Antonio J Cabrera
Paul Gazel-Anthoine
PAR1401

Lab 1: Experimental setup and tools

Spring 2018-19

Index

1.1 – Node architecture and memory

	boada-1 to boada-4	boada-5	boada-6 to boada-8
Number of sockets per node	1	1	1
Number of cores per socket	6	6	8
Number of threads per core	2	2	1
Maximum core frequency	2.4 GHz	2.6 GHz	1.7 GHz
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)	12 MB	15 MB	20 MB
Main memory size (per-socket)	12 GB	31 GB	16 GB
Main memory size (per-node)	12 GB	31 GB	16 GB

Table 1: Boada Architecture

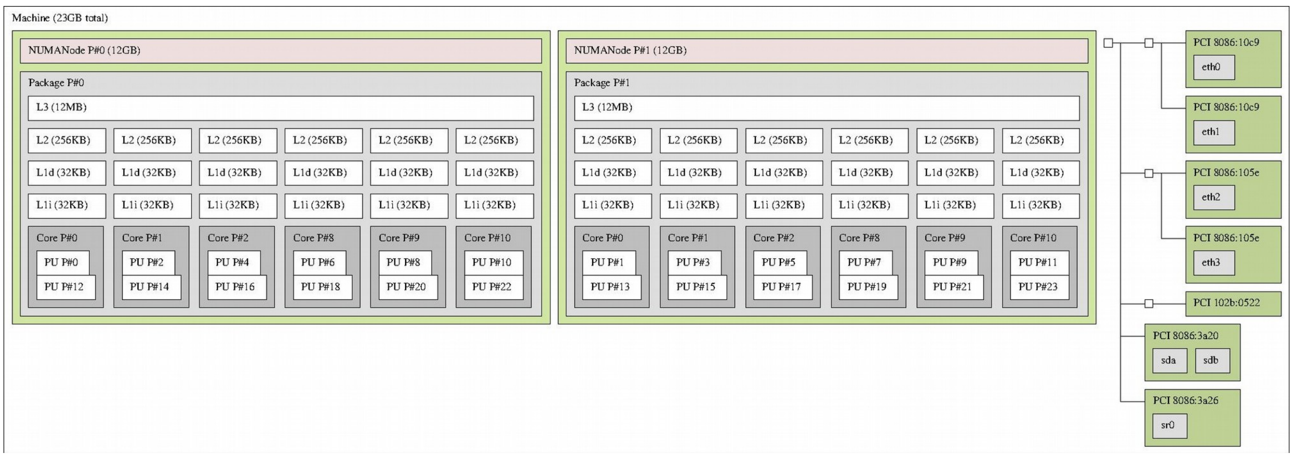


Figure 1: Boada-1 Architectural diagram

The architecture shown in figure 1 will allow us to work in parallel with 12 cores (24 threads) with two levels of cache each (64KB L1 + 256KB L2).

Each group of 6 cores share a cache memory (L3) of 12MB and a main memory of 12GB.

1.2 – Serial compilation and execution

In the pi_seq.c file, the function invocation and data structures used to measure execution time are getusec(), which uses the timeval struct to get the time of the day at a given time. By getting the time at the beginning of the program execution and at the end, it can compute the time spent in execution.

The binary is executed with two parameters (program and size) or else it will show the Usage guide. The execution is run through GNU usr/bin/time.

1.3 – Compilation and execution of OpenMP programs

The new lines indicate to the compiler which part of the code we want to execute in parallel.
Each thread has a private variable x that gets its value by using the thread number. All x's are used to compute the accumulated sum which are eventually summed via reduction as indicated in the 'pragma' instruction.

Results:

- 1 thread time elapsed → 3.96 seconds (3.94 user, 0.00 system) at 99% CPU
- 8 threads time elapsed → 0.63 seconds (4.84 user, 0.04 system) at 775% CPU

export OMP_NUM_THREADS=$3 → It gathers the information at the third parameter and sets the environment variable OMP_NUM_THREADS with that value. That variable is later read inside the program.

	1 thread	8 threads
Interactive	3.9434 seconds	0.5344 seconds
Queued	3.9431 seconds	0.5315 seconds

Table 2: Pi computation, execution time comparison

As seen in Table 2, there was not a major difference between interactive and queued executions. We believe this data is so similar because at the time of our testing, boada-1 was not heavily used by other students. We know that, in general, executing interactively while other people are using the same machine is not a good idea, since it would give skewed results because of some resources being used; testing is only to be done using the queued version.

1.4 – Strong vs Weak scalability

Strong scalability refers to how execution time is affected when adding more processors to deal with a constant amount of work.

Weak scalability refers to how execution time is affected when increasing proportionally the amount of work and the number of processors.

Strong Scalability (boada-4)

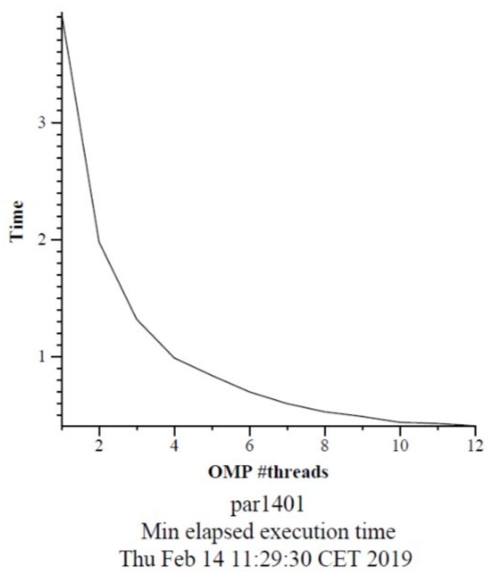


Figure 2: Time - Strong scalability (boada-4)

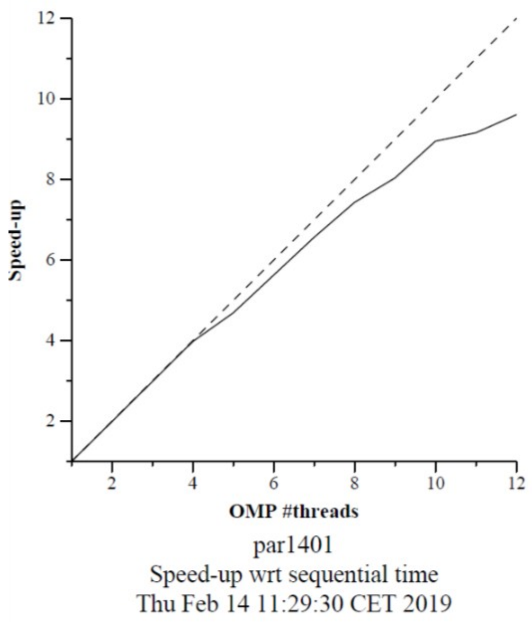


Figure 3: Speedup - Strong scalability (boada-4)

In figures 2 and 3, we can see that strong scalability for this program is almost ideal for up to 8 threads. After that, looking at figure 3, the line becomes flatter.

Weak Scalability (boada-3)

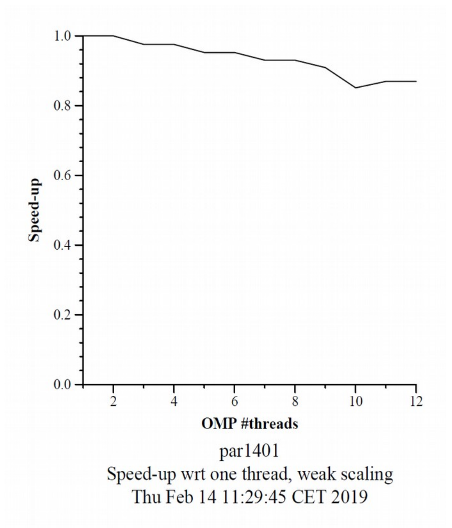


Figure 4: Speedup - Weak scalability (boada-4)

In figure 4, we can see that execution time is very stable, although not entirely flat. This means we lose some efficiency when we increase the size of the problem and the number of processors proportionally.

Strong Scalability (boada-5)

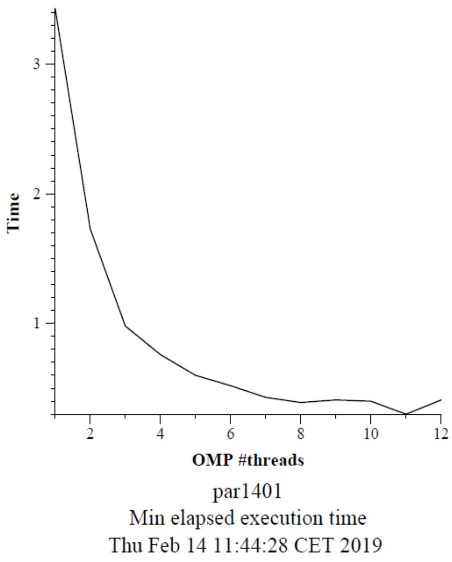


Figure 5: Time - Strong scalability (boada-5)

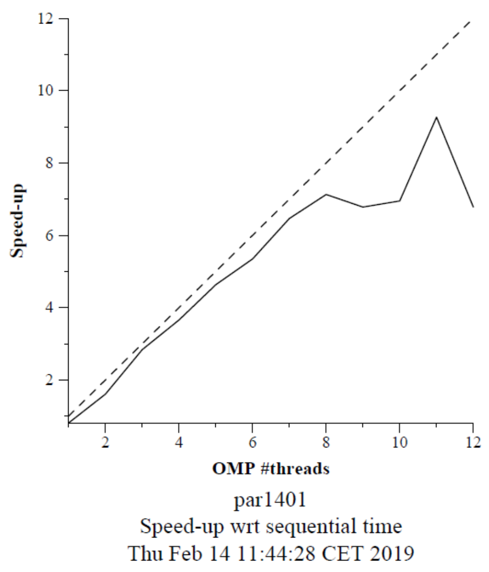


Figure 6: Speedup - Strong scalability (boada-5)

Similarly to boada-4, for up to 8 threads, execution time sees a quite ideal drop when increasing the number of processors. After that, when we get close to 0 seconds, the line flattens.

<u>Weak Scalability (boada-5)</u>

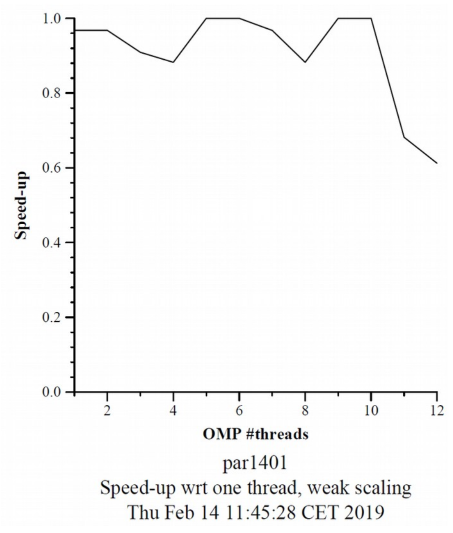


Figure 7: Speedup - Weak scalability (boada-5)

In figure 7, we see it is a bit of a mess and we cannot predict how execution time will evolve, although from 11 threads on, it seems that efficiency drops dramatically. It looks very efficient for a number of threads that follows this rule:

threads = 4k+2

<u>Strong Scalability (boada-6)</u>

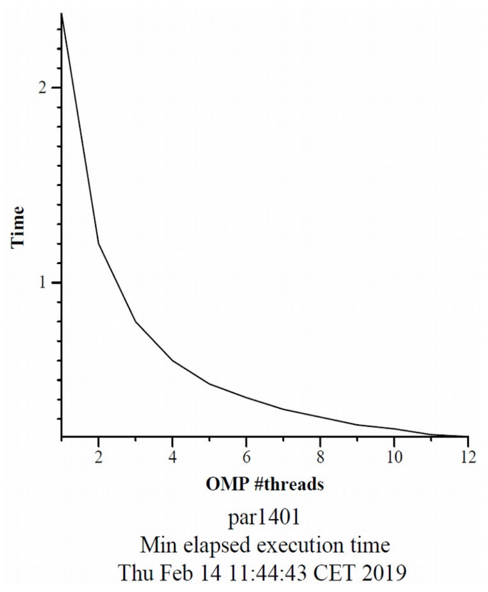


Figure 8: Time - Strong scalability (boada-6)

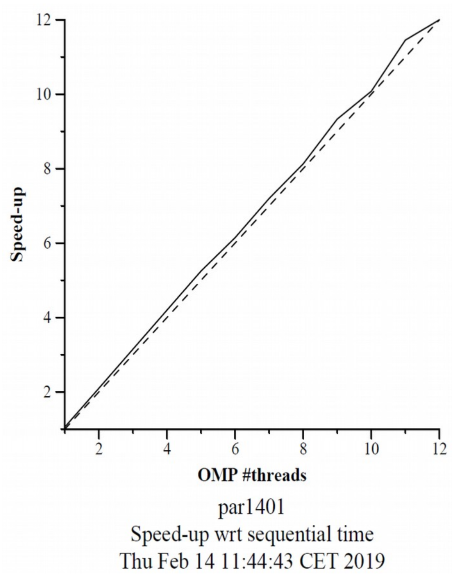


Figure 9: Speedup - Strong scalability (boada-6)

Figures 8 and 9 show that boada-6 follows the absolute ideal regarding strong scalability for this example.

Weak Scalability (boada-8)

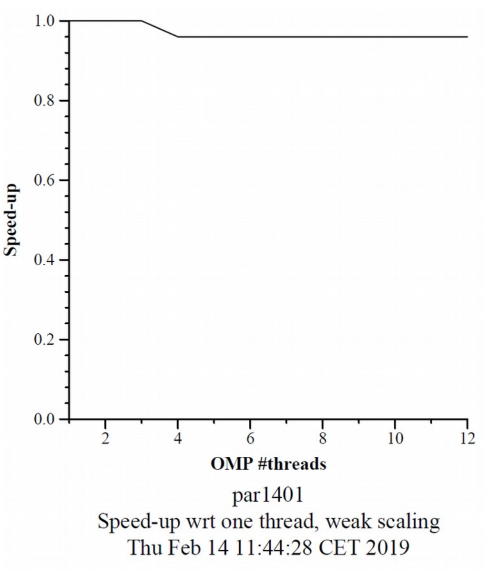


Figure 10: Speedup - Weak scalability (boada-8)

According to figure 10, boada-8 also behaves close to ideally regarding weak scalability.

Systematically analysing task decompositions with tareador

2.3 – Exploring new task decompositions for 3DFFT

	T1	T∞	Parallelism
Sequential	639780	639780	1
V1	639780	639760	1,00003
V2	639780	361525	1,77
V3	639780	155065	4,13
V4	639780	64750	9,88
V5	639780	7861	81,39

Table 3: Parallelism study

By looking at table 3, we can already see a good level of parallelism for versions 4 and, specially, 5. This means that, if we have enough processors available, we will get a great increase in performance.

Parallel execution for different number of processors (V4)

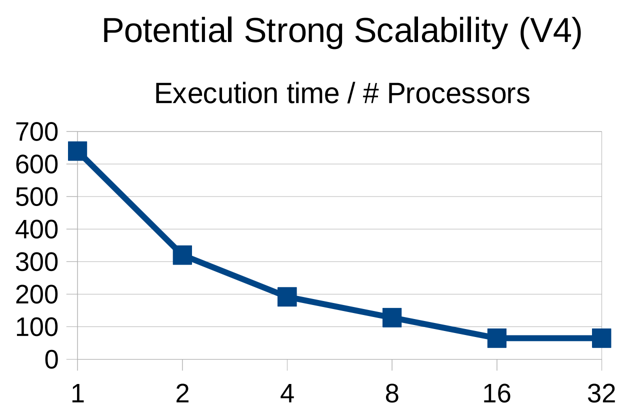


Figure 11: V4 time study

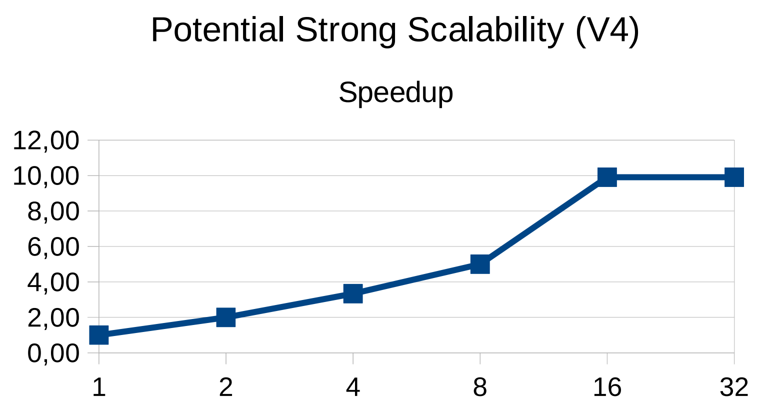


Figure 12: V4 speedup study

	V4	
Processors	Exec Time (ms)	Speedup
1	639,8	1,00
2	320,3	2,00
4	191,9	3,33
8	128,0	5,00
16	64,6	9,90
32	64,6	9,90

Figure 13: V4 Times and Speedup

The limit is N, which gives the number of iterations and maximum number of threads than can be working in parallel if we decide to create a task for a loop.

Once we reach 10 threads, we reach said limit.

Parallel execution for different number of processors (V5)

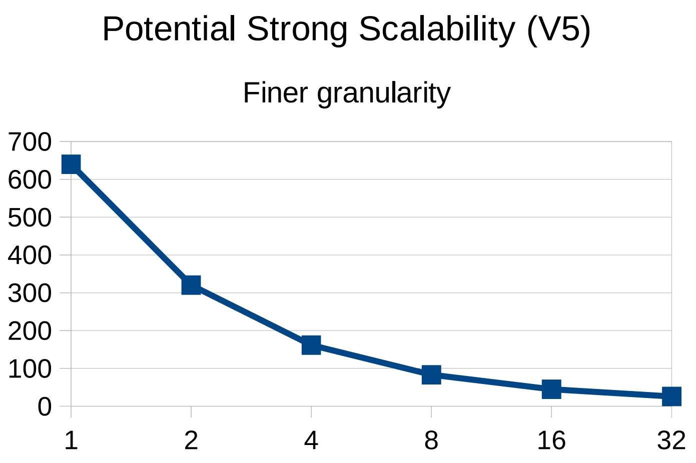


Figure 14: V5 time study

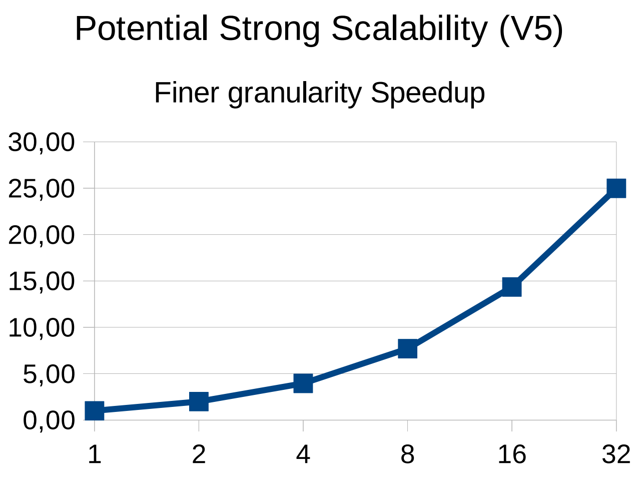


Figure 15: V5 speedup study

	V5	
Processors	Exec Time (ms)	Speedup
1	639,8	1,00
2	320,3	2,00
4	161,5	3,96
8	83,1	7,70
16	44,6	14,35
32	25,6	24,99

Figure 16: V5 Times and Speedup

Compared to V4, it is worth it to go to this granularity level if our machine is capable of working with 4 processors. In any case, the speedup increase is more obvious when we have 32 processors or more.

In this case, N = 100, so the speedup will not improve after we reach that number of processors.

We can scale to a higher number of processors compared to V4 thanks to the following code:

For each function with 3 nested loops:

```
for (k = 0; k < N; k++)                                         V4 version
      tareador_start_task("A")                                  V5 version
      for (j = 0; j < N; j++)
            tareador_start_task("B")
            for (i = 0; i < N; i++)
                  // some computation
            tareador_end_task("B")
      tareador_end_task("A")
```

V4 only has task "A", while V5 has only task "B". This allows V5 to subdivide even more the problem (higher granularity) and take advantage of having more processors to work with.

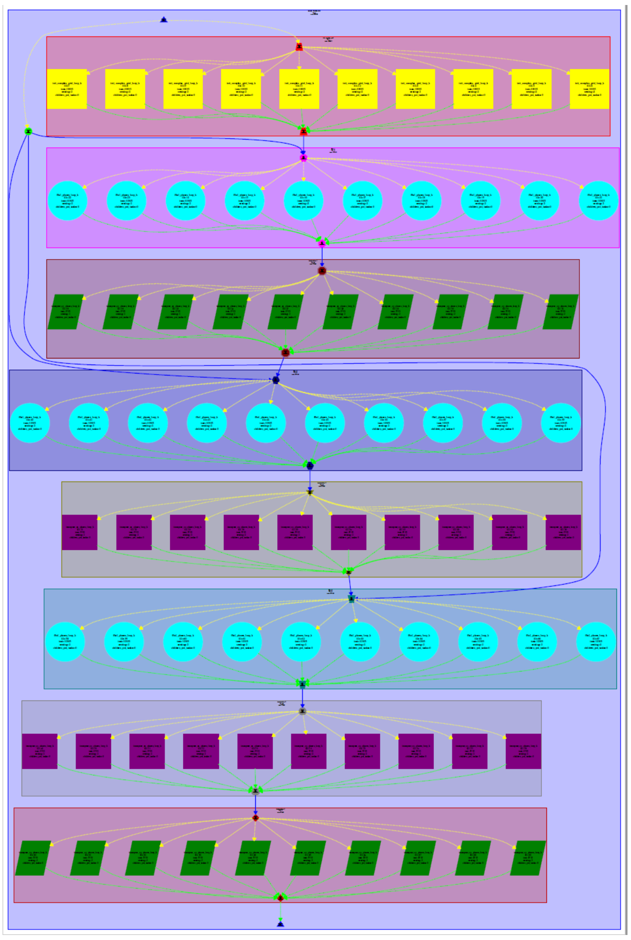


Figure 17: V4 - Dependence graph

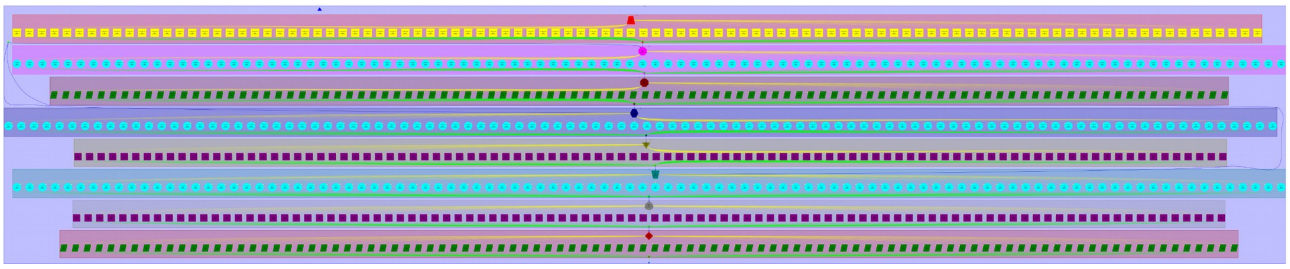


Figure 18: V5 - Dependence graph

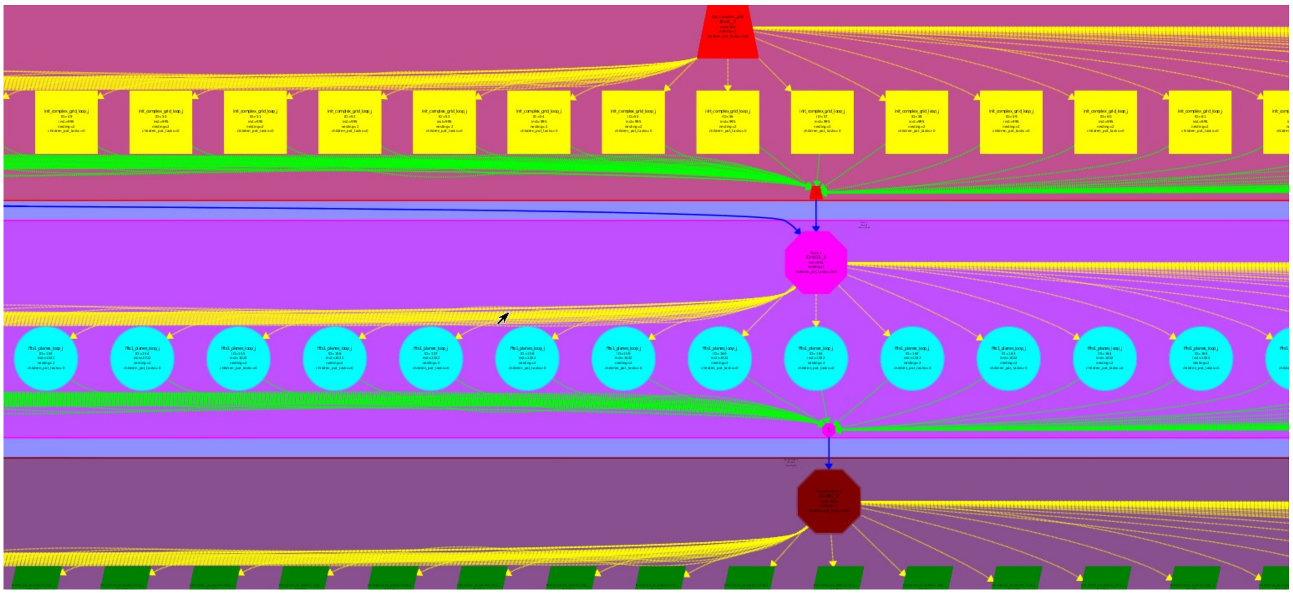


Figure 19: V5 - Dependence graph (zoom-in)

Comparing figures 17 and 18, we can see that they have the same amount of tasks but a great increase in granularity.

It is worth it to have this level of granularity when you have more than N (loop size) processors. If that happens, subdividing at the nested loop level increases the parallelism potential.

Understanding the execution of OpenMP programs

3. After analyzing the performance (see next sections), we can say the scalability is not appropriate.

3.2.1 – Initial Version

1)

We obtain Tpar with the configuration ‘Parallel functions duration’.

Tpar = 1,5315s

Tseq = T1 – Tpar = 2,54 – 1,53 = 1,0098s

Parallel fraction (ϕ) = 0,6026

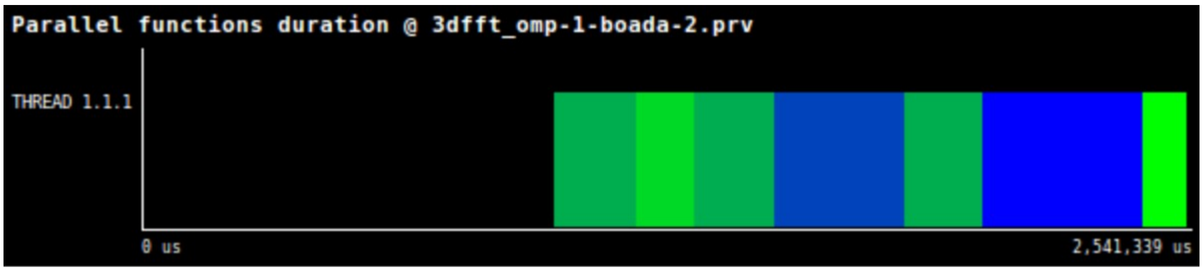


Figure 20: Paraver obtained applying 'Parallel functions duration' configuration (1 thread)

2)

T8 = 1,4744s

S8 = 1,7097s

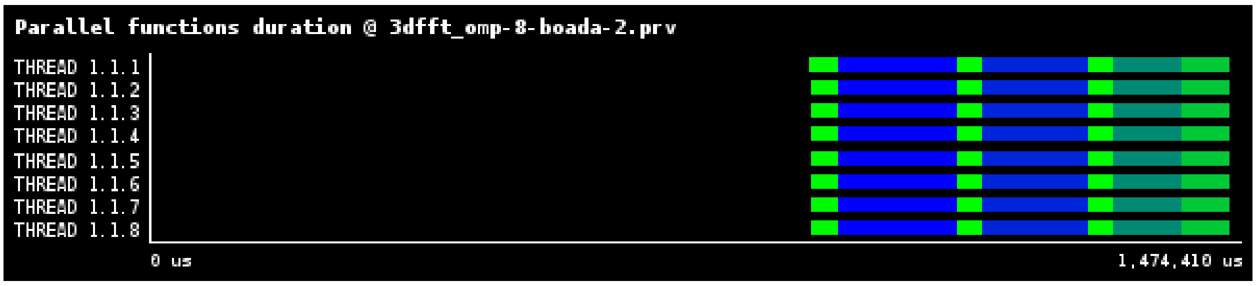


Figure 21: Paraver obtained applying 'Parallel functions duration' configuration (8 threads)

	Running	Not created	Synchronization	Scheduling And Fork/Join	I/O	Others
THREAD 1.1.1	95.94	0.00	3.94	0.10	0.02	0.00
THREAD 1.1.2	37.69	61.32	0.98	0.00	0.02	0.00
THREAD 1.1.3	37.38	61.32	1.29	0.00	0.02	0.00
THREAD 1.1.4	34.40	61.32	4.27	0.00	0.02	0.00
THREAD 1.1.5	34.56	61.32	4.11	0.00	0.02	0.00
THREAD 1.1.6	37.37	61.32	1.30	0.00	0.02	0.00
THREAD 1.1.7	34.93	61.32	3.73	0.00	0.02	0.00
THREAD 1.1.8	37.57	61.32	1.10	0.00	0.02	0.00
Total	349.83	429.23	20.71	0.10	0.13	0.00
Average	43.73	61.32	2.59	0.10	0.02	0.00
Maximum	95.94	61.32	4.27	0.10	0.02	0.00
Minimum	34.40	61.32	0.98	0.10	0.02	0.00
Stdev	19.78	0.00	1.43	0.00	0.00	0.00
Avg/Max	0.46	1.00	0.61	1.00	0.92	1.00

Figure 22: Percentage of time spent in the different OpenMP states (8 threads)

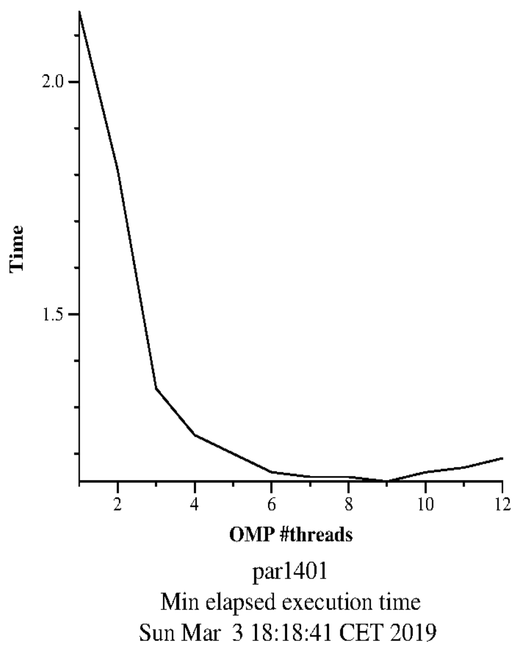
3)

T1 = 2,1065s

T8 = 1,1410s

S8 = 1,846

4)



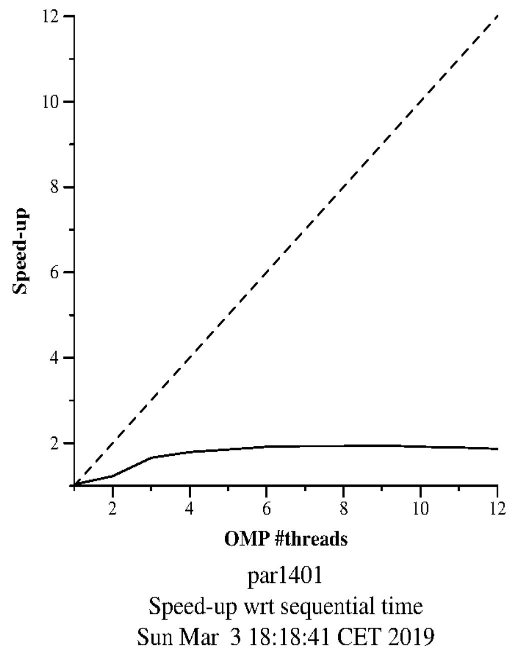


Figure 23: Strong Scalability plots (Initial 3dfft_omp)

The ideal S∞ is 2,51 (by Amhdal's law) and for S8 we get 1,85. However, watching at the plots in figure 23, we see it's difficult to significantly increase the speedup. Thus, we assume there is some overhead that will not be reduced by parallelism and estimate the actual upper limit somewhere around 2.

3.2.2 – Improving ϕ

The function causing the low value for ϕ is 'init_complex_grid'.

T1 = 2,3835s

T8 = 1,0023s

Tpar = 2,1363

ϕ = 0,896

S8 = 2,378

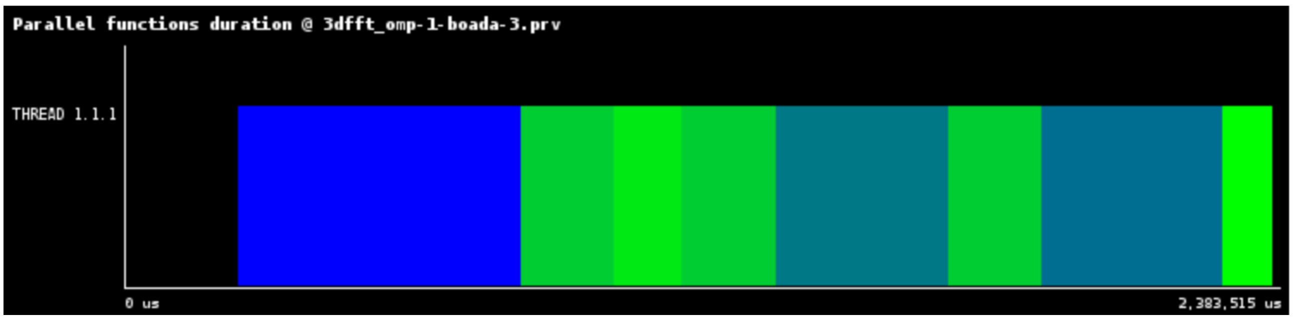


Figure 24: Paraver obtained applying 'Parallel functions duration' configuration (1 thread)

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	96.95	0.00	2.90	0.11	0.03	0.00
THREAD 1.1.2	56.83	39.98	3.17	0.00	0.03	0.00
THREAD 1.1.3	57.96	39.97	2.03	0.00	0.03	0.00
THREAD 1.1.4	57.36	39.98	2.63	0.00	0.03	0.00
THREAD 1.1.5	57.78	39.98	2.21	0.00	0.03	0.00
THREAD 1.1.6	57.12	39.99	2.87	0.00	0.03	0.00
THREAD 1.1.7	57.99	39.98	2.00	0.00	0.03	0.00
THREAD 1.1.8	56.88	40.00	3.09	0.00	0.03	0.00
Total	498.88	279.88	20.90	0.11	0.23	0.00
Average	62.36	39.98	2.61	0.11	0.03	0.00
Maximum	96.95	40.00	3.17	0.11	0.03	0.00
Minimum	56.83	39.97	2.00	0.11	0.03	0.00
Stdev	13.08	0.01	0.44	0.00	0.00	0.00
Avg/Max	0.64	1.00	0.82	1.00	0.91	1.00

Figure 25: Percentage of time spent in the different OpenMP states (8 threads)

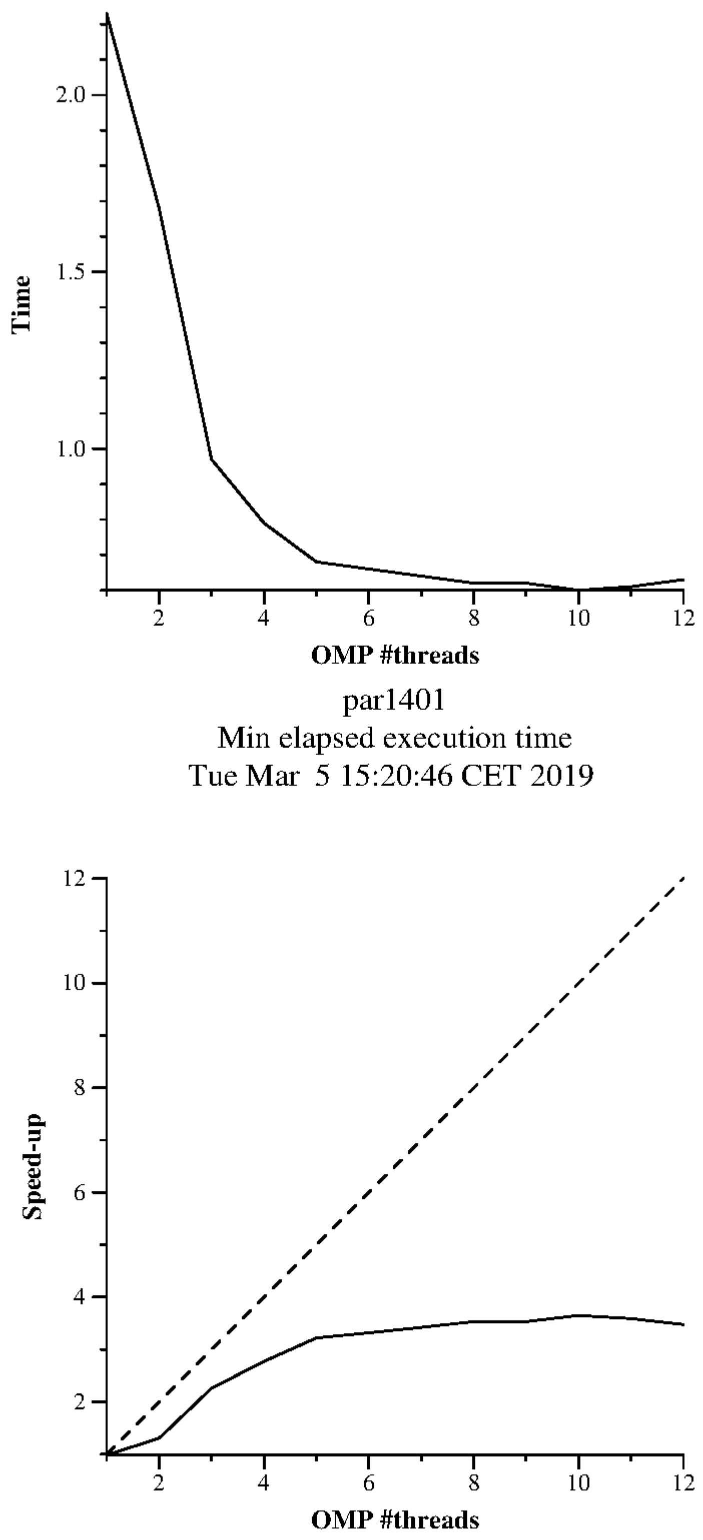


Figure 26: Strong Scalability plots (Improved ϕ)

Similarly to 3.2.1.4, if we compute the ideal speedup for infinite processors, we should achieve an approximated speedup of 9,6. However, looking at the plots in figure 26, we estimate the upper bound to be closer to 4 due to expected overheads.

3.2.3 – Reducing parallelism overheads

T1 = 2,5116s

T8 = 0,8836s

Tpar = 2,2467

ϕ = 0,8945

S8 = 2,8425

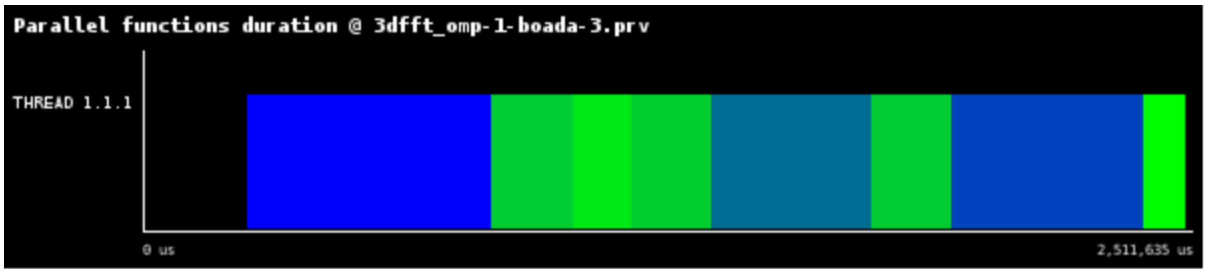


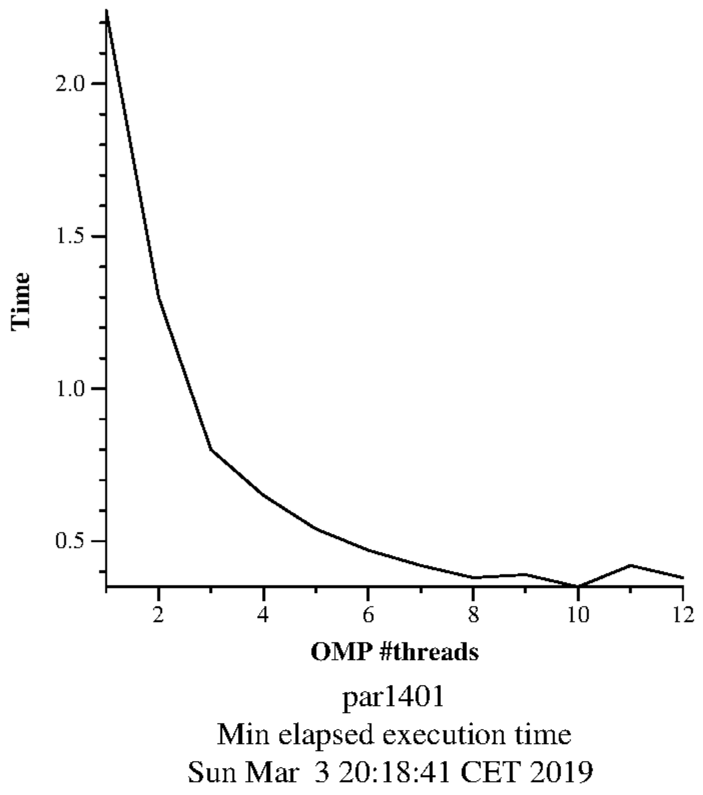
Figure 27: Paraver obtained applying 'Parallel functions duration' configuration (1 thread)

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	99.20	0.00	0.71	0.09	0.00	0.00
THREAD 1.1.2	41.34	51.76	6.90	0.00	0.00	0.00
THREAD 1.1.3	40.02	51.76	8.21	0.00	0.00	0.00
THREAD 1.1.4	45.19	51.76	3.05	0.00	0.00	0.00
THREAD 1.1.5	45.17	51.80	3.03	0.00	0.00	0.00
THREAD 1.1.6	36.95	51.81	11.24	0.00	0.00	0.00
THREAD 1.1.7	47.58	51.80	0.61	0.00	0.00	0.00
THREAD 1.1.8	45.20	51.81	2.99	0.00	0.00	0.00
Total	400.64	362.52	36.74	0.09	0.01	0.00
Average	50.08	51.79	4.59	0.09	0.00	0.00
Maximum	99.20	51.81	11.24	0.09	0.00	0.00
Minimum	36.95	51.76	0.61	0.09	0.00	0.00
Stdev	18.84	0.02	3.55	0.00	0.00	0.00
Avg/Max	0.50	1.00	0.41	1.00	0.35	1.00

Figure 28: Percentage of time spent in the different OpenMP states (8 threads)

Finally, we obtain S∞ of approximately 9,5 using Amhdal’s law. In this case, by reducing overheads, we can get closer to that theoretical number.

The speedup plot in figure 29 doesn't allow for us to estimate an upper bound, but we know it is at least 6, which is a good 50% increase compared to the 3.2.2 results.



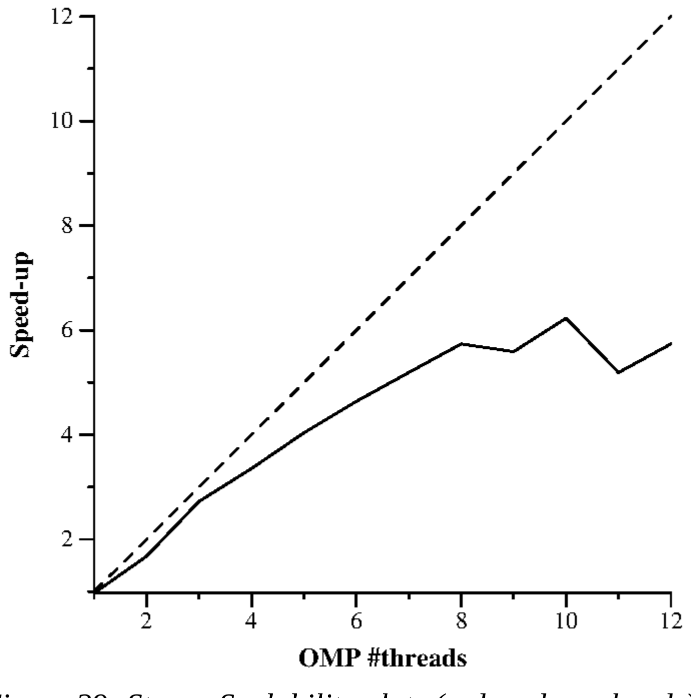


Figure 29: Strong Scalability plots (reduced overheads)

Summary: understanding the parallel execution of 3DFFT

Version	ϕ	S∞	T1	T8	S8
Initial version in 3dfft_omp.c	0,6	2,51	2,54	1,47	1,71
New version with improved ϕ	0,9	9,61	2,38	1	2,38
Final version with reduced parallelisation overheads	0,89	9,48	2,51	0,88	2,84

Table 4: Performance results of 3dfft

Conclusions

In this sessions we have learned the basics of parallelism and some tools to evaluate potential parallelism.

We have been working with dependence graphs, Paraver, Strong and Weak scalability graphs.

Through a few examples, we have seen the importance of overheads in limiting the maximum parallelism attainable due to physical limitations, even though the theoretical calculation (Amhdal’s law) aims higher.

It is really important that we understand these concepts so we can minimize overheads and reach the maximum potential in our parallel programs.