Assignment 3

Introduction to Parallel Programming

Venkata Sriram Krishna Prasanth Kondapi

Pranav Vijayachandran Nair

Make files are executed (make -f <makeFileName>)

Exercise 1

The program *sievePrimesOMP.cpp* executed using *'makePrimesOMP'* makefile. (Command: make -f makePrimesOMP)

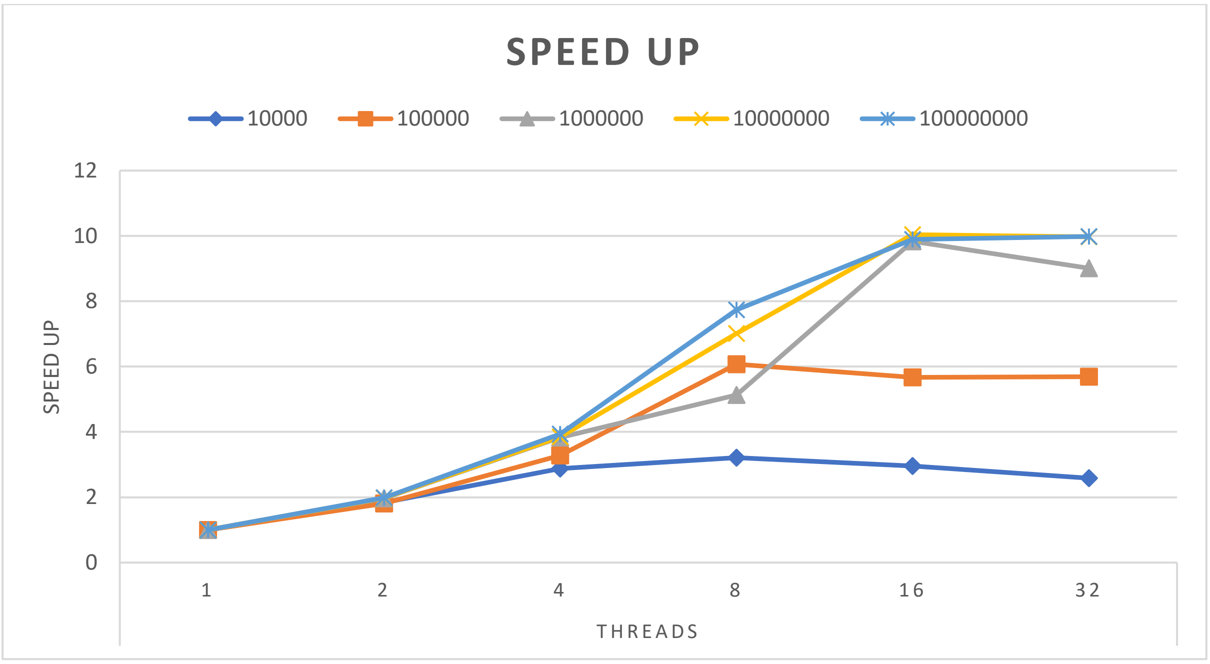


Chart 1: Speed Up in Calculating primes with OpenMP

This algorithm uses vectors to store prime numbers, just like the previous one. Two changes are made to the previous algorithm to implement Sieve of Eratosthenes using OpenMP.

First change is about distribution of work. After calculating seeds, posix algorithm splits the remaining numbers into almost equal number of pieces for each thread to sieve and inside each thread two for loops are used to run through all numbers in the piece and to check if they are divisible by seeds.

But in the OpenMP algorithm, threads are created using "parallel for" directives and a custom 'Reduction' declaration to insert vectors of each thread. OpenMP automatically distributes the remaining numbers in for loop among threads.

Second change is about sieving primes in each thread. Previously, as the threads know the starting point and ending point. A vector is created with all numbers in the piece and all divisible numbers are removed to get the final result.

By in OpenMP algorithm, previous method cannot be used as each thread where it starts and ends. Alternatively, 'parallel for' is used to loop through all remaining numbers and they are added to the vector if they are not divisible.

These two changes made the implementation smaller and faster. Almost 14 times while running on 32 threads and nearly 200 times on a single thread for calculating upto 100,000. Hence, we are able to push the algorithm too check upto 100 million numbers in just 36 seconds while running on 32 threads.

As observed previously the speedup plateaus after 16 threads.

	10000	**100000**	**1000000**	**10000000**	**100000000**
1	0.003752	0.046592481	0.803228	16.02593	364.3747375
2	0.002042	0.025792568	0.407582	8.116414	183.6410216
4	0.001305	0.014199817	0.209613	4.159523	92.51859787
8	0.001168	0.00767044	0.156434	2.285723	47.11709952
16	0.001268	0.008213457	0.081681	1.596184	36.84523848
32	0.001453	0.008194829	0.089154	1.607102	36.49539214

Table 1: Calculation time (Sec) ; Threads vs Maximum Number

	10000	**100000**	**1000000**	**10000000**	**100000000**
1	1	1	1	1	1
2	1.837688	1.806430479	1.970713	1.974508	1.984168538
4	2.875171	3.28120292	3.831955	3.852828	3.938394505
8	3.212239	6.074290513	5.134624	7.011315	7.733386417
16	2.959872	5.672700415	9.833747	10.04015	9.889330413
32	2.581908	5.685595273	9.009479	9.971944	9.984129943

Table 2: Speed Up: Threads vs Max Number

Exercise 2

The program *GameOfLifeOMP.c* executed using *'makePrimesOMP'* makefile. (Command: make -f makePrimesOMP)

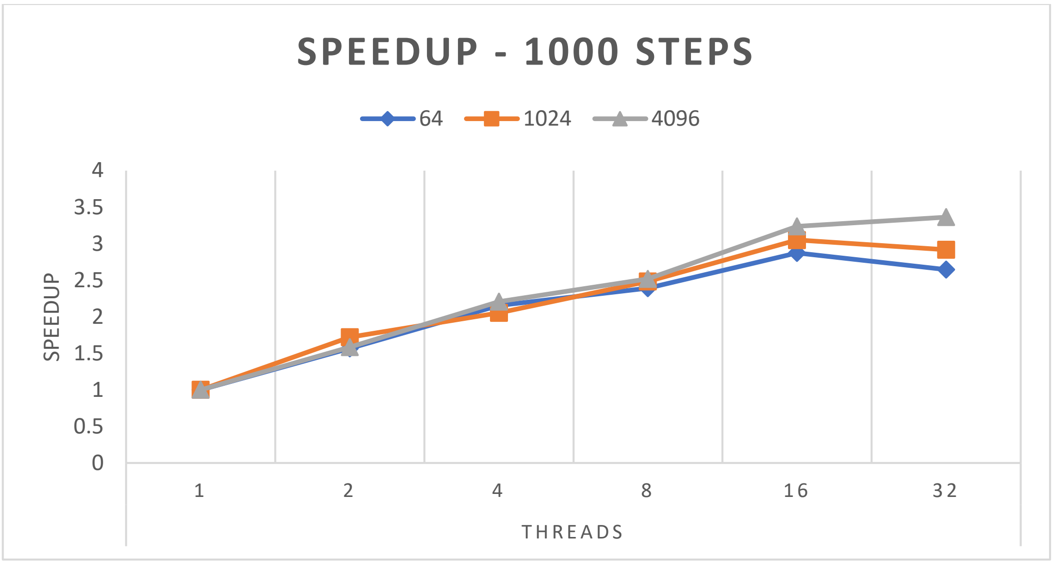


Chart 2: Speed Up for 1000 steps in Game of Life using OpenMP

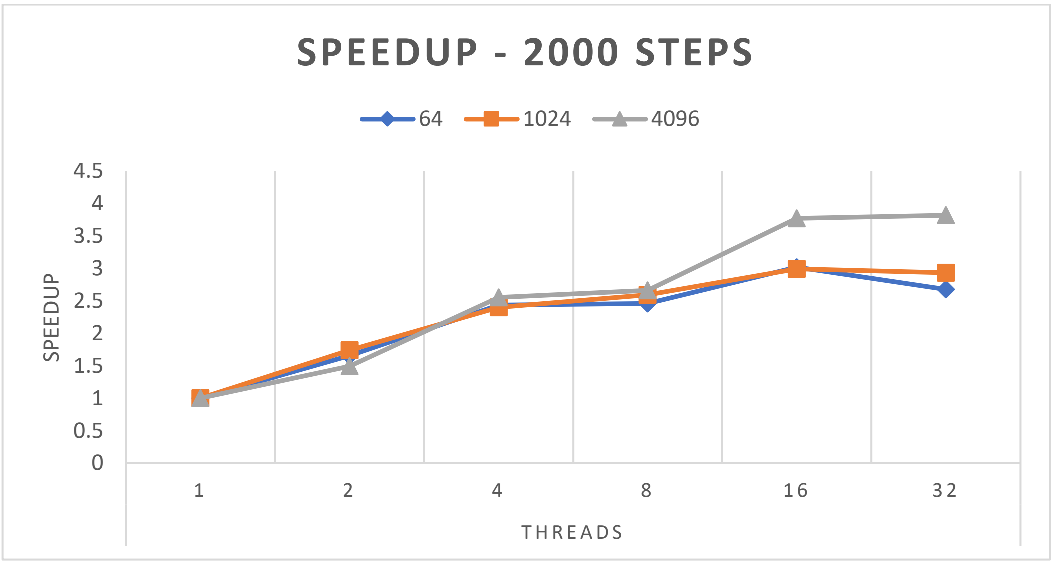


Chart 3: Speed Up for 2000 steps in Game of Life using OpenMP

Parallelising the provided algorithm using OpenMP is very simpler and easy to implement compared to the Sieve of Eratosthenes. The only major change is to add the #pragma directives to run the for loops in parallel. As the calculations are based on a two-dimensional array “collapse” directive is used to distribute both for loops over various threads. “Reduction” directive is not used as each thread is working on different array elements.

Speed up curve isn’t much different 1000 steps and 2000 steps as all threads have to wait for each other at the end of every step.

	1000			**2000**		
	64	**1024**	**4096**	**64**	**1024**	**4096**
1	0.106417	27.90065	449.2438	0.210419	55.63002	989.1245
2	0.06794	16.22757	283.583	0.127349	32.02029	664.0186
4	0.049417	13.58783	203.3348	0.086399	23.19405	387.4058
8	0.044489	11.24351	178.2687	0.085612	21.45188	371.6312

16	0.03703	9.144284	138.7946	0.069729	18.5939	262.3631
32	0.040195	9.560541	133.5838	0.078686	18.96744	259.0917

Chart 4: Calculation time (sec) - Game of Life: Threads Vs Size and time

		1000			**2000**	
	64	**1024**	**4096**	**64**	**1024**	**4096**
1	1	1	1	1	1	1
2	1.566338	1.719336	1.58417	1.652302	1.737337	1.489604
4	2.153449	2.053356	2.20938	2.435433	2.39846	2.5532
8	2.391985	2.48149	2.520037	2.457821	2.593247	2.661575
16	2.873805	3.051158	3.236753	3.017668	2.991843	3.77006
32	2.647518	2.918313	3.363011	2.674161	2.932922	3.817662

Chart 5: Speed Up - Game of Life: Threads Vs Size and time

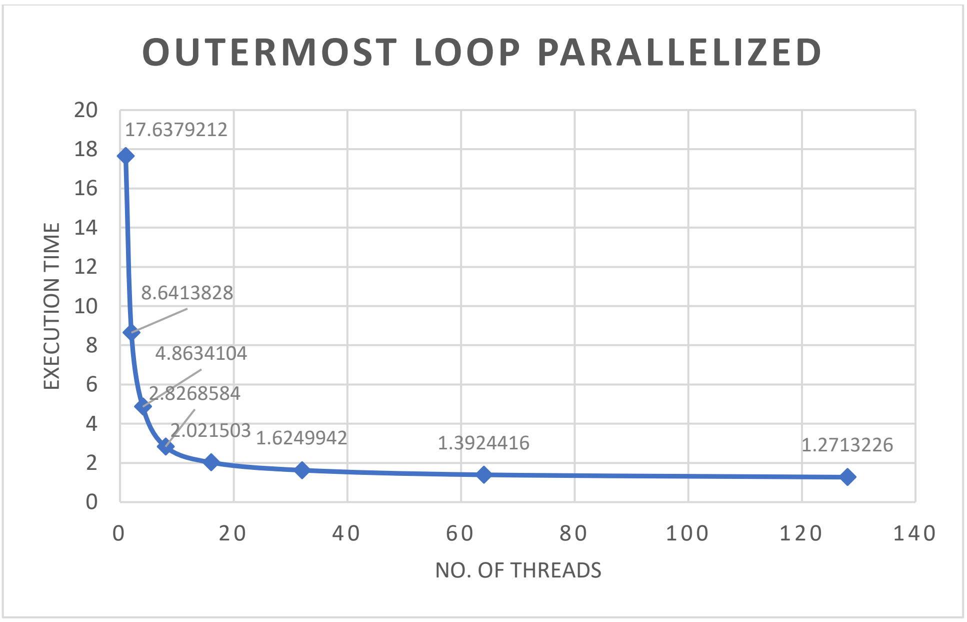
Exercise 3

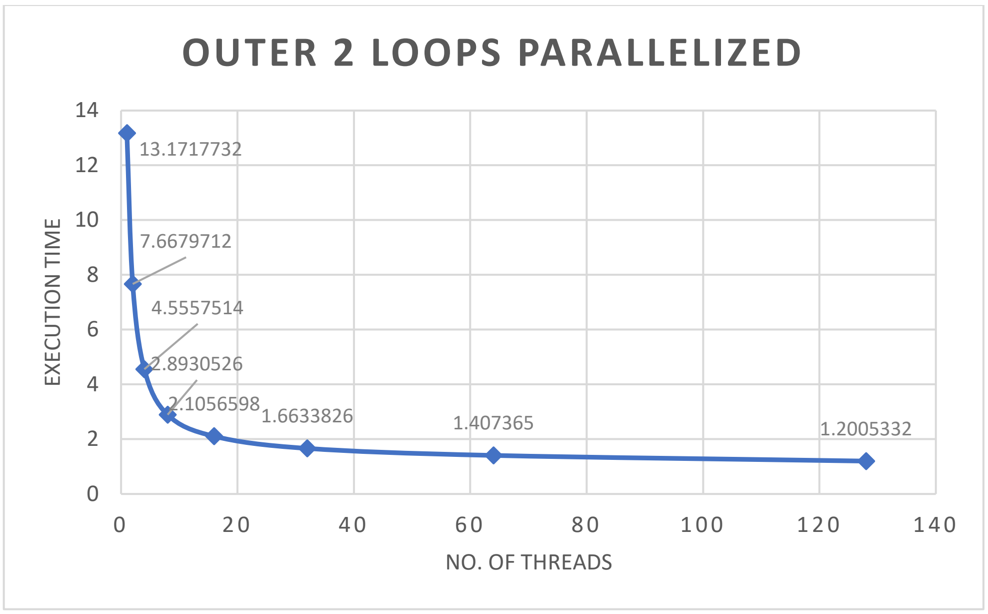
Compile commands:

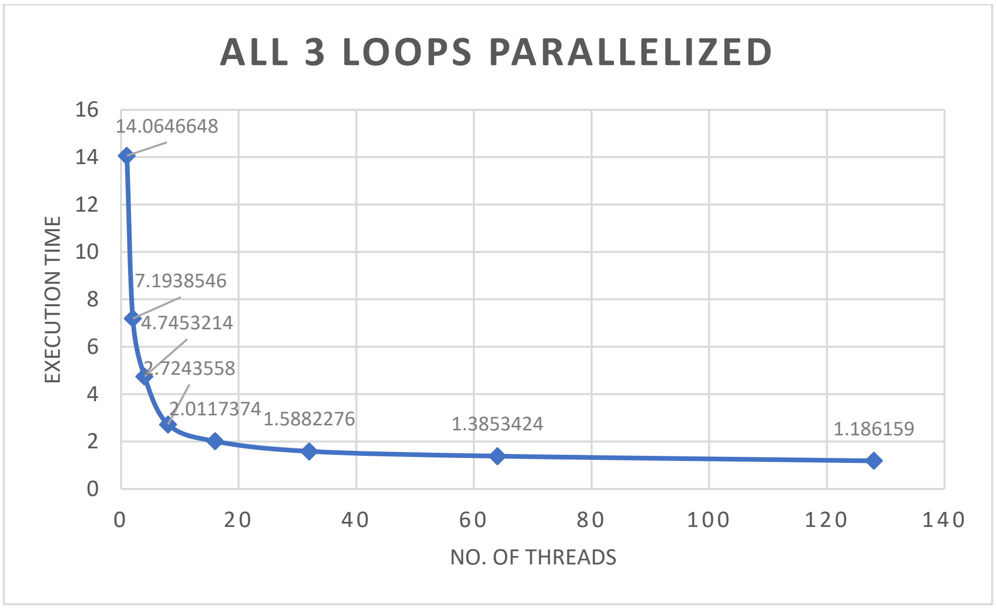
g++ -fopenmp matrix_new.cpp -o matrix_new {outer loop]

g++ -fopenmp matrix_new_2for.cpp -o matrix_new_2for[2loops]

g++ -fopenmp matrix_new_3for.cpp -o matrix_new_3for{all 3 loops]







The matrix-matrix product code given in the lecture was implemented and tested for thread numbers ranging from 1 to 2000. The average execution times of this range is considered for all the 3 cases with only the outermost loop parallelized, outer 2 loops parallelized, and all the 3 loops parallelized. Clearly, we can understand that as the number of threads increases, the execution time decreases, thereby improving speed-up. But it is not significant for more than 32 threads.

Parallelization of the inner loops were achieved using the ‘collapse’ function for better thread utilization and not opting the ‘nested for’ approach which proved to be more performance critical.

The problem with parallelizing arises when all the 3 loops are parallelized. When the innermost loop becomes parallelized, addition to a value for every c[i][j] happens due to different threads parallelly. This leads to a data race and can avoided by making it an atomic instruction. Adding this instruction, creates a sequential updating of values by the threads and therefore should have increased execution time.

But the above 3-loops parallelized execution performed better than the 2-loop parallelization, even with the atomic instruction overhead. It also provided better speed-up than just the outer loop parallelized because of more threads executing parallelly.

Exercise 4

```
g++ -fopenmp Gaussian_row.cpp -o Gaussian_row [Row-oriented]
```

```
g++ -fopenmp Gaussian_column.cpp -o Gaussian_column [Column-oriented]
```

Row-oriented:

Outer Loop cannot be parallelised. Inner loop can be parallelised. Data dependency can be avoided by parallelising the inner loop.

Column-oriented:

(Second) Outer Loop cannot be parallelised. Inner loop can be parallelised. Data dependency can be avoided by parallelising the inner loop.

Column oriented is observed to be slower than row-oriented.