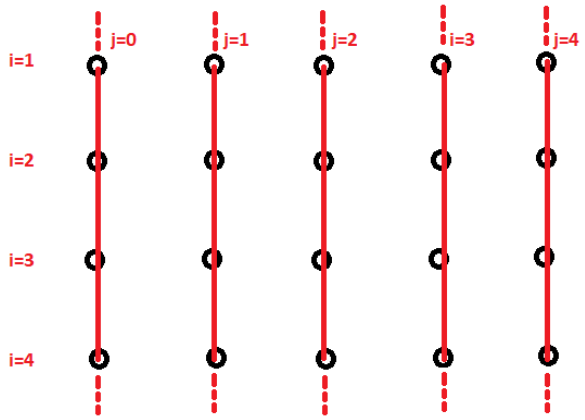
Haniye Kashgarani

Assignment 3

Prob. 1:

Iteration space dependency graph:



Flow Dependencies: $S_1\delta^F S_2$ **and** $S_2\,\delta^F S_1$

a[1][0] = a[0][0] + a[2][0]	a[2][0] = a[1][0] + a[3][0]	a[3][0] = a[2][0] + a[4][0]	a[4][0] = a[3][0] + a[5][0]
a[1][1] = a[0][1] + a[2][1]	a[2][1] = a[1][1] + a[3][1]	a[3][1] = a[2][1] + a[4][1]	a[4][1] = a[3][1] + a[5][1]
a[1][2] = a[0][2] + a[2][2]	a[2][2] = a[1][2] + a[3][2]	a[3][2] = a[2][2] + a[4][2]	a[4][2] = a[3][2] + a[5][2]
a[1][3] = a[0][3] + a[2][3]	a[2][3] = a[1][3] + a[3][3]	a[3][3] = a[2][3] + a[4][3]	a[4][3] = a[3][3] + a[5][3]
a[1][4] = a[0][4] + a[2][4]	a[2][4] = a[1][4] + a[3][4]	a[3][4] = a[2][4] + a[4][4]	a[4][4] = a[3][4] + a[5][4]

Prob. 2:

The code is not correct, because the variable oddCount is shared among all the thread and it can cause data racing which means multiple threads wants to write the shared variable at the same time. This causes a delay and may slow down the program. A data race occurs when two threads access the same memory without proper synchronization. This can cause the program to produce non-deterministic results in parallel mode.

For solving this issue we need to use REDUCTION clause for our parallel region. REDUCTION clause will useaddition operation over oddCount variable. The OpenMP REDUCTION clause provides a number of ways to accumulate the separate private variables back into the shared copy. So correct code will be:

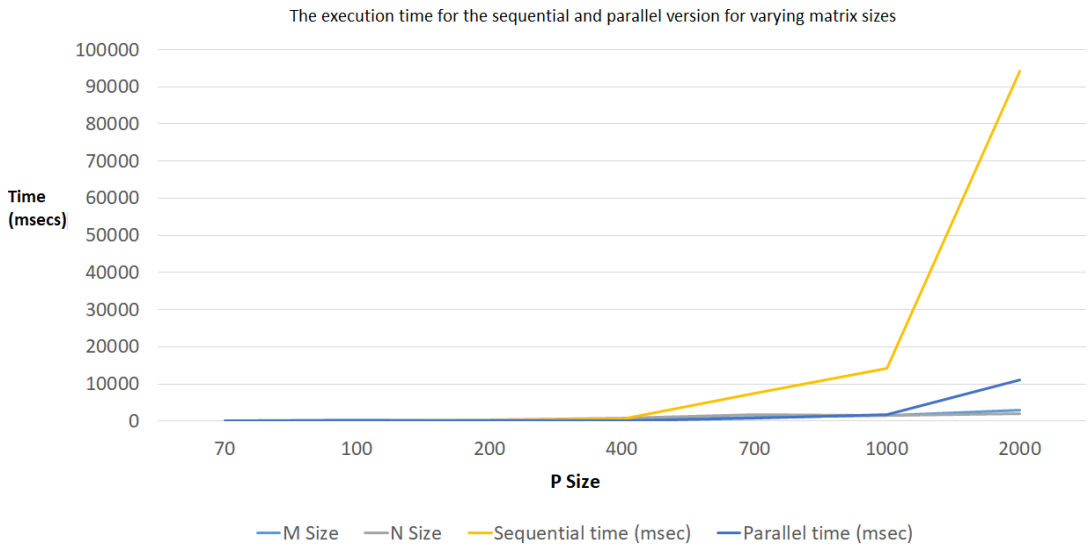
```
int data[N];
int oddCount = 0;
#pragma omp parallel for reduction(+:oddCount)
{
        for (int i = 0; i < 24; i++)
        {
                if (data[i] % 2)
                {
                        oddCount++;
                }
        }
}
```

Prob. 3:

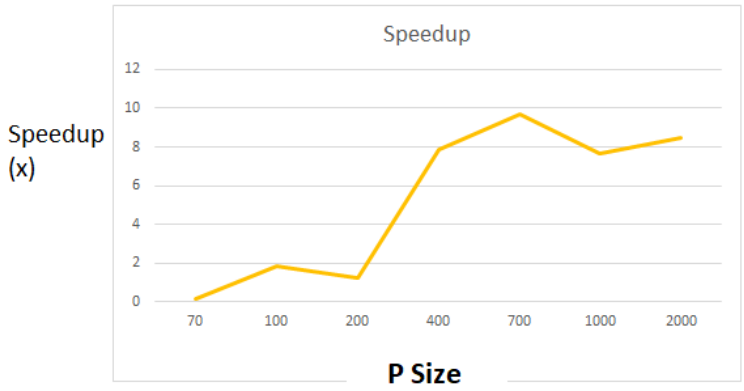
Difference between the previous code and current code for matrix multiplication is current multiplication code has defined a sum variable with doesn't exist in the previous code. This is kind of omitting one dependency which we had in previous code. So the inner loop can be easily parallelized with parallel for region. Since we have summation in this code we can use reduction clause over sum variable.

M Size	P Size	N Size	Sequential time (msec)	Parallel time (msec)	Speedup
50	70	90	1.32855	7.90095	0.16815067
200	100	150	11.6399	6.255	1.86089528
100	200	300	25.0445	20.1912	1.24036709
300	400	800	444.626	56.5965	7.85606884
1000	700	1700	7566.21	782.163	9.67344403
1500	1000	1500	14116.4	1845.24	7.65017017
3000	2000	2000	94330.2	11118	8.48445764

Part 1:



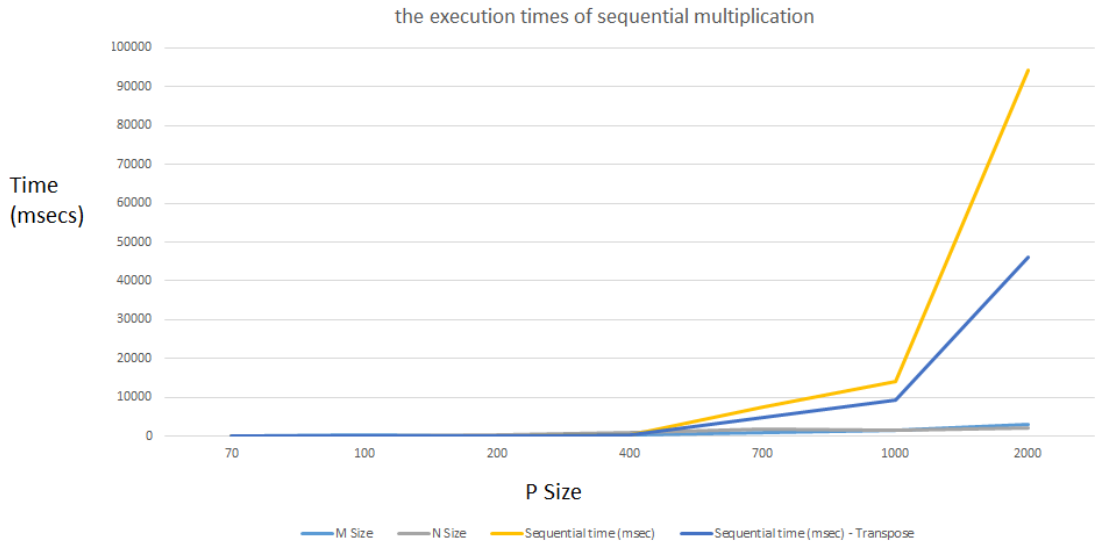
Part 2:



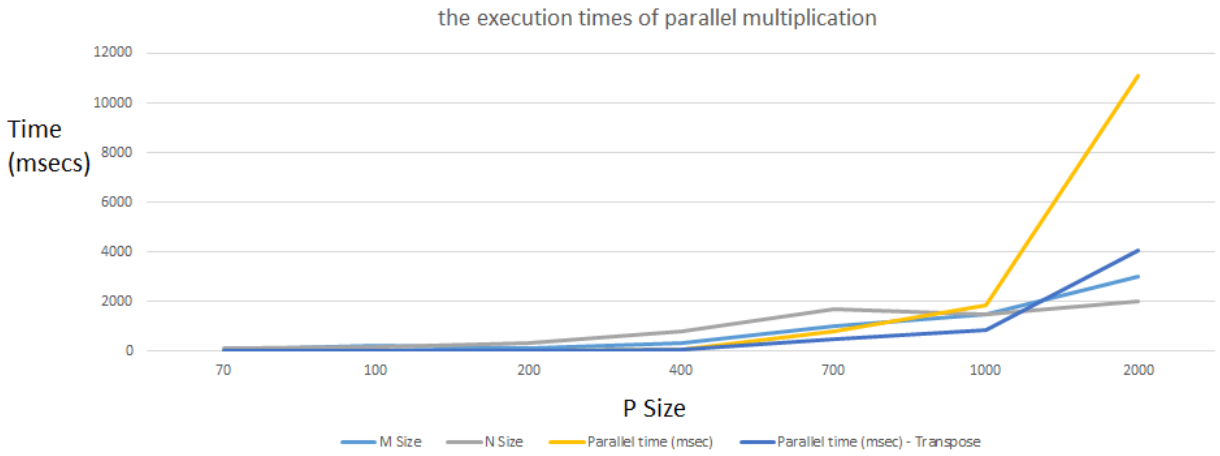
Part 3: The Computation complexity of the parallel version has changed for array size bigger than 100. Since we have speedup in each array size except the one less than 100. For arraysize less than 100, we have poorer performance in parallel execution and this is because, much time is spent for thread spawning than the computation itself.

Prob. 4:

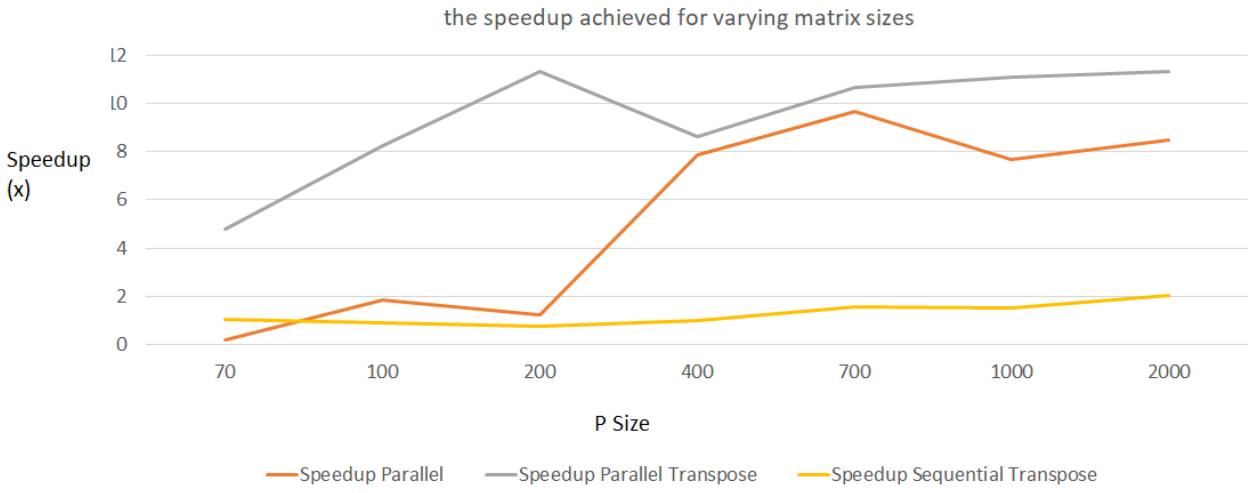
Part1:



Part 2:



Part 3:



Part 4:

As you can see, when we access a matrix, we access a[i][0], then a[i][1], a[i][2], a[i][3], and so on until we have hit the end. This is nice and sequential memory access and is much faster than haphazard ("random") accesses, and this is because of the locality of memory and the ith row does exist in the cache line. If we don't transpose the matrix_b, we have somewhat haphazard accesses, and for each value, we should fetch data from a higher level of caches and main memory. The first access is b[0][j], the second access is away from the first b[1][j], etc. There is a gap between every access. This kind of access is slow. It ruins the CPU's caching system. This is why transposed b in matrix multiplication code is having better performance. If you transpose the matrix, the rows will be the columns and the columns the rows, thus you get nice and sequential access to b.