Lab Report #1

CASSARD Sebastien 19960526-T313

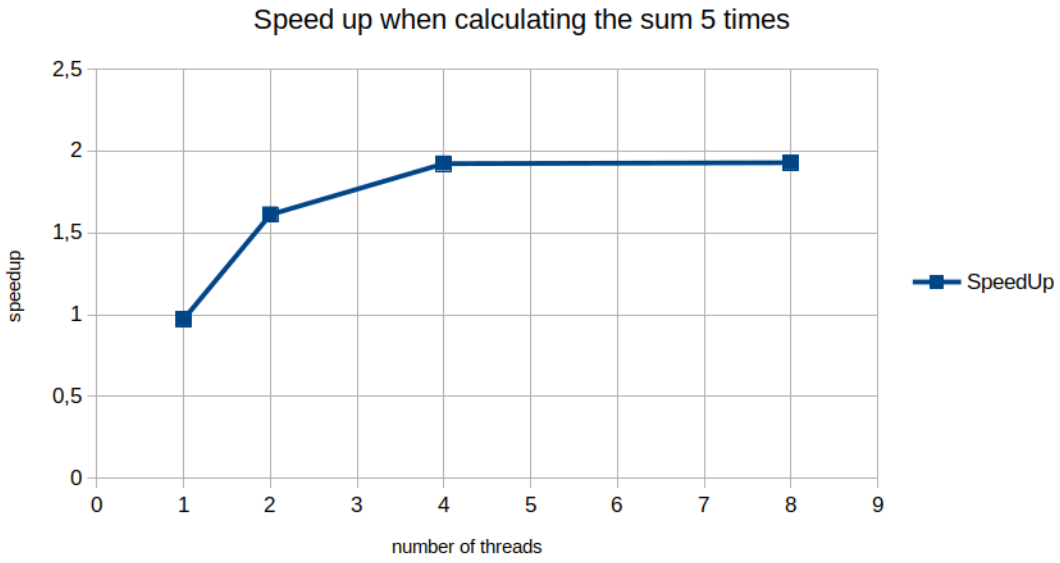
November 11, 2018

NOTE : *For the following results, the program is run for an array size of 500 000 000 elements.*

1 Task 1: Using Pthreads

1.1 Speedup for the Pthreads program

1.1.1 Result when calculating the sum 5 times



Number of threads	Serial	Pthreads	Speed up
1	11.628s	11.956s	0.97
2	11.628s	7.213s	1.61
4	11.628s	6.046s	1.92
8	11.628s	6.028s	1.93

The table above shows the different execution times of the program as well as the speedup for each number of threads. With the help of Amdahl's law it is possible to determine approximately which percentage of the program is serial and which percentage is parallel.

Amdahl's law tells us that:

$$Speedup = \frac{1}{(1-F) + \frac{F}{S}}$$

Where F is the fraction of the programme that is enhanced by a factor S. Here S correspond to the number of threads. So knowing S and $Speedup$ we can deduce $1-F$ and F which are respectively the percentage of the program which run in serial and the percentage which runs in parallel.

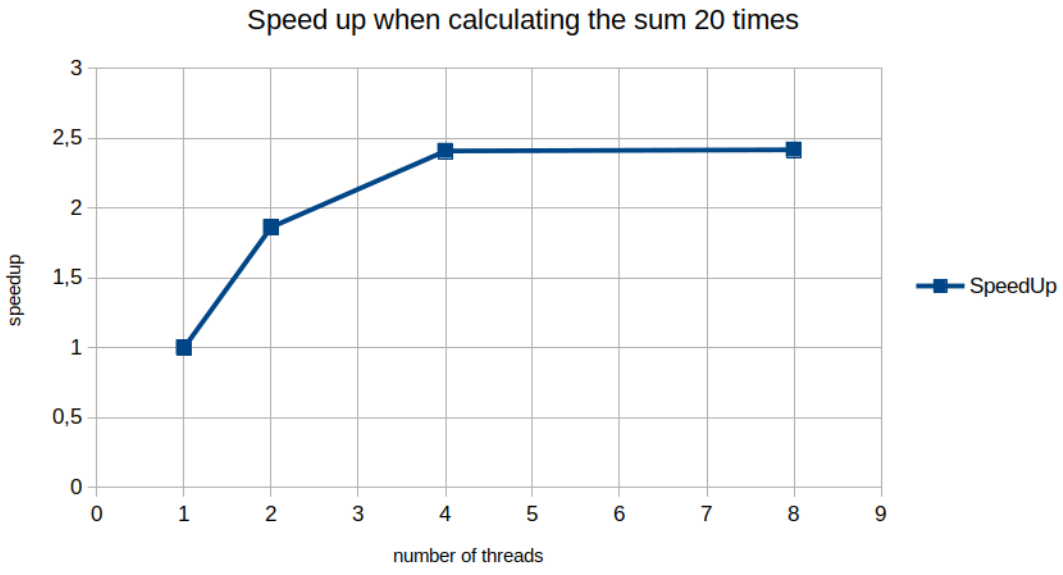
From the above formula, the following equality can be deduced:

$$F = \frac{S(\frac{1}{Speedup} - 1)}{1 - S}$$

Which gives us the following results :

Number of threads	F	1-F
2	0.76	0.24
4	0.64	0.36
8	0.55	0.45

1.1.2 Result when calculating the sum 20 times



Number of threads	Serial	Pthreads	Speed up
1	38.247s	38.245s	1.00
2	38.247s	20.525s	1.86
4	38.247s	15.874s	2.41
8	38.247s	15.819s	2.42

As in the previous section, the percentages of the serial $(1 - F)$ and parallel (F) fractions are deducted:

Number of threads	**F**	**1-F**
2	0.92	0.08
4	0.78	0.12
8	0.67	0.23

1.2 Conclusions

In the two cases studied above, we notice an improvement in execution time with the increase in the number of threads. However, several elements need to be put into perspective.

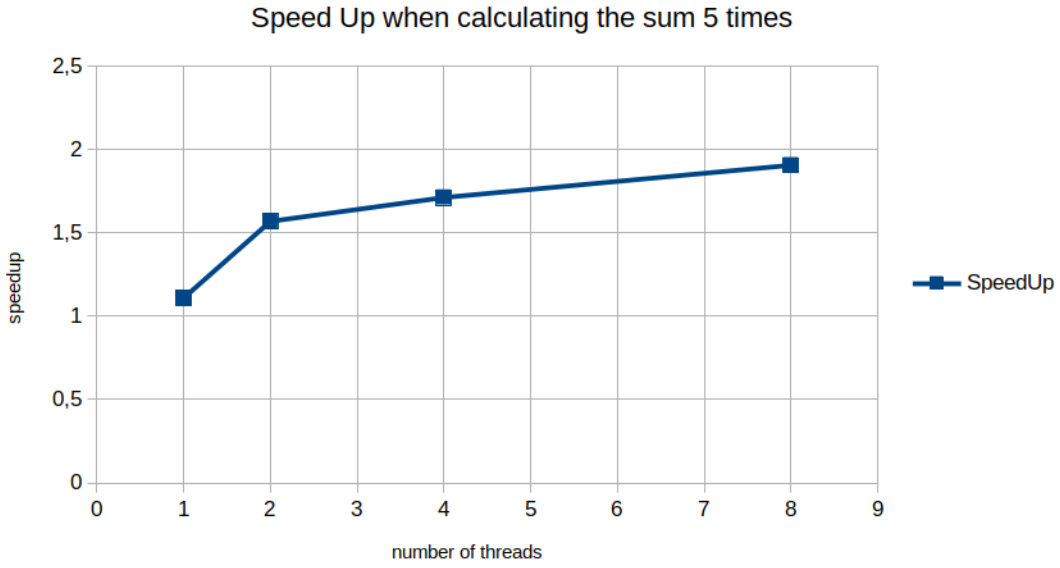
Firstly, we notice that the speedup stagnates for more than 4 threads. This can be explained by the fact that the measurements were made on an Intel Core i3-6100U cpu which has only 4 cores.

Secondly, we can see that the higher the number of threads is, the higher the serial code fraction is. This is probably due to the overhead caused by the creation of the different threads.

Despite these elements, we can observe much better results with a higher number of threads.

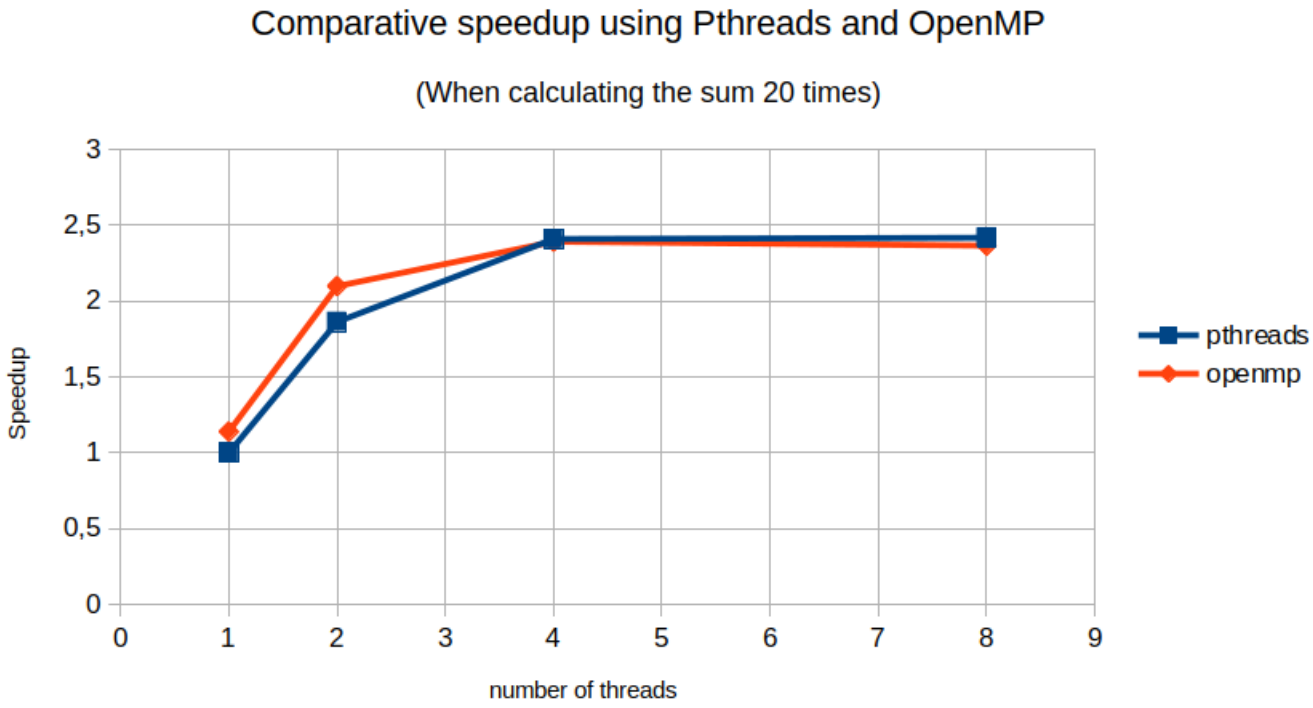
2 Task 2: Using OpenMP

2.1 Speedup for the OpenMP program



Number of threads	Serial	OpenMP	Speed up
1	11.628s	10.474s	1.11
2	11.628s	7.405s	1.57
4	11.628s	6.787s	1.71
8	11.628s	6.096s	1.91

2.2 Comparing OpenMP and Pthreads



Number of threads	Pthreads Speed up	OpenMP Speed up
1	1.00	1.14
2	1.86	2.10
4	2.41	2.39
8	2.42	2.36

We can see that the results obtained with OpenMP and Pthreads are very similar. However, OpenMP only requires one line of code where Pthreads requires about ten. In the case of a loop to be parallelized, the use of OpenMP will be preferred. However, if we need to start a separate process which shouldn't block the main thread, then maybe Pthreads would be a better choice as it allows us to have you have extremely fine-grained control over thread management.

Lab Report #2

CASSARD Sebastien 19960526-T313

November 11, 2018

NOTE : *For the following results, the program is run for an array size of 100 000 000 elements, summed 5 times. Moreover, the processor used for this TP only has 4 cores, so we will only test for a maximum number of threads equal to 4.*

1 Part A:

1.1 OpenMP vs Pthreads

	Nb of threads	Execution time	Nb of instructions
Serial	1	2,32s	6 585 596 212
Pthreads	2	1,43s	6 591 510 482
	4	1,18s	6 588 130 565
OpenMP	2	1,46s	8 091 644 996
	4	1,36s	8 096 455 632

As observed in the previous lab, the execution time using OpenMP or Pthreads is significantly better than that of the serial program. However, we notice that OpenMP is slightly slower than Pthreads, which can be explained by a larger number of instructions for OpenMP.

	Nb of threads	L1 data cache loads	L1 data cache loads misses
Serial	1	4 227 718 060	77 604 419
Pthreads	2	4 229 583 074	77 492 397
	4	4 228 680 613	77 825 815
OpenMP	2	5 228 893 077	77 634 564
	4	5 229 441 053	74 131 057

The number of L1 data cache loads are almost identical regardless of the program used.

	Nb of threads	Last level cache loads	Last level cache loads misses
Serial	1	417 765	204 371
Pthreads	2	509 702	254 084
	4	523 455	267 960
OpenMP	2	669 443	229 412
	4	472 027	223 814

Similarly, last level cache misses are very similar regardless of the program considered. More particularly, we notice that the number of cache misses is slightly higher during parallel executions.

1.2 Importance of loop order

The same measurements as in the previous section were made on modified versions of programs in which the inner and outer loops of the sum_array function were swapped.

	Nb of threads	**Execution time**	**Nb of instructions**
Serial	1	1,76s	7 486 298 854
Pthreads	2	1,14s	7 489 825 076
	4	1,18s	7 488 834 876
OpenMP	2	6,35s	11 979 911 869
	4	7,80s	10 680 977 720

	nb of threads	**L1 data cache loads**	**L1 data cache loads misses**
Serial	1	4 727 762 314	27 424 890
Pthreads	2	4 728 983 579	27 168 413
	4	4 728 660 561	27 331 033
OpenMP	2	7 292 566 075	206 406 395
	4	6 466 020 445	268 210 984

	Nb of threads	**Last level cache loads**	**Last level cache loads misses**
Serial	1	497 846	161 327
Pthreads	2	525 978	173 498
	4	617 231	183 161
OpenMP	2	533 410	163 163
	4	638 162	173 234

The inversion of the loops in the sum_array function allows better overall performance. We observe a better runtime, as well as much less cache misses, both for L1 cache misses and last level cache misses. By reversing the order of the loops, we take advantage of the principle of locality. The consecutively accessed values are on the same cache line, drastically reducing the number of misses.

However, it is noted that OpenMP has lower performances in this configuration. This is probably due to a bad implementation of *pragma* commands on my part.

2 Part B:

	Padding disabled		Padding enabled	
Nb of threads	**nb of inst**	**L1 data cache misses**	**nb of inst**	**L1 data cache misses**
1	7504956173	79603	8505204732	83776
2	15011202705	92859638	17007138438	166253
4	30047905704	265941494	34014299882	227491

Activating the padding significantly reduces the number of cache misses. Even if the number of instructions with padding is higher, this element is largely compensated by the low number of cache miss.

Nb of threads	**32 bytes padding**	**64 bytes padding**
1	75271	83776
2	95678028	166253
4	253685427	227491

The number of cache misses for a padding of 32 bytes gives almost the same results as the program without padding. Indeed, 64 bytes being the size of a cache line, a padding of 64 bytes allows to exploit the locality cache phenomenon (as the variables are no longer all on the same cache line, we have less false sharing miss).

Lab Report #3

CASSARD Sebastien 19960526-T313

November 11, 2018

1 Raw results:

1.1 Mutex lock

Nb of threads	Runtime	Nb of instructions	L1 data cache accesses	L1 dcache misses
1	1,62s	6 045 620 819	1 721 553 805	199 762
2	5,35s	13 222 059 660	3 564 390 543	56 876 518
4	4,94s	13 769 662 620	3 727 566 446	85 726 698

1.2 Basic CAS lock

This lock is implemented using a Compare and Swap atomic instruction. The latter is used when acquiring the lock, but not when releasing the lock (in our program, when the unlock release function is called, the lock value is necessarily 1).

Nb of threads	Runtime	Nb of instructions	L1 data cache accesses	L1 dcache misses
1	0,69s	1 330 547 211	442 945 574	370 153
2	3,21s	2 166 260 302	652 544 211	74 814 537
4	6,20s	4 497 775 771	1 245 982 490	136 007 213

Compared to the previous lock, the basic CAS lock seems more efficient for a small number of threads. For a large number of threads, the performance of the CAS lock is worse, especially because the threads are constantly trying to acquire the lock until they get it.

1.3 Optimized CAS lock with yield

As we have seen before, the weak point of the basic CAS lock is that threads constantly try to access the lock until they have it. Here we will consider an improved version of this lock, in which if a thread fails to acquire a lock, it yields the processor. This version offers better performance than the basic CAS lock for a large number of threads.

Nb of threads	Runtime	Nb of instructions	L1 data cache accesses	L1 dcache misses
1	0,67s	1 324 317 917	441 150 548	106 993
2	0,81s	2 358 698 096	779 336 794	6 130 807
4	0,86s	3 116 518 079	1 003 780 514	5 105 680

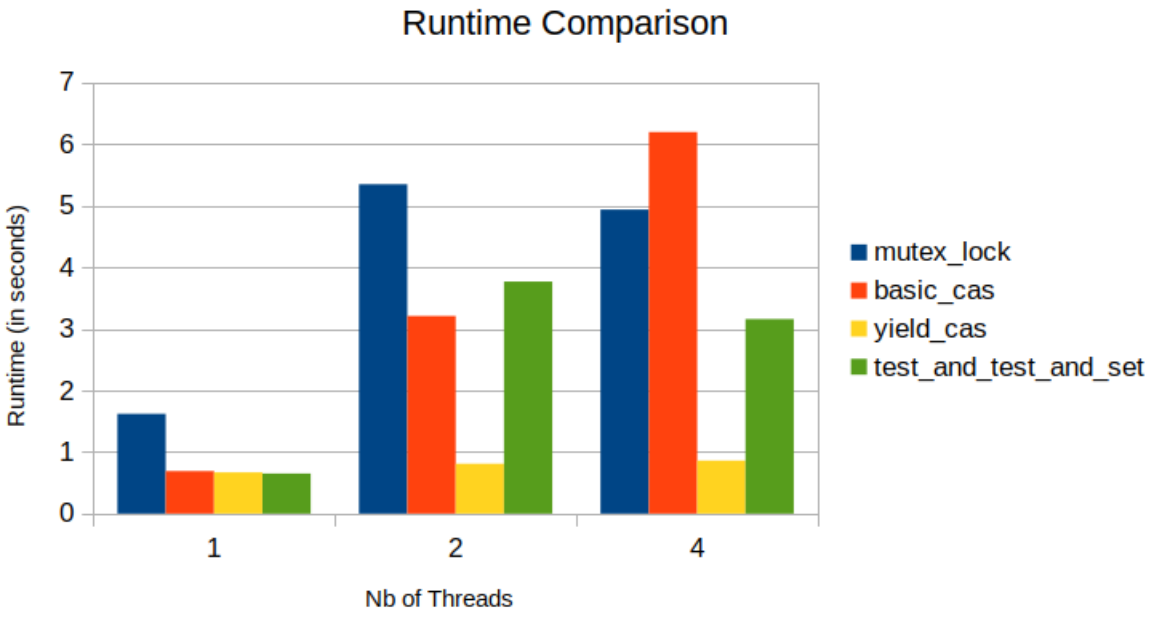
1.4 Test and test and set

This last implementation is a test and test and set lock. Its objective is to reduce the number of atomic operations compared to previous implementations. Its procedure is simple, we try to acquire the lock with a CAS, if we do not succeed we check the value of the lock until it changes, then we retry to acquire the lock with a CAS.

Nb of threads	Runtime	Nb of instructions	L1 data cache accesses	L1 dcache misses
1	0,65s	1 324 624 108	441 242 454	121 550
2	3,77s	4 981 871 508	1 365 046 754	116 194 182
4	3,16s	9 814 681 971	2 568 246 674	84 994 492

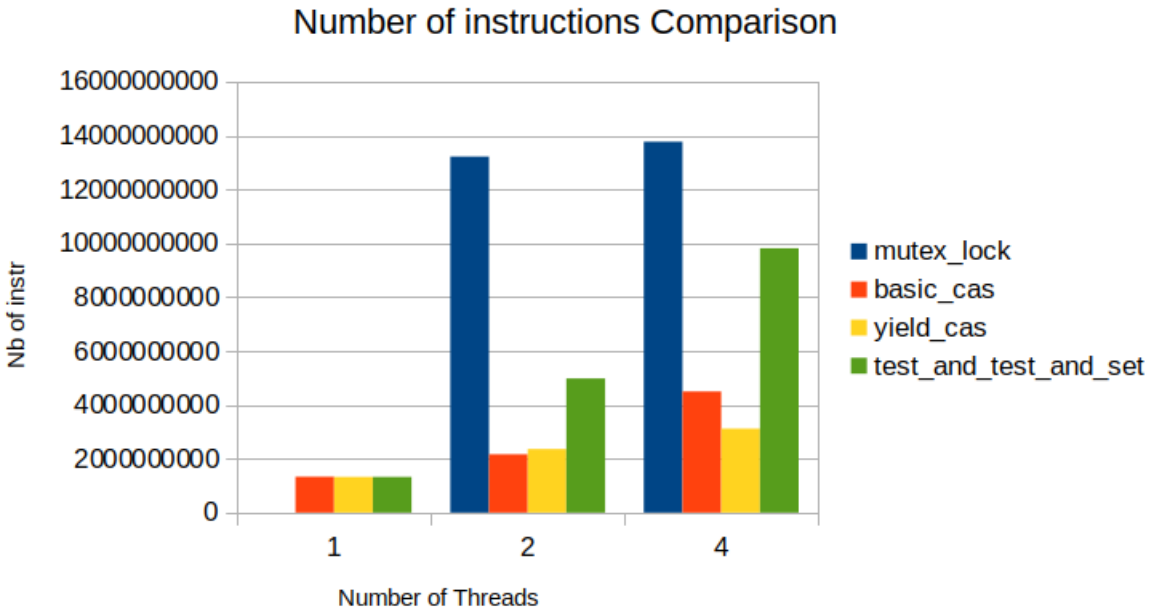
2 Comparison

2.1 Runtimes



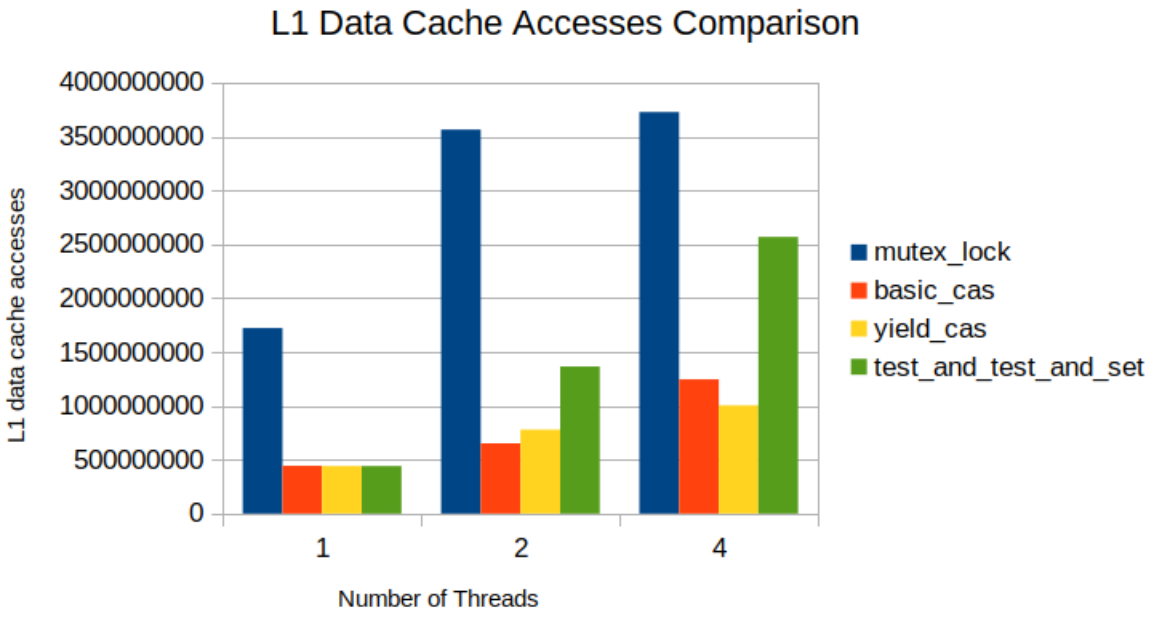
Concerning the runtime, it seems that the yield CAS outperforms all other implementations. We can see that the basic CAS gives weak results on a high number of threads, while the T&T&S seems to have better performance for a high number of threads.

2.2 Number of instructions



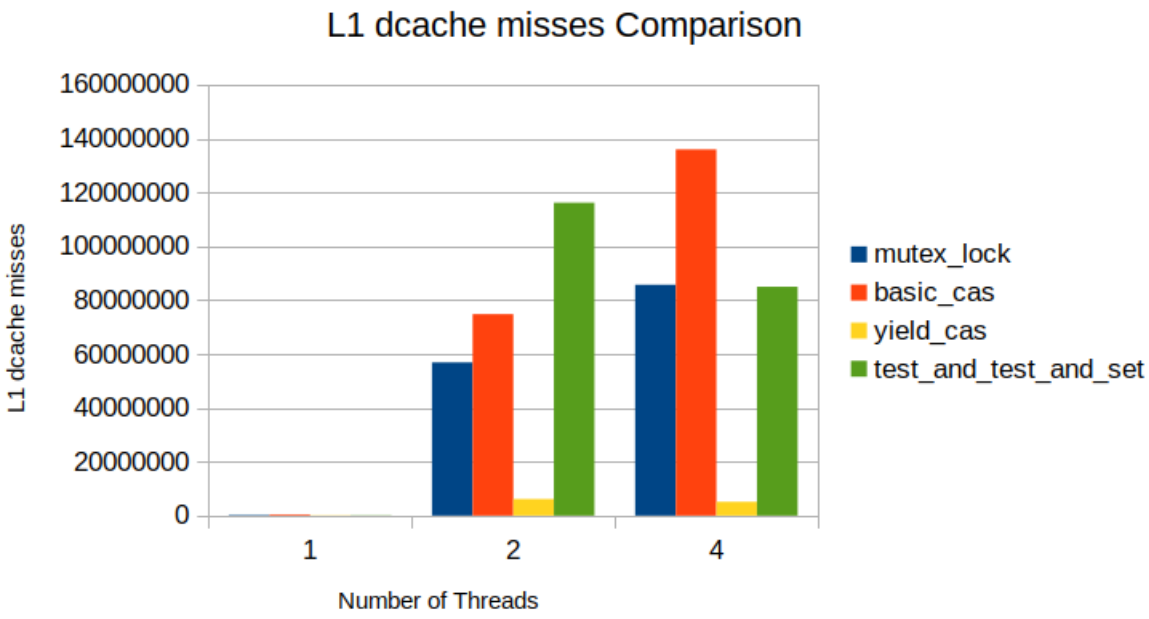
The number of instructions of the two versions of the CAS lock are quite similar for 1 and 2 threads, but for 4 threads, it is the yield CAS that has the lowest number of instructions.

2.3 L1 data cache accesses



In terms of data cache accesses, yield CAS seems more interesting than the others.

2.4 L1 dcache misses



According to the results, the yield CAS seems to outperform all the others in terms of cache misses. However, these values are so surprising that it could most likely be a measurement error. Indeed, given the previous measures, one would rather expect similar results between the basic CAS and the yield CAS. As for the T&T&S, we always notice that its performance improves with the increase in the number of threads.

2.5 Conclusions

After studying the results of the different implemented locks, we notice that for a small number of threads (in this case 1 or 2) the basic CAS lock and yield CAS lock have the best results. In return, for a larger number of threads (here 4), the basic CAS lock is not really viable, and we will rather choose the yield CAS lock which seems to present very good results. As for the T&T&S lock, we would have expected it to perform better for a large number of threads. Here the T&T&S lock results do not give great results. Maybe for an even higher number of locks we would see much better performance compared to other locks.