PAR Laboratory Assignment

Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

Aleix Boné
Alex Herrero
`par2109`

2020-05-15

Contents

1 Introduction

1.1 Analysis with *Tareador*

```
20 void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
21     if (length < MIN_MERGE_SIZE*2L) {
22         // Base case
23         tareador_start_task("basicmerge");
24         basicmerge(n, left, right, result, start, length);
25         tareador_end_task("basicmerge");
26     } else {
27         // Recursive decomposition
28         merge(n, left, right, result, start, length/2);
29         merge(n, left, right, result, start + length/2, length/2);
30     }
31 }
```

```
33 void multisort(long n, T data[n], T tmp[n]) {
34     if (n >= MIN_SORT_SIZE*4L) {
35         // Recursive decomposition
36         multisort(n/4L, &data[0], &tmp[0]);
37         multisort(n/4L, &data[n/4L], &tmp[n/4L]);
38         multisort(n/4L, &data[n/2L], &tmp[n/2L]);
39         multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
40 
41         merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
42         merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
43 
44         merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
45     } else {
46         // Base case
47         tareador_start_task("basicsort");
48         basicsort(n, data);
49         tareador_end_task("basicsort");
50     }
51 }
```

Listing 1: Calls to the tareador API added to multisort-tareador.c for the leaf task decomposition

As we can observe in listing 1 for the leaf task decomposition we added the `tareador_start_task()` and the `tareador_end_task()` in lines 23 and 25 respectively.

```
20 void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
21     tareador_start_task("merge");
22     if (length < MIN_MERGE_SIZE*2L) {
23         // Base case
24         basicmerge(n, left, right, result, start, length);
25     } else {
26         // Recursive decomposition
27         merge(n, left, right, result, start, length/2);
28         merge(n, left, right, result, start + length/2, length/2);
29     }
30     tareador_end_task("merge");
31 }
```

```
33 void multisort(long n, T data[n], T tmp[n]) {
34     tareador_start_task("multisort");
35     if (n >= MIN_SORT_SIZE*4L) {
36         // Recursive decomposition
37         multisort(n/4L, &data[0], &tmp[0]);
38         multisort(n/4L, &data[n/4L], &tmp[n/4L]);
39         multisort(n/4L, &data[n/2L], &tmp[n/2L]);
40         multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);
41
42         merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
43         merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);
44
45         merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
46     } else {
47         // Base case
48         basicsort(n, data);
49     }
50     tareador_end_task("multisort");
51 }
```

Listing 2: Calls to the tareador API added to multisort-tareador.c for the tree task decomposition

In this case for tree task decomposition we added the `tareador_start_task()` in line 47 and the `tareador_end_task()` in line 49 as shown in the listing 2.

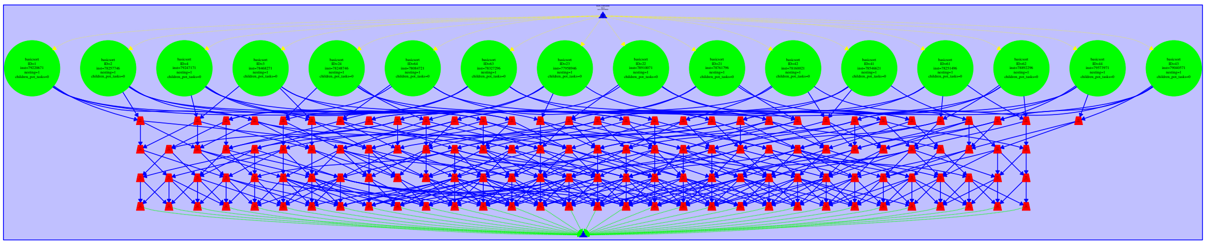


Figure 1: Task dependence graph for leaf decomposition

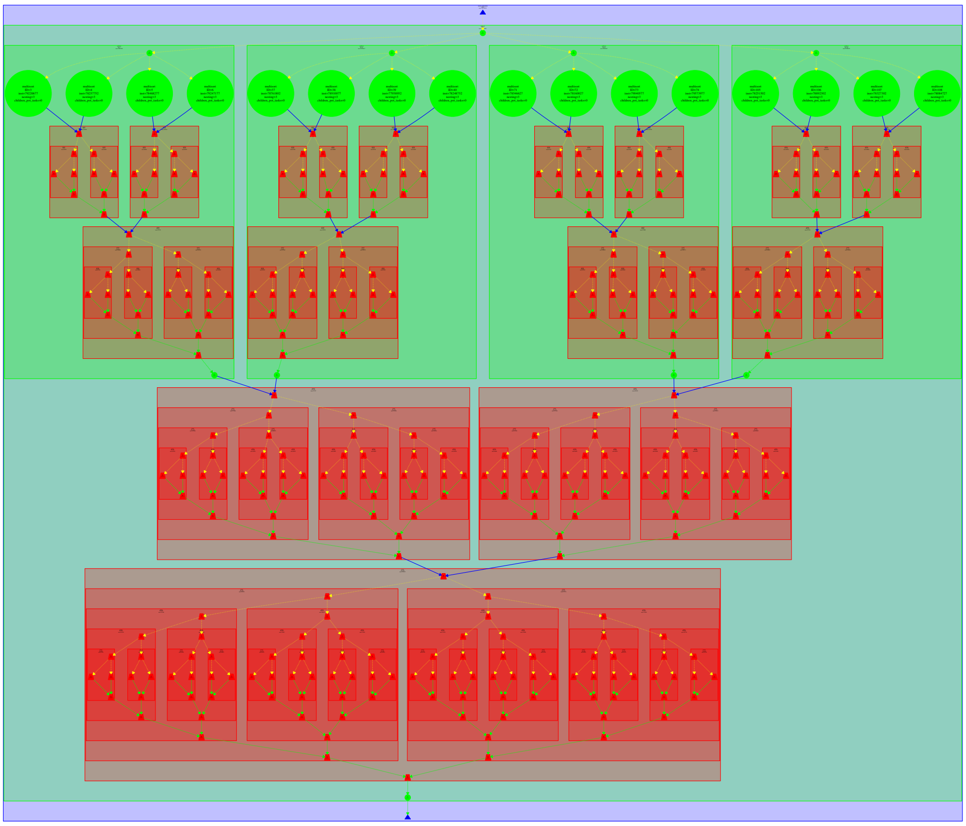


Figure 2: Task dependence graph for tree decomposition

We also generated the task dependence graphs for both strategies (Figures 1 and 2).

Leaf strategy					
Rec level:	1	2	3	4	5
Tasks:	16	32	32	32	32

Table 1: Number of tasks that are generated at each recursion level for leaf strategy

Tree strategy																	
Rec level:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Tasks:	1	1	4	8	16	4	8	16	2	4	8	16	1	2	4	8	16

Table 2: Number of tasks that are generated at each recursion level for tree strategy

Thanks to the task dependence graphs generated, a couple task ordering constraints have been identified. The first one we can observe is that that each merge depends on the previous sorts. And the second one is that the last merge depends on the two previous merges. Those constraints can be enforced using synchronisation constraints such as `taskwait` or `taskgroup` as shown below.

	Leaf strategy		Tree strategy	
Processors	Exec time (ms)	Speed-up	Exec time (ms)	Speed-up
1	1 263 350	1.0000	1 263 350	1.0000
2	631 688	2.0000	631 692	1.9999
4	315 851	3.9998	316 452	3.9922
8	158 381	7.9767	158 824	7.9544
16	79 788	15.8338	79 787	15.8340
32	79 746	15.8422	79 744	15.8426
64	79 746	15.8422	79 744	15.8426

Table 3: Table with the execution time and speed-up as predicted by *Tareador*

In the table 3 we can observe the execution time and speed-up as predicted by *Tareador* for both strategies. We can see that for 1 to 16 processors the speed-up is quite close to the ideal case, but when we arrive to 32 processors the speed-up curve flattens away from the ideal line.

As shown in the table, both strategies (leaf and tree) have no big differences neither in execution time nor in speed-up, but it should be noted that with a low number of processors, the leaf strategy is a bit faster than the tree one, and with many processors the opposite happens. We have to keep in mind however that these results don't take into account the task creation and synchronization overheads and therefore the results are not reliable since tree decomposition creates more tasks but also parallelizes the task creation, which doesn't happen with leaf decomposition.

In this case, the scalability is limited due to the problem size (n=32) which as we have shown before makes it so that there are at most 16 tasks that can be run in parallel at the same time, therefore once we have 16 threads we achieve maximum parallelism and any more threads added can't benefit the execution time. This wouldn't be the case if the problem size was bigger.

2 Parallelization strategies

We studied two approaches to paralellization, leaf and tree task decomposition. For the leaf decomposition, we defined tasks at the last level of the recursion: before calling `basicmerge` and `basicsort`. The code corresponding to this version can be seem in listing 3.

For the tree task decomposition we had to take into account the dependencies between tasks that we found while doing the analysis with *Tareador*. Those are that all tasks to `multisort` have to finish before calling the corresponding `merge`, and that the first two `merge` must finish before the last one. The code can be seen in listing 4.

```
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);
    }
}
```

Listing 3: OpenMP pragmas added for leaf decomposition

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

void multisort(long n, T data[n], T tmp[n], unsigned int depth) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        #pragma omp taskgroup
        {
            #pragma omp task
            multisort(n/4L, &data[0], &tmp[0], depth+1);
            #pragma omp task
            multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
            #pragma omp task
            multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
            #pragma omp task
            multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);
        }

        #pragma omp taskgroup
        {
            #pragma omp task
            merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);
            #pragma omp task
            merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, depth+1);
        }

        #pragma omp task
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);
    } else {
        // Base case
        basicsort(n, data);
    }
}
```

Listing 4: OpenMP pragmas added for tree decomposition

3 Performance evaluation

In figures 3 and 4 we can observe the strong scalability analysis of the leaf and the tree task decomposition respectively. It is clear from the plots that the tree decomposition gives much better results than the leaf approach (which flattens out at around 5 threads). Another fact to notice is that the speed-up of the multi-sort function has really good results, but the whole application does not, this means that there are portions of the code outside of the multisort that are affecting the performance noticeably.

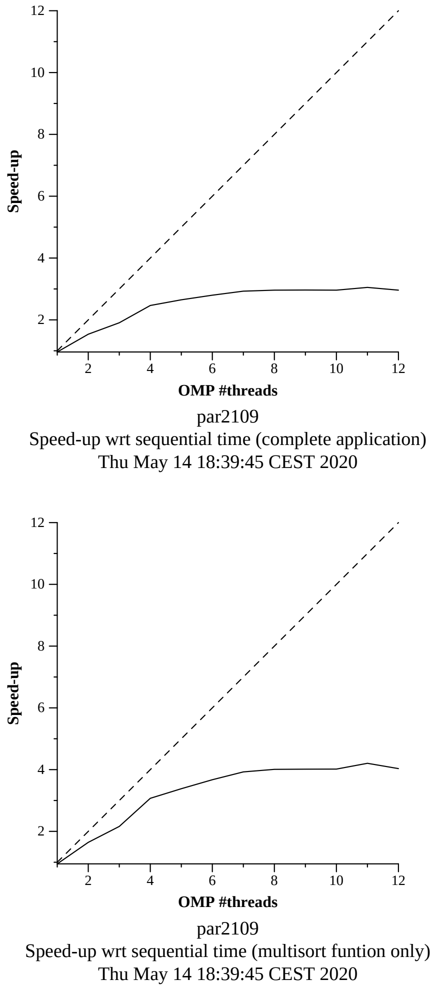


Figure 3: Strong scalability analysis leaf task decomposition

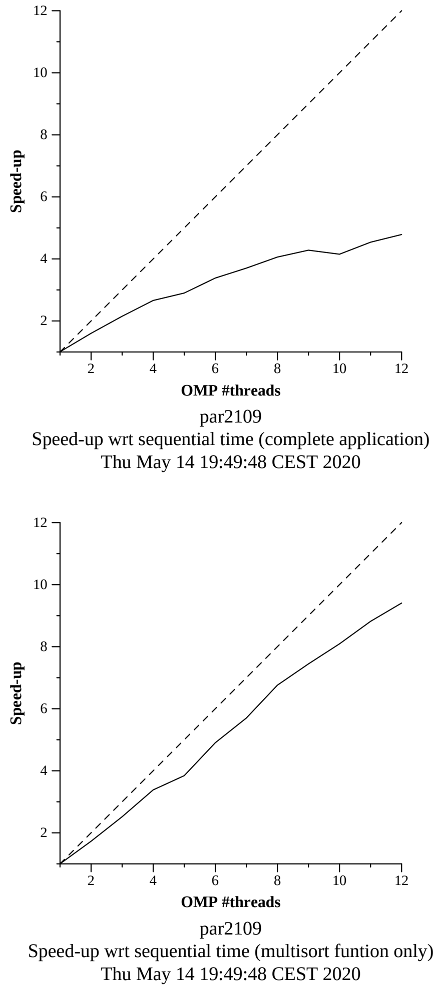


Figure 4: Strong scalability analysis tree task decomposition

3.1 Cut-off

To add a cut-off mechanism based on the recursion level, we added a parameter to the functions `multisort` and `merge` called `depth` to keep track of the depth in the recursion. With this parameter, we can add `final(depth > CUTOFF) mergeable` to the task pragmas to indicate that once the depth is higher than the specified `CUTOFF` the next tasks should not be created. The relevant parts of the code are shown in listing 5.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, unsigned 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) mergeable
        merge(n, left, right, result, start, length/2, depth+1);
        #pragma omp task final(depth > CUTOFF) mergeable
        merge(n, left, right, result, start + length/2, length/2, depth+1);
    }
}

void multisort(long n, T data[n], T tmp[n], unsigned int depth) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        #pragma omp taskgroup
        {
            #pragma omp task final(depth > CUTOFF) mergeable
            multisort(n/4L, &data[0], &tmp[0], depth+1);
            #pragma omp task final(depth > CUTOFF) mergeable
            multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
            #pragma omp task final(depth > CUTOFF) mergeable
            multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
            #pragma omp task final(depth > CUTOFF) mergeable
            multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);
        }

        #pragma omp taskgroup
        {
            #pragma omp task final(depth > CUTOFF) mergeable
            merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);
            #pragma omp task final(depth > CUTOFF) mergeable
            merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L, depth+1);
        }

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

Listing 5: OpenMP pragmas added for tree decomposition with cutoff

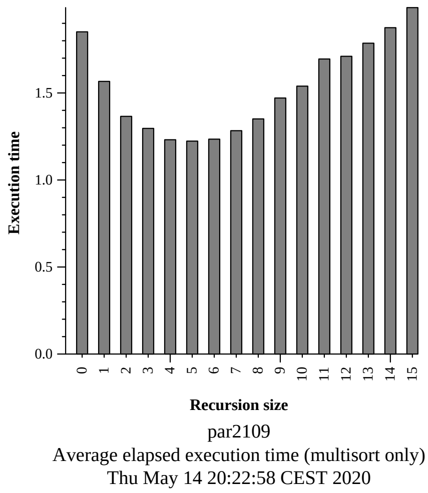


Figure 5: Cut-off 8 processors

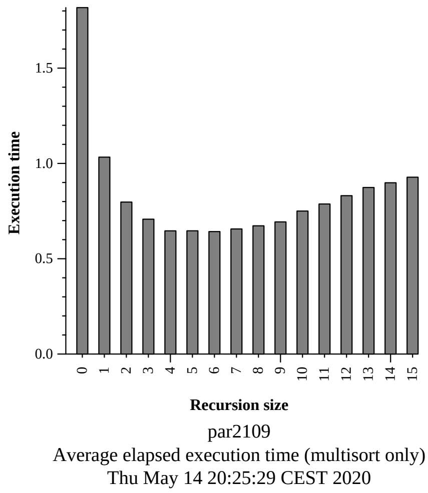


Figure 6: Cut-off 24 processors

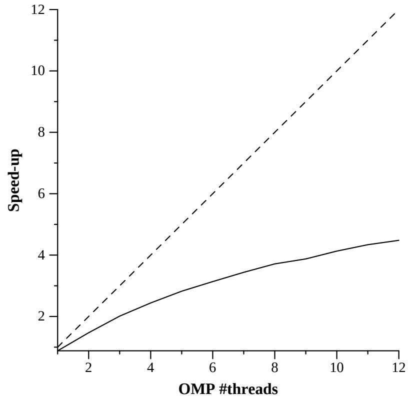
As we can see in figures 5 and 6, the best results are obtained with a recursion level of 5. If we analyze the traces of cutoff with values 0 and 1 with paraver, we can see that the execution time is reduced from cutoff 0 to 1, but the time spent in scheduling and fork/join increases, this explains the results shown in figures 5 and 6: the execution time is reduced with higher cutoff values since there are more tasks that can be executed in parallel, however the overhead of task creation and synchronization at the deeper levels out-weights the benefits of having more tasks.

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	316,232,567 ns	-	10,327,032 ns	8,972,878 ns	575,946 ns	2,778 ns
THREAD 1.1.2	14,403,563 ns	284,372,402 ns	7,307,242 ns	8,814,822 ns	798,174 ns	-
THREAD 1.1.3	15,854,116 ns	284,348,966 ns	5,254,816 ns	9,437,365 ns	839,252 ns	-
THREAD 1.1.4	16,513,105 ns	284,348,789 ns	3,930,154 ns	10,103,504 ns	854,131 ns	-
THREAD 1.1.5	13,957,836 ns	284,318,587 ns	8,110,102 ns	8,517,570 ns	646,839 ns	-
THREAD 1.1.6	12,965,469 ns	285,098,543 ns	9,303,135 ns	7,540,013 ns	584,405 ns	-
THREAD 1.1.7	13,653,350 ns	284,528,841 ns	8,601,592 ns	8,124,916 ns	931,446 ns	-
THREAD 1.1.8	15,726,006 ns	284,660,252 ns	5,633,030 ns	8,884,466 ns	893,468 ns	-
Total	419,306,012 ns	1,991,676,380 ns	58,467,103 ns	70,395,534 ns	6,123,661 ns	2,778 ns
Average	52,413,251.50 ns	284,525,197.14 ns	7,308,387.88 ns	8,799,441.75 ns	765,457.62 ns	2,778 ns

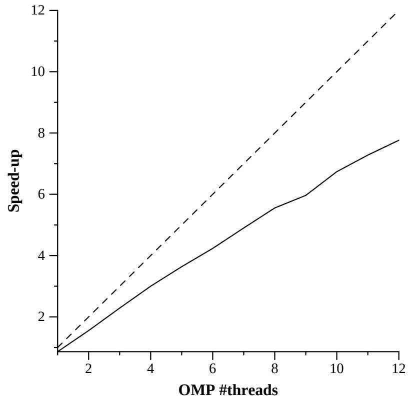
Figure 7: Paraver OMP_state_profile cutoff 0

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	241,729,476 ns	-	5,424,724 ns	11,018,193 ns	632,137 ns	2,853 ns
THREAD 1.1.2	16,532,976 ns	205,136,011 ns	3,613,896 ns	10,627,875 ns	845,557 ns	-
THREAD 1.1.3	17,517,879 ns	205,101,930 ns	1,824,093 ns	11,465,641 ns	744,379 ns	-
THREAD 1.1.4	17,660,460 ns	205,099,959 ns	1,755,464 ns	11,393,588 ns	811,729 ns	-
THREAD 1.1.5	17,124,598 ns	205,135,981 ns	1,779,710 ns	11,870,426 ns	854,440 ns	-
THREAD 1.1.6	15,884,848 ns	205,418,757 ns	3,977,992 ns	10,629,169 ns	788,918 ns	-
THREAD 1.1.7	15,730,494 ns	205,380,836 ns	3,984,932 ns	10,814,483 ns	760,517 ns	-
THREAD 1.1.8	15,046,791 ns	205,901,770 ns	5,583,423 ns	9,378,280 ns	702,982 ns	-
Total	357,227,522 ns	1,437,175,244 ns	27,944,234 ns	87,197,655 ns	6,140,659 ns	2,853 ns
Average	44,653,440.25 ns	205,310,749.14 ns	3,493,029.25 ns	10,899,706.88 ns	767,582.38 ns	2,853 ns

Figure 8: Paraver OMP_state_profile cutoff 1



par2109
Speed-up wrt sequential time (complete application)
Thu May 14 21:56:21 CEST 2020



par2109
Speed-up wrt sequential time (multisort funtion only)
Thu May 14 21:56:21 CEST 2020

Figure 9: Scalability analysis with cutoff=5

3.2 Scalability analysis on more than 12 threads

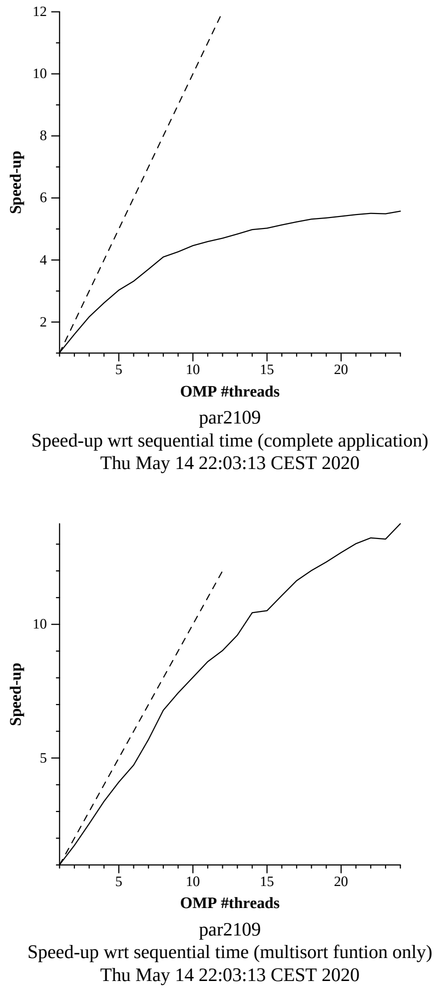


Figure 10: Scalability analysis with more than 12 threads

Figure 10 shows the scalability analysis on boada-4 with up to 24 threads instead of the default of 12, we can see that although boada-4 only has 12 cores, there is still improvement from 12 to 24, this is due that there are 2 threads per core, so the total number of threads that are available on boada-4 is 24.

3.3 Scalability analysis on different boada architectures

In figure 11 we can see the scalability analysis on boada-5 (cuda), interestingly, the speed-up plot obtained is much better than the one we obtained on the other boada nodes, although it has the same number of cores than the boada-2, 3 and 4. With boada-7, shown in figure 12 we can see that the speed-up plot obtained is similar to the ones in boada-4 (in this case we specified `np_MAX=16` since it has 16 cores with only 1 thread per core).

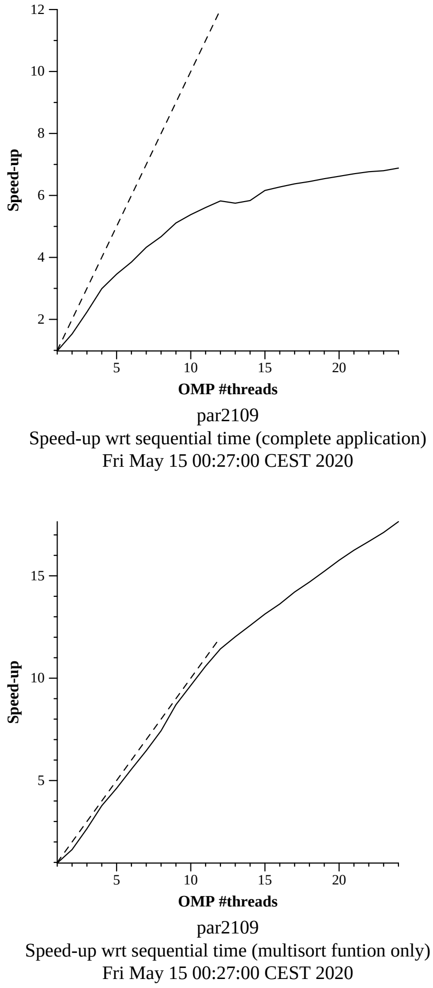


Figure 11: Scalability analysis on boada-5 (cuda)

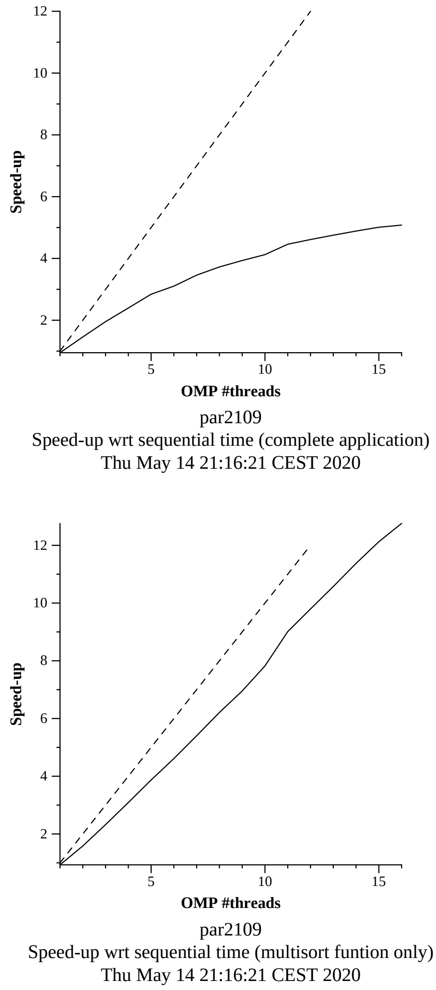


Figure 12: Scalability analysis on boada-7

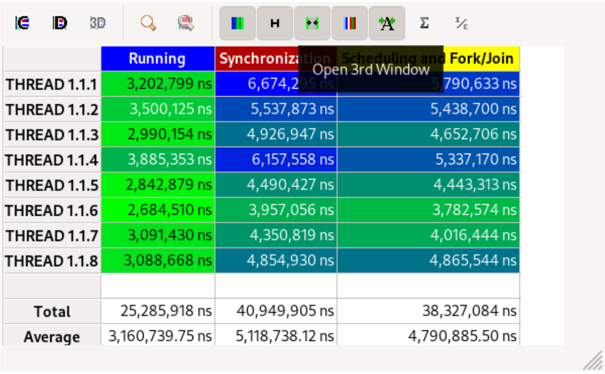
3.4 Task dependencies

Instead of using `taskgroup` and `taskwait` to manage the dependencies of the tasks we explicitly declared the dependencies between them. We specified the initial position of the array for which the tasks depended as show in listing 6.

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length, unsigned 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) mergeable
        merge(n, left, right, result, start, length/2, depth+1);
        #pragma omp task final(depth > CUTOFF) mergeable
        merge(n, left, right, result, start + length/2, length/2, depth+1);
        #pragma omp taskwait
    }
}

void multisort(long n, T data[n], T tmp[n], unsigned int depth) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        #pragma omp task final(depth > CUTOFF) mergeable depend(out: data[0])
        multisort(n/4L, &data[0], &tmp[0], depth+1);
        #pragma omp task final(depth > CUTOFF) mergeable depend(out: data[n/4L])
        multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);
        #pragma omp task final(depth > CUTOFF) mergeable depend(out: data[n/2L])
        multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);
        #pragma omp task final(depth > CUTOFF) mergeable depend(out: data[3L*n/4L])
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L], depth+1);

        #pragma omp task final(depth > CUTOFF) mergeable depend(in: data[0], data[n/4L]) depend(out: tmp[0])
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);
        #pragma omp task final(depth > CUTOFF) mergeable 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, depth+1);

        #pragma omp task final(depth > CUTOFF) mergeable depend(in: tmp[0], tmp[n/2L])
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);

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

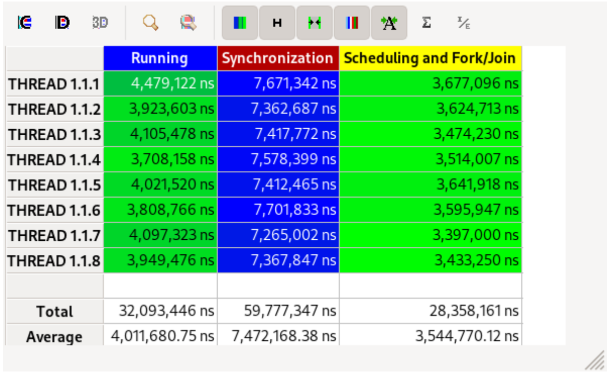
Listing 6: OpenMP pragmas added for tree decomposition with task dependencies

In figures 13 and 14 we can see that the scheduling and fork/join time are reduced when using dependencies, but the time spent in task synchronization increases noticeably, and in fact the total execution time is higher than without the dependencies.



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	3,202,799 ns	6,674,295 ns	5,790,633 ns
THREAD 1.1.2	3,500,125 ns	5,537,873 ns	5,438,700 ns
THREAD 1.1.3	2,990,154 ns	4,926,947 ns	4,652,706 ns
THREAD 1.1.4	3,885,353 ns	6,157,558 ns	5,337,170 ns
THREAD 1.1.5	2,842,879 ns	4,490,427 ns	4,443,313 ns
THREAD 1.1.6	2,684,510 ns	3,957,056 ns	3,782,574 ns
THREAD 1.1.7	3,091,430 ns	4,350,819 ns	4,016,444 ns
THREAD 1.1.8	3,088,668 ns	4,854,930 ns	4,865,544 ns
Total	25,285,918 ns	40,949,905 ns	38,327,084 ns
Average	3,160,739.75 ns	5,118,738.12 ns	4,790,885.50 ns

Figure 13: Paraver trace with taskgroup and taskwait



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	4,479,122 ns	7,671,342 ns	3,677,096 ns
THREAD 1.1.2	3,923,603 ns	7,362,687 ns	3,624,713 ns
THREAD 1.1.3	4,105,478 ns	7,417,772 ns	3,474,230 ns
THREAD 1.1.4	3,708,158 ns	7,578,399 ns	3,514,007 ns
THREAD 1.1.5	4,021,520 ns	7,412,465 ns	3,641,918 ns
THREAD 1.1.6	3,808,766 ns	7,701,833 ns	3,595,947 ns
THREAD 1.1.7	4,097,323 ns	7,265,002 ns	3,397,000 ns
THREAD 1.1.8	3,949,476 ns	7,367,847 ns	3,433,250 ns
Total	32,093,446 ns	59,777,347 ns	28,358,161 ns
Average	4,011,680.75 ns	7,472,168.38 ns	3,544,770.12 ns

Figure 14: Paraver trace with dependencies

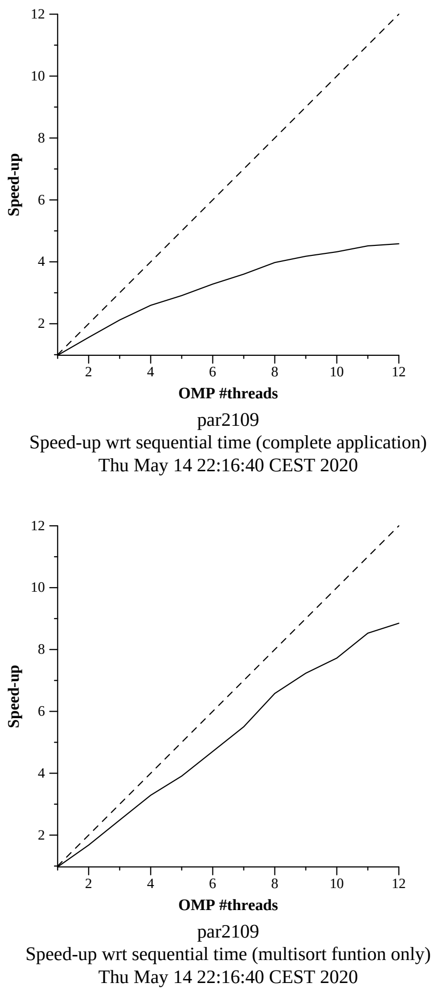


Figure 15: Strong scalability analysis with dependencies

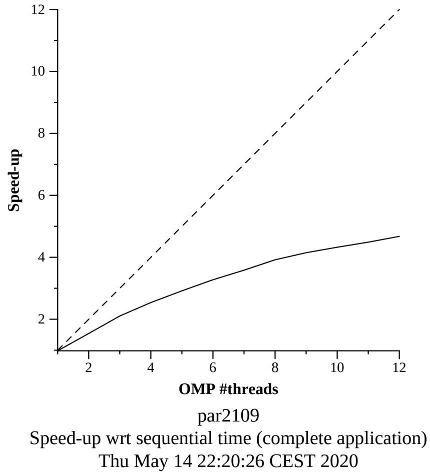
3.5 Parallelizing tmp and data initialization

To paralelize the tmp and data vector initizalization, we simply added a `pragma omp parallel for` clause and modified the for loops that initialized the vectors as shown in listing 7.

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

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

Listing 7: OpenMP pragmas added to paralelize the tmp and data vector initialization

With this optimization we obtained the scalability plot shown in figure 16. There is no noticeable improvement on the speed-up plot of the complete application, this is not what we expected. It is possible that the task creation creates unnecessary overheads (we could have specified a bigger grainsize), or that the compiler optimizes the initialization of the tmp array and by introducing OpenMP directives this optimization is no longer performed.



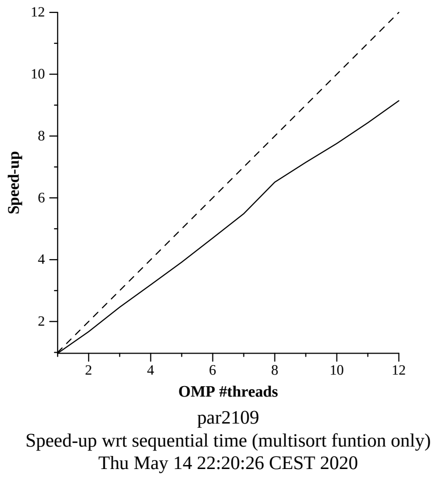


Figure 16: Strong scalability analysis with tmp and data initialization

4 Conclusions

We have seen that the best results are obtained with tree task decomposition using a cutoff mechanism (specifically a cutoff at depth 5) and that adding task dependencies instead of using `taskgroup` and `taskwait` constructs does not give noticeable benefits. Moreover, a big part of the application execution time is spent on data initialization and despite our approach to parallelize that section we could not manage to obtain a significant enough speed-up result for the whole application.