PAR Laboratory Assignment
Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Marc Domínguez de la Rocha
Joan Sánchez García
par2103

Fall 2017-2018
05/12/17

Index

4.1 Introduction

The *Divide and Conquer* strategy is so useful in the computer science and concretely in the algorithms field, that deserves to take a first approach on how to parallelize algorithms that uses that strategy. In this case, *multisort* algorithm is used as an example.
First a *Tareador* analysis of dependences will be performed. It will be followed by a study on the different strategies of task generation such as *Leaf* and *Tree* and finally a study on how the *synchronization* between tasks and the *recursion size* can modify the performance of our programs.
Every section demonstrates what exposes, a set of plots, diagrams and code can be found with it's own explanation whether on the section itself or the annexes.

4.2 Tareador analysis

```
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("basicmerge1");
		basicmerge(n, left, right, result, start, length);
		tareador_end_task("basicmerge1");
	} else {
		// Recursive decomposition
		tareador_start_task("merge-sub1");
		merge(n, left, right, result, start, length/2);
		tareador_end_task("merge-sub1");
		tareador_start_task("merge-sub2");
		merge(n, left, right, result, start + length/2, length/2);
		tareador_end_task("merge-sub2");
	  }
}

void multisort(long n, T data[n], T tmp[n]) {
	if (n >= MIN_SORT_SIZE*4L) {
		// Recursive decomposition
		tareador_start_task("sort1 task");
		multisort(n/4L, &data[0], &tmp[0]);
		tareador_end_task("sort1 task");
		tareador_start_task("sort2 task");
		multisort(n/4L, &data[n/4L], &tmp[n/4L]);
		tareador_end_task("sort2 task");
		tareador_start_task("sort3 task");
		multisort(n/4L, &data[n/2L], &tmp[n/2L]);
```

```
            tareador_end_task("sort3 task");
            tareador_start_task("sort4 task");
            multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
            tareador_end_task("sort4 task");

            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 {
            tareador_start_task("leaf task");
            // Base case
            basicsort(n, data);
            tareador_end_task("leaf task");
        }
}
```

code 1: multisort.c code with Tareador calls. Complete code can be found at annexes/src/multisort-tareador.c

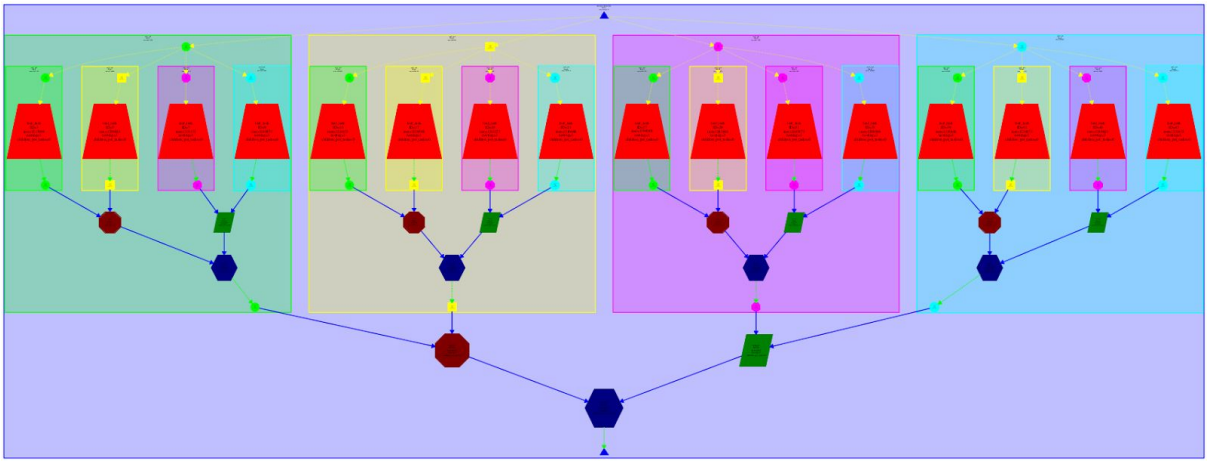

image 1:simplified dependences graph using Tareador. HD image and HD image of the complete graph can be found at annexes/images

The instrumented code enables the Tareador and creates the tasks where we placed the *tareador_start_task(name)* and *tareador_end_task(name*), which is in every function call, allowing us to analyze the *leaf* and *tree* strategy. The *leaf* strategy consists in creating tasks that executes the base case of the recursion. If you take a look at the task graph, every leaf is independent to each other, thus, we could parallelize without any overhead than creating the tasks itself. The *tree* strategy consists in creating tasks at every step we make across our tree, so, every task encapsulates all the next tasks to it. Important to see in the dependency

graph that every sort can be made in parallel but not the merges, you cannot merge before sorting the two parts that you are going to merge.

	1	2	4	8	16	32	64
Execution time	2033442001001 ns	1017371002001 ns	508680100001 ns	255037700001 ns	128989900001 ns	128989900001 ns	128989900001 ns
Speedup	1	1.998	3.997	7.973	15.764	15.764	15.764

Table 1: predicted execution times and Speedups for the multisort.c code with different number of processors

All execution diagrams can be found at the annexes/images/tareador directory.

From one to sixteen processors the speedups are pretty close to the ideal case, but from thirty-two to sixty-four they are far away due to the lack of tasks to distribute among threads. So, we have got some threads completely without workload, this can be seen in its execution diagrams.

4.3 Performance

Leaf

```
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 2: multisort.c with leaf parallelization. Complete code can be found at annexes/src/tasks-leaf/multisort-omp.c

In the *Leaf* strategy version we just need to parallelize the last level of the recursion, the so called leafs of the tree. According to this, we generate tasks before the *basicsort* and *basicmerge* calls, that are the ones called when the desired size of each vector that is going to be sorted or merged is achieved. In order to avoid that the program starts generating a new level of recursion before the above level is completed, we need to place a taskwait after all same level calls are called.

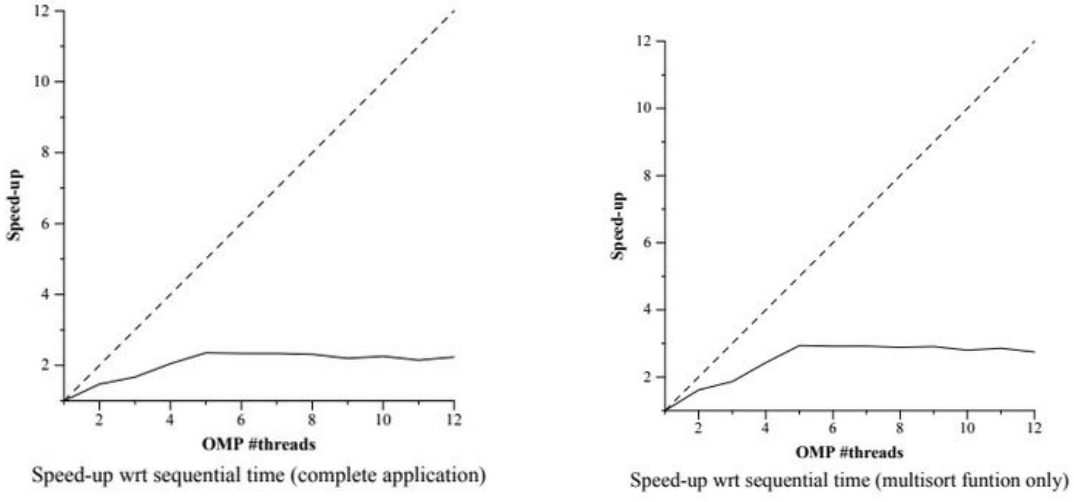

image 2: Speed-Up for the leaf parallelization. HD image can be found at annexes/images/tree-leaf/speedup-leaf-parallel.ps

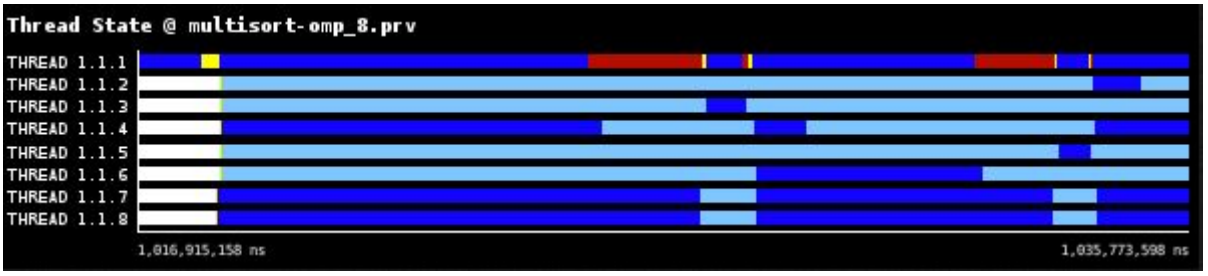

image 3: Execution diagram for the leaf parallelization.

Let's see first how much speed-up we can get in a theoretical frame. As we are only making tasks on our leafs, there will be as many tasks as leafs our recursivity tree has. Take a look at the trace analyzed with *Paraver*, so the maximum number of tasks is the number of leafs, but our tree has four leafs and our trace shows five threads. That is because we have a single thread executing the sequential code and the rest executing the tasks, which is one plus four, five threads. As that strategy can only give work to five threads, increasing the resources (number of threads) is pointless. This is demonstrated by the plot, that shows how the speed-up maintains constant at the point five threads are reached.
Note that the cyan parts of the trace are *Idle* and the deepest blue are *Working*.

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
		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);

        }
}
```

code 3: multisort.c with tree parallelization. Complete code can be found at annexes/src/tasks-tree/multisort-omp.c

In the *Tree* strategy, we parallelize every call inside a same level of recursivity (we create a new task), improving a lot the speed up. To avoid that a sublevel starts executing before the parent level has finished, we make use of the taskwait directive to do synchronizations. So, if we compare both strategies, we can see that one is the opposite of the other. While the leaf parallelization only parallelizes the leafs (the last level of recursion), the tree parallelization parallelizes every call except the last one.

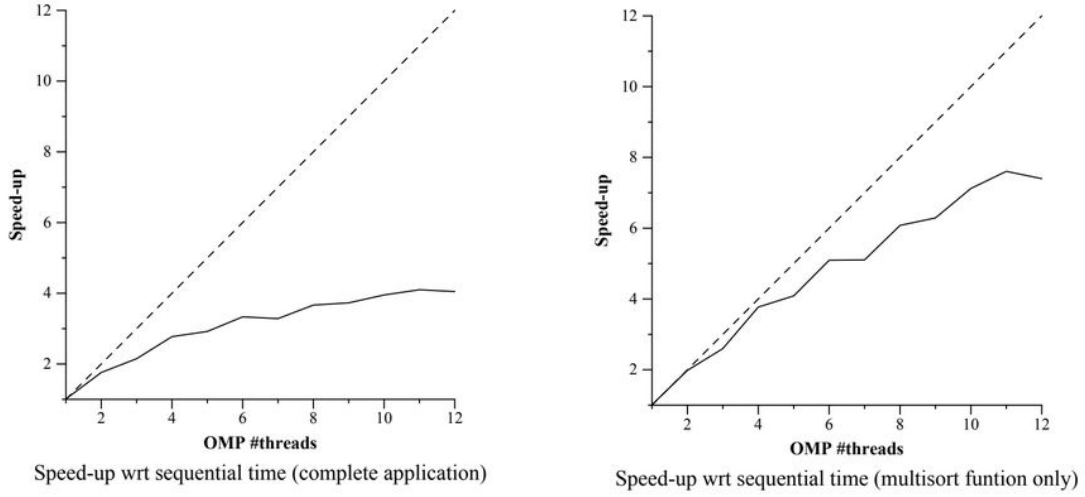

image 4: Speed-Up for the tree parallelization. HD image can be found at annexes/images/tree-leaf/speedup-tree-parallel.ps

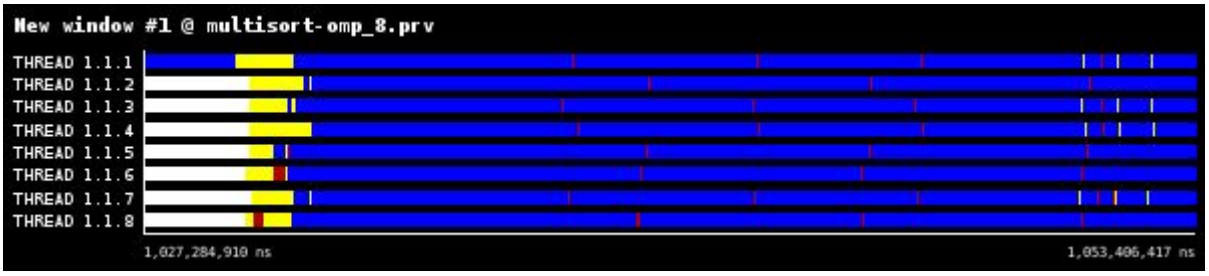

image 5: Execution diagram for the tree parallelization.

In the *leaf* case we observed a lack of tasks to keep our threads working that lead to a poor strategy where strong scalability was roughly null. In the *tree* strategy every node creates a tasks, then that lack is avoided so every ready thread in our parallel program can work, that is taking a task from the pool and execute it. Despite is not ideal as we would want to be, the speed-up is greater than with the *leaf* one, as the deepest blue of our *Paraver* trace is time whereas a thread is working, compare both blue densities of each trace, *tree* strategy trace blue density is clearly greater than the correspondent to *leaf* strategy trace.

Task dependencies

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

        }
}
```

code 4: multisort.c with task-dependencies parallelization. Complete code can be found at annexes/src/tasks-tree-dependencies/multisort-omp.c

Now we are making use of the depend clauses, which behaves similar to taskwait but allow us to specify which part of the parent task are we waiting for. The first level of recursion, as do not have any parent task, just needs to notify the dependent task when finish. The second level needs to wait for the two parent tasks that will merge, that is why there are two *depend in* specified in each one. As the last level only has two parents, it will be the same to place a taskwait than a task depend in from the second level.

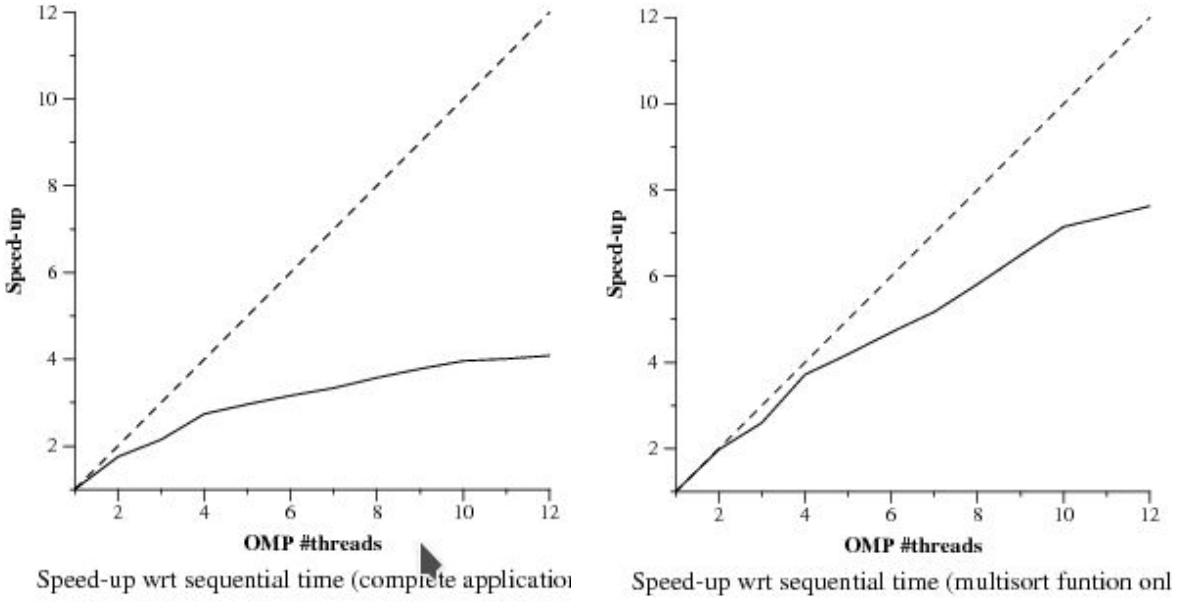

image 6: Speed-Up for the task-dependencies parallelization. HD image can be found at annexes/images/tree-strategy-depend/multisort-omp-strong.ps

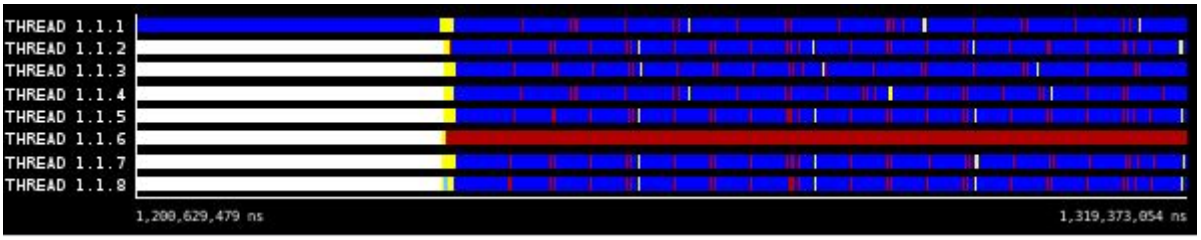

image 7: Execution diagram for the task-dependencies parallelization.

The speed-up obtained in the task depend version is a little bit better than the taskwait version we used in the tree section, also, is softer than any speed-up plot we showed before. Introducing task dependences allows us to wait less time, now we are only waiting for the **real** dependences between those tasks. So ideally, threads will be more time working than before (that means not being in *synchronization* state), exactly in our case a thread will be waiting for tasks to complete if there is a direct dependence between them, which will result in *merge* tasks waiting only for the *multisort* that actually sort the vector region belonging to it. Then, threads will wait around one half the time they used to wait, this would be a greater speed-up if one of the threads, exactly THREAD 1.1.6 in the *Paraver* trace would not be used for the synchronization among threads.

4.4 Depth analysis

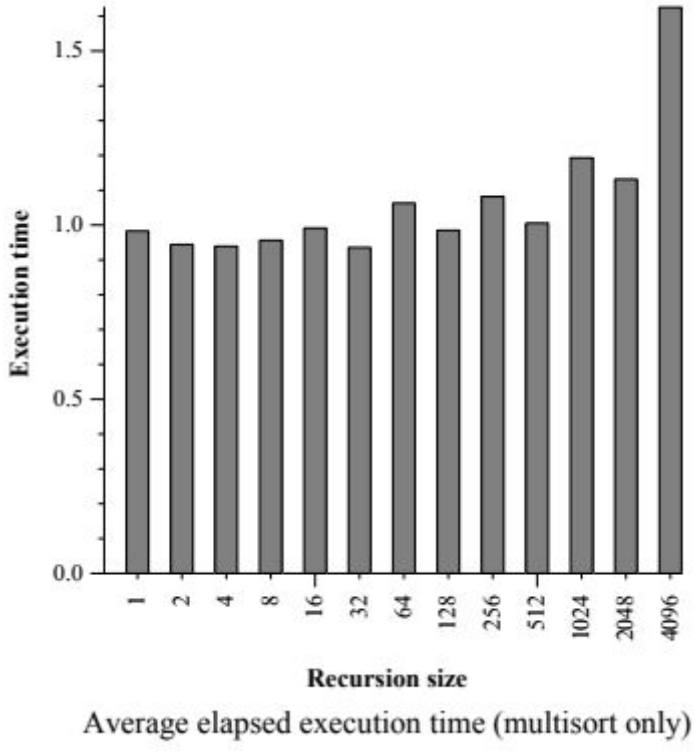

image 8: Average elapsed execution diagram changing the recursion size.

As we can see in the graph above, as bigger the MIN_SORT_SIZE is, longer time the program takes to be executed. This can be explained looking at the code at tree strategy.

When the recursion size is kept small, it means that the vector will be looping in the

```
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

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

for more time, and as the tree parallelization allow us to task every call to the recursion, we will be executing it in parallel nearly all the time.
Only when the size reaches 1 (in the first case, 2 in the second, 4 in the third...) we will call the basicsort, the non parallel function.

So as we increase the size of the minimum vector size to be sent to the basicsort function, it means that we are decreasing the possible parallelization of the code and charging with more work load the serial part. That’s why the 4096 size takes the bigger execution time.

As smaller the MIN_SORT_SIZE is, more smaller tasks we will have, and as bigger it is, less bigger tasks we obtain.

4.5 Optionals

Optional 1

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

code 5: multisort.c with vector initialization parallelized. Complete code can be found at annexes/src/optional1/multisort-omp.c

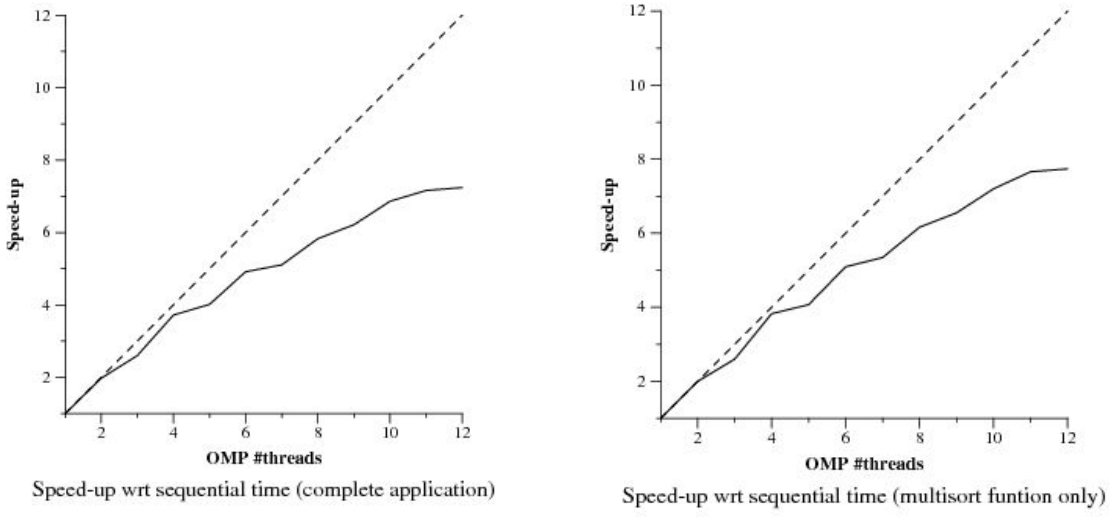

image 9: Speed-Up for the vector initialization parallelization. HD image can be found at annexes/images/optional1/multisort-omp-strong.ps

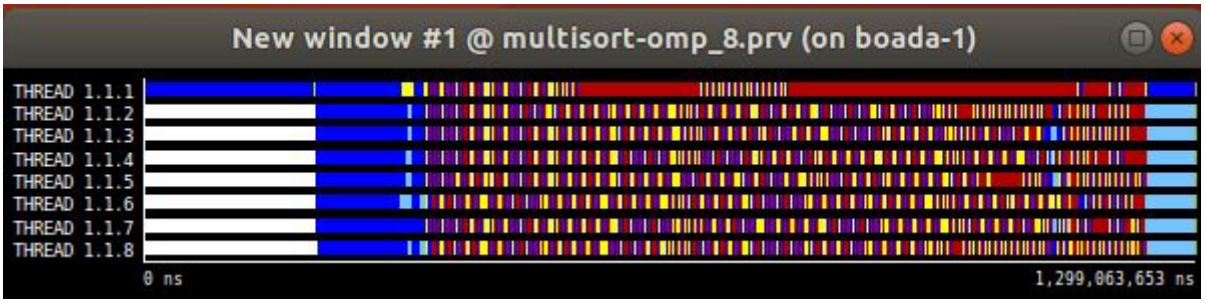

image 10: Execution diagram for the vector initialization parallelization.

As we can see comparing image 9 and image 4, the speedup obtained is much higher with the vector initialization parallelization strategy. This can be simply explained by the fact that we have parallelized a region of the code that used to be sequential. As we have used the *#pragma omp parallel for* directive, an assignation of iterations will be done for each thread, so the expected gain will be diminished by all the synchronization (red and yellow) between threads, as we can observe in image 10.

Optional 2

Before inserting the plots is required to understand which parameters are being changed, as sort size or merge size is modified we are changing elementally the same, a moment when an action is done, sorting or merging in this case. So, we found out by trial and error that the best parameters for our code are 4 and 32 for sort size and merge size respectively.

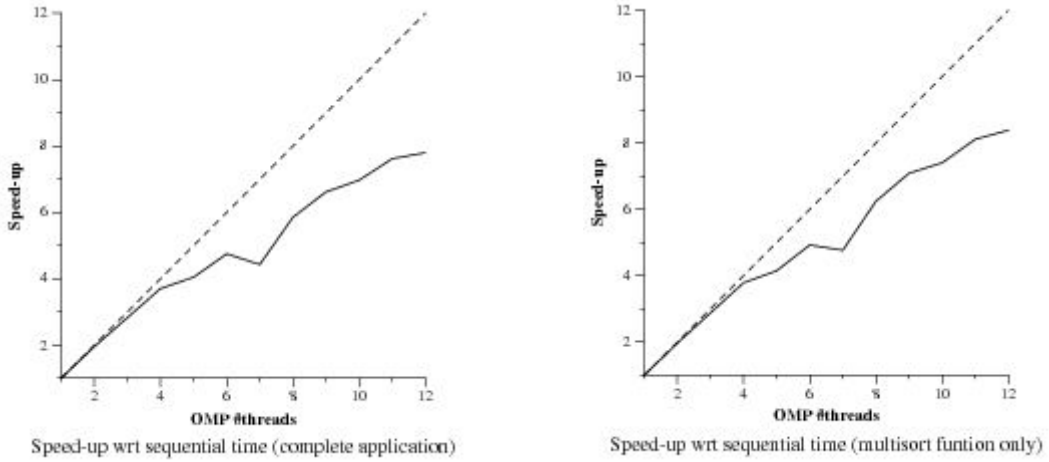

image 11: Execution diagram submitting with the modified sort size and merge size.

The performance with the new parameters better than before, but not significantly.

4.6 Conclusions

As we have seen along the document, the speed up of a code is not improved by just applying or not parallelism, but the place where it is applied. That is why the execution time of the program with tree parallelization, which means creating a new task for every new level of recursion, will be much better than the leaf parallelization, where we just parallelize the last level. Once we have seen which parallelization strategy is better, we try to improve the speed up with the tools OMP provide us, the "*depend*" directive. Thus, this give us the great opportunity of waiting what is really needed. This little change allow us to start a dependent task as soon as its data is ready, avoiding waits for the upper level to be completed.
Finally in the optionals we parallelize also the initialization of the data vectors for the tree strategy, improving by far the complete application speed up.
Once we finished this part, we were curious to see if the depend strategy also improved the speed up by a lot, and so it was as you can appreciate in the following graphs.

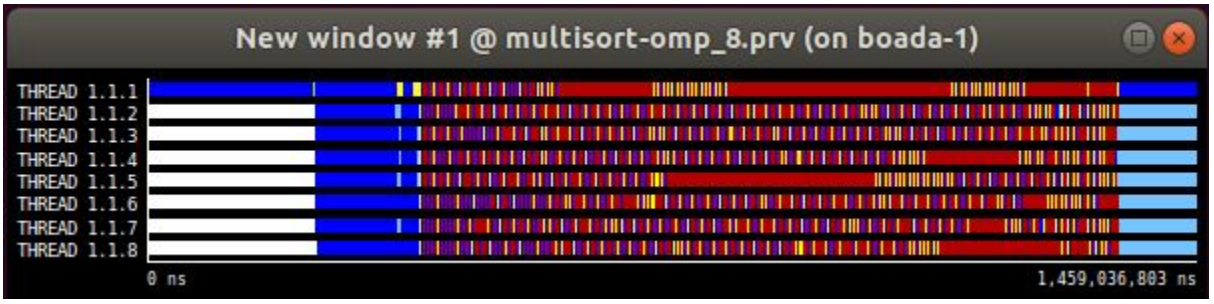

image 12:Execution time for the depend strategy with vector initialization strategy.

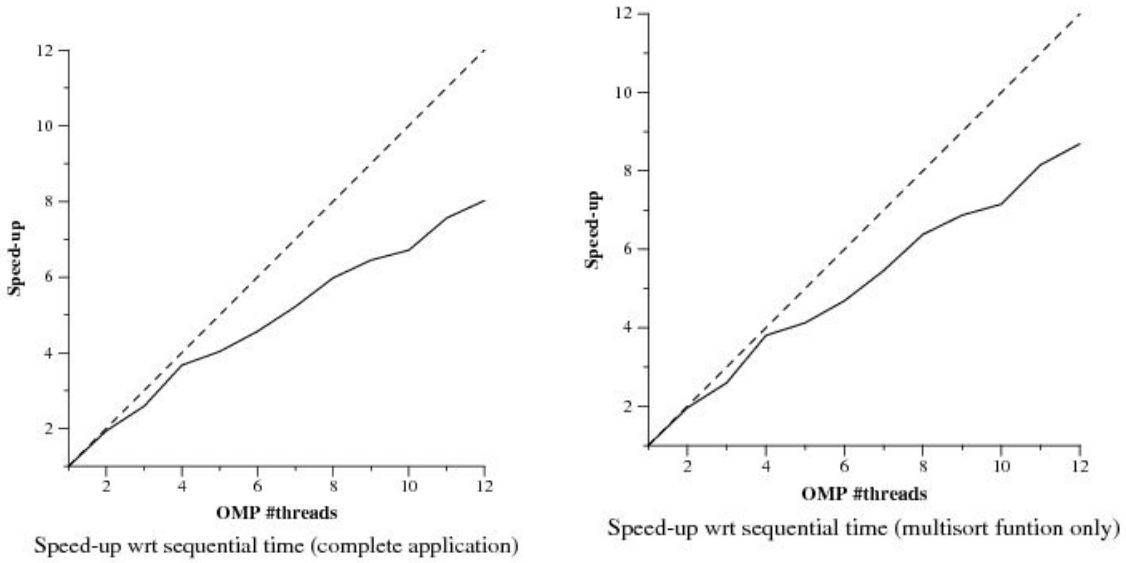

image 13:Speed Up for the depend strategy with vector initialization strategy.