Name: Rohit Pujar Nagraj
USC ID: 4664 4939 18

Programming Homework 3 - Report

1. Parallel Matrix Multiplication

1.1 Naive MatMul with OpenMP

No. of Threads	Execution Time	Performance
1	1366.138831 sec	100.603943 MFLOPS
4	375.451539 sec	366.063098 MFLOPS
16	394.123210 sec	348.720781 MFLOPS
64	390.727256 sec	351.751641 MFLOPS
256	386.786432 sec	355.335508 MFLOPS

1.2 Block MatMul with OpenMP

No. of Threads	Execution Time	Performance
1	319.371676 sec	430.341711 MFLOPS
4	101.459692 sec	1354.616314 MFLOPS
16	95.469828 sec	1439.606168 MFLOPS
64	101.700691 sec	1351.406295 MFLOPS
256	90.865101 sec	1512.560400 MFLOPS

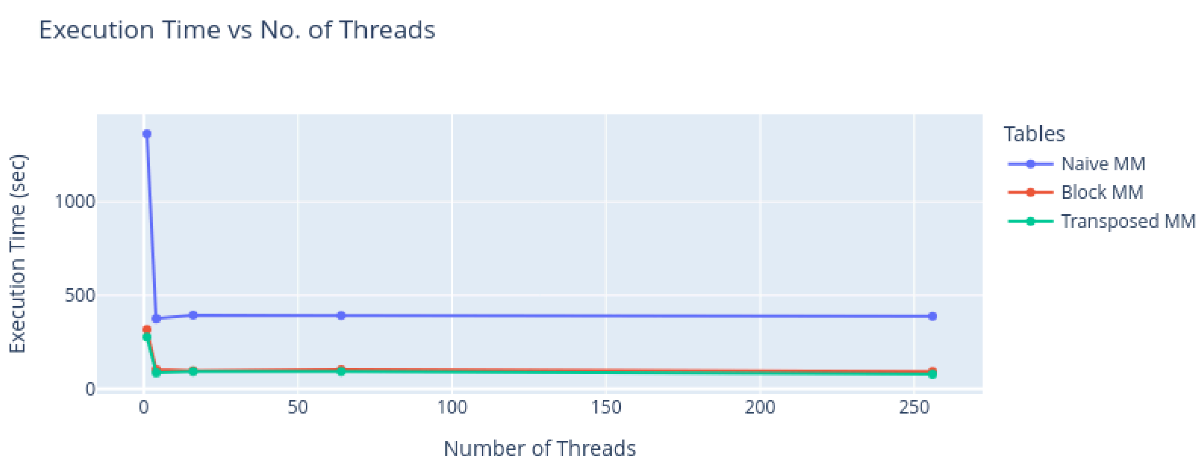
Name: Rohit Pujar Nagraj
USC ID: 4664 4939 18

1.3 Transposed MatMul with OpenMP

No. of Threads	Execution Time	Performance
1	277.867787 sec	494.619960 MFLOPS
4	86.697244 sec	1585.274763 MFLOPS
16	91.441576 sec	1503.024768 MFLOPS
64	91.454943 sec	502.805084 MFLOPS
256	77.599605 sec	1771.129540 MFLOPS

Sample Output

```
Number of FLOPs = 137438953472, Execution time = 390.727256 sec,
351.751641 MFLOPs per sec
C[100][100]=879616000.000000
```



Observations:

1. The execution time of naive matrix multiplication reduces with the number of threads as expected. However, it plateaus quickly because it is being run on a 4-core machine, and there's not much room left to improve performance with more threads.

2. The execution times for Block and Transposed Matrix Multiplication are much lower compared to the Naive Matrix Multiplication because both block and transposed Matrix Multiplication are cache-optimized methods, wherein the next element in the loop is usually found in the cache. In contrast, the naive matrix multiplication iterates column-wise on a row-major matrix.

2. Estimating Pi

Serial time: 0.243284 sec

Version a: DO/for OpenMP parallel with 4 threads: 0.100009 sec

Version b: 2 Threads with SECTIONS: 0.122512 sec

Observation: We observe a significant speed-up in both versions compared to the serial time, especially when using the REDUCTION data attribute.

Output:

```
❯ ./p2b
Estimated pi is 3.142402, execution time = 0.134847 sec
```

3. Quick Sort

3.1 **Serial Implementation**: 7663.608671 sec

Note: I had to increase my system stack size using “ulimit -s unlimited” to fix a segmentation fault caused by stack overflow.

3.2 **Sequential initialization, OpenMP parallel, 2 threads**: 7225.312135 sec

Note: Here, OpenMP’s child processes don’t have the same stack size as the master thread, and to increase the stack size of the child processes, I set an env variable called OMP_STACKSIZE. Without this, I was facing stack overflow.

3.3 **Random initialization, OpenMP parallel, 2 threads:** 0.294041 sec

Name: Rohit Pujar Nagraj
USC ID: 4664 4939 18

Observation:

1. We do not observe a significant time difference between the serial and OpenMP parallel implementation. This is because, in both cases, the stack memory (which is in RAM) acts as the bottleneck, and accessing RAM using one thread or two threads won't make a meaningful difference as the unit delay will be similar.
2. The random initialization is significantly faster because the quicksort algorithm has a worst-case time complexity of $O(n^2)$ when the array is sorted in reverse order (Case 3.1 and 3.2). In contrast, the average case time complexity is O(nlogn) which is when the array is in no particular order (Case 3.3). The significant performance gain in 3.3 can also be attributed to the recursion stack not going as deep, avoiding the RAM access delay by many folds.

Output:

```
❯ ./p3c
122 2066 2073 3406 5536 6518 6565 8520 9121 11895 12170 13050 14097 14195 14385 14939
Execution time = 0.294041 sec
```