HW2

Tapojyoti Mandal

February 14, 2020

Contents

1 Compilation and Execution

There are two *cpp* files submitted:

1. **matrix_inverse_serial.cpp** - This file contains a serial implementation of the lagorithm described in the homework guidelines.
2. **matrix_inverse_parallel.cpp** - This file contains a parrallel implementation of the serial algorithm of matrix_inverse calculation. OpenMP has been used for the parallelization objective.

I have used two different compilers to analyze the performance of the code. One is *g++* and the other is *icpc*. Before compilation we need to load the intel module as follows:

```
module load intel/2017A
```

1.1 G++

Steps to compile and run with g++ are as follows:

1. Compile the matrix_inverse_serial.cpp with the following command:

   ```
   g++ -O3 -o matrix_inverse_serial_gcc matrix_inverse_serial.cpp
   ```

2. For individual runs the following command can be used where the matrix dimension needs to be added to the command:

   ```
   ./matrix_inverse_serial_gcc 2000
   ```

   This will run the matrix_inverse simulation for a 2000x2000 upper triangular matrix. The result will look something like this:

   ```
   Time elapsed: 18751.5ms
   ```

   The runtime will be shown in *milliseconds*.

3. For batch runs I have added a file **serial_gcc.job** which has been used to obtain the serial run times of simulation.

1.2 ICPC

Steps to compile and run with icpc are as follows:

1. Compile the matrix_inverse_serial.cpp with the following command:

   ```
   icpc -O3 -o matrix_inverse_serial_icpc matrix_inverse_serial.cpp
   ```

2. For individual runs the following command can be used where the matrix dimension needs to be added to the command:

   ```
   ./matrix_inverse_serial_icpc.exe 2000
   ```

3. For batch runs I have added a file **serial_icpc.job** which has been used to obtain the serial run times of simulation.

2 Strategy used for Parallelization

Two strategies have been used for parallelizing the serial code:

1. **Recursion** - The homework guidelines present a recursive algorithm to find the matrix inverse of an upper triangular matrix. One way to parallelize this is to allow multiple parallel threads work on the sub-problems parallely. For example, for a matrix dimension of 32x32, the inverse of the top 16x16 matrix and the inverse of the bottom 16x16 matrix can be calculated parallely using two threads. And as we move on to matrix of larger dimensions we get more performance benefit out of the more number of available threads. In order to parallelize the matrix_inverse recrusive calls need to be task scheduled, and then the tasks need to wait till both the are done.

```
#pragma omp task shared(A, A_inv_top)
A_inv_top = matrix_inverse(A, start_row, start_row+(top_dim-1),
    start_col, start_col+(top_dim-1));
#pragma omp task shared(A, A_inv_btm)
A_inv_btm = matrix_inverse(A, start_row+top_dim, end_row, start_col+
    top_dim, end_col);
#pragma omp taskwait
```

2. **Matrix Multiplication** - Matrix multiplication has three for loops in it. It has a time complexity of $O(n^3)$ where n is the dimension of the matrices, assuming they are square matrices(for simplicity). To parallelize the for loops, I used OpenMP directives as shown in the code here.

```
#pragma omp parallel for
for(int i=0; i<A_row_dimension; i++){
    for(int j=0; j<B_col_dimension; j++){
        for(int k=0; k<A_col_dimension; k++){
            C[i][j] += A[i][k]*B[k][j];
        }
    }
}
```

Here, A and B are the input matrices and C is the result matrix. One thing which is important is that we need to enable **nested parallelism** feature. Because the matrix multiply resides in a parallel region. I used

```
export OMP_NESTED=TRUE
```

to enable nested parallelism feature.

These two are the primary workloads which have to be parallelized to get performance improvements. Other than that, I have for loops in other sections of code to do matrix copying, which also I have parallelized for better performance.

3 Results

The following table shows the serial runtime in *ms* for certain matrix dimensions. It can be observed the both g++ and icpc do a good job interms of relative runtime performance.

Matrix Dimension	Serial Time(GCC)	Serial Time(ICPC)
1000	627	1082
2000	18798	19282
3000	77245	76431
4000	203514	200042
5000	459281	465661

3.1 GCC

The following table shows the speedup obtained with matrix of different dimensions with respect to the number of processors used for parallelizing the code. One thing which is worth noting is that for matrix dimensions of 1000x1000 and 2000x2000 we observe that the speedup first increases and then drops. This is primarily because as the number of processors increase there is more communication overhead and the workload is not big enough to take advantage of the larger number of available processors. But for matrix of 4000x4000 we can see that the speedup keeps increasing, which means that it's workload of matrix multiplication is able to make the best use of the large number of processors available.

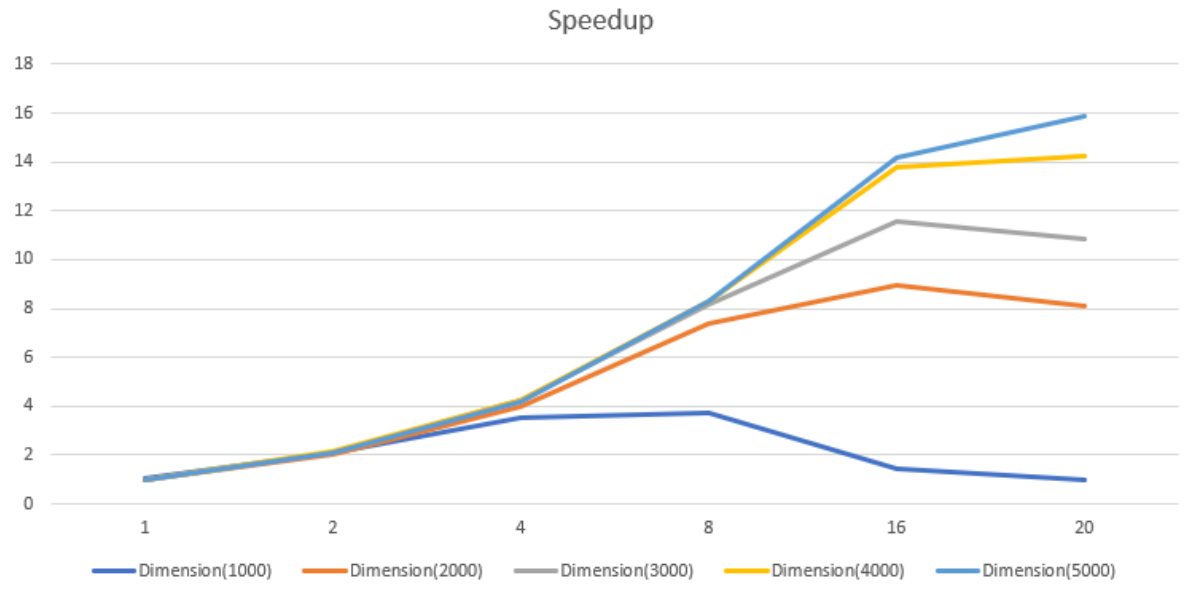

Figure 1: Speedup

And the following table shows th efficiency. We can observe the efficiency drop for all of the workloads. But for larger matrix dimensions such as the 5000x5000 the efficiency with larger number of processors is around 80% whereas for smaller matrix dimension such as 1000x1000 the efficiency is below 10%. This means that larger the workload more will be the benefit of running the parallel code on larger number of processors.

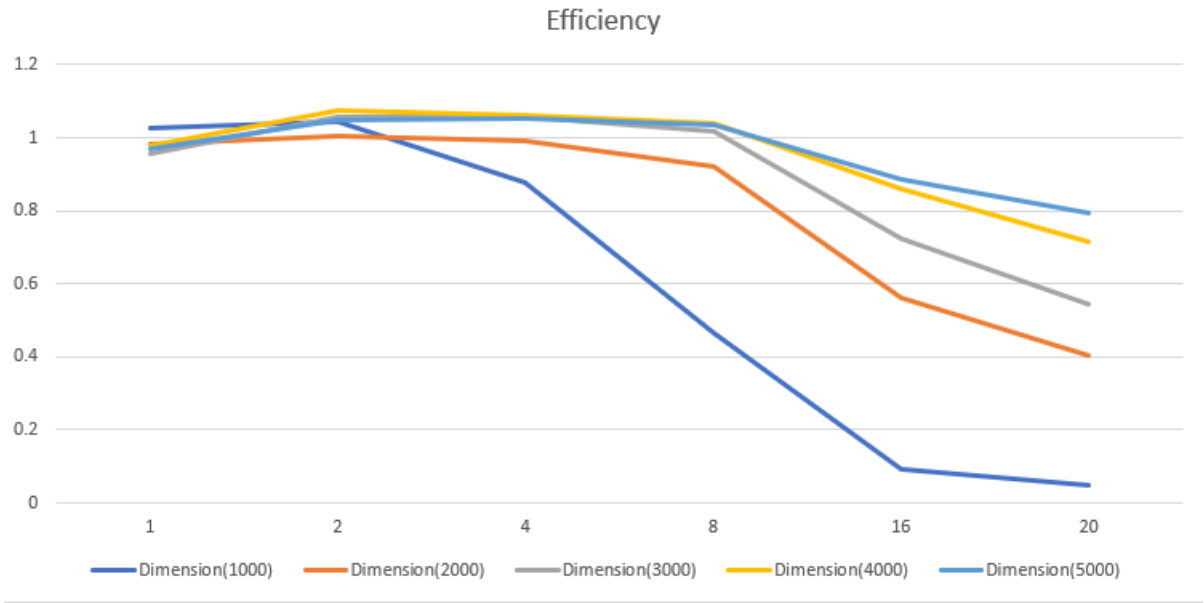

Figure 2: Efficiency

3.2 ICPC

The following table shows the speedup for different matrix dimensions with respect to different processors. We can observe that the speedup oibtained in *g++* is more as compared to *icpc*. But the trend seems to be the same. The larger the matrix dimension, more the speedup as the number of processors in increased. For smaller workloads the speedup first increases but then starts dropping due to communication overheads.

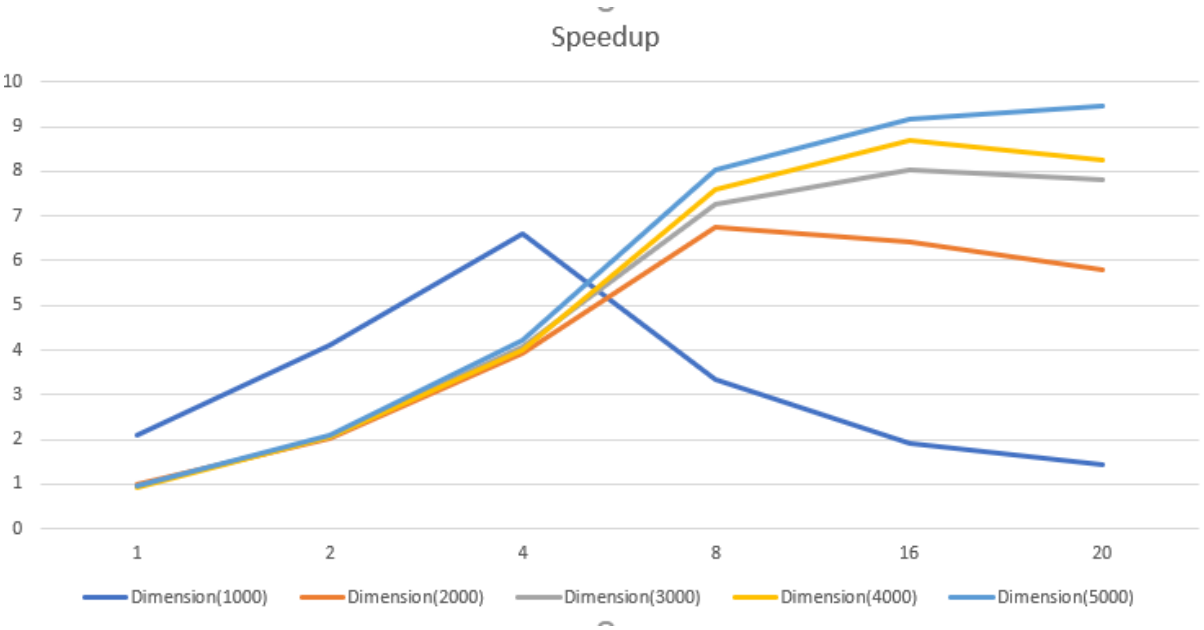

Figure 3: Speedup

The following table shows the efficiency for different matrix dimensions with respect to different processors. We can observe that the *g++* executable is more efficient as compared to the *icpc* executable. But here also the trend seems to be the same as *g++*.

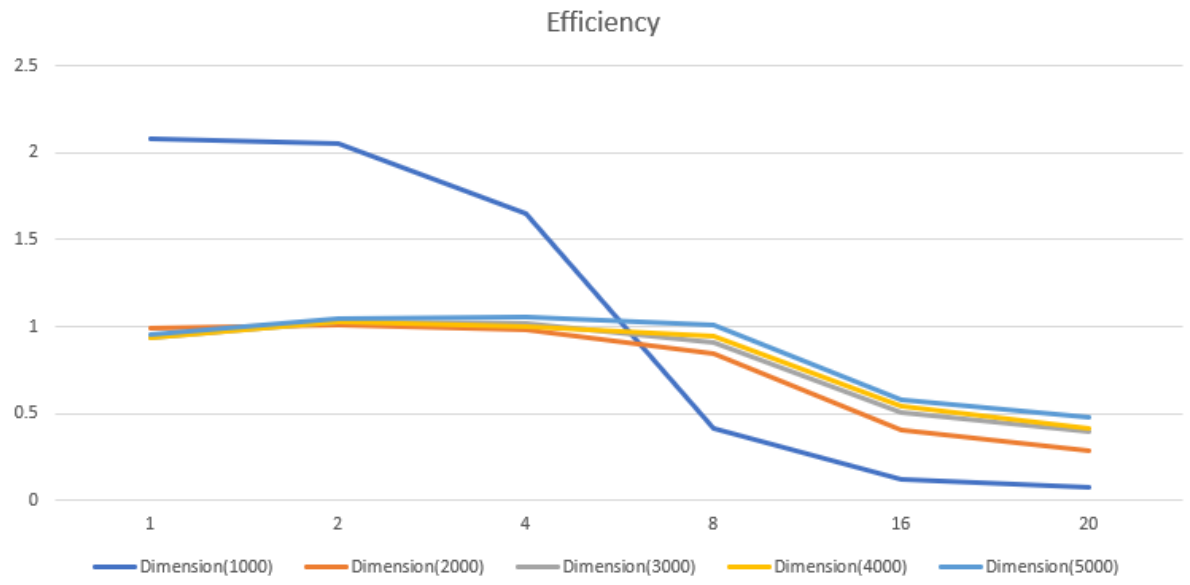

Figure 4: Efficiency

The plot for matrix of 1000x1000 seems to be weird. I checked the data set manually running it and it seems to be correct. Not sure why it would behave in such a manner.