say that the tree version of the program is more efficient and consequently, better than the leaf version. In any case, it is important to consider the overhead and synchronization time.

Influence of the recursivity depth in the Tree version

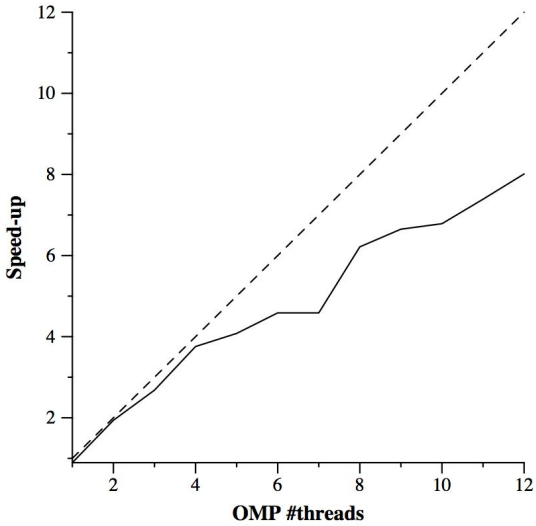


Average elapsed execution time (multisort only)

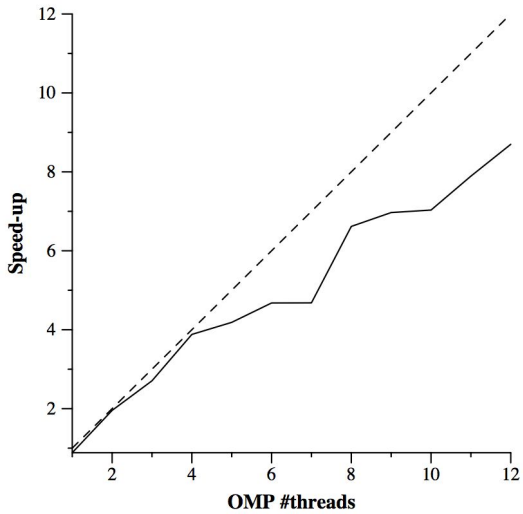
In order to look whether changing the recursivity depth in the tree version improves its behaviour or not, we have studied this plot. By looking at it, it is clear that there is no general changes until the recursion size is 4096. Therefore, the recursion size doesn't have a big impact on the execution time.

Full parallelization

Now we have found the best way to parallelize the multisort but the program has more code, like the initialization function and the clean function. Fortunately, those functions are a simple *for* loop so we can parallelize them with `#pragma omp parallel for`.

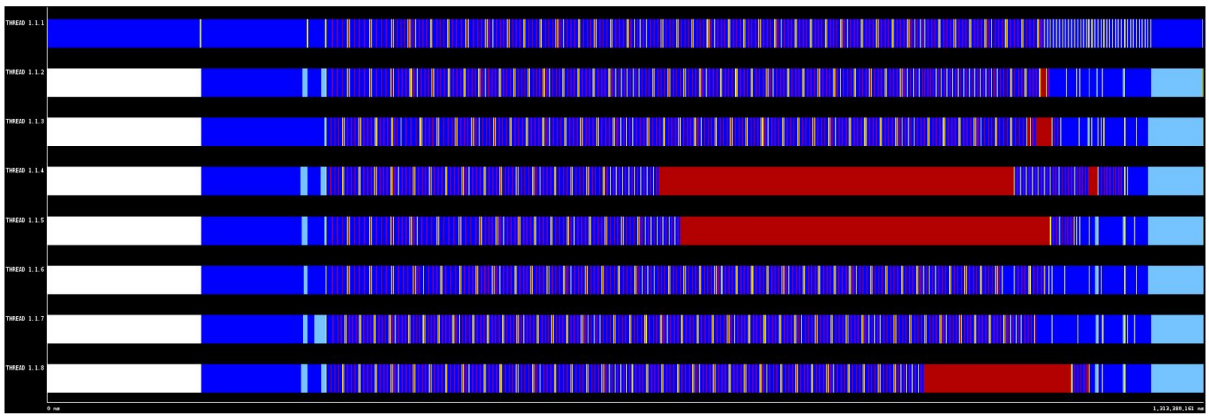


Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)

Executing and looking at the Speed-up we can see that this does not improve a lot the whole program. It is very similar than the Tree version. In any case, we would keep it because it is not problematic.



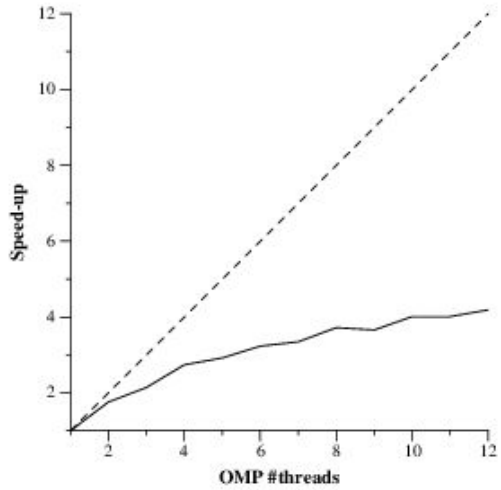
We see that the parallelization of the program starts earlier.

Dependent tasks

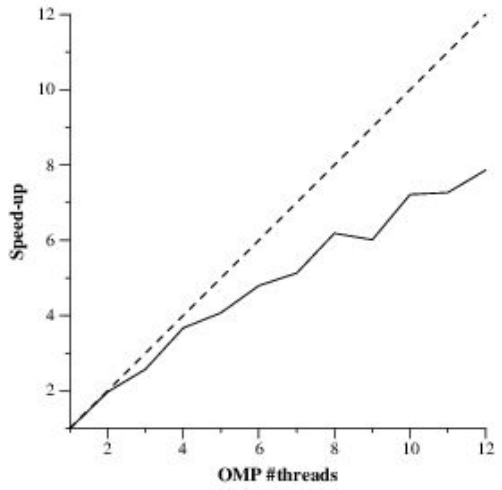
So far, the threats have **waited for all the tasks** it creates to continue. But sometimes it is not necessary. We can start the next tasks because it have no dependencies with the previous that still rinning.

First of all, it is really important to study the code, understand how it works and look for any task that may depend on another. Once these tasks are identified, **`#pragma omp task depend`** can be used to establish the proper execution sequence of tasks. That means that the **`depend(in)`** tasks wait until the **`depend(out)`** tasks are done.

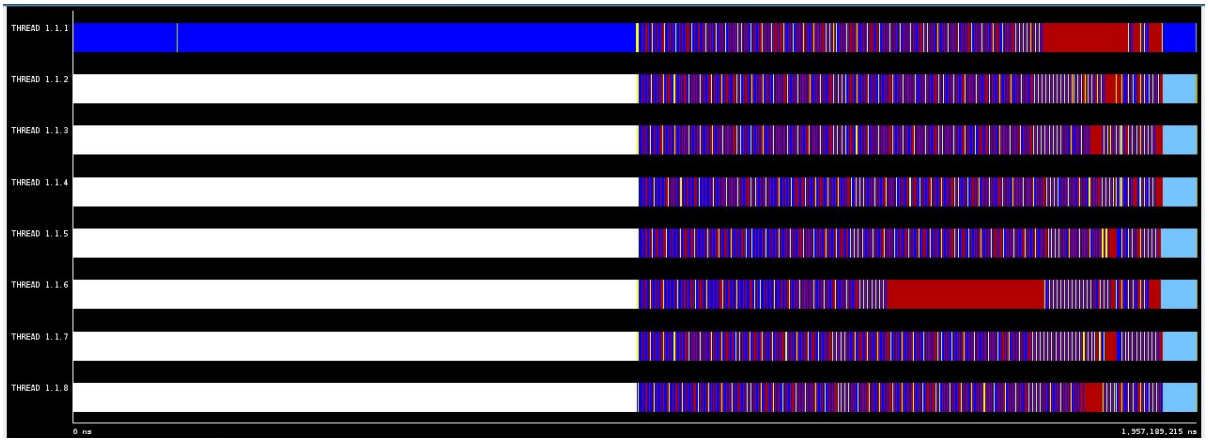
```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
	if (length < MIN_MERGE_SIZE*2L) {
		// Base case
		basicmerge(n, left, right, result, start, length);
	} else {
		// Recursive decomposition
		#pragma omp taskgroup
		{
			#pragma omp task
			merge(n, left, right,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



Speed-up wrt sequential time (complete application)



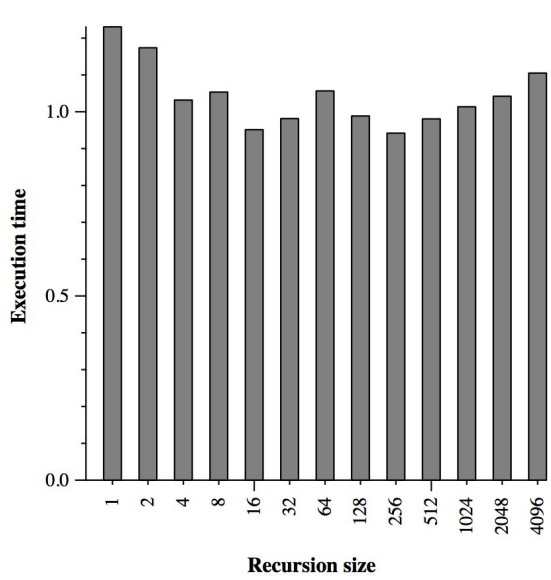
Speed-up wrt sequential time (multisort funtion only)



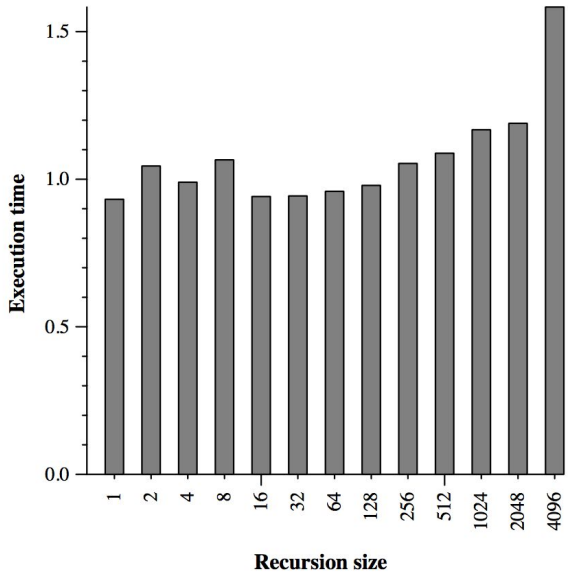
The parallelism that can be obtained is really limited because of the dependences between tasks, most of them have to wait to be executed. We can see in the plots above the results of these tasks dependences, the Speed-Up does not improve significantly. Also, looking at the paraver's traces we come to the same conclusion.

Sort and merge size

Now that we have found the best strategy and some little improvements like the initialization parallelism and the dependence pragma utility, we are going to modify both the sort and merge size for the basic cases.



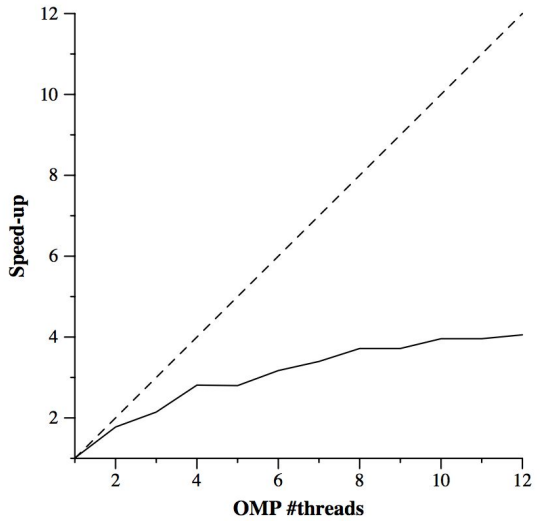
Average elapsed execution time (multisort only)



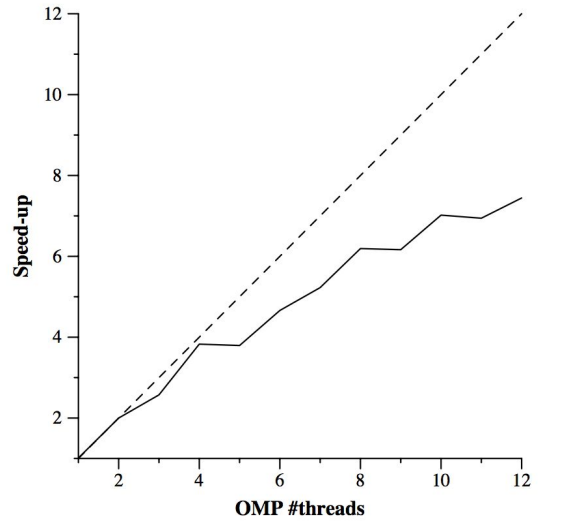
Average elapsed execution time (multisort only)

The first plot is the execution time with 8 processors changing the merge size, the second is the execution time changing the sort size.

We can see that the best merge size is 256 and the best sort size 16.



Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)

But when we execute the program with these paramethers we do not see a big change. It is better but not too much.

4. Conclusions

After all this different tests and all the little changes we have done we can say that in this type of problem the most important thing is the parallelization strategy, in this case the **Tree Strategy**.

The size, depth and the parallelization of the initial tasks are not that important.

We think that for this multisort problem the best you can do is:

1. Tree strategy
2. Dependencies ~~taskwait~~
3. `omp parallel for` in the initial loops
4. Sort size = 16
5. Merge size = 256