UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

LAB 4: Parallel Task Decomposition Implementation and Analysis

Arnau Garcia Gonzalez
Jan Santos Bastida

Table of contents

[1] Note: The analysis of each image is done underneath the same.

Iterative task decomposition

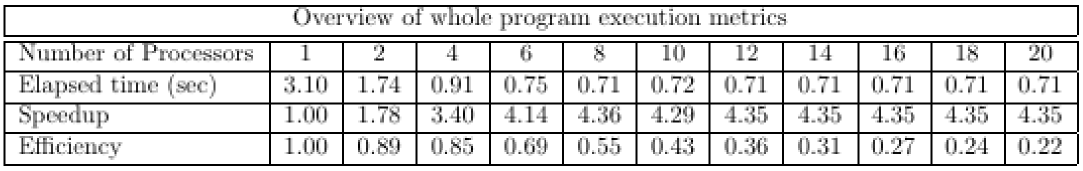
Iterative: Tiled

<u>Code</u>

The file name is mandel-omp-iter-tile.c

To obtain the desired parallelism, aside from creating the explicit tasks on the code we had to protect some variables. First of all, we had to protect the variable histogram, to do so we added a #pragma omp atomic. After that, we had to protect X_COL11, to protect this variable we had to use #pragma omp critical since we use it for complex operations.

<u>Modelfactor analysis</u>

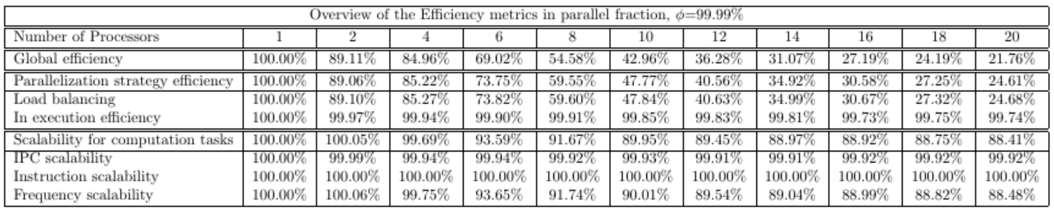


Overview of whole program execution metrics											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	3.10	1.74	0.91	0.75	0.71	0.72	0.71	0.71	0.71	0.71	0.71
Speedup	1.00	1.78	3.40	4.14	4.36	4.29	4.35	4.35	4.35	4.35	4.35
Efficiency	1.00	0.89	0.85	0.69	0.55	0.43	0.36	0.31	0.27	0.24	0.22

Table 1: Analysis done on Fri Nov 8 11:30:58 AM CET 2024, par2110

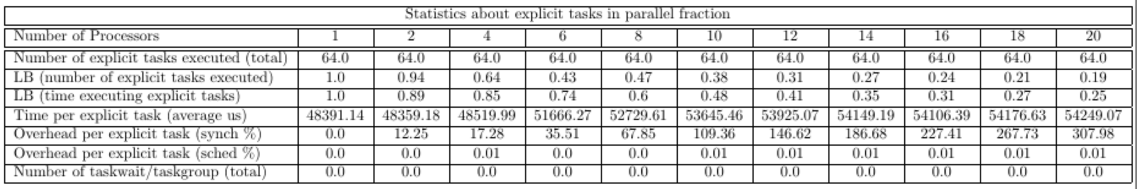
Modelfactor Table 1

If we analyze this first table we can see how the efficiency and the speedup increase by increasing the number of processors. We notice that when we have up to 6 processors we start losing a lot of efficiency and the improvement we get on the speedup is not very high from that point so this strategy, although it works is not very efficient nor effective.



Overview of the Efficiency metrics in parallel fraction, ϕ=99.99%											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	100.00%	89.11%	84.96%	69.02%	54.58%	42.96%	36.28%	31.07%	27.19%	24.19%	21.76%
Parallelization strategy efficiency	100.00%	89.06%	85.22%	73.75%	59.55%	47.77%	40.56%	34.92%	30.58%	27.25%	24.61%
Load balancing	100.00%	89.10%	85.27%	73.82%	59.60%	47.84%	40.63%	34.99%	30.67%	27.32%	24.68%
In execution efficiency	100.00%	99.97%	99.94%	99.90%	99.91%	99.85%	99.83%	99.81%	99.73%	99.75%	99.74%
Scalability for computation tasks	100.00%	100.05%	99.69%	93.59%	91.67%	89.95%	89.45%	88.97%	88.92%	88.75%	88.41%
IPC scalability	100.00%	99.99%	99.94%	99.94%	99.92%	99.93%	99.91%	99.91%	99.92%	99.92%	99.92%
Instruction scalability	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%	100.00%
Frequency scalability	100.00%	100.06%	99.75%	93.65%	91.74%	90.01%	89.54%	89.04%	88.99%	88.82%	88.48%

Table 2: Analysis done on Fri Nov 8 11:30:58 AM CET 2024, par2110



Statistics about explicit tasks in parallel fraction											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Number of explicit tasks executed (total)	64.0	64.0	64.0	64.0	64.0	64.0	64.0	64.0	64.0	64.0	64.0
LB (number of explicit tasks executed)	1.0	0.94	0.64	0.43	0.47	0.38	0.31	0.27	0.24	0.21	0.19
LB (time executing explicit tasks)	1.0	0.89	0.85	0.74	0.6	0.48	0.41	0.35	0.31	0.27	0.25
Time per explicit task (average us)	48391.14	48359.18	48519.99	51666.27	52729.61	53645.46	53925.07	54149.19	54106.39	54176.63	54249.07
Overhead per explicit task (synch %)	0.0	12.25	17.28	35.51	67.85	109.36	146.62	186.68	227.41	267.73	307.98
Overhead per explicit task (sched %)	0.0	0.0	0.01	0.0	0.0	0.01	0.01	0.01	0.01	0.01	0.01
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 3: Analysis done on Fri Nov 8 11:30:58 AM CET 2024, par2110

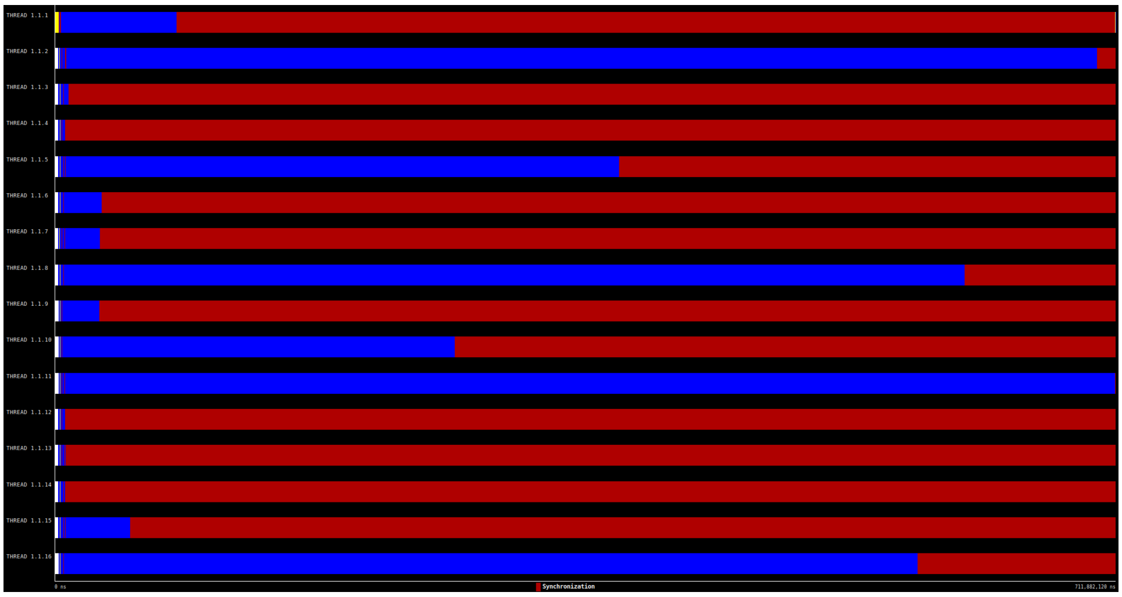
Modelfactor Table 2&3

In the second Table, we can see again the decrease in efficiency. Here we can notice the reasons behind that. By looking at load balancing we know it goes down very fast, especially when executing with more than four processors. That is because we have some big and some small tasks, probably by making smaller tasks we would decrease this unbalance.

In the third table, we can see another cause of inefficiency. Specifically, each task is executed for a long time, confirming again what we thought about the unbalance being caused by the size of the tasks. The high synchronization overhead also attracts our attention.

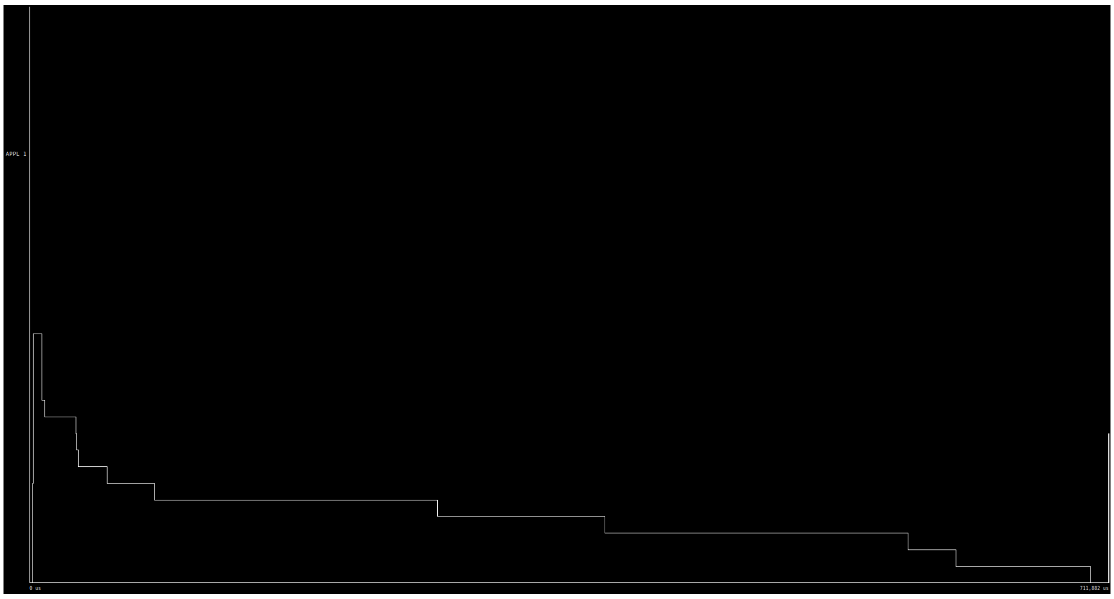
In conclusion, we can see by analyzing Modelfactor's tables that with this strategy we obtain poor efficiency, caused by the load unbalance (as a consequence of the size of the tasks) and the high task synchronization overhead.

Paraver analysis



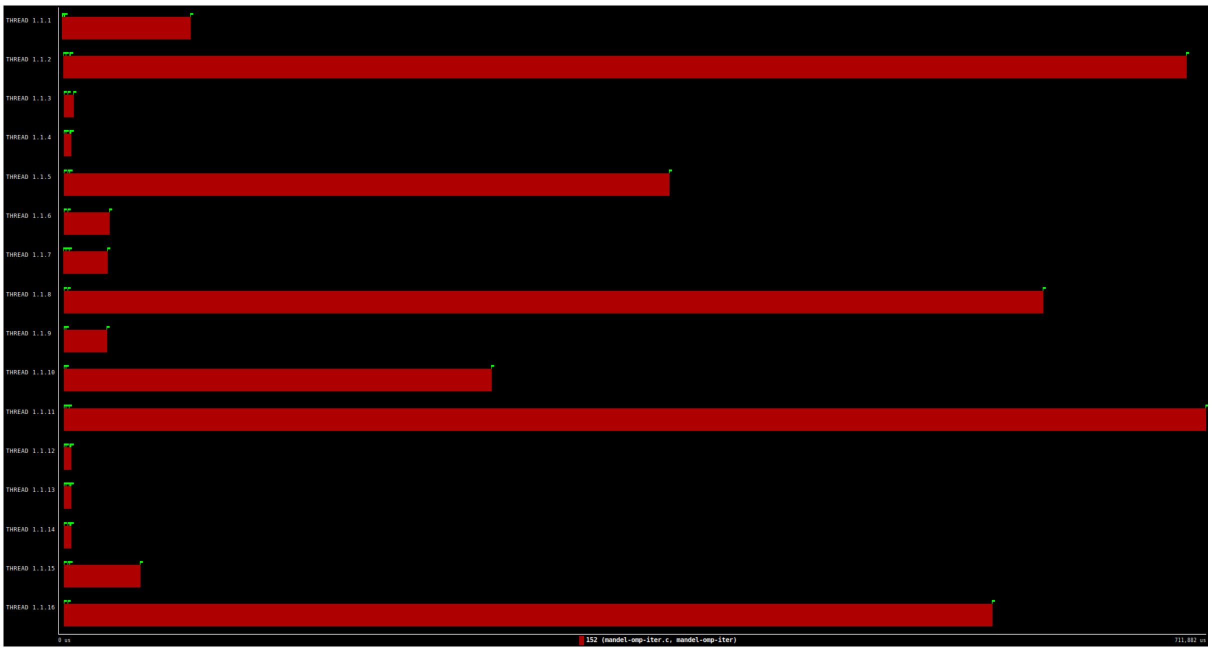
Paraver execution 16 threads →711.882.120 ms

By looking at paraver execution we can see that most processors spend most of the time not executing any task because some tasks take little time while one task acts as a bottleneck, making other processors useless.



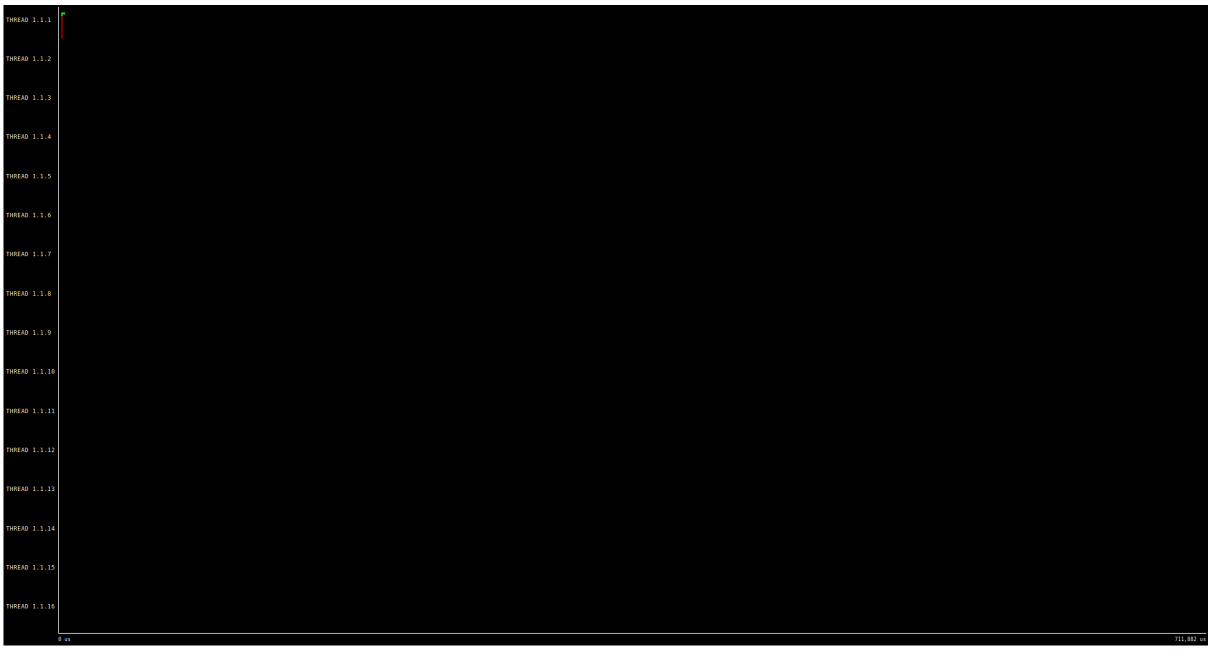
Paraver hint instantaneous parallellism

On the Paraver instantaneous parallelism, we can see that parallelism is not maintained at all through the execution. This shows us again that tasks are unbalanced and for this reason, processors don't work all the time, only some do.



Paraver hint explicit task function execution

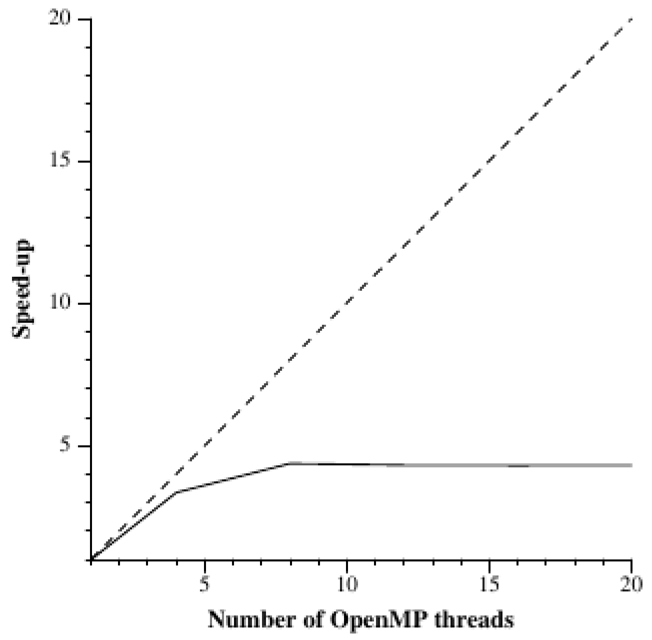
This graph offers a clear view of how some tasks execute for much more time than others, again proving the load inbalance.



Paraver hint explicit task function created

In this strategy, we don't have a high task creation overhead so this graph is not very useful for the analysis.

Strong scalability



par2110
Speed-up wrt sequential time (mandel funtion only)
Generated by par2110 on Wed Nov 27 07:42:58 PM CET 2024

Strong scalability graph

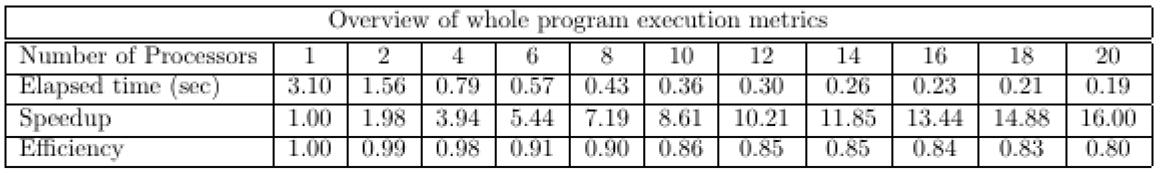
The graph of strong scalability shows us that speedup is almost constant at 4 processors and more. We can see that the obtained speedup is nowhere close to the ideal. However, it’s important to point out that this strategy has an acceptable performance with a low number of processors.

Iterative: Finer grain

<u>Code</u>
The file name is mandel-omp-iter-finergrain.c.

In this code our objective was to obtain a finer-grain parallelism, to do so we will create more and smaller tasks. To achieve this we created more explicit tasks through the program, splitting the computation phase into smaller tasks, which can be seen in the code. Doing this caused some variables to create conflicts so we had to protect them. To protect the variable equal we split it in two and used a taskwait to ensure everything is checked before continuing. Then we have the same protections as in the tiled strategy, an atomic for histogram and a critical for X_COL_11 on the computation phase.

<u>Modelfactor analysis</u>

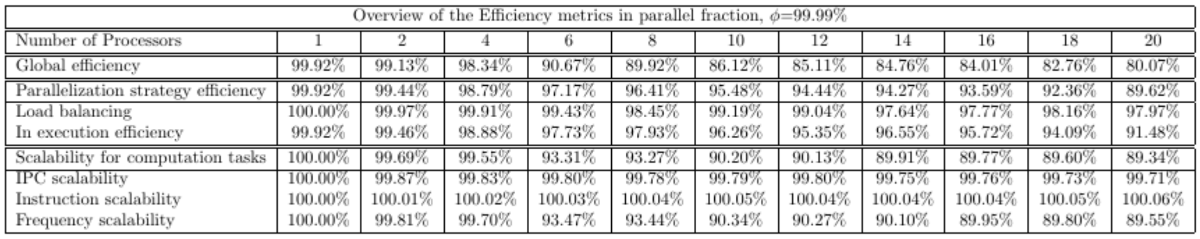


Overview of whole program execution metrics											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	3.10	1.56	0.79	0.57	0.43	0.36	0.30	0.26	0.23	0.21	0.19
Speedup	1.00	1.98	3.94	5.44	7.19	8.61	10.21	11.85	13.44	14.88	16.00
Efficiency	1.00	0.99	0.98	0.91	0.90	0.86	0.85	0.85	0.84	0.83	0.80

Table 1: Analysis done on Mon Dec 2 07:59:37 PM CET 2024, par2110

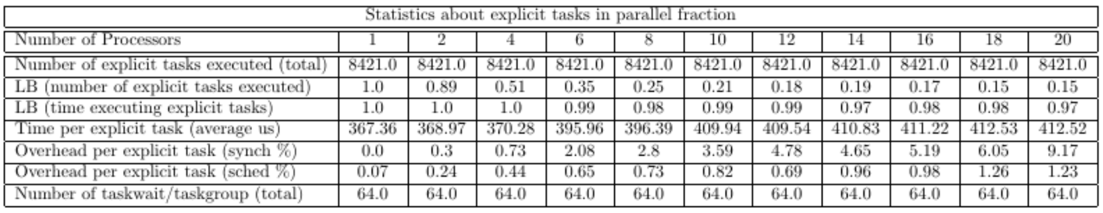
Modelfactor Table 1

By analyzing this table, we can easily see that it's better than the tile version. We obtain a higher speedup and speed while the efficiency does not drop sharply. This looks like a better option in terms of parallelism.



Overview of the Efficiency metrics in parallel fraction, ϕ=99.99%											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	99.92%	99.13%	98.34%	90.67%	89.92%	86.12%	85.11%	84.76%	84.01%	82.76%	80.07%
Parallelization strategy efficiency	99.92%	99.44%	98.79%	97.17%	96.41%	95.48%	94.44%	94.27%	93.59%	92.36%	89.62%
Load balancing	100.00%	99.97%	99.91%	99.43%	98.45%	99.19%	99.04%	97.64%	97.77%	98.16%	97.97%
In execution efficiency	99.92%	99.46%	98.88%	97.73%	97.93%	96.26%	95.35%	96.55%	95.72%	94.09%	91.48%
Scalability for computation tasks	100.00%	99.69%	99.55%	93.31%	93.27%	90.20%	90.13%	89.91%	89.77%	89.60%	89.34%
IPC scalability	100.00%	99.87%	99.83%	99.80%	99.78%	99.79%	99.80%	99.75%	99.76%	99.73%	99.71%
Instruction scalability	100.00%	100.01%	100.02%	100.03%	100.04%	100.05%	100.04%	100.04%	100.04%	100.05%	100.06%
Frequency scalability	100.00%	99.81%	99.70%	93.47%	93.44%	90.34%	90.27%	90.10%	89.95%	89.80%	89.55%

Table 2: Analysis done on Mon Dec 2 07:59:37 PM CET 2024, par2110



Statistics about explicit tasks in parallel fraction											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Number of explicit tasks executed (total)	8421.0	8421.0	8421.0	8421.0	8421.0	8421.0	8421.0	8421.0	8421.0	8421.0	8421.0
LB (number of explicit tasks executed)	1.0	0.89	0.51	0.35	0.25	0.21	0.18	0.19	0.17	0.15	0.15
LB (time executing explicit tasks)	1.0	1.0	1.0	0.99	0.98	0.99	0.99	0.97	0.98	0.98	0.97
Time per explicit task (average us)	367.36	368.97	370.28	395.96	396.39	409.94	409.54	410.83	411.22	412.53	412.52
Overhead per explicit task (synch %)	0.0	0.3	0.73	2.08	2.8	3.59	4.78	4.65	5.19	6.05	9.17
Overhead per explicit task (sched %)	0.07	0.24	0.44	0.65	0.73	0.82	0.69	0.96	0.98	1.26	1.23
Number of taskwait/taskgroup (total)	64.0	64.0	64.0	64.0	64.0	64.0	64.0	64.0	64.0	64.0	64.0

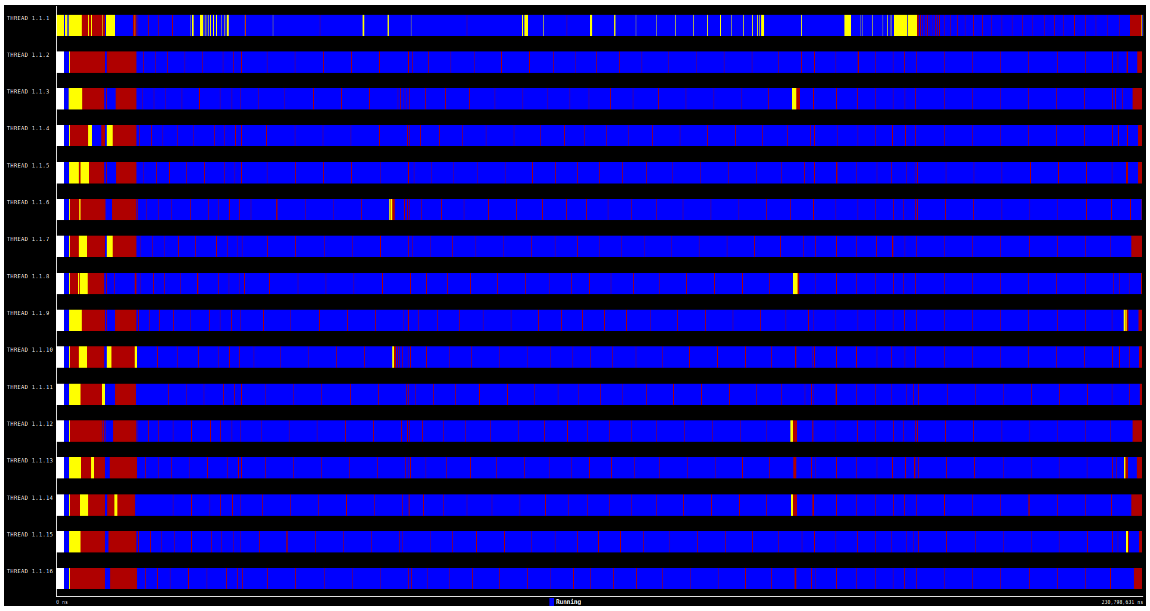
Table 3: Analysis done on Mon Dec 2 07:59:37 PM CET 2024, par2110

Modelfactor Table 2&3

In the second table, we can analyze the load balancing. As we can see, tasks are better balanced, almost 100% in all cases. This is what makes this version much better than the previous one since we are using parallelism more, thus having better efficiency.
The third table shows the reasons why this version is better. Now we have more tasks, which creates a little bit more task creation overhead. However, the smaller tasks (we can see on the table that the time per explicit task is smaller) give us less synchronization time.

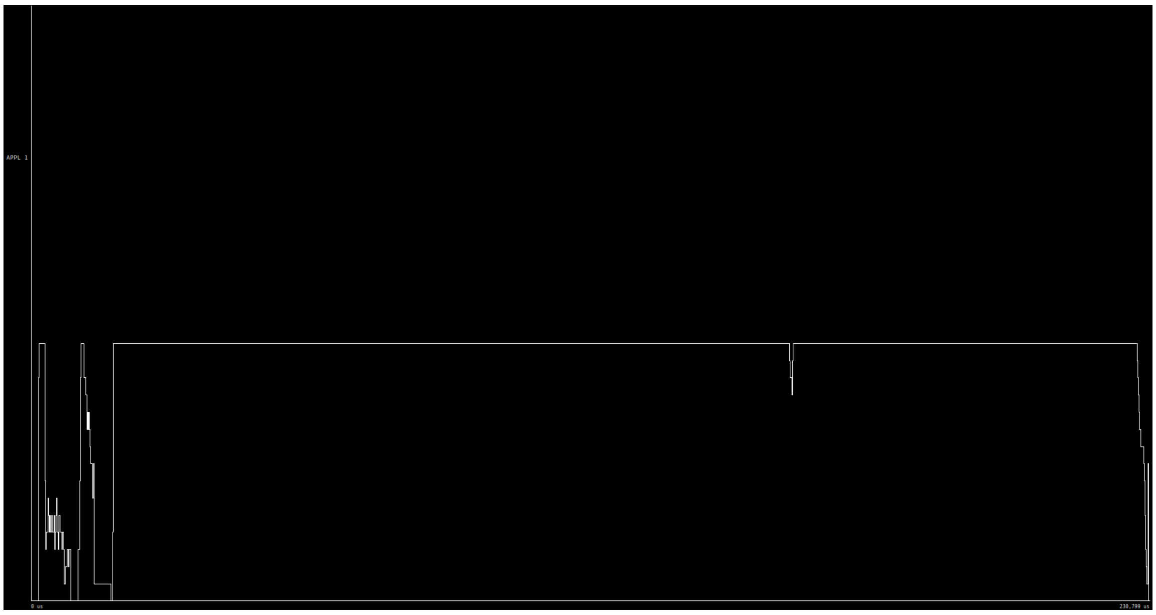
In conclusion, analyzing the tables we can say that we have a very good task balance, thanks to the creation of more but smaller tasks. We have a very good efficiency for this reason. The creation of more tasks creates a little bit more task creation overhead, but it's nothing compared to the improvement we get by making finer grain tasks.

<u>Paraver analysis</u>



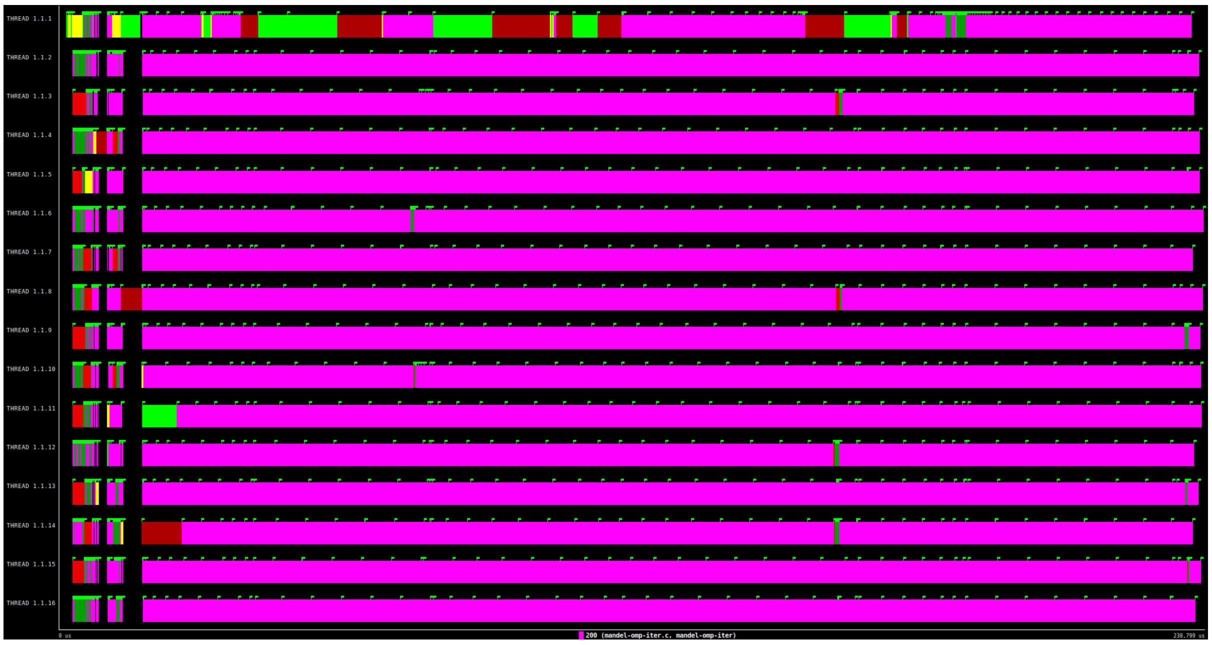
Paraver execution 16 threads → 280.889.194

The execution graph lets us see how through the execution of the program all processors spend most of the time working.



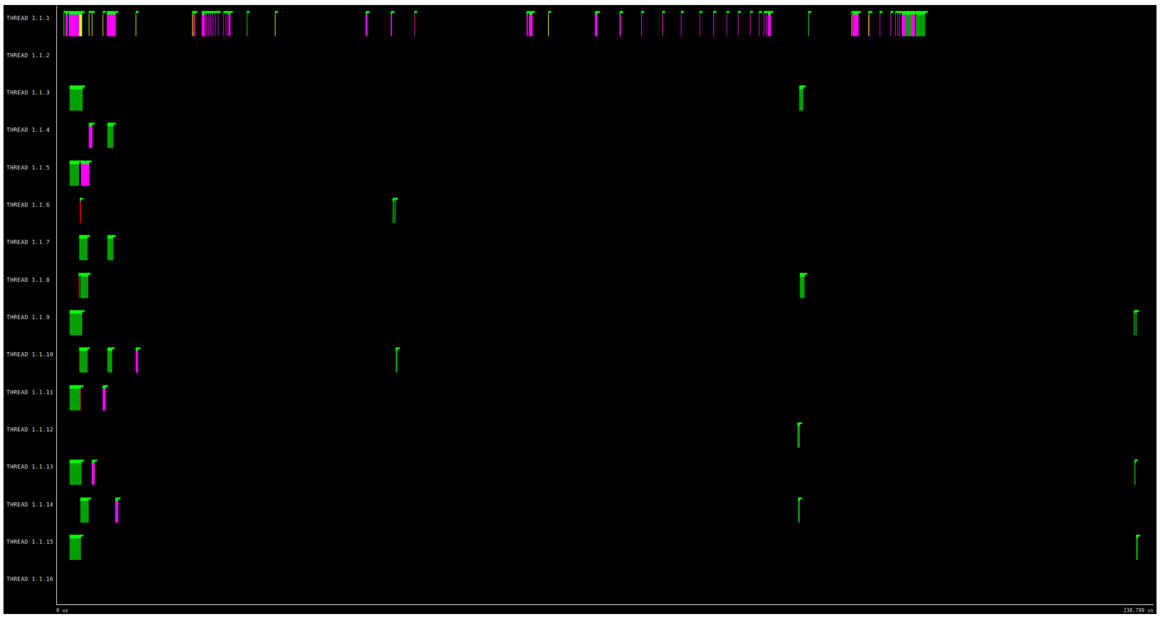
Paraver hint instantaneous parallelism

The instantaneous parallelism confirms what we analyzed previously. At the start the parallelism is irregular since that's when some tasks are created and wait for others to end, but after that parallelism stays constant, giving us a great result in efficiency.



Paraver hint explicit task function execution

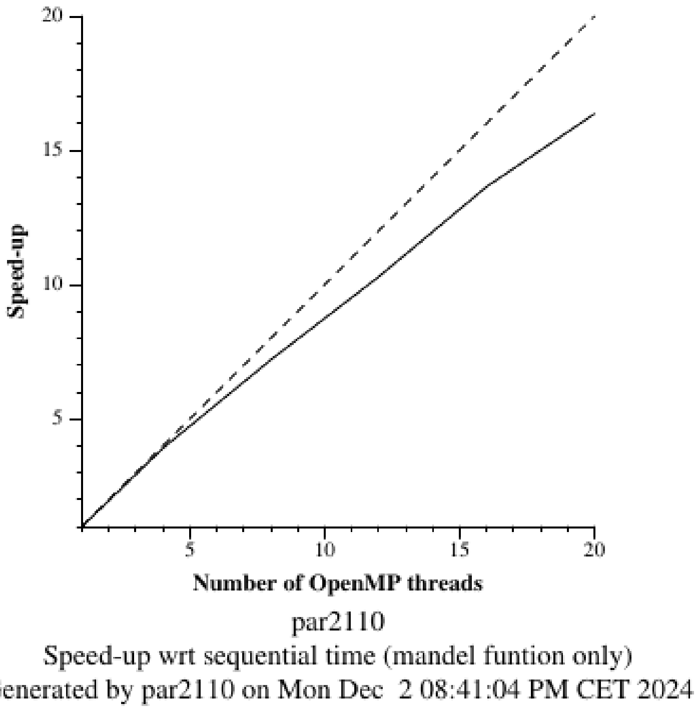
Again in the execution hint, we can see the load balancing is very good and all processors spend most of the time working.



Paraver hint explicit task function creation

On this graph we can see when the tasks are created and what processors create them, this is what causes the task creation overhead, and this overhead is small. Although on the tiled version, we did not have this overhead the finer grain strategy is still much better.

Strong scalability



Strong scalability graph

The graph of strong scalability shows us how good the speedup is when increasing the number of processors, thanks to the high efficiency we obtain a speedup very close to the ideal. However, we can see that as we use more processors the slope of the graph decreases slowly. Even with that, we can say that it’s a good strategy. We obtain great efficiency and speedup.

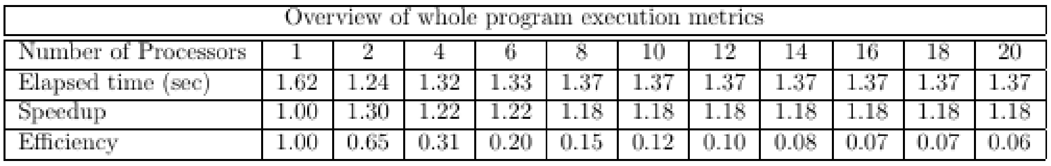
Recursive task decomposition

Recursive: Leaf

Code

The file name is mandel-omp-rec-leaf.c.

Modelfactor analysis

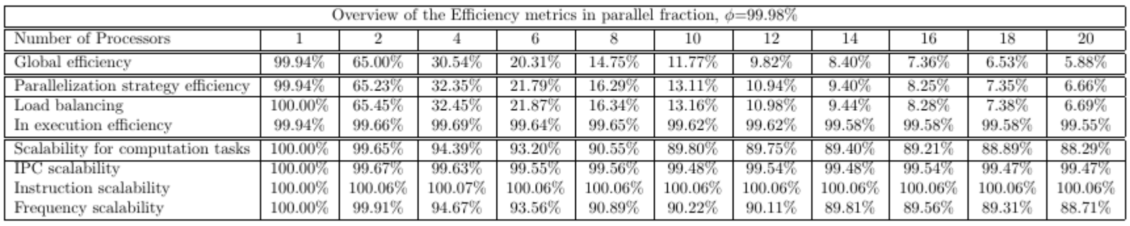


Overview of whole program execution metrics											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	1.62	1.24	1.32	1.33	1.37	1.37	1.37	1.37	1.37	1.37	1.37
Speedup	1.00	1.30	1.22	1.22	1.18	1.18	1.18	1.18	1.18	1.18	1.18
Efficiency	1.00	0.65	0.31	0.20	0.15	0.12	0.10	0.08	0.07	0.07	0.06

Table 1: Analysis done on Fri Nov 15 10:40:47 AM CET 2024, par2110

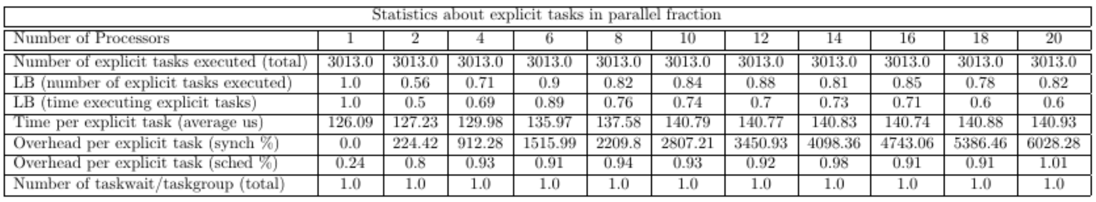
Modelfactor Table 1

Analyzing this first table, we can appreciate a great decrease in efficiency since the beginning, resulting in a very poor value. We can also appreciate that the execution time's reduction is very little in comparison with the obtained value for 1 thread, as we can see with the speedup.



Overview of the Efficiency metrics in parallel fraction, ϕ=99.98%											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	99.94%	65.00%	30.54%	20.31%	14.75%	11.77%	9.82%	8.40%	7.36%	6.53%	5.88%
Parallelization strategy efficiency	99.94%	65.23%	32.35%	21.79%	16.29%	13.11%	10.94%	9.40%	8.25%	7.35%	6.66%
Load balancing	100.00%	65.45%	32.45%	21.87%	16.34%	13.16%	10.98%	9.44%	8.28%	7.38%	6.69%
In execution efficiency	99.94%	99.66%	99.69%	99.64%	99.65%	99.62%	99.62%	99.58%	99.58%	99.58%	99.55%
Scalability for computation tasks	100.00%	99.65%	94.39%	93.20%	90.55%	89.80%	89.75%	89.40%	89.21%	88.89%	88.29%
IPC scalability	100.00%	99.67%	99.63%	99.55%	99.56%	99.48%	99.54%	99.48%	99.54%	99.47%	99.47%
Instruction scalability	100.00%	100.06%	100.07%	100.06%	100.06%	100.06%	100.06%	100.06%	100.06%	100.06%	100.06%
Frequency scalability	100.00%	99.91%	94.67%	93.56%	90.89%	90.22%	90.11%	89.81%	89.56%	89.31%	88.71%

Table 2: Analysis done on Fri Nov 15 10:40:47 AM CET 2024, par2110



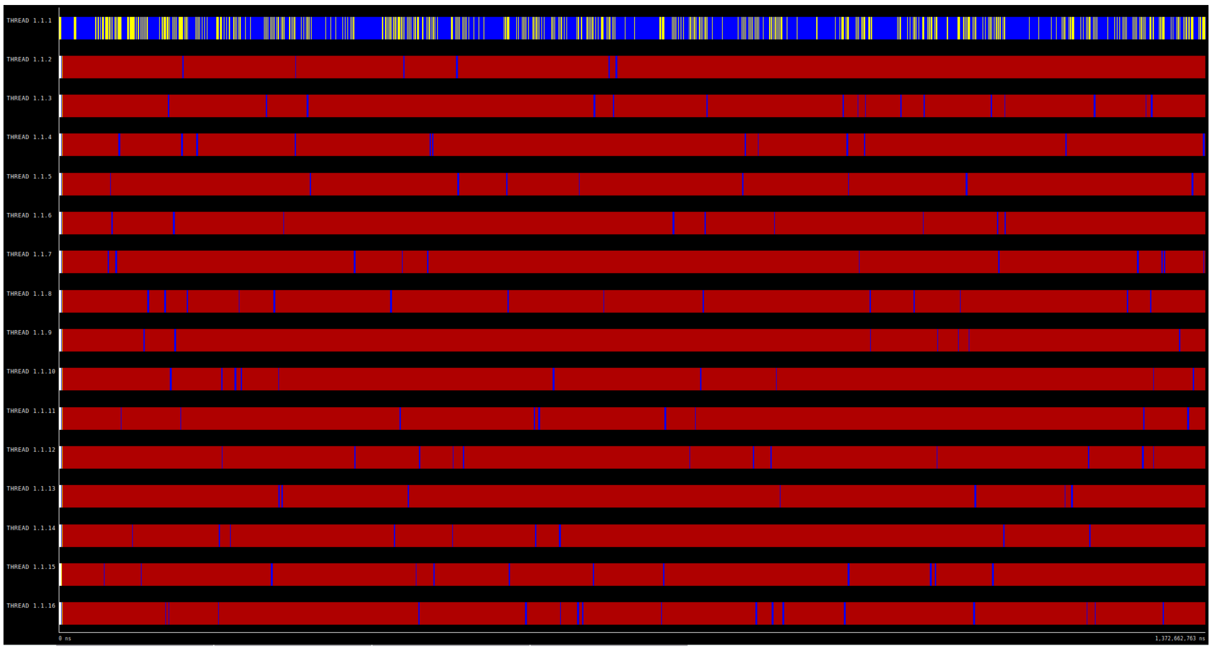
Statistics about explicit tasks in parallel fraction											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Number of explicit tasks executed (total)	3013.0	3013.0	3013.0	3013.0	3013.0	3013.0	3013.0	3013.0	3013.0	3013.0	3013.0
LB (number of explicit tasks executed)	1.0	0.56	0.71	0.9	0.82	0.84	0.88	0.81	0.85	0.78	0.82
LB (time executing explicit tasks)	1.0	0.5	0.69	0.89	0.76	0.74	0.7	0.73	0.71	0.6	0.6
Time per explicit task (average us)	126.09	127.23	129.98	135.97	137.58	140.79	140.77	140.83	140.74	140.88	140.93
Overhead per explicit task (synch %)	0.0	224.42	912.28	1515.99	2209.8	2807.21	3450.93	4098.36	4743.06	5386.46	6028.28
Overhead per explicit task (sched %)	0.24	0.8	0.93	0.91	0.94	0.93	0.92	0.98	0.91	0.91	1.01
Number of taskwait/taskgroup (total)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table 3: Analysis done on Fri Nov 15 10:40:47 AM CET 2024, par2110

Modelfactor Table 2&3

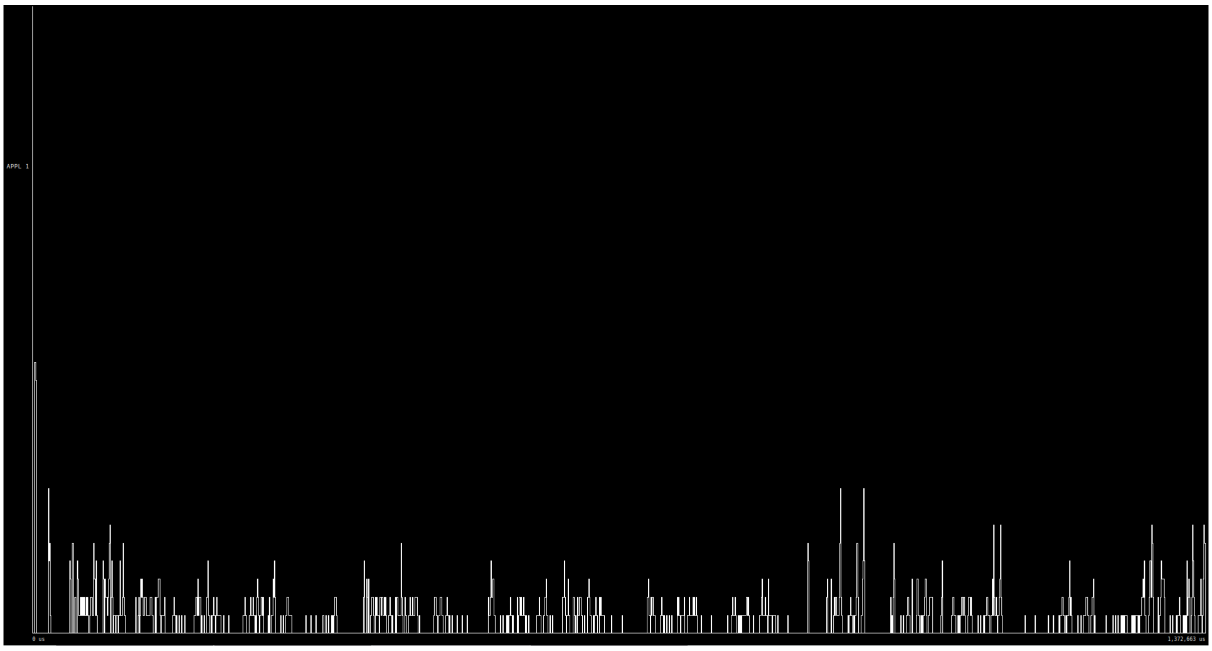
With these two tables, we can appreciate the most important problem for this implementation. There's one task that does all the work, and the others spend most of the time waiting for this bigger task to finish. This aspect can be seen when we analyze the values of load balancing (in the table 2), which decreases to 6.69% (a very poor value), and the values of overhead per explicit task, which is very big, arriving at 6028.28%. This indicates that the threads have very little work to do for all their time, and the majority of it consists of synchronization.

Paraver analysis



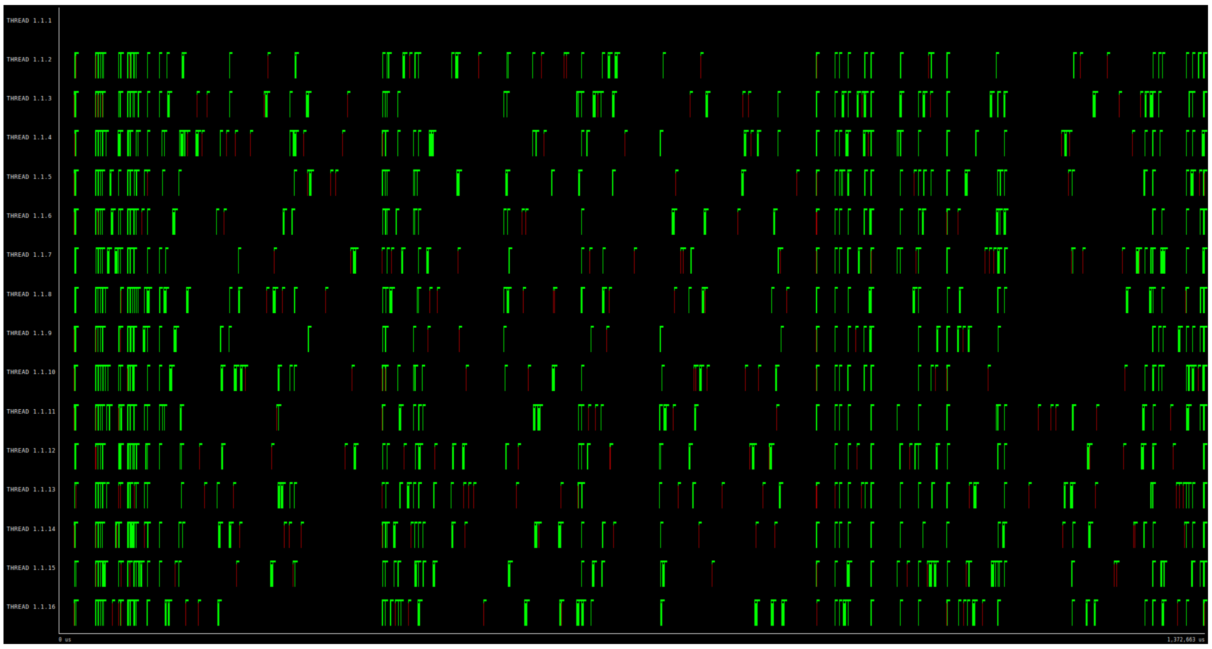
Paraver execution 16 threads → 1.372.662.763 ns

As we have commented before, this graphic shows us the same conclusion as the model factor tables. We can clearly identify that the majority of the tasks spend all tier time synchronizing themselves (color red), while there's one task (the top one) that does all the work.

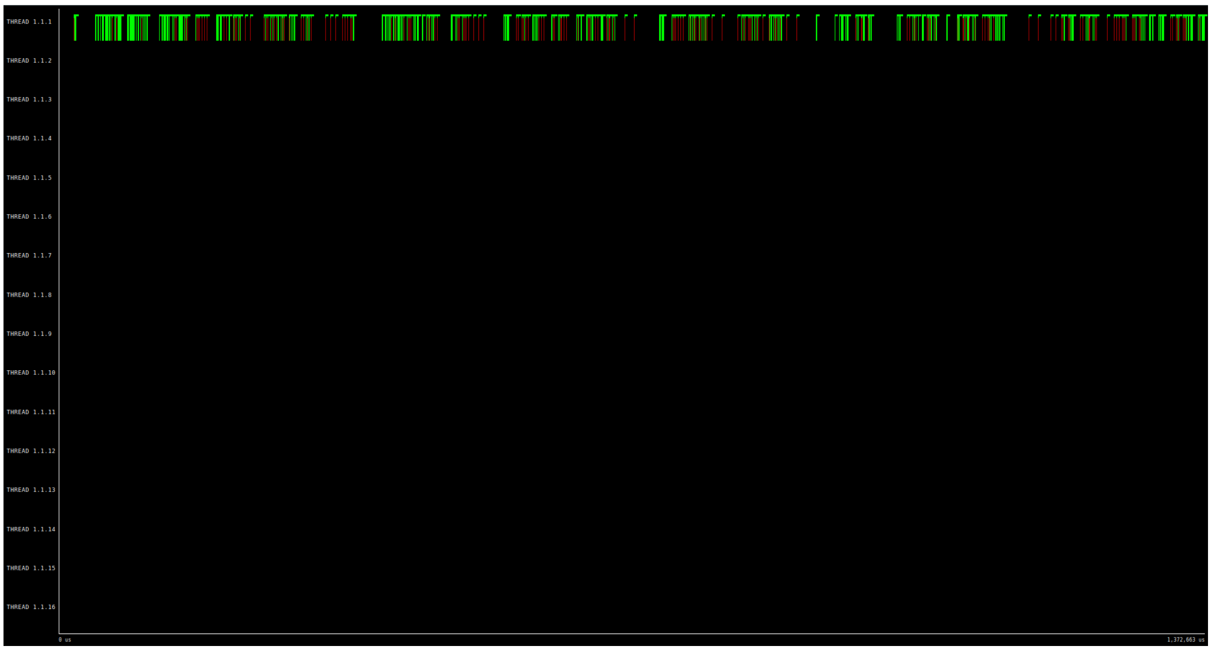


Paraver hint instantaneous parallelism

This graphic shows us that the parallelism is not continuous, and has a lot of ups and downs, which agrees with leaf implementation's unload balancing problem.



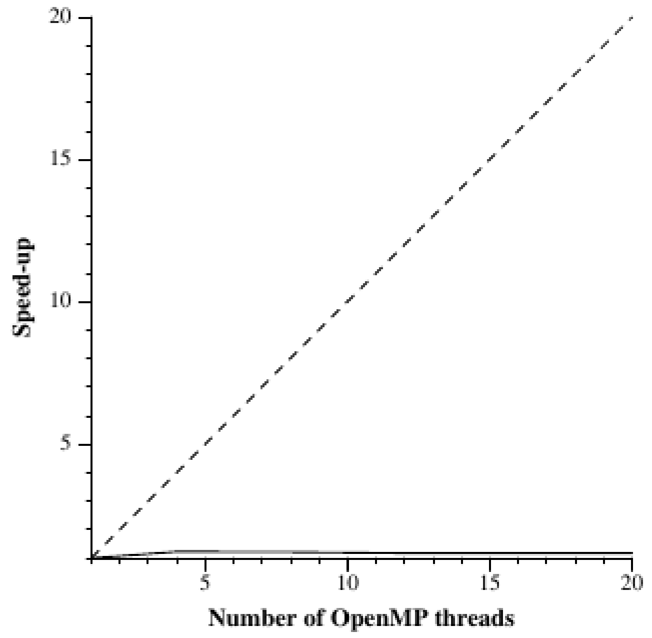
Paraver hint explicit task function execution



Paraver hint explicit task function creation

With both of these graphs, we clearly appreciate that the tasks are executing the code very few times, and in a very interspersed way (first image), which confirms the analysis we've been doing with the previous images. We can also see that there's only one task which is in charge of creating all the other ones (second image), causing a lot of work for itself but not for the ones it's creating.

Strong scalability



par2110
Speed-up wrt sequential time (mandel funtion only)
Generated by par2110 on Fri Nov 15 11:01:31 AM CET 2024

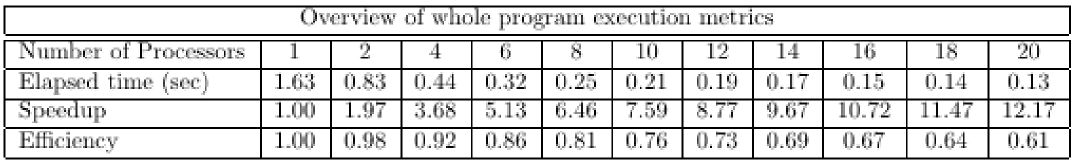
Strong scalability graph

The strong scalability test shows us the same as the first model factor table. The speed-up is very poor and is not worth increasing the number of threads that execute this implementation, as there will not be any major upgrade.

Recursive: Tree Without Cutoff

<u>Code</u>
The file name is mandel-omp-rec-tree-nocutoff.c

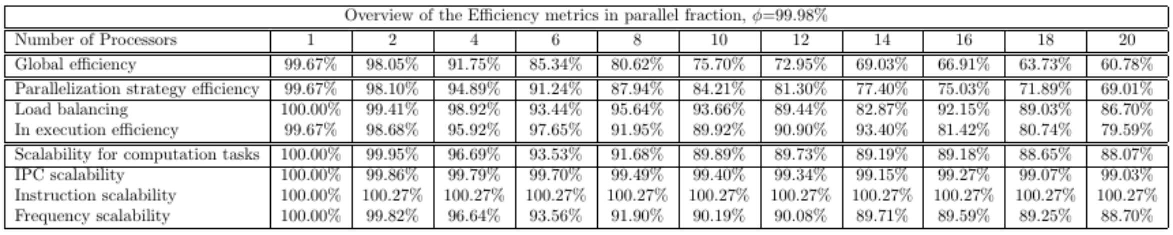
<u>Modelfactor analysis</u>



Overview of whole program execution metrics											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	1.63	0.83	0.44	0.32	0.25	0.21	0.19	0.17	0.15	0.14	0.13
Speedup	1.00	1.97	3.68	5.13	6.46	7.59	8.77	9.67	10.72	11.47	12.17
Efficiency	1.00	0.98	0.92	0.86	0.81	0.76	0.73	0.69	0.67	0.64	0.61

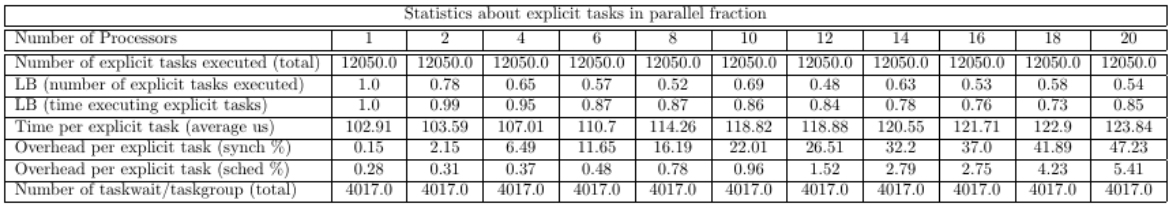
Table 1: Analysis done on Fri Nov 15 11:43:30 AM CET 2024, par2110

Modelfactor Table 1



Overview of the Efficiency metrics in parallel fraction, ϕ=99.98%											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	99.67%	98.05%	91.75%	85.34%	80.62%	75.70%	72.95%	69.03%	66.91%	63.73%	60.78%
Parallelization strategy efficiency	99.67%	98.10%	94.89%	91.24%	87.94%	84.21%	81.30%	77.40%	75.03%	71.89%	69.01%
Load balancing	100.00%	99.41%	98.92%	93.44%	95.64%	93.66%	89.44%	82.87%	92.15%	89.03%	86.70%
In execution efficiency	99.67%	98.68%	95.92%	97.65%	91.95%	89.92%	90.90%	93.40%	81.42%	80.74%	79.59%
Scalability for computation tasks	100.00%	99.95%	96.69%	93.53%	91.68%	89.89%	89.73%	89.19%	89.18%	88.65%	88.07%
IPC scalability	100.00%	99.86%	99.79%	99.70%	99.49%	99.40%	99.34%	99.15%	99.27%	99.07%	99.03%
Instruction scalability	100.00%	100.27%	100.27%	100.27%	100.27%	100.27%	100.27%	100.27%	100.27%	100.27%	100.27%
Frequency scalability	100.00%	99.82%	96.64%	93.56%	91.90%	90.19%	90.08%	89.71%	89.59%	89.25%	88.70%

Table 2: Analysis done on Fri Nov 15 11:43:30 AM CET 2024, par2110



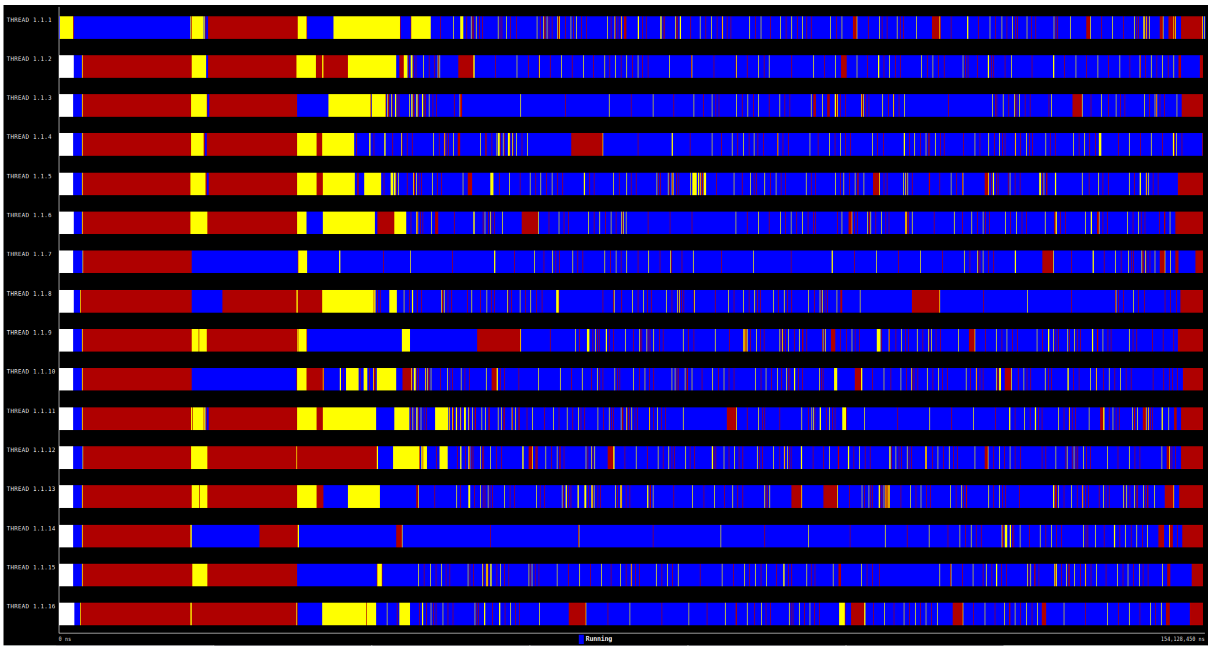
Statistics about explicit tasks in parallel fraction											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Number of explicit tasks executed (total)	12050.0	12050.0	12050.0	12050.0	12050.0	12050.0	12050.0	12050.0	12050.0	12050.0	12050.0
LB (number of explicit tasks executed)	1.0	0.78	0.65	0.57	0.52	0.69	0.48	0.63	0.53	0.58	0.54
LB (time executing explicit tasks)	1.0	0.99	0.95	0.87	0.87	0.86	0.84	0.78	0.76	0.73	0.85
Time per explicit task (average us)	102.91	103.59	107.01	110.7	114.26	118.82	118.88	120.55	121.71	122.9	123.84
Overhead per explicit task (synch %)	0.15	2.15	6.49	11.65	16.19	22.01	26.51	32.2	37.0	41.89	47.23
Overhead per explicit task (sched %)	0.28	0.31	0.37	0.48	0.78	0.96	1.52	2.79	2.75	4.23	5.41
Number of taskwait/taskgroup (total)	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0

Table 3: Analysis done on Fri Nov 15 11:43:30 AM CET 2024, par2110

Modelfactor Tables 2&3

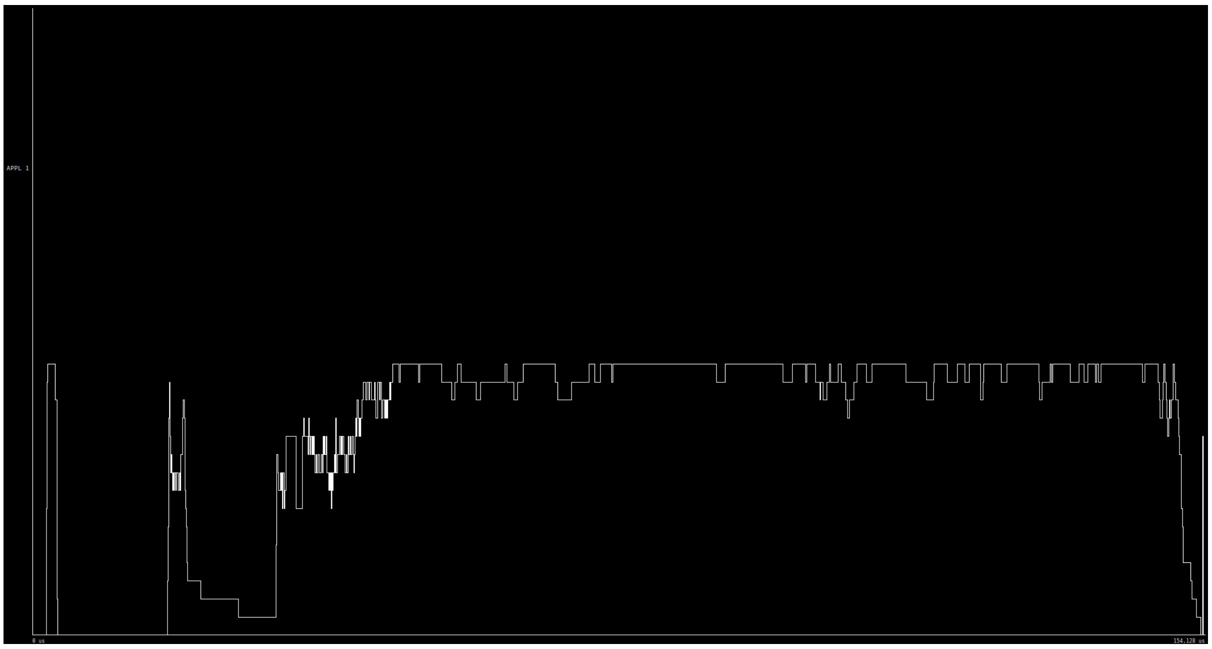
With these two pictures, we can appreciate that this tree implementation is better than the leaf one in all values. The principal problem is that many tasks are created (12050), but the synchronization % and load balancing are by far better in this implementation than the leaf strategy.

Paraver analysis



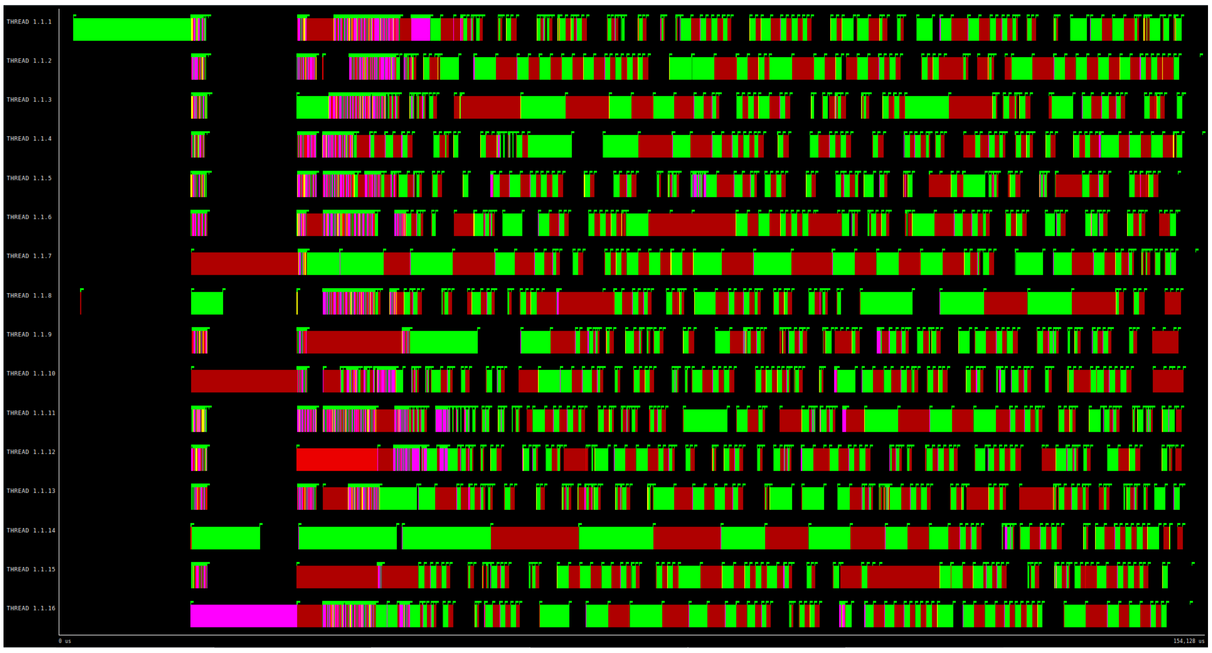
Paraver execution 16 threads → 154.128.450 ns

We can appreciate that the execution is distributed approximately so that every task makes the same amount of work and that the synchronization time is very little.

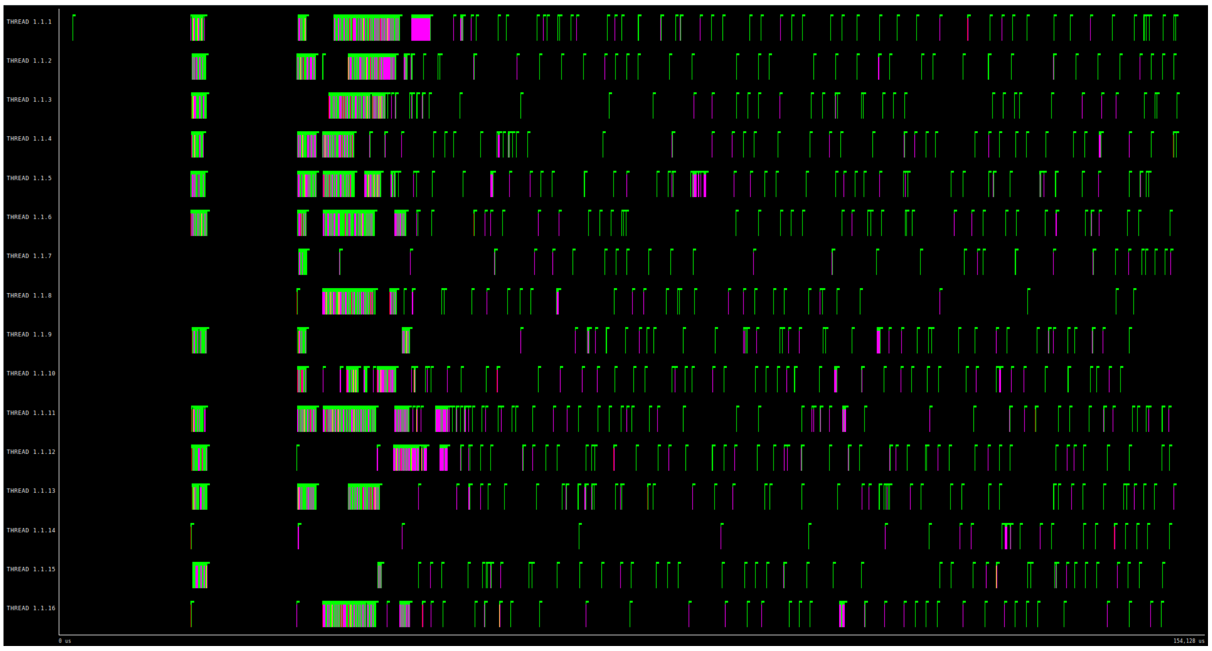


Paraver hint instantaneous parallelism

With this image, we see that at the beginning the parallelism is not continuous. Then, in the middle we obtain something approximately continuous, but at the end, it falls off again.



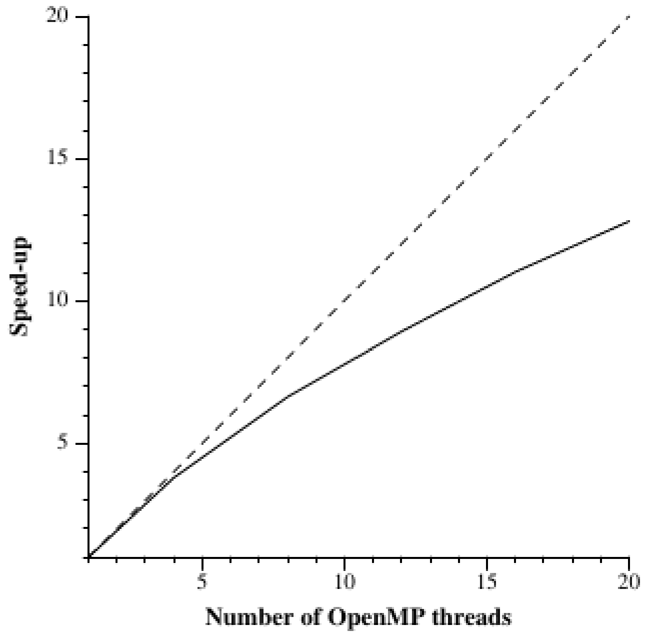
Paraver hint explicit task function execution



Paraver hint explicit task function creation

This time, we can see that all tasks create other tasks (as Tree implementation is different from Leaf), and the execution time is well-balanced between tasks. Some tasks indeed do more work than others, but in general, it's well-balanced.

Strong scalability



par2110
Speed-up wrt sequential time (mandel funtion only)
Generated by par2110 on Fri Nov 15 11:45:08 AM CET 2024

Strong scalability graph

With the strong scalability graph, we see that the line starts being perfect, but in the end, it falls a bit. Although this fall, this implementation is very efficient.

Recursive: Tree With Cutoff

Code
The file name is mandel-omp-rec-tree-cutoff.c

Modelfactor analysis and cutoff selection

Cuttof 5

Overview of whole program execution metrics											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	1.62	0.82	0.45	0.32	0.25	0.21	0.19	0.17	0.16	0.14	0.13
Speedup	1.00	1.97	3.65	5.11	6.44	7.56	8.75	9.59	10.39	11.32	12.15
Efficiency	1.00	0.99	0.91	0.85	0.81	0.76	0.73	0.69	0.65	0.63	0.61

Table 1: Analysis done on Fri Nov 29 10:52:25 AM CET 2024, par2110

Overview of the Efficiency metrics in parallel fraction, ϕ=99.98%											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	99.72%	98.39%	91.05%	84.95%	80.39%	75.51%	72.80%	68.48%	64.90%	62.86%	60.71%
Parallelization strategy efficiency	99.72%	98.37%	94.11%	90.96%	87.78%	84.25%	81.63%	77.15%	73.63%	71.96%	69.92%
Load balancing	100.00%	99.55%	98.96%	95.99%	94.45%	90.06%	91.02%	89.79%	92.42%	86.00%	86.93%
In execution efficiency	99.72%	98.81%	95.10%	94.76%	92.94%	93.55%	89.67%	85.93%	79.67%	83.68%	80.43%
Scalability for computation tasks	100.00%	100.02%	96.74%	93.39%	91.58%	89.63%	89.19%	88.76%	88.14%	87.34%	86.83%
IPC scalability	100.00%	99.93%	99.86%	99.73%	99.61%	99.48%	99.32%	99.26%	99.12%	99.02%	99.06%
Instruction scalability	100.00%	100.04%	100.04%	100.04%	100.04%	100.04%	100.04%	100.04%	100.04%	100.04%	100.04%
Frequency scalability	100.00%	100.05%	96.84%	93.60%	91.90%	90.06%	89.77%	89.39%	88.89%	88.17%	87.62%

Table 2: Analysis done on Fri Nov 29 10:52:25 AM CET 2024, par2110

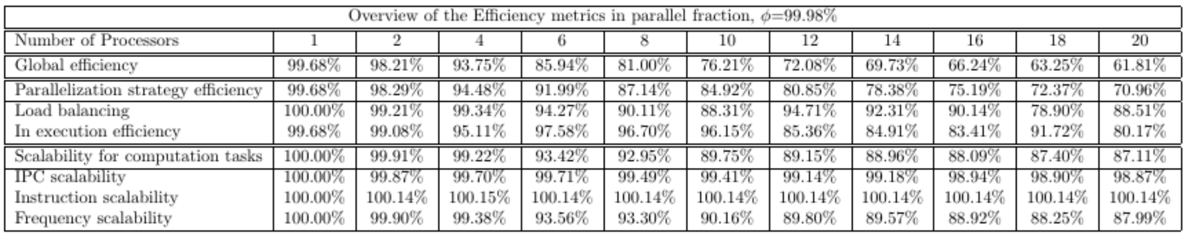
Statistics about explicit tasks in parallel fraction											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Number of explicit tasks executed (total)	9430.0	9430.0	9430.0	9430.0	9430.0	9430.0	9430.0	9430.0	9430.0	9430.0	9430.0
LB (number of explicit tasks executed)	1.0	0.78	0.87	0.5	0.62	0.54	0.54	0.52	0.43	0.46	0.55
LB (time executing explicit tasks)	1.0	1.0	0.99	0.9	0.86	0.86	0.84	0.76	0.83	0.74	0.72
Time per explicit task (average us)	131.44	131.47	135.84	140.56	144.6	148.86	149.27	151.14	149.11	151.91	150.1
Overhead per explicit task (synch %)	0.15	1.88	7.7	12.28	16.89	22.21	26.66	34.59	42.72	44.61	50.16
Overhead per explicit task (sched %)	0.22	0.25	0.3	0.41	0.52	0.98	1.34	1.8	2.55	3.07	3.87
Number of taskwait/taskgroup (total)	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0

Table 3: Analysis done on Fri Nov 29 10:52:25 AM CET 2024, par2110

Cuttof 6

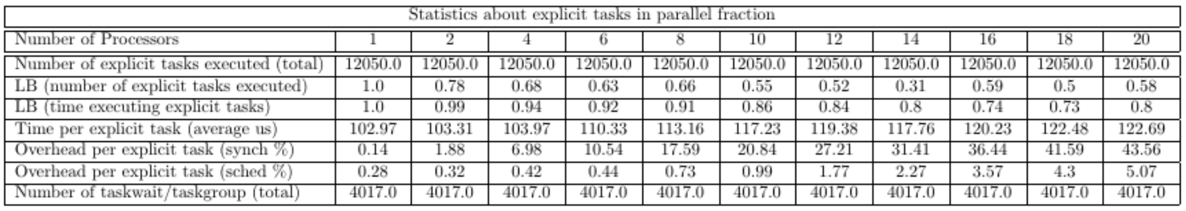
Overview of whole program execution metrics											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Elapsed time (sec)	1.63	0.83	0.43	0.31	0.25	0.21	0.19	0.17	0.15	0.14	0.13
Speedup	1.00	1.97	3.76	5.17	6.49	7.64	8.67	9.78	10.61	11.40	12.37
Efficiency	1.00	0.99	0.94	0.86	0.81	0.76	0.72	0.70	0.66	0.63	0.62

Table 1: Analysis done on Fri Nov 29 10:58:27 AM CET 2024, par2110



Overview of the Efficiency metrics in parallel fraction, ϕ=99.98%											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Global efficiency	99.68%	98.21%	93.75%	85.94%	81.00%	76.21%	72.08%	69.73%	66.24%	63.25%	61.81%
Parallelization strategy efficiency	99.68%	98.29%	94.48%	91.99%	87.14%	84.92%	80.85%	78.38%	75.19%	72.37%	70.96%
Load balancing	100.00%	99.21%	99.34%	94.27%	90.11%	88.31%	94.71%	92.31%	90.14%	78.90%	88.51%
In execution efficiency	99.68%	99.08%	95.11%	97.58%	96.70%	96.15%	85.36%	84.91%	83.41%	91.72%	80.17%
Scalability for computation tasks	100.00%	99.91%	99.22%	93.42%	92.95%	89.75%	89.15%	88.96%	88.09%	87.40%	87.11%
IPC scalability	100.00%	99.87%	99.70%	99.71%	99.49%	99.41%	99.14%	99.18%	98.94%	98.90%	98.87%
Instruction scalability	100.00%	100.14%	100.15%	100.14%	100.14%	100.14%	100.14%	100.14%	100.14%	100.14%	100.14%
Frequency scalability	100.00%	99.90%	99.38%	93.56%	93.30%	90.16%	89.80%	89.57%	88.92%	88.25%	87.99%

Table 2: Analysis done on Fri Nov 29 10:58:27 AM CET 2024, par2110

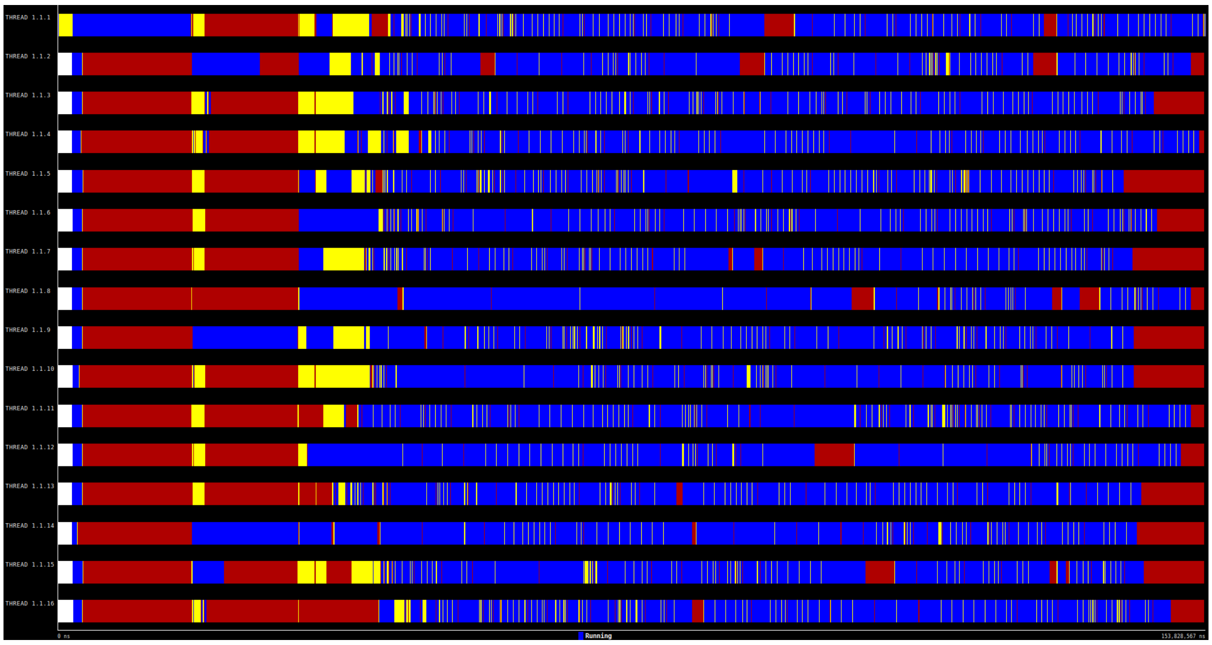


Statistics about explicit tasks in parallel fraction											
Number of Processors	1	2	4	6	8	10	12	14	16	18	20
Number of explicit tasks executed (total)	12050.0	12050.0	12050.0	12050.0	12050.0	12050.0	12050.0	12050.0	12050.0	12050.0	12050.0
LB (number of explicit tasks executed)	1.0	0.78	0.68	0.63	0.66	0.55	0.52	0.31	0.59	0.5	0.58
LB (time executing explicit tasks)	1.0	0.99	0.94	0.92	0.91	0.86	0.84	0.8	0.74	0.73	0.8
Time per explicit task (average us)	102.97	103.31	103.97	110.33	113.16	117.23	119.38	117.76	120.23	122.48	122.69
Overhead per explicit task (synch %)	0.14	1.88	6.98	10.54	17.59	20.84	27.21	31.41	36.44	41.59	43.56
Overhead per explicit task (sched %)	0.28	0.32	0.42	0.44	0.73	0.99	1.77	2.27	3.57	4.3	5.07
Number of taskwait/taskgroup (total)	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0	4017.0

Table 3: Analysis done on Fri Nov 29 10:58:27 AM CET 2024, par2110

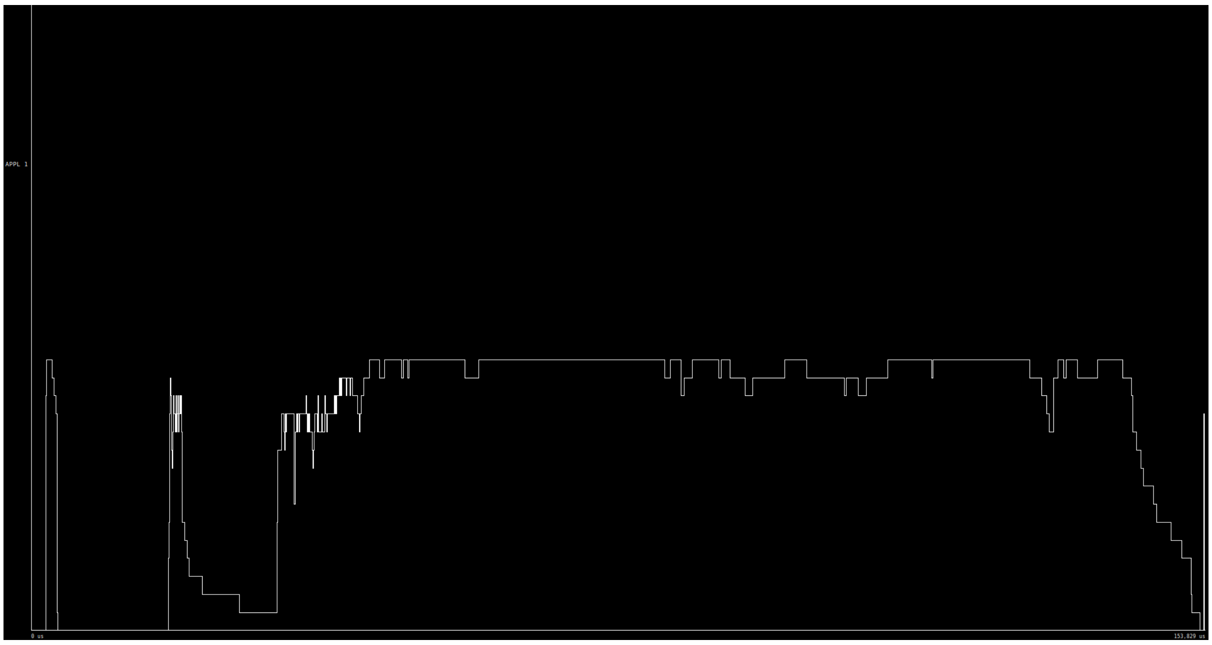
For the selection of the most suitable value for the cutoff, we decided between 5 and 6. As we can see in the model factor tables, the values are very similar, but the most important difference is that the level 6 cutoff creates the maximum amount of possible tasks (12050), and the level 5 cutoff is only 9430. This is not worth it as the execution time of both levels is the same, and load balancing and synchronization % are better, but not for so long. The efficiency also shows us that cutoff 6 has a 1% better efficiency than cutoff 5, which is not significant.

Paraver analysis



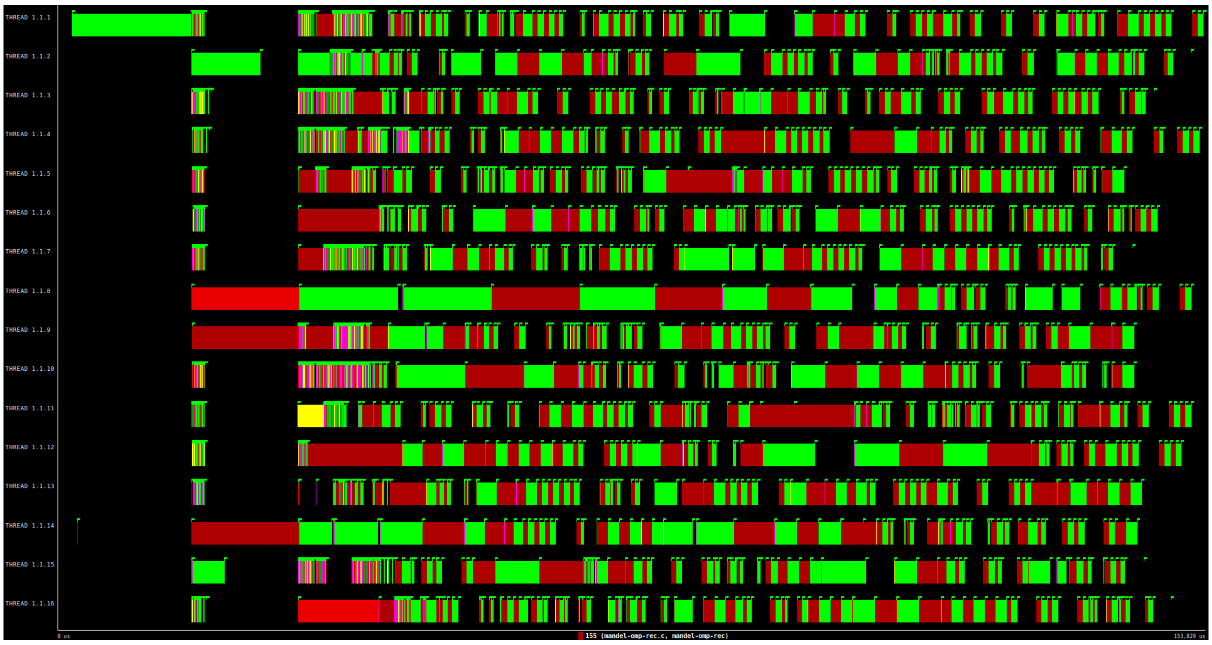
Paraver execution 16 threads → 153.828.567 ns

We can appreciate that with cutoff, this image is very similar to the one without cutoff, but we can appreciate that there is more synchronization (red), as the load balancing is a little worse than the version without cutoff. We can also appreciate (which will be corroborated in the next image) that at the beginning the parallelization is not very good as many tasks are in synchronization, in the middle they all work together (very good parallelization), but in the end, there is a lot of synchronization again.

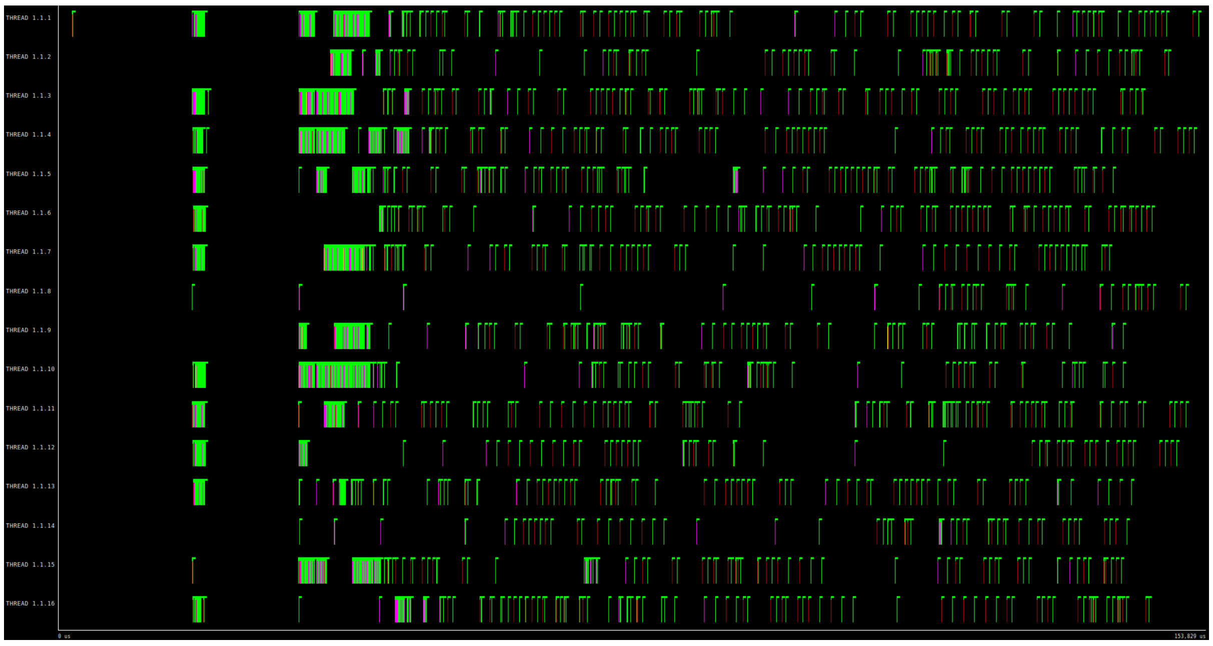


Paraver hint instantaneous parallelism

As we said, the same conclusion can be appreciated in this image as in the previous one. The parallelism is very good in the middle of the execution, but in the sides, it falls. This could happen because some tasks have to wait a lot for the other ones to finish, as the recursive tasks may not last the same in all cases.



Paraver hint explicit task function execution

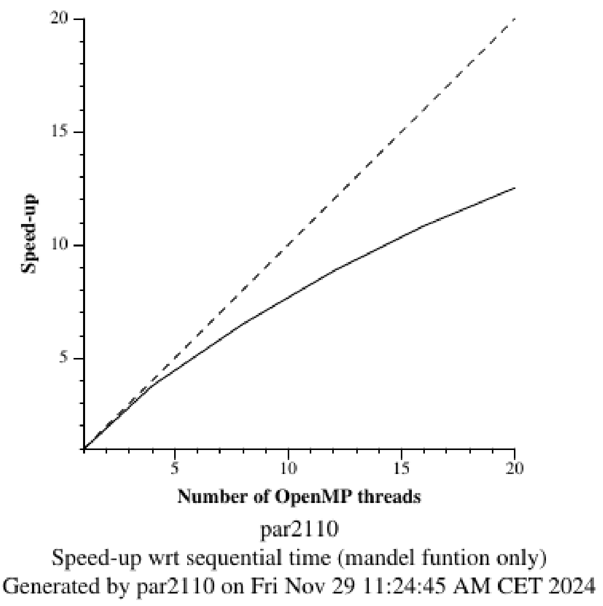


Paraver hint explicit task function creation

The same as before, the execution is well-balanced, but some tasks have to wait for others tasks to finish. Also, the creation from the tasks is distributed between all the tasks, not as the leaf strategy.

All the images are very similar to the tree implementation without cutoff.

Strong scalability



Strong scalability graph

We obtain a very good efficiency with the implementation of the cutoff, upgrading the original one (without cutoff), as we can appreciate that the line does not fall off as the original strong scalability test. This is the best implementation to parallelize the program.

When the cutoff level is above 6, there’s no upgrade in the efficiency and the important values because all possible tasks are created since level 6.

Final results

The next table represents the execution time of the program with n threads expressed in seconds. It gives us a general overview of the best strategies, however, it is important to analyze other aspects aside from the execution time, just as we did on the deliverable.

	Number of threads (n)					
Version	1	4	8	12	16	20
Iterative: Tile	3,10	0,91	0,71	0,71	0,71	0,71
Iterative: Finer grain	3,10	0,82	0,48	0,35	0,28	0,23
Recursive: Leaf	1,62	1,32	1,37	1,37	1,37	1,37
Recursive: Tree (no cutoff)	1,63	0,44	0,25	0,19	0,15	0,13
Recursive: Tree (cutoff = 5)	1,62	0,45	0,25	0,19	0,16	0,13