PAR Laboratory Assignment

Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Alvaro Moreno Ribot, **PAR 2214**
Albert Escoté Alvarez, **PAR 2205**

Recursive task decompositions: *Leaf vs Tree* strategies

In this assignment we will be looking at two different strategies for task decomposition. The first one we will look at is the one we call *leaf recursive task decomposition.* In this kind of task decomposition a task is executing **one** or **more leafs**. On the other hand, with *tree recursive task decomposition* each time a **recursive call** takes place a new task is created; in the example we can see in the assignment document, we will have that every recursive task will create two more tasks (as there are two calls to the recursive function).

Questions:

- *Is there a big difference in how the tasks are generated?*
- *Which is this difference? In other words, after how many steps is the last task generated in each strategy?*

Yes, the big difference resides in 1) the final number of tasks and 2) when the tasks are created. As we said above, with *leaf task decomposition* we will have tasks that compute **one** or **more** leaves, and with the *tree task decomposition* we will have a task for each recursive call.

If we take a look at the given example (Divide and conquer with dot product) we can see that, for the *leaf* decomposition strategy the last (and first) task is generated with the leaves, this means that for the first task to be generated we will need to complete all the recursive calls and then the task generation will begin; in the *tree* strategy we can note that the first task is generated right after the first recursive call takes place. Then, the last task will be generated with the leaves.

- *Is the granularity for tasks executing invocations to dotproduct the same in both strategies?*

It depends, we could control the number of leaves that a particular task executes so we could change the granularity. But yes, we can reach the same granularity in both strategies.

- *How could you control the number of leaves a task reaches (executes)?*

If we want to control the number of leaves that a task executes, we can

Analysis with Tareador

In this section we are asked to analyze the task decomposition for mergesort. First of all, we take a look at the code of the program multisort.c and try to understand how the "divide and conquer" is implemented. We compile it, as told, and execute it with the command **./multisort-seq -n 32768 -s 1024 -m 1024**. The results obtained are the following:

```
par2205@boada-1:~/lab4$ ./multisort-seq -n 32768 -s 1024 -m 1024
*******************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
*******************************************************************************
Initialization time in seconds: 0.852912
Multisort execution time: 6.240794
Check sorted data execution time: 0.015254
Multisort program finished
*******************************************************************************
```

We are gonna save these results to analyse them afterwards.

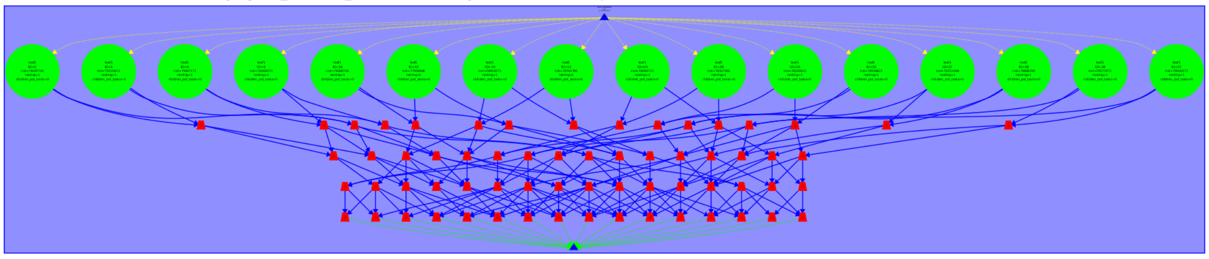
Now, we are going to investigate different potential task decomposition strategies and their implications in terms of parallelism and task interactions required using the Tareador tool. As we could see in the first point of this document, we have two recursive task strategies: leaf and tree strategy.

Leaf strategy

So we are going to start with the leaf strategy. We add the task creation in the base cases of the recursion functions to create a task for every leaf of the recursion tree. This is achieved by adding the clausule **tareador_start_task("X")** and **tareador_end_task("X")** on the basicmerge and basicsort calls, being **X** the name of the task.

```
void basicsort(long n, T data[n]);

void basicmerge(long n, T left[n], T right[n], T result[n*2], long start, long length);

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("leaf2");
        basicmerge(n, left, right, result, start, length);
        tareador_start_task("leaf2");
    } 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]);

        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("leaf1");
        basicsort(n, data);
        tareador_start_task("leaf1");
    }
}
```

After compiling this code, we execute the binary generated with the **run-tareador.sh** script and we obtain the following graph dependence generated by the Tareador tool.



The green tasks are the **leaf1** in the code, and the red tasks are the **leaf2** in the code. As we can see, we have 16 tasks for the basicsort function call which are totally parallelizable. Then, we have 64 tasks for the basicmerge function call which have 4 levels of parallelisation, each one with 16 tasks that depend from the previous 16 tasks.

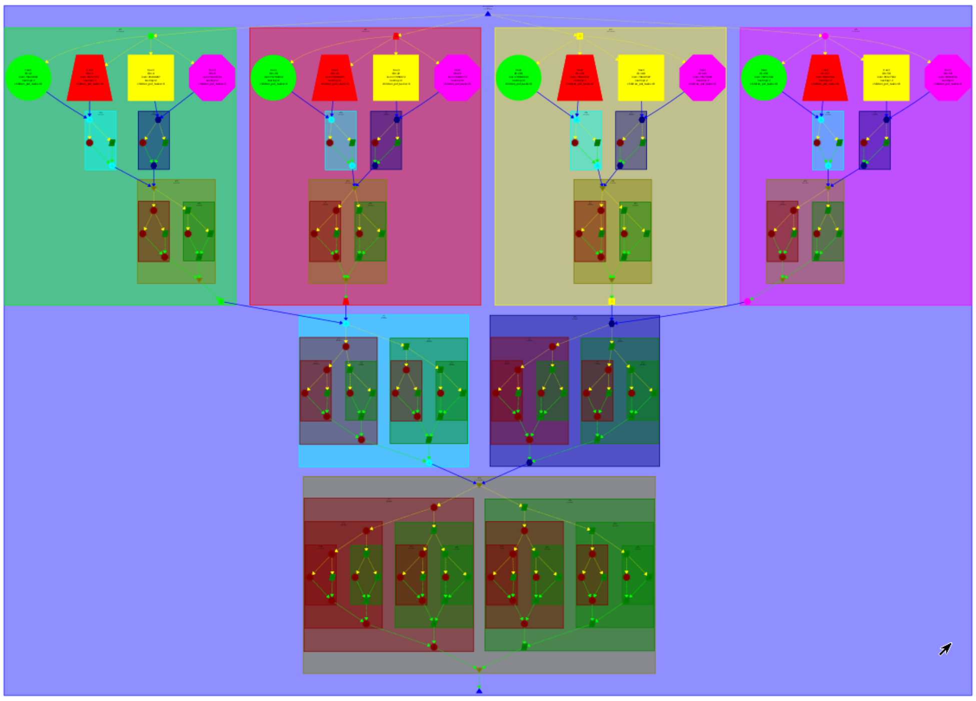
Tree strategy

After analysing the leaf strategy, we are now going to analyse the tree strategy. We add the task creation before every recursive call to make a task for every node of the recursion tree. This is achieved by adding the clausule **tareador_start_task("X")** and **tareador_end_task("X")** on the multisort and merge calls, being **X** the name of the task.

```
void basicsort(long n, T data[n]);

void basicmerge(long n, T left[n], T right[n], T result[n*2], long start, long length);

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
       tareador_start_task("tree8");
       merge(n, left, right, result, start, length/2);
       tareador_end_task("tree8");

       tareador_start_task("tree9");
       merge(n, left, right, result, start + length/2, length/2);
       tareador_end_task("tree9");
       }
}

void multisort(long n, T data[n], T tmp[n]) {
       if (n >= MIN_SORT_SIZE*4L) {
       // Recursive decomposition
       tareador_start_task("tree1");
       multisort(n/4L, &data[0], &tmp[0]);
       tareador_end_task("tree1");
```

```
        tareador_start_task("tree2");
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        tareador_end_task("tree2");

        tareador_start_task("tree3");
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        tareador_end_task("tree3");

        tareador_start_task("tree4");
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
        tareador_end_task("tree4");

        tareador_start_task("tree5");
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        tareador_end_task("tree5");

        tareador_start_task("tree6");
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
        tareador_end_task("tree6");

        tareador_start_task("tree7");
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
        tareador_end_task("tree7");
        } else {
        // Base case
        basicsort(n, data);
        }
}
```

After compiling this code, we execute the binary generated with the **run-tareador.sh** script and we obtain the following graph dependence generated by the Tareador tool.



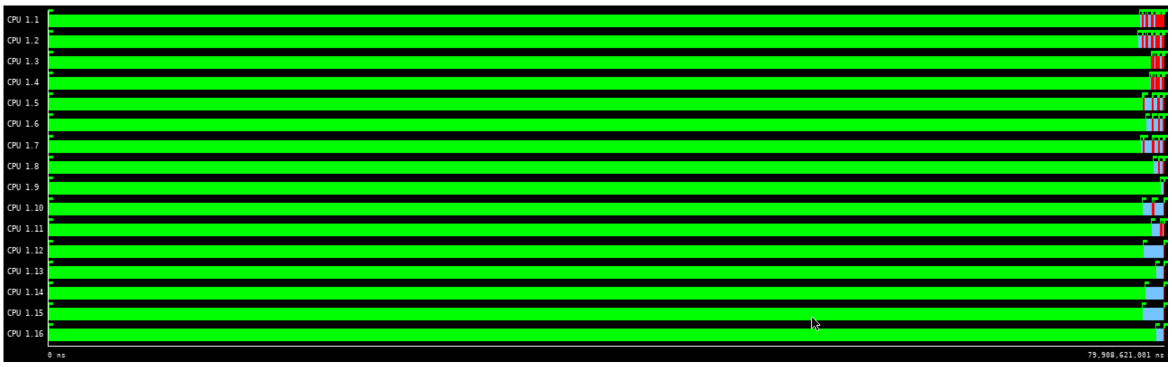
To obtain the table asked we modify the given code and we add some lines to count automatically the number of internal tasks and tasks doing computation per level of recursion and strategy. We add a variable called **depth** in the merge and multisort functions and when the code realizes an internal task or a computation task we increase in one the value stored in a vector. After executing the program we obtain the following results:

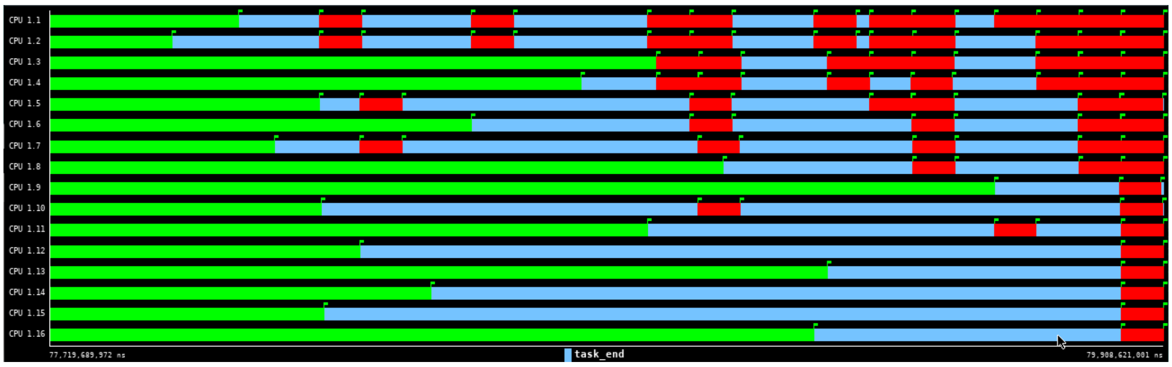
Level of recursion	Leaf strategy		Tree strategy	
	basicsort	basicmerge	multisort	merge
0	0	0	4	9
1	0	0	16	48
2	16	16	0	40
3	0	32	0	16
4	0	16	0	0

In the leaf decomposition strategy dependencies are generated by the fact that the basicmerge function needs both left and right arrays to be sorted previously by the basicsort function. This could be fixed by adding a **taskgroup** (which has an implicit **taskwait** at the end) that contains all the recursive calls of multisort and one that contains the two first merge calls.

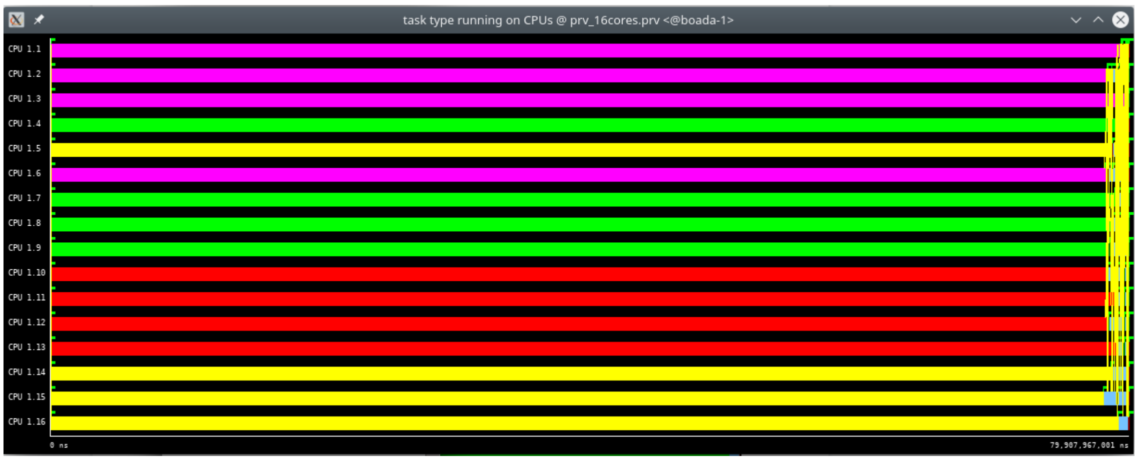
In the tree decomposition task we find the same dependencies as every recursive call to merge needs its array parameters to be sorted before. This can be seen clearly on the task graph decomposition generated by Tareador. We can fix this dependency by putting the same taskgroups as in leaf strategy, and generating all tasks with the implicit clauses.

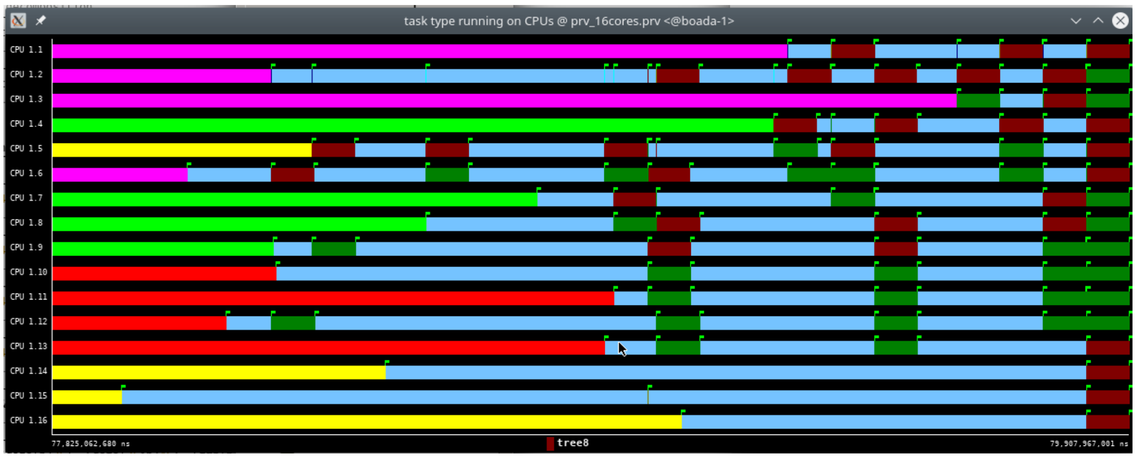
Now, we have a look at the trace timeline for both strategies. For the leaf strategy we can observe that tasks doing computation have a very different lifetime. The basicsort tasks spends most of the time and basicmerge only spends a few quantity of time. The two following timelines are the same, being the second one an expanded version of the first one.





Here we take a look at the tree strategy timeline. The tasks represented in pink, green, yellow and red represent the multisort function calls. As we can see these tasks last the most of the timeline. Then we have the merge call functions that have a practically negligible time.





Shared-memory parallelisation with OpenMP tasks

Now we will apply all we learned from the previous analysis to parallelize the original sequential code in **multisort.c**. We will be exploring the same two different versions as before (*leaf* and *tree*).

Leaf strategy

```
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) {
        #pragma omp taskgroup
        {
            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 taskgroup
        {
            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
      #pragma omp task
       basicsort(n, data);
    }
}
```

*We also added the `#pragma omp parallel` and `#pragma omp single` directives right before the **multisort** invocation in the ***main***.

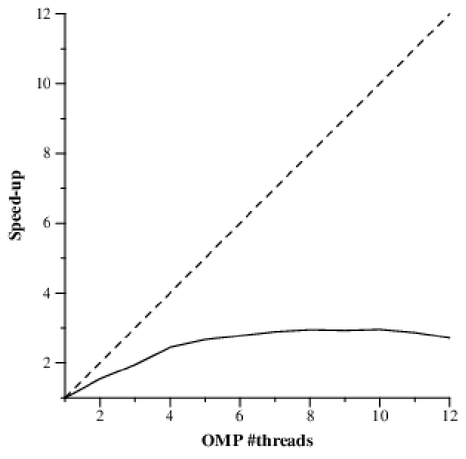
To achieve this recursion strategy we placed a ***omp task*** for each non-recursive call (leaf) of the code (*basicsort* and *basicmerge*); as a way of synchronization we used ***taskgroup*** directives (to avoid dataraces).

In this code a thread is generating all the tasks, also we will have a thread executing a task for each leaf. Note that parallel tasks are only being created when we reach the leaves of the recursion. This means that we will have many sequential parts until we reach the leaves in the recursion tree.

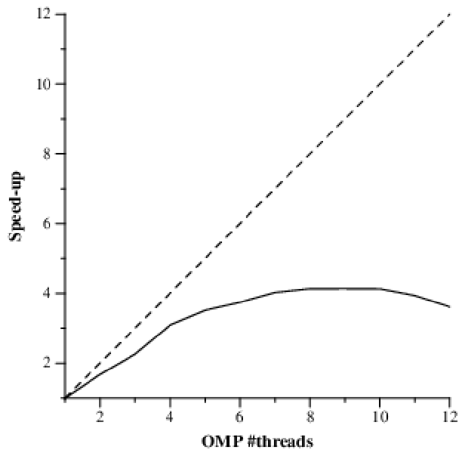
We sent this code for batch execution (with 2 threads) and we got the following output:

```
::::::::::::::
multisort-omp_2_boada-2.times.txt
::::::::::::::
************************************************************************
*
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024,
MIN_MERGE_SIZE=1024
Cut-off level:                        CUTOFF=16
Number of threads in OpenMP:          OMP_NUM_THREADS=2
************************************************************************
*
Initialization time in seconds: 0.854992
Multisort execution time: 3.732622
Check sorted data execution time: 0.016331
Multisort program finished
************************************************************************
*
```

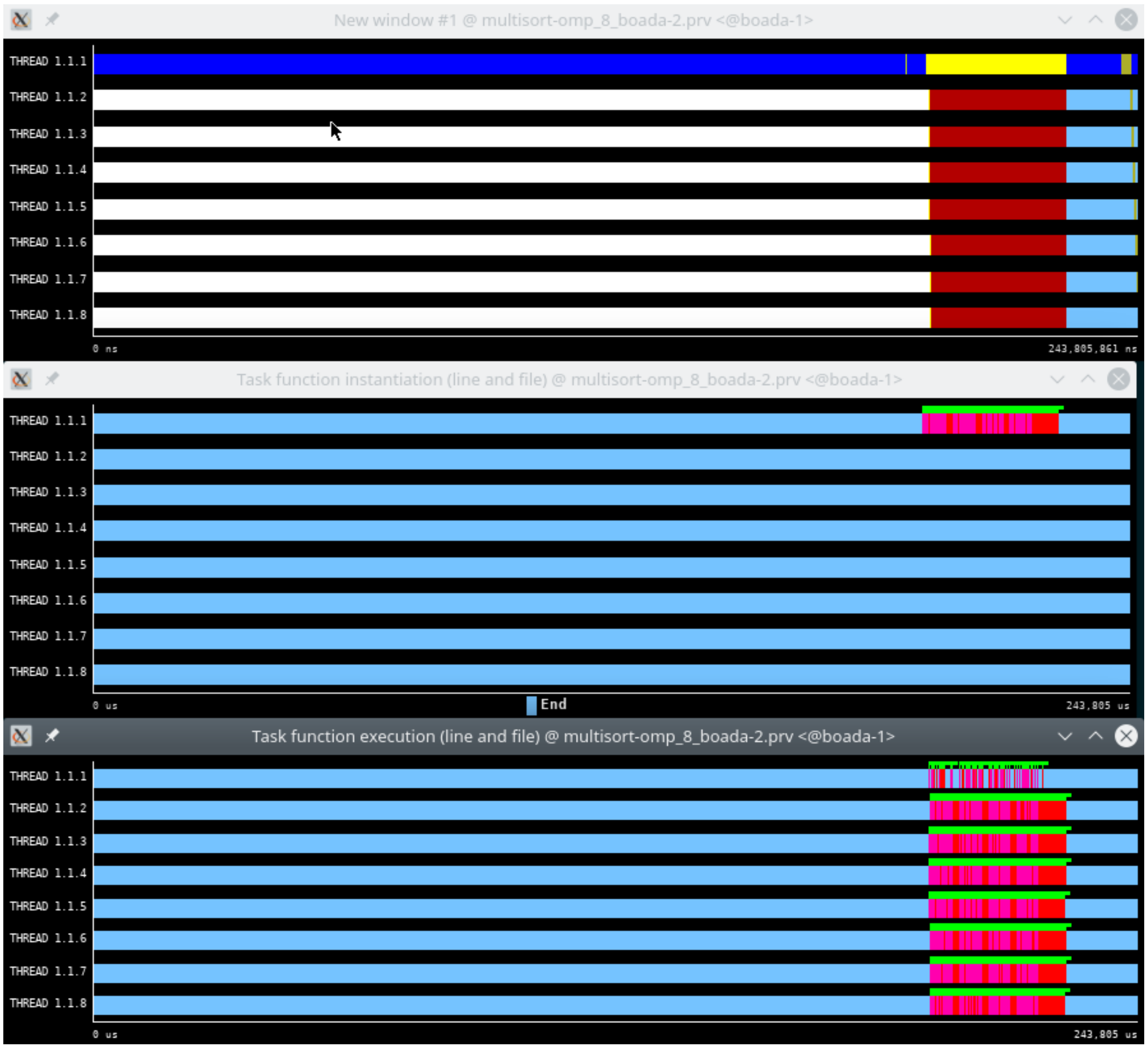
This plot we have on the right shows the speed-up for the execution of the above code for a different number of threads (from 1 to 12). As we can see, the speed-up is not as good as we would expect from a parallel code; this is because of the issue that we mentioned earlier: the path to the leaves of the recursion tree is sequential so we spend a considerable amount of time executing sequential code until we reach the leaves.



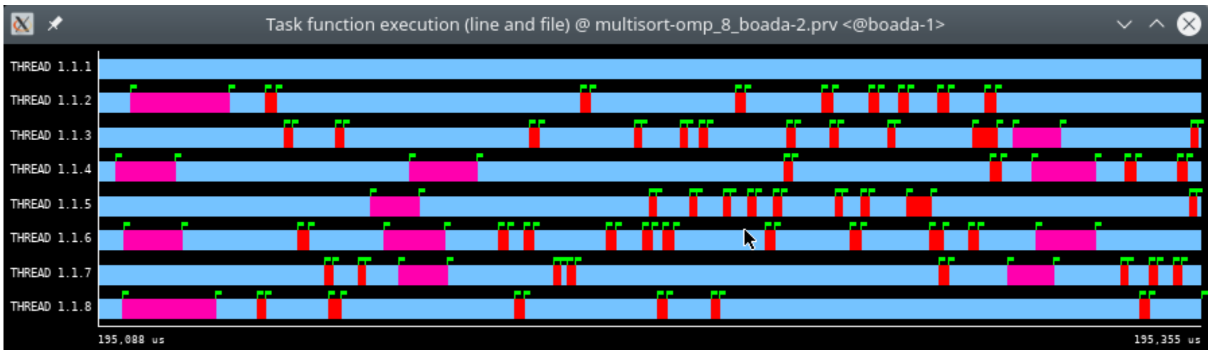
par2214
Speed-up wrt sequential time (complete application)
Thu May 6 13:06:08 CEST 2021



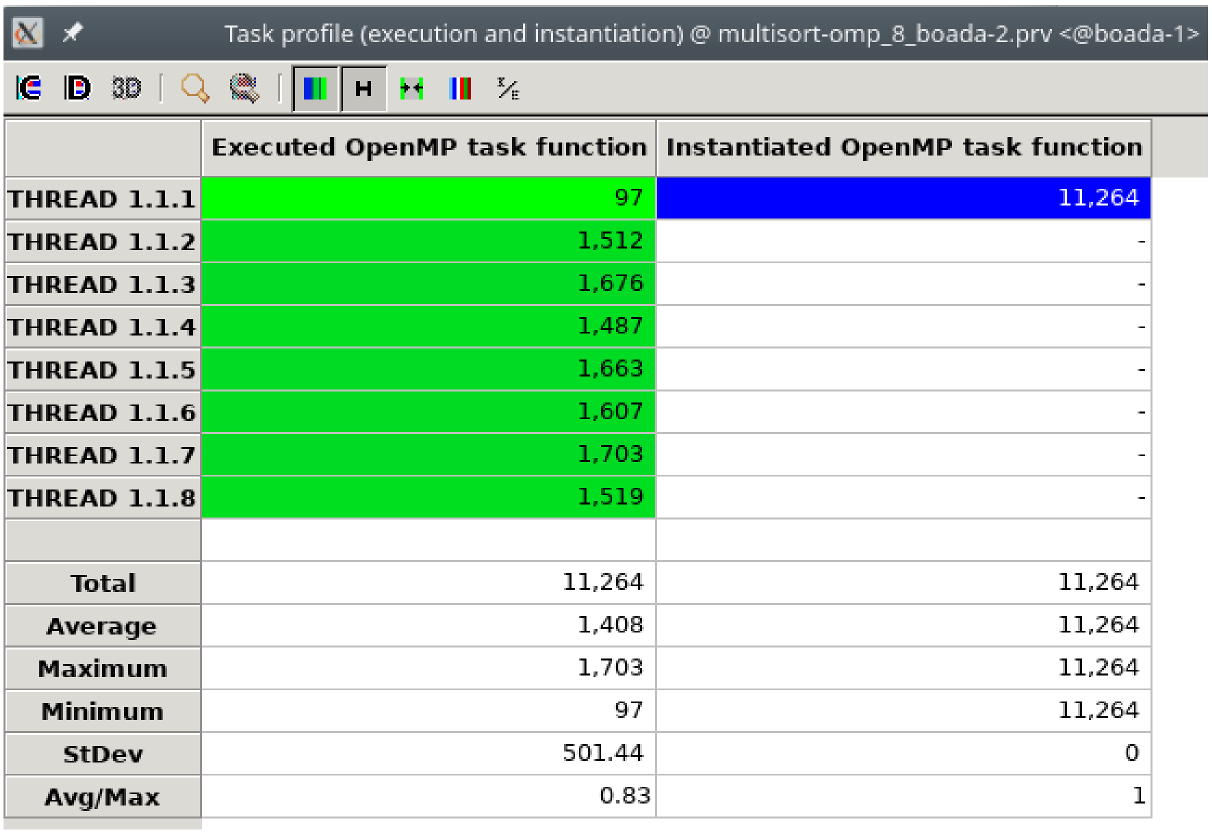
par2214


New window #1 @ multisort-omp_8_boada-2.prv <@boada-1>
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 ns
243,805,861 ns
Task function instantiation (line and file) @ multisort-omp_8_boada-2.prv <@boada-1>
0 us
End
243,805 us
Task function execution (line and file) @ multisort-omp_8_boada-2.prv <@boada-1>
0 us
243,805 us

Here we can clearly see that the parallel execution does not take place until we reach the leaves (where it starts creating tasks) and that all the other execution is sequential. We can see that thread 1 is traversing the recursion tree until it reaches the leaves and then other threads start executing.



In this image we can see the very same timeline as the last image from the three we have seen above but zoomed in so we can see more clearly what is happening. As we can see, the pink corresponds to the ***merge*** and the red to the ***quicksort***. Note that ***merge*** takes a considerable bigger amount of time compared to ***quicksort***. Also we can remark that at any given moment there are only 4 threads either executing the sort or the merge, but we can never take advantage of all 8 threads. This is because the vector is divided into 4 parts which are sorted and later merged and as they have dependencies (parts need to be sorted before merging) we can't have more than 4 threads working at a given time.



Task profile (execution and instantiation) @ multisort-omp_8_boada-2.prv <@boada-1>

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	97	11,264
THREAD 1.1.2	1,512	-
THREAD 1.1.3	1,676	-
THREAD 1.1.4	1,487	-
THREAD 1.1.5	1,663	-
THREAD 1.1.6	1,607	-
THREAD 1.1.7	1,703	-
THREAD 1.1.8	1,519	-
Total	11,264	11,264
Average	1,408	11,264
Maximum	1,703	11,264
Minimum	97	11,264
StDev	501.44	0
Avg/Max	0.83	1

Tree Strategy

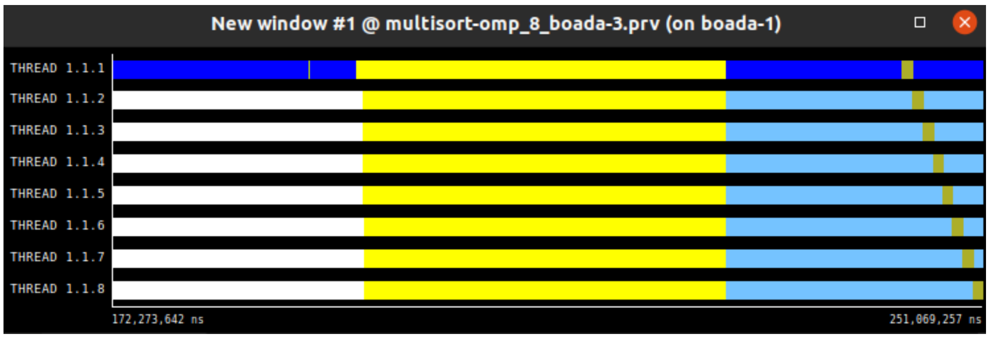
This time we will be creating a task for each level in the recursion tree; this will allow us to have more parallelism and a better speed-up (we think).

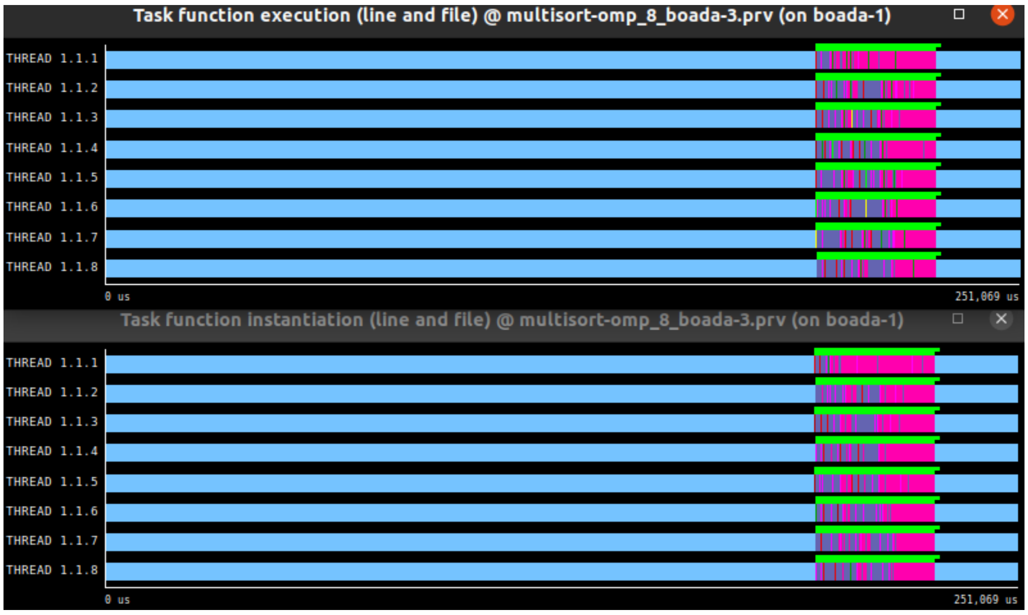
To do this we will use a **#pragma omp task** before each recursive call in each of the recursive functions. In order to do the synchronization and avoid data races we will be using a **taskgroup** directive (as we did in *leaf* strategy):

```
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);
    }
}
```

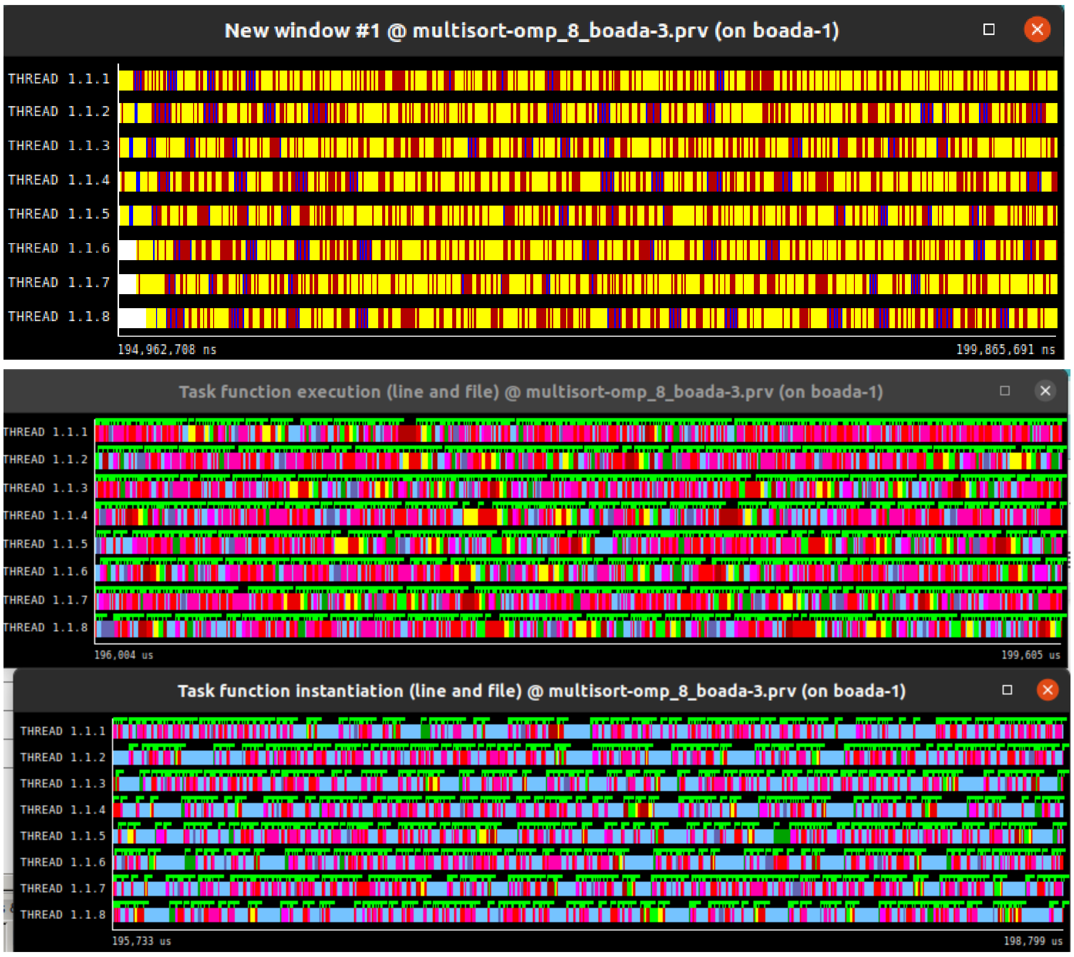
```
void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
       #pragma omp taskgroup
       {
               #pragma omp task
               multisort(n/4L, &data[0], &tmp[0]);
               #pragma omp task
               multisort(n/4L, &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);
       }
       #pragma omp task
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

This time we will have more tasks as we will have one different call for each call to the recursion this way we will be able to have more threads working at the same time, for this the speed-up should be better than the *Leaf Strategy* but we also think that creating a task for each recursive call will create a lot of overhead.

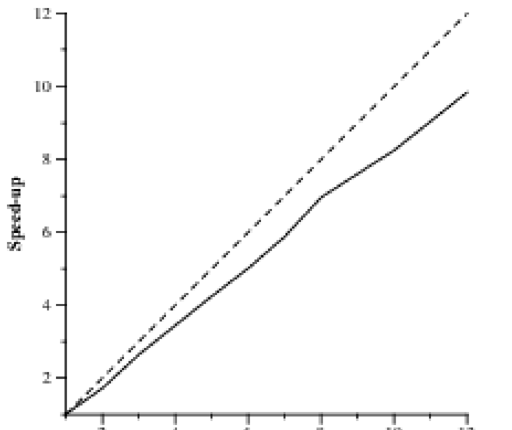




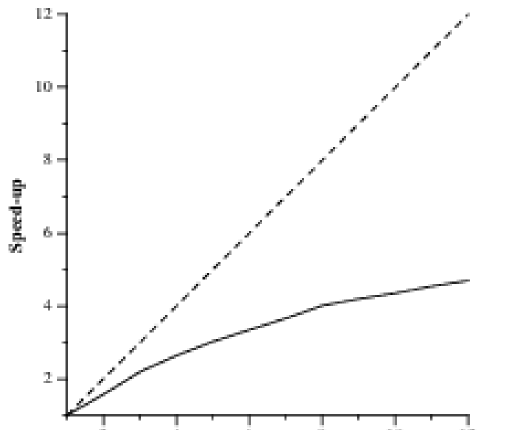
Here we can see that we don't have just one thread creating tasks, instead we have all of them creating tasks. We now can take a look at the next traces which show how all threads are busy creating tasks (task instantiation zoomed in):



Note that now all threads are busy creating and executing tasks; this is not the best strategy we can see that a lot of threads' busyness is caused by the overhead of the task creation, so adding a cut-off should improve this performance.



par2214
Speed-up wrt sequential time (multisort funtion only)
Thu May 13 08:28:32 CEST 2021



par2214
Speed-up wrt sequential time (complete application)
Thu May 13 08:28:32 CEST 2021

As we can see, checking the above scalability plots, we can say that the more threads we add the better the performance gets. We will have a lot of non-dependent tasks that will be taking advantage of the threads.

Tree strategy with a cut-off mechanism

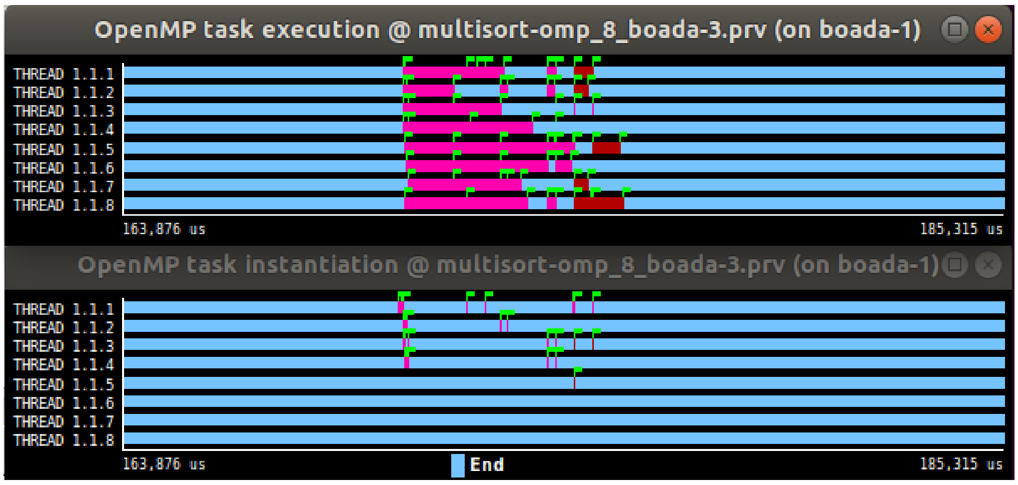
At this point we are told to add a mechanism called *cut-off* which will allow us to control the maximum recursion level for task generation. We make some changes in the tree strategy implemented before and we obtain the following code:

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length,int
depth){
    if (length < MIN_MERGE_SIZE*2L){
           //BASE case
        basicmerge(n,left,right,result,start,length);
    } else {
           //Recursive decomposition
           #pragma omp task final(depth >= CUTOFF)
           merge(n,left,right,result,start,length/2,depth+1);
           #pragma omp task final(depth >= CUTOFF)
           merge(n,left,right,result,start+length/2, length/2,depth+1);
    }
}

void multisort(long n, T data[n], T tmp[n],int depth){
    if(n >= MIN_SORT_SIZE*4L){
    //RECURSIVE decomposition
```

```
            #pragma omp taskgroup
            {
               #pragma omp task final(depth >= CUTOFF)
               multisort(n/4L, &data[0], &tmp[0],depth+1);
               #pragma omp task final(depth >= CUTOFF)
               multisort(n/4L, &data[n/4L], &tmp[n/4L],depth+1);
               #pragma omp task final(depth >= CUTOFF)
               multisort(n/4L,&data[n/2L], &tmp[n/2L],depth+1);
               #pragma omp task final(depth >= CUTOFF)
               multisort(n/4L,&data[3L*n/4L], &tmp[3L*n/4L],depth+1);
            }
            #pragma omp taskgroup
            {
               #pragma omp task final(depth >= CUTOFF)
               merge(n/4L, &data[0], &data[n/4L], &tmp[0],0,n/2L,depth+1);
               #pragma omp task final(depth >= CUTOFF)
               merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L],0,n/2L,depth+1);
            }
            #pragma omp task final(depth >= CUTOFF)
            merge(n/2L, &tmp[0], &tmp[n/2L],&data[0],0,n,depth+1);
    }else{
     //Base case
         basicsort(n,data);
     }
 }
```

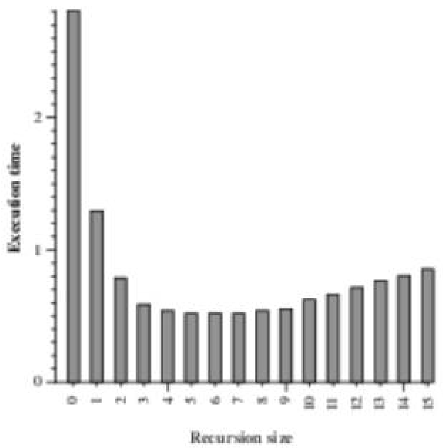
Now we follow a tree task creation scheme just as before but, at a certain depth in the recursion, the tasks stop being created and the rest is executed sequentially.

This strategy should be better than the ones we tried before, we will still have a lot of parallelization but we will reduce the overhead of the tasks creation in the levels near the leaves, where it will start being sequential.



This trace above is the one we got when using CUTOFF=1. We can now see how different threads that instantiate tasks, so it means that in the second level of recursion tasks are also created by various threads. Also note that a very little number of tasks are being created.

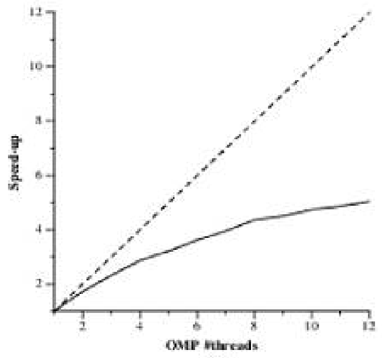
8 Threads



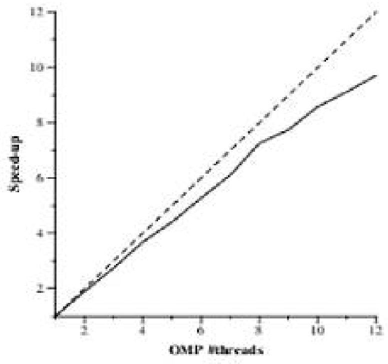
par2214
Average elapsed execution time (multisort only)
Thu May 13 18:27:05CEST 2021

In the plot on the left we can see different execution time values for different values of the CUTOFF variable. We could say that the best possible value would be between 5 and 7, maybe we could say 6. Beyond this numbers the overhead of task creation makes the execution time increase again.

The plots on the right were generated with a CUTOFF value of 6, extracted from the previous analysis. A small improvement of the scalability compared with the previous chart can be seen, the excess of tasks generated on the first tree strategy version generated an overhead that can be reduced in the cutoff version.



par2214
Speed-up wrt sequential time (complete application)
Thu May 13 08:53:08 CEST 2021



par2214
Speed-up wrt sequential time (multisort funtion only)
Thu May 13 08:53:08 CEST 2021