Counter Locks Threads: 12 Iterations: 1000

Lock	TAS	TTAS	Ticket	MCS	Pthread lock
Execution Time	0.018796	0.017742	0.009961	0.015798	0.001679
L1-dcache-loads	3,530,099	3,516,798	2,978,157	3,829,059	2,080,529
L1-dcache-load-misses	162,020	150,982	148,182	148,753	95,706
L1-dcache miss rate	4.5%	5.06%	4.97%	3.88%	4.6%
L1-icache-load-misses	74,438	72,064	72,408	103,876	76,413
branch -instructions	1,829,784	2,080,014	1,785,903	2,622,142	1,085,735
branch-misses	22,418	44,059	33,368	28,910	24,847
page-faults	141	142	141	149	143

Counter Barrier Threads: 12

Barrier	Sense barrier	Pthread barrier	Sense barrier	Pthread barrier
Iterations	1000	1000	100	100
Execution Time	Took too long to execute	0.7667	1.4958	0.07875
L1-dcache-loads		256,976,796	243,410,515,394	53,494,187
L1-dcache-load-misses		29,173,470	8,610,982	3,032,355
L1-dcache miss rate		11.35%	0.0035%	5.66%
L1-icache-load-misses		19,662,156	20,109,676	1,708,876
branch -instructions		157,715,356	108,215,249,092	20,978,244
branch-misses		2,449,872	5,322,030	308,945
page-faults		146	148	144

Bucket Sorting
Threads: 12
File Size: 500 elements

Lock/ Barrier	TAS	TTAS	Ticket	MCS	Pthread lock
Execution Time	0.005328	0.163863	0.011111	0.033371	0.001319
L1-dcache-loads	3,510,315	232,103,785	17,672,801	55,169,633	1,749,772
L1-dcache-load-misses	114,172	172,307	115,964	210,201	116,942
L1-dcache miss rate	3.25%	0.074%	0.65%	0.38%	6.68%
L1-icache-load-misses	115,557	219,206	141,123	182,449	137,341
branch -instructions	3,310,217	167,955,165	12,553,649	26,881,998	1,260,048
branch-misses	27,226	49,807	1,942	30,791	31,474
page-faults	222	215	216	222	219

Every TAS attempt is likely a cache miss as it causes a coherence transition.

TTAS avoids contention misses by waiting for threads by leveraging multiple reads copies. TTAS has more cache loads because of this.

In ticket lock, all waiting threads have a cache miss every time lock is released.

MCS uses a queue for waiting threads and thus should have lower cache misses.

Sense reversal barrier flips sense during every iteration, and each thread its own local copy of sense to detect changes. Therefore it has a very high number of cache loads.

Merge Sort using multithreading
The algorithm divides the data set into equal-sized subsets. The number of subsets equals the number of threads used. The individual threads use the merge sort code to sort the data individually. After all the threads complete the sorting the data is merged back into the bigger array.

Bucket Sort using multithreading
The bucket sort algorithm uses presorted arrays, known as multiset in c++. Similar to the merge sort algorithm, the data is divided into equal sets, where the total number of sets equal the number of threads used. Each thread inserts the elements into buckets which are sorted. The sorted buckets can then be copied into the bigger array.

Compilation instructions

make: to compile both mysort.cpp and counter.cpp
Make clean: to clean the executables

Execution instructions

./mysort --name : prints name

./mysort <inputfile> -t <number of threads> -o <outputfile> --alg=<fj,bucket>
--bar=<sense,pthread> --lock=<tas,ttas,ticket,mcs,pthread>

./counter --name : prints name

./counter -t <number of threads> -i <number of iterations> -o <outfile> --bar=<sense,pthread>
--lock=<tas,ttas,ticket,mcs,pthread>