4th deliverable
Divide and Conquer Parallelism with OpenMP Sorting

Omair Iqbal
Victor Gallego
Group 11
Par1103
May 2016

Introduction

In this lab deliverable we are going to analyze the performance of the multisort algorithm applying different parallelization strategies.
The algorithm combines a ”divide and conquer” mergesort strategy that divides the initial list into multiple sublists recursively, a sequential quicksort that is applied when the size of these sublists is sufficiently small, and a merge of the sublists back into a single sorted list.

Parallelization strategies

Two parallelization strategies are going to be implemented: leaf and tree.
Leaf strategy consists in create a single task for every leaf of the recursion tree (when the recursive call stops), this can a be a good strategy when the height of the tree is small but when it's not, it can create a big overhead and degrade the performance.

In the other hand the tree strategy consists of creating a task for every recursive call, the problem we can have is that we are creating a single task for every recursive call and that can create a big overhead when reaching the leafs of the recursion tree, this can be solved by putting a limit of task created.

This two basics strategies are going to be implemented and their performance to be compared and analyzed.

Analysis with Tareador

1. Include the relevant parts of the modified multisort-tareador.c code and comment where the calls to the Tareador API have been placed. Comment also about the task graph generated and the causes of the dependences that appear.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
      tareador_start_task("Merge");
      /* merge function code */
      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");

            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);

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

Figure 1: Multisort-tareador.c with Tareador APIs added in bold.

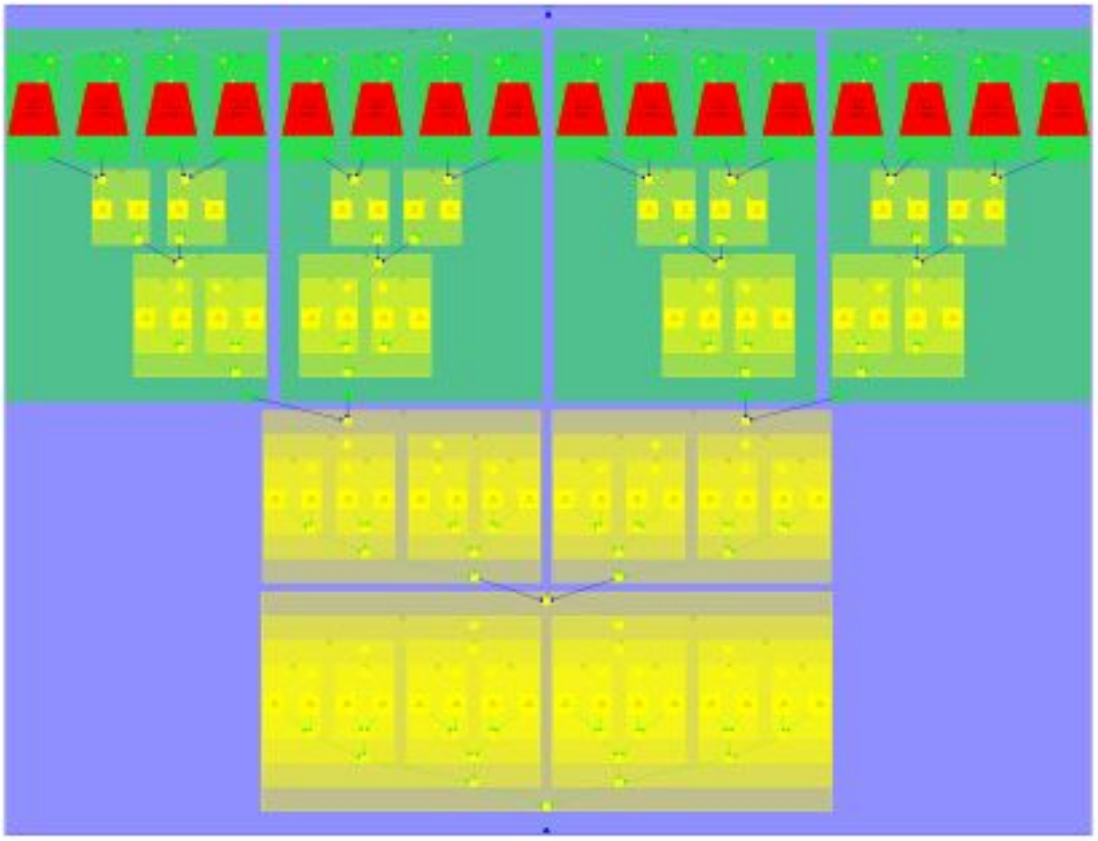


Figure 2: Task graph obtained by generating the code in figure 1 with Tareador.

In the function multisort, if the size of the vector to order is larger than the determined size then it will make 4 recursive calls that divide the vector into 4 parts (v1, v2, v3, v4). Then it will merge v1 and v2 creating v12, doing the same with v3 and v4.
Finally, the latest call to function merge will return an ordered vector with the results of v12 and v34. The recursion will end when the size of the subvector is small enough to reach the base case and call basicsort function.
The two first merge calls will have to wait for the 4 multisort recursive call to end. The third and last merge call won't start until the previous merge calls have ended.
The merge function receives two vectors which are already ordered and merges them in a one single vector result. When the length is small enough, we'll have reached the base case (we'll call basicmerge).

2. Write a table with the execution time and speed-up predicted by Tareador (for 1, 2, 4, 8, 16, 32 and 64 processors) for the task decomposition specified with Tareador. Are the results close to the ideal case? Reason about your answer.

	Execution time	Speed-up (respect from 1 processor execution)
1 processor	20,286,799,406 ns	-
2 processors	10,384,905,001 ns	1.95348917
4 processors	5,387,245,001 ns	3.76570945
8 processors	2,924,461,001 ns	6.93693621
16 processors	1,691,696,001 ns	11.9919887
32 processors	1,691,696,001 ns	11.9919887
64 processors	1,691,696,001 ns	11.9919887

The results is closer to the ideal Speed-up until we reach 16 processors after which theres no improvement because this 16 processors are the Pmin, which means that even if we add more threads the time execution will be the same. However, Tareador just simulates, it does not consider overheads, which is an important factor to consider.

Parallelization and performance analysis with tasks

1. Include the relevant portion of the codes that implement the two versions (Leaf and Tree), commenting whatever necessary.

```
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);
	}
}


Int main (int argc, char **argv) {
/* Code */
 #pragma omp parallel
 #pragma omp single
```

```
multisort(N, data, tmp);
/* Code */
}
```

Figure 3: multisort-omp.c with leaf parallelization strategy implemented with openMP directives in bold.

```
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 {
        #pragma omp taskgroup
        {
        #pragma omp task
        // Recursive decomposition
        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) {
        #pragma omp taskgroup
        {
        // 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]);
        }
```

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

Int main (int argc, char **argv) {
/* Code */
 #pragma omp parallel
 #pragma omp single
multisort(N, data, tmp);
/* Code */
}
```

Figure 4: multisort-omp.c with tree parallelization strategy implemented with bold directives in the code.

2. For the the Leaf and Tree strategies, include the speed–up (strong scalability) plots that have been obtained for the different numbers of processors. Reason about the performance that is observed, including captures of Paraver windows to justify your explanations.

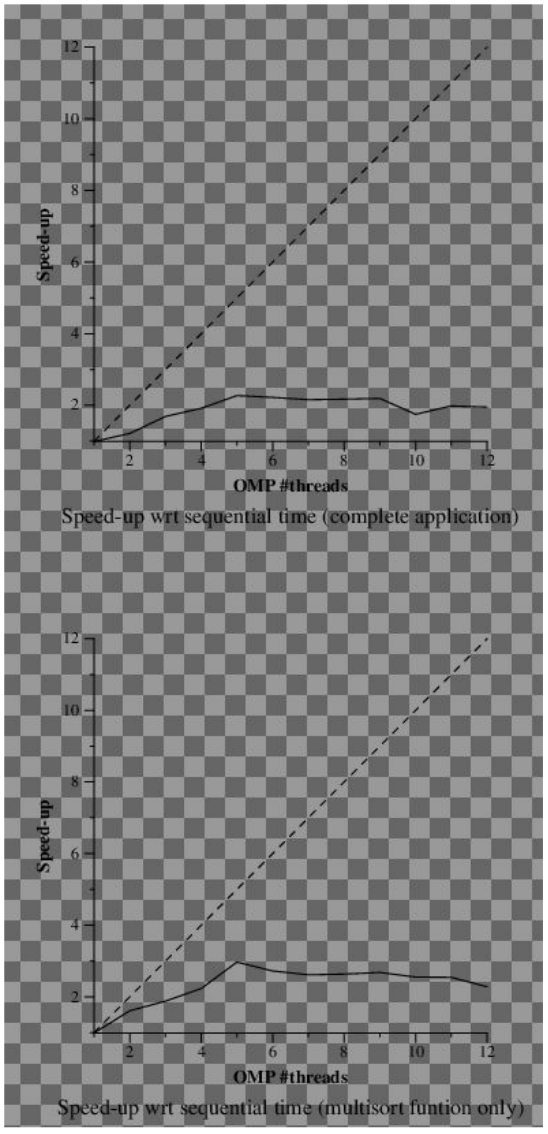


Figure 5: Speed-up of leaf strategy

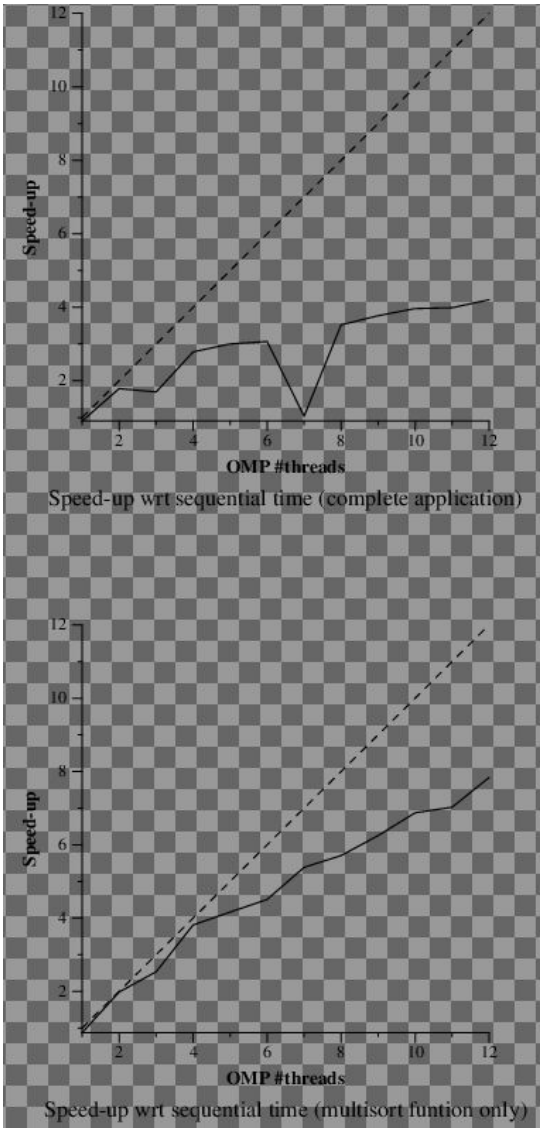


Figure 6: Speed-up of tree strategy

The speed-up of the leaf version is lower than the tree version because in the leaf strategy we have just one thread creating the leaf tasks, by the time it reaches the base case that thread won't be able to create a new task until the previous one is finished, so the task generation will be done sequentially.
In the tree strategy we have a task every time we do a recursive call.
Conclusively the execution time in the tree version is smaller than the leaf version.

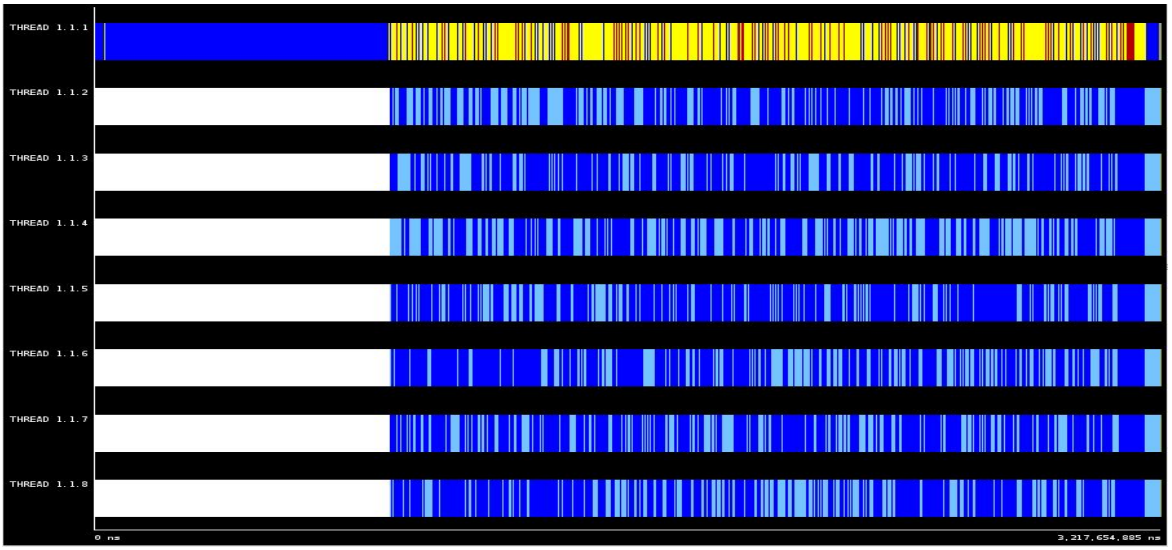


Figure 7: Paraver capture of the leaf version of multisort-omp.c

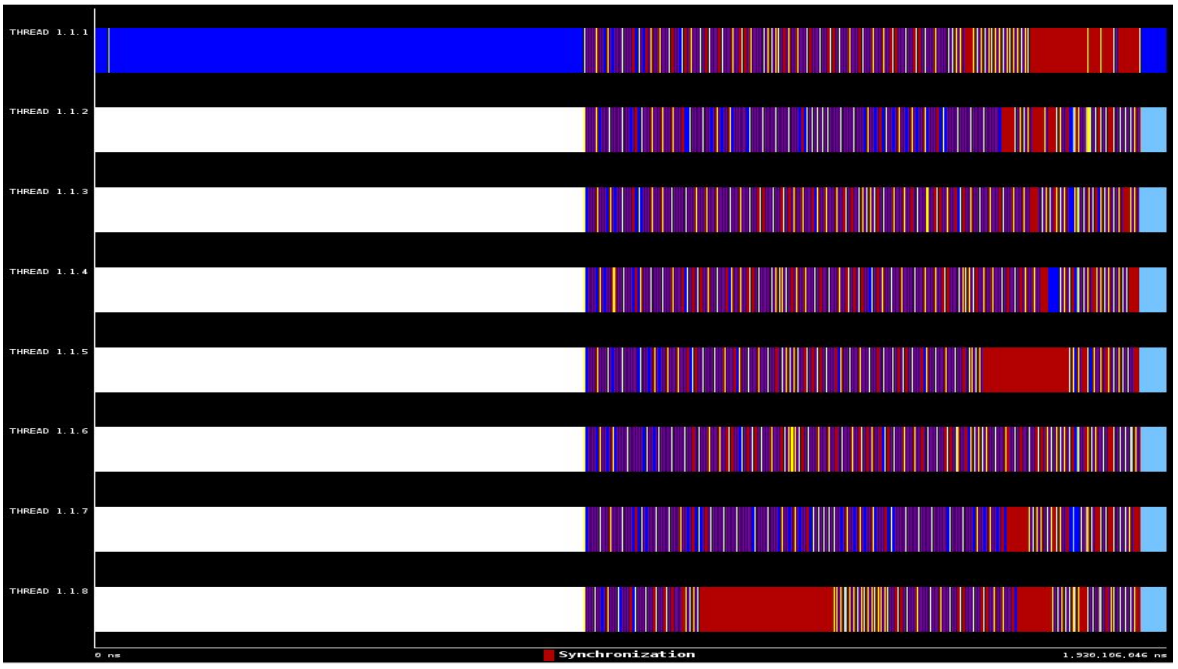


Figure 8: Paraver capture of the tree version of multisort-omp.c

Comparing the two previous traces, we can see that the tree strategy is a lot faster than the leaf one.

3. Analyze the influence of the recursivity depth in the Tree version, including the execution time plot, when changing the recursion depth and using 8 threads. Reason about the behavior observed. Is there an optimal value?

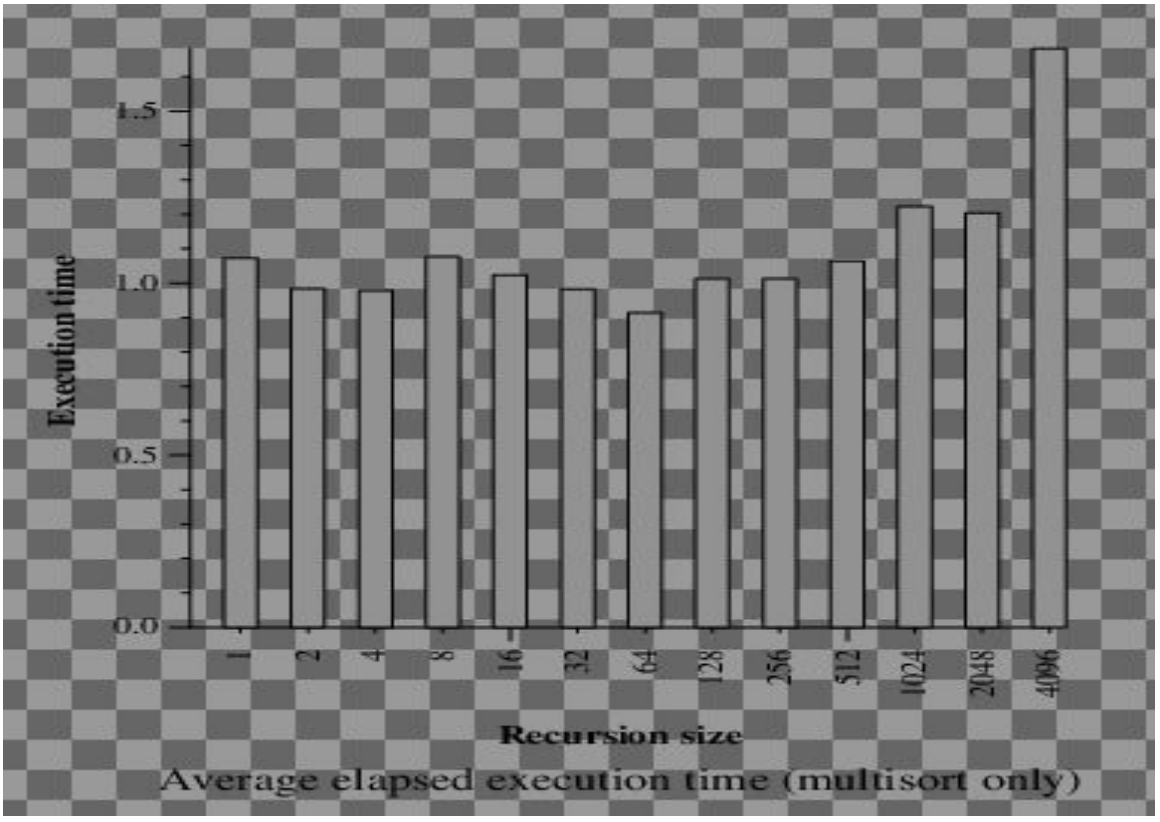


Figure 9: Depth in tree version 8

As we can see in the execution plot at the biggest values the execution time also increases, but looking at the rest of the values we cannot see a real tendency between recursion size and execution time. This can be explained due to the fact that when we cut the tree with a higher value, we lose potential parallelism (but we also get less overhead because there will be created a less number of task and that makes it difficult to determine which recursion size is the optimal).

Parallelization and performance analysis with dependent tasks

1. Include the relevant portion of the code that implements the Tree version with task dependencies, commenting whatever necessary.

```
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]) 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);
   }
}
```

Figure 10: multisort-omp.c with a tree parallelization strategy using task dependences

In the multisort function part of the code we've exchanged the taskgroups for the task depends which we can use to determine the variable dependences between tasks.
In the merge function it is not necessary to change the previous pragmas because there are no dependences between them.
For code correctness we have added the taskwaits at the end to wait for all the tasks of the same level of recursion.

2. Reason about the performance that is observed, including the speed–up plots that have been obtained different numbers of processors and with captures of Paraver windows to justify your reasoning.

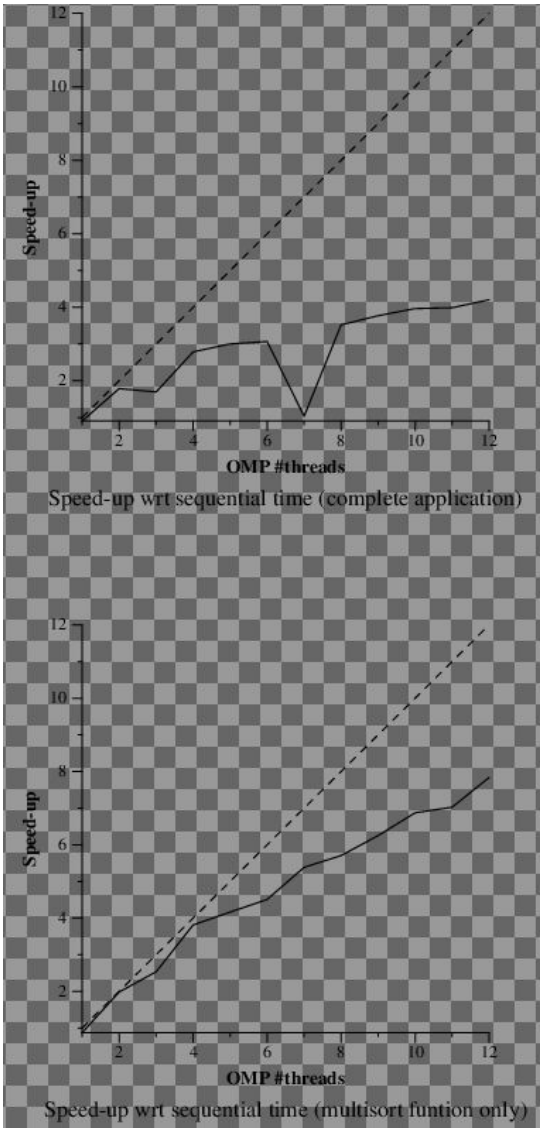


Figure 11: Speed-up of tree strategy without dependent tasks

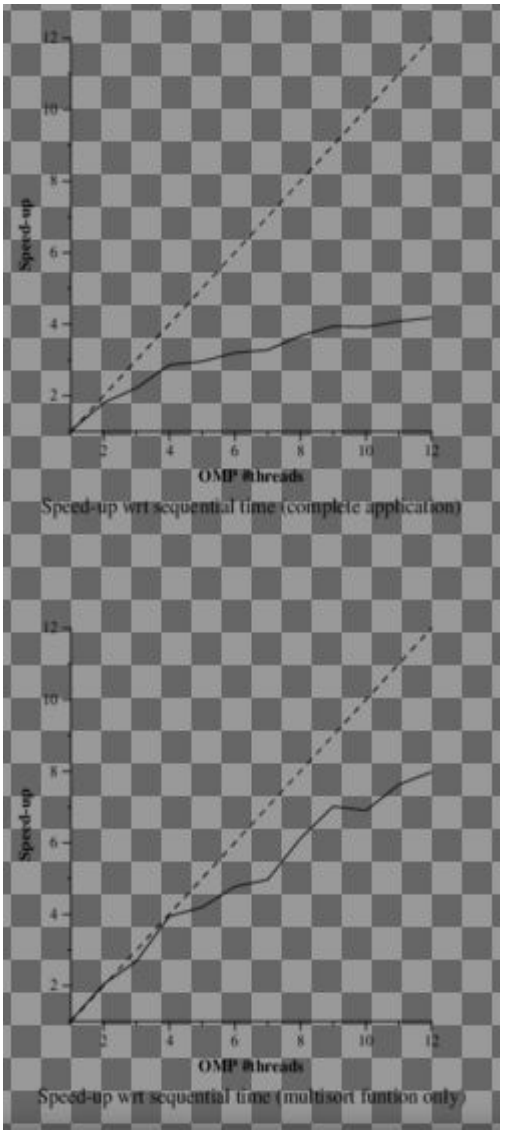


Figure 12: Speed-up of tree strategy with dependent tasks

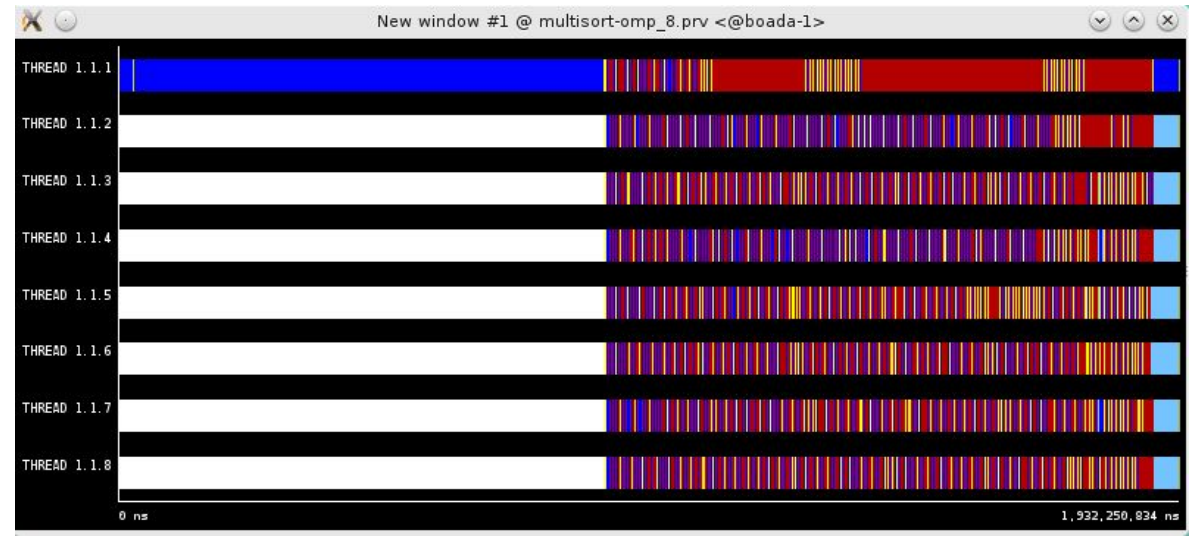


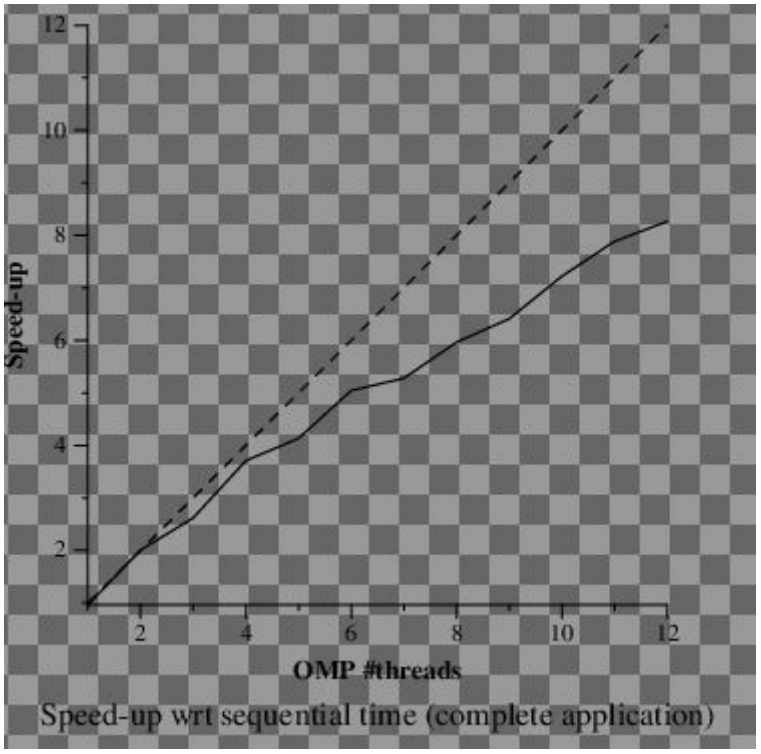
Figure 13: Paraver trace of tree strategy with dependent tasks

We can see in the speed-up plots that there is no significative gain between both implementations. Even though the version with dependent tasks is faster than the tree version ((if we look at the traces time of tree we can see that is a bit slower than this one) and a lot faster than leaf version version for the same reason.

Optional 1: Complete the parallelization of the Tree version by parallelizing the two functions that initialize the data and tmp vectors1 . Analyze the scalability of the new parallel code by looking at the two speed–up plots generated when submitting the submit-strong-omp.sh script. Reason about the new performance obtained with support of Paraver timelines.

```
static void initialize (long length, T data[length]){
   #pragma omp parallel
   {
   long i;
   #pragma omp 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;
   }
}
```

Figure 14: initialize and clear functions parallelized with pragma directives in bold.



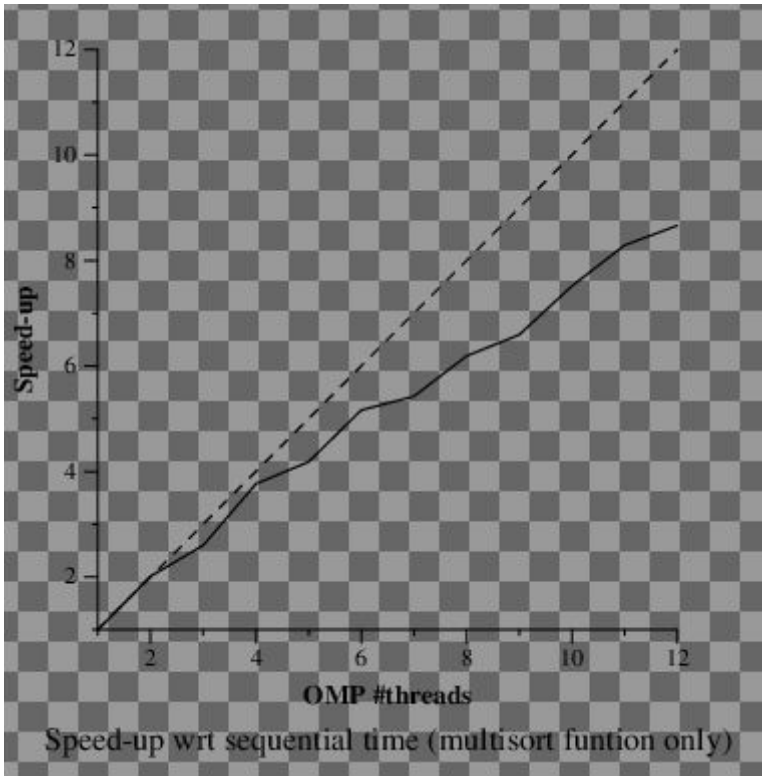


Figure 15: Speed-up plots of the tree version with the 2 extra functions parallelized

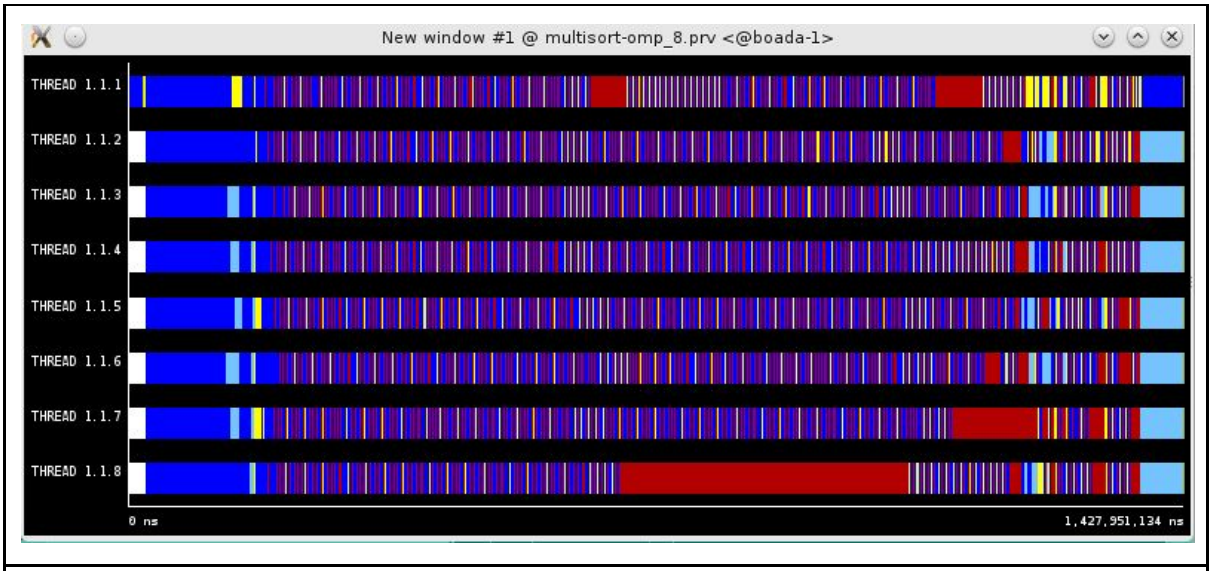


Figure 16: Paraver trace of the tree version with the 2 extra functions parallelized

As we can see in the figuree 15, the global speed-up has increased and we can also see in figure 16 in the Paraver trace that the execution time of this version is slower thatn the tree version.

Conclusions

The main conclusion we can get from this lab session is that the tree version is faster than the leaf one, obviously the tree strategy is not the best in every case but at least we can say that in our sort function is the appropriate as we can saw throughout the multiple exercises done.