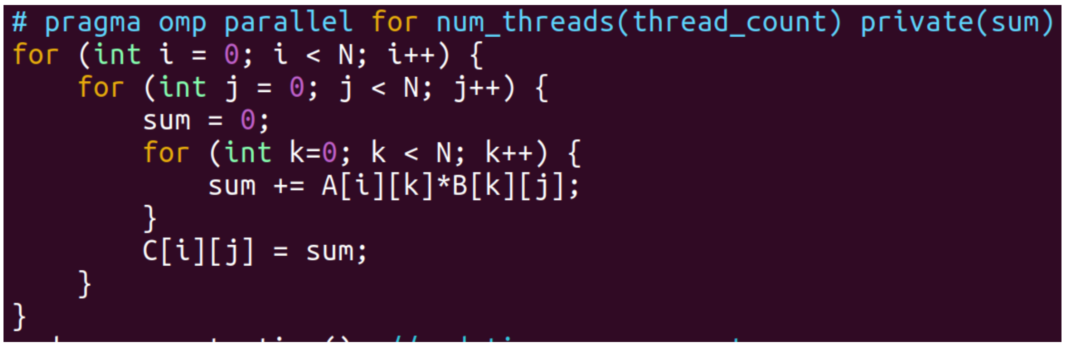
Problem 1

The two experiments both run in a 64-bit Ubuntu virtual machine with Core: 4 and Memory: 8GB.

Experimental Ideas

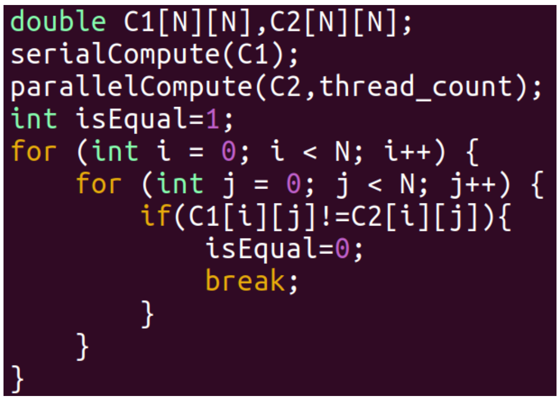
Matrix multiplication mainly consists of three loops, and omp pragma can be added before each loop. After verification, the outermost loop can achieve the highest degree of parallelism. The code is as follows:



```
# pragma omp parallel for num_threads(thread_count) private(sum)
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        sum = 0;
        for (int k=0; k < N; k++) {
            sum += A[i][k]*B[k][j];
        }
        C[i][j] = sum;
    }
}
```

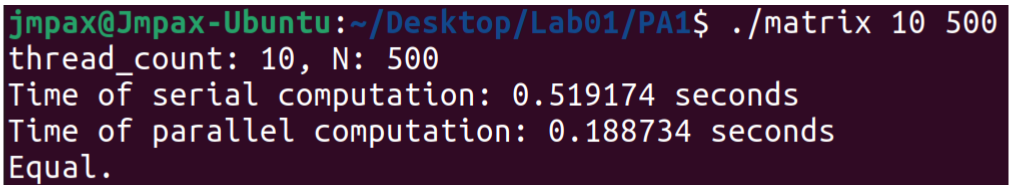
In addition, I also tested the row-first matrix multiplication algorithm, but the effect was not as good as the basic algorithm, because in the priority algorithm, pragma can only be added to the middle loop, and every time the outer loop variable is updated, the thread will wait.

I used functional programming to perform serial compute and parallel compute and validate each element of the resulting matrix.



```
double C1[N][N],C2[N][N];
serialCompute(C1);
parallelCompute(C2,thread_count);
int isEqual=1;
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        if(C1[i][j]!=C2[i][j]){
            isEqual=0;
            break;
        }
    }
}
```

Result Verification



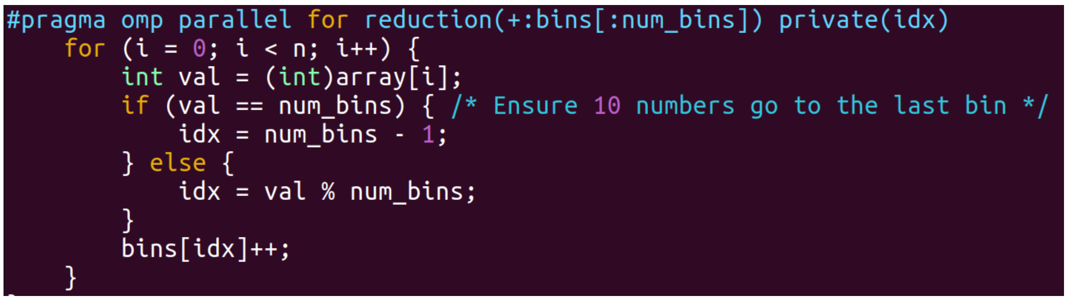
```
jmpax@Jmpax-Ubuntu:~/Desktop/Lab01/PA1$ ./matrix 10 500
thread_count: 10, N: 500
Time of serial computation: 0.519174 seconds
Time of parallel computation: 0.188734 seconds
Equal.
```

As you can see, we can get about 2.75 times improvement by using 10 thread concurrency on a 500-dimensional matrix.

Problem 2

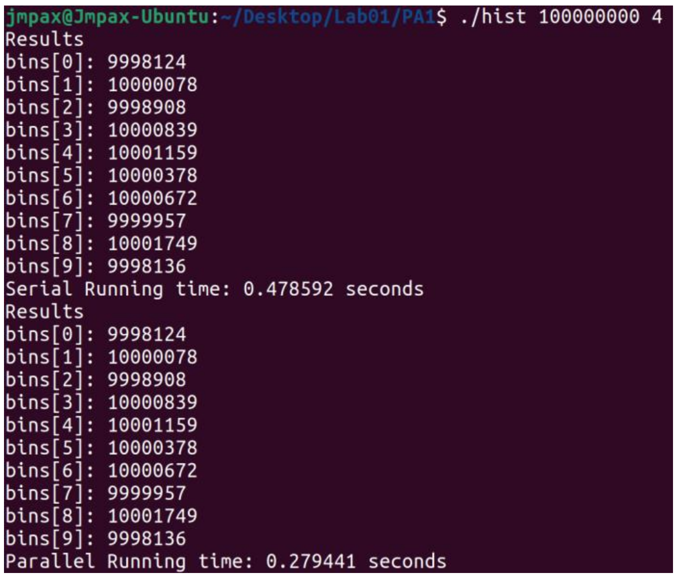
Experimental Ideas

The problem that needs to be solved in this experiment is the concurrent operation of multi-threaded histogram result statistical arrays. We can use the reduction operation provided by OpenMP to let each thread process the results of processing each part and finally combine them into a global variable. The code is as follows:



```
#pragma omp parallel for reduction(+:bins[:num_bins]) private(idx)
    for (i = 0; i < n; i++) {
        int val = (int)array[i];
        if (val == num_bins) { /* Ensure 10 numbers go to the last bin */
            idx = num_bins - 1;
        } else {
            idx = val % num_bins;
        }
        bins[idx]++;
    }
```

Result Verification



```
jmpax@Jmpax-Ubuntu:~/Desktop/Lab01/PA1$ ./hist 100000000 4
Results
bins[0]: 9998124
bins[1]: 10000078
bins[2]: 9998908
bins[3]: 10000839
bins[4]: 10001159
bins[5]: 10000378
bins[6]: 10000672
bins[7]: 9999957
bins[8]: 10001749
bins[9]: 9998136
Serial Running time: 0.478592 seconds
Results
bins[0]: 9998124
bins[1]: 10000078
bins[2]: 9998908
bins[3]: 10000839
bins[4]: 10001159
bins[5]: 10000378
bins[6]: 10000672
bins[7]: 9999957
bins[8]: 10001749
bins[9]: 9998136
Parallel Running time: 0.279441 seconds
```

It can be seen that at the data level of 10^8, the concurrency of 4 threads has been improved by about 1.71 times. And the results of histogram statistics are consistent.