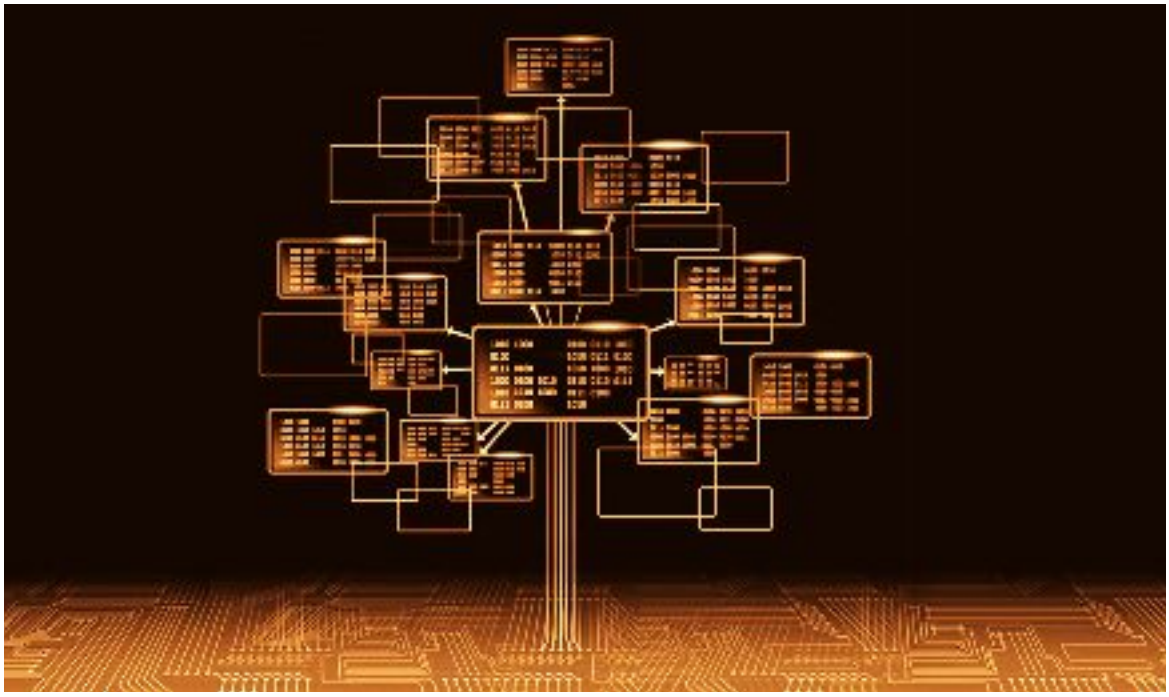
LAB 4

Divide and Conquer parallelism with OpenMP: Sorting



Antonio J. Cabrera

Paul Gazel-Anthoine

PAR1401

Spring 2018-19

TABLE OF CONTENTS

INTRODUCTION

In this laboratory, we study the behavior of a *mergesort* algorithm implementation where, for each level of recursion, the working section of the vector is split in four.

After studying the task decomposition and analysing the predictions of Tareador, we will implement different parallelization mechanisms in order to improve the execution time and obtain better strong scalability projections.

The parallelization mechanisms will have to consider tightly dependant variables that travel deep in the recursion tree. The structures we will use for the program parallelization will be the Leaf and the Tree structures.

TASK DECOMPOSITION ANALYSIS FOR MERGESORT

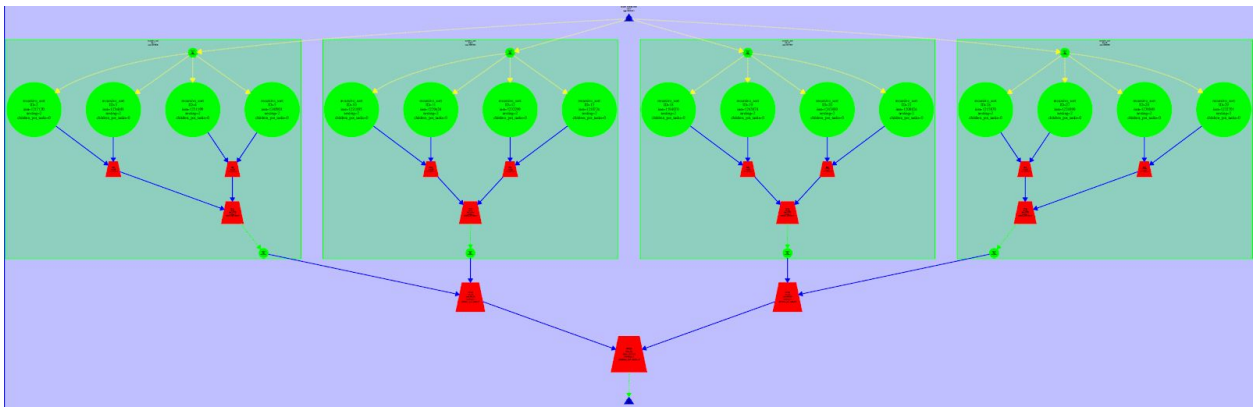


Figure 1: *Tareador* dependence graph v1 for *Multisort (-n 32 -s 2 -m 2)*

As we can see in figure 1, this small example has two levels of recursion, because each level divides the number of elements of a task by 4. In this chart, the 16 green circles correspond to groups of two elements that are sorted with the *basicsort* function. The red shapes represent the *merge* tasks.

In figures 2 and 3 we can see a more in-depth Tareador analysis by adding a task for *basicsort*, which is called when the number of elements in the current task cannot be subdivided in 4 different tasks.

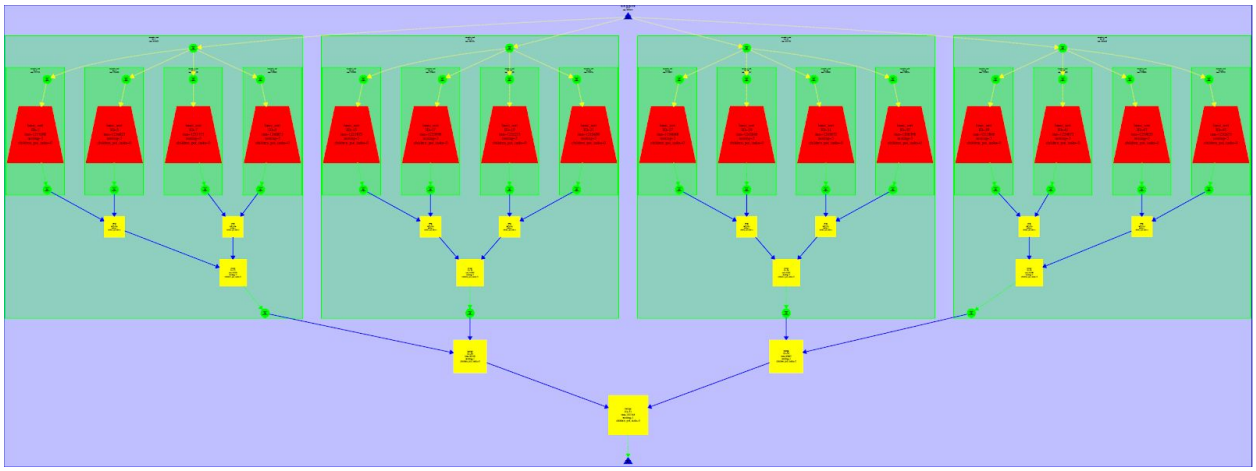


Figure 2: *Tareador* dependence graph v2 for *Multisort (-n 32 -s 2 -m 2)*

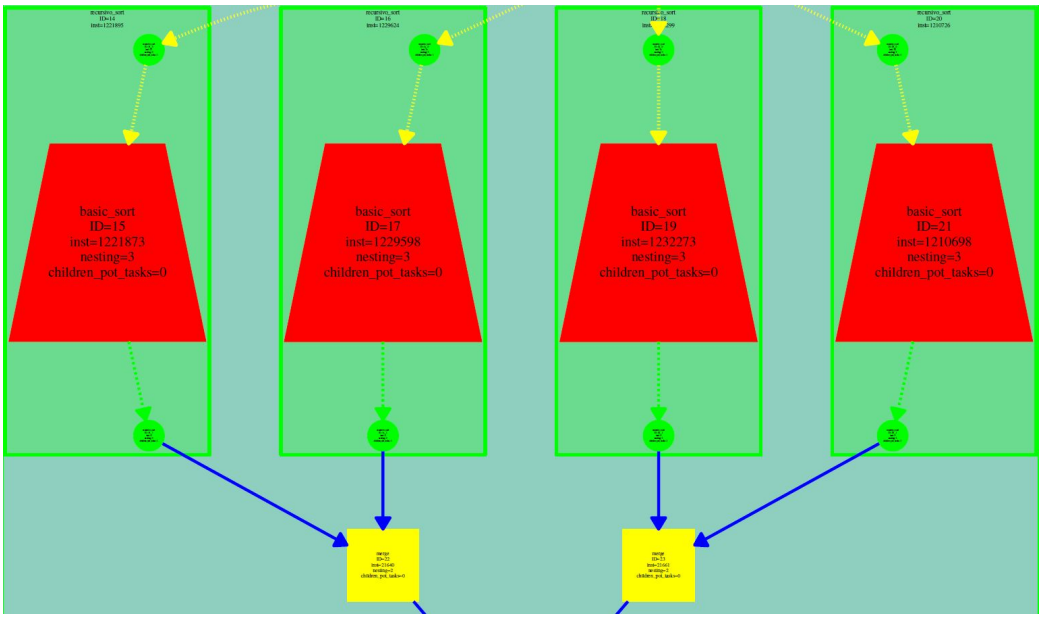


Figure 3: Zoom for *Tareador* dependence graph v2 for *Multisort (-n 32 -s 2 -m 2)*

By simulating executions with Tareador, we have a look at the time evolution with varying number of threads (table 1).

It is apparent that we will be able to heavily increase performance by using more resources, if parallelization is well implemented. We see that the improvement gets stuck at 16 threads, but that's because the test uses only 32 elements and for each comparison you need two elements. In this case, having more resources means idle resources, but that will change with bigger vectors. The speed-up looks quite promising.

#Threads	Time (ms)	Speed-up
1	20.33	1
2	10.32	1.97
4	5.31	3.83
8	2.84	7.16
16	1.58	12.87
32	1.58	12.87
64	1.58	12.87

Table 1: Time and speed-up evolution predicted with Tareador: Multisort (*-n 32 -s 2 -m 2*)

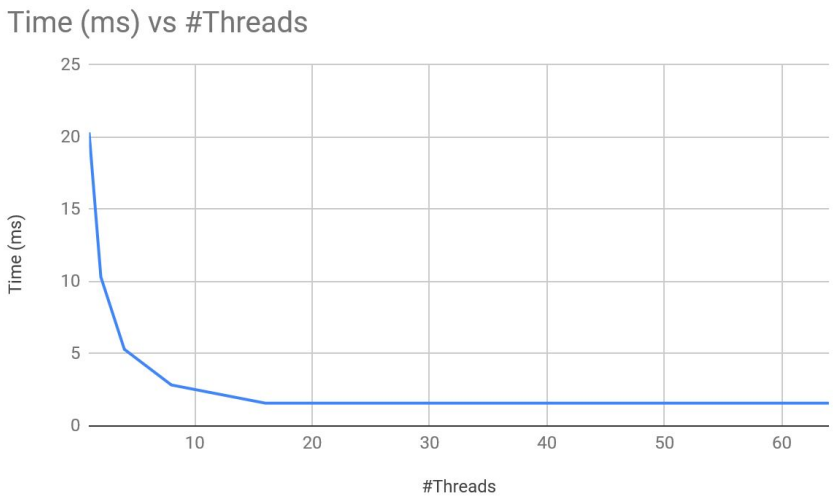


Figure 4: Time evolution predicted with Tareador: Multisort (*-n 32 -s 2 -m 2*)

In figure 4, we can see very clearly what we stated before, time cannot be lowered by using more than 16 processors, so there is no point in using more resources.
By looking at figure 5, we see that using less than 10 threads gives more efficient executions: as we increase the number of processors, efficiency decreases (understanding efficiency as the fraction of time for which a processor is doing useful work).

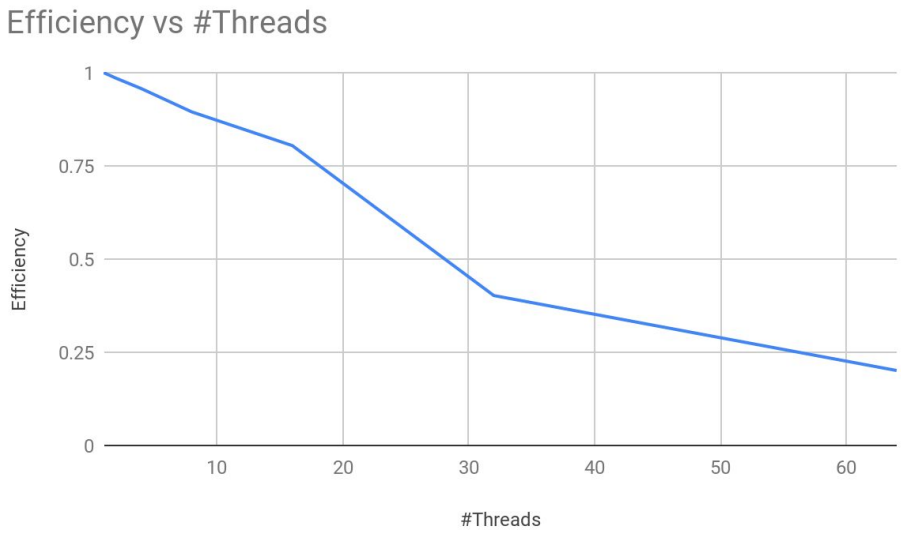


Figure 5: Efficiency evolution prediction with Tareador: Multisort (*-n 32 -s 2 -m 2*)

SHARED-MEMORY PARALLELIZATION WITH OPENMP TASKS

In this part of the laboratory, we explore the different structures of parallelization for this recursive algorithm, the Leaf structure and the Tree structure.

Code for the LEAF version

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
      (...)
      #pragma omp task
      basicmerge(n, left, right, result, start, length);
      (...)
}

void multisort(long n, T data[n], T tmp[n]) {
     (...)
      #pragma omp task
      basicsort(n, data);
     (...)
}
```

Code for the TREE version

```
void merge(long n, T left[n], T right[n], T result[n*2], long start, long length) {
      (...)
      #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]) {
     (...)
     #pragma omp taskgroup
     {
          #pragma omp task
          multisort(n/4L, &data[0], &tmp[0]);
          (...)
     }
     #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);
     (...)
}
```

As we can see in table 2, the Tree version executes approximately 1.65 times faster than the Leaf version. As we know from previous sessions, finer granularity can often bring overheads with an excesses of task creation and synchronization.

Version	Time (s)
Leaf	1.558502
Tree	0.949749

Table 2: Multisort execution time (*-n 32768 -s 1 -m 1*)

PARAVER ANALYSIS

For each of the structures mentioned above, we have extracted the trace and exported some useful images to study the behavior of the different versions.

In figures 6 through 9 we can see that most of the program is sequential and all tasks are created by the main thread.

Figure 6: Paraver of leaf version, original view (*-n 8192 -s 1 -m 1*)

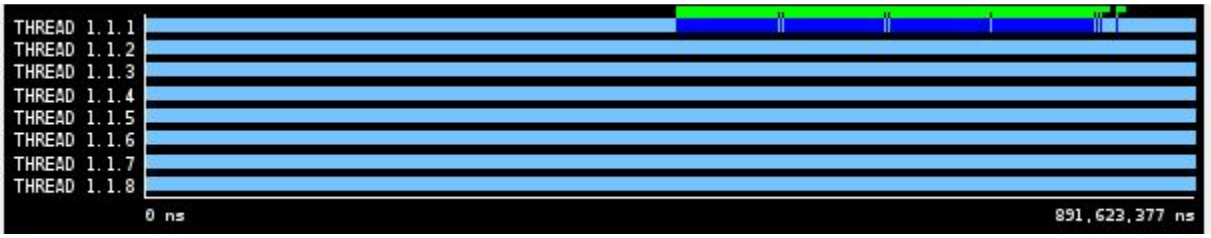


Figure 7: Paraver of leaf version, taskwait construct (*-n 8192 -s 1 -m 1*)

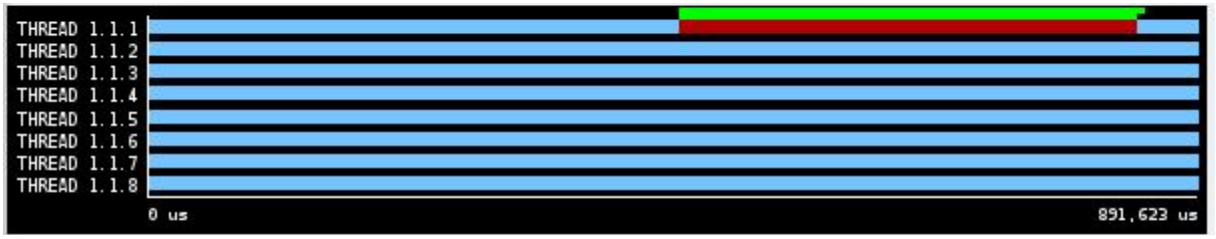


Figure 8: Paraver of leaf version, task instantiation (*-n 8192 -s 1 -m 1*)

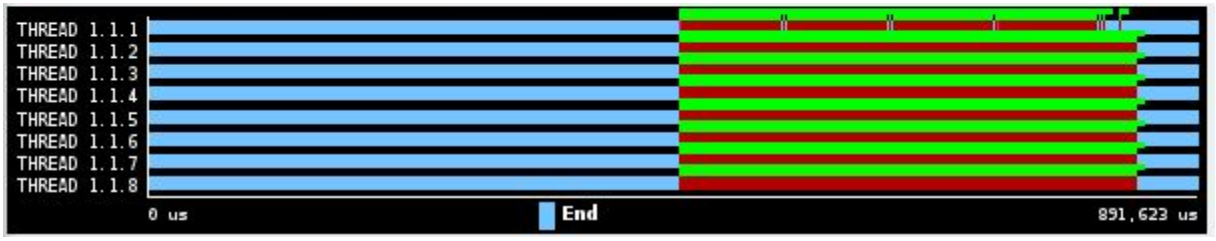


Figure 9: Paraver of leaf version, task execution (*-n 8192 -s 1 -m 1*)

In figures 10 through 13, we include a zoom to better see what goes on in each thread. As we can see in figure 13, synchronization is intermittently present (blue color) and tasks are somewhat small, so plenty of time is spent managing tasks, not solving the problem.

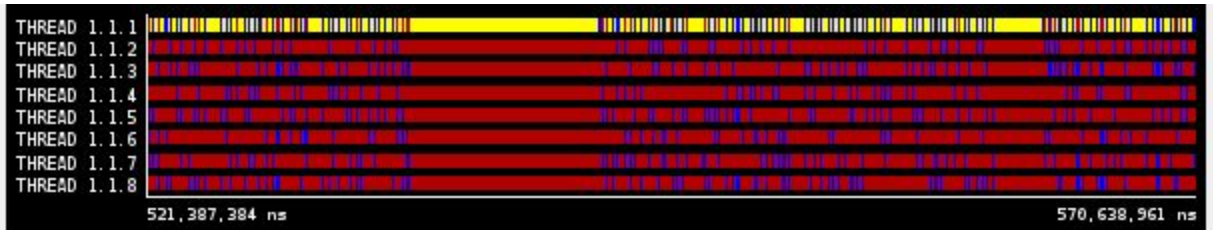


Figure 10: Paraver zoom of leaf version, original view (*-n 8192 -s 1 -m 1*)

Figure 11: Paraver zoom of leaf version, taskwait construct (*-n 8192 -s 1 -m 1*)

Figure 12: Paraver zoom of leaf version, task instantiation (*-n 8192 -s 1 -m 1*)

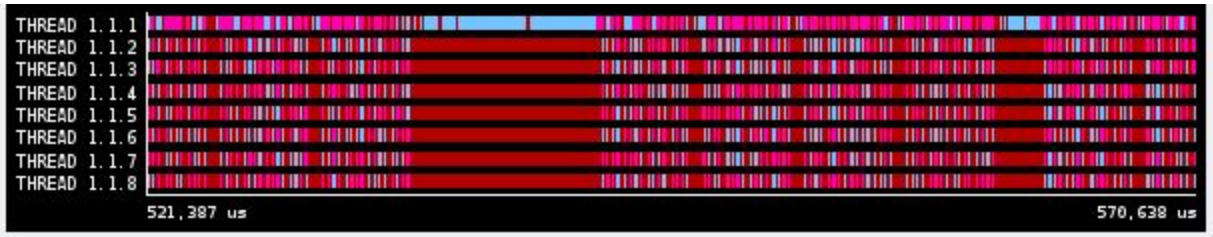


Figure 13: Paraver zoom of leaf version, task execution (*-n 8192 -s 1 -m 1*)

As per the Tree version, in figures 14 through 17 we can see very quickly that the program is faster, because the sequential part of the code uses more space relative to the parallel code. In this case, as figure 16 indicates, all threads create tasks, not just the main thread.

Figure 14: Paraver of tree version, original view (*-n 8192 -s 1 -m 1*)

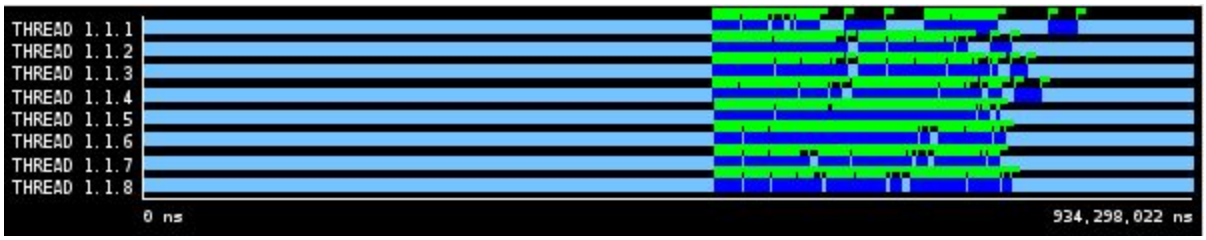


Figure 15: Paraver of tree version, taskwait construct (*-n 8192 -s 1 -m 1*)

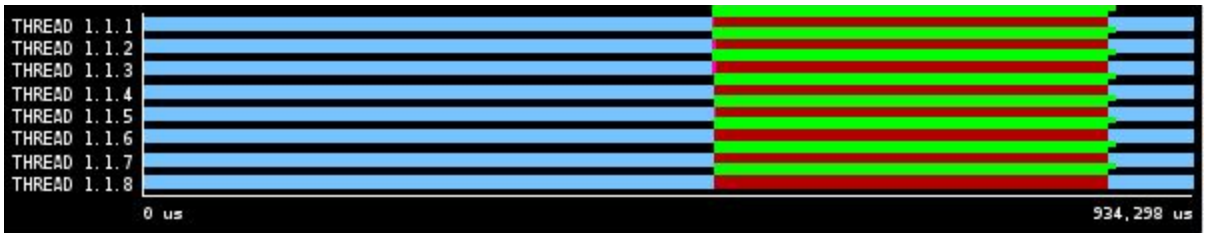


Figure 16: Paraver of tree version, task instantiation (*-n 8192 -s 1 -m 1*)

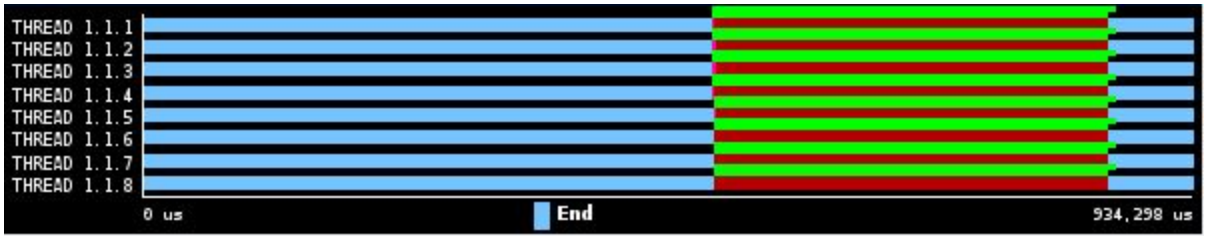


Figure 17: Paraver of tree version, task execution (*-n 8192 -s 1 -m 1*)

In figures 18 to 21, we see a close-up of the execution. Unsurprisingly, blue color is barely noticeable. Task management is not a huge obstacle and thus the execution is faster.

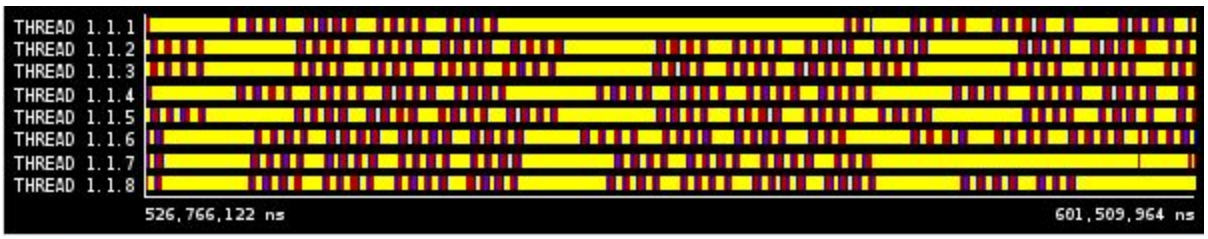


Figure 18: Paraver zoom of tree version, original view (*-n 8192 -s 1 -m 1*)

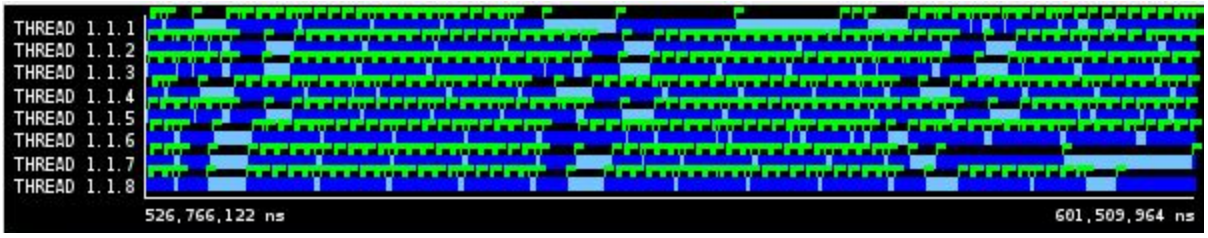


Figure 19: Paraver zoom of tree version, taskwait construct (*-n 8192 -s 1 -m 1*)

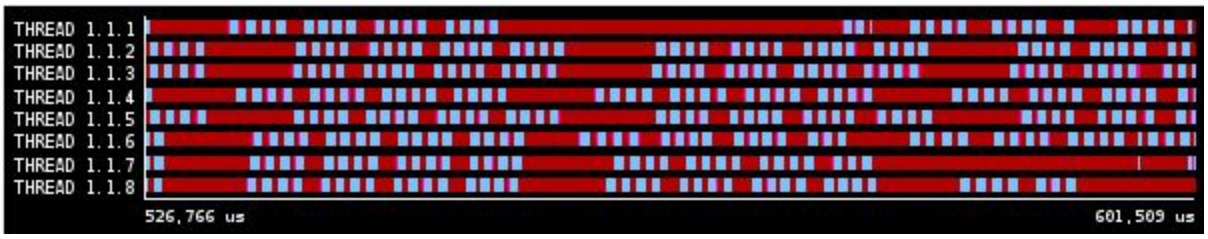


Figure 20: Paraver zoom of tree version, task instantiation (*-n 8192 -s 1 -m 1*)

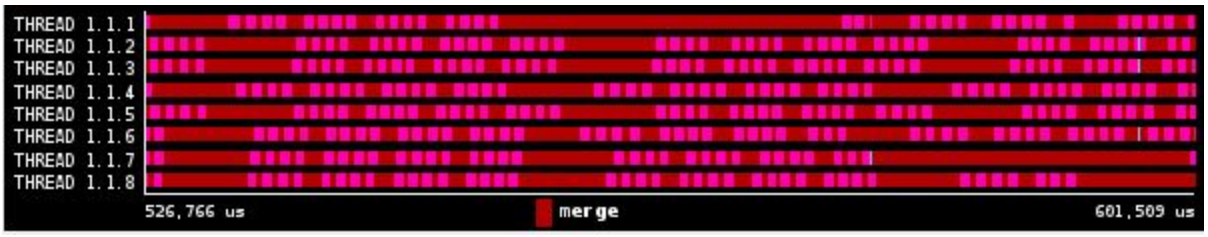


Figure 21: Paraver zoom of tree version, task execution (*-n 8192 -s 1 -m 1*)

STRONG SCALABILITY ANALYSIS

As we could predict looking at the flaws of previous executions, the Leaf version has bad strong scalability (figure 22). For more than 4 threads, the increase in performance is almost negligible. However, the Tree version looks very good, almost ideal, for the multisort function (figure 23). The main thing that prevents the Tree version from being ideal is dependencies, which is something that cannot be avoided in this case. Anyway, we will try different optimizations in some other experiments.

By looking at the program as a whole, the clear limitation is the initialize functions (initialize and clear), plus the “check” function that verifies that the array is sorted.

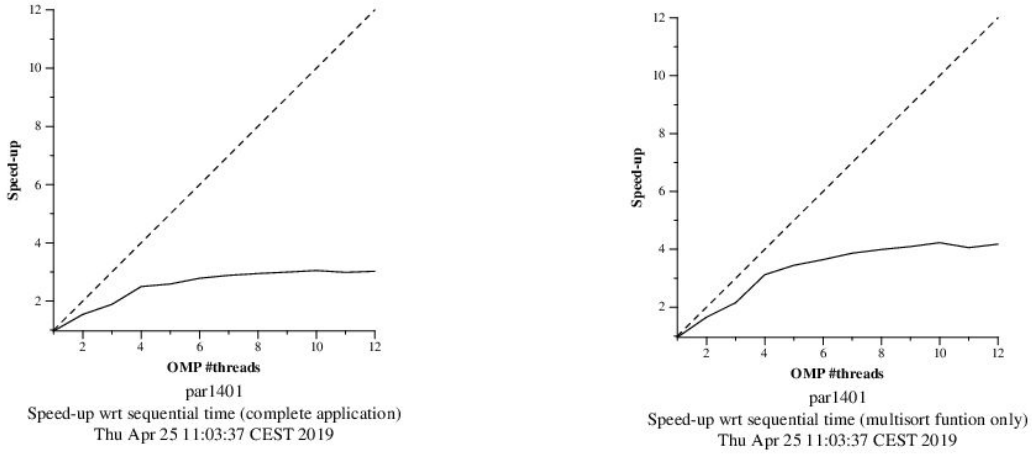


Figure 22: Strong scalability plots. Multisort-Leaf (max 12 threads, *-n 32768 -s 1 -m 1 -c 5*)

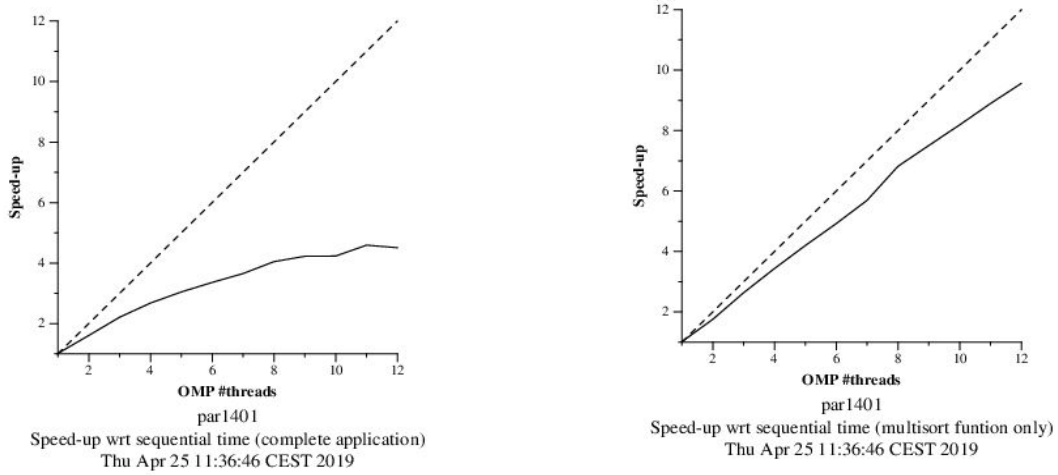


Figure 23: Strong scalability plots. Multisort-Tree (max 12 threads, *-n 32768 -s 1 -m 1 -c 5*)

TASK CUT-OFF MECHANISM

Now, we will explore a possible optimization, the cut-off mechanism, to reduce the number of tasks created by avoiding going to deeper levels of recursion if it does not improve the execution time. This way we won't generate the tasks that imply more work creating and synchronizing them.

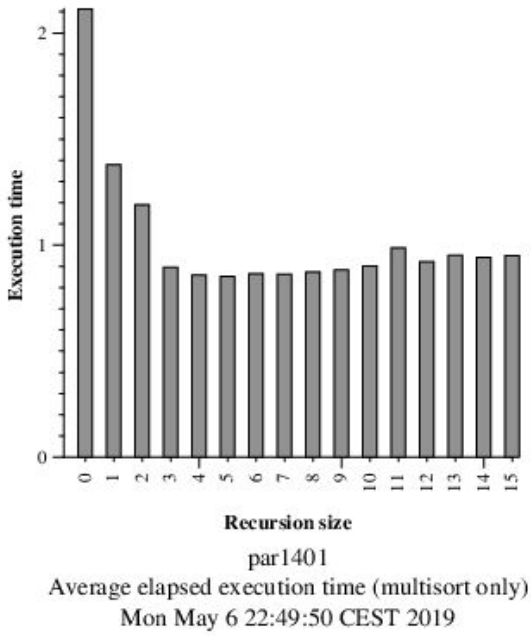


Figure 24: Execution time for different cut-off values to control the maximum recursion level and number of tasks, Multisort-Tree-Cutoff with 8 threads (*-n 32768 -s 1 -m 1*)

After implementing the cut-off mechanism to our Tree version, we found that for the optimal cutoff value is 4 or 5, with the best time accomplished being 0.870101 seconds. So, for the strong scalability study, we worked with 5 recursion levels.

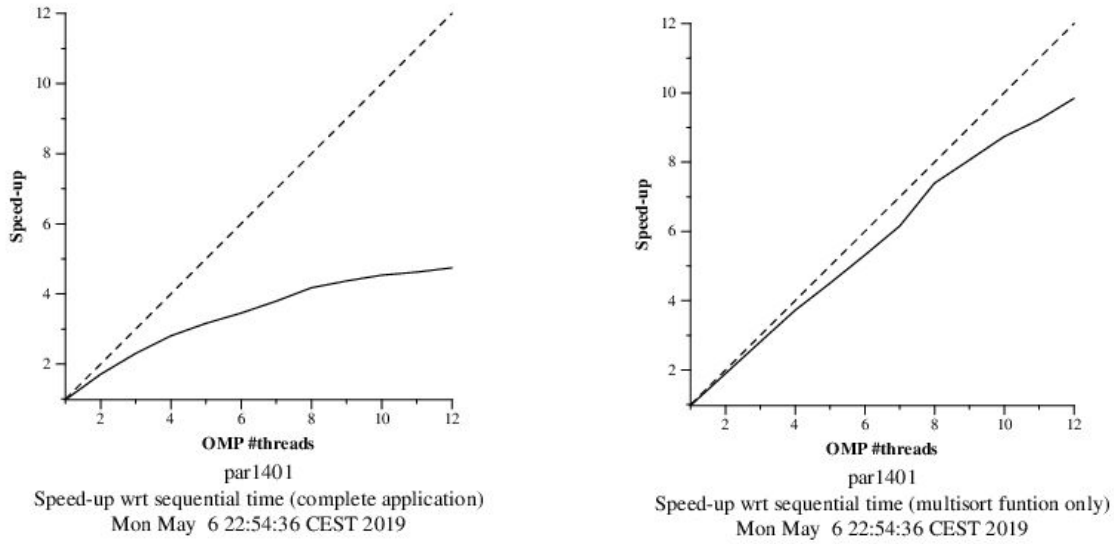


Figure 25: Strong scalability plots. Multisort-Tree-Cutoff (max 8 threads, *-n 32768 -s 1 -m 1 -c 5*)

After analyzing the scalability of the Tree version with Cut-off mechanism, we can see a small upgrade regarding multisort function relative to the previous version, but the problem regarding the scalability of the whole program is not solved yet.

(OPTIONAL 1: *Scalability analysis for the Tree version in all boada nodes*)

When executed in the other boada nodes, we get slightly different results. For example, the scalability analysis of the Tree version shows that, for boada-5 and a maximum of 12 threads tested (figure 26), the speedup obtained for the multisort function is ideal. However, in boada 6-8, with a maximum of only 8 threads tested (figure 27), we could say that it is slightly worse. In the three cases, the strong scalability is considerably good anyway.

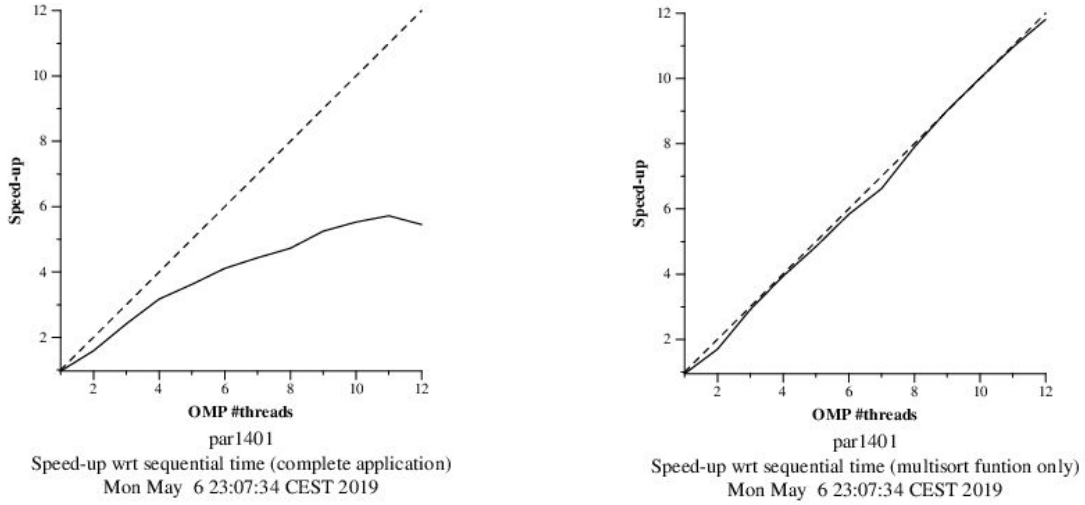


Figure 26: Strong scalability in boada-5. Multisort-Tree (max 12 threads, *-n 32768 -s 1 -m 1*)

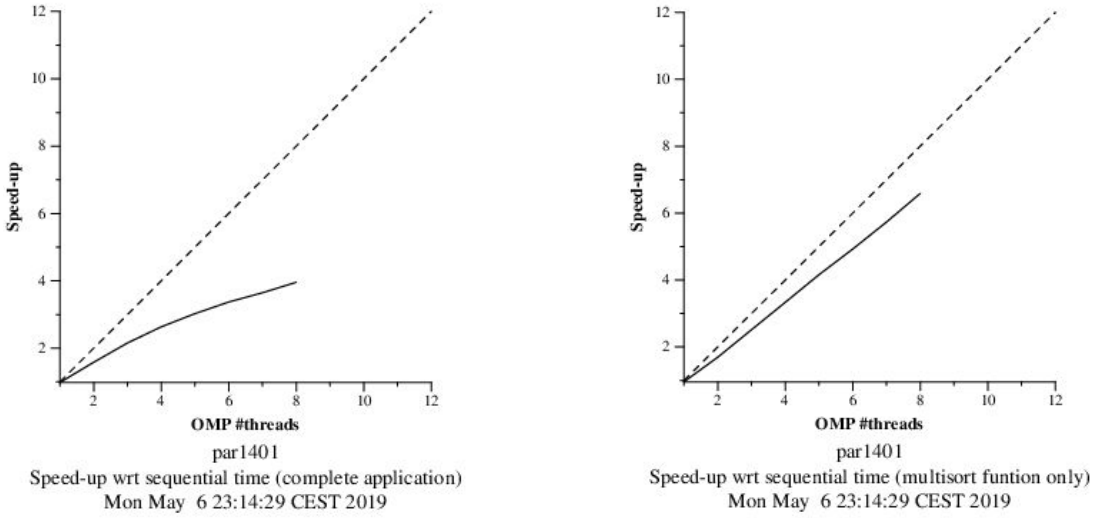


Figure 27: Strong scalability in boada 6 to 8. Multisort-Tree (max 8 threads, *-n 32768 -s 1 -m 1*)

(OPTIONAL 2: *Complete parallelization of the Tree version*)

Code for the optimized TREE version

```
static void initialize(long length, T data[length]) {
   int np = omp_get_num_threads();
   #pragma omp parallel for
   for (long i = 0; i < length; i++) {
      if (i%(length/np) == 0) {
         data[i] = rand();
      } else {
         data[i] = ((data[i-1]+1) * i * 104723L) % N;
      }
   }
}

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

For this implementation we had to think of a smart way of initializing a truly random array. The previous version included a highly dependant loop that did not allow any parallelization.

The obvious change was to generate a random number every time, but we found out that the 'rand()' function takes a very long time, so even when we made a parallel version of it, the execution time was multiplied by 7.

Finally, we came up with the idea of splitting the array in *np* blocks, where *np* is the number of threads available. Thus, only calling the 'rand()' function *np* times, iterating from them in the same way the original version did and parallelizing the loop gave us a speedup of 19 for initialization.

STRONG SCALABILITY & PARAVER ANALYSIS

After parallelizing the two initial functions, we can see a performance improvement regarding the overall execution time of the program. The strong scalability of the whole program has improved a lot (figure 28) and through the paraver analysis we see that execution times of the "initialize" and "clear" functions have decreased dramatically.

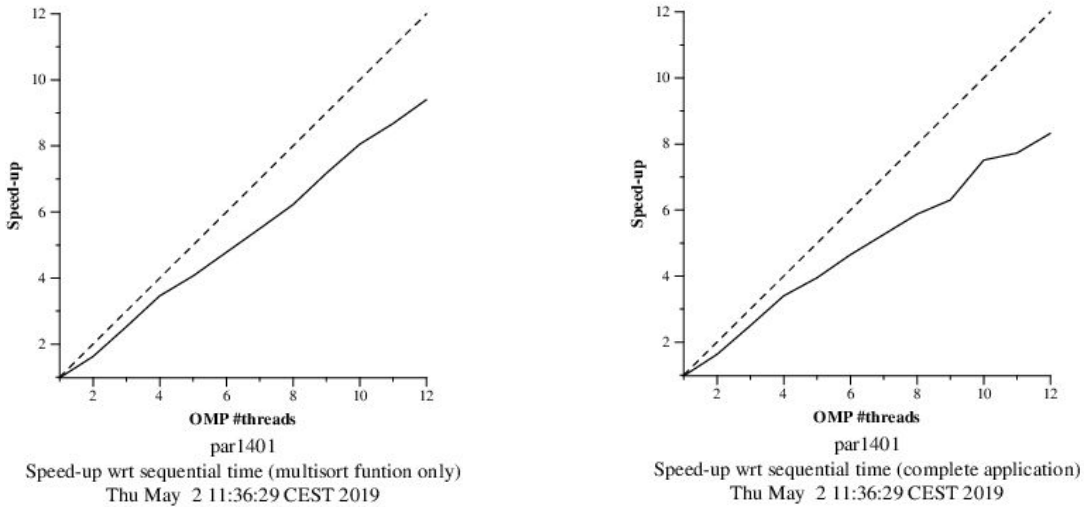


Figure 28: Strong scalability. Optimized Multisort-Tree (max 12 threads, *-n 32768 -s 1 -m 1*)

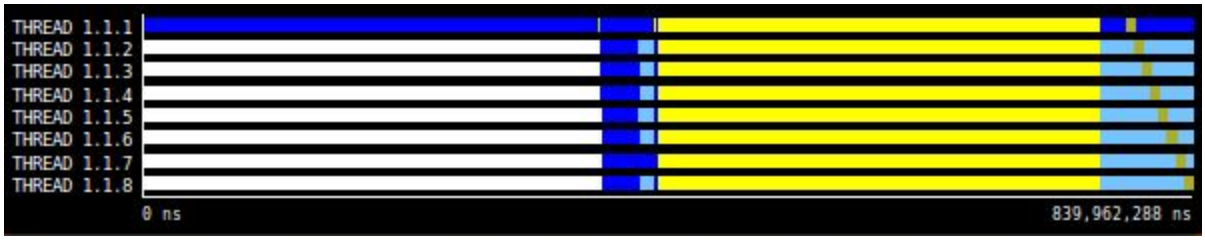


Figure 29: Paraver of optimized tree version, original view (*-n 8192 -s 1 -m 1*)

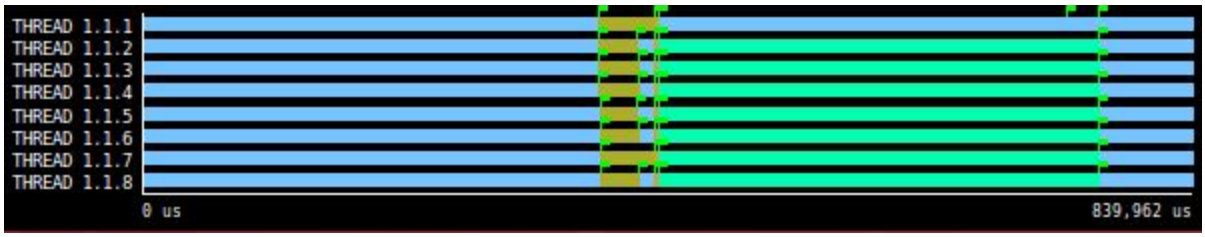


Figure 30: Paraver of optimized tree version, parallel functions (*-n 8192 -s 1 -m 1*)

If we now look at figure 14, we can see a very thin yellow line in the main thread that is the point where the initialize function starts. If we compare the distance between that line and the start of the parallelized multisort and the distance we can see in figure 29, we can immediately see that the time spent in initialization has decreased a lot.

Furthermore, the sequential part of the code represents a smaller percentage of the bar, even though the Tree version used to be dominated by the sequential part (figure 14).

Figure 31: Paraver zoom of optimized tree version, original view (*-n 8192 -s 1 -m 1*)

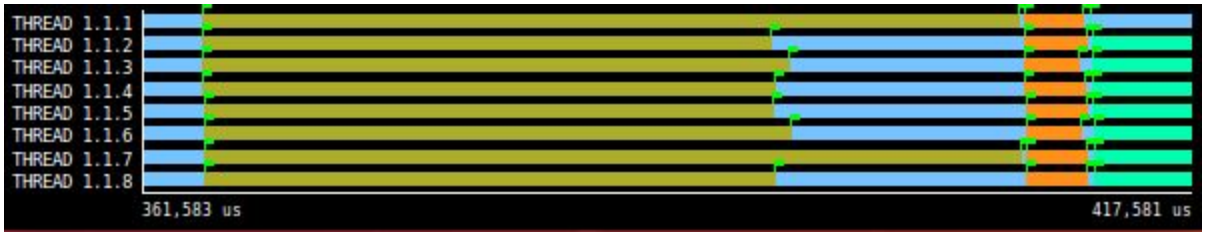


Figure 32: Paraver zoom of optimized tree version, parallel functions (*-n 8192 -s 1 -m 1*)

USING OPENMP TASK DEPENDENCIES

Another way to implement the tree version is using the *depend* directive to explicitly indicate data dependency between tasks to reduce the number of taskwait/taskgroup directives used. This way we can lower overheads. After executing the strong scalability analysis, we see that this change doesn't seem to bring any significant improvement.

This is probably due to the unavoidable dependency between function calls. The structure of the mergesort requires for the smaller arrays to be sorted before they can be merged into a bigger one. Whether we respect the dependencies with taskwait directives or using explicit variable dependencies, the bottleneck here that will not allow for ideal strong scalability is the fact that some of the execution time will have to consist of smaller arrays being sorted while the bigger array waits for it to finish.

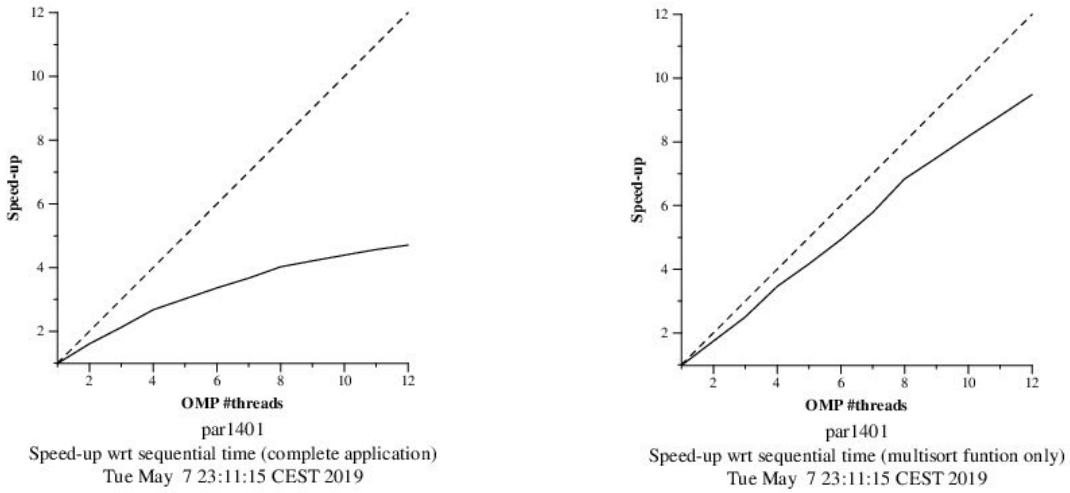


Figure 33: Strong scalability. Multisort-Tree-Dependency (max 12 threads, *-n 32768 -s 1 -m 1*)

After extracting the trace of the Tree version implemented with dependencies, we can see that the program execution matches the original tree version and we don't see any big difference.

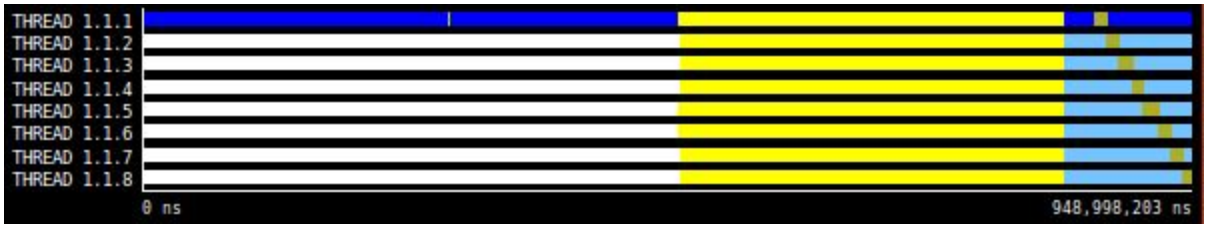


Figure 34: Paraver of tree version with dependencies, original view (*-n 8192 -s 1 -m 1*)

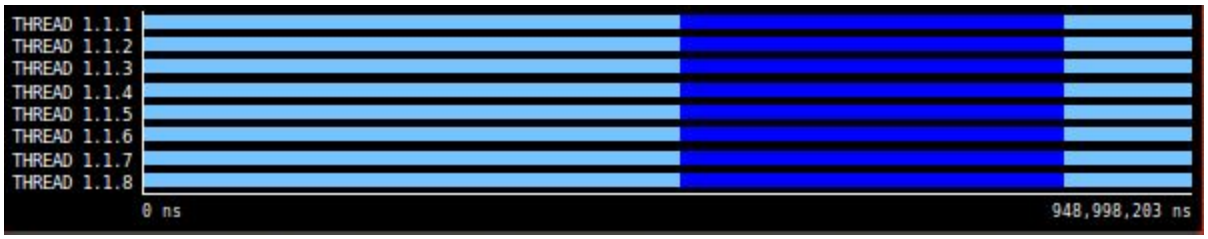


Figure 35: Paraver of tree version with dependencies, taskwait construct (*-n 8192 -s 1 -m 1*)

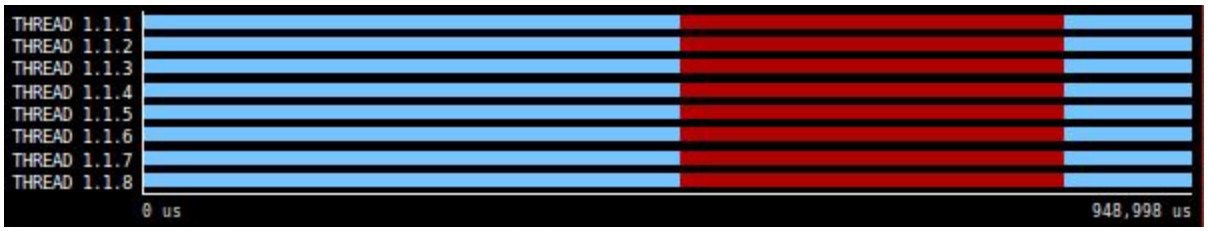


Figure 36: Paraver of tree version with dependencies, task instantiation (*-n 8192 -s 1 -m 1*)

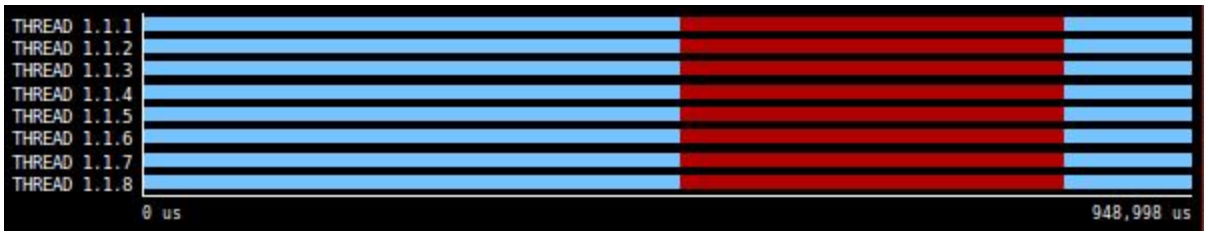


Figure 37: Paraver of tree version with dependencies, task execution (*-n 8192 -s 1 -m 1*)

In figures 38 to 41, we include a zoom to better observe the behaviour of the program, and the result obtained is identical to the zoom realized on the paraver obtained with the original implementation of the Tree version (figures 18 to 21).

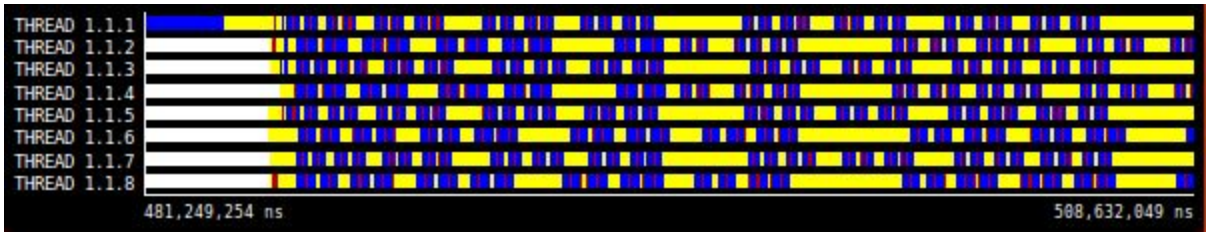


Figure 38: Paraver zoom of tree version with dependencies, original view (*-n 8192 -s 1 -m 1*)

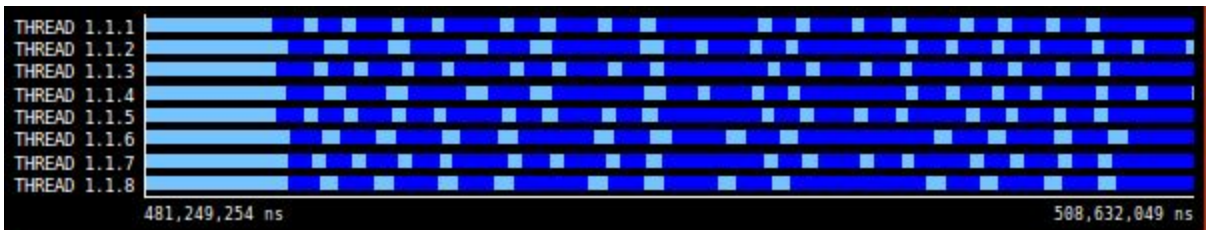


Figure 39: Paraver zoom of tree version with dependencies, taskwait construct (*-n 8192 -s 1 -m 1*)

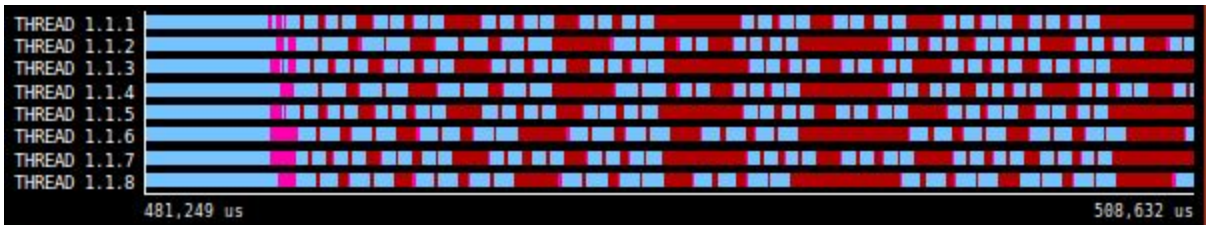


Figure 40: Paraver zoom of tree version with dependencies, task instantiatiation (*-n 8192 -s 1 -m 1*)

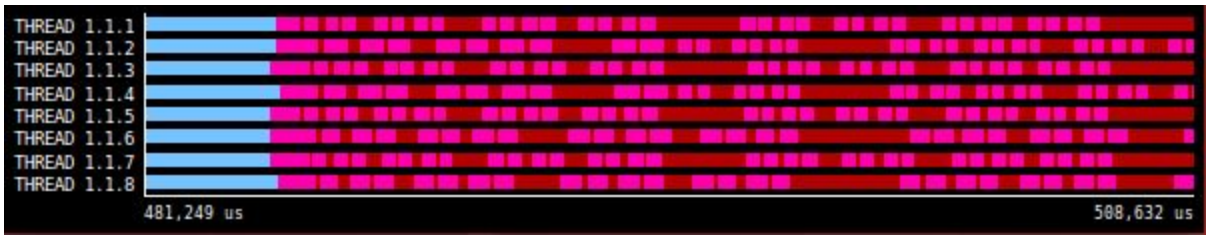


Figure 41: Paraver zoom of tree version with dependencies, task execution (*-n 8192 -s 1 -m 1*)

CONCLUSION

As we have seen in this study, the structure of an implementation and its data dependencies will severely limit the parallelism we can obtain from the code.

One of the most interesting experiments, though simple, has been the optional exercise number 2. We did not expect the random function to be so time consuming, and having to come up with a new approach using the same idea of the static scheduler for a loop directive in openMP was entertaining.

We have seen that finer granularity can be an obstacle with all the overhead that brings with it. We have also seen that the cut-off mechanism can be very useful, since deeper levels of recursion will worsen your execution times with their overheads and we have hit a wall in the last exercise, using explicit task dependencies and understanding that we would not be able to get better implementations than that.

Looking at the study overall, we are impressed by how much impact the smaller functions can have in the performance of the program. At first, everything that was not the multisort and merge functions looked like they were not very important, but when you are getting to the limit of parallelization of the important functions, you see that you need to tackle the simple functions as well, if you want a strongly scalable program.

In the end, we have accomplished a close to ideal multisort implementation (boada-5 will argue that it is in fact ideal) and with our initialization parallel function we have obtained a great strong scalability.