UNIVERSITAT POLITÈCNICA DE CATALUNYA

PARALELISMO

Deliverable4 - Sorting

Divide and Conquer parallelism with OpenMP

Autores:
Guo Haobin
Jin Haonan

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Mayo Q2 2023-2024

Index

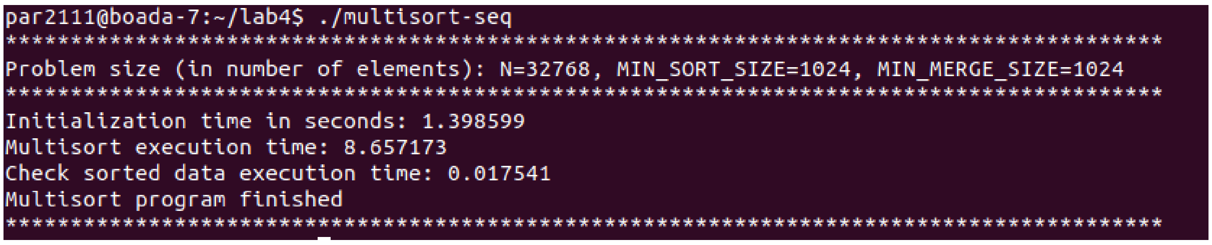
1. Introduction

In this laboratory task, we are going to examine the "Mergesort" program, which has been implemented using a "divide and conquer" approach employing both merge techniques. We will evaluate it using both the leaf and tree approaches. Furthermore, within the tree methodology, we will assess it under three conditions: without cut-off, with cut-off, and with cut-off along with a dependency clause.

2. Task decomposition analysis for Merge Sort

2.1 Divide and conquer

To begin with, we examine the given code in multisort.c and ensure our comprehension of the merge and multisort functions that constitute the Merge Sort algorithm. Subsequently, for the purpose of observing an execution instance of this code, we compile and run a sequential version of the code (available as the executable file multisort-seq). Upon executing the command "./multisort-seq -n 32768 -s 1024 -m 1024" (assuming default values when an option is unspecified), the following output is obtained:



```
par2111@boada-7:~/lab4$ ./multisort-seq
*****************************************************************************************
Problem size (in number of elements): N=32768, MIN_SORT_SIZE=1024, MIN_MERGE_SIZE=1024
*****************************************************************************************
Initialization time in seconds: 1.398599
Multisort execution time: 8.657173
Check sorted data execution time: 0.017541
Multisort program finished
*****************************************************************************************
```

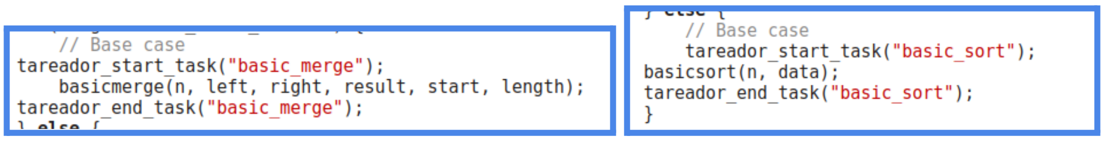
Figure1: multisort-seq output

We will use these results as reference times to check the scalability of the following versions that we will develop.

2.2 Task decomposition analysis with Tareador

To analyze the task decomposition strategies, namely the Leaf Strategy and Tree Strategy, we will employ Tareador. Let's begin by defining the appropriate tasks for each strategy:

Leaf Strategy:

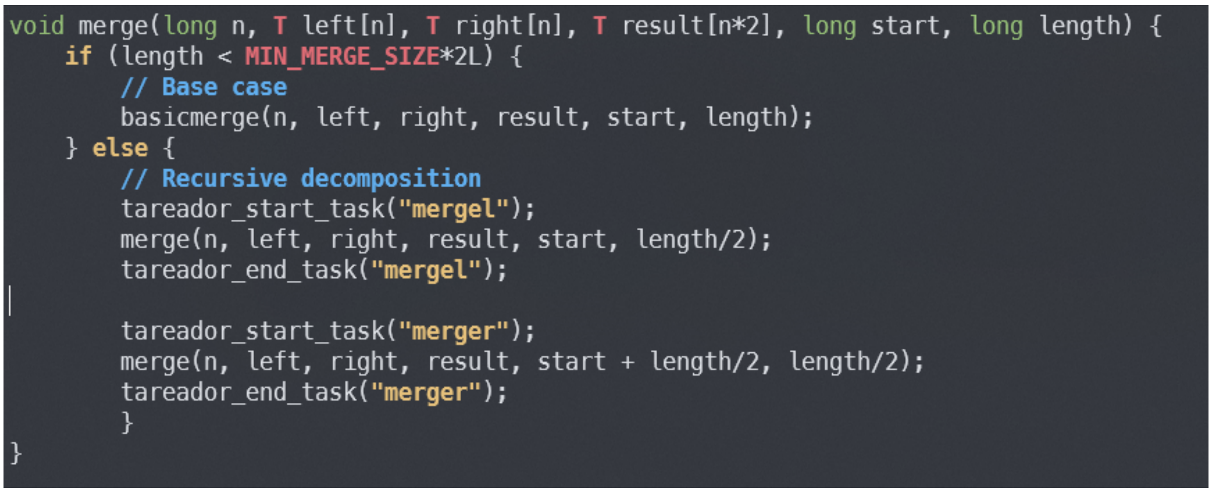


```
    // Base case
tareador_start_task("basic_merge");
    basicmerge(n, left, right, result, start, length);
tareador_end_task("basic_merge");
```

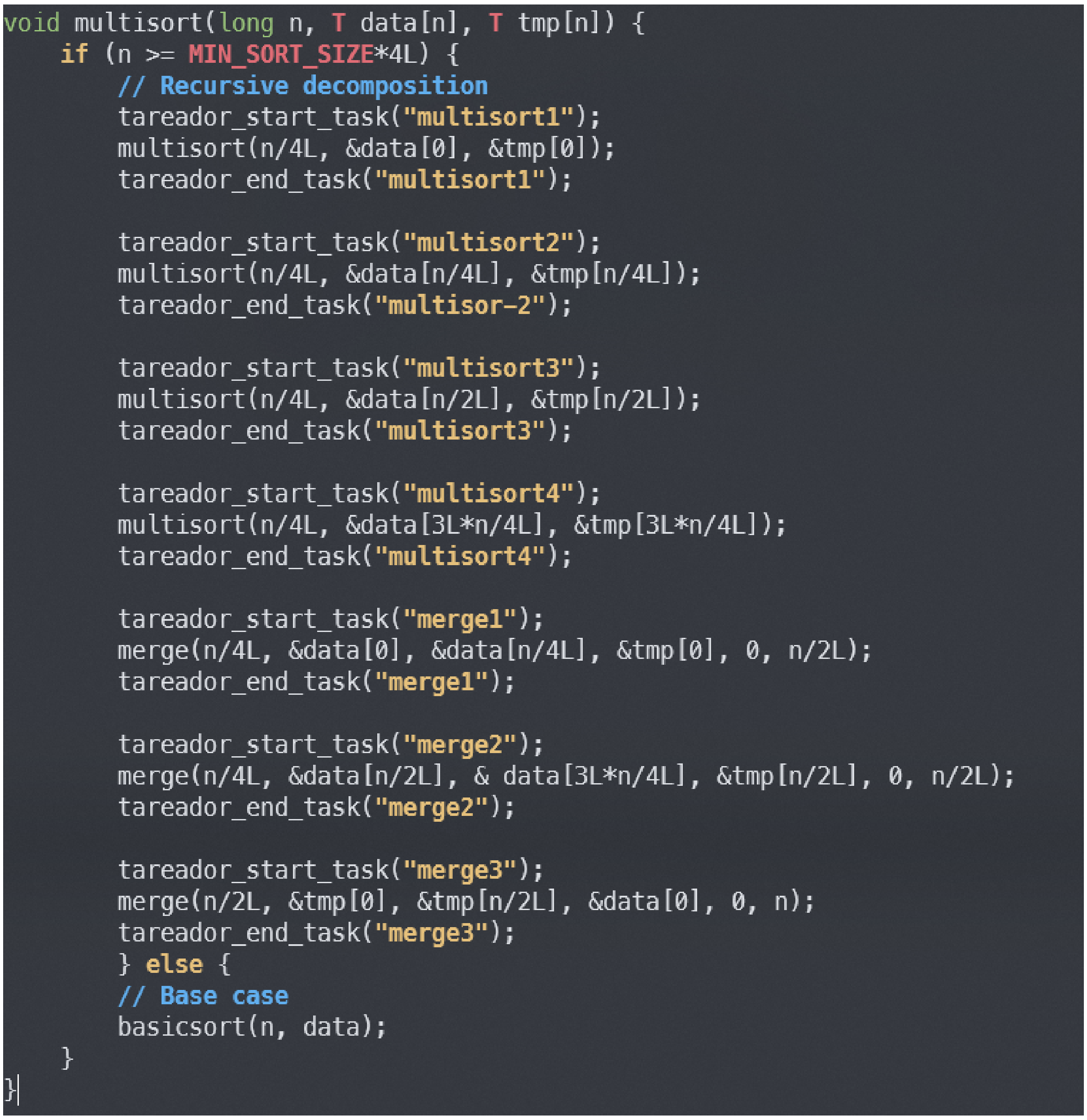
```
    // Base case
    tareador_start_task("basic_sort");
basicsort(n, data);
tareador_end_task("basic_sort");
}
```

Figure 2: Tareador Leaf strategy code

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

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



```
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("multisor-2");

        tareador_start_task("multisort3");
        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
        basicsort(n, data);
    }
}
```

Figure 3: Tareador Tree strategy code

2.3 Task dependence graphs

To generate the task dependence graphs for the Leaf Strategy and Tree Strategy using the script "run-tareador.sh,"

Leaf Strategy Task Dependence Graph:

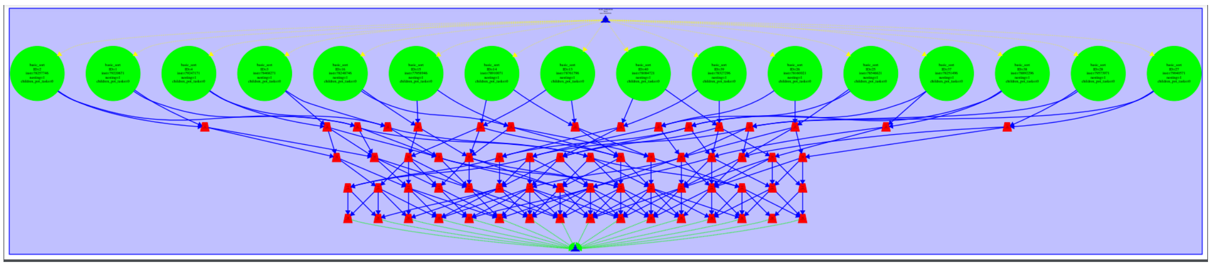


Figure 4: Leaf Strategy Task Dependece Graph

In this depiction of task dependencies, the tasks denoted by green represent the sorting tasks generated by invoking basicsort, while the tasks marked in red represent the merge tasks performed by basicmerge. Additionally, we observe that the sorting tasks can be executed independently, allowing for parallel execution. However, the merge tasks depend on both the preceding sorting tasks and the previous merges. As a result, this configuration enables four levels of parallelization, with each level comprising 16 tasks.

Tree Strategy Task Dependence Graph:

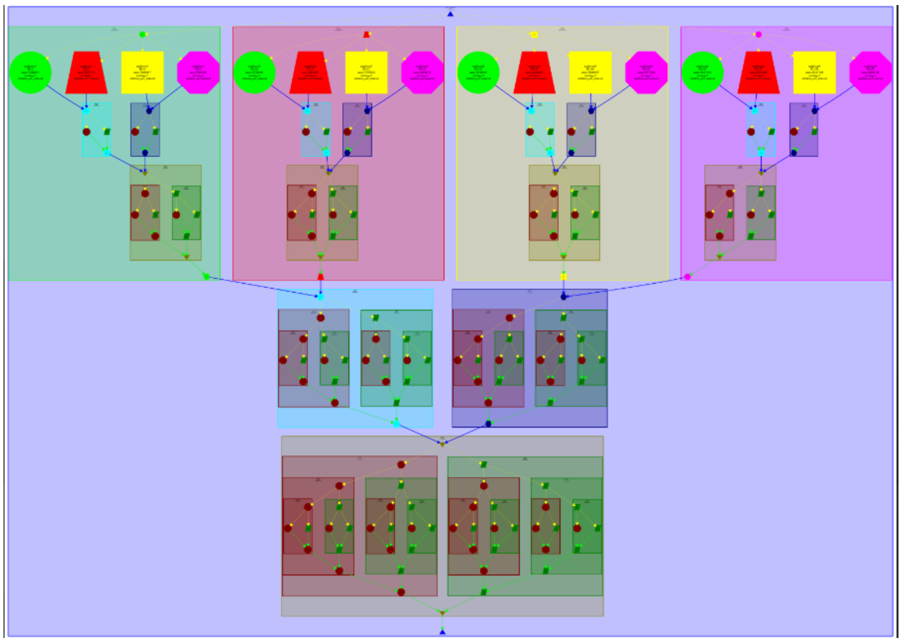
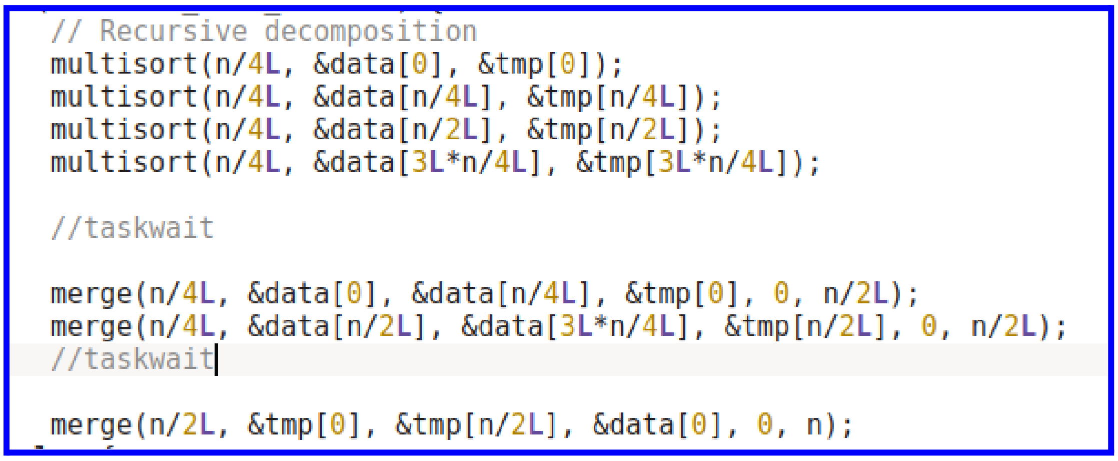


Figure 5: Tree Strategy Task Dependece Graph

The two task dependence graphs exhibit significant dissimilarities between the two strategies. Evidently, the tree strategy generates a considerably larger number of tasks due to the increased invocations of the merge and multisort functions compared to basicsort and basicmerge in the leaf strategy. The graph structure differs slightly since the leaf strategy only generates tasks at the base case of the merge and multisort functions, whereas the tree strategy involves recursive invocations.

Regarding granularity, the leaf strategy displays a substantial imbalance between the tasks basicsort and basicmerge. Conversely, the tree strategy also exhibits significant variations in task granularity, specifically between the initial multisort tasks (four of them, each corresponding to an invocation) displayed prominently in bright green, red, yellow, and magenta at the top of Fig. 2, and the subsequent merge and multisort tasks performed beneath these four tasks in each branch. Notably, the tasks representing multisort do not create dependencies since they do not rely on the main vector like the merge tasks do.

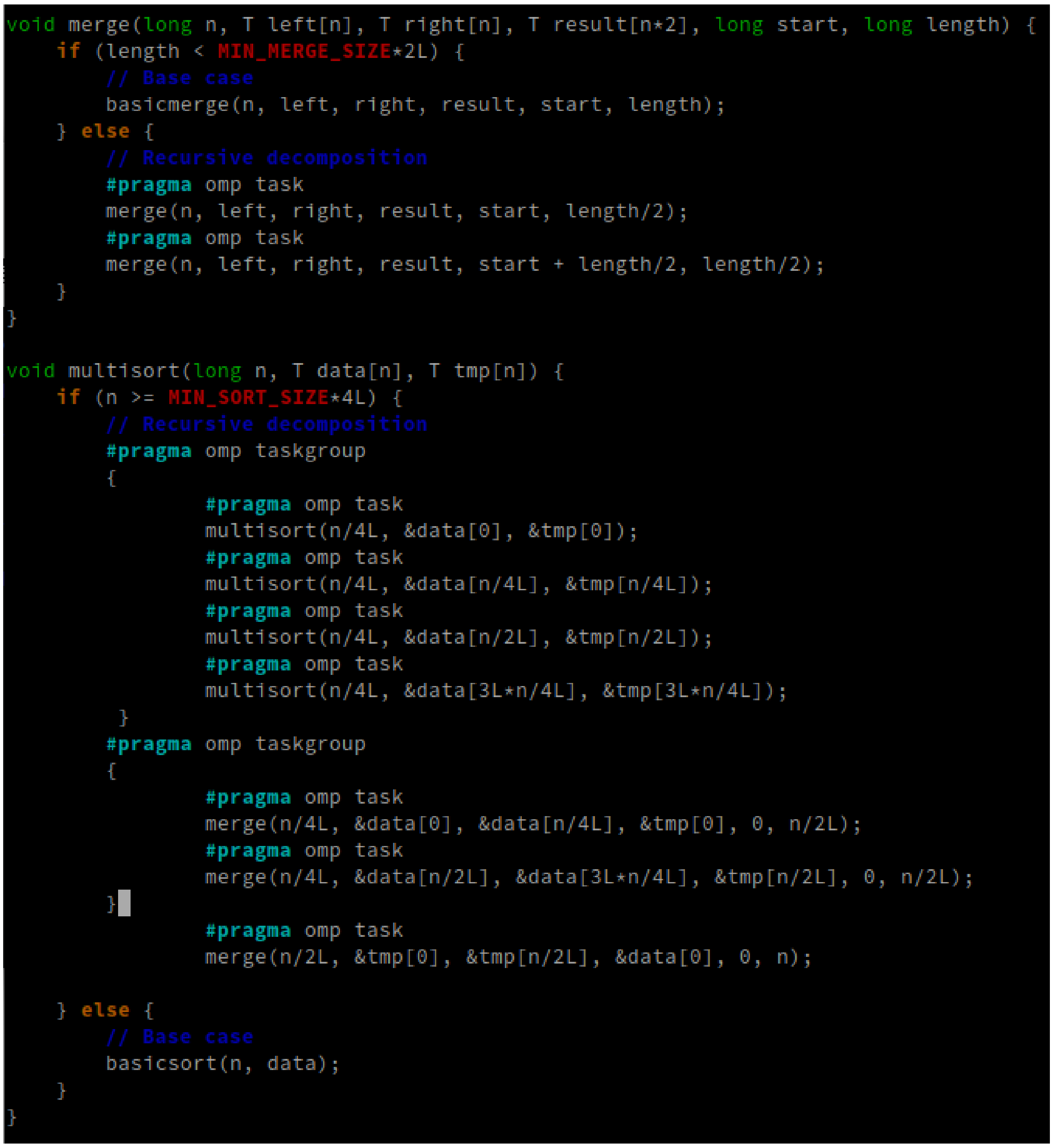
Upon examining the graphs, we have determined that incorporating synchronizations is necessary in both strategies to ensure proper handling of dependencies. As a result, we have made the decision to include specific synchronization points within the leaf strategy. Consequently, we have added a taskwait directive after the execution of the four multisort calls and another taskwait directive following the completion of the two merge calls. This synchronization arrangement is depicted in the accompanying illustration:



```
// 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]);

//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);
//taskwait

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

Figure 6: incorporating synchronizations Leaf

After careful consideration, we have inserted a taskgroup directive following the completion of the four multisort functions, as well as an additional taskgroup directive subsequent to the two merge functions within the merge operation. These operations placements are visually represented in the figures provided below.



```
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);
    }
}
```

Figure 7:incorporating synchronizations Tree

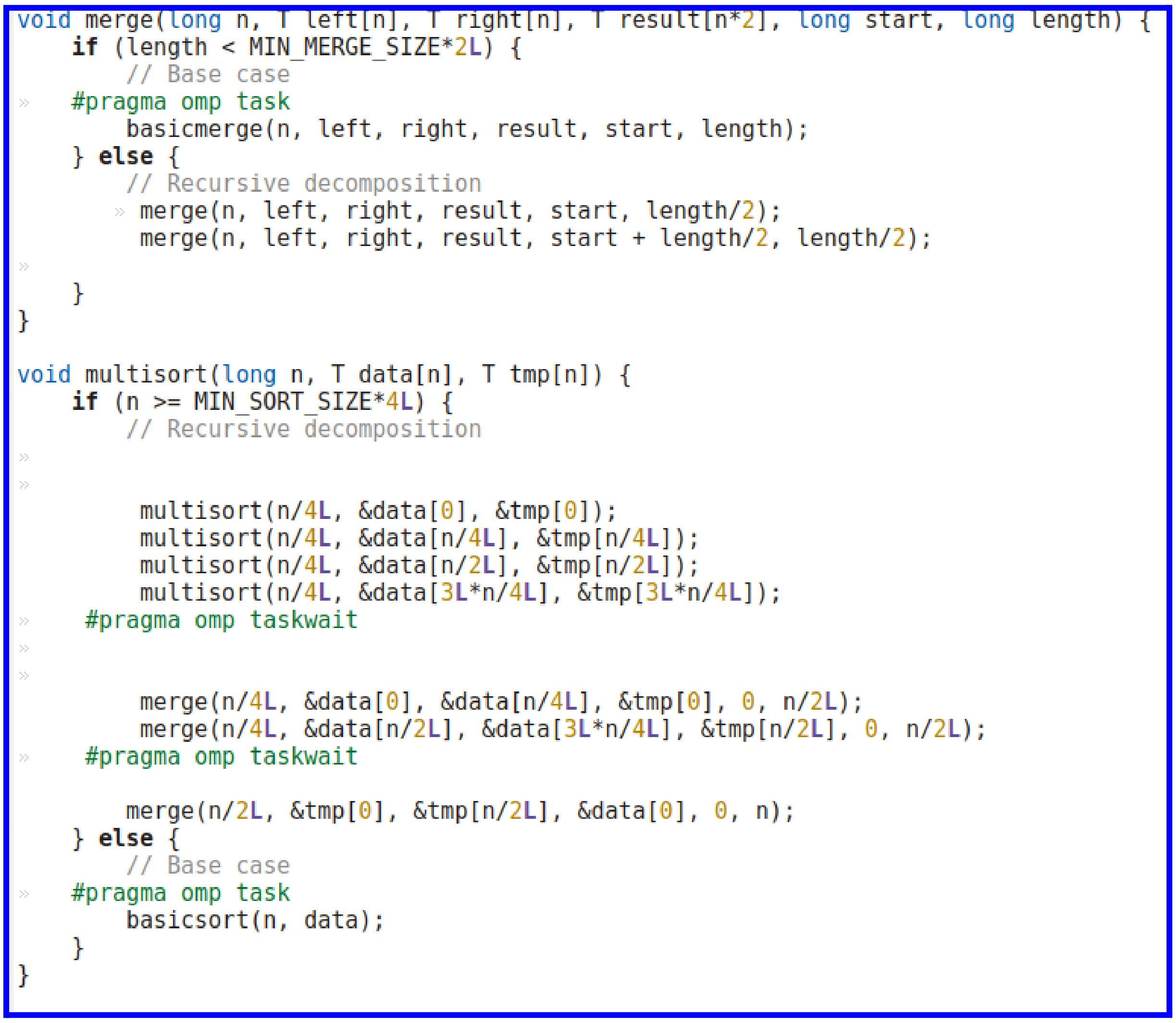
3. Shared-memory parallelisation with OpenMP task

In this segment, our focus shifts to the multisort.c file as we delve into exploring two distinct parallelization methodologies. Initially, we will commence with the leaf strategy, and subsequently proceed to implement the tree strategy.

3.1 Leaf strategy in OpenMP

To examine the leaf strategy using OpenMP, we established tasks within the base cases of the recursive merge and multisort functions. Moreover, we incorporated the necessary closure taskwait, as elaborated in section 2.2.3, to ensure correct task ordering constraints.

Subsequently, we executed the script and obtained the ensuing scalability graphs



```
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);
    }
}
```

Figure 8: OpenMP Leaf strategy code

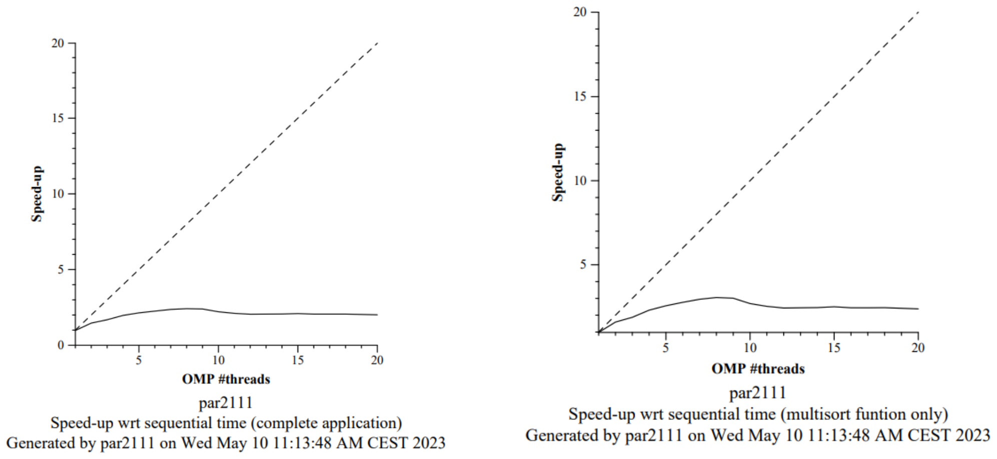
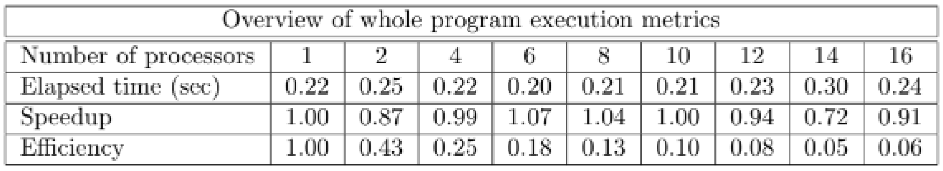


Figure 9: Speedup of Leaf Strategy

The observed speed-up achieved by this strategy does not meet our initial expectations. Consequently, we intend to conduct a more comprehensive analysis using modelfactors and Paraver to gain deeper insights into its performance.

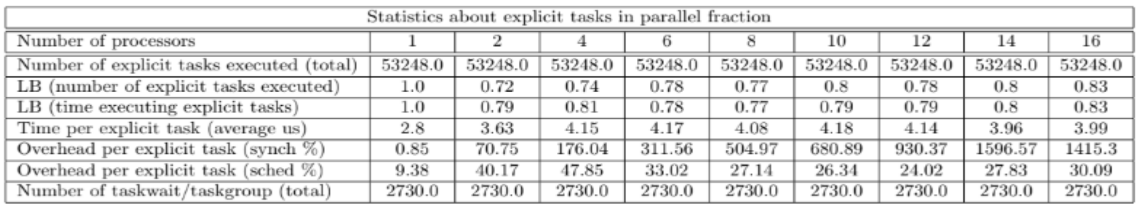
3.1.1 Analysis with modelfactors and Paraver

Those are the three tables generated for the leaf strategies



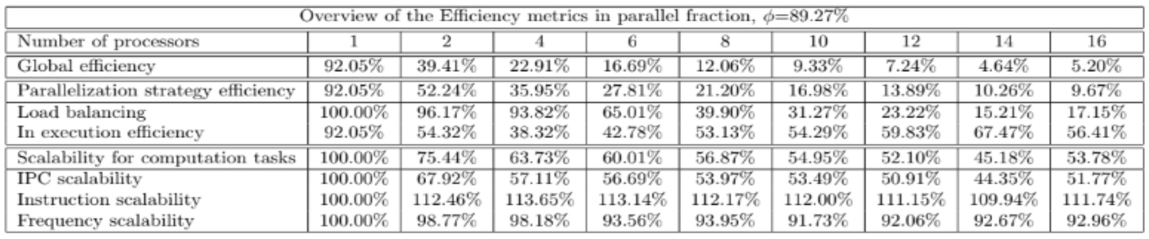
Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.22	0.25	0.22	0.20	0.21	0.21	0.23	0.30	0.24
Speedup	1.00	0.87	0.99	1.07	1.04	1.00	0.94	0.72	0.91
Efficiency	1.00	0.43	0.25	0.18	0.13	0.10	0.08	0.05	0.06

Table 1: Analysis done on Wed May 10 11:12:06 AM CEST 2023, par2111



Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0	53248.0
LB (number of explicit tasks executed)	1.0	0.72	0.74	0.78	0.77	0.8	0.78	0.8	0.83
LB (time executing explicit tasks)	1.0	0.79	0.81	0.78	0.77	0.79	0.79	0.8	0.83
Time per explicit task (average us)	2.8	3.63	4.15	4.17	4.08	4.18	4.14	3.96	3.99
Overhead per explicit task (synch %)	0.85	70.75	176.04	311.56	504.97	680.89	930.37	1596.57	1415.3
Overhead per explicit task (sched %)	9.38	40.17	47.85	33.02	27.14	26.34	24.02	27.83	30.09
Number of taskwait/taskgroup (total)	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0

Table 3: Analysis done on Wed May 10 11:12:06 AM CEST 2023, par2111



Overview of the Efficiency metrics in parallel fraction, ϕ=89.27%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	92.05%	39.41%	22.91%	16.69%	12.06%	9.33%	7.24%	4.64%	5.20%
Parallelization strategy efficiency	92.05%	52.24%	35.95%	27.81%	21.20%	16.98%	13.89%	10.26%	9.67%
Load balancing	100.00%	96.17%	93.82%	65.01%	39.90%	31.27%	23.22%	15.21%	17.15%
In execution efficiency	92.05%	54.32%	38.32%	42.78%	53.13%	54.29%	59.83%	67.47%	56.41%
Scalability for computation tasks	100.00%	75.44%	63.73%	60.01%	56.87%	54.95%	52.10%	45.18%	53.78%
IPC scalability	100.00%	67.92%	57.11%	56.69%	53.97%	53.49%	50.91%	44.35%	51.77%
Instruction scalability	100.00%	112.46%	113.65%	113.14%	112.17%	112.00%	111.15%	109.94%	111.74%
Frequency scalability	100.00%	98.77%	98.18%	93.56%	93.95%	91.73%	92.06%	92.67%	92.96%

Table 2: Analysis done on Wed May 10 11:12:06 AM CEST 2023, par2111

Upon examining the three aforementioned tables, it becomes apparent that the primary factor contributing to the low efficiency of parallelization is likely the "In execution efficiency" resulting from synchronization overhead. However, the most significant impact on efficiency stems from "Load balancing." Additionally, the IPC scalability of the Scalability for computation tasks may also affect the overall efficiency.

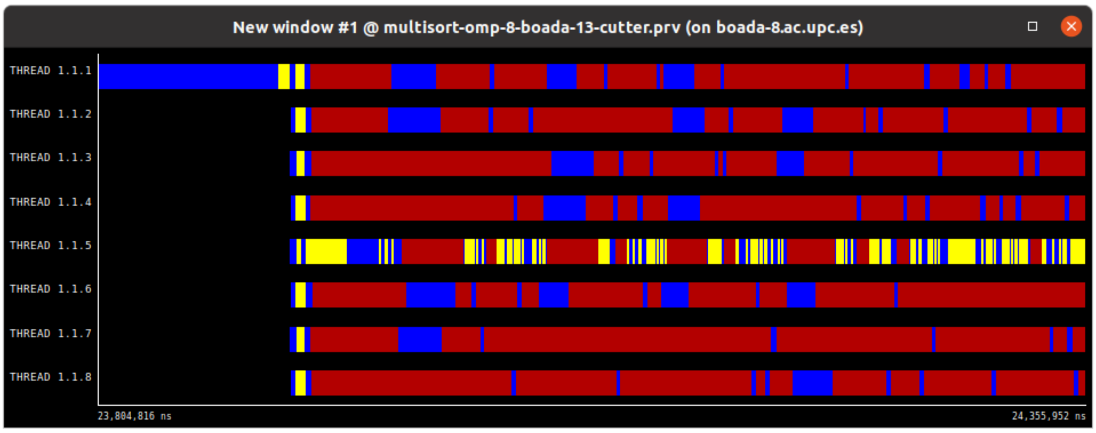


Figure 10: Trace Leaf strategy

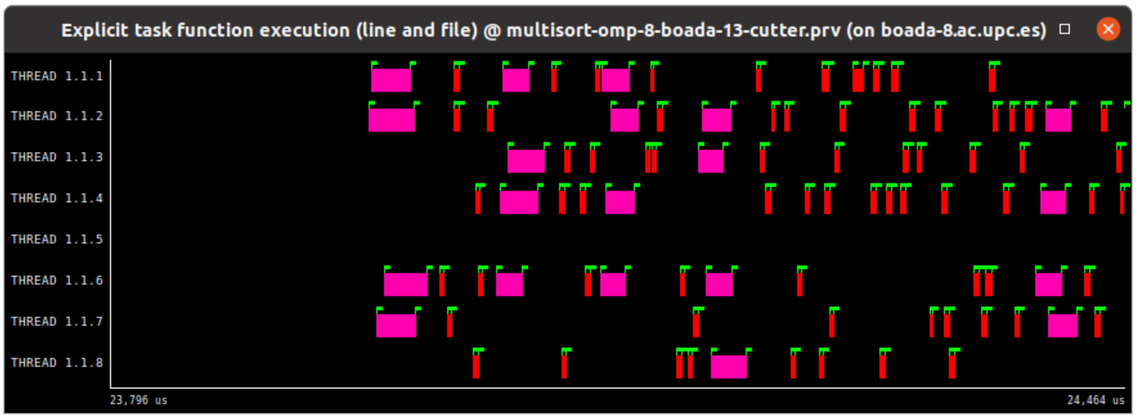


Figure 11: Trace of explicit task function execution Leaf strategy

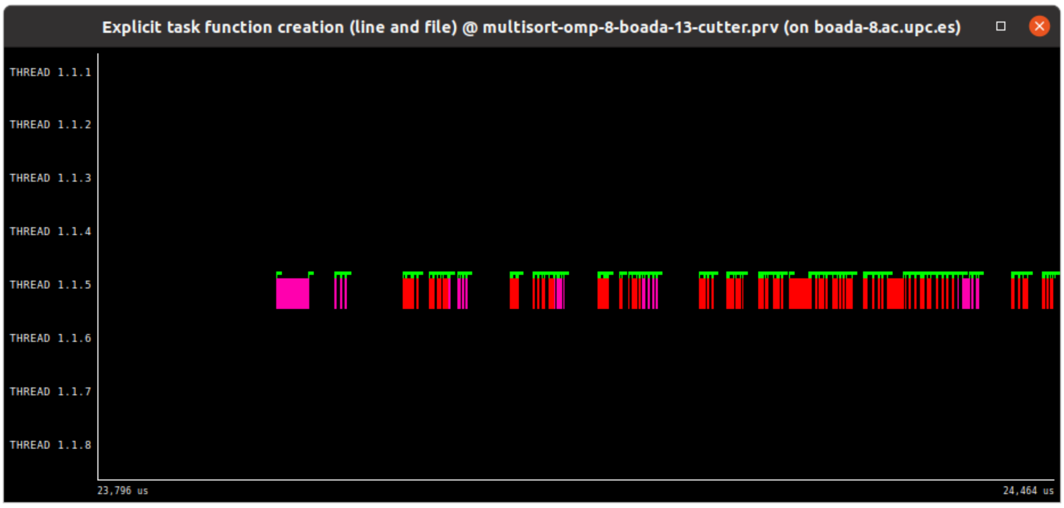


Figure 12: Trace of explicit task function creation Leaf strategy

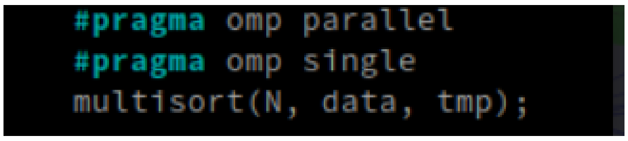
Upon analyzing the histogram, it becomes apparent that thread 5 is primarily responsible for creating tasks, as it has executed fewer tasks compared to the other threads. Furthermore, the workload appears to be evenly distributed among the various threads, with the exception of thread 5, which, as previously mentioned, is primarily dedicated to task generation.

Initially, it generates four multisort tasks, with the first task being the most time-consuming while the remaining three exhibit lower execution times due to the principle of "locality of reference." Concurrently, threads 1, 2, 6, and 7 each execute one multisort task. As a consequence of the taskwait directive, Thread 5 refrains from generating additional tasks until all multisort tasks have been executed.

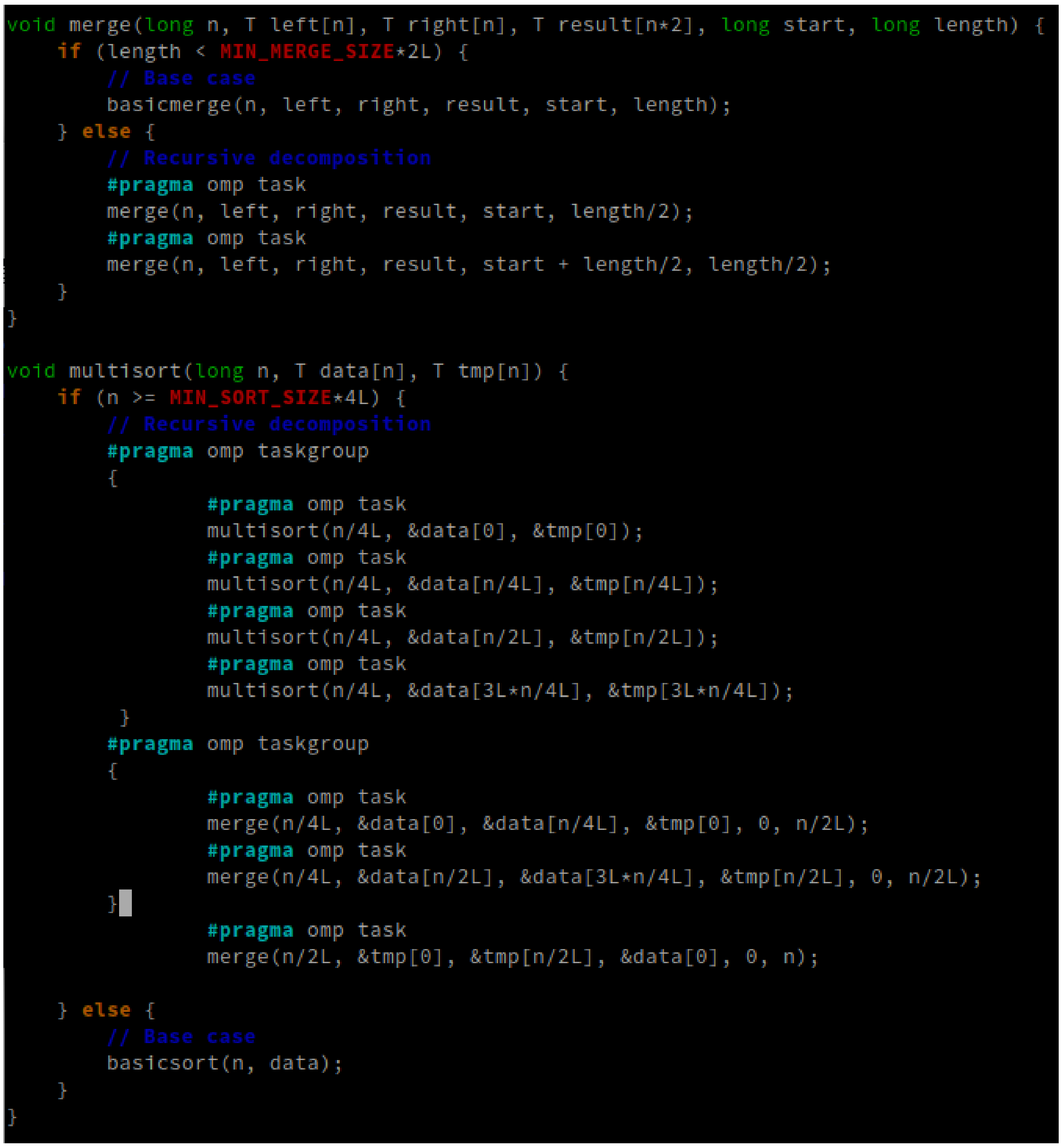
Subsequently, Thread 5 proceeds to create two merge tasks, while the other threads diligently execute their respective assigned task

Based on the presented figure and the preceding analysis, we can conclude that the program lacks a sufficient number of tasks to fully utilize all available threads concurrently. Consequently, only four tasks can be executed simultaneously, leading to underutilization of the available computational resources.

3.2 Tree strategy in OpenMP



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

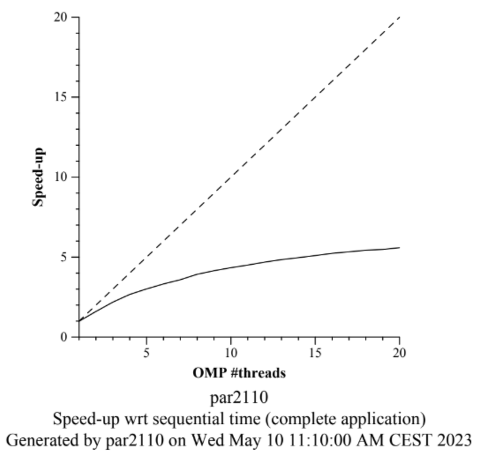


```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        #pragma omp task
        merge(n, left, 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);
    }
}
```

Figure 13: OpenMP Tree strategy code

In the tree strategy, we create task in the calls to the merge and multisort functions and using #pragma omp taskgroup from the OpenMp directive, we manage to group the tasks and synchronize the different threads.

Subsequently, by executing the "submit-strong-omp.sh" script with the binary, we generate the following graphics, providing insights into the performance and scalability of the implementation.



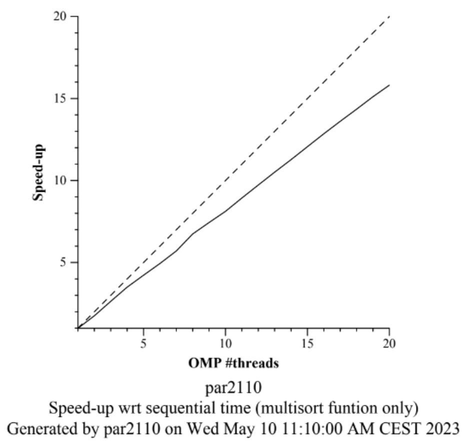


Figure 14: Speedup of Tree Strategy

Upon examining Figure 14, a significant improvement in the speedup of the multisort function becomes apparent when compared to the leaf strategy. Consequently, this enhancement has a positive impact on the overall performance of the entire application, let's analyze using modelfactors and Paraver to gain deeper insights into its performance.

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.26	0.31	0.25	0.23	0.21	0.22	0.22	0.22	0.22
Speedup	1.00	0.85	1.05	1.14	1.23	1.19	1.19	1.20	1.19
Efficiency	1.00	0.43	0.26	0.19	0.15	0.12	0.10	0.09	0.07

Table 1: Analysis done on Wed May 10 11:29:43 AM CEST 2023, par2110

Overview of the Efficiency metrics in parallel fraction, ϕ=91.33%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	88.93%	37.51%	23.39%	17.29%	14.07%	10.91%	9.09%	7.86%	6.80%
Parallelization strategy efficiency	88.93%	47.72%	34.56%	26.13%	21.04%	17.01%	14.08%	12.12%	10.52%
Load balancing	100.00%	94.94%	96.78%	93.59%	92.42%	92.16%	91.39%	89.87%	90.46%
In execution efficiency	88.93%	50.27%	35.72%	27.93%	22.76%	18.46%	15.40%	13.49%	11.63%
Scalability for computation tasks	100.00%	78.60%	67.66%	66.17%	66.87%	64.12%	64.60%	64.88%	64.66%
IPC scalability	100.00%	65.33%	56.36%	57.89%	58.42%	57.97%	58.83%	59.32%	59.12%
Instruction scalability	100.00%	121.81%	121.80%	121.83%	121.87%	121.75%	121.86%	121.84%	121.92%
Frequency scalability	100.00%	98.77%	98.56%	93.83%	93.93%	90.85%	90.11%	89.78%	89.71%

Table 2: Analysis done on Wed May 10 11:29:43 AM CEST 2023, par2110

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0
LB (number of explicit tasks executed)	1.0	0.99	0.95	0.98	0.98	0.97	0.96	0.98	0.96
LB (time executing explicit tasks)	1.0	0.98	0.99	0.99	0.98	0.99	0.99	0.99	0.99
Time per explicit task (average us)	1.83	3.95	5.7	7.24	8.63	10.87	12.82	14.63	16.7
Overhead per explicit task (synch %)	1.01	43.04	58.71	69.08	74.48	78.61	81.64	84.16	86.47
Overhead per explicit task (sched %)	13.58	32.71	46.64	57.57	65.11	71.6	76.54	79.83	82.67
Number of taskwait/taskgroup (total)	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0	2730.0

Table 3: Analysis done on Wed May 10 11:29:43 AM CEST 2023, par2110

From the tables generated using modelfactor, it is evident that the "load balancing" property has improved in the tree strategy. This improvement is attributed to the creation of larger tasks, as indicated in Table 3, where the number of explicit tasks compared to the leaf strategy. However, there is a decrease in the "In execution efficiency" due to this tradeoff. Consequently, the overall efficiency of the parallelization strategy, as well as the tree strategy, remains relatively low.

Moreover, larger tasks result in reduced synchronization overhead since the threads are more occupied with executing tasks. This highlights the influence of task granularity on performance, with the tree strategy exhibiting a coarser granularity compared to the leaf strategy.

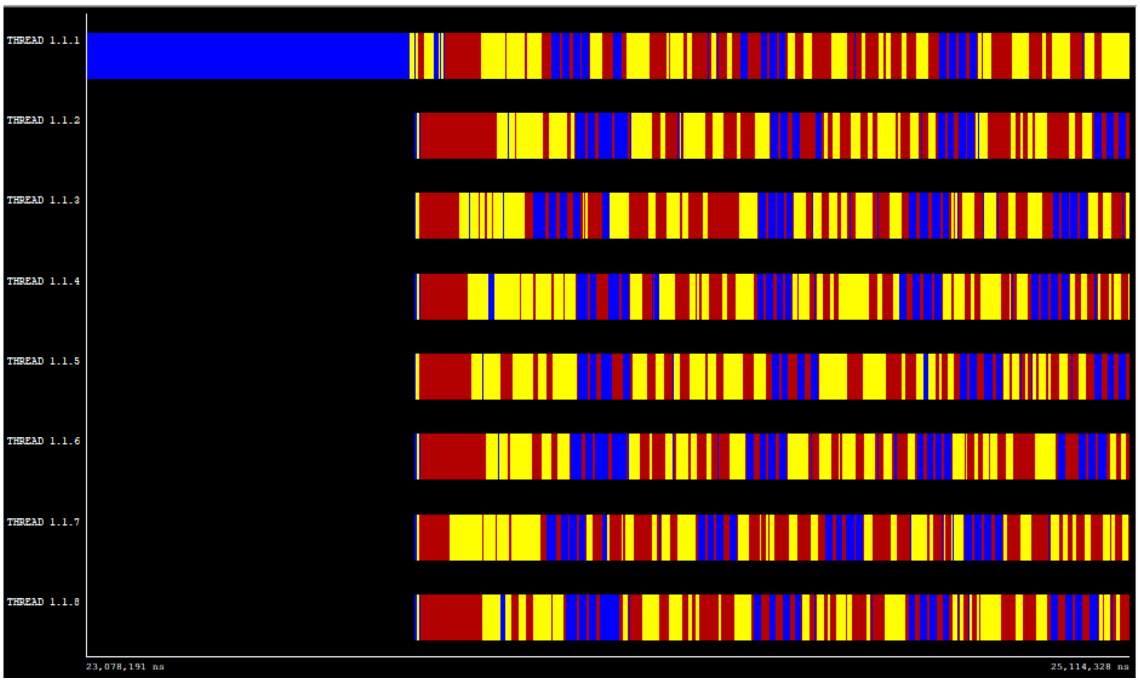


Figure 15: Trace Tree strategy

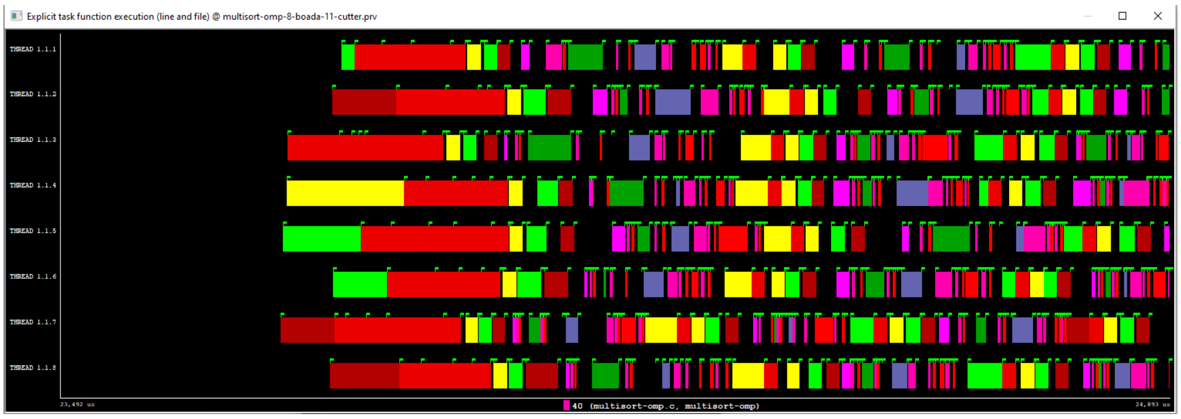

Explicit task function execution (line and file) @ multisort-omp-8-boada-11-cutter.prv
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
23,492 us
40 (multisort-omp.c, multisort-omp)
24,893 us

Figure 16: Trace of explicit task function execution Tree strategy

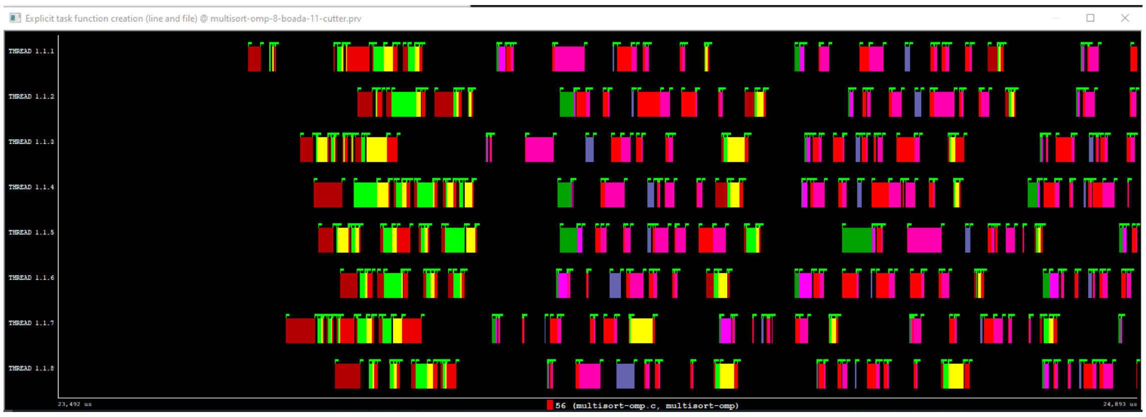
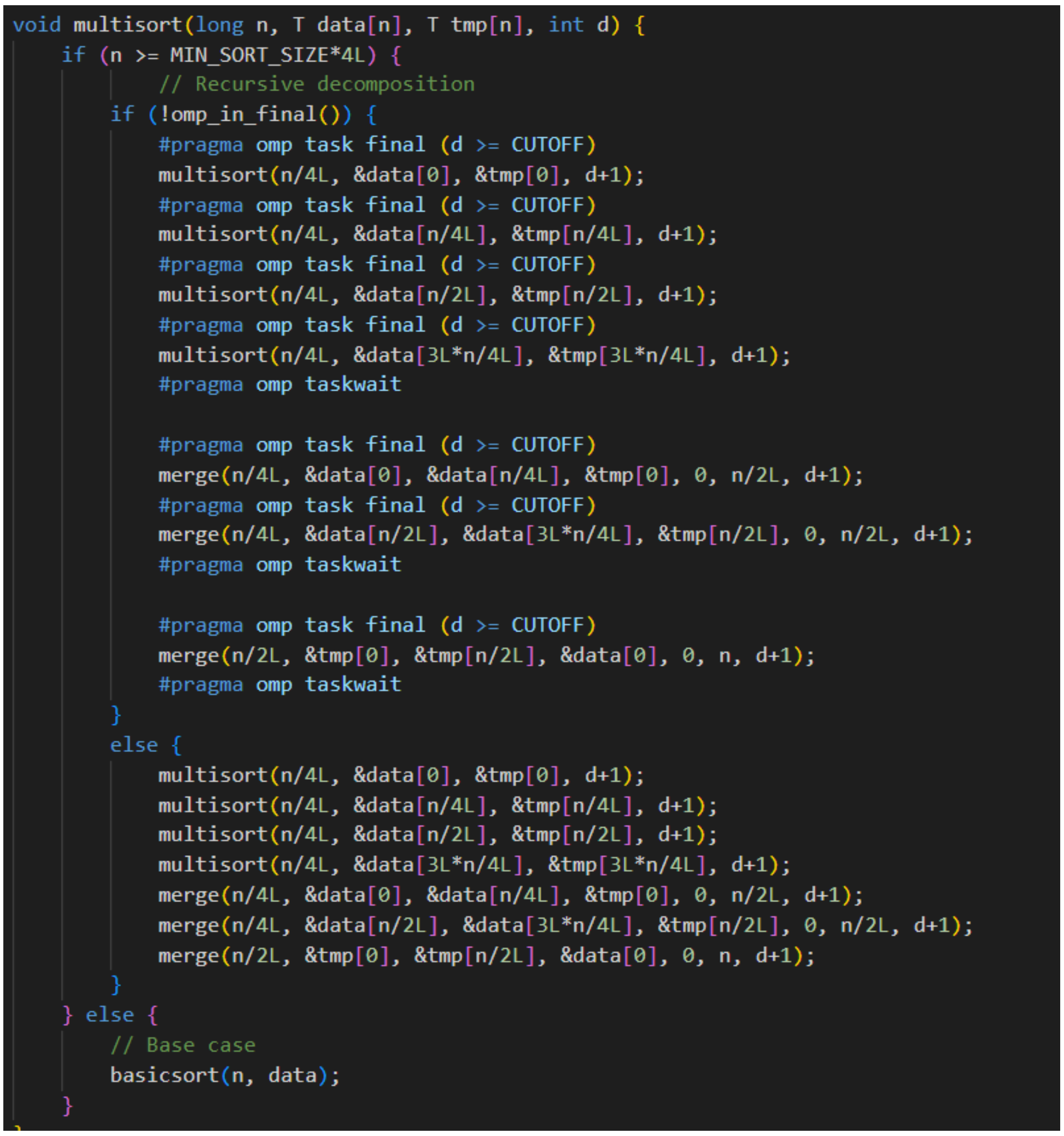

Explicit task function creation (line and file) @ multisort-omp-8-boada-11-cutter.prv
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
23,492 us
56 (multisort-omp.c, multisort-omp)
24,893 us

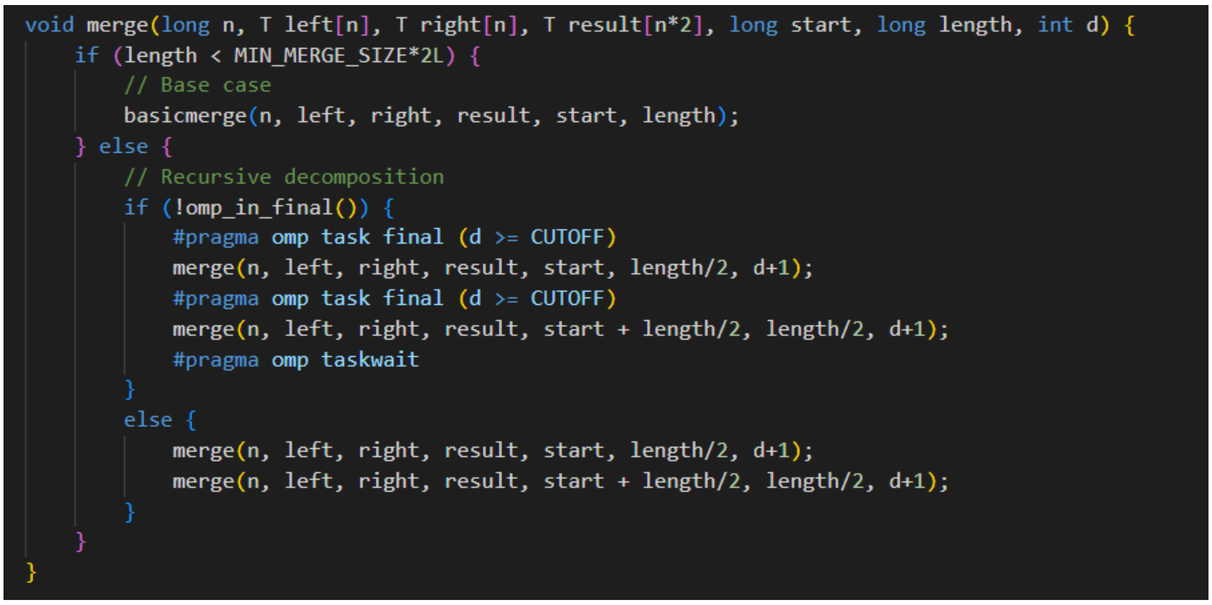
Figure 17: Trace of explicit task function creation Tree strategy

3.3 Task granularity control: the cut–off mechanism

In this section, we will be incorporating a cutoff mechanism into our tree strategy to regulate the level of task granularity. The following is the modification we have made to the code:



```
void multisort(long n, T data[n], T tmp[n], int d) {
    if (n >= MIN_SORT_SIZE*4L) {
            // Recursive decomposition
        if (!omp_in_final()) {
            #pragma omp task final (d >= CUTOFF)
            multisort(n/4L, &data[0], &tmp[0], d+1);
            #pragma omp task final (d >= CUTOFF)
            multisort(n/4L, &data[n/4L], &tmp[n/4L], d+1);
            #pragma omp task final (d >= CUTOFF)
            multisort(n/4L, &data[n/2L], &tmp[n/2L], d+1);
            #pragma omp task final (d >= CUTOFF)
            multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], d+1);
            #pragma omp taskwait

            #pragma omp task final (d >= CUTOFF)
            merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, d+1);
            #pragma omp task final (d >= CUTOFF)
            merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, d+1);
            #pragma omp taskwait

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

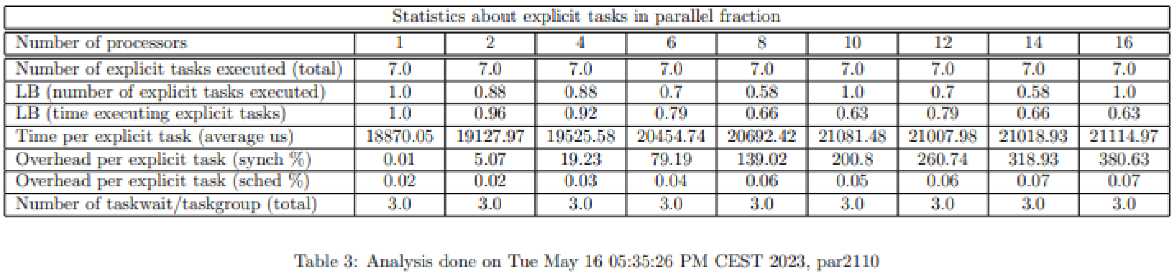


```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int d) {
    if (length < MIN_MERGE_SIZE*2L) {
        // Base case
        basicmerge(n, left, right, result, start, length);
    } else {
        // Recursive decomposition
        if (!omp_in_final()) {
        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

Figure 18: Tree strategy with cutoff code

Now we submit the submit-string-extrae.sh script with cutoff value 0 and 1:

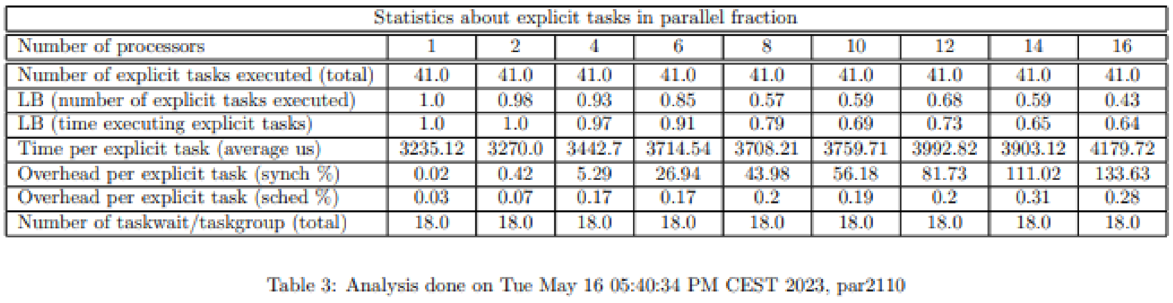
Cutoff 0



Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0	7.0
LB (number of explicit tasks executed)	1.0	0.88	0.88	0.7	0.58	1.0	0.7	0.58	1.0
LB (time executing explicit tasks)	1.0	0.96	0.92	0.79	0.66	0.63	0.79	0.66	0.63
Time per explicit task (average us)	18870.05	19127.97	19525.58	20454.74	20692.42	21081.48	21007.98	21018.93	21114.97
Overhead per explicit task (synch %)	0.01	5.07	19.23	79.19	139.02	200.8	260.74	318.93	380.63
Overhead per explicit task (sched %)	0.02	0.02	0.03	0.04	0.06	0.05	0.06	0.07	0.07
Number of taskwait/taskgroup (total)	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.0

Table 3: Analysis done on Tue May 16 05:35:26 PM CEST 2023, par2110

Cutoff 1



Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	41.0	41.0	41.0	41.0	41.0	41.0	41.0	41.0	41.0
LB (number of explicit tasks executed)	1.0	0.98	0.93	0.85	0.57	0.59	0.68	0.59	0.43
LB (time executing explicit tasks)	1.0	1.0	0.97	0.91	0.79	0.69	0.73	0.65	0.64
Time per explicit task (average us)	3235.12	3270.0	3442.7	3714.54	3708.21	3759.71	3992.82	3903.12	4179.72
Overhead per explicit task (synch %)	0.02	0.42	5.29	26.94	43.98	56.18	81.73	111.02	133.63
Overhead per explicit task (sched %)	0.03	0.07	0.17	0.17	0.2	0.19	0.2	0.31	0.28
Number of taskwait/taskgroup (total)	18.0	18.0	18.0	18.0	18.0	18.0	18.0	18.0	18.0

Table 3: Analysis done on Tue May 16 05:40:34 PM CEST 2023, par2110

We can see that, when the cutoff value is set to 0, the program generates 7 tasks. This is because tasks are no longer created beyond level 0. With a cutoff value of 1, a total of 41 tasks are generated. This comprises the 7 tasks from level 0, 4 multisort tasks multiplied by 7 (28 tasks), and 3 merge tasks multiplied by 2 (6 tasks), resulting in a total of 41 tasks.

Now we include the table of modelfactor for cutoff 4:

Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.16	0.09	0.06	0.05	0.05	0.04	0.04	0.04	0.04
Speedup	1.00	1.71	2.67	3.12	3.51	3.74	3.94	4.18	4.25
Efficiency	1.00	0.85	0.67	0.52	0.44	0.37	0.33	0.30	0.27

Table 1: Analysis done on Tue May 16 05:46:39 PM CEST 2023, par2110

Overview of the Efficiency metrics in parallel fraction, ϕ=85.43%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	98.80%	94.58%	90.44%	81.00%	77.22%	72.33%	69.47%	67.68%	63.86%
Parallelization strategy efficiency	98.80%	95.84%	94.35%	89.20%	86.26%	82.54%	80.49%	78.25%	74.63%
Load balancing	100.00%	99.98%	99.58%	98.23%	95.36%	94.90%	95.33%	95.97%	93.40%
In execution efficiency	98.80%	95.86%	94.74%	90.81%	90.45%	86.98%	84.44%	81.53%	79.90%
Scalability for computation tasks	100.00%	98.69%	95.86%	90.81%	89.52%	87.63%	86.31%	86.50%	85.57%
IPC scalability	100.00%	97.75%	96.80%	95.85%	95.17%	95.52%	94.30%	94.46%	93.66%
Instruction scalability	100.00%	101.23%	101.22%	101.21%	101.21%	101.21%	101.18%	101.19%	101.18%
Frequency scalability	100.00%	99.74%	97.83%	93.60%	92.94%	90.64%	90.46%	90.49%	90.29%

Table 2: Analysis done on Tue May 16 05:46:39 PM CEST 2023, par2110

Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	3317.0	3317.0	3317.0	3317.0	3317.0	3317.0	3317.0	3317.0	3317.0
LB (number of explicit tasks executed)	1.0	1.0	1.0	0.95	0.94	0.93	0.93	0.9	0.92
LB (time executing explicit tasks)	1.0	1.0	0.99	0.98	0.96	0.96	0.96	0.96	0.94
Time per explicit task (average us)	40.37	41.69	43.14	45.8	46.77	48.81	50.14	50.93	52.04
Overhead per explicit task (synch %)	0.54	3.19	4.19	9.48	12.59	16.48	17.47	20.13	23.02
Overhead per explicit task (sched %)	0.69	1.12	1.72	2.38	2.96	3.73	5.38	5.66	8.18
Number of taskwait/taskgroup (total)	1488.0	1488.0	1488.0	1488.0	1488.0	1488.0	1488.0	1488.0	1488.0

Table 3: Analysis done on Tue May 16 05:46:39 PM CEST 2023, par2110

Compared to the initial version, there is a significant change in terms of task granularity in the cutoff versions. In the original version, tasks were very short, leading to scheduling overhead. However, in the cutoff versions, tasks are larger, resulting in synchronization overhead but reducing the time spent on scheduling.

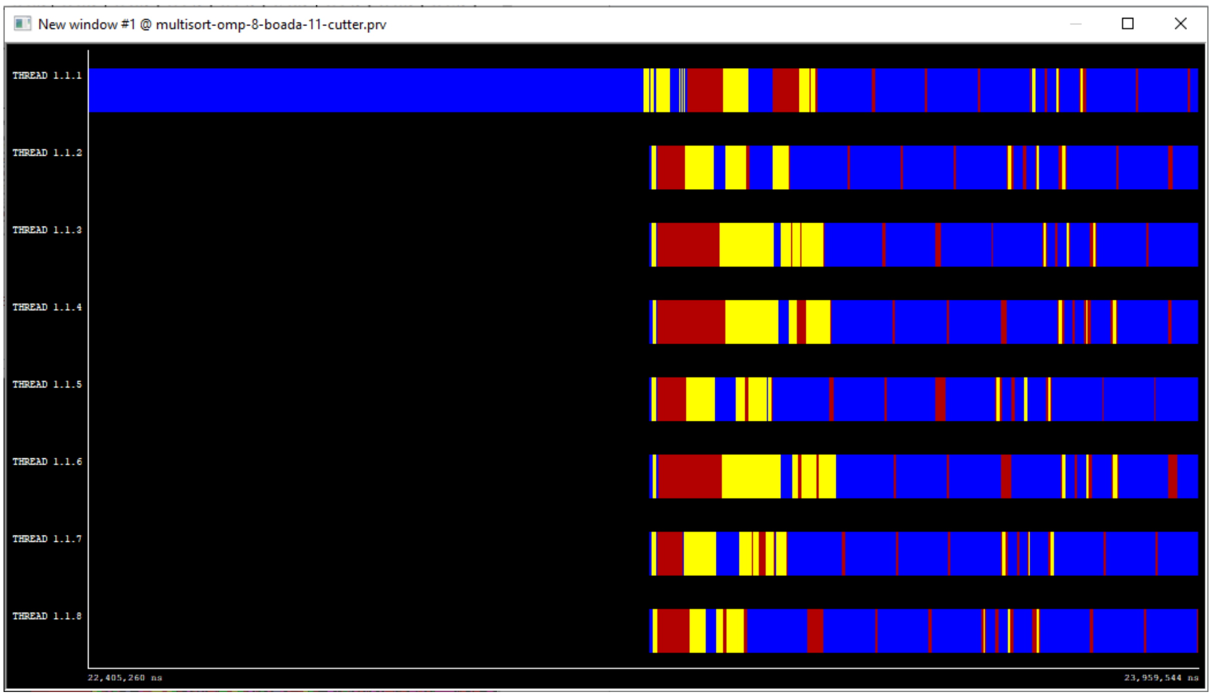


Figure 19 : Trace Tree strategy with cutoff

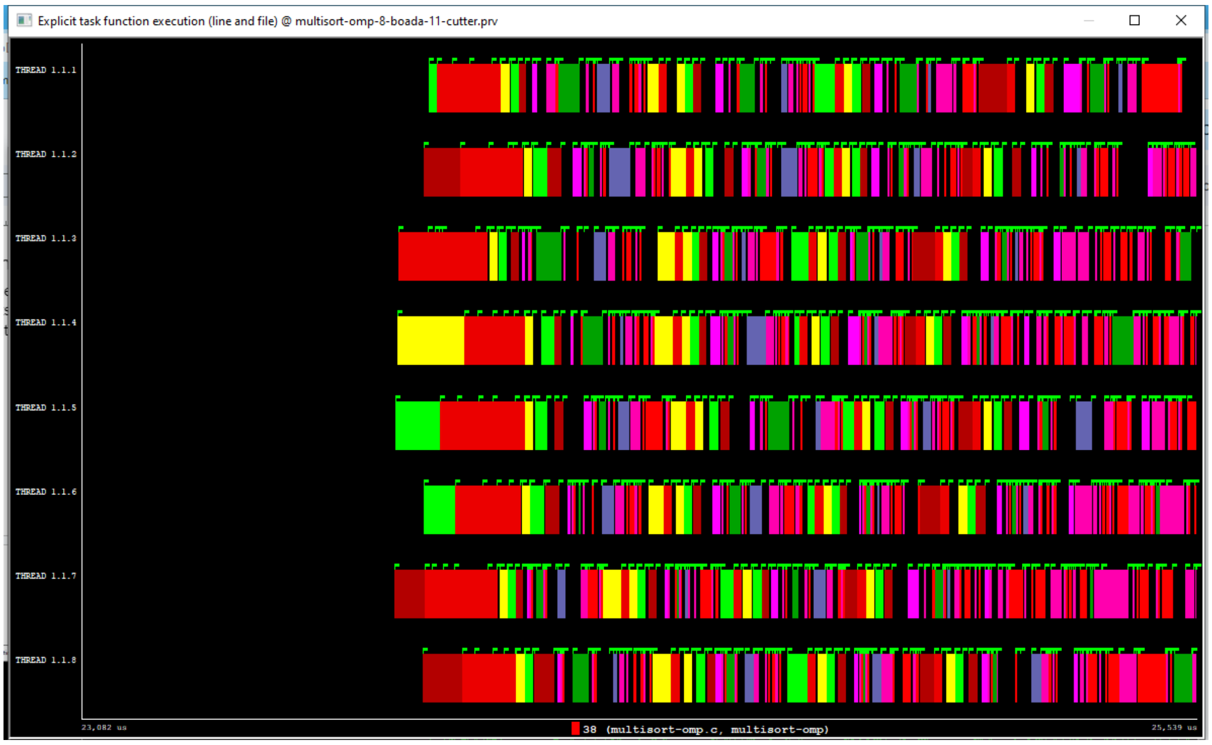

Explicit task function execution (line and file) @ multisort-omp-8-boada-11-cutter.prv
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
23,082 us
38 (multisort-omp.c, multisort-omp)
25,539 us

Figure 20: Trace of explicit task function execution Tree strategy with cutoff

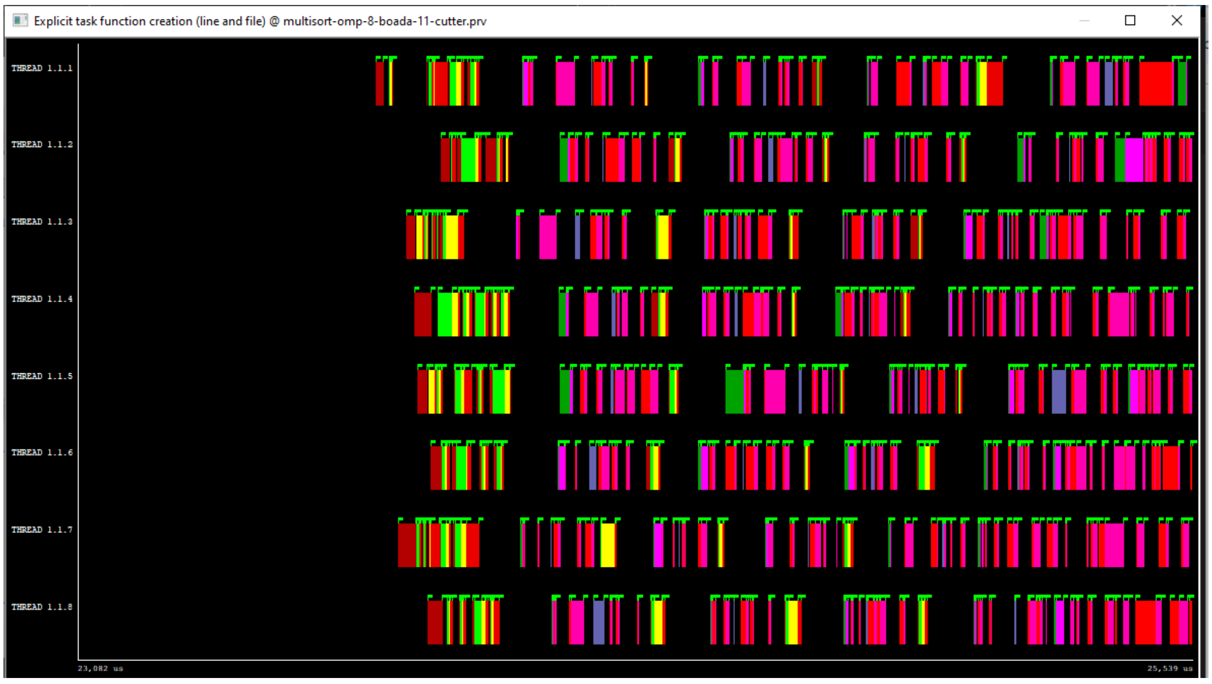
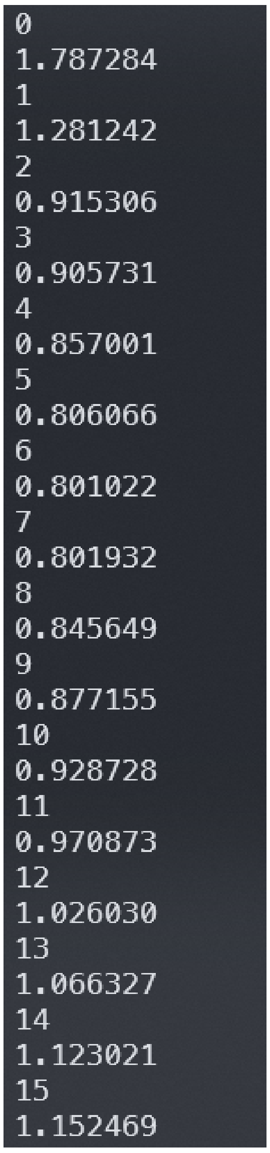


Figure 21: Trace of explicit task function creation Tree strategy with cutoff

We utilized the script submit-cutoff-omp.sh to compare various cutoff values based on the number of threads employed. We ran the script and here are the obtained results:



```
0
1.787284
1
1.281242
2
0.915306
3
0.905731
4
0.857001
5
0.806066
6
0.801022
7
0.801932
8
0.845649
9
0.877155
10
0.928728
11
0.970873
12
1.026030
13
1.066327
14
1.123021
15
1.152469
```

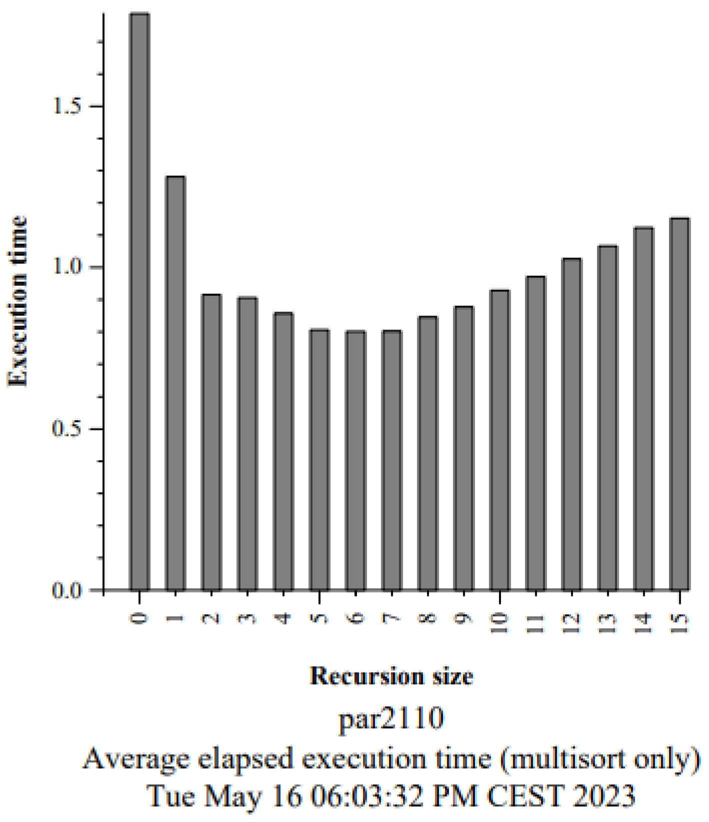


Figure 22: Graphic of the average elapsed execution time and its values.

We can see the best value for the cutoff is 6 , so. we use the value 6 to submit the submit-omp-strong.sh script.

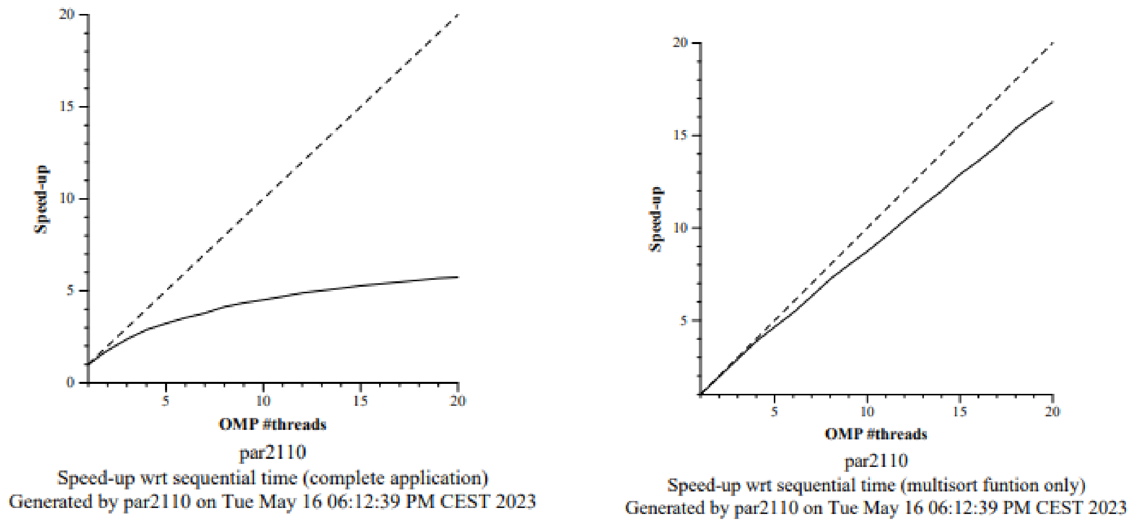


Figure 23: Speedup of Tree Strategy with cutoff

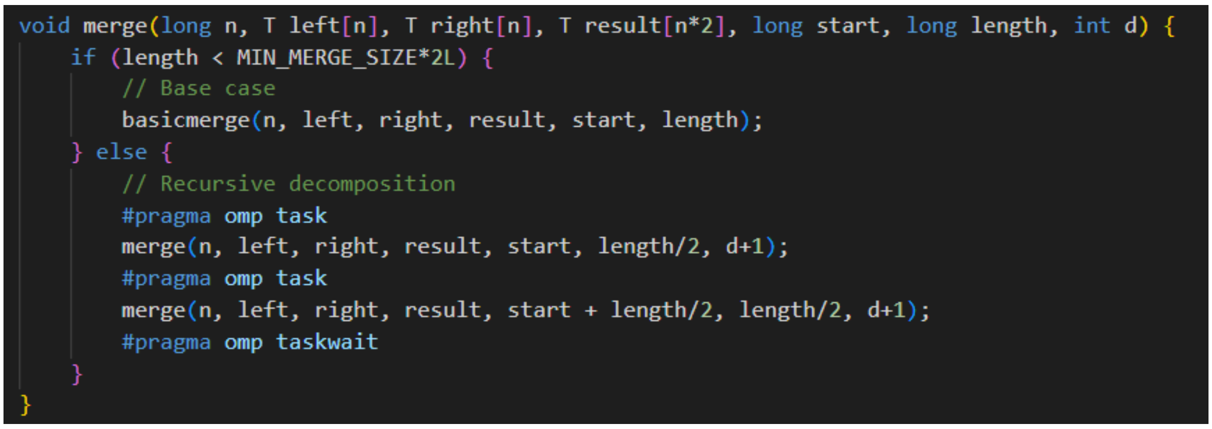
With the best value of cutoff, the version performs similarly to the tree strategy in terms of scalability, as shown in Figure 23. The overall program experiences a rapid loss in speed, but the multisort function manages to closely follow the optimal scalability line.

When using our own values, specifically a cutoff of 6 (one of the optimal values identified in the previous analysis), we observe even better scalability. The scalability line is even closer to the optimal line compared to using the maximum value of cutoff. In other words, the performance scales more effectively with the chosen cutoff value.

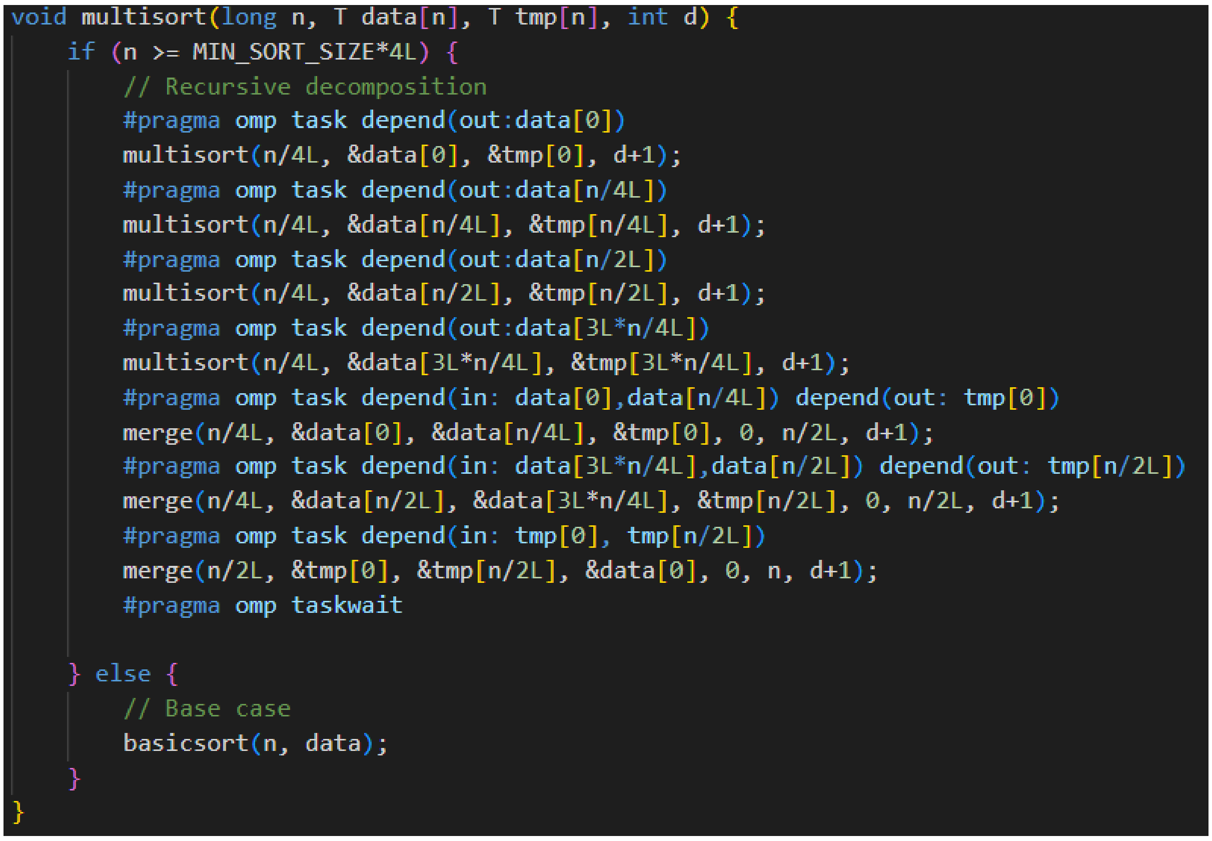
This suggests that selecting an appropriate cutoff value can significantly impact the scalability and performance of the code, and in this case, using a cutoff value of 6 yields better results.

4. Shared-memory parallelisation with OpenMP task using dependencies

In this final section, we have made modifications to our code in the Tree Strategy with the cutoff mechanism to introduce task dependencies and eliminate certain previously added synchronizations. To achieve this, we have incorporated the "depend" clause. Here is the updated code:



```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, int d) {
    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, d+1);
        #pragma omp task
        merge(n, left, right, result, start + length/2, length/2, d+1);
        #pragma omp taskwait
    }
}
```



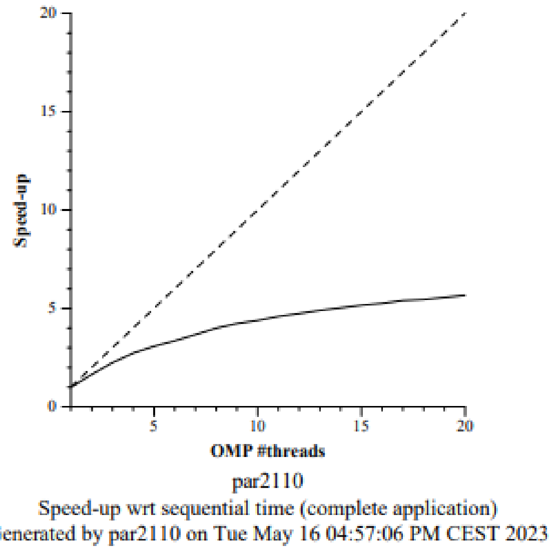
```
void multisort(long n, T data[n], T tmp[n], int d) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        #pragma omp task depend(out:data[0])
        multisort(n/4L, &data[0], &tmp[0], d+1);
        #pragma omp task depend(out:data[n/4L])
        multisort(n/4L, &data[n/4L], &tmp[n/4L], d+1);
        #pragma omp task depend(out:data[n/2L])
        multisort(n/4L, &data[n/2L], &tmp[n/2L], d+1);
        #pragma omp task depend(out:data[3L*n/4L])
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], d+1);
        #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, d+1);
        #pragma omp task depend(in: data[3L*n/4L],data[n/2L]) depend(out: tmp[n/2L])
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, d+1);
        #pragma omp task depend(in: tmp[0], tmp[n/2L])
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, d+1);
        #pragma omp taskwait

    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Figure 24: Tree strategy with dependencies code

In the modified code, we have established dependencies between the merges and the corresponding multisort functions based on the output data generated. The first merge depends on the output of the first and second multisort, while the second merge depends on the output of the third and fourth multisort. Additionally, the last merge depends on the other two merges. To reflect these dependencies, we have added output dependencies in the multisort recursive calls and input dependencies in the merge recursive call.

To ensure the correctness of the data, we have retained the original "taskwait" after the last merge.

In the next step, we will generate strong scalability plots for this new code and compare them with the plots from the previous section.



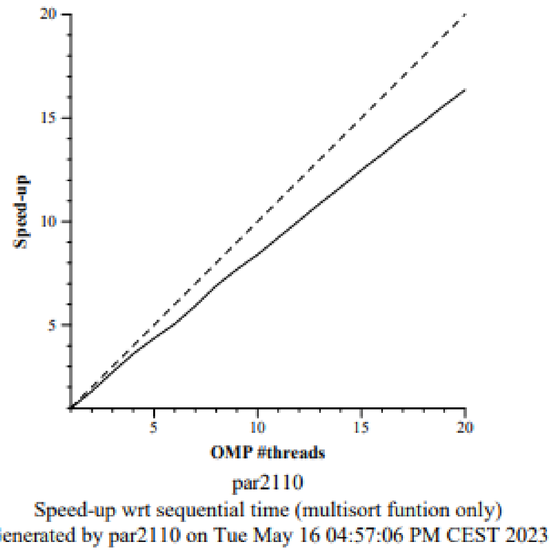
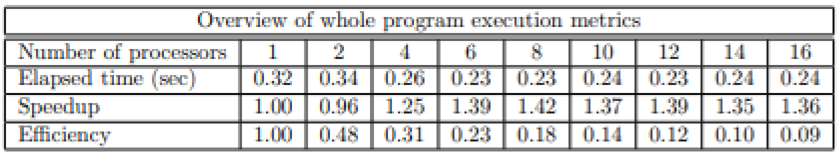


Figure 25: Speedup of Tree Strategy with dependencies

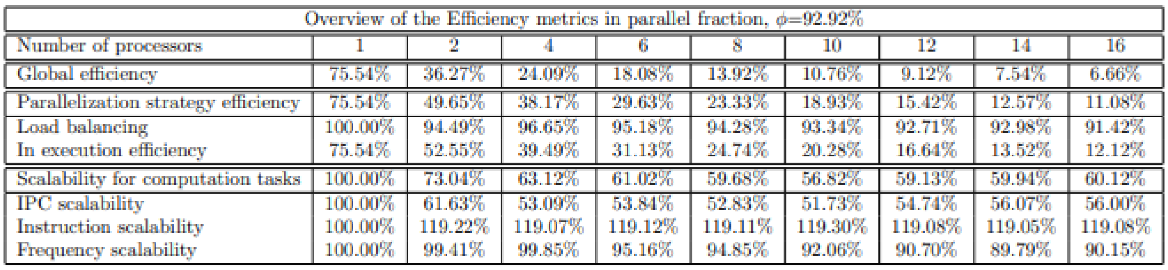
Upon comparing the previous plots with Figure 23, it appears that there is no noticeable difference in terms of performance between the two versions. Both graphics exhibit similar speedup patterns. However, when it comes to programmability, the previous version is considered easier to program compared to the new version with dependencies. In the new version, it becomes necessary to analyze the dependencies of variables in each task.

Nonetheless, the version with dependencies is more comprehensible and aids in understanding the code's behavior.



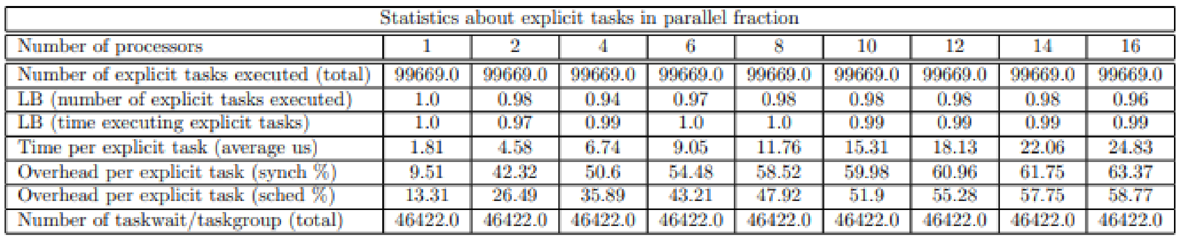
Overview of whole program execution metrics									
Number of processors	1	2	4	6	8	10	12	14	16
Elapsed time (sec)	0.32	0.34	0.26	0.23	0.23	0.24	0.23	0.24	0.24
Speedup	1.00	0.96	1.25	1.39	1.42	1.37	1.39	1.35	1.36
Efficiency	1.00	0.48	0.31	0.23	0.18	0.14	0.12	0.10	0.09

Table 1: Analysis done on Tue May 16 04:32:06 PM CEST 2023, par2110



Overview of the Efficiency metrics in parallel fraction, ϕ=92.92%									
Number of processors	1	2	4	6	8	10	12	14	16
Global efficiency	75.54%	36.27%	24.09%	18.08%	13.92%	10.76%	9.12%	7.54%	6.66%
Parallelization strategy efficiency	75.54%	49.65%	38.17%	29.63%	23.33%	18.93%	15.42%	12.57%	11.08%
Load balancing	100.00%	94.49%	96.65%	95.18%	94.28%	93.34%	92.71%	92.98%	91.42%
In execution efficiency	75.54%	52.55%	39.49%	31.13%	24.74%	20.28%	16.64%	13.52%	12.12%
Scalability for computation tasks	100.00%	73.04%	63.12%	61.02%	59.68%	56.82%	59.13%	59.94%	60.12%
IPC scalability	100.00%	61.63%	53.09%	53.84%	52.83%	51.73%	54.74%	56.07%	56.00%
Instruction scalability	100.00%	119.22%	119.07%	119.12%	119.11%	119.30%	119.08%	119.05%	119.08%
Frequency scalability	100.00%	99.41%	99.85%	95.16%	94.85%	92.06%	90.70%	89.79%	90.15%

Table 2: Analysis done on Tue May 16 04:32:06 PM CEST 2023, par2110



Statistics about explicit tasks in parallel fraction									
Number of processors	1	2	4	6	8	10	12	14	16
Number of explicit tasks executed (total)	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0	99669.0
LB (number of explicit tasks executed)	1.0	0.98	0.94	0.97	0.98	0.98	0.98	0.98	0.96
LB (time executing explicit tasks)	1.0	0.97	0.99	1.0	1.0	0.99	0.99	0.99	0.99
Time per explicit task (average us)	1.81	4.58	6.74	9.05	11.76	15.31	18.13	22.06	24.83
Overhead per explicit task (synch %)	9.51	42.32	50.6	54.48	58.52	59.98	60.96	61.75	63.37
Overhead per explicit task (sched %)	13.31	26.49	35.89	43.21	47.92	51.9	55.28	57.75	58.77
Number of taskwait/taskgroup (total)	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0	46422.0

Table 3: Analysis done on Tue May 16 04:32:06 PM CEST 2023, par2110

Upon examining the tables, it becomes evident that the addition of dependencies has not resulted in significant differences. As anticipated, the synchronization overhead has decreased, leading to a reduction in the number of taskwait statements. However, contrary to initial expectations, this version does not demonstrate a performance improvement. In theory, after executing the first two multisort operations, the first merge could be executed without waiting for all four multisort operations to complete.

Also, we used the Paraver to generate the instantaneous parallelism:

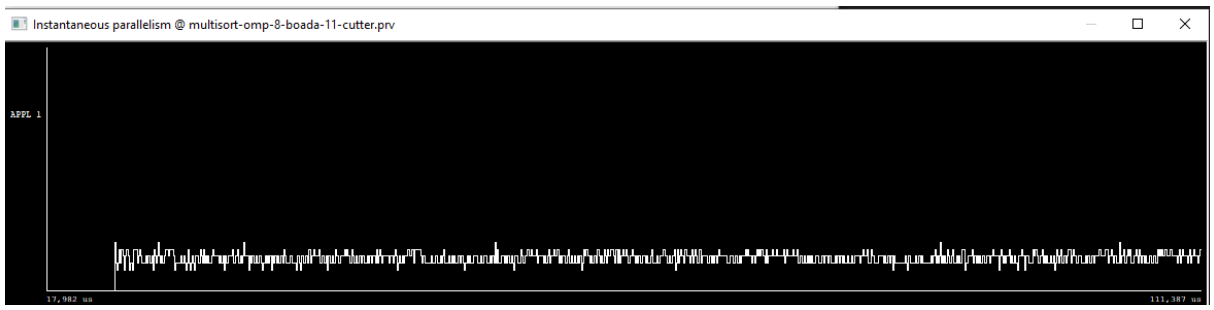

Instantaneous parallelism @ multisort-omp-8-boada-11-cutter.prv
APPL 1
17,982 us
111,387 us

Figure 26: Instantaneous parallelism

Looking at the instantaneous graph, we can see that the tasks have been distributed equally and feed all threads.

5.CONCLUSIONS

To summarize, this laboratory assignment has provided us with a deeper understanding of two parallelization strategies: the tree strategy and the leaf strategy. We have also gained proficiency in utilizing the cut-off mechanism to determine the optimal number of tasks to create and the depend clause to reduce taskwaits.

In conclusion, we have determined that the tree strategy is superior for parallelizing divide and conquer algorithms, as it exhibits a significant improvement in speedup compared to the leaf strategy. Additionally, we conducted experiments with different cut-off values for the mergesort algorithm using eight threads and found that a cut-off value of 6 yields the best results in our case. We also attempted to enhance the program by introducing taskwaits with dependencies, but the resulting improvements were minimal. Lastly, we parallelized the initialize and clear functions of the main to further optimize the program's performance.