Parallel Algorithms Assignment #1

Participants:
Florian Groguelin
Bashar Qamhiyeh

1 - OpenMP Scheduling (Matrix Vector Product)

1 - Static Scheduling

#pragma omp parallel for schedule(static), private(j, r) to the outer loop of the sequential part.

2 - Dynamic Scheduling

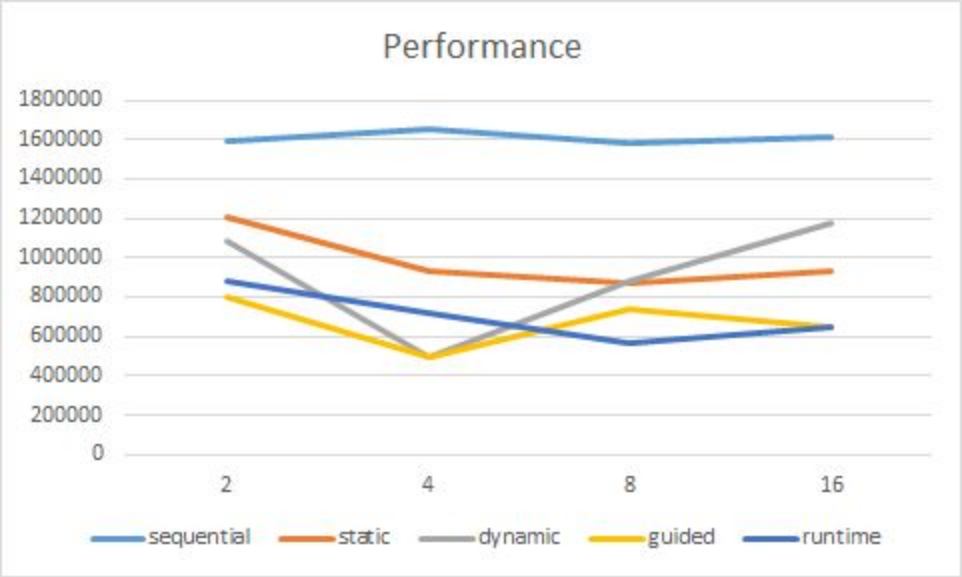
Same as the static scheduling but replace **static** key work with **dynamic**

3 - Guided Scheduling

Same as the static scheduling but replace **static** key work with **guided**

4 -

#threads	2	4	8	16
Sequential	1594793	1654391	1585065	1615392
static	1202986	932689	870522	930166
dynamic	1081271	499519	880466	1175633
guided	803904	491440	737401	645403
runtime	882134	716814	566884	645123



5 - We ran the process on a computer with 4 physical processors. The performance with parallelism is better in all cases. When the number of threads getting more than the number of physical cores the performance decreases. And the best performance on average for all scheduling algorithms is the best when number of threads equals 4

D - Runtime Scheduling

We added **chunk_size** and changed the scheduling algorithm to **runtime.** The table shows the result for different chunk_sizes with 5 tries for each

chunk_size	2	4	8	16	64	256
Exp1	2834454	2146083	1621461	1594858	2207638	1611446
Exp2	2213377	2414760	2406962	2501857	2518778	1583729
Exp3	2659782	1871556	1961748	2213970	1814997	2535164
Exp4	2524973	1613722	2628038	1915199	1643488	1824536
Exp5	2446594	2432964	1607263	1607923	1591477	1637239
Average	2535836	2095817	2045094	1966761	1955275	1838422

We notice that the performance is better for larger chunk size we benefit from the cache locality.

2 - Bubble sort:

For the Bubble we used **Dynamic** scheduling. The array was divided to number of chunks. The chunk size is (size / number_of_threads). Each thread sorts a chunk of the array to benefit from cache locality.

We also defined a limit for the chunk size. If the chunk size exceed the limit we do not calculate chunk size as (size / number_of_threads) but it becomes the limit and we have more chunks. The idea behind the limit is to fit the hole chunk in the cache.

Performance:

N	sequential	Parallel (dynamic)
4	2395	86425
8	173289	1464546
12	27753994	29460708
16	4084165190	7428302253

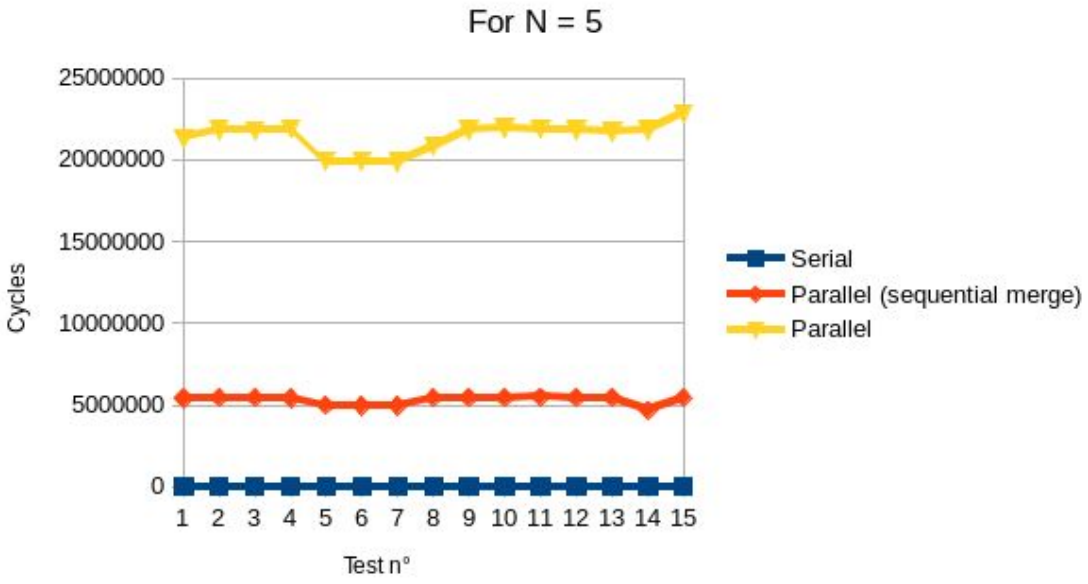
We notice form the table that small array size the performance is better for the sequential algorithm. That's because time used to divide the task between thread is more than the time needed to perform the sort. But for the large numbers the performance for parallelism was better.

3 - Quicksort:

For the parallel quicksort, we divide the array in some chunks and each thread applies the sequential quicksort to these chunks. After that, the threads merge the chunks two by two and we repeat the merge part until the array is sorted.

- Number of cycles for 2^5 elements:

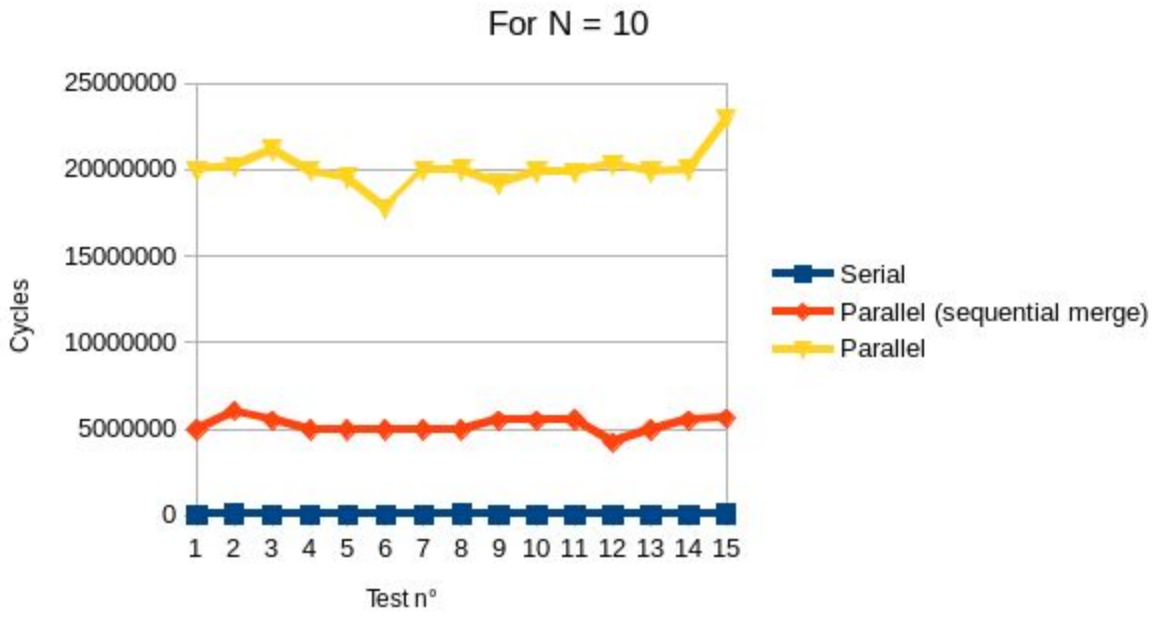
	Serial	Parallel (sequential merge)	Parallel
	1485	5432975	21330000
	1161	5457148	21840568
	1258	5448105	21781465
	1265	5385370	21889211
	1440	4977389	19903615
	1260	4960590	19920509
	1080	4949875	19882381
	1478	5449328	20821531
	1275	5451337	21813005
	1261	5453617	21929124
	1263	5485828	21873900
	1249	5444729	21826073
	1273	5435432	21709947
	1273	4690431	21852873
	1273	5455832	22824417
Average	1286	5298532	21413241



- Number of cycles for 2^{10} elements:

Serial	Parallel (sequential merge)	Parallel
55664	4935742	19940212
67031	6025807	20164678
55710	5491705	21164855
54912	4942941	19920282
55708	4943070	19540707
55750	4939340	17742099
48920	4942941	19948334
64821	4950102	20006514
55438	5483497	19189367
48078	5473143	19868739
56139	5507384	19835801
55706	4222978	20290491
56131	4954609	19882678

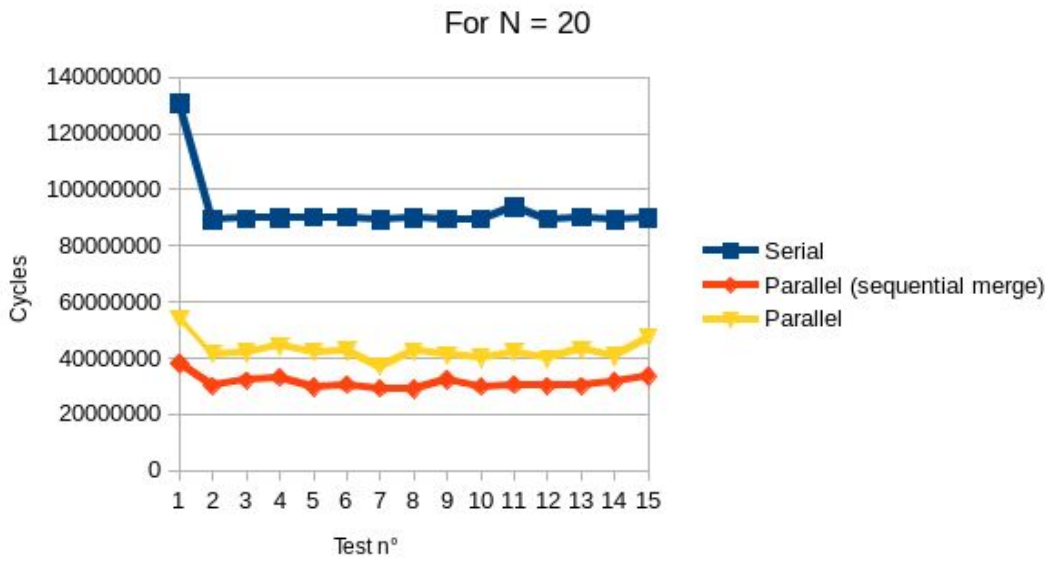
	48460	5483093	19986230
	66818	5594881	22930680
Average	56352	5192748	20027444



- Number of cycles for 2^{20} elements:

Serial	Parallel (sequential merge)	Parallel
130841733	38157537	54084267
89387033	30029348	41581166
89739323	31959875	42098444
89836666	33102614	44336236
90031535	29713399	42212276
90128051	30449265	42684118
89363595	29221579	37072877
89906027	28960823	42299346

	89416780	32229823	41003242
	89529092	29901942	39955974
	93864696	30547820	42236517
	89539239	30097432	39729478
	90040372	30011090	43007994
	89226375	31551732	40666891
	89876612	33583576	47386507
Average	92715141	31301190	42690355



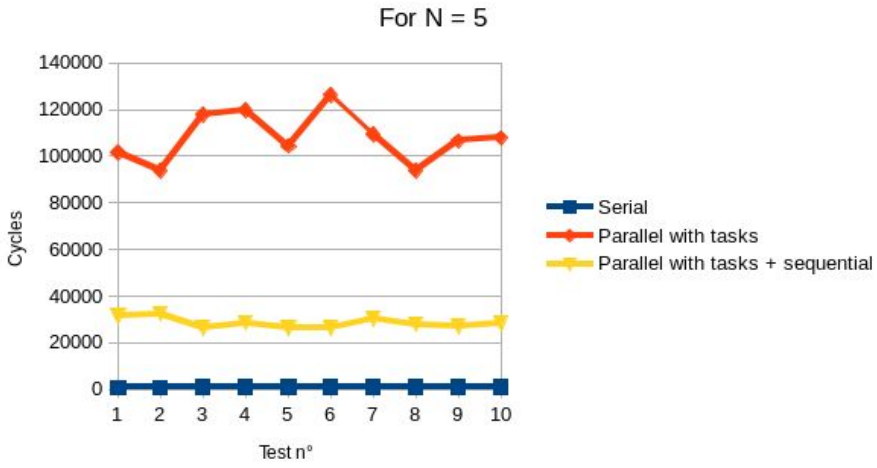
As the graphs show, the serial quicksort is really efficient when the array to sort is composed of few elements. More the number of elements increases, more the parallel sort is better. For the third example, the parallel sort with the sequential merge has less cycles the full parallel one but if the length of array increases a lot the full parallel sort will be the fastest.

4 - Merge sort:

To program the merge sort we implemented it recursively, indeed as we divide by two the array at each step and we merge the two chunks at the end so a recursive version is natural. For our third version, we mixed the parallel merge sort with taks and the sequential sort.

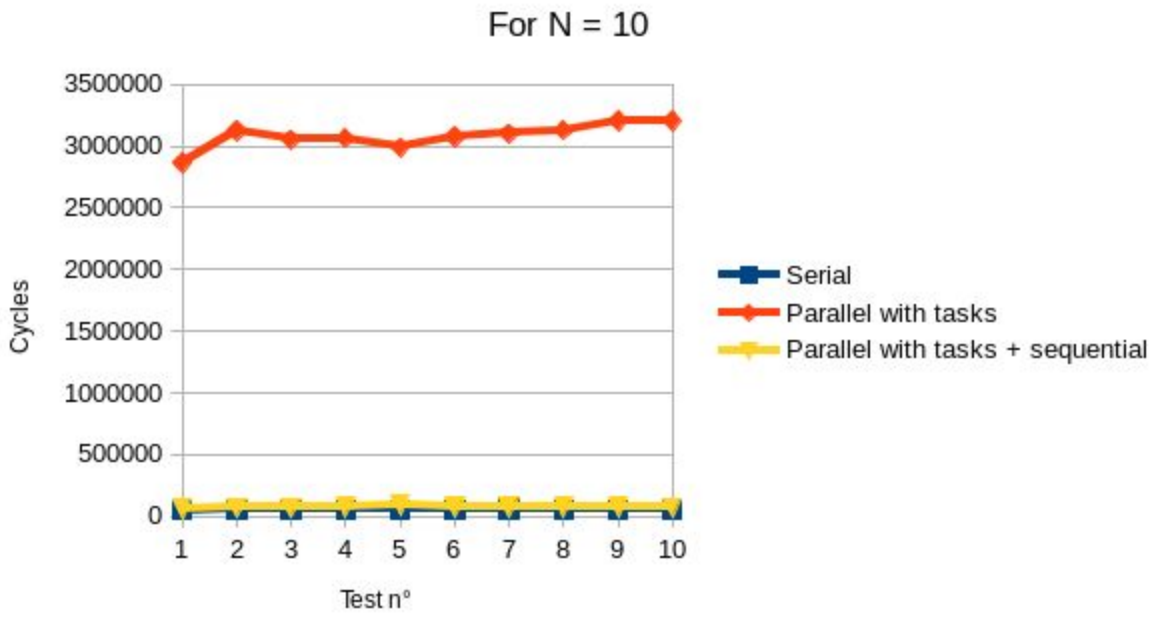
- Number of cycles for 2^5 elements:

	Serial	Parallel with tasks	Parallel with tasks + sequential
	949	101537	31468
	951	93841	32529
	1040	117734	26419
	1041	119648	28658
	1039	104446	26554
	1040	126038	26445
	1040	109333	30281
	1041	93892	27859
	1041	106465	26924
	1041	107820	28496
Average	1022	108075	28563



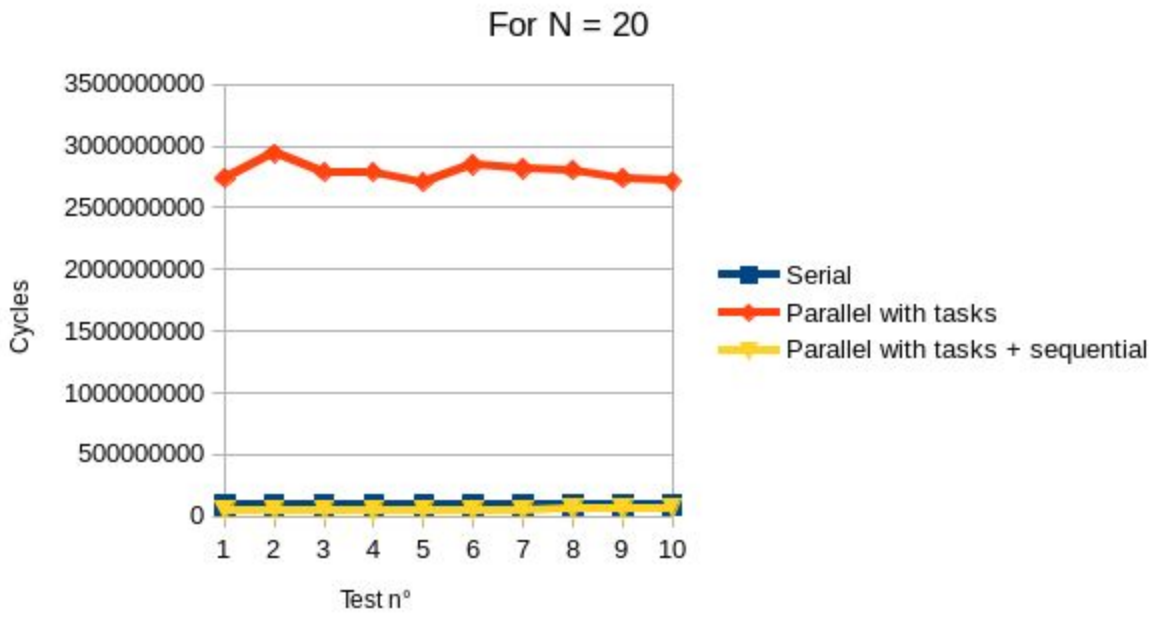
- Number of cycles for 2^{10} elements:

	Serial	Parallel with tasks	Parallel with tasks + sequential
	48684	2863218	64189
	53514	3125362	70422
	53552	3051064	69229
	53548	3062470	75040
	53604	2986329	94103
	53542	3074940	76907
	63105	3101710	77826
	53518	3129290	77437
	53642	3205394	76133
	59300	3200039	70638
Average	54600	3079981	75192



- Number of cycles for 2^{20} elements:

	Serial	Parallel with tasks	Parallel with tasks + sequential
	96409937	2736978380	48824492
	96389320	2939137681	51067256
	96387957	2788734349	51717559
	96291823	2786573007	45632706
	96546840	2708624587	50841000
	96651588	2848650133	51829156
	96320032	2814401475	52211453
	96318352	2801972154	60461356
	96275273	2739469031	57881420
	96513894	2711664606	68035648
Average	96410501	2787620540	53850204



Such as the quicksort, the sequential version is really fast when the array is not big. Nevertheless the parallel version is very slow and more the length of the array is high more the number of cycles increases. That can be explained by number of tasks, in

the parallel version the number of tasks is $2log^2(N)$. However, the third version is almost twice faster than the sequential merge sort for huge array.