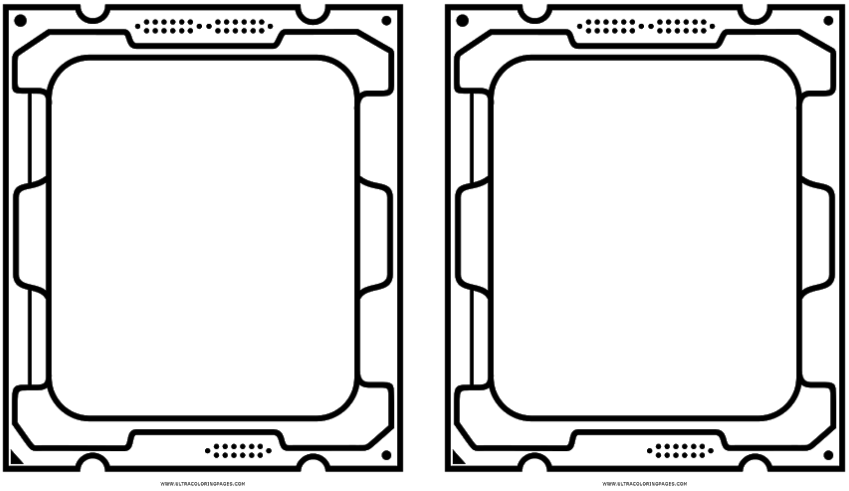
PAR Laboratory Assignment

Lab 1: Experimental setup and tools



Edgar Perez Blanco

Bartomeu Perelló Comas

Group: **PAR2301**

PAR - GEI FIB

Spring 2019-2020

Contents

Node architecture and memory

Through the use of “lstopo” command under the different systems, we have collected some information about their node’s processors. All these data have been summarized in the following table:

	boada-1 to 4	boada-5	boada-6 to 8
Number of sockets per node	2	2	2
Number of cores per socket	6	6	8
Number of threads per core	2	2	1
Maximum core frequency	2395 MHz	2600 MHz	1700 MHz
L1-I cache size (per-core)	32 KB	32 KB	32 KB
L1-D cache size (per-core)	32 KB	32 KB	32 KB
L2 cache size (per-core)	256 KB	256 KB	256 KB
Last-level cache size (per-socket)	12288 KB	15360 KB	20480KB
Main memory size (per socket)	12 GB	32 GB	16 GB
Main memory size (per node)	24 GB	64 GB	32 GB

* **Note**: *During our explanations we would refer to “boada-X’s CPUs” as “BX”. We would also assume that B1, B2, B3 and B4 are equal, so if we refer to the specs of B1 we are also referring to the other ones’ specs and analogously with B6, B7 and B8.*

As you can see in the previous figure, the similarities have been marked in green color. The CPUs are similar in some aspects, such as low level cache sizes and the number of cores and threads per node/core/socket between B1 and B5 systems. We also noticed that B6 has no SMT because it just has a single thread per core.

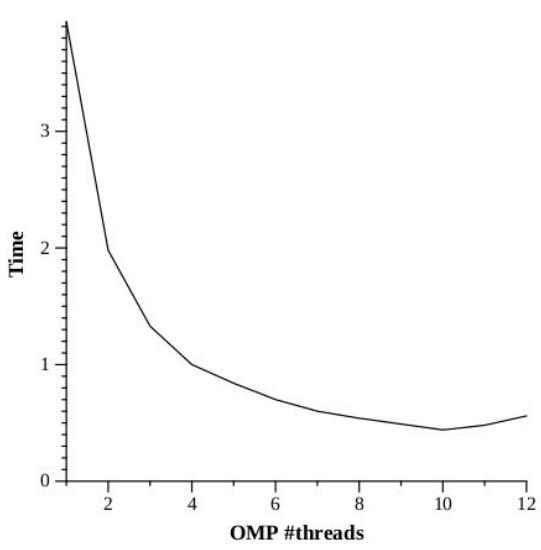
Despite B1 and B5 have the same core distribution, their last-level cache size is different. We can consider B5 the most advanced ones because in terms of frequency, B5 are the fastest and in terms of memory they have access to 32GB each socket. It is important to remark that we are just considering the specs and not the actual performance.

Strong vs. weak scalability

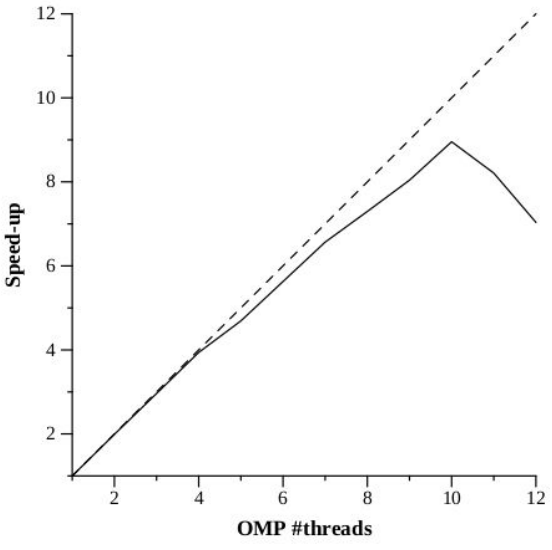
Strong scalability is used to see how the execution time evolves with the increasing number of threads used for each different execution of the exact same problem.

Otherwise weak scalability as the number of threads increases we increase the size of the problem so we can get a better result of the program we are executing, our intention is not to make the program run faster.

Here we can see an example of each type of scalability being the Strong one the first to be shown.

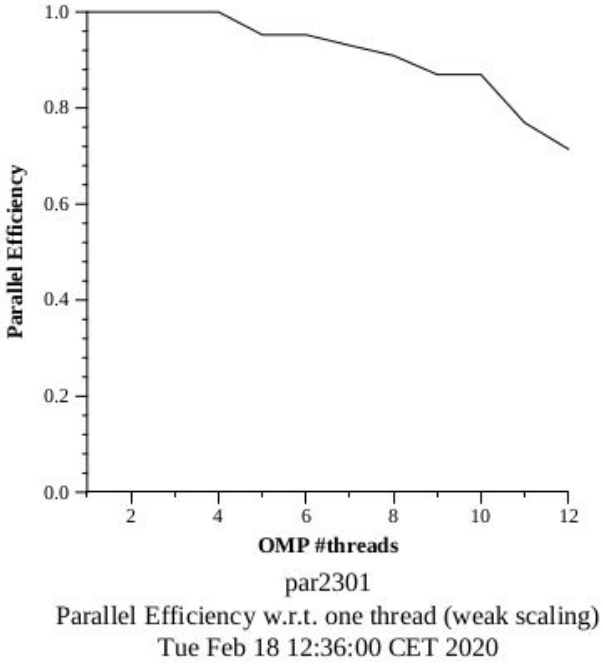


par2301
Min elapsed execution time
Tue Feb 18 12:11:47 CET 2020



par2301
Speed-up wrt sequential time
Tue Feb 18 12:11:47 CET 2020

As we can see in the two graphs, as the number of threads increases up to 10, the execution time got reduced and therefore, the speed-up increases. This means that it has an strong scaling up to 10 threads where, if we continue increasing the number of threats, due to the parallelism overhead, we will obtain worse results. We do not know exactly where the bottleneck is, but we suppose it must be caused by the synchronizations or the memory accesses the program has to do in order to run in parallel.



Finally here is the graph representing the evolution of the weak scaling. As we can see the parallel efficiency decreases as the number of threads increases as well as the size of the problem. Probably due to the same reason as before with the strong scaling, accessing memory has its own cost as well as synchronizing the threads.

Analysis of task decompositions for 3DFFT

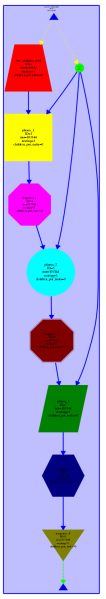
We have been asked to parallelize a program in different tasks. We modified the code according to the statement's instructions and we obtained 5 different versions. These correspond to different granularity levels of parallelization. The following table shows the estimated running time of the program over one and infinite* CPUs of each of version.

* When we talk about an infinite number of CPUs, we refer to one bigger than the number of parallel tasks the program has.

Version	T_1	T_∞	Parallelism
SEQ	639.780.000 ns	639.707.000 ns	≈ 1
v1	639.780.000 ns	639.707.000 ns	≈ 1
v2	639.933.000 ns	361.565.000 ns	1.77
v3	640.177.000 ns	361.802.000 ns	1.77
v4	640.188.000 ns	64.950.000 ns	9.86
v5	642.938.001 ns	41.740.000 ns	15.40

The Parallelism factor, the quotient between T_1 and T_∞, give us a theoretical idea of how much parallel is our program.

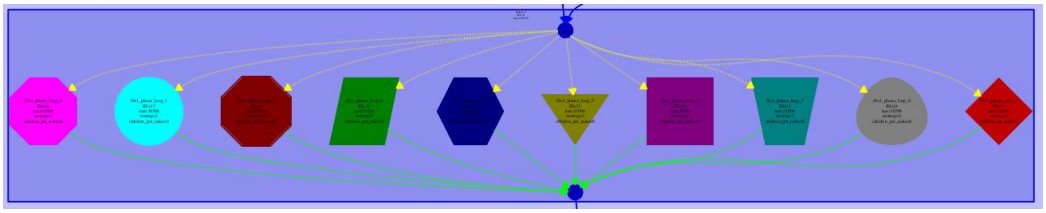
Figure 1



As shown in the figure 1, the task division just generates a program that runs almost sequential but divided into tasks. These tasks have data dependencies between them, so they are forced to run sequentially in order to run correctly.

To sum up, as we are just running sequentially a set of tasks, the performance improvement is null. We can not consider this version a parallelization strategy because it almost can not run in parallel; there is just a little parallel task, but it is negligible in comparison to the rest.

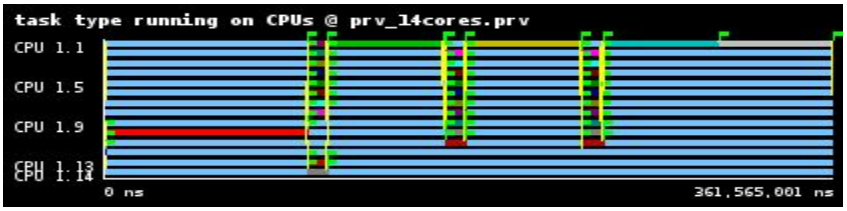
In the second version, we achieved some parallelism thanks to divide some tasks of the first version, into smaller ones; which ones correspond to the different iterations of a for-loop without data dependencies among them.



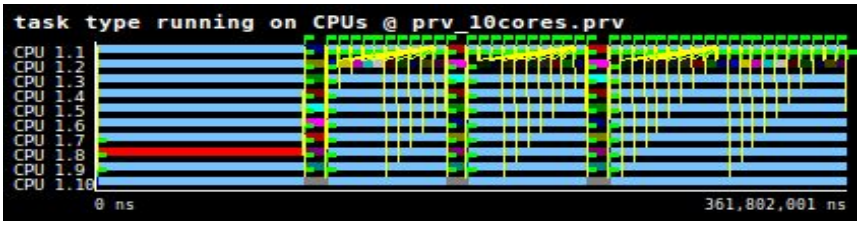
Example of one task divided by iterations

The task division on the third version consisted in, as in the second version, generating different tasks, inside a previously big one, for each iteration of a for-loop. The second and third versions have gave us similar results in terms of parallelism. Even though the third's granularity is finer than the second's one, due to data dependencies we did not improve noticeably the performance of the previous version.

As we can see in the following simulation graphs, even though the big green and yellow tasks are treated as sequential tasks in the second version, due to the dependencies inside them, they do not speed up in the third version



v2 simulation



v3 simulation

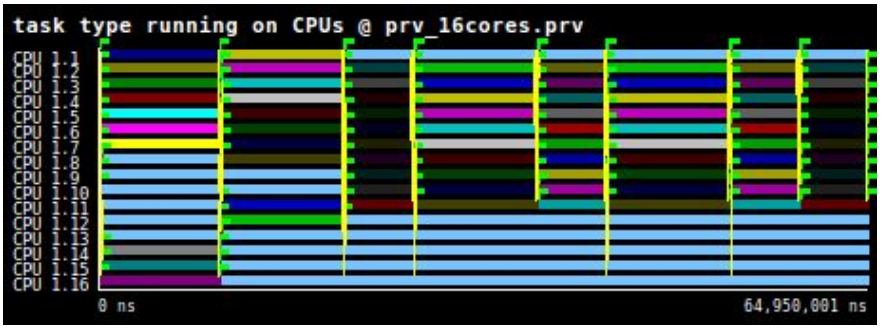
As a conclusion about the first, second and third versions, their performance do not differ a lot from the sequential version in terms of scalability. In general, this is due to a problem of load balancing and parallelism overheads.

Eventually, we arrived to the fourth version, which has a finer granularity that has given the program a significant boost in performance. Following up we have a chart showing the results of the simulation of the program.

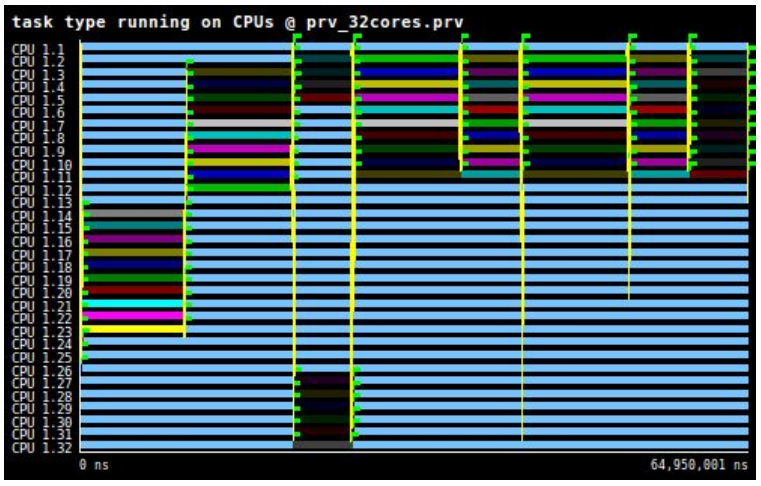
numCores	v4
1	640188000
2	320625000
4	191978000
8	128080000
16	64950000
32	64950000

As we can see in the chart above when using only one core the execution time is higher then the first version executed sequentially that's because the creation of all the new tasks takes some extra time. As we go deeper on the chart, we can appreciate a significant decrease on the execution time. On the 16 core simulation we find the limit of the strong scalability, as we can see on the execution time of 16 and 32 cores is the same.

Finally if we compare both graphs we can see that most of the cores don't really do nothing during the whole run of the program.



v4 simulation 16 CPU



v4 simulation 32 CPU

In this final version of the program we were told to experiment with even finer granularities, so we decided to move the task creator on the last loop on the following functions:

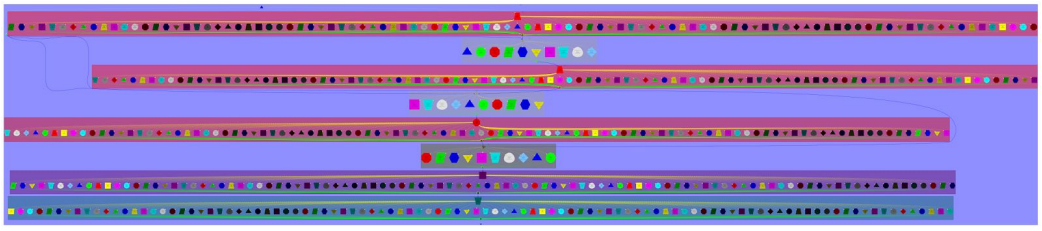
transpose_zx_planes, transpose_xy_planes and init_complex_grid, the next image shows an example of the functions, the three of them have a combination of 3 loops one inside another one.

The highlighted part of the image represents what it is considered a task.

```
void transpose_xy_planes(fftwf_complex  tmp_fftw[][N][N], fftwf_complex in_fftw[][N][N]) {
    int k,j,i;
    char msg[128];
    for (k=0; k<N; k++) {
     for (j=0; j<N; j++) {
        sprintf(msg,"tranpose_xy_planes_loop_%d_%d",k,j);
        tareador_start_task(msg);
       for (i=0; i<N; i++)
       {
         tmp_fftw[k][i][j][0] = in_fftw[k][j][i][0];
         tmp_fftw[k][i][j][1] = in_fftw[k][j][i][1];
       }
       tareador_end_task(msg);
     }
    }
}
```

Our decision of not going further and make every iteration of the loop we considered as a task was that if we were to create that many tasks the overhead produced to create every single one of them would be high enough that the execution time would have been worse than the actual version.

The image above shows the task dependency obtained after the execution of tareador.

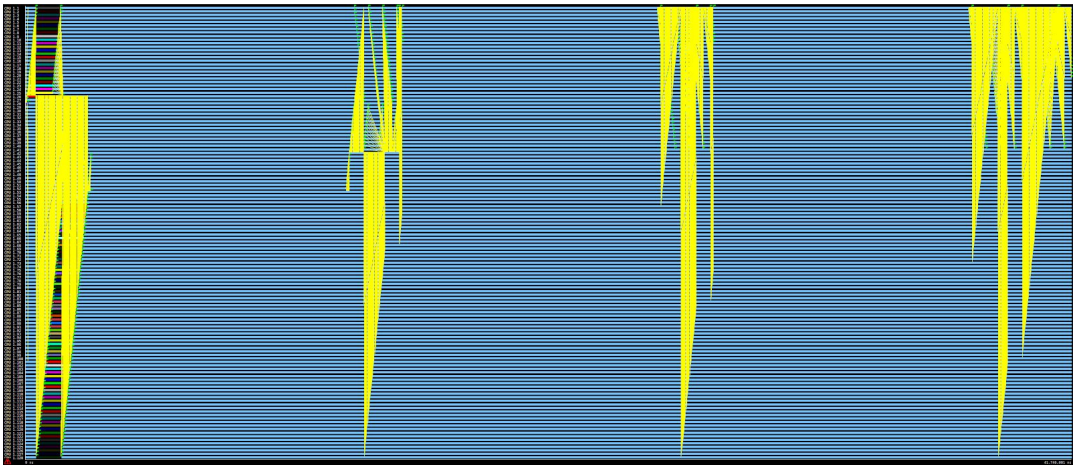


v5 task dependency graph

Finally on the last version of the program we can appreciate similar results in terms of strong scalability, as we increase the number of cores the execution time is reduced, but in this case we haven't reached the limit of the scalability at 16 core, in this case it goes further.

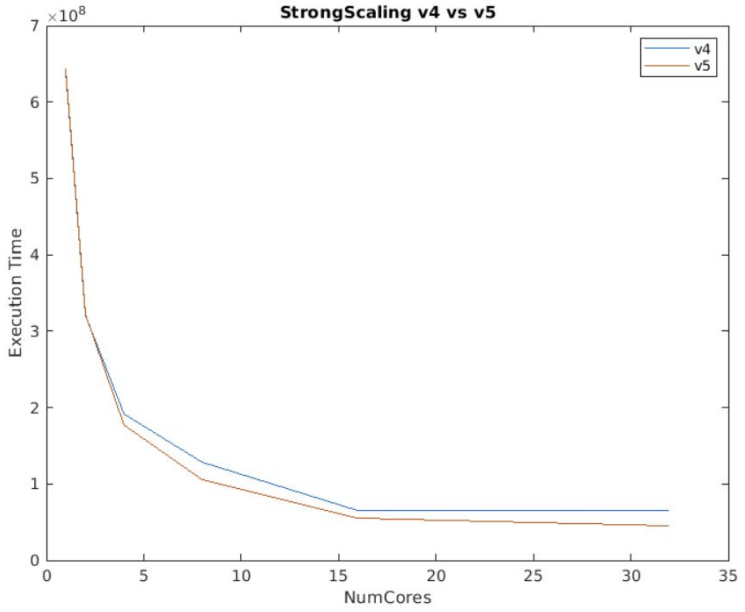
numCores	v5
1	642938001
2	322617000
4	177820000
8	104782000
16	54592000
32	44812000

In this image we can see the execution of the fifth version with 128 cores which gives us a run time of 410 740 000 ns which is better than the result obtained with 32 cores in the chart above but it's not a great improvement with the amount of extra cores used.

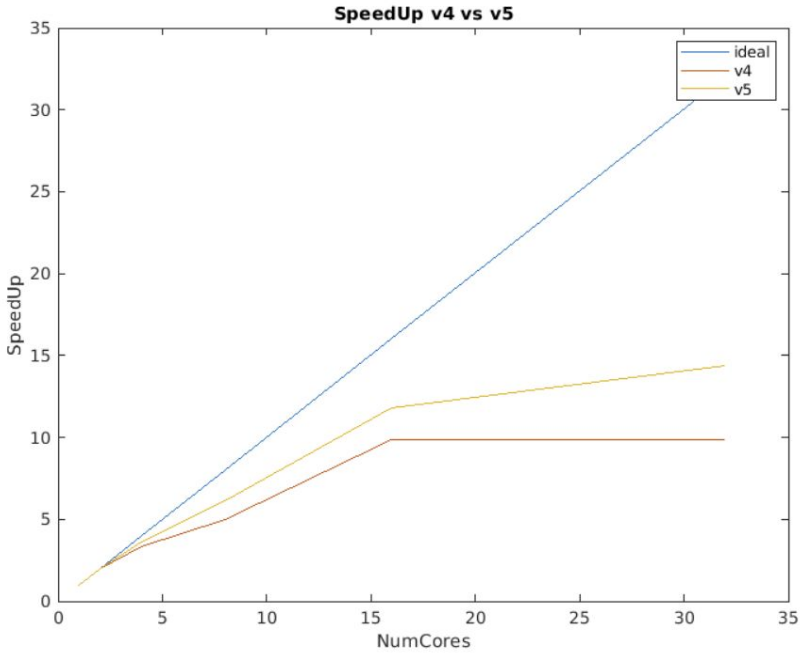


v5 simulation

In the first graph below we can see a comparison of the execution time of the last two versions, we can appreciate that the last version is getting a lower run time while the forth one stays the same, due to its limit on dependencies of the selected task.



Last but not least this graph represents the speedup of the two versions of the same program, clearly if the fifth version hasn't reach it's scalability limit it has room to improve its speedup, sadly i won't get any far better because as we explained earlier with 128 cores the gain of execution time is not that much which means it may be close to the strong scalability limit.



Understanding the parallel execution of 3DFFT

In this section we are going to review the timings of a program execution under three different parallelization strategies. The initial source code does not change between versions; the only added changes are the OMP calls in order to obtain different tasks decompositions.

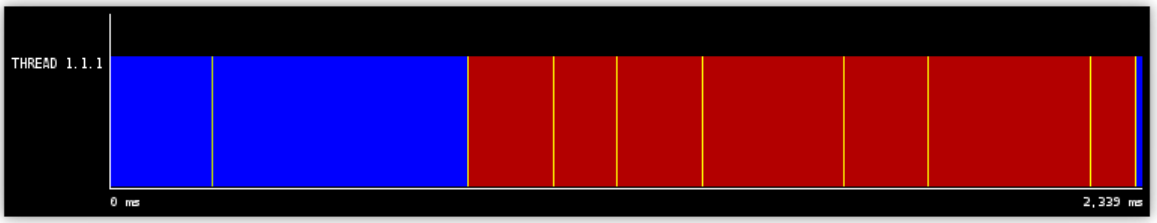
The following table summarizes the obtained results. The first column corresponds to an approximation of the parallel fraction (φ) for each code version, and the next one ($\mathbf{S}_{\infty}$) corresponds to an speed-up upper bound based on the parallel fraction. This two columns are just theoretical approximations. The third, fourth and fifth columns instead, are real values, extracted with *Extrae* during the execution.

Version	φ	$\mathbf{S}_{\infty}$	$\mathbf{T}_1$	$\mathbf{T}_8$	$\mathbf{S}_8$
Initial	≈ 0.65	2.86	2339 ms	1352 ms	1.73
Improved φ	≈ 0.90	10	2347 ms	1037 ms	2.26
Final	≈ 0.90	10	2479 ms	625 ms	3.94

Paraver execution inspection

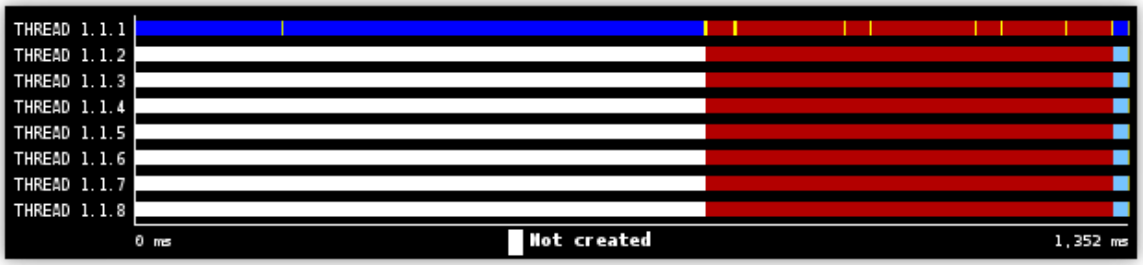
Initial Version

In this first iteration we have three out of four major loops parallelized, when we execute this version on sequential we obtain the following time graph from paraver:



Initial 1 thread exec.

As we can appreciate the sequential section of the program is located on the first blue part of the execution, the red side corresponds to the parallelized one, but this are not good results because most of the time the program is waiting for synchronization between the end of the tasks which means we are have a huge overhead problem, when we change to the 8 thread simulation we appreciate the same results and confirm which is the sequential section.

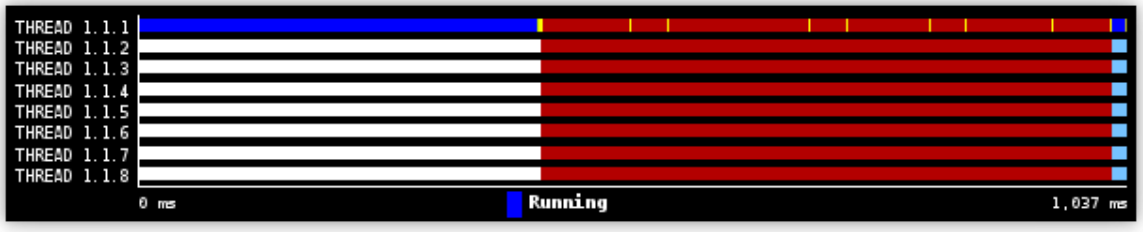


The speedup obtained with 8 threads is 1.73 whereas compared to the theoretical 2.86 it's not a bad result.

Improved φ version

In this second iteration we are told to uncomment the last of the four loops so we have the full heavy load of the program parallelized.

When it comes to the theoretical results we get that the potential maximum speedup is 10 so when we executed this improved version the results are pretty mediocre, the synchronization section is not improved.



Improved φ ver.

If we compare the execution time on the last version to this one we only get a bit of improvement and if we compare the 2.26 speedup to the maximum 10 we theoretically can get, it's a really bad optimization.

The problem in this case stays the same as before, the overhead on creating so much tasks makes our program stay so much time in synchronizing than executing code.

Final version

Finally in this last iteration we get rid of the overheads that come from finer granularity and we have a great improvement compared to the last to iterations of the code.

When we move the parallelization one loop back the amount of tasks is considerably lower so the program can stay most of the time computing the result.

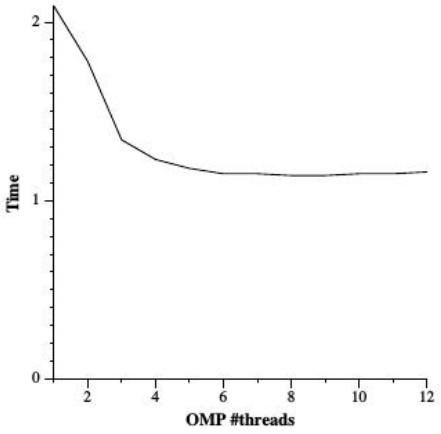
Overhead Reduction

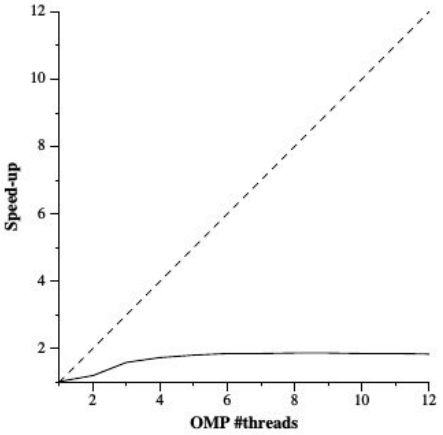
In this case the execution time is reduced from 1037 ms to 625 ms, that's a quite an improvement and so does the speedup that in this case is almost doubled. The comparison between the theoretical one and the one we got doing this upgrade is not that bad we still lack a 60% of the total theoretical speedup, but it's not that bad than the cases before.

Strong Scaling Study

Initial Version

Previously in this report we talked about strong scalability. These graphs represent how much the first version of the program scales in terms of absolute time and speed up.

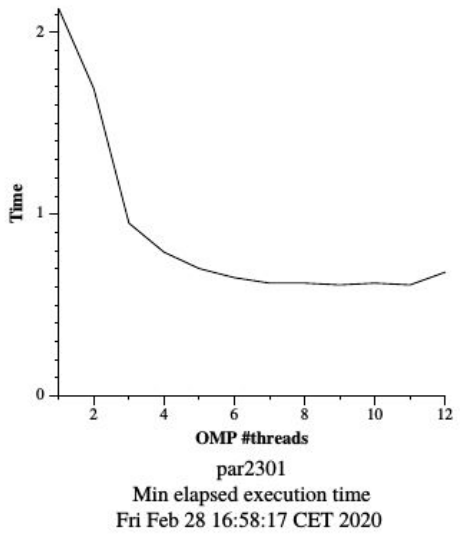


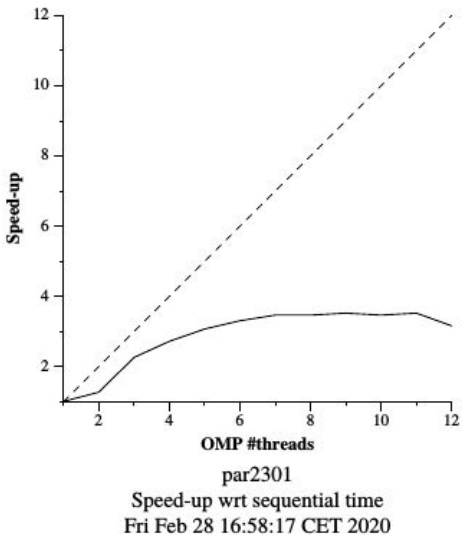


As we can see, there is a clear stabilization of the first curve when the program uses six threads. This is also shown at the speed up graph; even though the program can use up to six threads, the execution speed has not even got doubled, so we can consider this program version almost does not scale.

Improved φ version

In the second version, after increasing the parallelization portion of the code, the scalability improved.



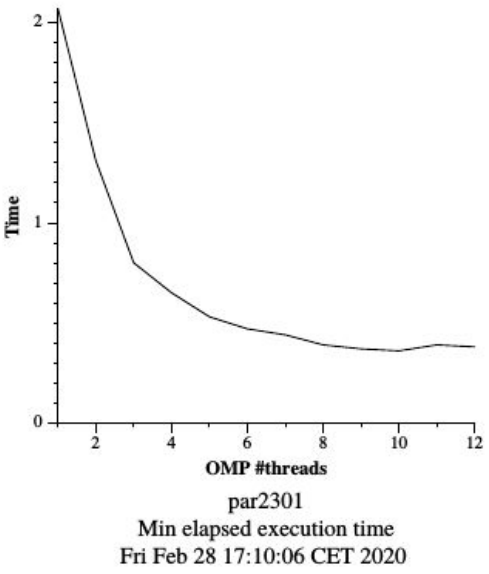


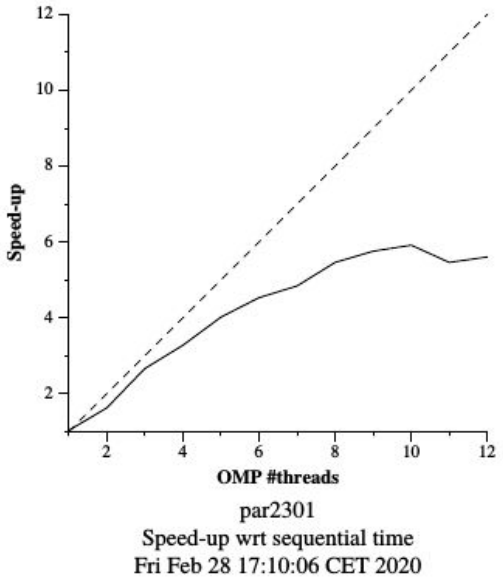
In front of the initial version, we continue having an stabilization when using six threads. The thing which makes the difference is that the first downhill goes deeper than before, achieving doubling the speed.

Nevertheless after using ten threads, due to the overheads, the execution time grows, which suggests that increasing the number of threads would not be a good idea.

Final version

And last but not least, these are the scalability graphs of the last version.





In the first graph, we can see how the time gots reduced as the number of threads increases up to ten. Once the program reaches the use of ten cores, the parallelization overheads start to penalize.

The first downhill goes even deeper than the previous one, achieving a larger speed up (up to six times faster with ten threads).