**PAR Laboratory Assignment Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set**

Oscar Faixat Llanas

Xavier Algarra Torelló

Grup: PAR1212

PAR 2017Q2

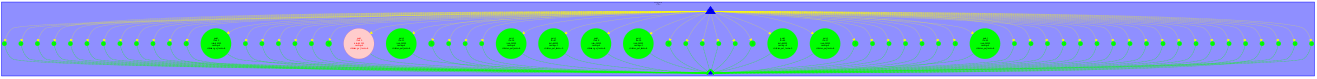
06/04/2017

# **6.1 Task granularity analysis**

## **Which are the two most important common characteristics of the task graphs generated for the two task granularities (Row and Point) for the non-graphical version of mandel-tareador? Obtain the task graphs that are generated in both cases for -w 8.**

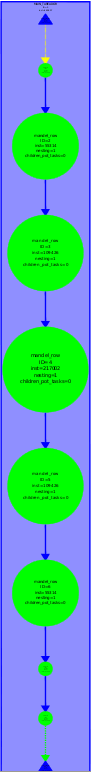
## As we can see from the figures below, the tasks don’t have dependencies between them. Therefore they can be parallelized without any problems. We do have to note that in both cases the work that each task does is not balanced evenly between all of the tasks. This means that the speed-up will not end up scaling properly.

## **Row Granularity**Screenshot_20170419_194626.png



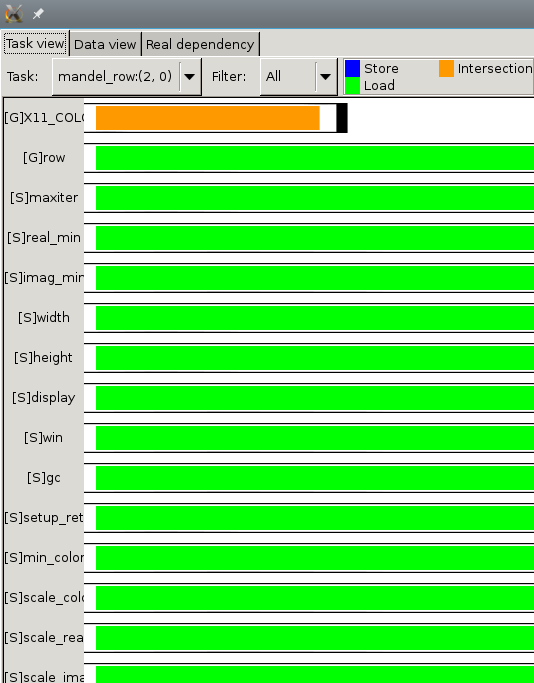
**Point Granularity**

## **Which section of the code is causing the serialization of all tasks in mandeld-tareador? How do you plan to protect this section of code in the parallel OpenMP code?**

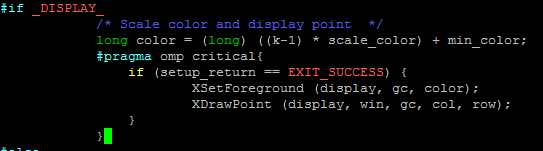


**Tareador sequential display for Row**

The calls to functions XSetForeground and XDrawPoint are the ones causing the serializations of all tasks. This happens when they read or modify the variable X11\_COLOR\_FAKE. The image below is a screenshot of the tasks tab from tareador where it indicates there is an intersection with the varibale X11\_COLOR\_fake.



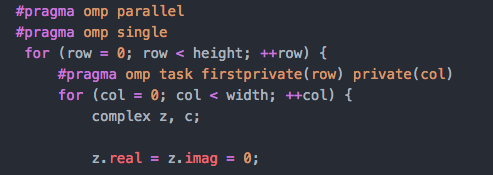
To protect this section of code in openmp, we can encapsulate the code region with the pragma omp critical. The resulting code would be:



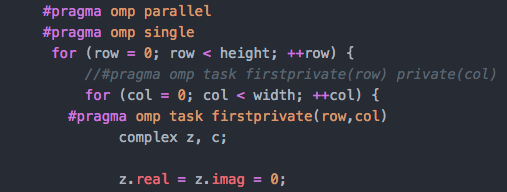
Note: We could also use the pragma atomic

# **6.2 OpenMP task–based parallelization**

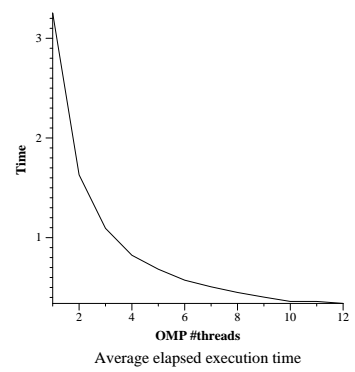
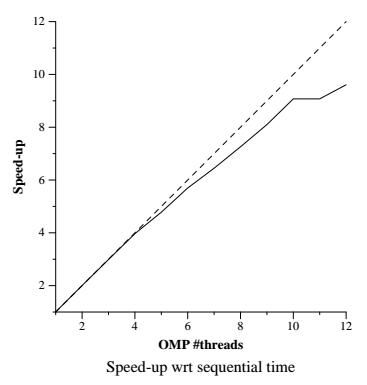
## **For the Row and Point decompositions of the non-graphical version, include the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000). Reason about the causes of good or bad performance in each case.**

The code for row:

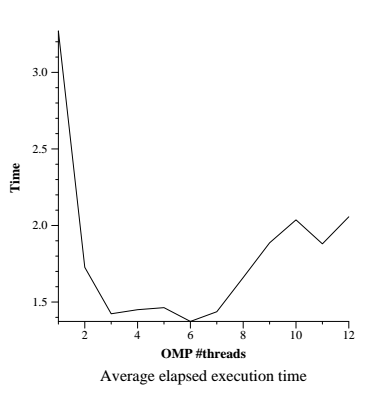
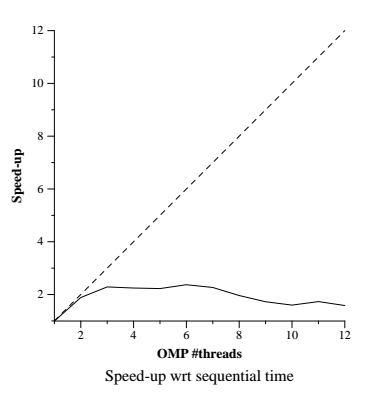
The code for point:



The row execution time and speedup plots:



The point execution time and speedup plots:



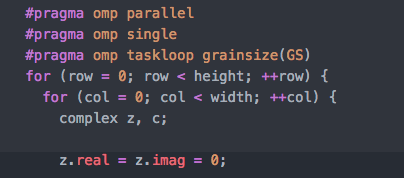
As we can see from the strong scalability of the row granularity above, this one executes almost ideally as it almost completely models the ideal function (dashed lines). However, because of having to manage threads and the unbalanced workload that we mentioned before the scalability is not fully ideal.

On the other hand, for the point granularity above the speedup is not at all ideal. We can observe that the overhead that is caused when managing all of the created tasks, makes it big enough to end up having worse speed-ups when dealing with a higher number of threads. We can conclude that in this case the point granularity is too fine-grained and as such parallelising the tasks isn’t compensating for the overhead of creating and managing them.

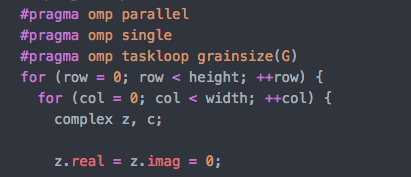
# **6.3 OpenMP taskloop–based parallelization**

# **1. For the Row and Point decompositions of the non-graphical version, include the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000). Reason about the causes of good or bad performance in each case.**

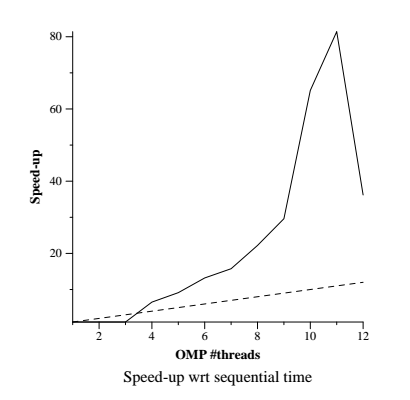
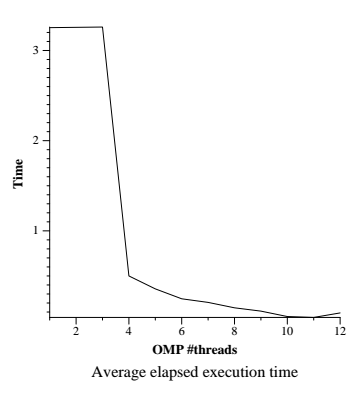
The code for row:

****

The code for point:

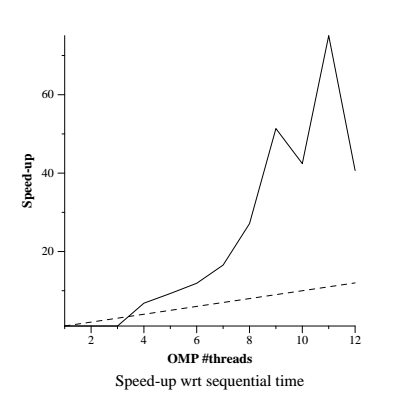


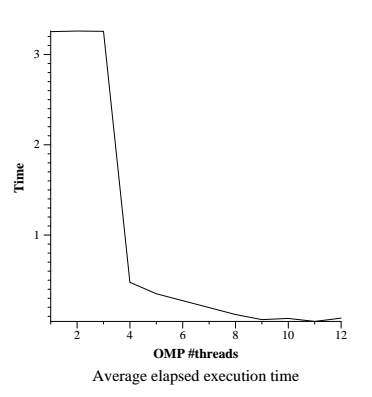
The row execution time and speedup:



# 

# 

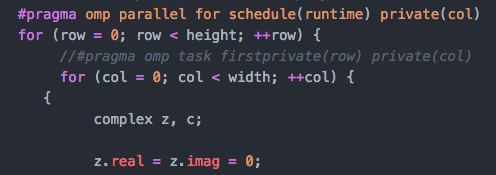


The point execution time and speedup:

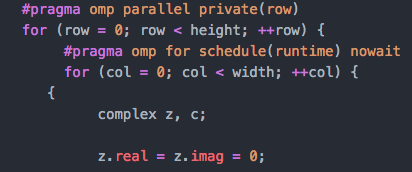
As we can observe from the execution times of both point and row granularity it seems like both greatly benefit from using task-loop distribution. It is worth noting that with few processors the execution does not benefit at all, it is only when using more than 4 cores that it starts being useful. Regarding the speedup, it is more than lineal which is impressive to say the least. It is clear that this distribution greatly benefits the execution of the program. However, once it reaches 12 processors, the speedup quickly decreases as the overheads are probably too big and thus they can’t be compensated.

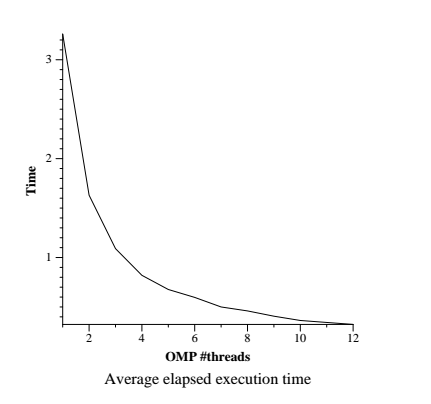
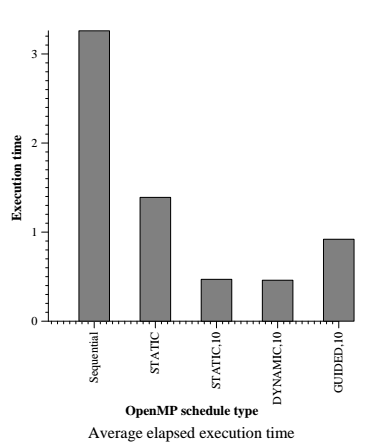
**6.4 OpenMP for–based parallelization**

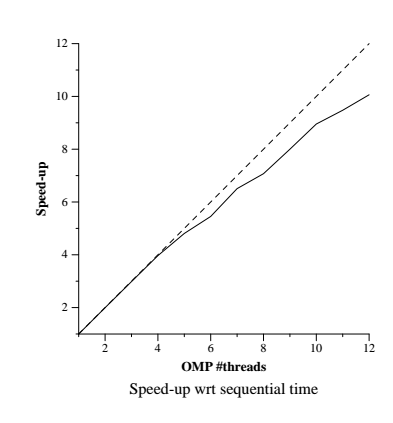
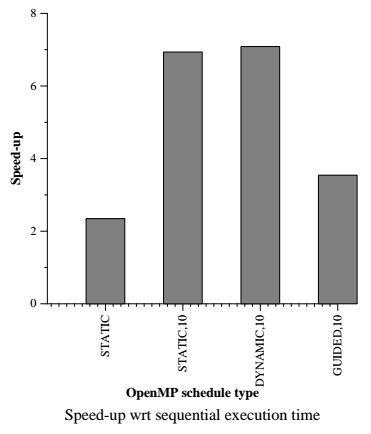
**1. For the the Row and Point decompositions of the non-graphical version, include the execution time and speed–up plots that have been obtained for the 4 different loop schedules when using 8 threads (with -i 10000). Reason about the performance that is observed.**

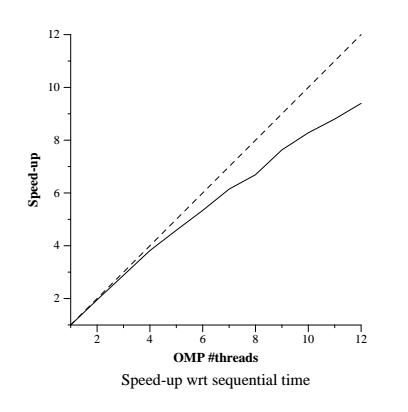
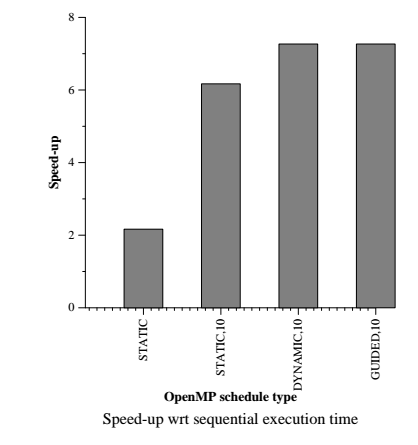
The code for row:

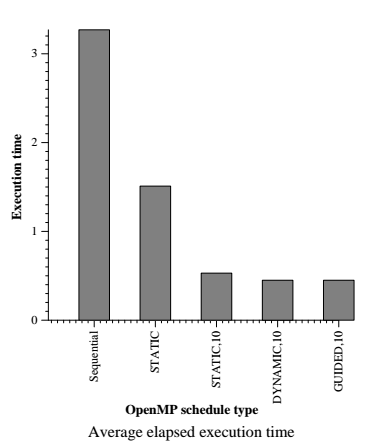
The code for point:

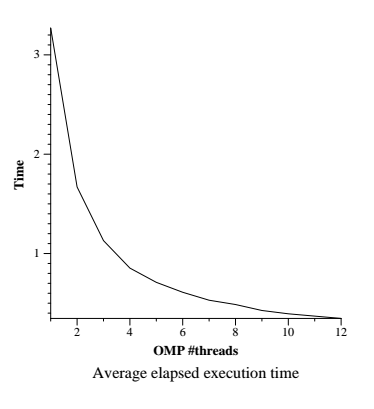
****







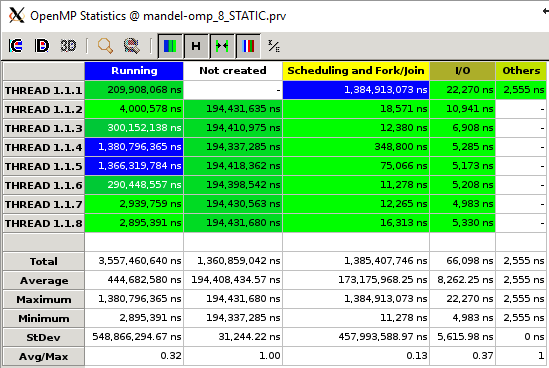




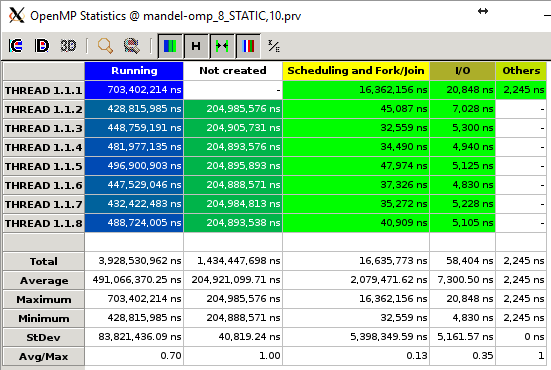
**2. For the Row parallelization strategy, complete the following table with the information extracted from the Extrae instrumented executions (with 8 threads and -i 10000) and analysis with Paraver, reasoning about the results that are obtained.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **static** | **static,10** | **dynamic,10** | **guided,10** |
| Running average time per thread | 444,682,580 | 491,066,370.25 | 476,638,739.62 | 453,463,327.38 |
| Execution unbalance (average time divided by maximum time) | 0.32 | 0.70 | 0.71 | 0.49 |
| SchedForkJoin (average time per thread or time if only one does) | 173,175,968.25 | 2,079,471.62 | 1,752,683.50 | 106,161,766.75 |

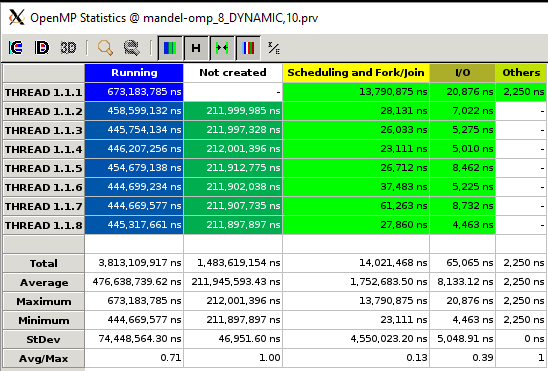
From the table, we can conclude that the execution obtains a better performance when the unbalance is closer to 1. In this case, that would be static 10 and dynamic 10. To get a better idea of these results and look at the data more closely we have included the paraver screenshots of each of the schedules below.



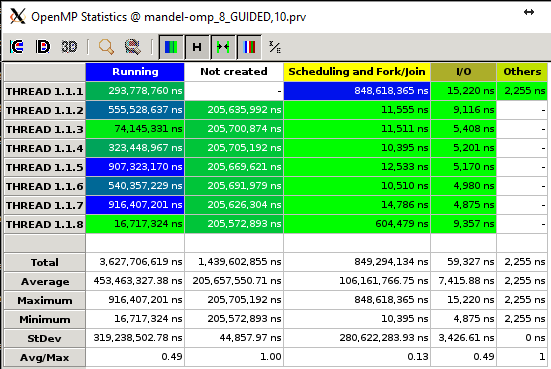
**Static statistics**

****

**Static, 10 statistics**

****

**Dynamic, 10 statistics**

****

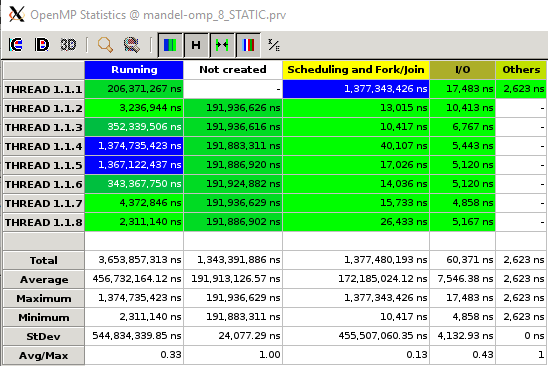
**Guided, 10 statistics**

**6.5 Optional**

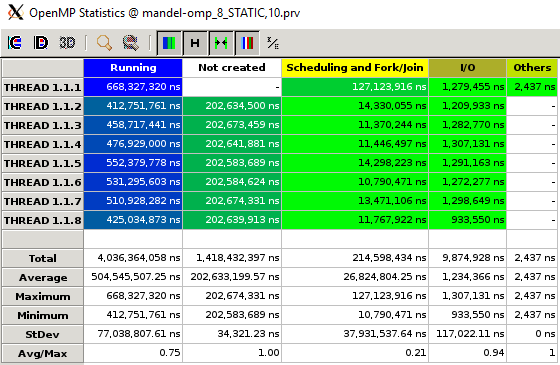
1. **If you have done any of the optional parts in this laboratory assignment, please include your experience, additional information collected or the relevant portion of the code and performance plots obtained in your report.**

**Optional 1.**

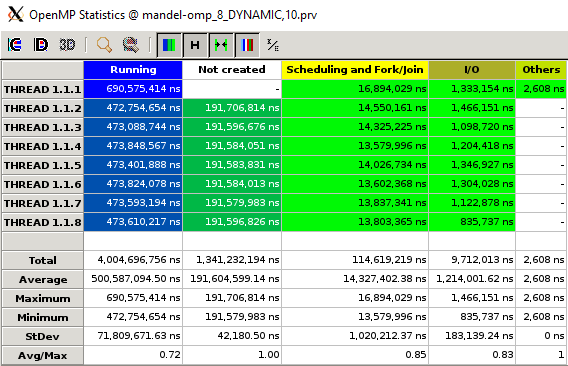
From what we can observe, using the collapse clause doesn’t give a very good result, in general the execution times have worsened. This has happened because the Scheduling and fork/join times have generally worsened for all the executions. Below, you can observe the screenshots we took from paraver

****

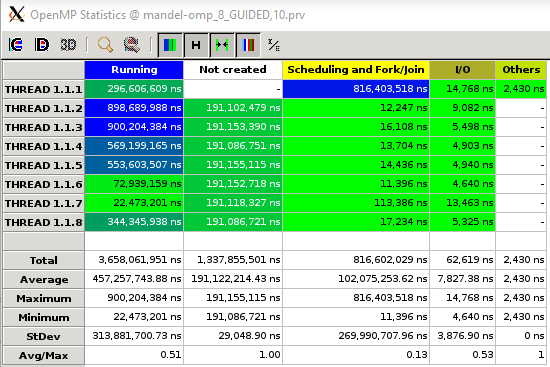
**Static statistics**

****

**Static, 10 statistics**

****

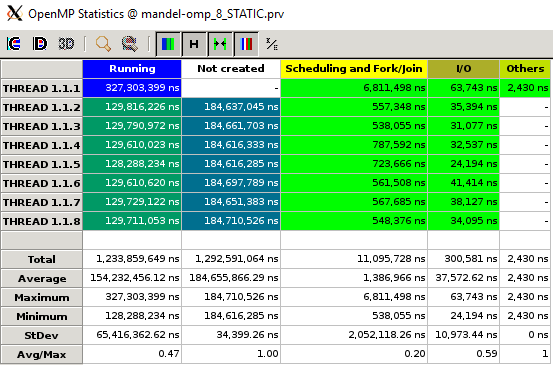
**Dynamic, 10 statistics**

****

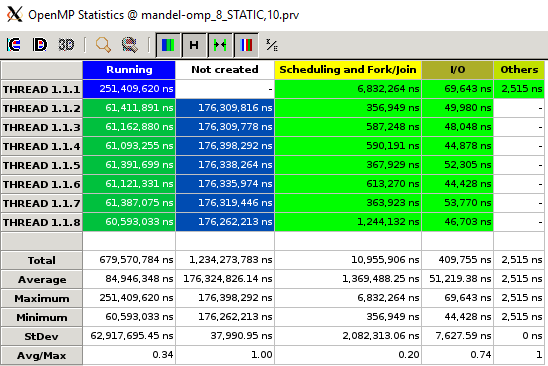
**Guided, 10 statistics**

**Optional 2.**

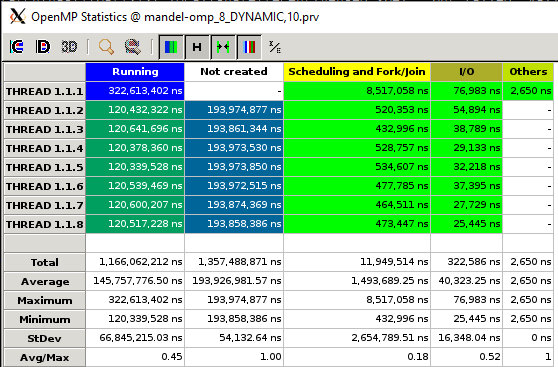
Contrary to what we observed in the optional 1, it seems that when combining the for and task clauses we get much better execution times than if we use the collapse clause like in the optional 1. In general, we also get much better execution times than with the the previous exercise.

****

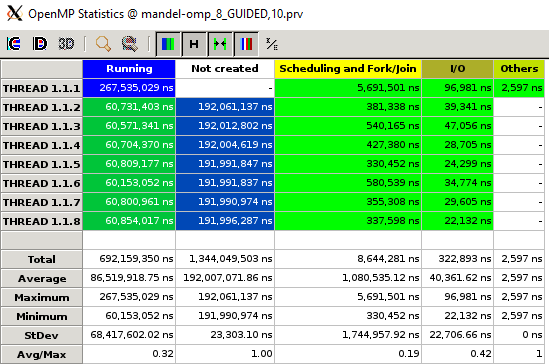
**Static, statistics**

****

**Static, 10 statistics**

****

**Dynamic, 10 statistics**

****

**Guided, 10 statistics**