SPP Assignment 2 Report

Objective

Optimizing matrix multiplication code using vectorization, data parallelism, multi-threading, etc

Part 0: OpenMP Overview

OpenMP is a programming model for parallel programming with a shared memory. The implementers of the compilers look at the specification and they implement it. Therefore, the compilers know how to compile a program which uses OpenMP.

In order to enable OpenMP for the C++ compiler, we must add the flag `-fopenmp` to the other compilation flags. For the GNU compiler, it would look like this
`g++ program.cpp -fopenmp -o program`

There are three ways to use OpenMP functionalities:

1. Using OpenMP functions which start with `omp_`. Example: `omp_get_thread_num()` returns the identification number of current thread.
2. Through environment variables
3. The last way is to use the so-called pragmas. They all start with `#pragma omp`.

We need to import `<omp.h>` header file to use functions from OpenMP specification.

1. **Parallel Construct**
   Using this construct we can execute the structured block in multiple threads. We can also specify the number of threads to use using environment variables or by adding a clause in pragma method.

   ```
   #pragma omp parallel
   {
   // structured block
   }
   ```

2. **For Construct**
   The for loop construct is probably one of the most widely used features of the OpenMP. The construct's aim is to parallelize an explicitly written for loop. The syntax of the loop construct is:

   ```
   #pragma omp parallel [clauses]
   {
   ```

```
#pragma omp for [clauses]
for (...)
    {
// body
    }
}
```

The `parallel` construct specifies the region which should be executed in parallel. A program without the parallel construct will be executed sequentially.

The `for` loop construct (or simply the loop construct) specifies that the iterations of the following for loop will be executed in parallel. The iterations are distributed across the threads that already exist.

If there is only one `#pragma omp for` inside `#pragma omp parallel` we can simplify both constructs into the combined construct.

```
#pragma omp parallel for [clauses]
for (...)
{

}
```

The clauses are additional options which we can set to the constructs. An example of a clause for `parallel` construct is `shared(...)` clause. When a program encounters the `parallel` construct, the program forks a team of threads. The variables, which are listed in the `shared(...)` clause, are then shared between all the threads.

3. **For and Reduction**

   In order to specify the reduction in OpenMP, we must provide

   - an operation (`+` / `*` / `o`)
   - and a reduction variable (`sum` / `product` / `reduction`). This variable holds the result of the computation.

   For example, the following for loop

```
sum = 0;
for (auto i= 0; i< 10; i++)
{
    sum+= a[i]
}
```

   can be parallelized with the `reduction` clause where the operation is equal to `+` and the reduction variable is equal to `sum`:

```
sum = 0;
#pragma omp parallel for shared(sum, a) reduction(+: sum)
for (auto i= 0; i< 10; i++)
{
    sum+= a[i]
}
```

There are many other constructs and clauses which can studied as per the use case.

Part 1: Matrix Multiplication Optimizations

Compilation command used while testing performance:

`gcc -O2 -fopenmp -std=gnu99 program`

Optimal Chain matrix multiplication

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:
`(ABC)D = (AB)(CD) = A(BCD) = ....`

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

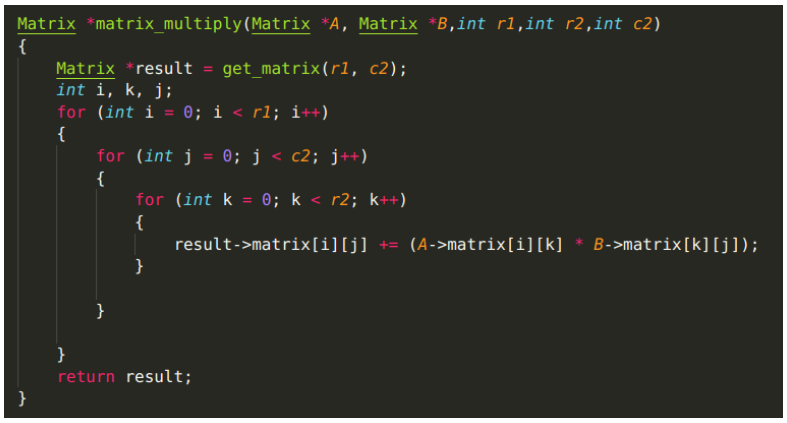
```
(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations
A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.
```

Clearly the first parenthesization requires less number of operations. Hence I used DP to find the optimal chain multiplication order.

Naive Sequential Algorithm

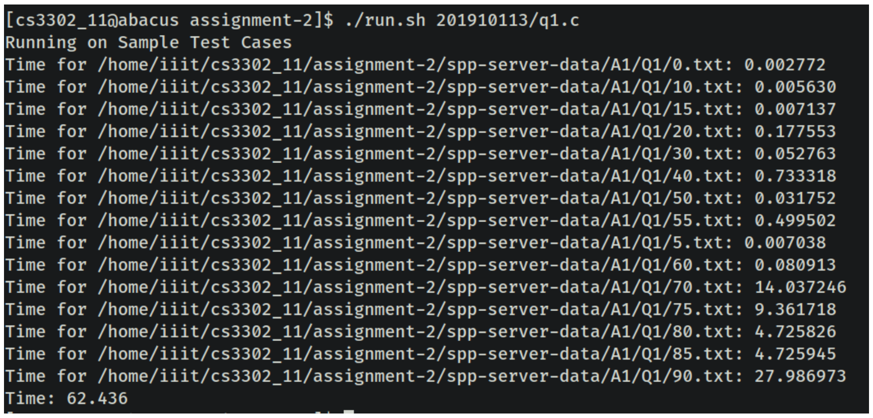
Unoptimized matrix multiplication algorithm

1. Code



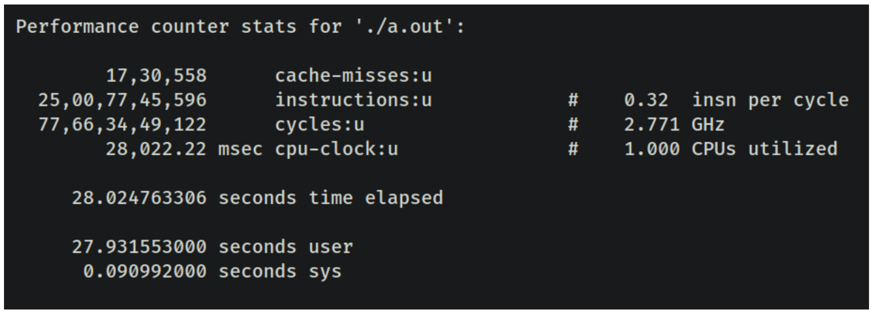
```
Matrix *matrix_multiply(Matrix *A, Matrix *B,int r1,int r2,int c2)
{
    Matrix *result = get_matrix(r1, c2);
    int i, k, j;
    for (int i = 0; i < r1; i++)
    {
        for (int j = 0; j < c2; j++)
        {
            for (int k = 0; k < r2; k++)
            {
                result->matrix[i][j] += (A->matrix[i][k] * B->matrix[k][j]);
            }

        }

    }
    return result;
}
```

2. Runtime : 62 seconds



```
[cs3302_11@abacus assignment-2]$ ./run.sh 201910113/q1.c
Running on Sample Test Cases
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/0.txt: 0.002772
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/10.txt: 0.005630
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/15.txt: 0.007137
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/20.txt: 0.177553
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/30.txt: 0.052763
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/40.txt: 0.733318
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/50.txt: 0.031752
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/55.txt: 0.499502
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/5.txt: 0.007038
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/60.txt: 0.080913
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/70.txt: 14.037246
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/75.txt: 9.361718
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/80.txt: 4.725826
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/85.txt: 4.725945
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/90.txt: 27.986973
Time: 62.436
```

3. Perf

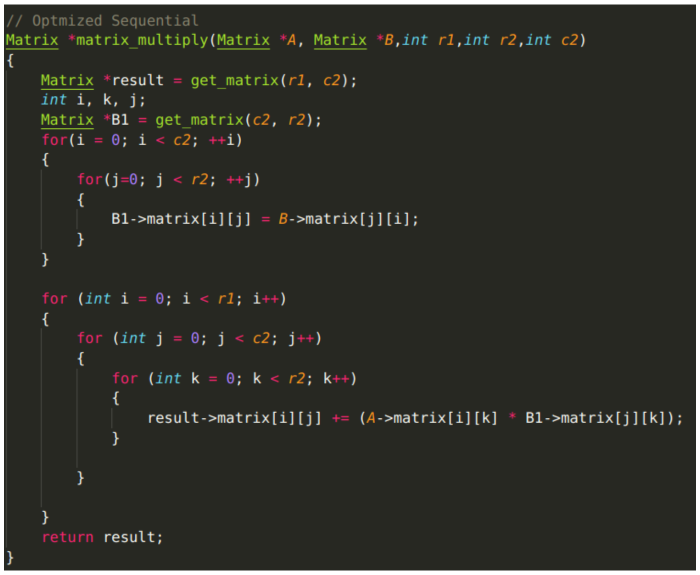


```
Performance counter stats for './a.out':

        17,30,558      cache-misses:u
  25,00,77,45,596      instructions:u            #    0.32  insn per cycle
  77,66,34,49,122      cycles:u                  #    2.771 GHz
        28,022.22 msec cpu-clock:u               #    1.000 CPUs utilized

     28.024763306 seconds time elapsed

     27.931553000 seconds user
      0.090992000 seconds sys
```

Cache Optimized Sequential Algorithm

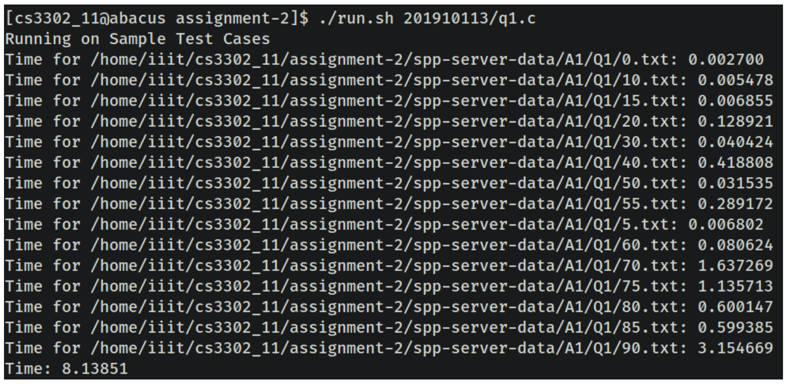
changing the loop order as to improve the chances of getting cache hit, as the neighbouring elements will now be accesed versus accesing different columns of the different matrices.

1. Code



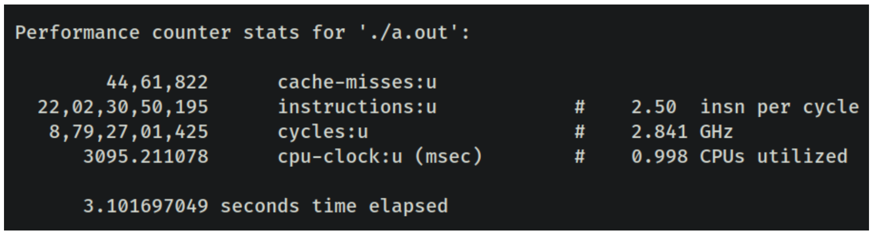
```
// Optmized Sequential
Matrix *matrix_multiply(Matrix *A, Matrix *B,int r1,int r2,int c2)
{
    Matrix *result = get_matrix(r1, c2);
    int i, k, j;
    Matrix *B1 = get_matrix(c2, r2);
    for(i = 0; i < c2; ++i)
    {
        for(j=0; j < r2; ++j)
        {
            B1->matrix[i][j] = B->matrix[j][i];
        }
    }

    for (int i = 0; i < r1; i++)
    {
        for (int j = 0; j < c2; j++)
        {
            for (int k = 0; k < r2; k++)
            {
                result->matrix[i][j] += (A->matrix[i][k] * B1->matrix[j][k]);
            }

        }

    }
    return result;
}
```

2. Runtime : 8.13 seconds



```
[cs3302_11@abacus assignment-2]$ ./run.sh 201910113/q1.c
Running on Sample Test Cases
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/0.txt: 0.002700
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/10.txt: 0.005478
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/15.txt: 0.006855
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/20.txt: 0.128921
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/30.txt: 0.040424
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/40.txt: 0.418808
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/50.txt: 0.031535
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/55.txt: 0.289172
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/5.txt: 0.006802
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/60.txt: 0.080624
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/70.txt: 1.637269
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/75.txt: 1.135713
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/80.txt: 0.600147
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/85.txt: 0.599385
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/90.txt: 3.154669
Time: 8.13851
```

3. Perf



```
Performance counter stats for './a.out':

       44,61,822      cache-misses:u
  22,02,30,50,195      instructions:u            #    2.50  insn per cycle
   8,79,27,01,425      cycles:u                  #    2.841 GHz
      3095.211078      cpu-clock:u (msec)        #    0.998 CPUs utilized

      3.101697049 seconds time elapsed
```

Adding Parallelization

To parallelize the the loops we need to add the following command before it.

`#pragma omp parallel for`

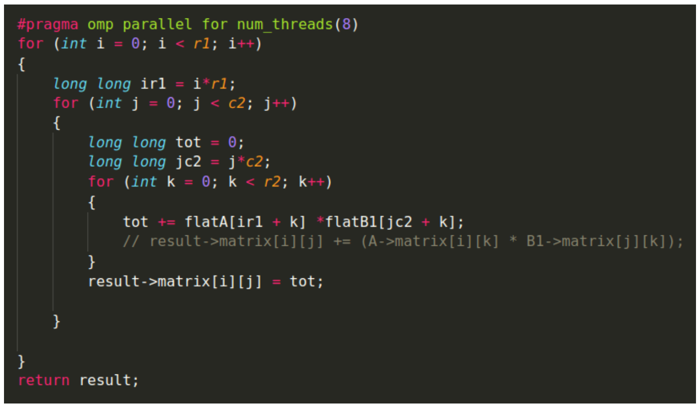
The following are the possible positions to add parallelization:

1. **To Loop i**
   Runtime: 3.26 seconds.
2. **To Loop j**
   Runtime: 3.35 seconds.
3. **To Loop k**
   We can't do this as this will lead to a race condition between various threads for `result->matrix[i][j]`

Using Flat Matrix

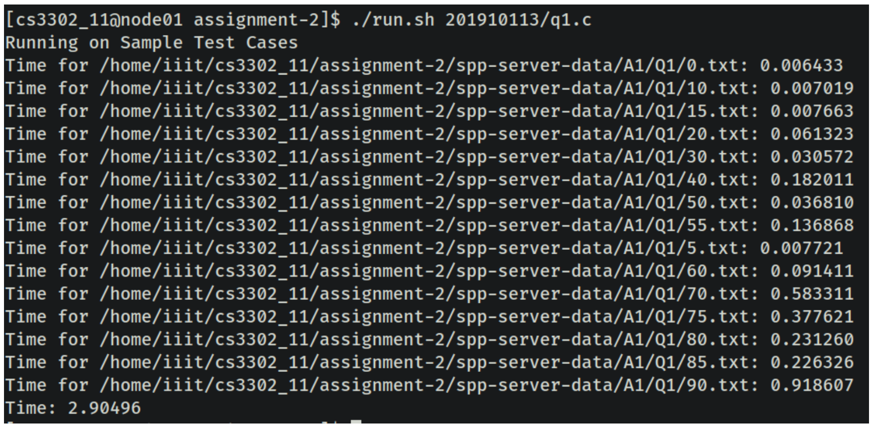
Over here I unrolled a 2D matrix into a 1D matrix to get better cache hits. This approach indeed performed better than the last approach.

1. Code



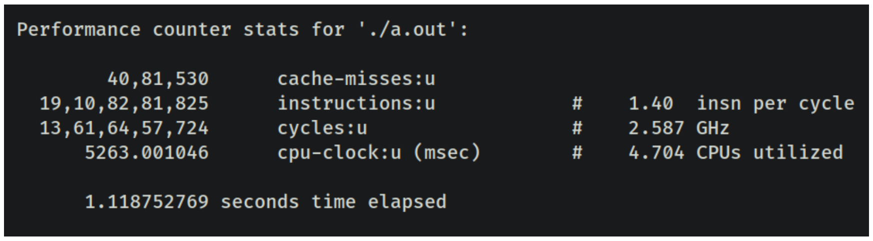
```
#pragma omp parallel for num_threads(8)
for (int i = 0; i < r1; i++)
{
    long long ir1 = i*r1;
    for (int j = 0; j < c2; j++)
    {
        long long tot = 0;
        long long jc2 = j*c2;
        for (int k = 0; k < r2; k++)
        {
            tot += flatA[ir1 + k] *flatB1[jc2 + k];
            // result->matrix[i][j] += (A->matrix[i][k] * B1->matrix[j][k]);
        }
        result->matrix[i][j] = tot;

    }

}
return result;
```

2. Runtime : 2.90 seconds



```
[cs3302_11@node01 assignment-2]$ ./run.sh 201910113/q1.c
Running on Sample Test Cases
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/0.txt: 0.006433
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/10.txt: 0.007019
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/15.txt: 0.007663
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/20.txt: 0.061323
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/30.txt: 0.030572
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/40.txt: 0.182011
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/50.txt: 0.036810
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/55.txt: 0.136868
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/5.txt: 0.007721
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/60.txt: 0.091411
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/70.txt: 0.583311
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/75.txt: 0.377621
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/80.txt: 0.231260
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/85.txt: 0.226326
Time for /home/iiit/cs3302_11/assignment-2/spp-server-data/A1/Q1/90.txt: 0.918607
Time: 2.90496
```

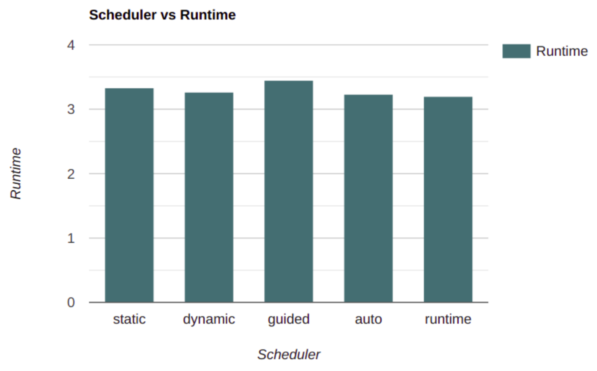
3. Perf



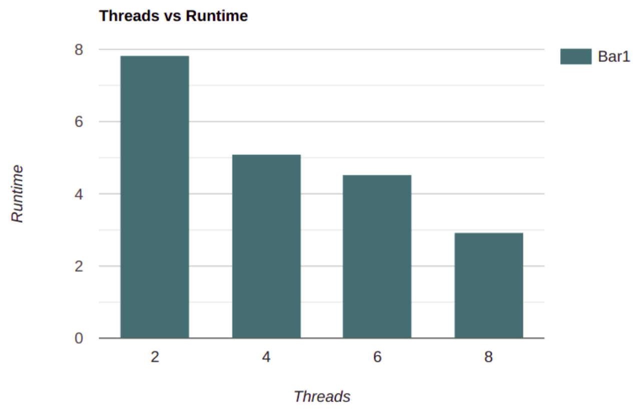
```
Performance counter stats for './a.out':

        40,81,530      cache-misses:u
  19,10,82,81,825      instructions:u            #    1.40  insn per cycle
  13,61,64,57,724      cycles:u                  #    2.587 GHz
      5263.001046      cpu-clock:u (msec)        #    4.704 CPUs utilized

      1.118752769 seconds time elapsed
```

Other Observations

1. The parallel programming in OMP is beneficial only when the input problem size is significantly larger. For smaller size problems, it is better to go with sequential programming
2. Tried various **scheduling methods** like **static, dynamic, guided** for scheduling threads of loops. But hardly any difference was observed in runtime. Hence finally decided to keep it default.

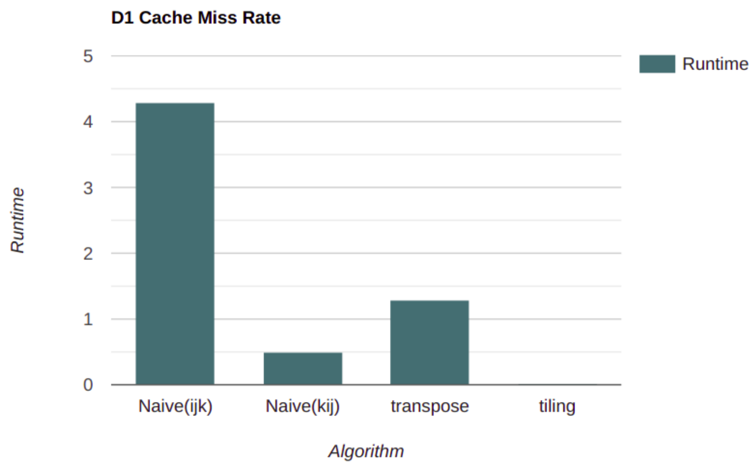


3. I ran my code for **different number of threads** and recorded the following observations. Only even number of threads are possible due to hyper threading in intel processors.



The code performed as per the expectation with runtime decreasing with increase in number of threads.

4. D1 cache comparison between various algorithms.



The results were as expected with Naive algorithm having most cache misses and tiling method having no cache miss.