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).