LAB 4:
DELIVERABLE

Jacobo Moral
Carles Pàmies
PAR1401
Q1 2017/2018

INDEX

1. 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 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("multisort_3");
                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
                tareador_start_task("basicsort");
                basicsort(n, data);
                tareador_end_task("basicsort");
        }
}
```

Figure 1: Code for multisort function using Tareador API

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

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

Figure 2: Code for merge function using Tareador API

We used *tareador_start_task(name)* and *tareador_end_task(name)* from Tareador API, before and after, respectively, on each invocation of *basicmerge, merge, multisort* and *basicsort (figures 2 and 3).* This way, we could then open the program graphically so we observed the functional dependences between the severall calls on the program (figure 3).

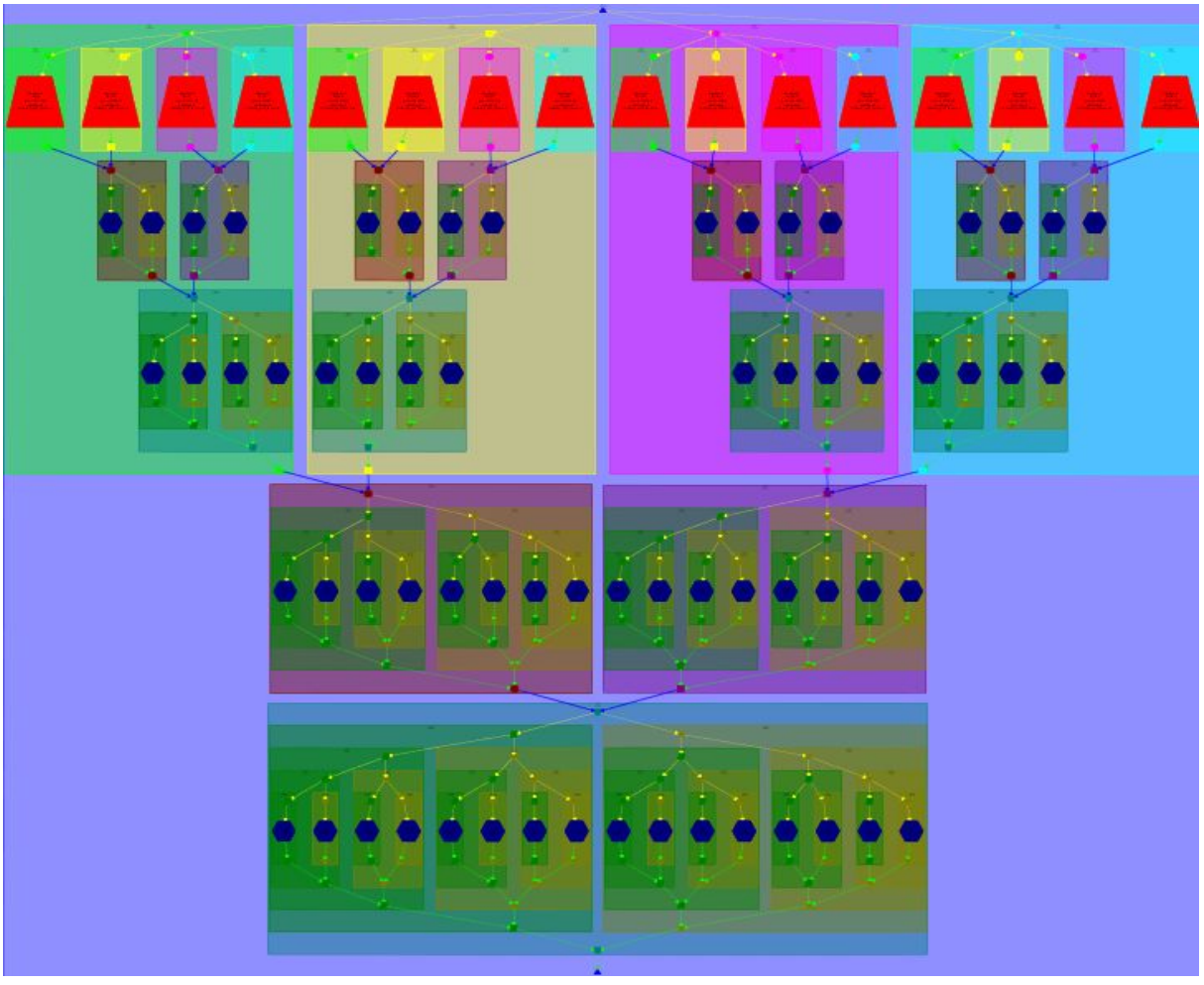


Figure 3: tasks dependence graph using Tareador API

As it's clearly shown in figures 1 and 2, we have used a mix of leaf and tree task decomposition, and this is reflected in figure 3.

The graph also shows us that the initial vector is divided in small segments over and over until its size reaches the size we want. Then we merge them ordered by pairs until we have the original vector once again, but ordered this time.

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.

Number of processors	Execution time	Speed-up
1	20334,42	1
2	10173	1,998862
4	5086	3,998116398
8	2550.60	7,972406493
16	1289.92	15,7640939
32	1289.92	15,7640939
64	1289.92	15,7640939

Figure 4: Execution time and speed-up according to Tareador.

We can see at the table above (figure 4) that the results are close to the ideal case -half the speed with double the threads and so on- until we reach 16 threads. This is the maximum number of threads we can have with maximum speed-up. 32 and 64 threads do not improve the program execution time any more.

2. 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
                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: Code for tree recursive strategy using OpenMP libraries

```
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);
                #pragma omp taskwait
        } else {
                // Base case
                #pragma omp task
                basicsort(n, data);
        }
}
```

Figure 6: Code for leaf recursive strategy using OpenMP libraries

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

Figure 7: Code for initial call in leaf and tree recursive strategies using OpenMP libraries

We can see in Figure 5 the implementation of the tree recursive strategy for resolving the problem. In this case, we define a task as every single call of merge and multisort functions. This way we create more and more tasks, and every new task creates some new ones. When the recursive case of the problem ends and we get to the base case, every task created before executes basicsort and basicmerge functions.

On the other hand, leaf recursive strategy (figure 6) works totally different. We define a task as the call of basicsort and basicmerge functions (the base cases). There's

only one thread creating all the tasks in the recursive part of the problem. When it gets to the base case, a new task is created (and executed therefore) and the main threads returns to the recursive case again.

Both implementations need to have *pragma omp parallel*, which defines that the following code will be parallelized using all available threads (unless otherwise is specified) and *pragma omp single* which notes that only one thread will be executing next code (figure 7). This allows us to have one initial thread to execute the code, which will be the one creating all the tasks for the other threads to execute.

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.

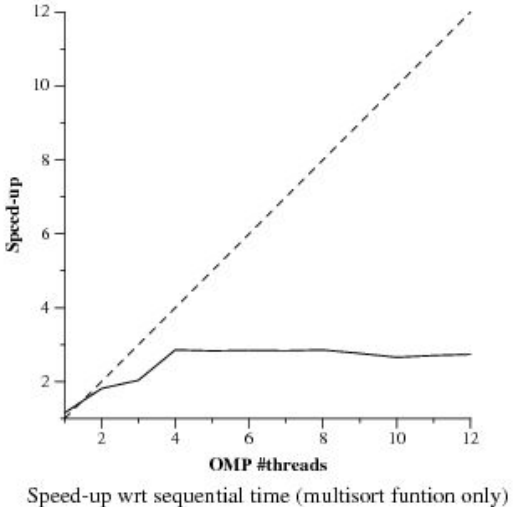


Figure 8: Strong scalability plot for the multisort function implementing leaf strategy

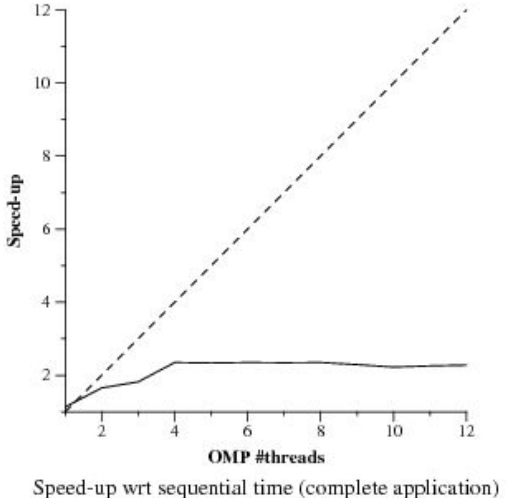


Figure 9: Strong scalability plot for the whole program implementing leaf strategy

Figure 10: Paraver trace for leaf strategy

We can see in figures 8 and 9 that the speed-up of leaf strategy is close to be linear up to 4 threads. Then it doesn't improve anymore if we add more threads to the execution. The reason is that every call to multisort function makes 4 more calls to itself.

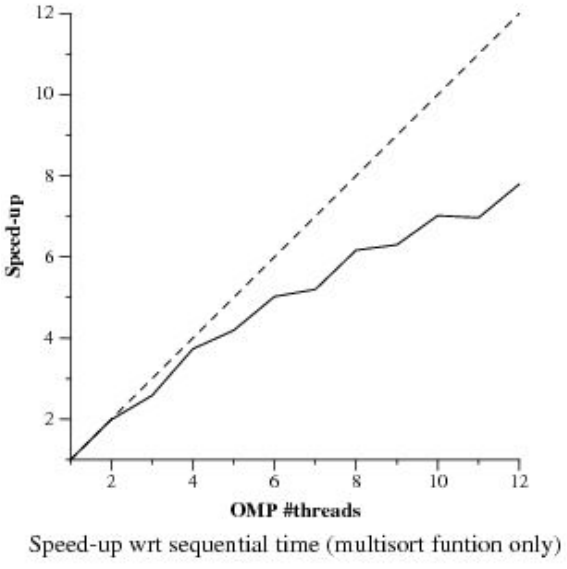


Figure 11: Strong scalability plot for the multisort function implementing tree strategy

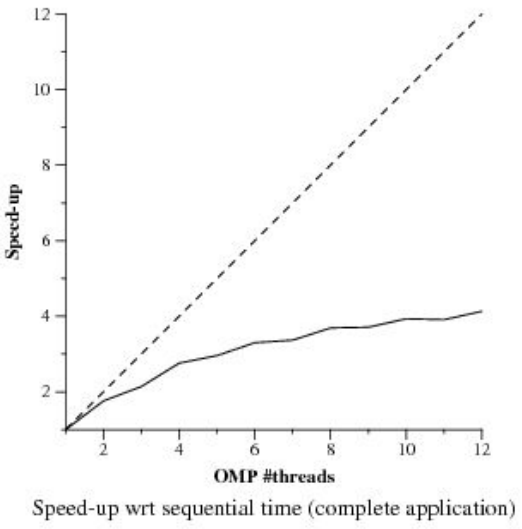


Figure 12: Strong scalability plot for the whole program implementing tree strategy

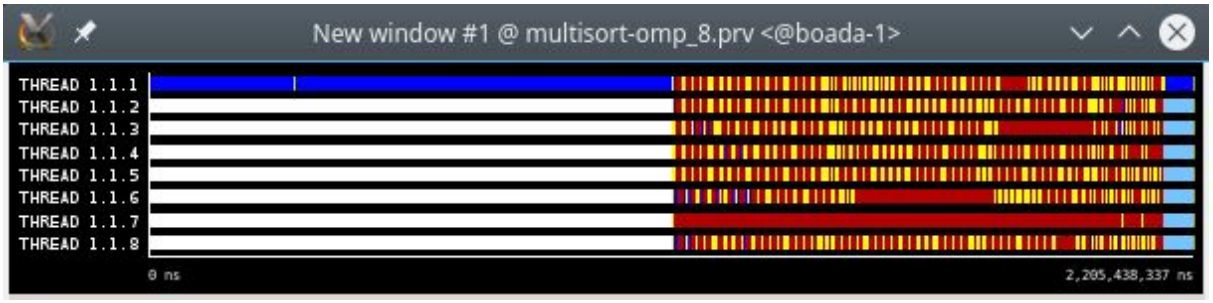


Figure 12: Paraver trace for tree strategy using taskwait

This version is clearly better than the leaf version despite the overhead associated, each recursive call before reaching the leaves in multisort receives a new task, exploiting parallelization in a much higher level.

Using tree strategy we obtain much better performance, as we can see in figures 11 and 12. The speed-up for the multisort function stays close to linear. We don't really care about complete application speed-up, as we are only making changes to multisort function, which is a small portion of the whole program.
We can improve the lack of parallelization caused using leaf decomposition strategy.

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?

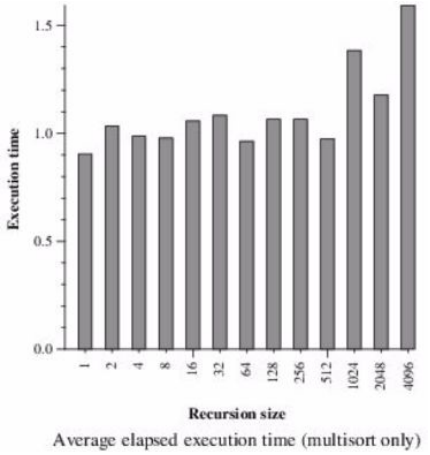


Figure 13

We can see in figure 13 that recursion size has no apparent impact on the elapsed time of the program. Even though, we can also see that large recursion times (1024 or higher) have a negative influence towards the time. This is caused because it leads to a lower number of generation of tasks, which also lowers the parallelization fraction of the program.

3. 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: t
                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 14: Code for tree recursive strategy with depend using OpenMP libraries

In this code from Figure 14 we see the needed instructions for the implementation of the Tree version with task dependencies. The recursive decomposition in the merge function works the same way as before but we add the “depend” clauses in the multisort recursive decomposition. This directive makes the tasks which contain a *in* in the depend wait for the tasks that generate that *in*. Those clauses create dependencies between the multisort calls pairing the data[0] with data[n/4L] and the data[n/2L] with [3*L/4*L]. When those pairs are done, the merge calls are executed.

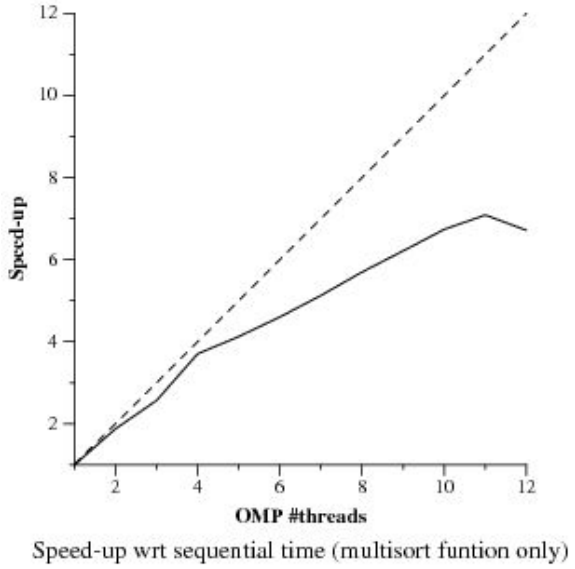


Figure 15: Strong scalability plot for the multisort function implementing tree strategy with depend clause

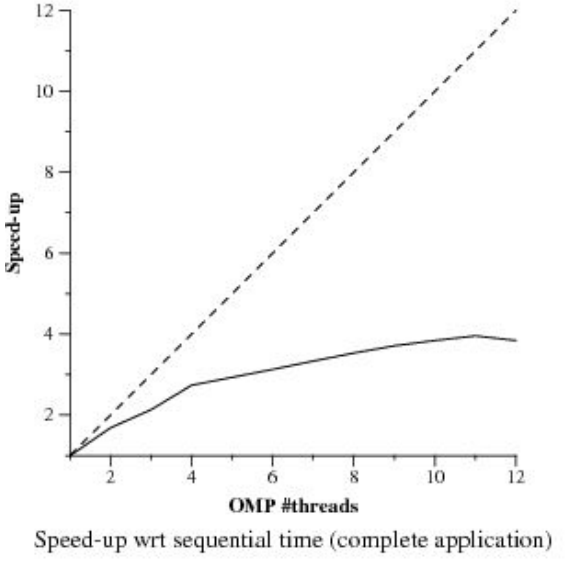
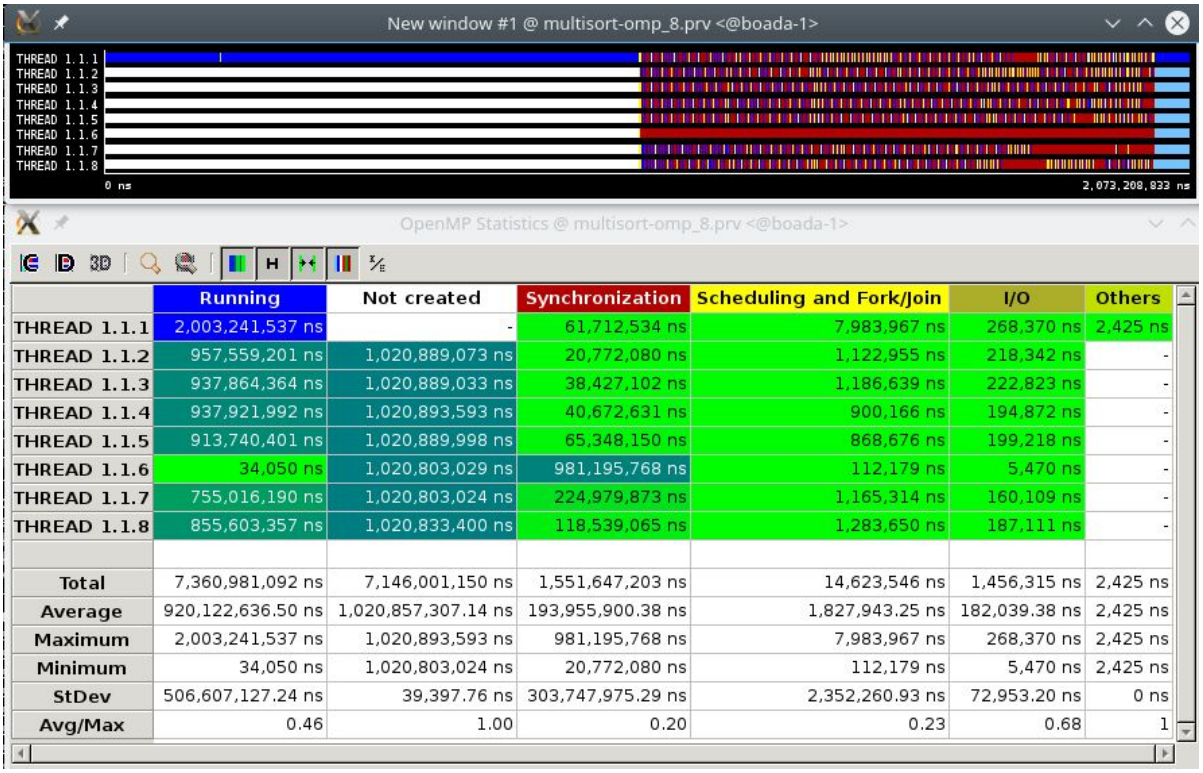


Figure 16: Strong scalability plot for the whole program implementing tree strategy with depend clause

Observing these plots (figures 15 and 16), we can say they’re are pretty similar to the plots generated when using taskwait (figures 11 and 12).

Seems obvious that depend clause would improve performance over taskwait usage. This is because we wait just the needed time, no more. This doesn’t happen with taskwait. Even though, it doesn’t seem to be happening and we can see pretty similar execution times between depend and taskwait usages.

Figure 17 shows execution times and overheads involved in the performance of the code. We can see there’s one thread that executed for a very short amount of time. Most of the time it had been waiting for others because the clause depend.



	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	2,003,241,537 ns	-	61,712,534 ns	7,983,967 ns	268,370 ns	2,425 ns
THREAD 1.1.2	957,559,201 ns	1,020,889,073 ns	20,772,080 ns	1,122,955 ns	218,342 ns	-
THREAD 1.1.3	937,864,364 ns	1,020,889,033 ns	38,427,102 ns	1,186,639 ns	222,823 ns	-
THREAD 1.1.4	937,921,992 ns	1,020,893,593 ns	40,672,631 ns	900,166 ns	194,872 ns	-
THREAD 1.1.5	913,740,401 ns	1,020,889,998 ns	65,348,150 ns	868,676 ns	199,218 ns	-
THREAD 1.1.6	34,050 ns	1,020,803,029 ns	981,195,768 ns	112,179 ns	5,470 ns	-
THREAD 1.1.7	755,016,190 ns	1,020,803,024 ns	224,979,873 ns	1,165,314 ns	160,109 ns	-
THREAD 1.1.8	855,603,357 ns	1,020,833,400 ns	118,539,065 ns	1,283,650 ns	187,111 ns	-
Total	7,360,981,092 ns	7,146,001,150 ns	1,551,647,203 ns	14,623,546 ns	1,456,315 ns	2,425 ns
Average	920,122,636.50 ns	1,020,857,307.14 ns	193,955,900.38 ns	1,827,943.25 ns	182,039.38 ns	2,425 ns
Maximum	2,003,241,537 ns	1,020,893,593 ns	981,195,768 ns	7,983,967 ns	268,370 ns	2,425 ns
Minimum	34,050 ns	1,020,803,024 ns	20,772,080 ns	112,179 ns	5,470 ns	2,425 ns
StDev	506,607,127.24 ns	39,397.76 ns	303,747,975.29 ns	2,352,260.93 ns	72,953.20 ns	0 ns
Avg/Max	0.46	1.00	0.20	0.23	0.68	1

Figure 17: Paraver trace for the execution of tree strategy using depend clause.

Optional

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]) {
   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;
      }
   }
}
```

Figure 18: Code for initialize function after parallelization

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

Figure 19: Code for parallelization on clear function

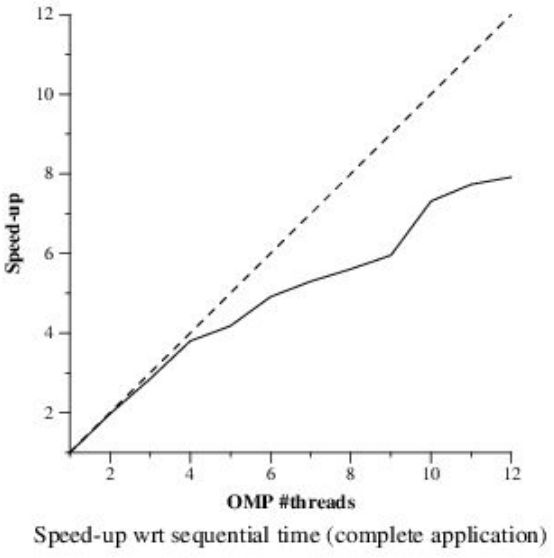


Figure 20: Strong scalability plot for the whole program after implementing tree strategy and parallelizing clear and initialize functions

In the previous figures we see the parallelization for the initialize and clear functions. In figure 20 we have the new speed-up plot regarding the complete application where we observe an upgrade to the graphic in Figure 12.
This improvement is caused by the addition of parallelization to initialize and clear functions.

Optional 2: Explore the best possible values for the sort size and merge size arguments used in the execution of the program. For that you can use the submit-depth-omp.sh script, modified to first explore the influence of one of the two arguments, select the best value for it, and then explore the other argument. Once you have these two values, modify the submit-strong-omp.sh script to obtain the new scalability plots.

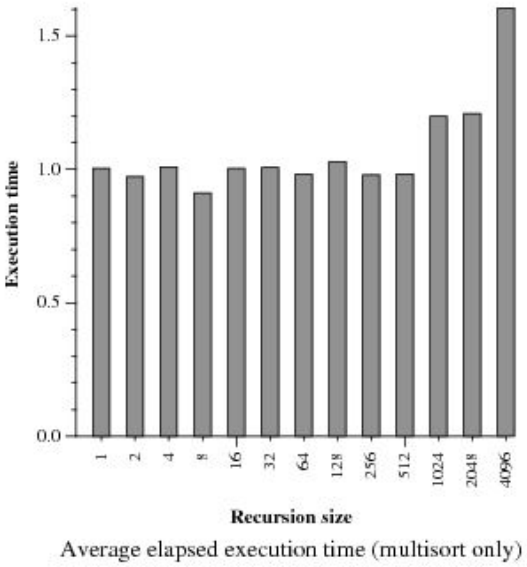


Figure 21: Execution time for every recursion size from 1 to 4096 for Merge size = 256, kilo-elements

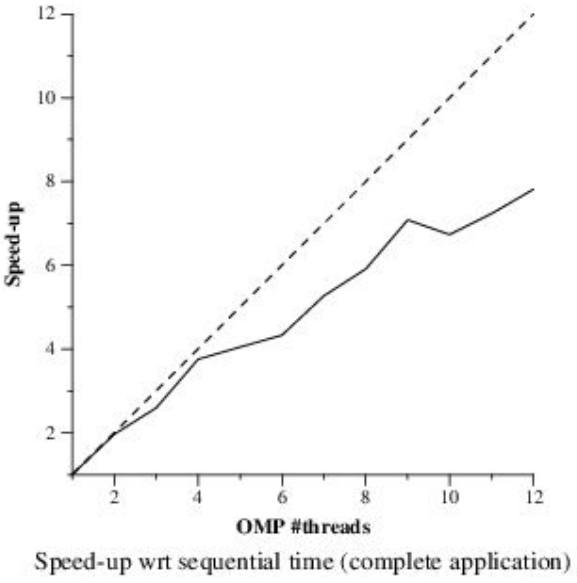


Figure 22: Speed-up plot for Recursion size 8 and Merge size 256, kilo-elements

Given all the plots obtained during this optional exercise (all of them attached in .zip file as bestValueForSortSize(MergeX), we got to the conclusion that the optimal combination is Recursion size = 8 and Merge size = 256 (both in kilo-elements, Figure 21)
With these numbers then we got the optimal speed-up plot (Figure 22)