CS4532: Concurrent Programming

Lab 4 and 5

120337H - Linganesan

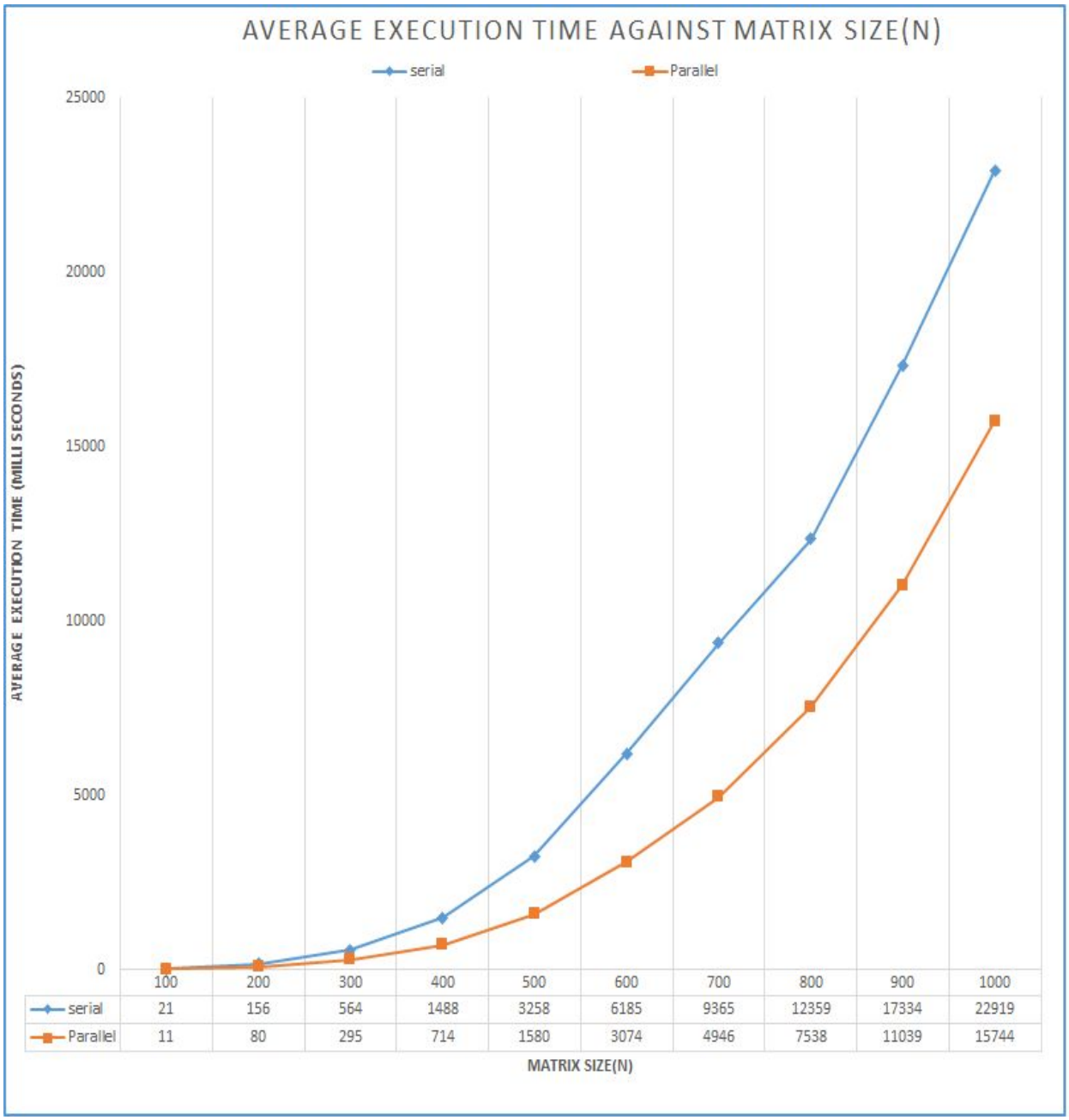
120678D - M.V.Vithulan

1. lab4_openmp.cpp file included in the folder.
2. The OpenMP
   The OpenMP library is an API (Application Programming Interface) for writing shared memory parallel applications in programming languages such as C, C++ and FORTRAN. This API consists of compiler directives, runtime routines, and Environmental variables. Some the advantages of OpenMP includes: good performance, portable (it is supported by a large number of compilers), requires very little programming effort and allows the program to be parallelized incrementally.
3. lab4_openmp.cpp file included in the folder.
4.

We have used 385 samples in order to achieve accuracy +/-5% and 95% confidence level. Then we obtained the mean of 385 samples and tabled them for each matrix size.

Graph of matrix-matrix multiplication time against increasing matrix size(n) - Serial process

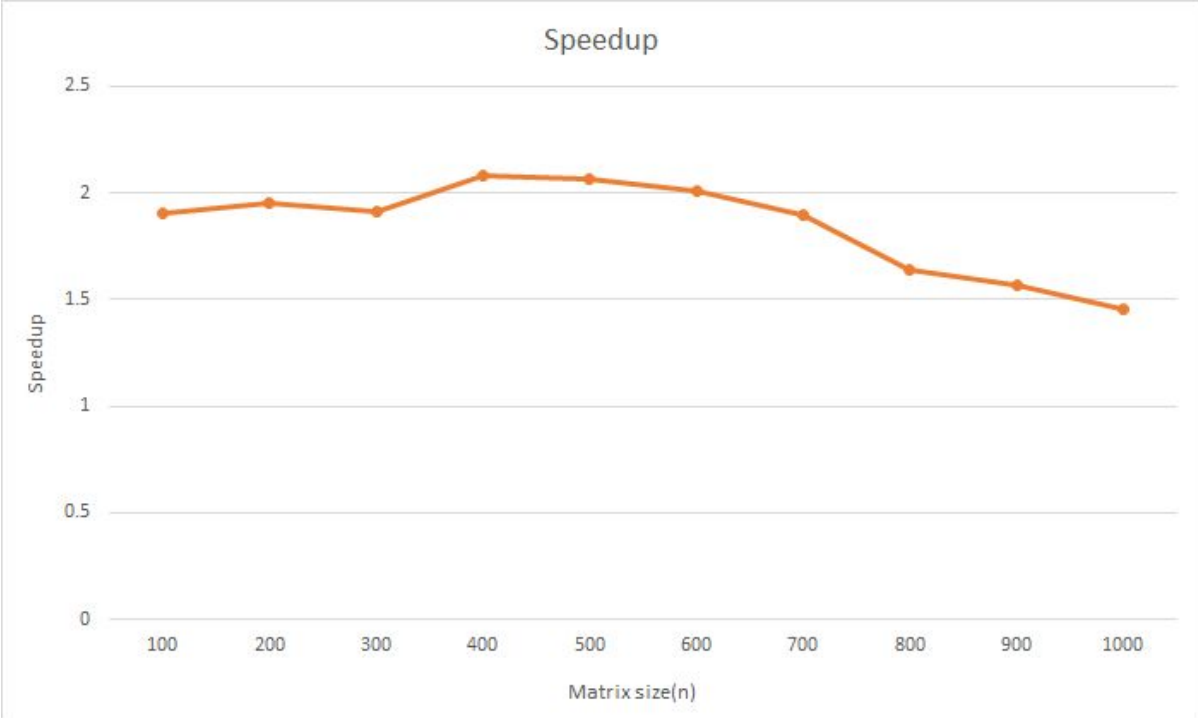
Matrix size(n)	100	200	300	400	500	600	700	800	900	1000
Serial	21	156	564	1488	3258	6185	9365	12359	17334	22919
Parallel	11	80	295	714	1580	3074	4946	7538	11039	15744



Speedup (Serial/Parallel against array size)

Speedup = Average serial execution time/Average parallel execution time

Matrix size(n)	100	200	300	400	500	600	700	800	900	1000
Speedup	1.91	1.95	1.91	2.08	2.06	2.01	1.89	1.64	1.57	1.46



5. The **machine specification** that used to run our simulations

- Fedora 24x64 bit
- Memory: 8 GB, 1600 MHz
- CPU: Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz - 2 cores, 4 HT [1]
- L1d cache:32K
- L1i cache:32K
- L2 cache:256K
- L3 cache:3072K

We used 2 core computer for these calculations, so we expected not more than 2 times speedup for increasing array size. But we observed the speedup goes upto maximum 2.08 times and then decreases. There could be few reasons for slowing down the speedup such as,

Read shared variables without flush
When reading a shared variable without flushing it first, it is not guaranteed to be up to date. Actually, the problem is even more complicated, as not only the reading thread has to flush the variable, but also any thread writing to it beforehand.

Use of unnecessary critical
We have used transpose function() within the critical region, which is unnecessary. Thereby it potentially blocking other threads longer than needed and probably paying the maintenance cost associated with such regions more often than needed.

6. **Optimization techniques**.

Removing unnecessary code and computations within the critical region [2]
We have used transpose function() within the critical region, which is unnecessary. Thereby it potentially blocking other threads longer than needed and probably paying the maintenance cost associated with such regions more often than needed. Thus doing transpose function outside the critical region can potentially increase the speed.

Using dynamic scheduling for 'for loop' [3]
Assignment can vary at run-time, and iterations are handed out to threads as they complete previously assigned iterations. Hence changing the scheduling dynamically will increase the speed, where as the schedule will remain same from 100 to 1000 in static scheduling.

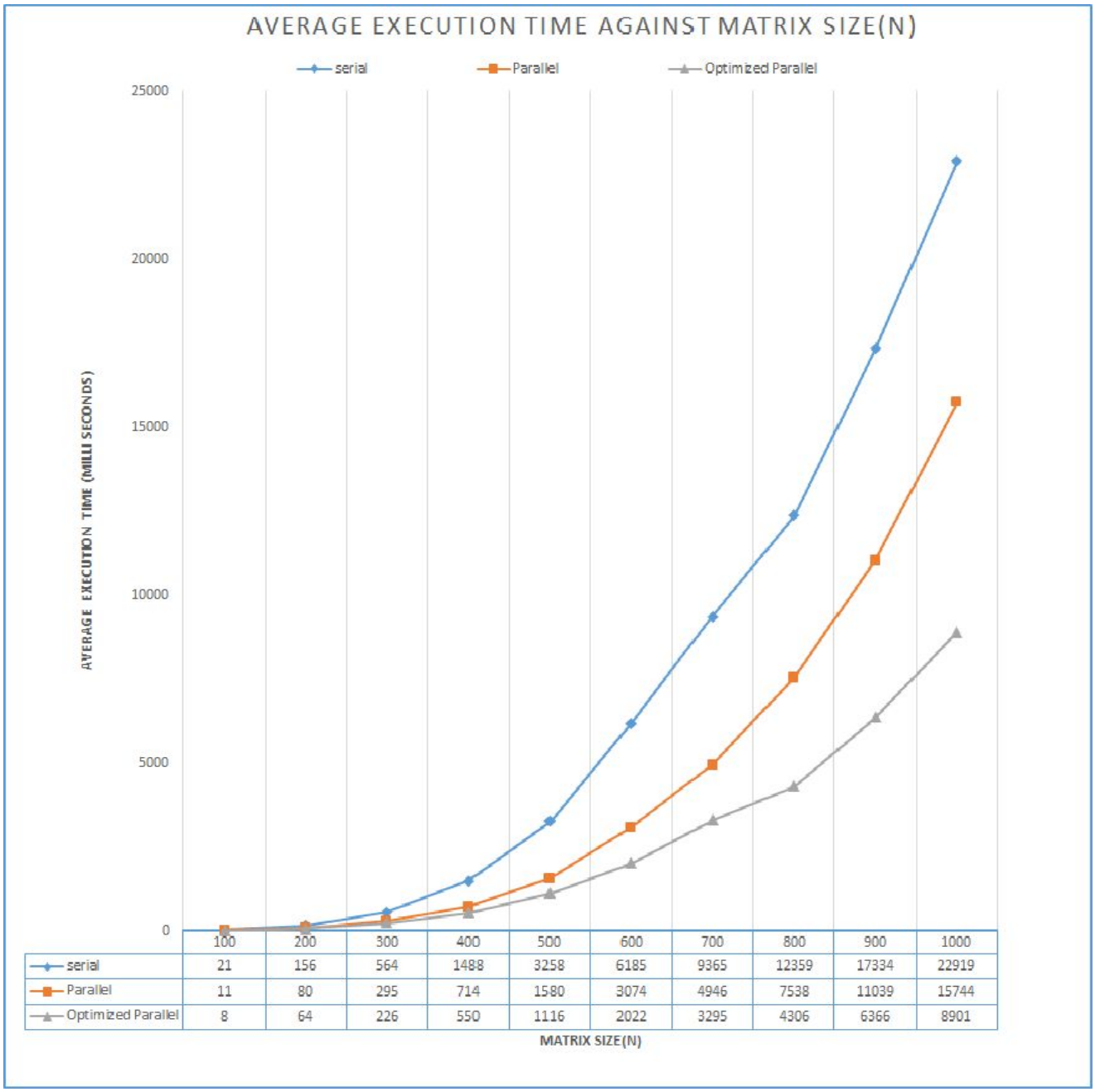
Read shared variables without flush [2]
When reading a shared variable without flushing it first, it is not guaranteed to be up to date. Actually, the problem is even more complicated, as not only the reading thread has to flush the variable, but also any thread writing to it beforehand. Therefore if we flush the shared variables we can achieve some amount of speedup.

7. lab4_openmp.cpp file included in the folder.
8.

We have used 385 samples in order to achieve accuracy +/-5% and 95% confidence level. Then we obtained the mean of 385 samples and tabled them for each matrix size.

Graph of matrix-matrix multiplication time against increasing matrix size(n) - Parallel processing

Matrix size(n)	100	200	300	400	500	600	700	800	900	1000
Serial (ms)	21	156	564	1488	3258	6185	9365	12359	17334	22919
Parallel (ms)	11	80	295	714	1580	3074	4946	7538	11039	15744
Optimized Parallel (ms)	8	64	226	550	1116	2022	3295	4306	6366	8901

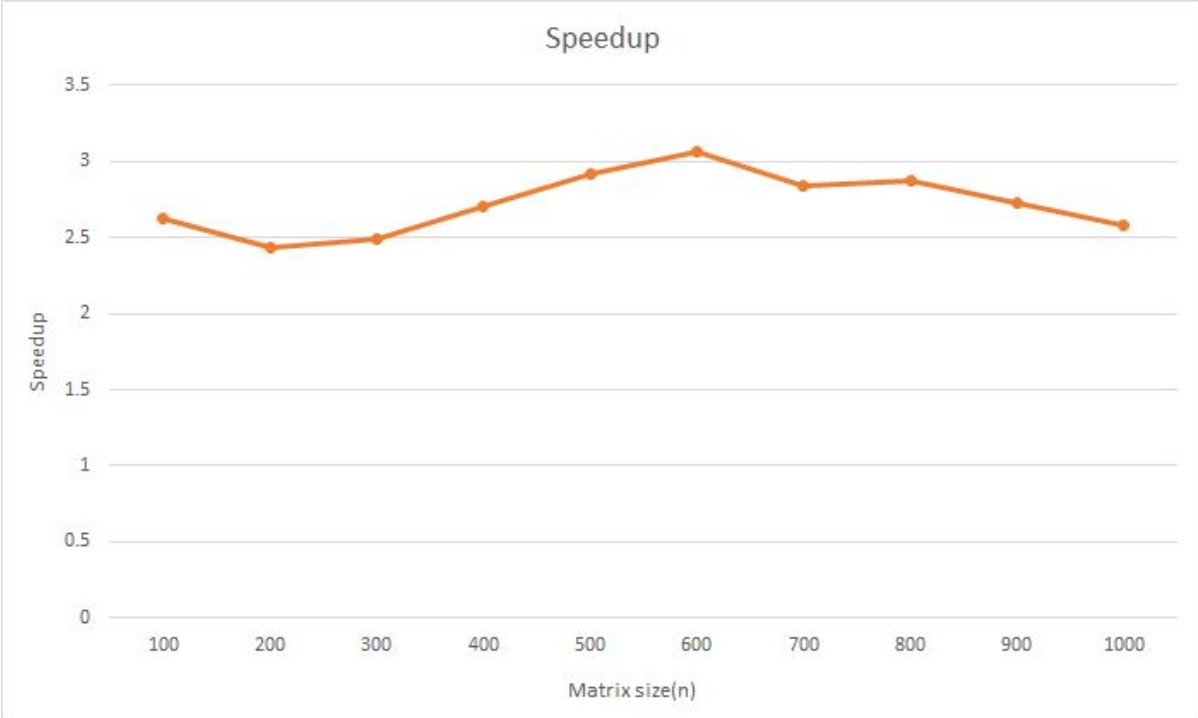


	100	200	300	400	500	600	700	800	900	1000
serial	21	156	564	1488	3258	6185	9365	12359	17334	22919
Parallel	11	80	295	714	1580	3074	4946	7538	11039	15744
Optimized Parallel	8	64	226	550	1116	2022	3295	4306	6366	8901

Speedup (Serial/Optimized Parallel against array size)

Speedup =Average serial execution time/Average optimized parallel execution time

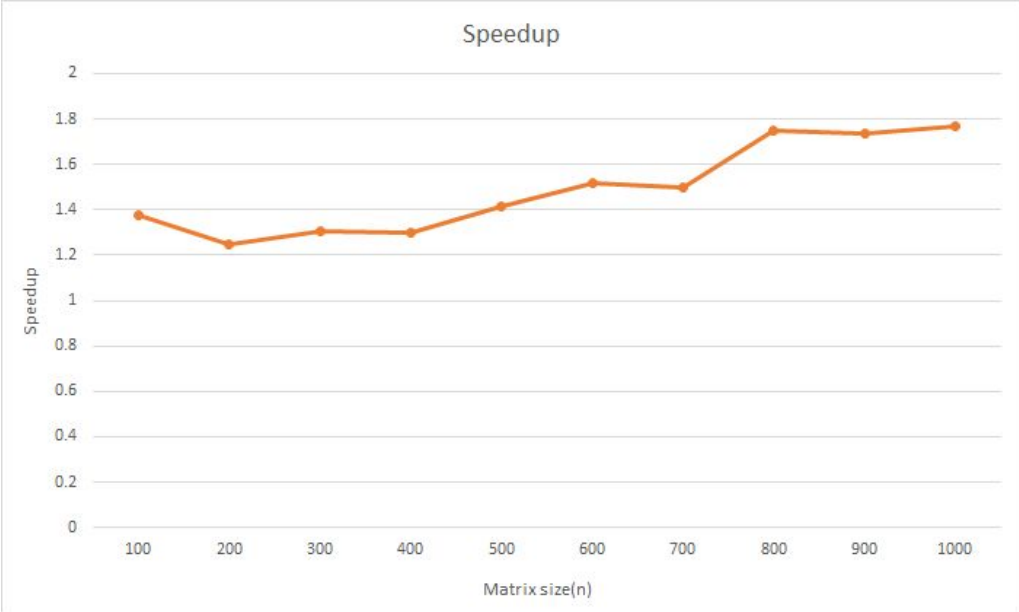
Matrix size(n)	100	200	300	400	500	600	700	800	900	1000
Speedup	2.63	2.44	2.50	2.71	2.92	3.06	2.84	2.87	2.72	2.57



Speedup (Parallel/Optimized Parallel against array size)

Speedup =Average previous parallel execution time/Average optimized parallel execution time

Matrix size(n)	100	200	300	400	500	600	700	800	900	1000
Speedup	1.38	1.25	1.31	1.30	1.42	1.52	1.50	1.75	1.73	1.77



9. We did the calculations in 2 core, 4 HT computer, where we expected speedup around 2 times. In the Non-Optimized solution we achieved at most 2 times speedup and in most cases it tend to be between 1.6 - 1.9, average of 1.84 times. After applying optimization techniques as explained in task 6, we have achieved at most 3.06 times speed up and average of 2.726 times. Earlier Hyper threads didn't add much effect in speedup but after optimizations hyperthreads have also added their contributions for the speedup. That's probably because of we removed unwanted codes from the critical region, hence, the waiting time for the threads should have been reduced.

References

[1] http://ark.intel.com/products/85214/Intel-Core-i7-5500U-Processor-4M-Cache-up-to-3_00-GHz?q=Intel%C2%AE%20Core%E2%84%A2%20i7-5500U%20Processor%20(4M%20Cache,%20up%20to%203.00%20GHz)

[2] https://pdfs.semanticscholar.org/4e60/37127e85445079272e1cb5574cbcce2e175e.pdf

[3] https://www.buffalo.edu/content/www/ccr/support/training-resources/tutorials/advanced-topics--e-g--mpi--gpgpu--openmp--etc--/2011-09---practical-issues-in-openmp--hpc-1-/_jcr_content/par/download/file.res/omp-II-handout-2x2.pdf