PARAL·LELISME: LABORATORI 4

# 

Pol Casacuberta Gil i Salvador Grimalt BInimelis

Quadrimestre 1, curs 22-23

Paral·lelisme, grup 21

## ÍNDEX

[**ÍNDEX**](#_uz27bsvwoitn) **2**

[**1 Introduction**](#_hslogf7nbhgm) **3**

[**Parallelisation strategies**](#_ut93tyv4hw9m) **4**

[**2 Task decomposition analysis for Mergesort**](#_sg525yci2c14) **4**

[2.1 Divide and conquer](#_3w3r5rrxt74j) 4

[2.2 Task decomposition analysis with Tareador](#_qxj0uc6asi5a) 4

[**Performance evaluation**](#_d3r4mk9fzcda) **12**

[**3 Shared-memory parallelisation with OpenMP tasks**](#_25njy7njsw6g) **12**

[**3.3 Optimization: Task granularity control: the cut–off mechanism**](#_3h0n6lf61wuf) **20**

[**Optional 1:**](#_qnzt641bwrwi) **25**

[**4 Shared-memory parallelisation with OpenMP task using dependencies**](#_mmh702q7svz) **26**

[**Optional 2:**](#_nlyrcd6thliw) **31**

[**Final Conclusions**](#_8zn09vlmh6sj) **35**

# 1 Introduction

In this laboratory assignment our task is to study the use of parallelism in recursive programs. Specifically we are working with the Merge Sort algorithm. The Merge Sort Algorithm divides the input array into two halves, makes the recursive for both halves and then merges the sorted two halves using merge. We’re working with the file multisort.c and the multisort-tareador.c

Recursive task decompositions

In the first session we are working with the multi-tareador.c file and we investigate with the Tareador, looking after solving the code’s parallelism problems with the OMP directives explained at class.

The first solution explained is the divide-and-conquer solution and there we see the decomposition of the process in tasks and we discover two ways of doing it: the leaf recursive task decomposition and the tree recursive task decomposition. This affects basically two aspects of the result: the number of tasks generated and the moment when they are generated because, looking at the first example (Divide and conquer with dot product) then we see that in the case of the leafs first task initialized is also the last finished one and in the case of the tree the last task is created with the leaves.

# 

# Parallelisation strategies

# 2 Task decomposition analysis for Mergesort

## 2.1 Divide and conquer

1. Reference times

Initialization time in seconds: 0.683067

Multisort execution time: 5.189703

Check sorted data execution time: 0.011634

## 2.2 Task decomposition analysis with Tareador

2.

Excerpt of the leaf strategy:

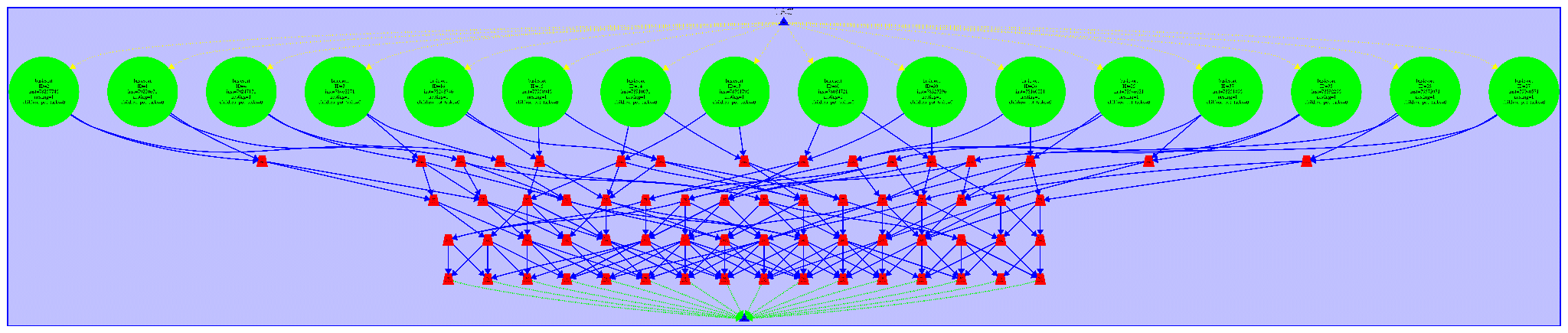
| void merge(long n, T left[n], T right[n], T result[n\*2], long start, long length) {  if (length < MIN\_MERGE\_SIZE\*2L) {  // Base case  tareador\_start\_task("basicmerge");  basicmerge(n, left, right, result, start, length);  tareador\_end\_task("basicmerge");   } else {  // Recursive decomposition  merge(n, left, right, result, start, length/2);  merge(n, left, right, result, start + length/2, length/2);  } }  void multisort(long n, T data[n], T tmp[n]) {  if (n >= MIN\_SORT\_SIZE\*4L) {  // Recursive decomposition  multisort(n/4L, &data[0], &tmp[0]);  multisort(n/4L, &data[n/4L], &tmp[n/4L]);  multisort(n/4L, &data[n/2L], &tmp[n/2L]);  multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L]);   merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);  merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L);   merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);  } else {  // Base case  tareador\_start\_task("basicsort");  basicsort(n, data);  tareador\_end\_task("basicsort");  } } |
| --- |

Excerpt of the 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("merge");  merge(n, left, right, result, start, length/2);  tareador\_end\_task("merge");    tareador\_start\_task("merge");  merge(n, left, right, result, start + length/2, length/2);  tareador\_end\_task("merge");   } }  void multisort(long n, T data[n], T tmp[n]) {  if (n >= MIN\_SORT\_SIZE\*4L) {  // Recursive decomposition  tareador\_start\_task("multisort");  multisort(n/4L, &data[0], &tmp[0]);  tareador\_end\_task("multisort");    tareador\_start\_task("multisort");  multisort(n/4L, &data[n/4L], &tmp[n/4L]);  tareador\_end\_task("multisort");    tareador\_start\_task("multisort");  multisort(n/4L, &data[n/2L], &tmp[n/2L]);  tareador\_end\_task("multisort");    tareador\_start\_task("multisort");  multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L]);  tareador\_end\_task("multisort");    tareador\_start\_task("merge");  merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);  tareador\_end\_task("merge");    tareador\_start\_task("merge");  merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L);  tareador\_end\_task("merge");    tareador\_start\_task("merge");  merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);  tareador\_end\_task("merge");   } else {  // Base case  basicsort(n, data);  } } |
| --- |

3.

Leaf strategy



Tasks:

There are 80 tasks in total, 64 are basicmerge (red tasks) and 16 basicsort (green) calls.

Because the first 4 calls to multisort branch into 4 multisort 16 basicsorts are created.

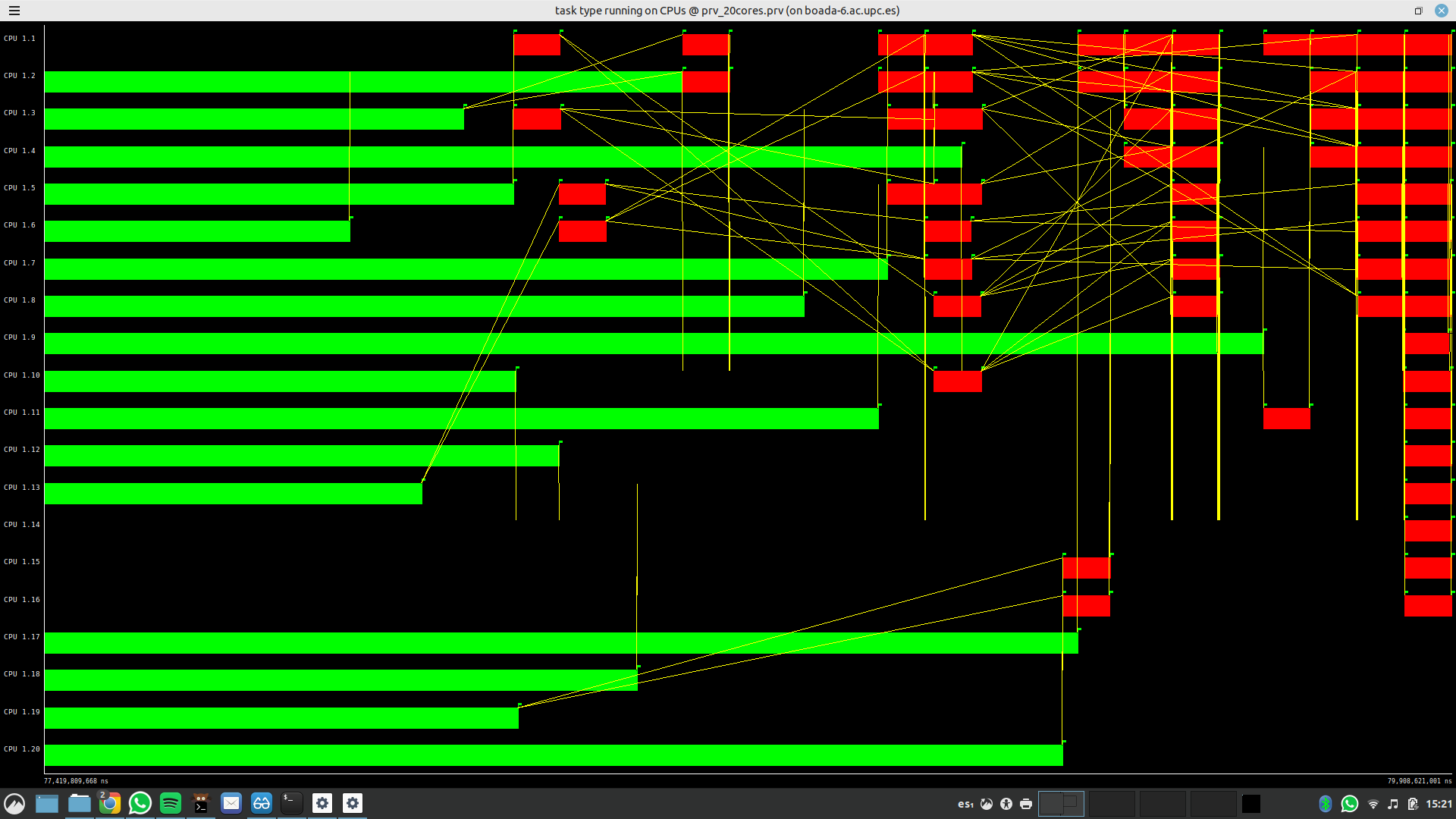
Now let’s analyse why there are 64 basicmerge. On the multisort calls of the second level there are no merges since they reach the base case, and they are just sorted but on the level one there are 4 calls to multisort. The first 2 would call upon the merge with k= 8192/4 = 2048 and would cause a merge recursion each reaching n=2048/2 = 1024. Meaning there are 4 basicmerge in each. 4\*4 =16 basicmerges.

The third merge call would have to merge the resulting vectors of the previous 2 now double the size this totals to 16, with the previous basicmerges that sums up to 16\*2=32.

Now on level 0 the 2 merges sum up to 16 because of the bigger size and the final merge takes 16 more merges. 32+16+16=64 in total.

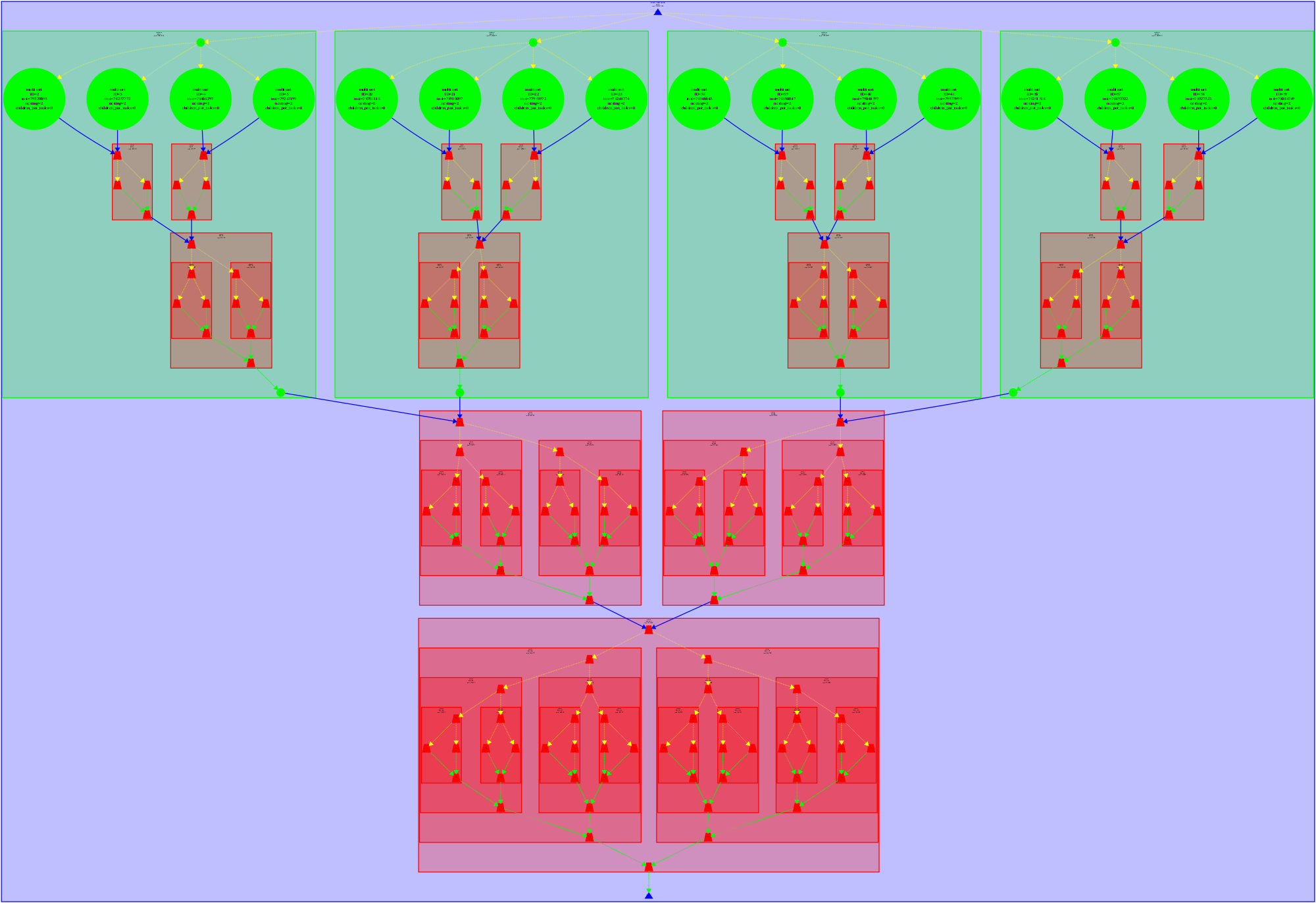
Granularity: if we think of it as the number of calls to basicsort and basicmerge by each task in this strategy, it equals to 1.

Synchronization: basic sort does not seem need to have any parallelization.

Basicmerge however, every merge depends on the prior two tasks concerning the fragment of the list that would be merged depends on those being done before merging. We need it’s children to be done and only it’s children, we could use omp taskwait.

From this timelines we can appreciate that most of the time is being spent on the basicsort tasks. Near the end is where parallelization is less homogenous and where the merge tasks are performed.

Tree strategy



Tasks:

In this strategy there are a total number of 133 tasks, of which 113 were merge (red) and multisort tasks (in green) they go up to 20, much lower than merges.

To see why we got 20 multisort tasks we can simply see that on the first level of recursion where we have 4 calls to multisort and each of those made 4 calls to multisort (second recursion level). 4\*4 + 4 = 20

Now let’s look at the 113 merges, for each level one multisort call there are 2 merge calls which would be divided into two for each of those making 3 tasks each merge. There are 4 multisort tasks. 4\*(3+3) = 24 merges.

Then the third merge would divide into two and those two divisions would then divide into two. Totalling now: 4\*(3+3) + 4\*(1+2+4) = 52

On the level 0 multisort there were two merges which unify the previous blocks. Which have to unify two blocks of 7 tasks each. 4\*(3+3) + 4\*(1+2+4) + 2\*(2\*(1+2+4)+1) = 82

Finally, the last merge unifies the whole list back together.

4\*(3+3) + 4\*(1+2+4) + 2\*(2\*(1+2+4)+1) + 1+2\*(2\*(1+2+4)+1) =113

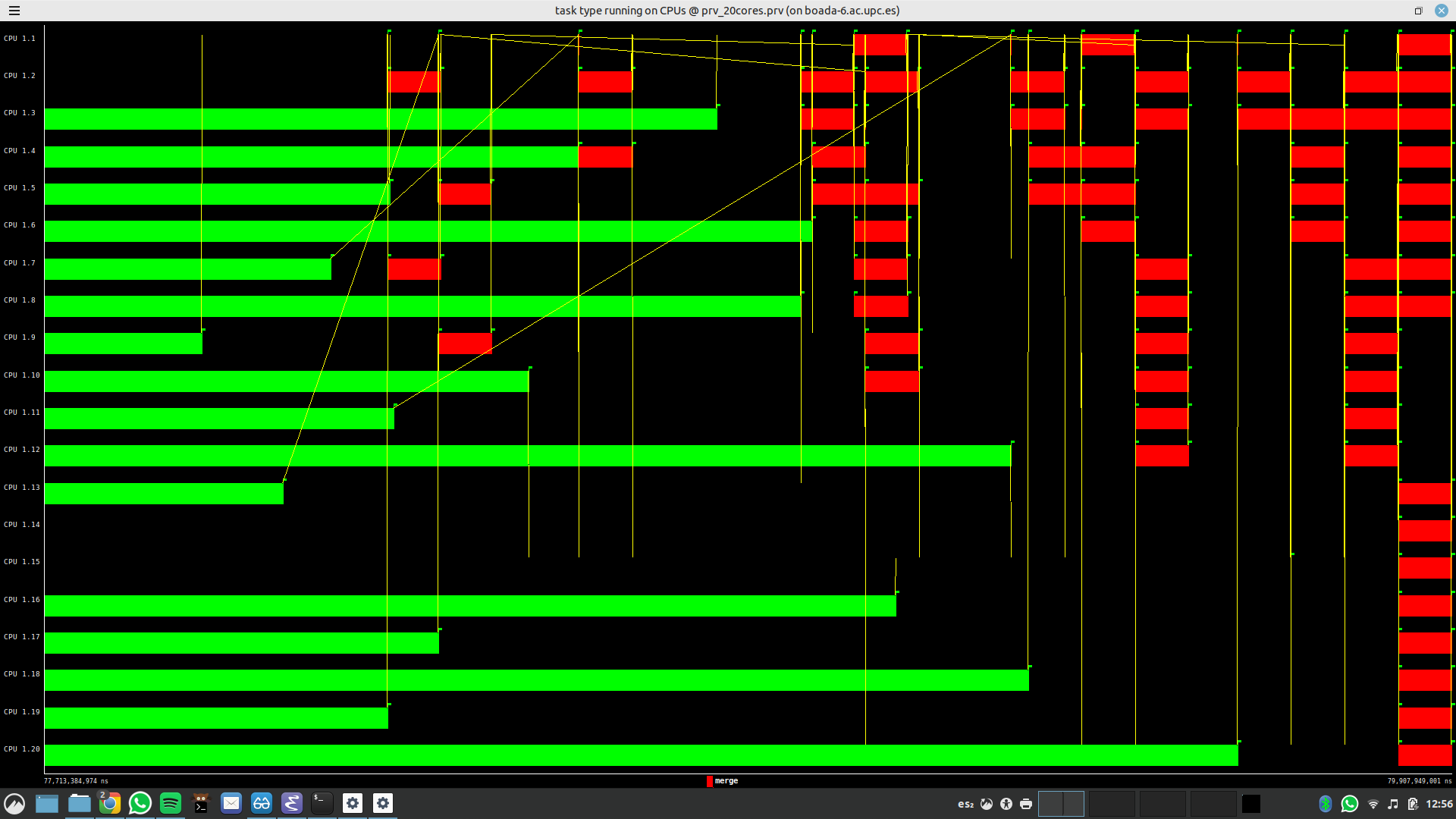
Granularity: we’ll separate granularities into basicsort, basicmerge, and merge.

As for multisort granularity ranges from 1 to 4. Because the level one multisorts will call to 4 multisorts resulting in 4 basicsorts each. So the maximum is 4 and the minimum is 1.

Now we’ll take a look at merge the range goes from 2 to 16 calls per task. All merges at the recursive stage call 2 merges. And at most 1 task creates all other merges, 16 in total.

If we think of granularity in terms of basicmerge, then understanding that only the last tasks execute the basicmerge call, given that the others merely receive the result and not actually execute them, then the granularity would be 1 since the last call, the base case, calls to basicmerge and the others don’t.





On this timeline we see a very similar situation to the leaf task decomposition, the differences reside in the last part of the execution, where many more merges appear in this tree version.

Another difference that can be appreciated is that the task dependencies in the leaf tasks are only data dependencies. The tree dependencies call for data and inheritance dependencies.

Synchronization Leaf and Tree: synchronization needed by the tree task decomposition is greater than the Leaf strategy. It’s nested nature made it be forced to wait for the completion of the innermost tasks to finish before bringing the execution of the next task to an end. This however generates the need for a task synchronization construct for all its descendants. For example, a *taskgroup* clause. And for the leaf synchronization we’ll use omp taskwait.

Leaf synchronization:

| 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);  } } |
| --- |

Tree synchronization:

| 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);  } } |
| --- |

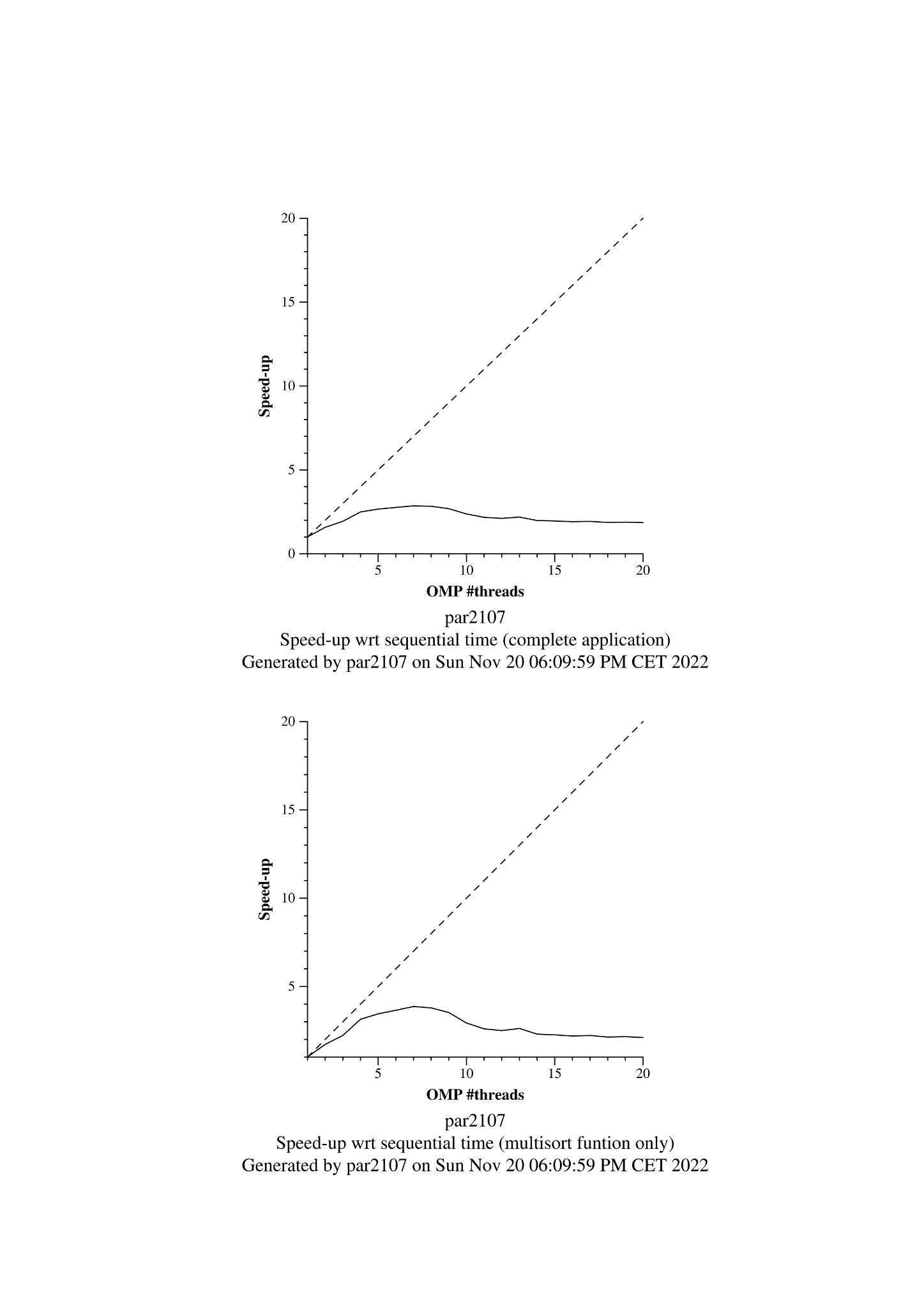
# Performance evaluation

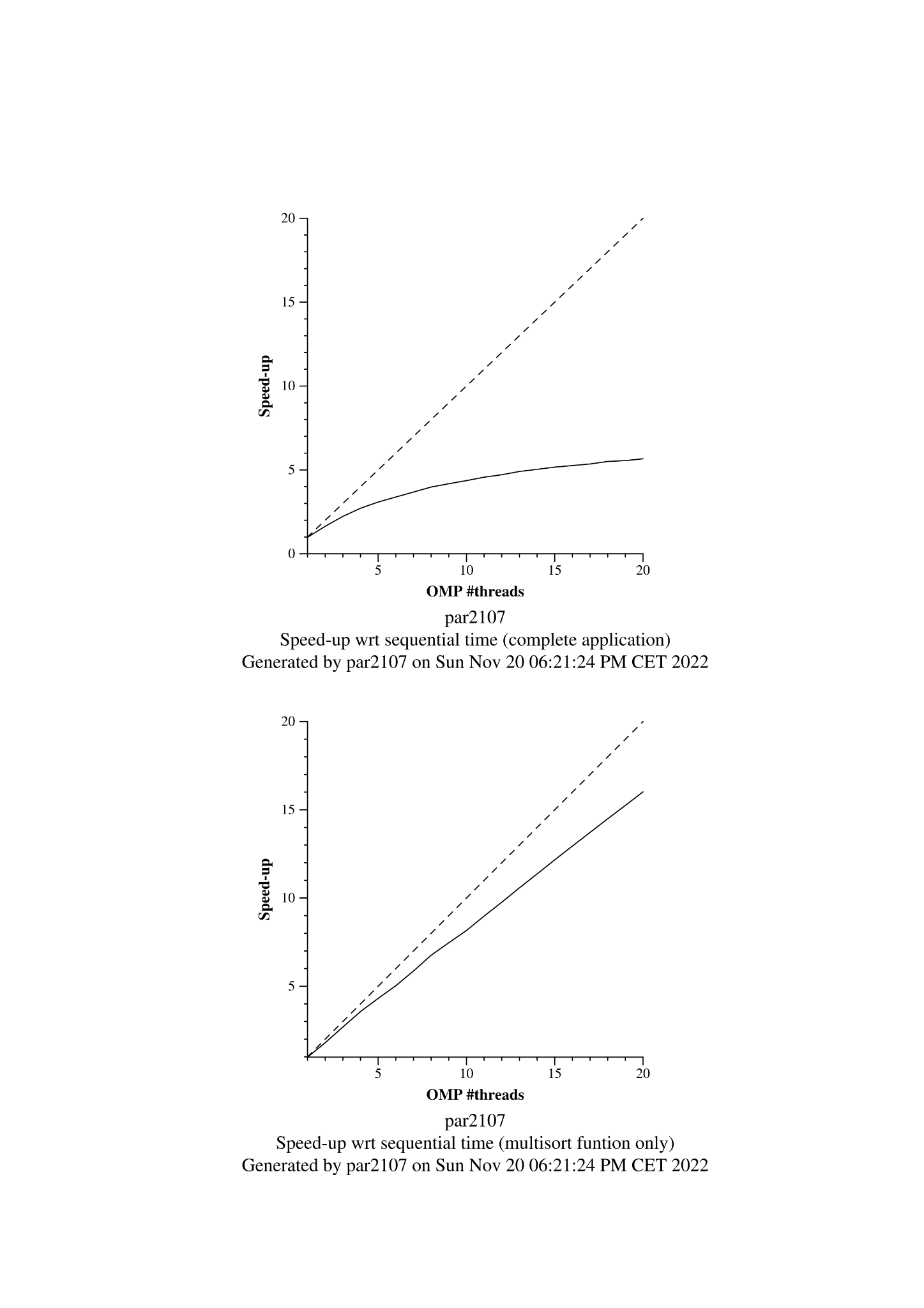
# 3 Shared-memory parallelisation with OpenMP tasks

Leaf strategy:

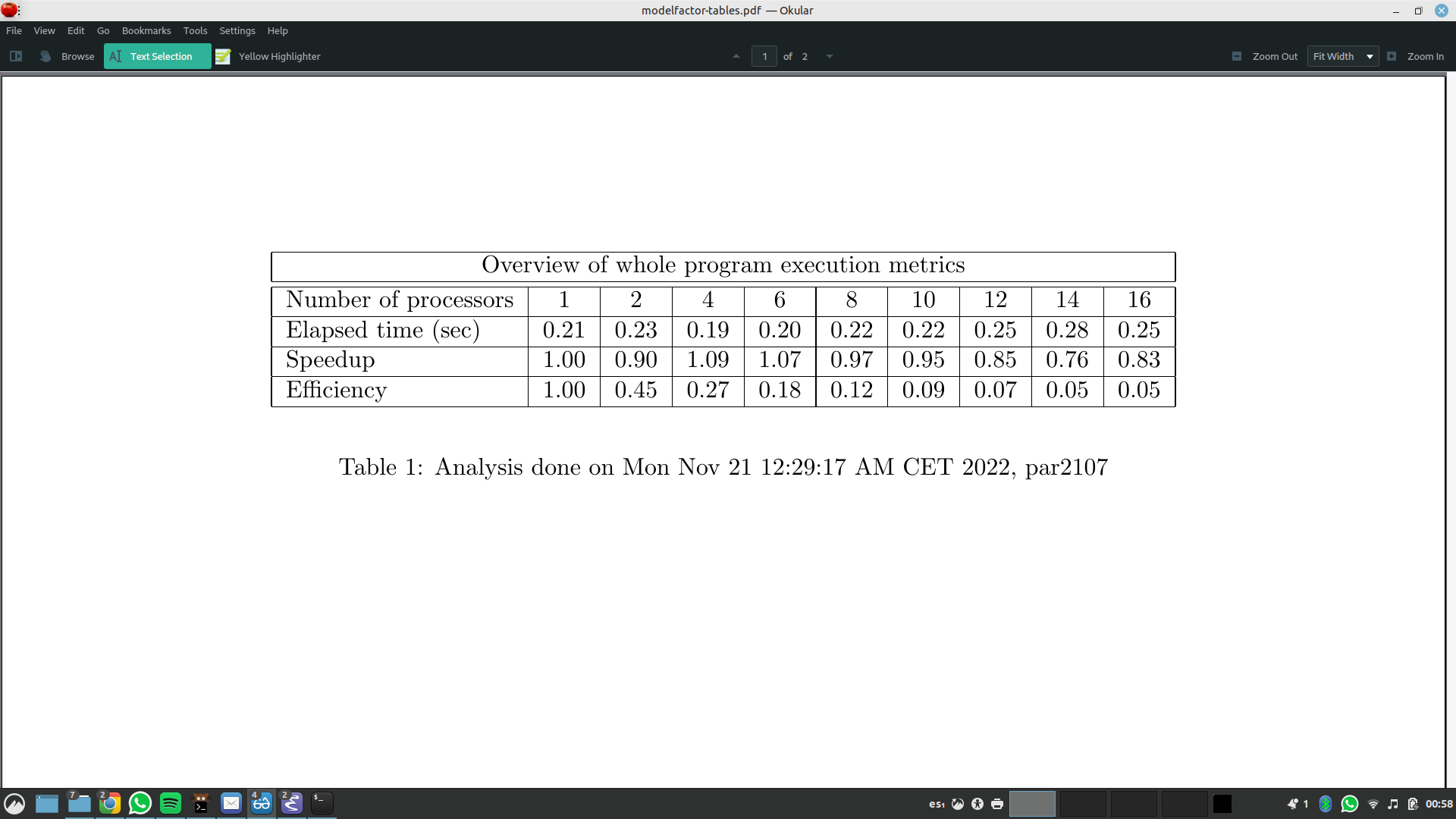
We will be using the leaf synchronization code shown above, after making sure that the array is properly sorted we executed the following command: sbatch ./submit-strong-omp.sh

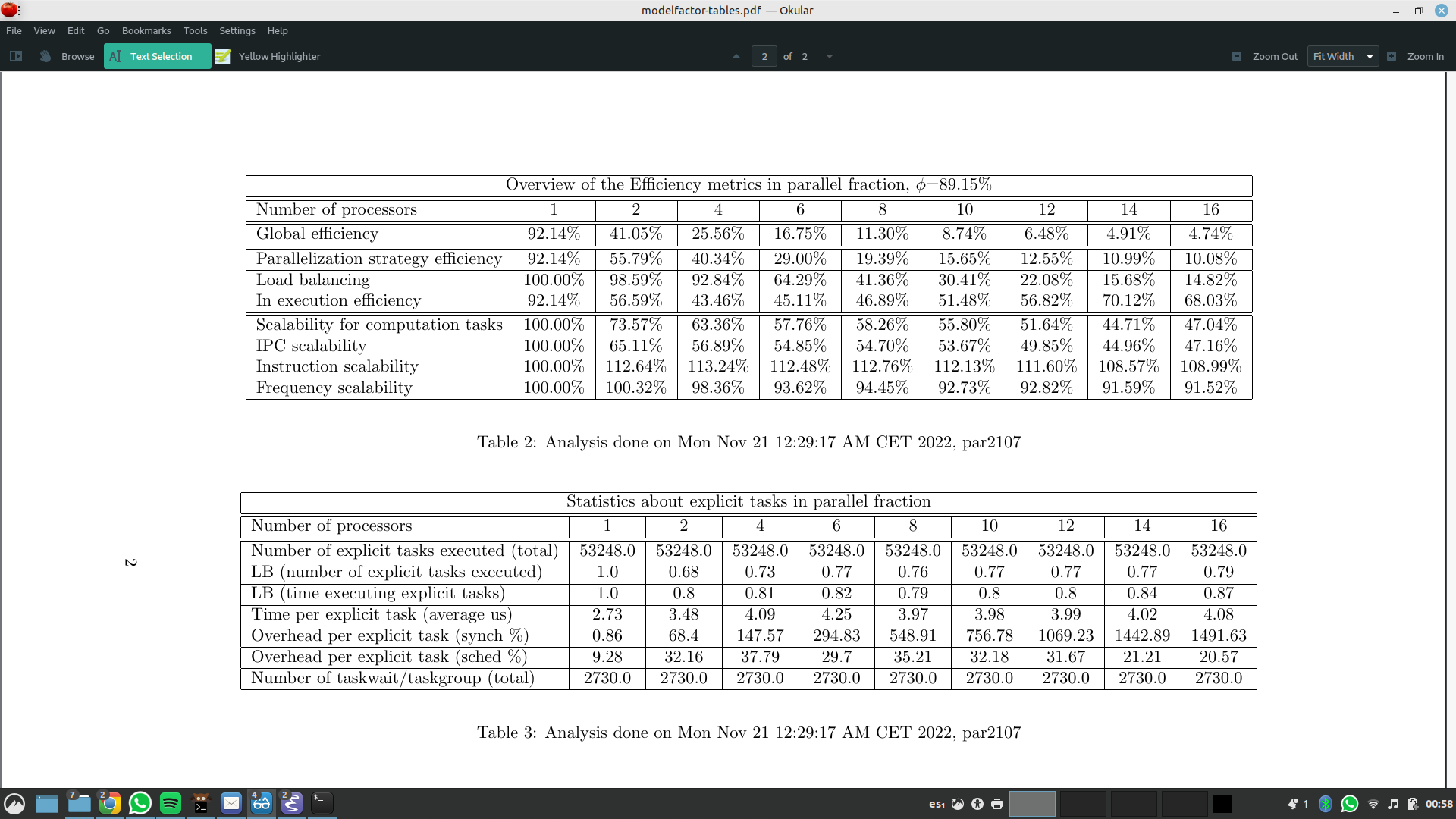
And obtained the following plots:

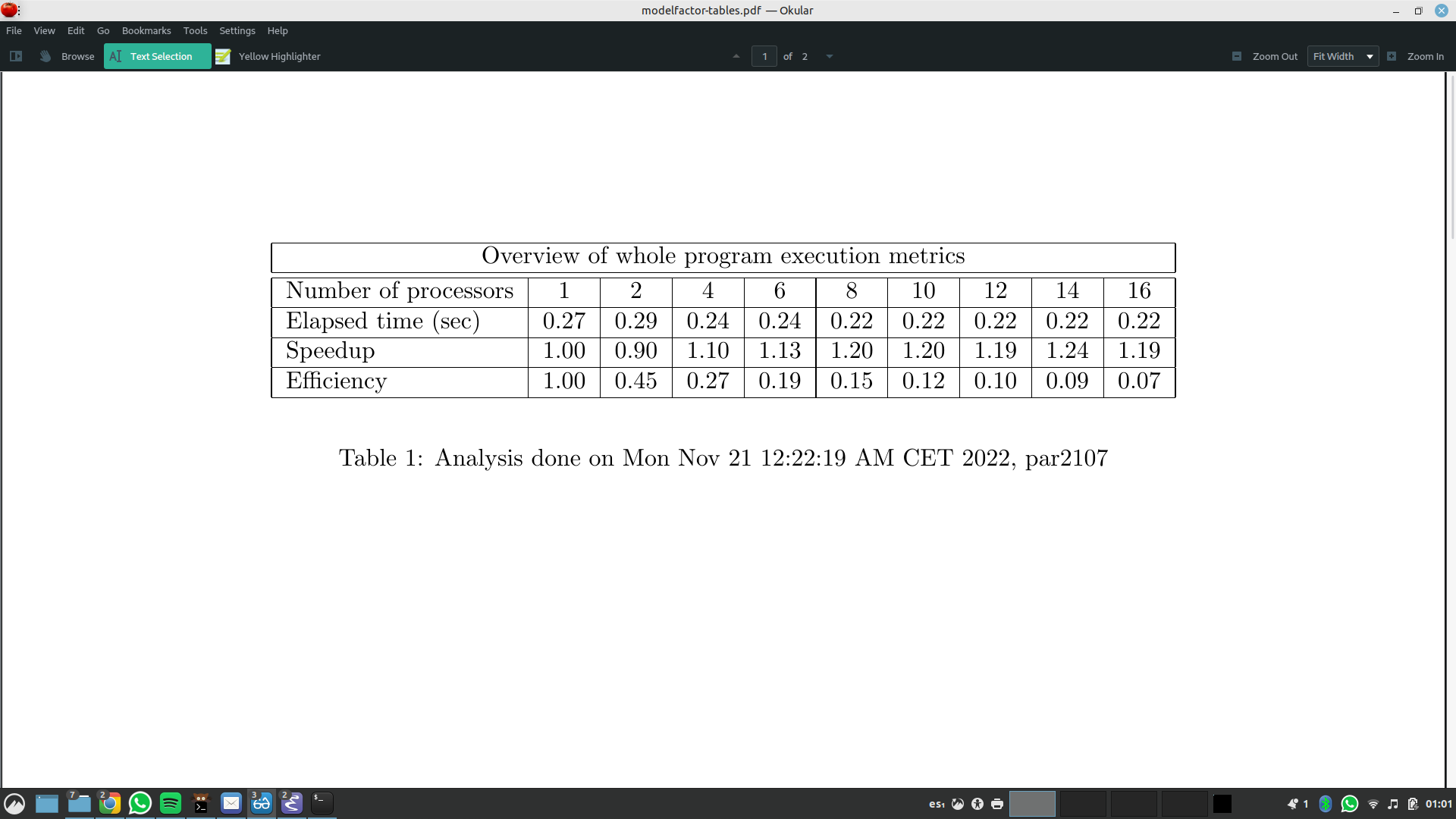
Leaf:

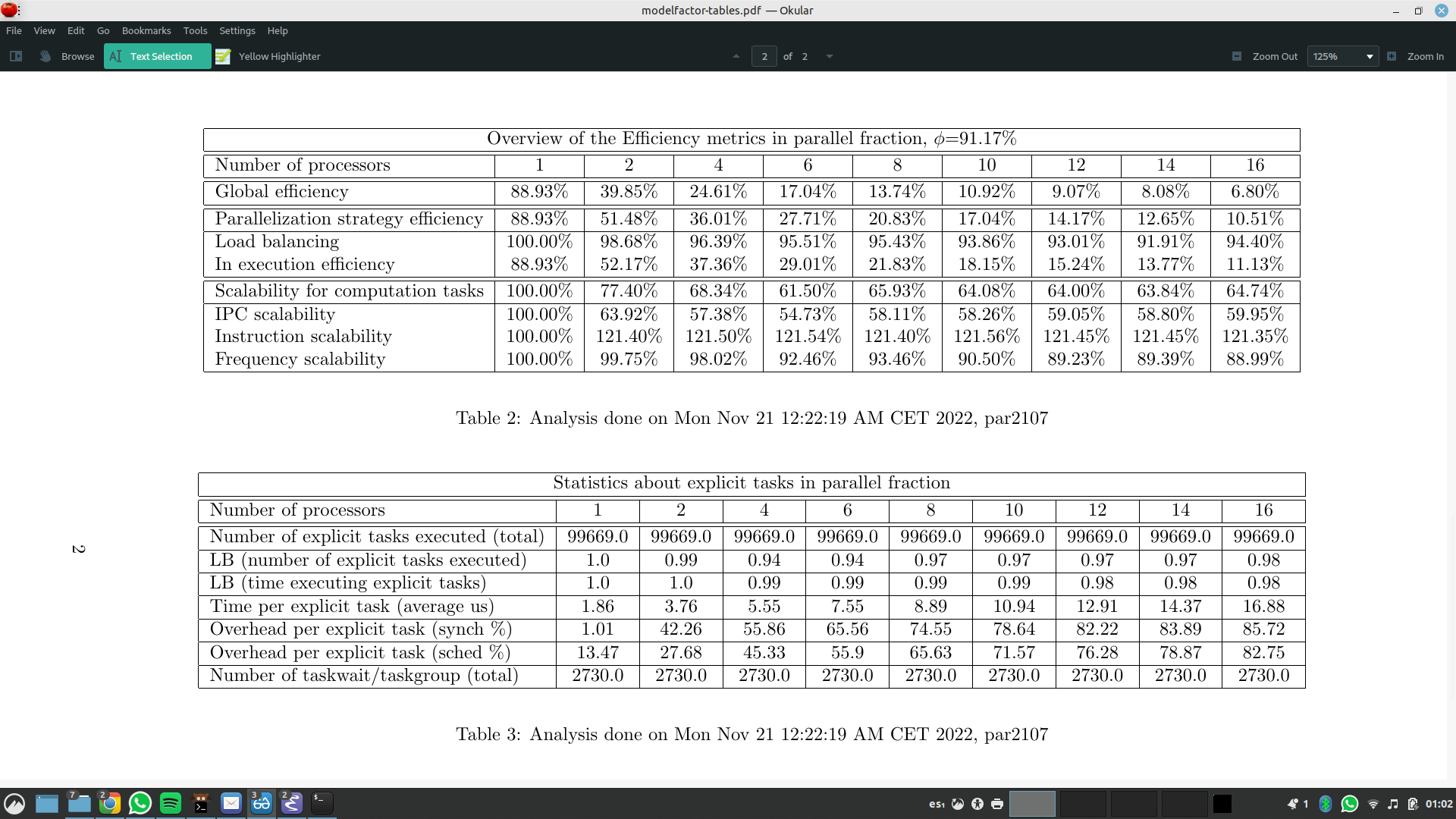
Tree:

Leaf:





Tree:

Performance seems to be pretty poor, there are several options to think about why this could be happening, first we take a look at the parallelization fraction, in both executions this parameter seems to be about 90% which is pretty good, this doesn’t seem to be the problem here.

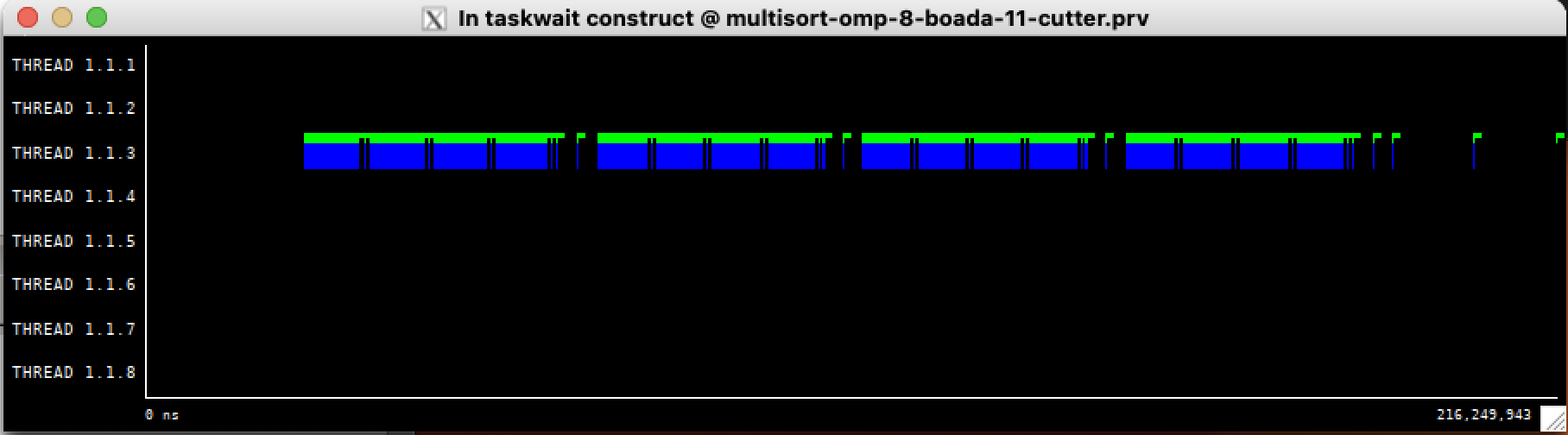
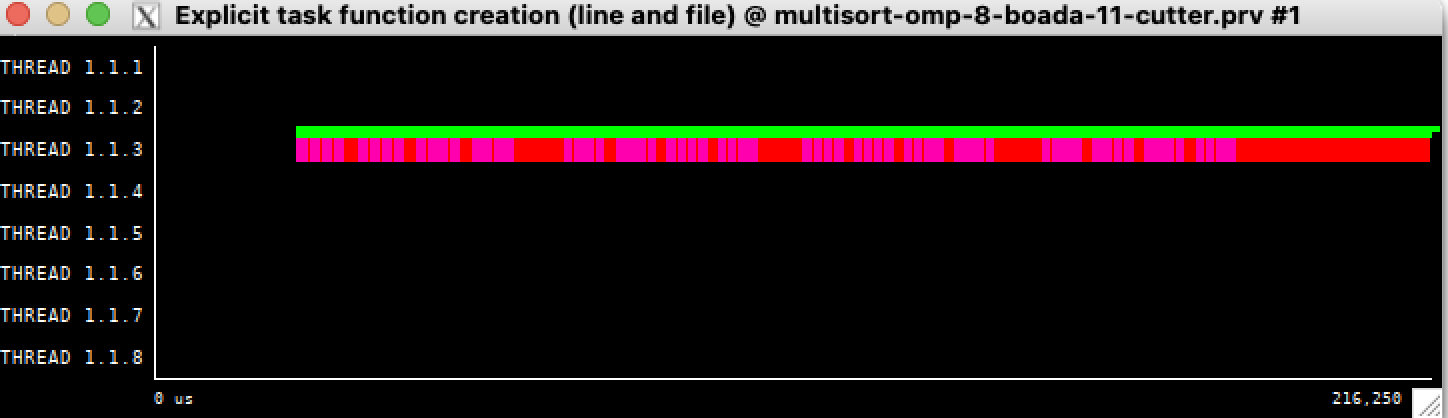
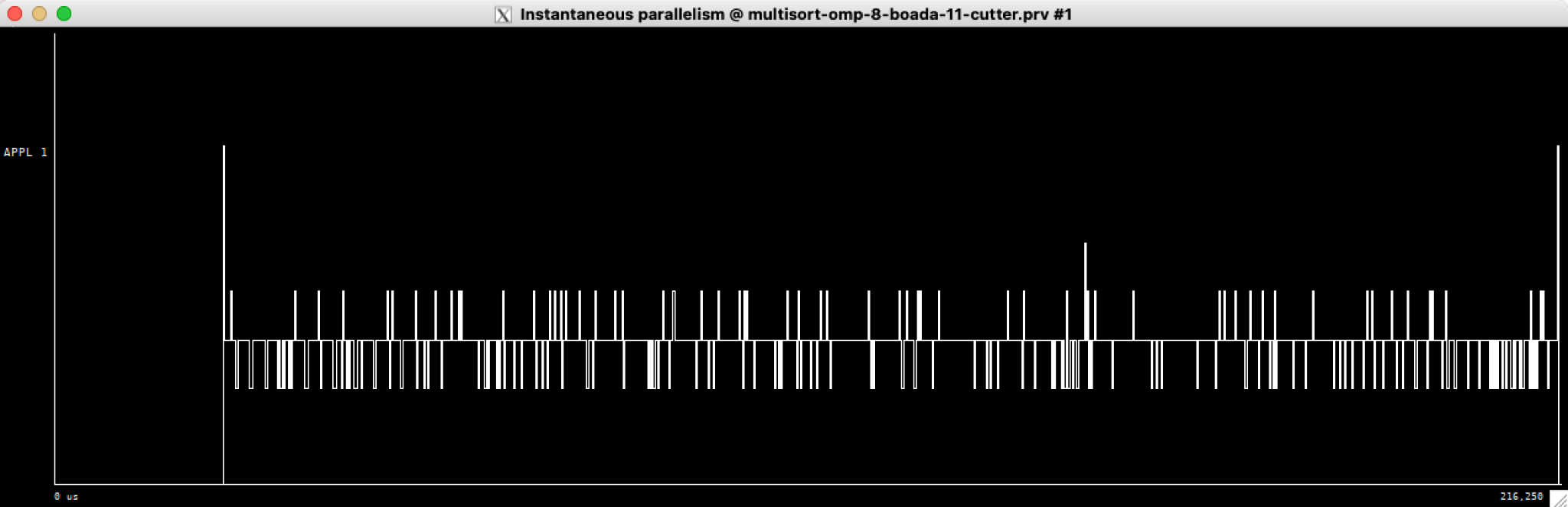
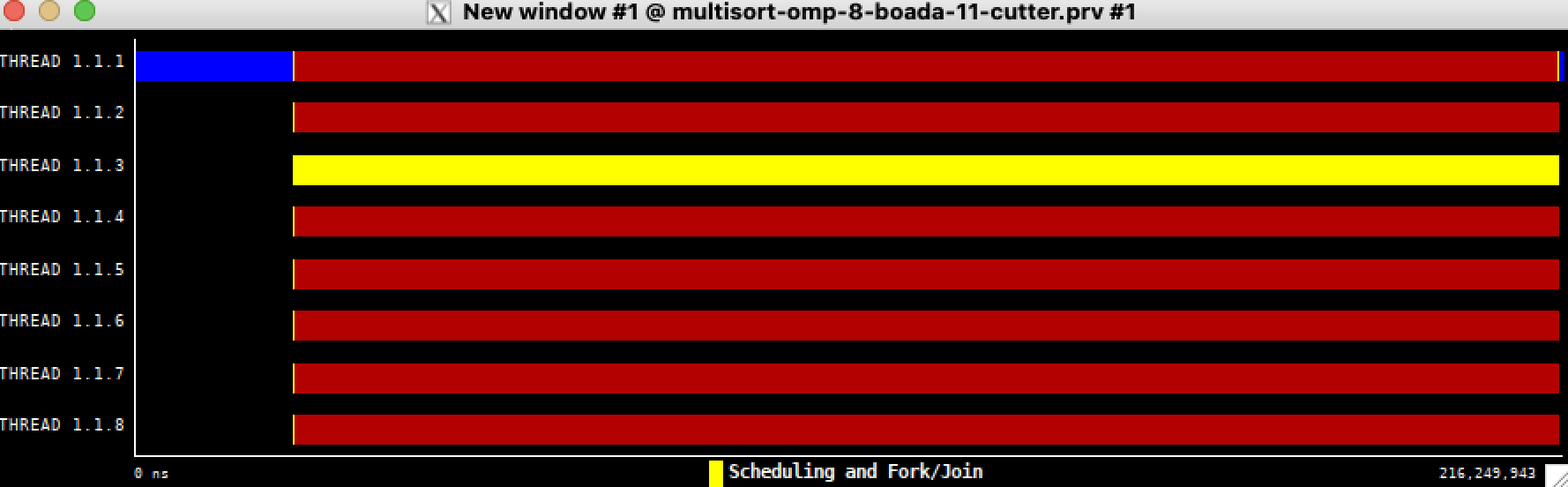
Load balancing on the leaf strategy seems to be pretty terrible, however on the tree strategy this seems to be massively improved.

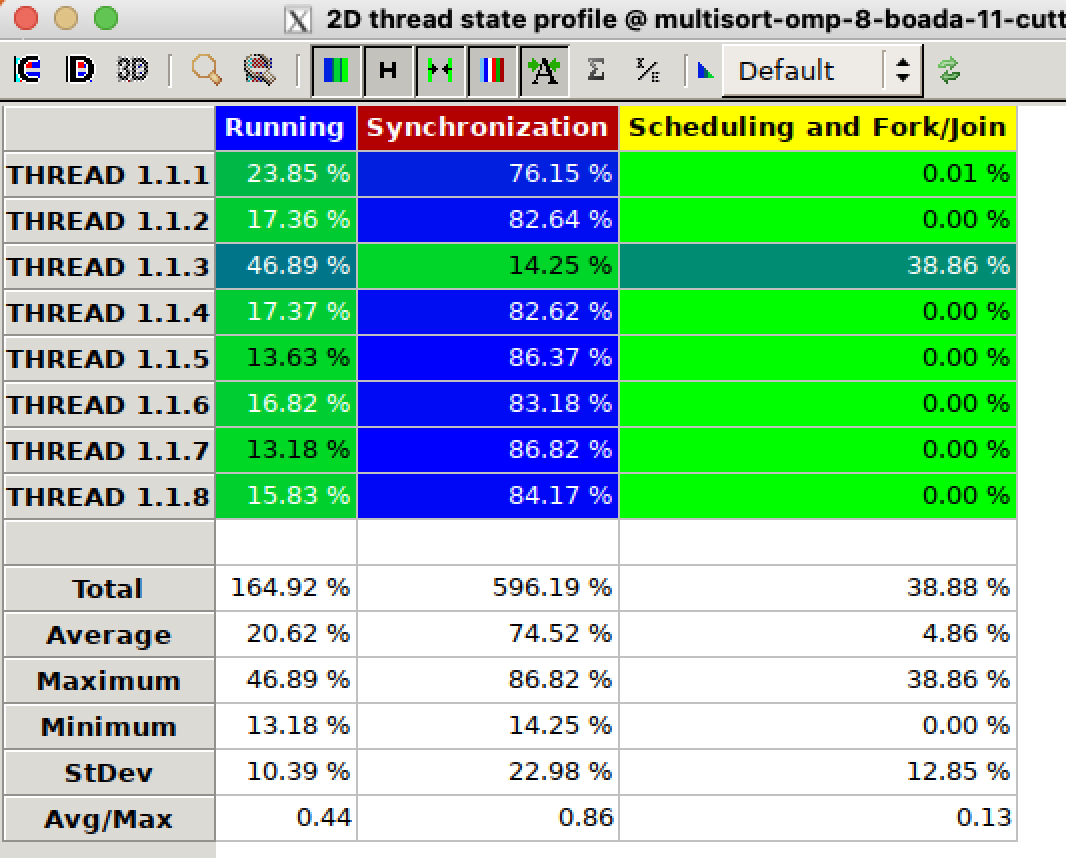
As to in execution efficiency on the tree strategy, it seems to be pretty bad as well, with up to 85% overheads per explicit tasks. On the leaf strategy, the synchronization overhead per explicit task seems outrageously high.

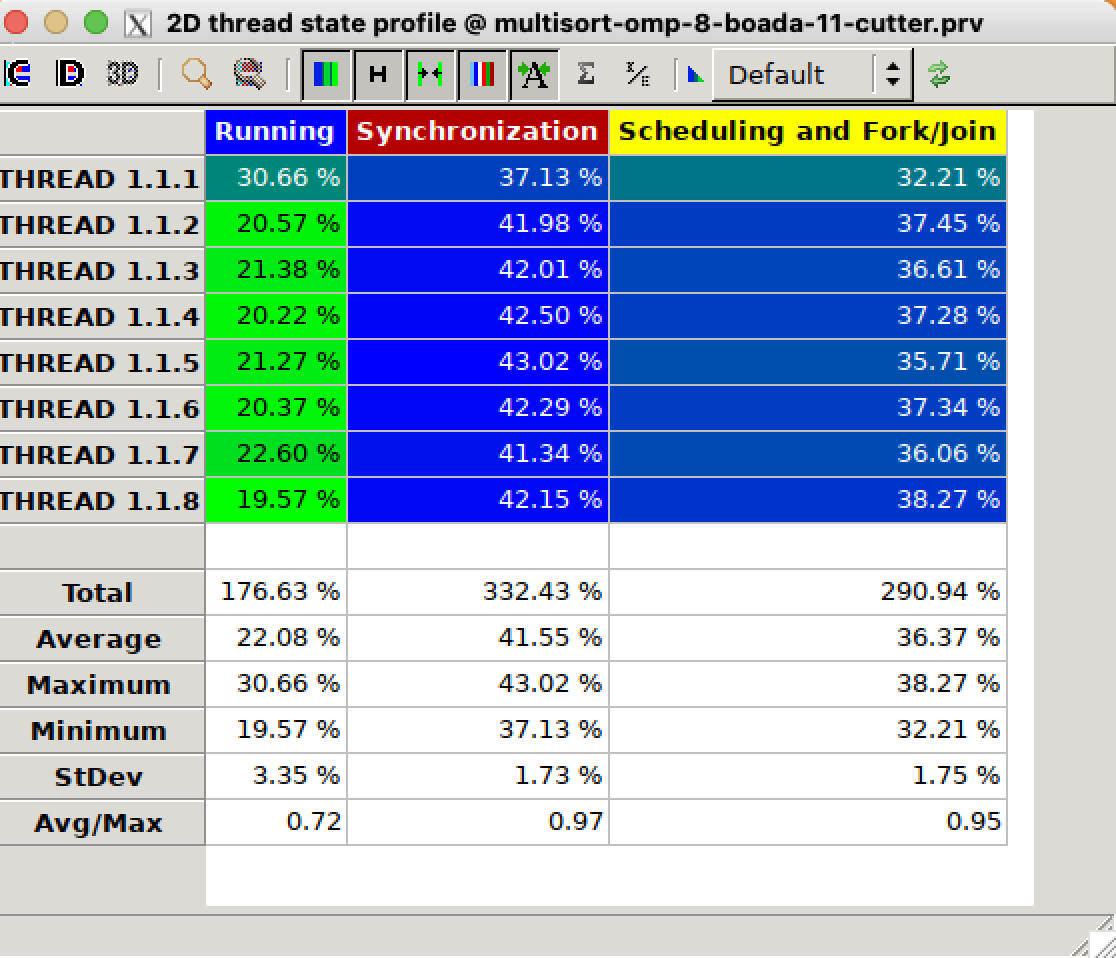
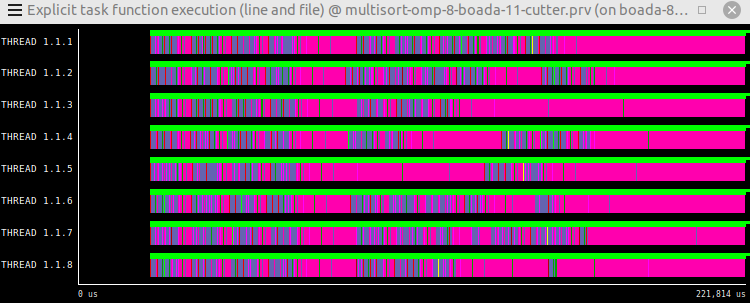
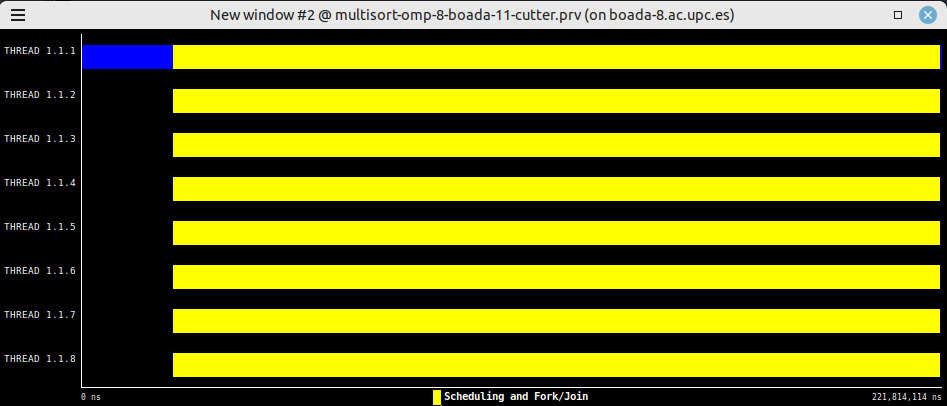
There also seem to be double the number of explicit tasks executed on the three strategy, which might also be hindering the efficiency.

Let’s take a look at some paraver hints to further inspect the problem at hand.

Leaf:





Tree:

The main difference between the leaf and the tree timeline is that on the leaf, there’s one thread creating all the tasks whilst on the tree all 8 cores contribute to creating and joining tasks.

The rest of the structure is very similar. This makes sense given the nature of the methods used to divide the tasks.

Taskwaits on the leaf strategy seem to only be performed by one thread, the one creating the tasks. But on the tree strategy, all threads seem to be doing taskgroup tasks.

If we take a look at the creation of explicit tasks we can clearly see that one thread is creating all the tasks on the leaf while on the tree strategy they are all responsible for the creation of tasks.

Now taking a look at the execution of these tasks we can see that on the tree strategy the load is very balanced while on the leaf there’s one thread that is clearly working less, the one creating the tasks.

We’ll use the hint Instantaneous Parallelism to better help us understand the amount of parallelism used. On the tree we see very easily that there is better parallelism, with an average of 6,7 and spiking at 8 sometimes. On the leaf, most of the time only 4 processors are working, sometimes getting to 5 or 3 processors.

Now we see why leaf might not be the best choice, since all of that parallelization looking at the histogram of the parallel execution is mostly spent on synchronization.

So all in all it seems as if the tree would be a much better option, why is it not performing as we would expect?

Let’s look at the histogram of the parallel execution of the tree. We can see a much better synchronization overhead, but another factor has been impacted by this change, there is now a 20-30 percent being spent on scheduling and fork/join spread across all threads, which is hindering our performance.

Overall, the tree seems to be the better and less flawed alternative.

# 3.3 Optimization: Task granularity control: the cut–off mechanism

| void merge(long n, T left[n], T right[n], T result[n\*2], long start, long length, int depth) {  if (length < MIN\_MERGE\_SIZE\*2L) {  // Base case  basicmerge(n, left, right, result, start, length);  } else {  // Recursive decomposition  if(!omp\_in\_final()) {  #pragma omp task final(depth >= CUTOFF)  merge(n, left, right, result, start, length/2, depth+1);  #pragma omp task final(depth >= CUTOFF)  merge(n, left, right, result, start + length/2, length/2,depth+1);  }  else {  merge(n, left, right, result, start, length/2, depth+1);  merge(n, left, right, result, start + length/2, length/2,depth+1);  }  } }  void multisort(long n, T data[n], T tmp[n], int depth) {  if (n >= MIN\_SORT\_SIZE\*4L) {  // Recursive decomposition  if(!omp\_in\_final()) {  #pragma omp taskgroup  {  #pragma omp task final(depth >= CUTOFF)  multisort(n/4L, &data[0], &tmp[0], depth+1);  #pragma omp task final(depth >= CUTOFF)  multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);  #pragma omp task final(depth >= CUTOFF)  multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);  #pragma omp task final(depth >= CUTOFF)  multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L], depth+1);  }  #pragma omp taskgroup  {  #pragma omp task final(depth >= CUTOFF)  merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);  #pragma omp task final(depth >= CUTOFF)  merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L, depth+1);  }  #pragma omp task final(depth >= CUTOFF)  merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);  }  else {  multisort(n/4L, &data[0], &tmp[0], depth+1);  multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);  multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);  multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L], depth+1);   merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);  merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L, depth+1);   merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);   }  } else {  // Base case  basicsort(n, data);  } }  int main(int argc, char \*\*argv) { ...  #pragma omp parallel  #pragma omp single  multisort(N, data, tmp,0); ...  return 0; } |
| --- |

To check the correctness of our code we ran sbatch ./submit-omp.sh multisort-omp n\_threads n\_cutoff.

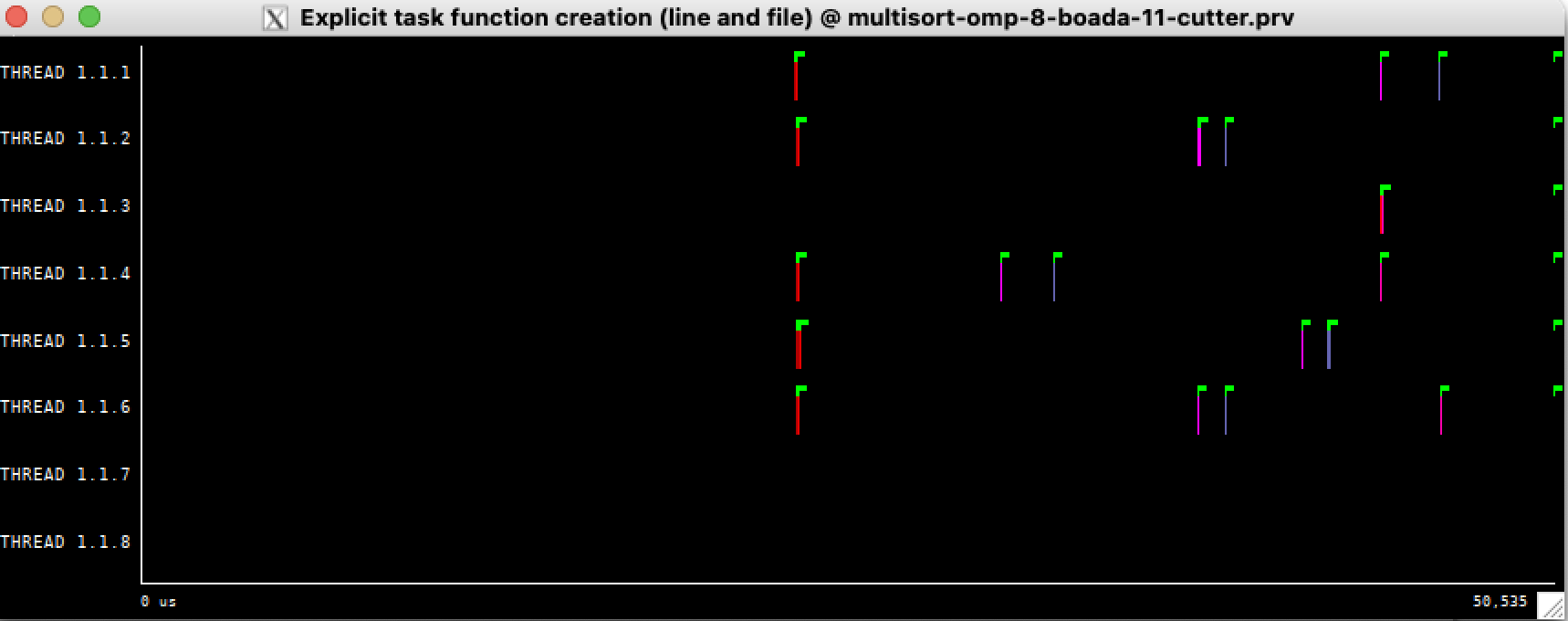
Upon multiple executions of submit-cutoff-omp.sh we’ve seen that evidently the best cut-off value heavily depends on the number of threads we use, so we would have to tune that parameter depending on how many threads we have available.

For example, for number\_of\_threads = 8 the best cutoff value is 4 or 5.

We’ll first use a cutoff = 0, we would be expecting that tasks are created up until the first level of recursion which would be 4 tasks for multisorts and 2 tasks for the first two merges and the last task is the final merge (the last task that be seen on the paraver window).

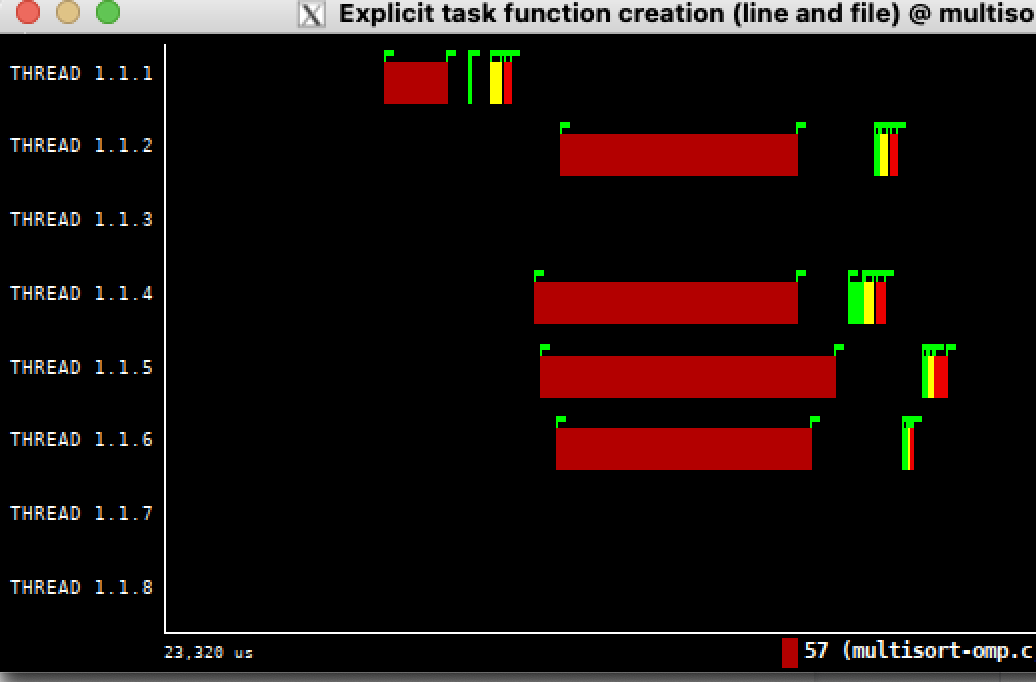
It definitely is hard to see, but if we enhance, we are able to see each individual task.

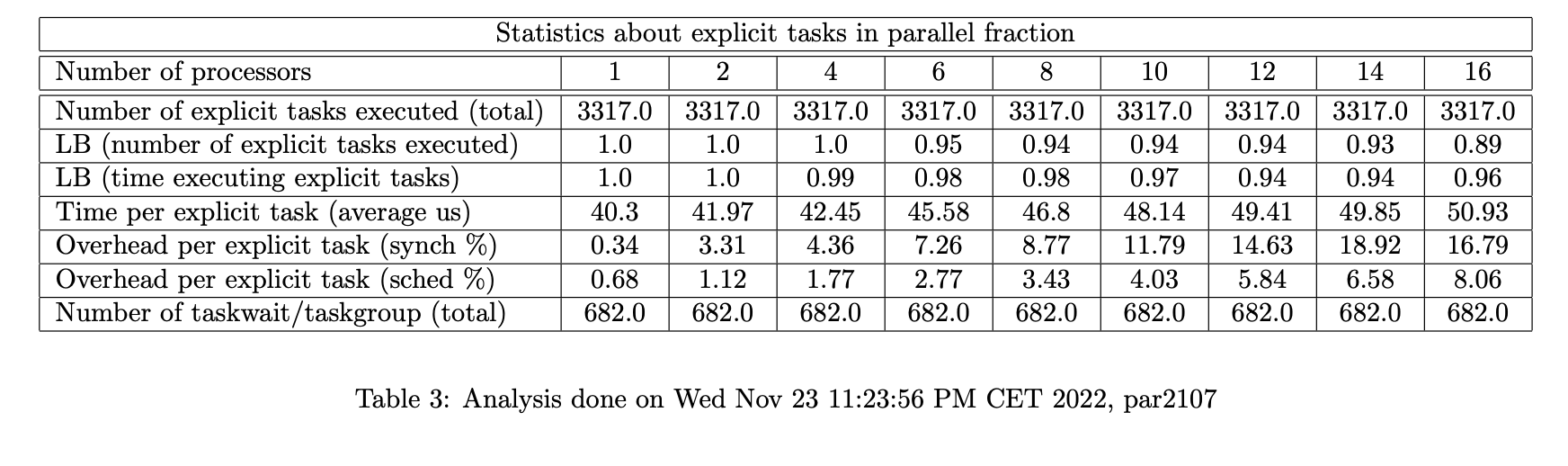
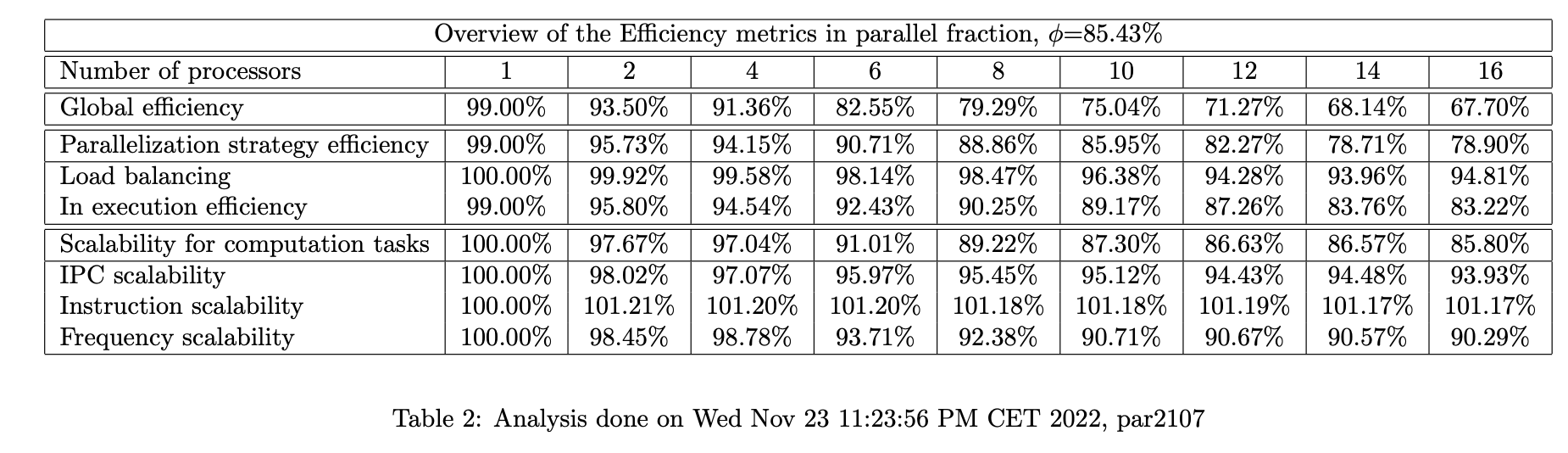
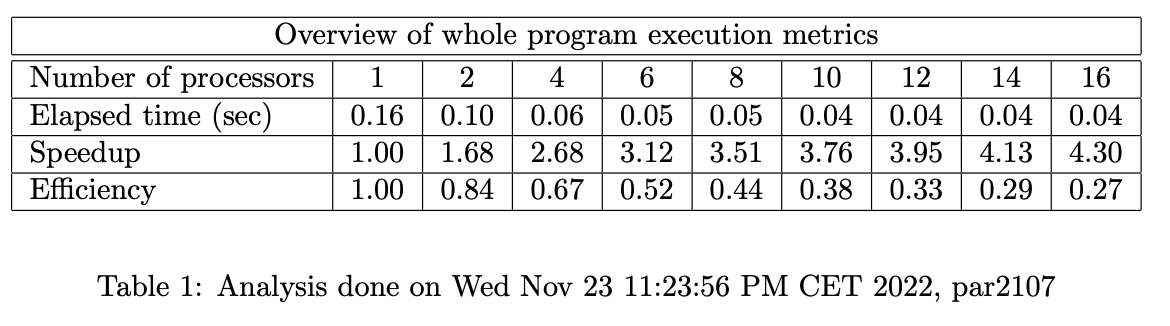
Now we’ll use a cutoff = 1, and we’ll look at the amount of tasks being generated.



This is going to be tough to count because we have to zoom at every single one, but if we do that we should find that the number of multisort tasks is 16+4, 4 that are created on the first level and then those will create 4 each, totalling 20 tasks. The number of merges is 3+6+4\*3 = 21 merges. 3 are the first three merges and those merges will call to another two merges each, now the 4 multisort tasks will each call 3 merges, totalling 21. In total 20+21 = 41 tasks. If we zoom in every task we’ll see that the blue tasks are just a single task the pink ones are two tasks together that i presume to be two merges together.

The tasks at the beginning look like this, presumably the 4 multisorts that are created at first.

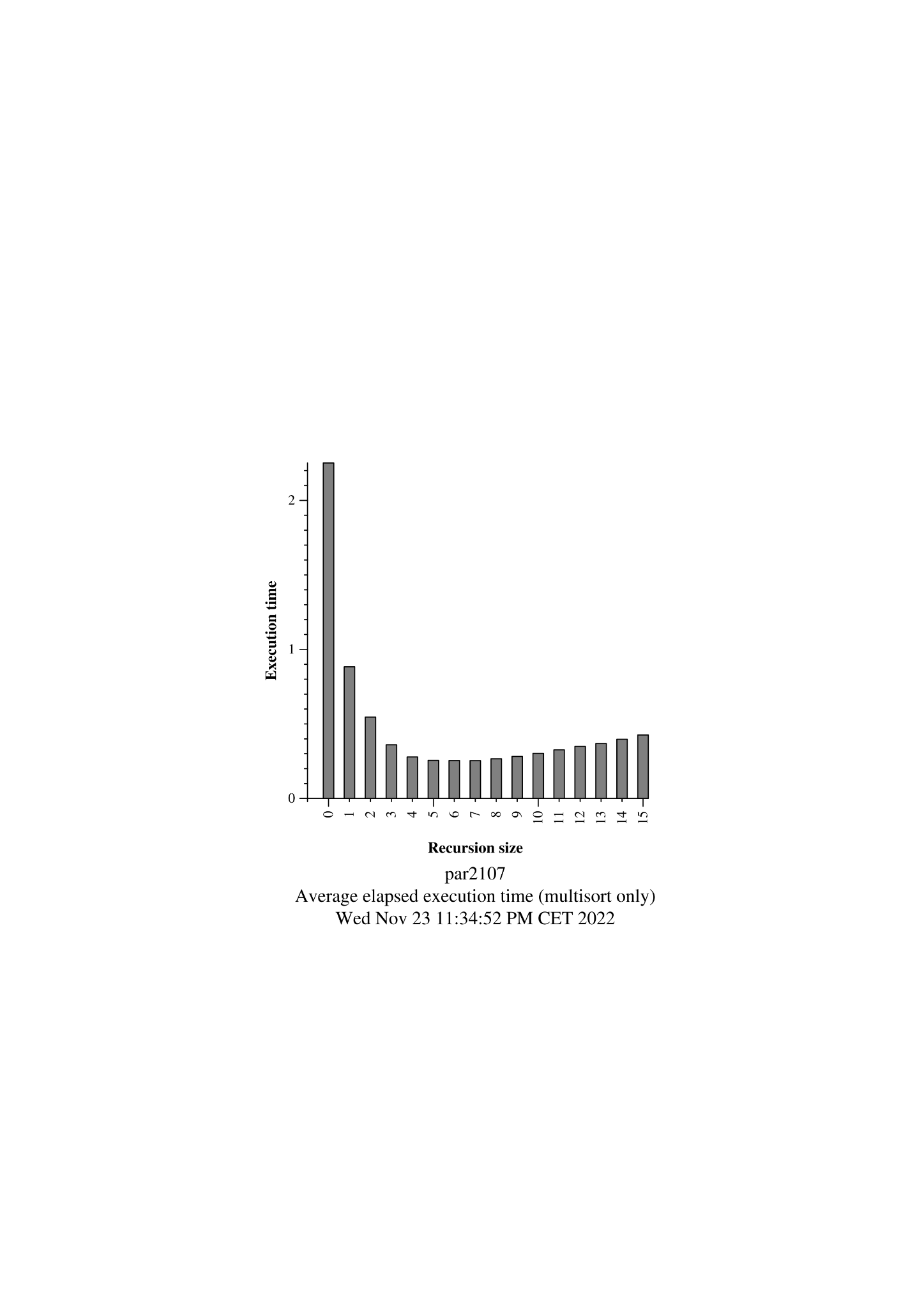




Using cutoff = 4 and 8 threads, we get the results shown above, much better than those we have seen so far. Overheads per task have been reduced heavily, down from up to 80% to just 16 or less. All in all very optimistic results.

# 

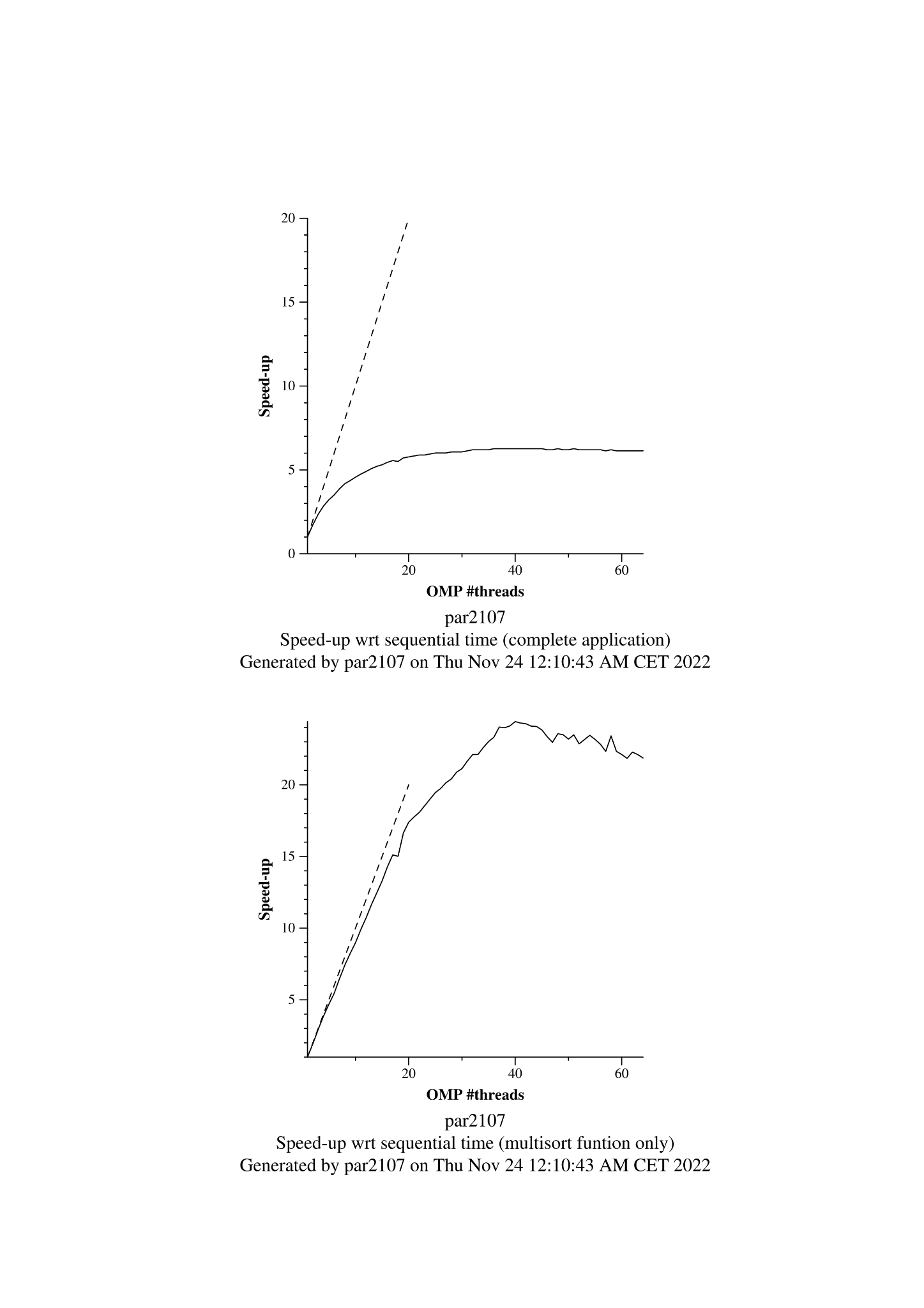
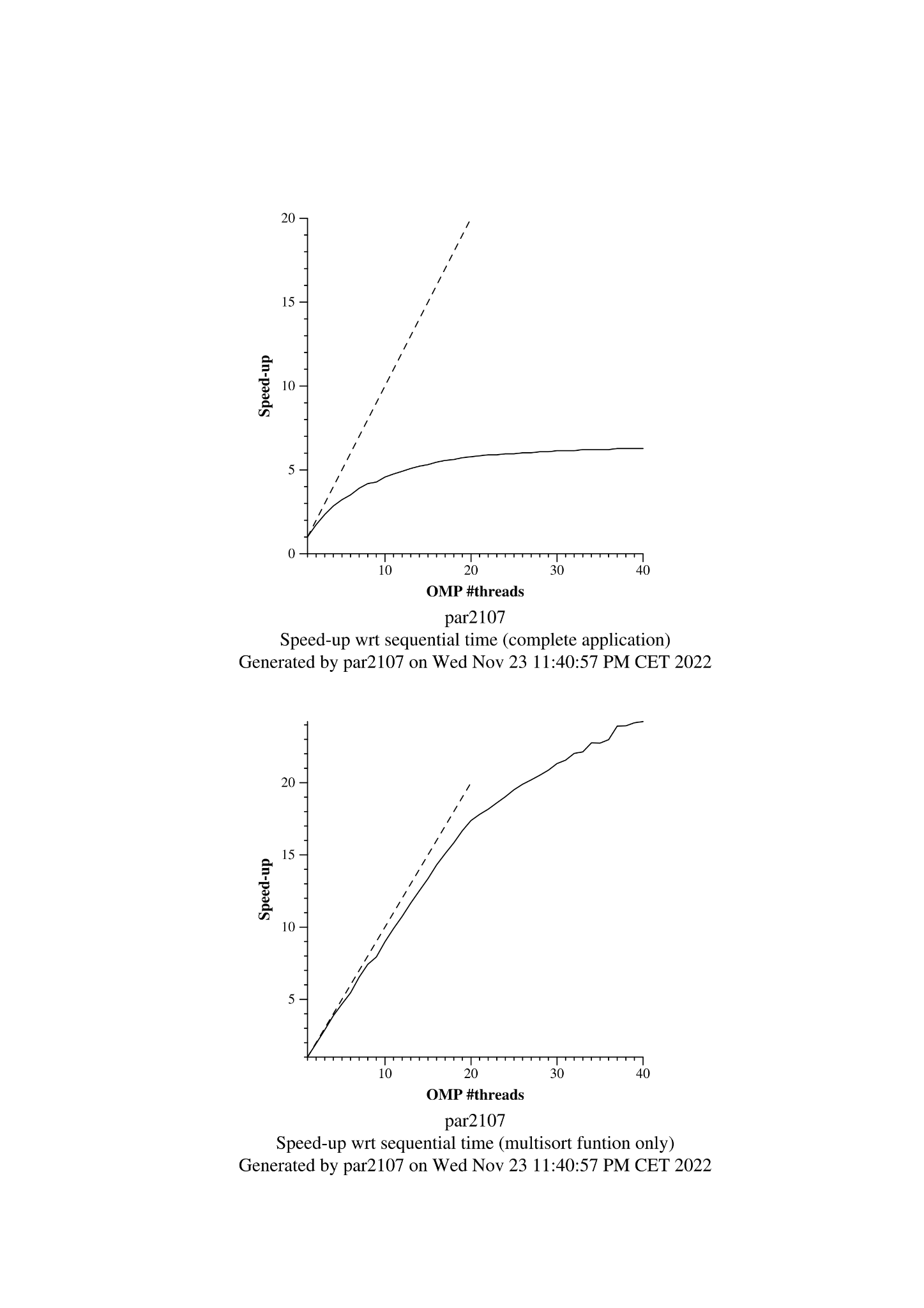
# Optional 1:



Best cutoff for 40 threads seems to be 5 to 7.

We have used sbatch submit-strong-omp.sh with cutoff = 6 and max processors = 40.

We can clearly see that the speedup keeps increasing even though there are only 20 physical cores available. This is because each boada core has two threads we can confirm this by running it for 64 processors for example.

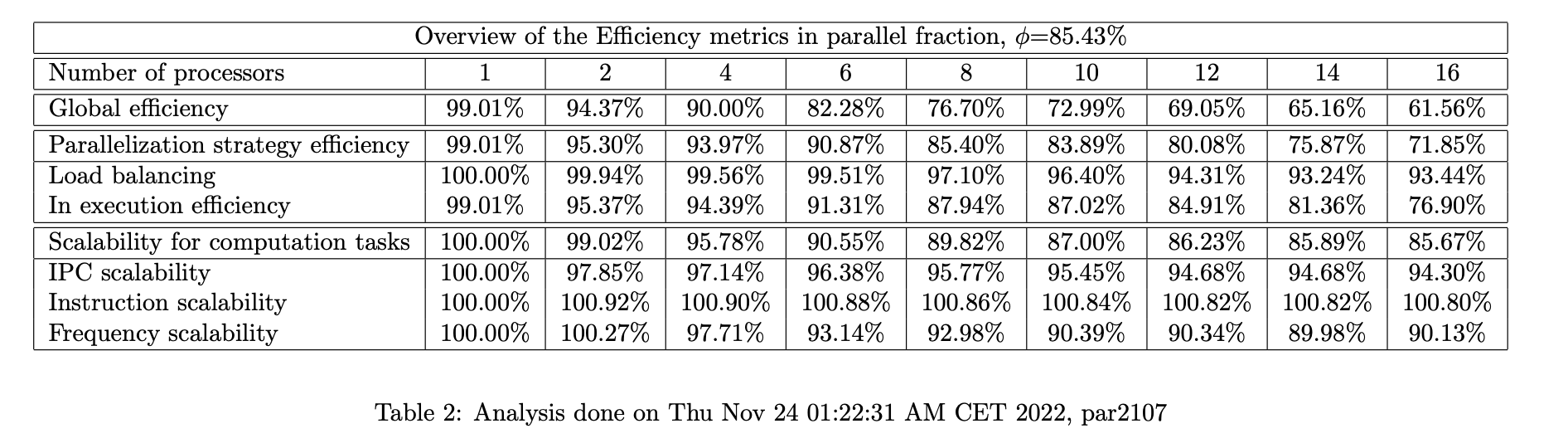
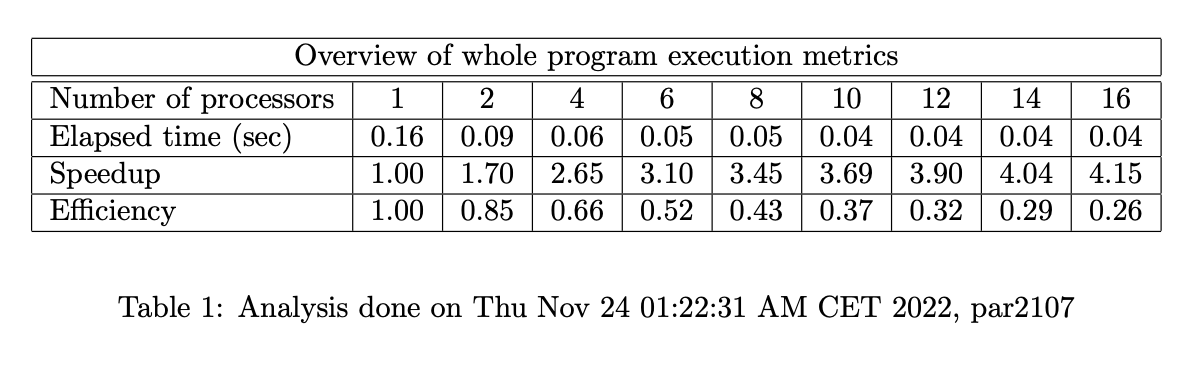


We can see that performance clearly spikes at 40 threads, which is what we expected, given that boada nodes have 2 threads each.

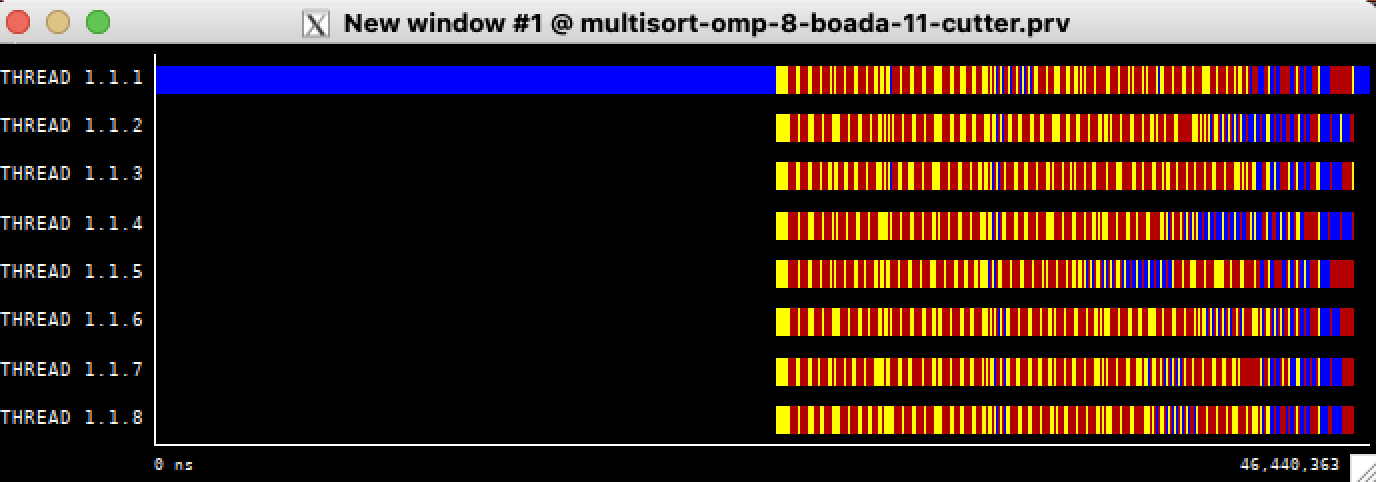
# 4 Shared-memory parallelisation with OpenMP task using dependencies

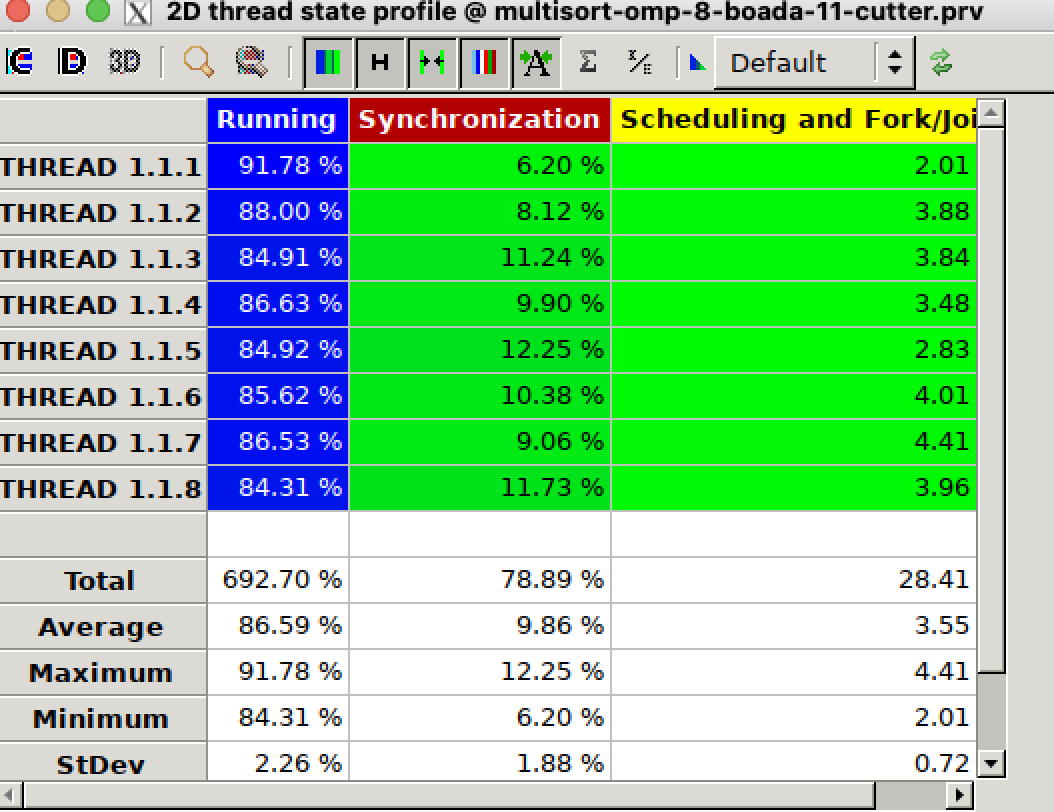
In order to ensure it worked as intended for **8** threads, we ran the command: sbatch submit-omp.sh multisort-omp 8

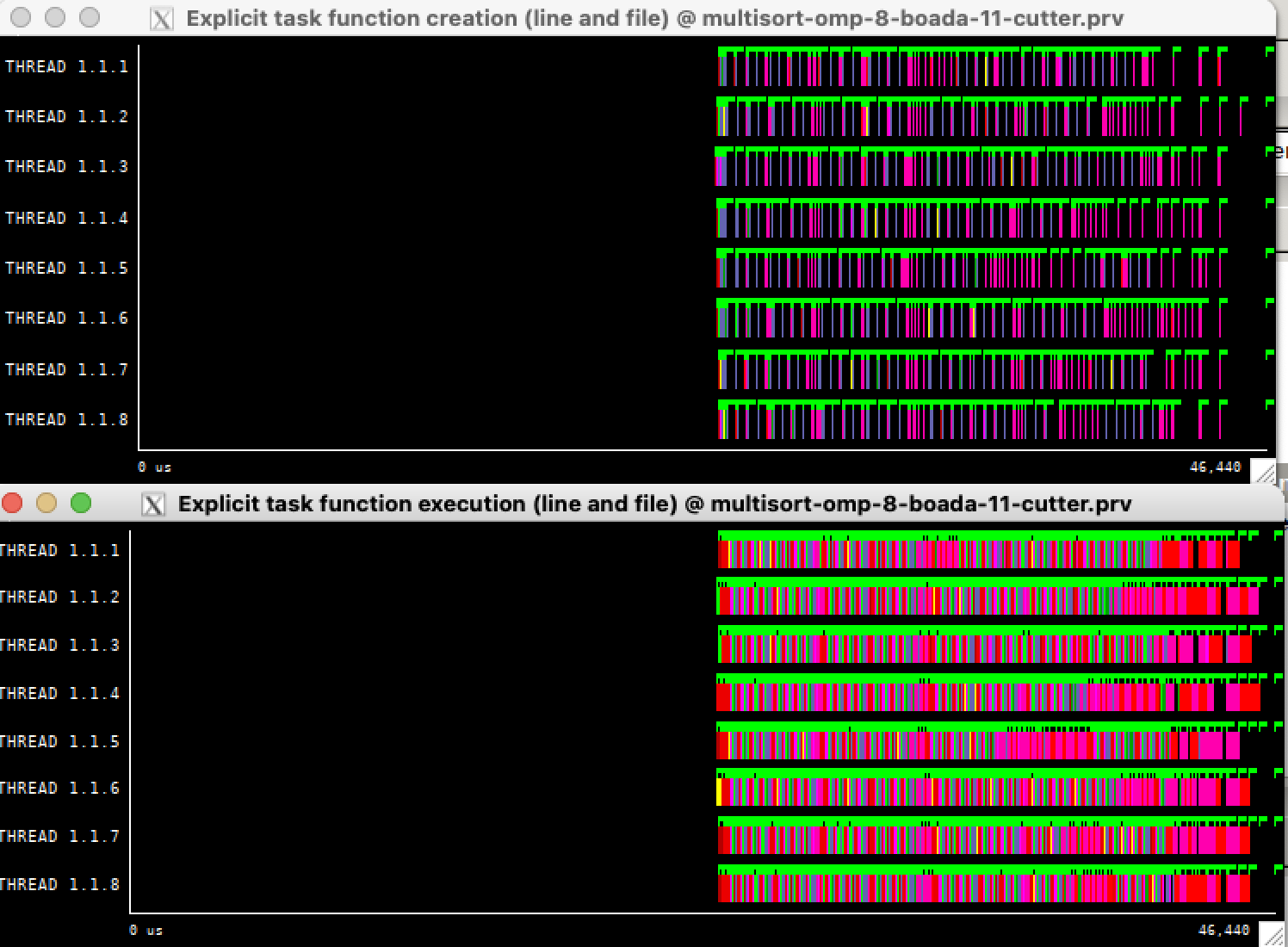
| void merge(long n, T left[n], T right[n], T result[n\*2], long start, long length, int depth) {  if (length < MIN\_MERGE\_SIZE\*2L) {  // Base case  basicmerge(n, left, right, result, start, length);  } else {  // Recursive decomposition  if(!omp\_in\_final()) {  #pragma omp task final(depth >= CUTOFF)  merge(n, left, right, result, start, length/2, depth+1);  #pragma omp task final(depth >= CUTOFF)  merge(n, left, right, result, start + length/2, length/2,depth+1);  #pragma omp taskwait  }  else {  merge(n, left, right, result, start, length/2, depth+1);  merge(n, left, right, result, start + length/2, length/2,depth+1);  }  } }  void multisort(long n, T data[n], T tmp[n], int depth) {  if (n >= MIN\_SORT\_SIZE\*4L) {  // Recursive decomposition  if(!omp\_in\_final()) {  #pragma omp task final(depth >= CUTOFF) depend(out:data[0])  multisort(n/4L, &data[0], &tmp[0], depth+1);  #pragma omp task final(depth >= CUTOFF) depend(out:data[n/4L])  multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);  #pragma omp task final(depth >= CUTOFF) depend(out:data[n/2L])  multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);  #pragma omp task final(depth >= CUTOFF) 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) 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) 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) 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 {  multisort(n/4L, &data[0], &tmp[0], depth+1);  multisort(n/4L, &data[n/4L], &tmp[n/4L], depth+1);  multisort(n/4L, &data[n/2L], &tmp[n/2L], depth+1);  multisort(n/4L, &data[3L\*n/4L], &tmp[3L\*n/4L], depth+1);   merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L, depth+1);  merge(n/4L, &data[n/2L], &data[3L\*n/4L], &tmp[n/2L], 0, n/2L, depth+1);   merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n, depth+1);   }  } else {  // Base case  basicsort(n, data);  } } |
| --- |



As we can see in these tables, performance has not improved at all, but it also hasn’t gone down, which is not necessarily a bad thing.

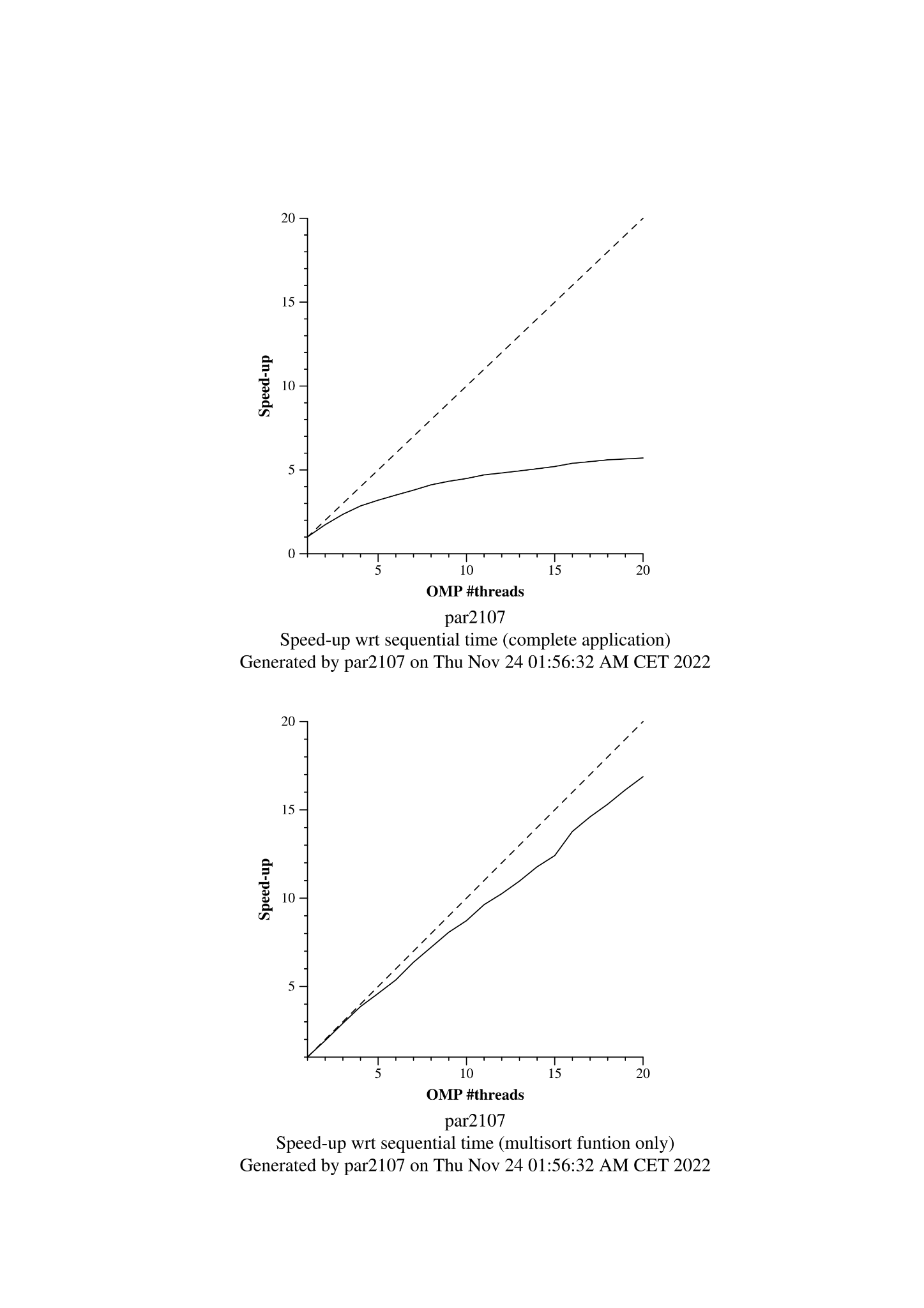






From the paraver instantaneous parallelism, we can see that almost all the time it’s taking advantage of all of those 8 cores to parallelize. Synchronization overheads and scheduling and forking seems to be similar to previous results as well.

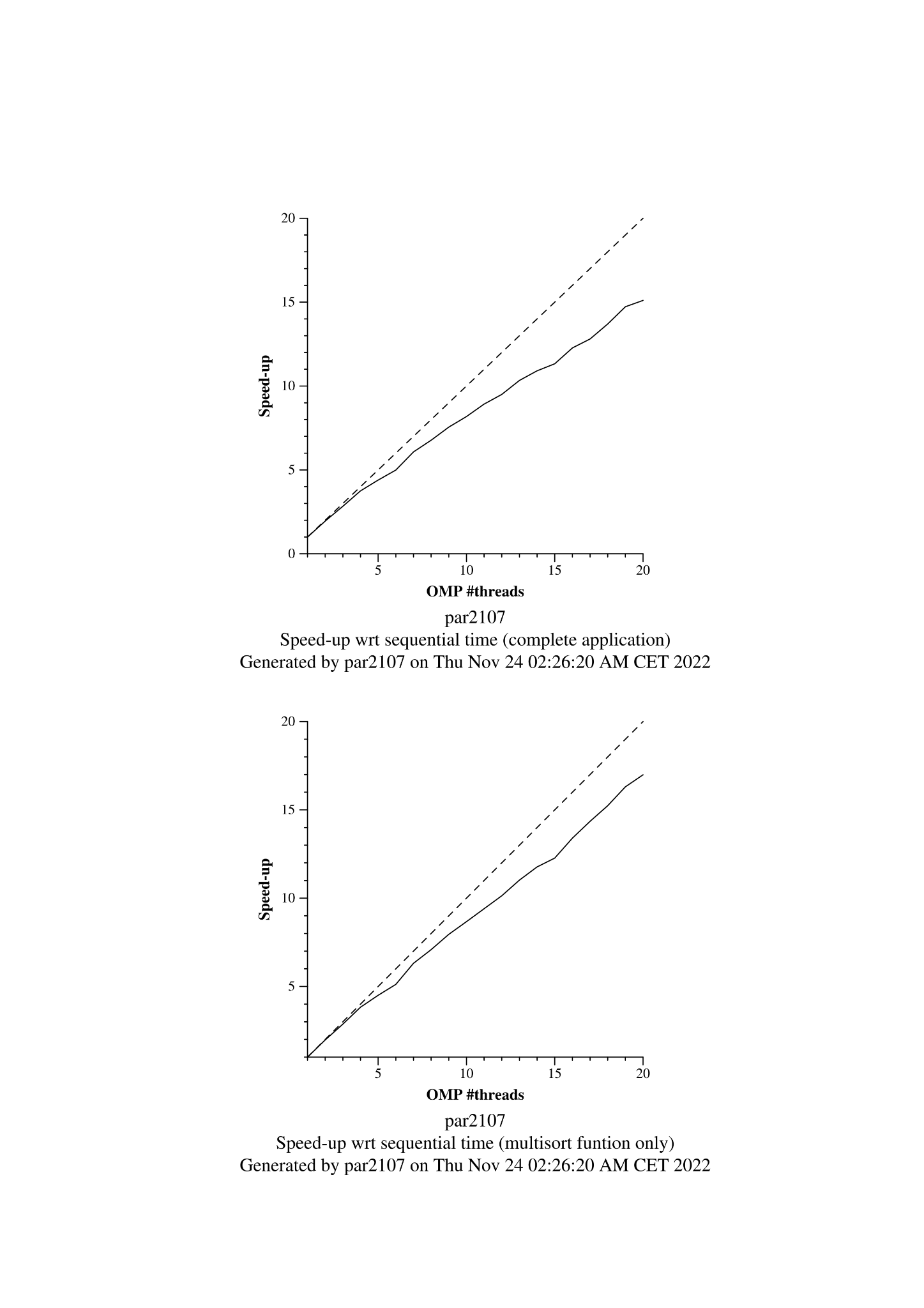
In terms of programmability and then again, this might be personal preference, but it was harder to program than just using taskgroups and taskwaits all around, it makes you have to be much more specific and accurate about what you want and how you want it which i guess could be useful in a more complex scenario so that you don’t miss any edge cases, but in this case and this is personal opinion, I would probably go with the previous version of the code.

 In this graphic, we can also see that it is very similar to what we saw on the previous iteration of our code, meaning we didn’t gain or lose much from using depend clauses.

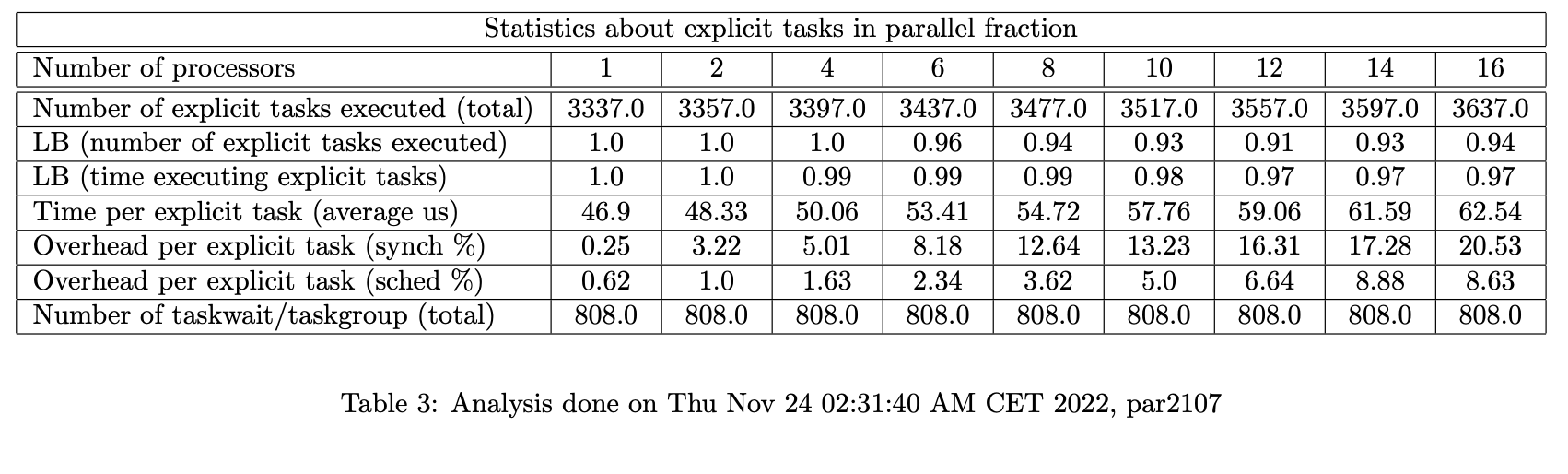
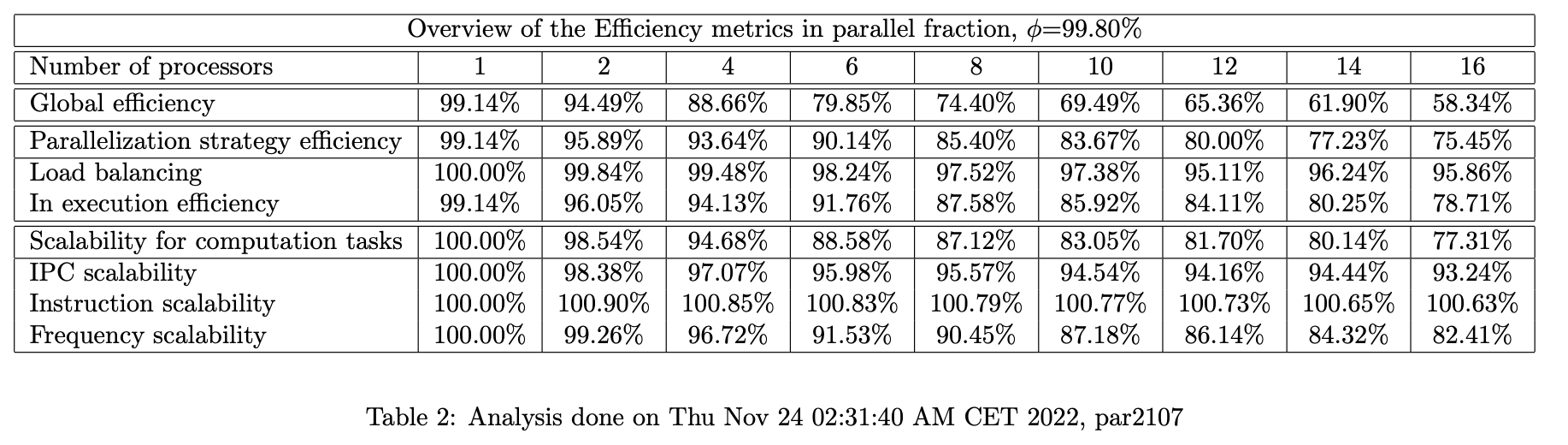
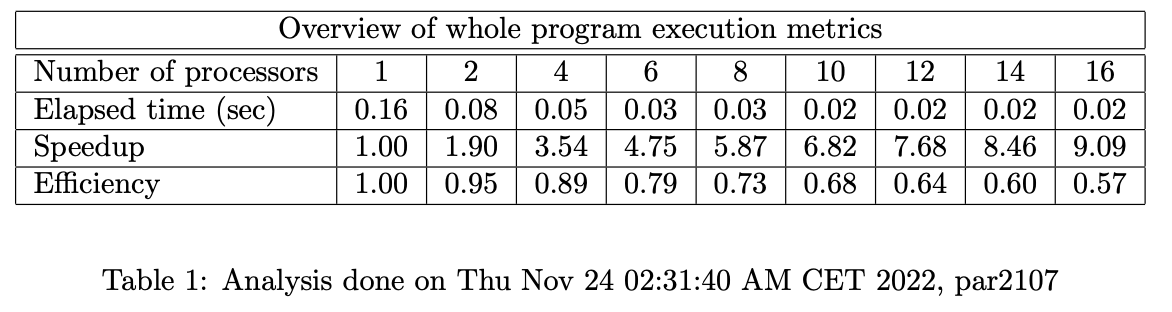
# Optional 2:

We tested our code with: *sbatch ./****submit-omp****.sh multisort-omp* **8**

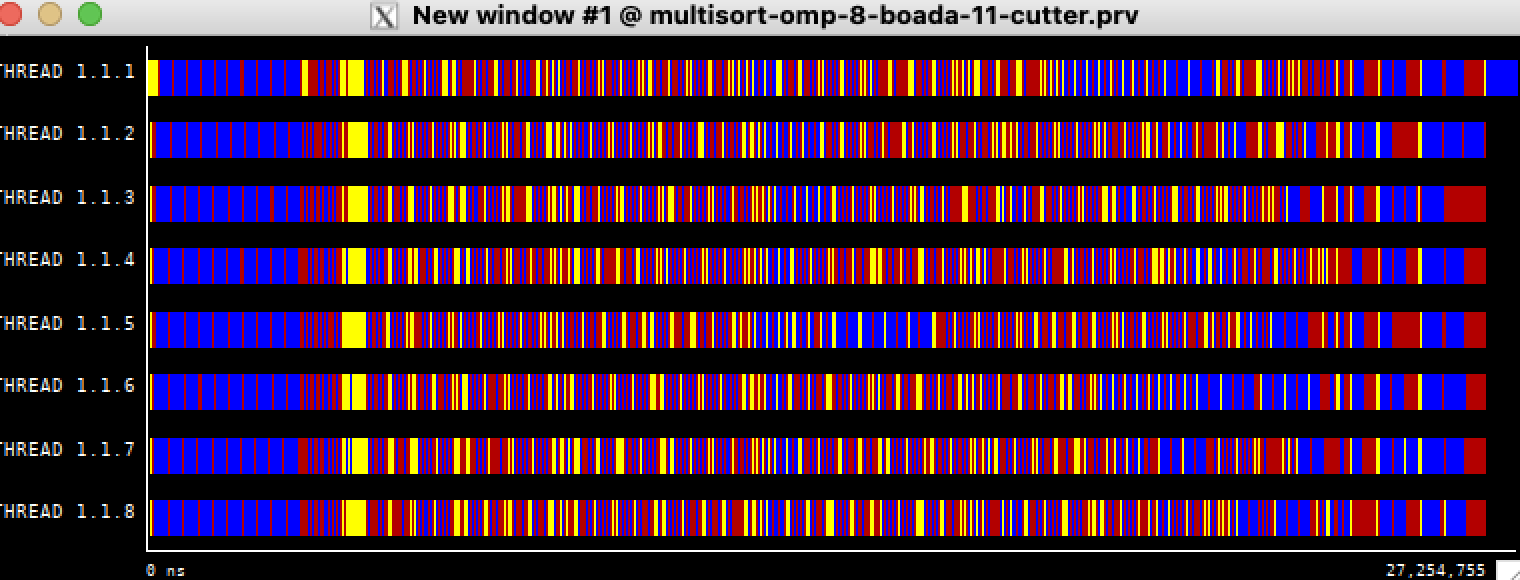
| static void initialize(long length, T data[length]) {  long i;  #pragma omp taskloop  for (i = 0; i < length; i++) {  if (i==0) {  data[i] = rand();  } else {  data[i] = ((data[i-1]+1) \* i \* 104723L) % N;  }  } }  static void clear(long length, T data[length]) {  long i;  #pragma omp taskloop  for (i = 0; i < length; i++) {  data[i] = 0;  } }  int main(int argc, char \*\*argv) { ...  #pragma omp parallel  #pragma omp single  {  initialize(N, data);  clear(N, tmp);  } ... } |
| --- |



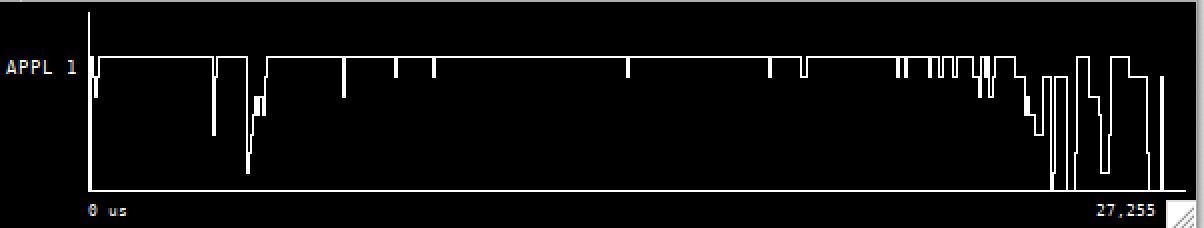
We can clearly see a massive improvement on the speedup of the complete application, meaning this change to the code has greatly increased our performance, let’s have a look at a couple of paraver hints to see why that is.



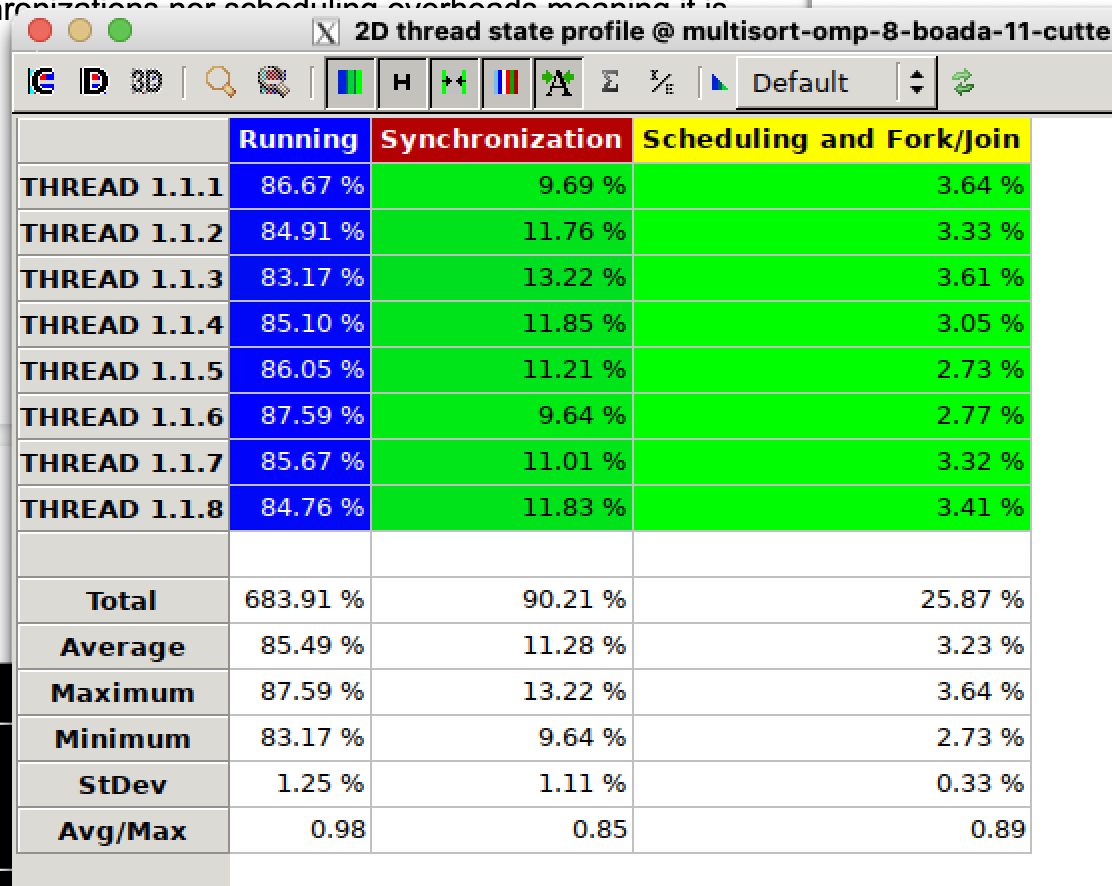
We can immediately see one of the reason’s of the much better performance, the parallel fraction is considerably higher with up to 99.8%.



Immediately we see this in our paraver hints, much higher parallelization. Also in the first part of the code there aren’t many synchronizations nor scheduling overheads meaning it is very efficient.



We can see that all cores are now being used on that first section of the code where before there wasn’t any.



Similar results in this table to our previous version of the code.

# Final Conclusions

All in all this has been a very instructive view on the subject, diving deep on the parallelization strategies leaf and tree, which, to be honest we might have understood the concept but when you have to get down to it and get your hands dirty it is a whole different story.

As to the dependency oriented method of using tasks, it seems like a powerful tool and very versatile, but for this particular purpose I might say it was not necessary to speed up our code.