Fourth Deliverable

Marlen Avila
Ricard Meyerhofer

4.1 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.
R/

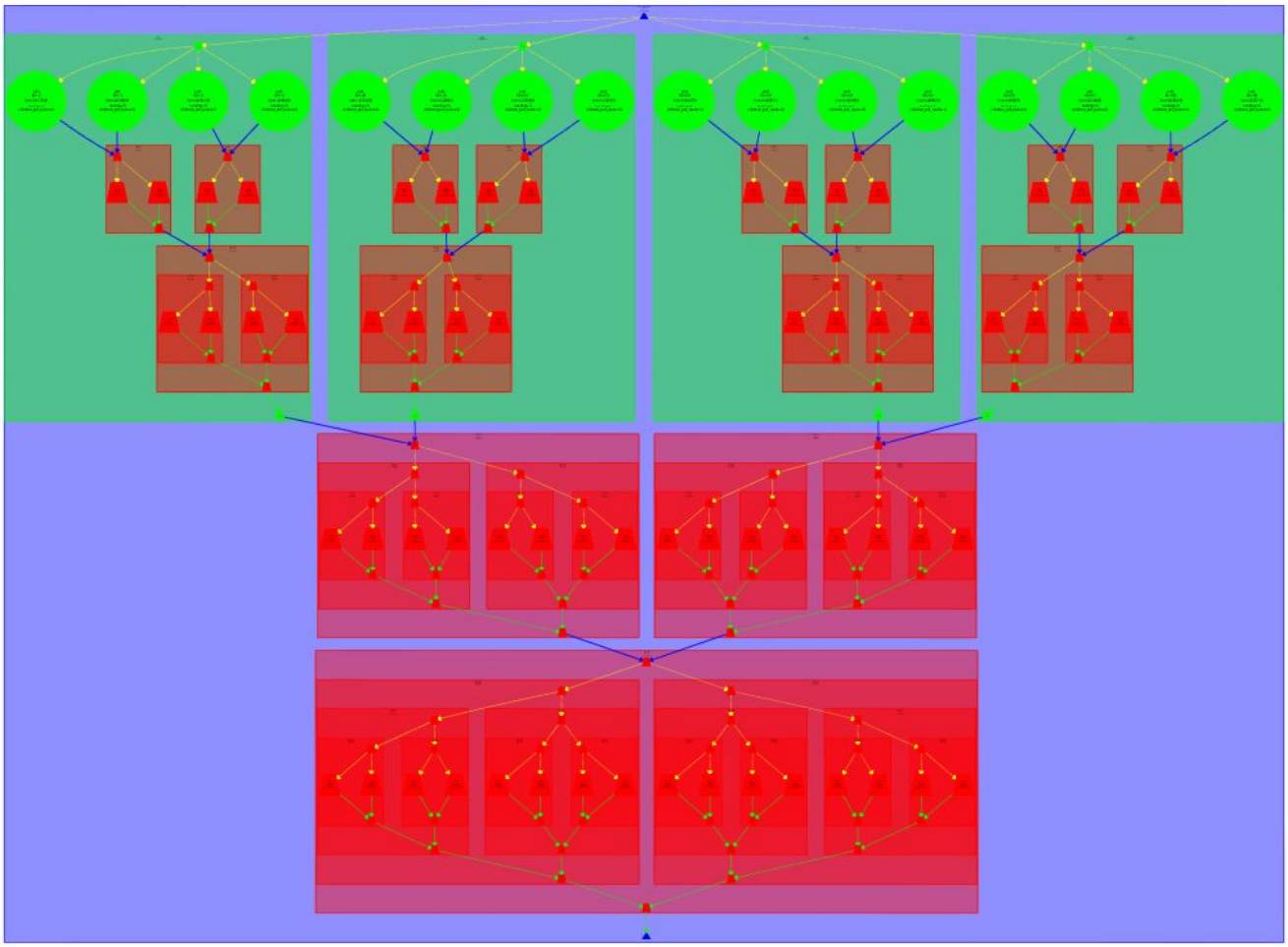
multisort-tareador.c

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

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

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

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

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

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

                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);
   }
}
```

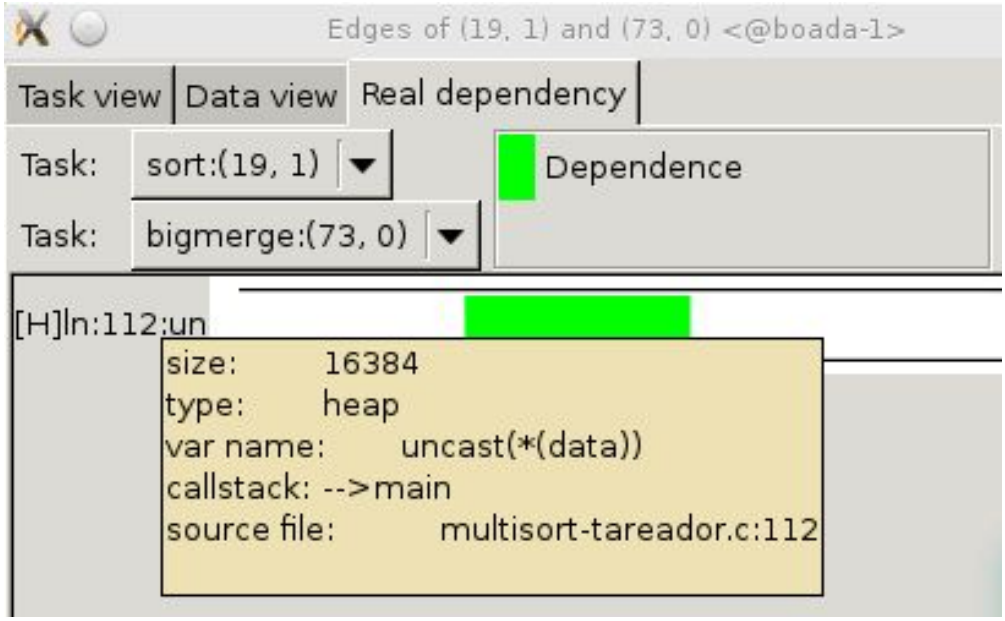
The clauses in **bold** are the ones we have added to instrumentate he code.

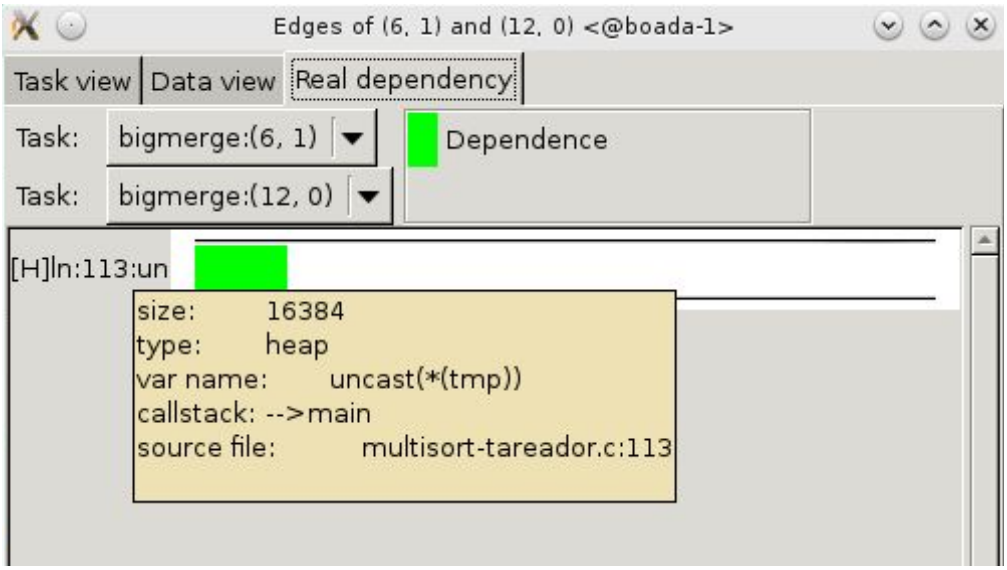
In function multisort, if the size of the vector to order is larger than the determined size it will make 4 recursive calls that will divide the vector in 4 parts (v1,v2,v3,v4). Then it will merge v1 and v2 creating v12, and it will do the same with v3 and v4. Finally, the latest call to function merge 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 (basicsort).
The two first merge calls (the ones above the 4 multisort recursive calls) will have to wait for the 4 multisort recursive calls 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).

The graph generated by executing the code in Tareador is the following:



Data dependences shown in the task dependence graph:





We have the following variable dependences, that are represented as blue arrows in the graph above:

- data (sort -> merge).
- tmp (merge -> merge)

In the table below, we've added the important parts of the code that show us the reason why these dependences exist:

Code dependences in multisort-tareador.c

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
    /*Code Merge*/
}
void multisort(long n, T data[n], T tmp[n]) {
   /*Code*/
   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);
   /*Code*/
}
```

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.
R/

	1	2	4
T exec (ns)	20334421001	10173741001	5086422001
SpeedUp	1	1.99	3.99

	8	16	32	64
T exec (ns)	2550549001	1289926001	1289926001	1289926001
SpeedUp	7.97	15.76	15.76	15.76

The result is close to the ideal Speed-Up until we reach 16 threads. Once we reach these 16 threads there is no improvement because 16 is 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.

4.2 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.
R/

multisort-omp.c (Leaf)

```
/*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
         #pragma omp task
                basicmerge(n, left, right, result, start, length);
        } else {/*Code*/}
}

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*/

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

```
}
```

multisort-omp.c (Tree)

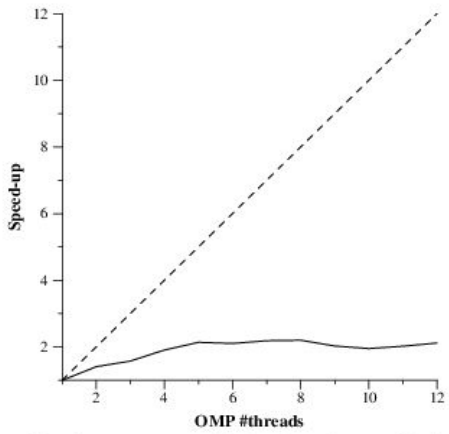
```
/*Code*/

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 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) {
         #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);
   }
}
```

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

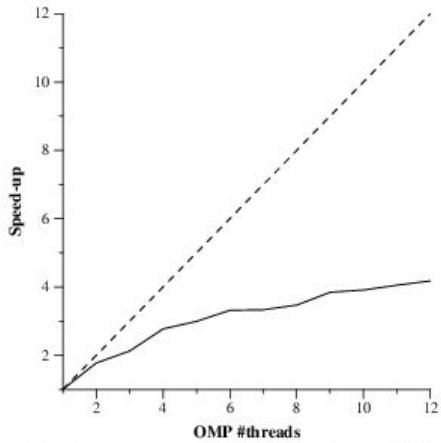
2. For the the Leaf and Tree strategies, include the speed–up (strong scalability) plots that have been obtained for the different numbers of processors. Reason about the performance that is observed, including captures of Paraver windows to justify your explanations.

R/

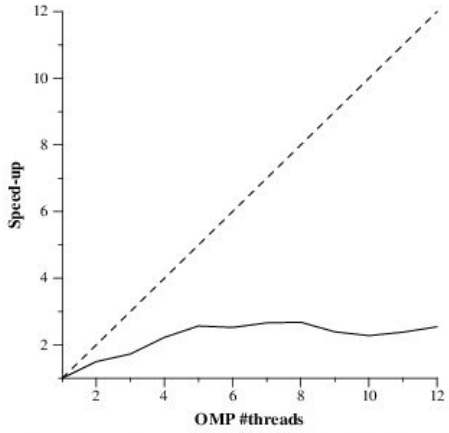
Leaf Strategy	Tree Strategy



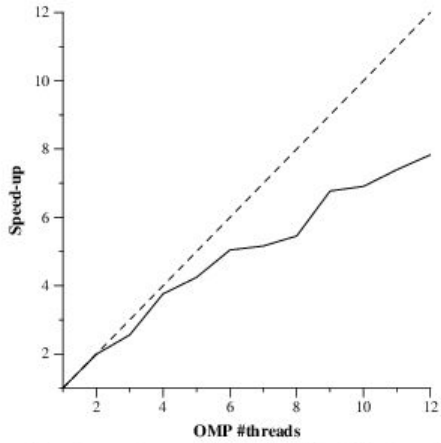
Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)

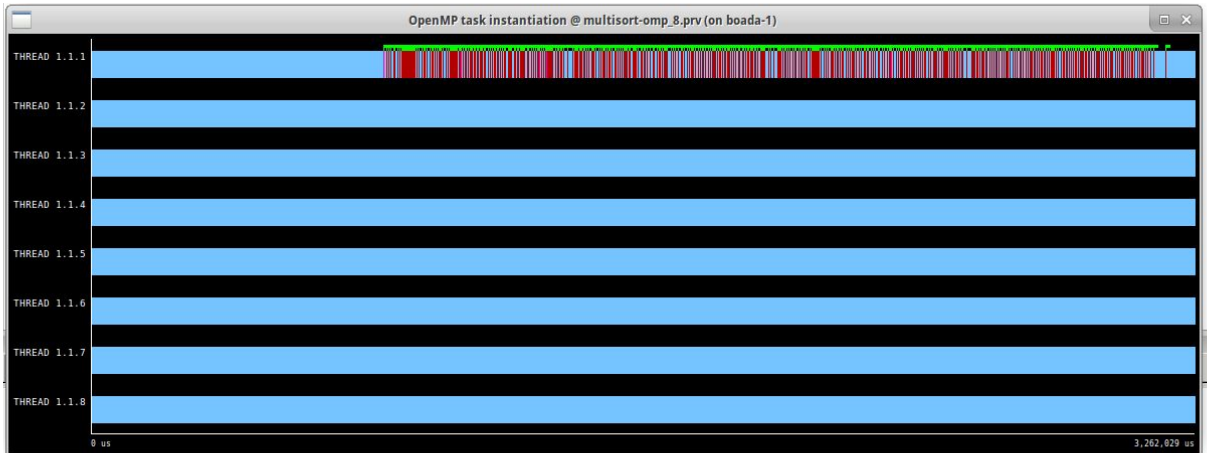


Speed-up wrt sequential time (multisort funtion only)

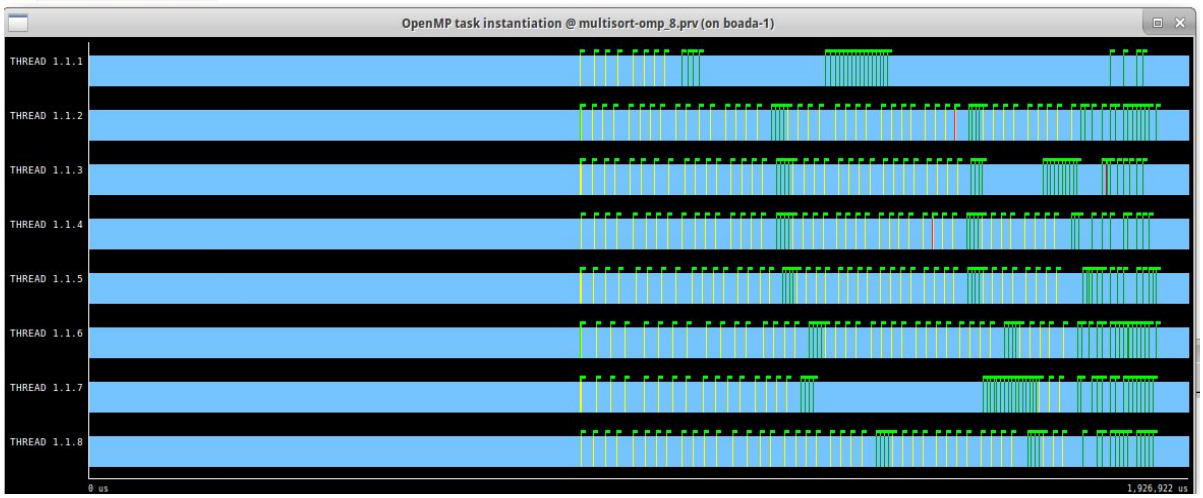
The speed-up of the leaf version is lower than the tree version one. That's because in the first strategy we have just one thread creating tasks by the time it reaches the base case (this way, that thread won't be able to create a new task until the previous one is finished, so the tasks generation will be done sequentially). In the second strategy we create a task every time we do a recursive call (that means all 8 threads are generating tasks at the same time).
Because of that, the execution time in the tree version is smaller than the leaf version one.

Tasks generation paraver captures:

In the following traces we can see the task generation in both versions and the differences described above. We've used the **task_instantiation** filter:



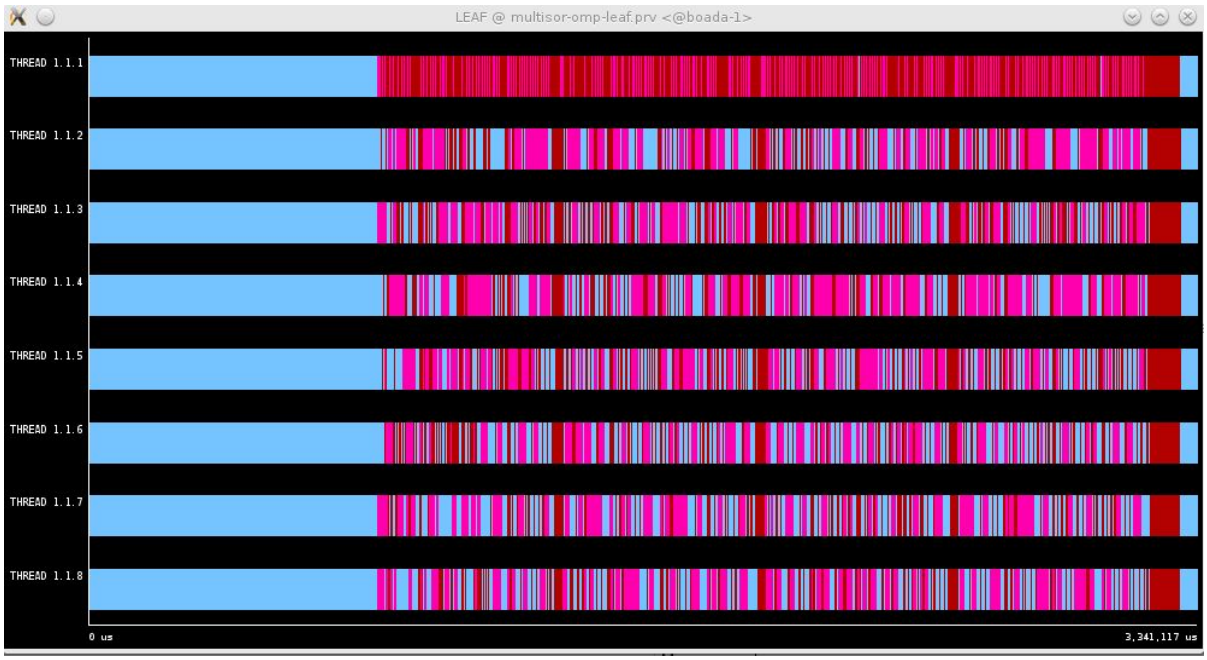
Leaf version task generation



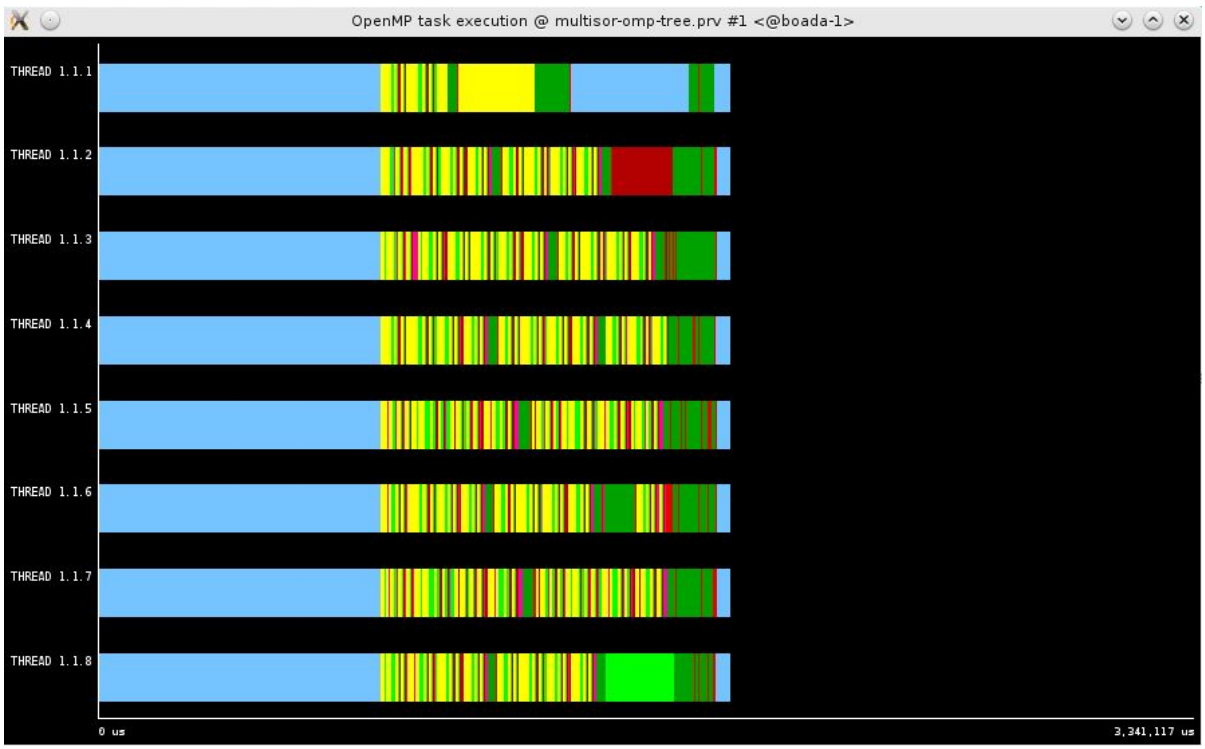
Tree version task generation

Task execution paraver captures:

Comparing the two following traces, we can see that the tree strategy is a lot faster than the leaf one (we've normalized the time so that both traces have the same time scale. This way is easier to visualize the difference). The filter used this time is the **task_execution** one:



Leaf version task execution

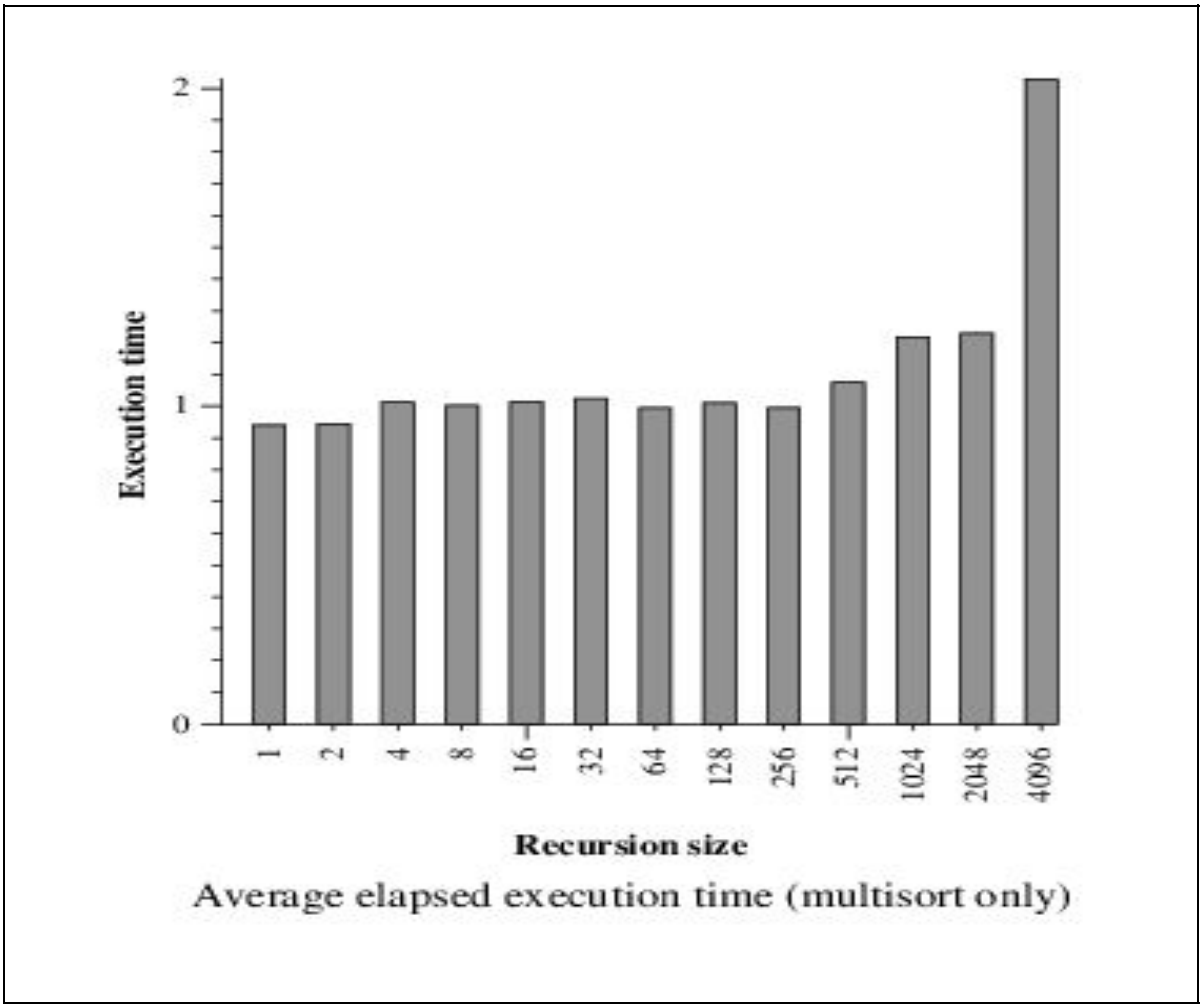


Tree version task execution

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?

R/

Depth in Tree Version 8



As we can see in the execution plot at the biggest values (1024,2048,4096) the execution time also increases. Even though we see these relation time-size, we cannot guarantee that as smaller the size is, the faster will be executed because for example, the execution time of 128 is bigger than the 256 execution time and the same happens with other recursion sizes like 32 and 256 (when 256 is 8 times bigger) so 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 depth with a higher value, we lose potential parallelism (but we also get less overhead because there will be created less tasks which makes it difficult to determine which recursion size is the optimal).

4.3 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.
R/

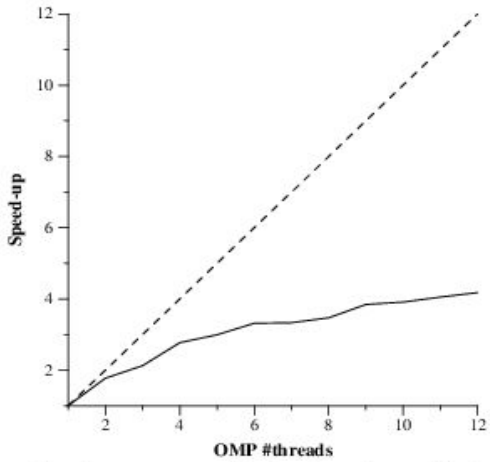
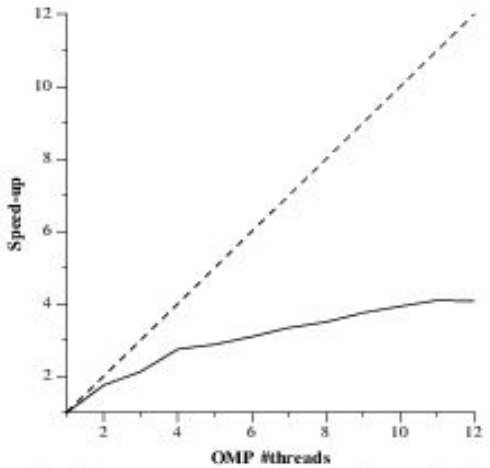
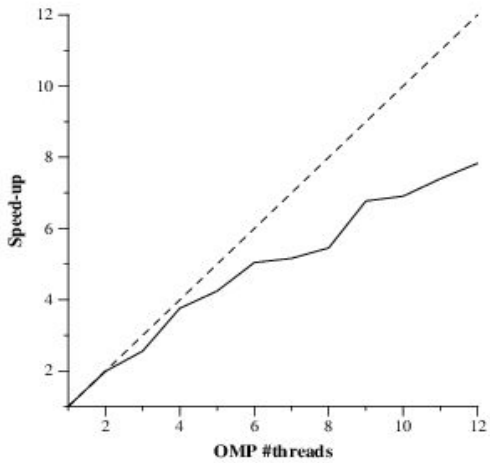
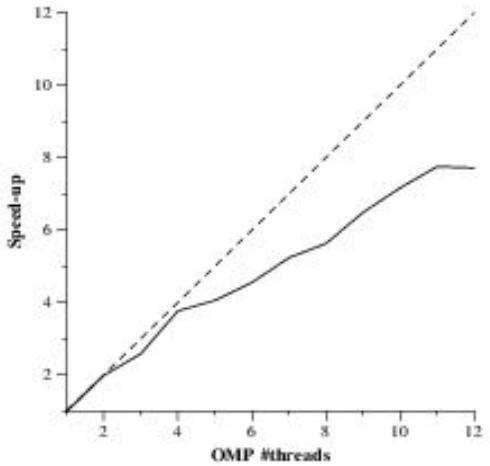
multisort-omp.c with dependent tasks

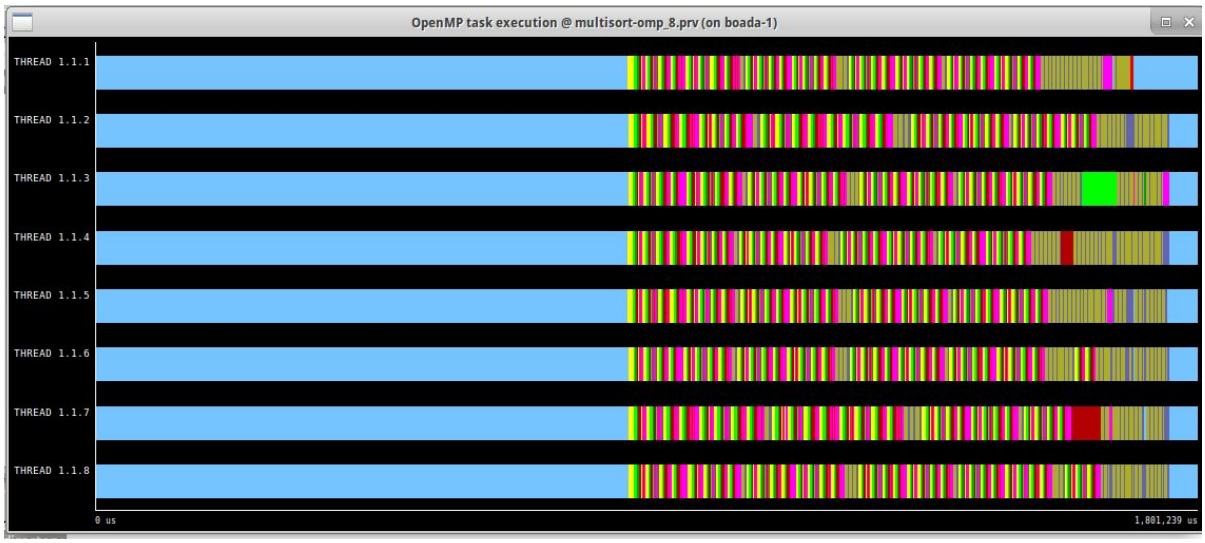
```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {

        if (length < MIN_MERGE_SIZE*2L) {
                // Base case
         //#pragma omp task
                basicmerge(n, left, right, result, start, length);
        } else {
                // Recursive decomposition
         #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
         //#pragma omp task
         basicsort(n, data);
   }
}
```

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 isn't necessary to change the previous pragmas because there are no dependences between them.

We've added for the code correctness, the taskwaits at the end to wait all the tasks of a 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.
R/

Tree Strategy	Dependent tasks
 Speed-up wrt sequential time (complete application)	 Speed-up wrt sequential time (complete application)
 Speed-up wrt sequential time (multisort funtion only)	 Speed-up wrt sequential time (multisort funtion only)



We can see that **there's no significative gain between both implementations.** Even though the version with pragma dependence 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 (for the same reason than in the tree implementation) because we have applied a thinner granularity to our data dependences.

4.4 Optional

Optional 1: Complete the parallelization of the Tree version by parallelizing the two functions that initialize the data and tmp vectors . 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.

R/

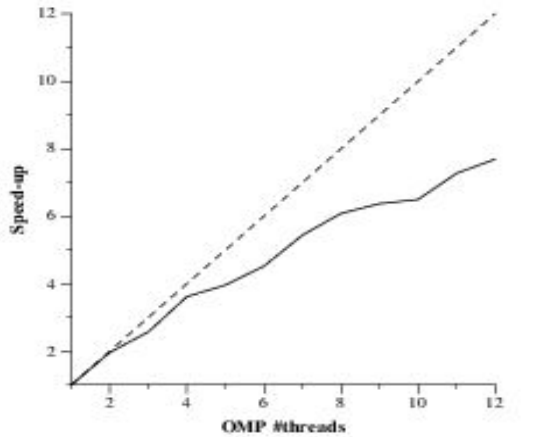
optional 1 code

```
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;
      }
   }
   }
}
```

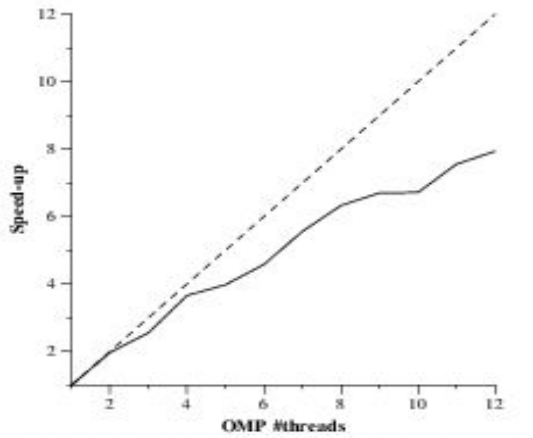
```
static void clear (long length, T data[length]) {
   long i;
   #pragma omp parallel for
   for (i = 0; i < length; ++i) {
      data[i] = 0;
   }
}
```

Clauses in **bold** have been added.

Scalability plot



Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)

As we can see the global speed-up has increased and we can see also in the paraver trace that the execution time is lower than in the tree execution (what it was quite obvious).

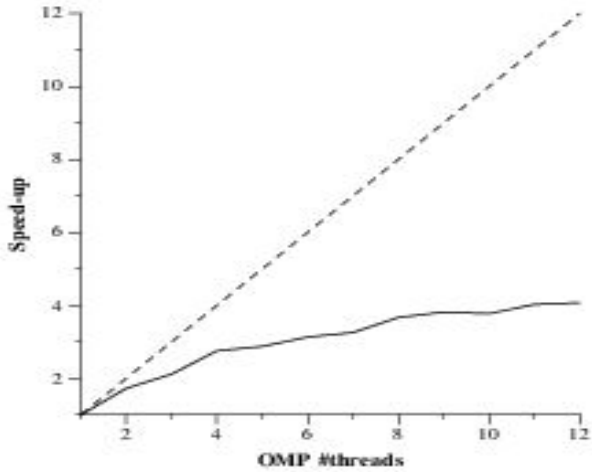
<u>Optional</u> <u>2:</u> Explore the best possible values for the sort size and merge size arguments used in the execution of the program. For that you can use the submit-depth-omp.sh script, modified to first explore the influence of one of the two arguments, select the best value for it, and then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

R/ The best values are:

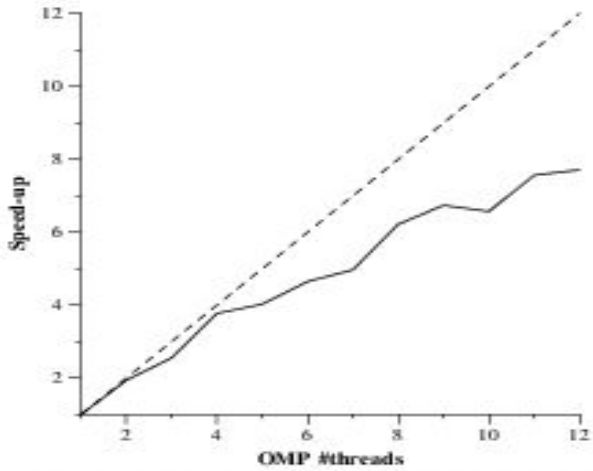
- **Sort_size =** 8
- **Merge_size =** 256

(Scalability plot on the next page)

Scalability plot



Speed-up wrt sequential time (complete application)



Speed-up wrt sequential time (multisort funtion only)