CS410: Parallel Computing

Assignment 2

1. Introduction

In this assignment, we implemented a parallel version of SUMMA matrix multiplication algorithm using OpenMP and pthreads and conducted various experiments related to parallelism whose observations can be seen in this report.

2. Implementation

Version 1: OpenMP

OpenMP_SUMMA(A, B, C, n):

```
1. for k=0...n:
2.      #pragma omp parallel for schedule(static, chunk_size)
   num_threads (NUM_THREADS)
3.      for i=0...n:
4.           for j=0...n:
5.                C[i][j] += A[i][k] * B[k][j]
```

Version 1: Pthreads

Chunk_Matrix_Mul(A, B, C, n, k, thread_id):

```
1.  chunk_i = thread_id/sqrt(NUM_THREADS)
2.  chunk_j = thread_id%sqrt(NUM_THREADS)
3.  max_chunk_i = min((chunk_i + 1)*chunk_size, n)
4.  max_chunk_j = min((chunk_j + 1)*chunk_size, n)
5.  for i=chunk_i*chunk_size...max_chunk_i:
6.       for j=chunk_j*chunk_size...max_chunk_j:
7.            C[i][j] += A[i][k]*B[k][j]
```

```
1. pthread_t workers[NUM_THREADS];
2. for k=0...n:
3.    for thread=0...NUM_THREADS:
```

```
4.          data.k = k
5.          data.thread_id = thread
6.          pthread_create(workers       +       thread_id,       NULL,
   Chunk_matrix_mul, &data)
7.    for thread=0...NUM_THREADS:
8.          pthread_join(workers[thread], NULL)
```

So in this version we parallelize the inner two loops of SUMMA. For pthreads, we divide the work of the inner two loops over a grid of sqrt(NUM_THREADS) x sqrt(NUM_THREADS) threads, i.e. each thread computes the matrix product of N/sqrt(NUM_THREADS) x N/sqrt(NUM_THREADS) size matrix.

Version 2:

In this version we implement Block matrix multiplication using SUMMA algorithm. In order to do so, we divide the matrix into blocks where each block can run parallely. However, the outer k loop still runs sequentially as in the original SUMMA algorithm. The OpenMP version of our implementation is as follows:

OpenMP_SUMMA_Block(A, B, C, n):

```
1. for kk=0:BLOCK_SIZE:n #increment of BLOCK_SIZE in every iter:
2.      #pragma omp parallel for schedule(static, chunk_size)
   num_threads (NUM_THREADS)
3.      for ii=0:BLOCK_SIZE:n:
4.           for jj=0:BLOCK_SIZE:n:
5.                for k=kk...kk+BLOCK_SIZE:
6.                     for i=ii...ii+BLOCK_SIZE:
7.                          for j=jj...jj+BLOCK_SIZE:
8.                               C[i][j] += A[i][k] * B[k][j]
```

Similar version can be implemented using Pthreads also. We choose to exploit blocking this way because this ensures that the block of each matrix is a square and occupies one-third of the cache size. We know when these conditions are satisfied, the communication cost is also minimized. The other way to implement block matrix multiplication is to use 1D row or column

blocked layout. We didn't choose this implementation because the communication cost in this case is high.

Version 3:

In this version, we use BLAS library routine `dgemm` to compute the matrix product of each of the blocks of the matrix. In other words, the innermost 3 for loops in Version 2 which compute the matrix product of each of the blocks, is replaced by the BLAS library routine `dgemm` in Version 3.

3. Experiments

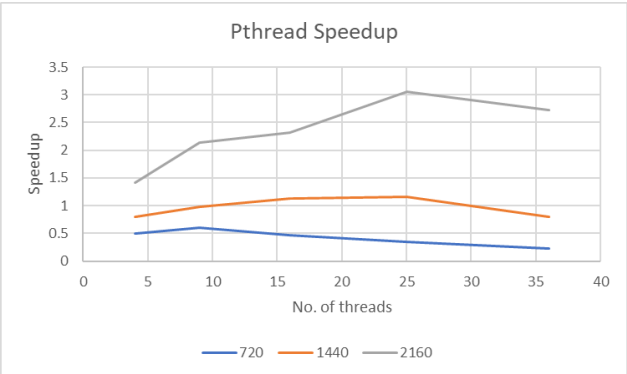
Experiment 1:

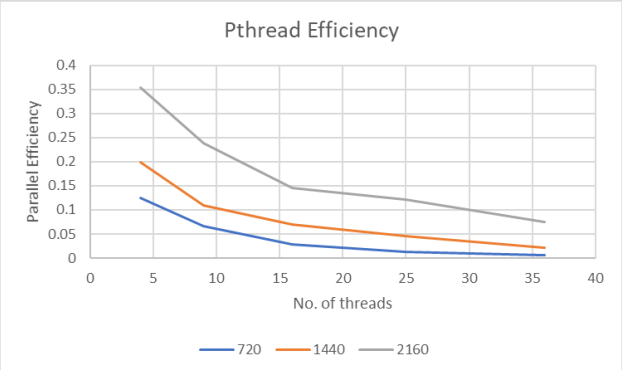
In this experiment, we execute the code for the SUMMA algorithm serially for n={720, 1440, 2160}. The execution time (in ms) obtained are as follows:

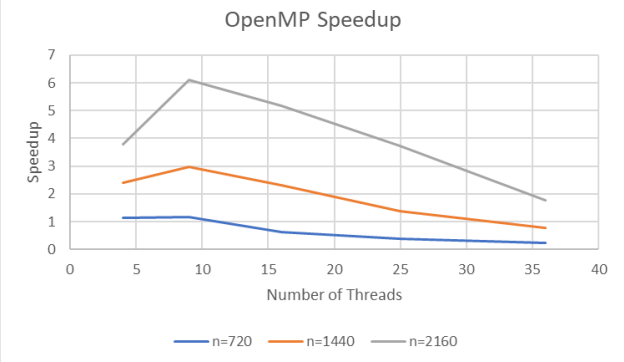
Size	n=720	n=1440	n=2160
Execution Time (ms)	253	1758	6411

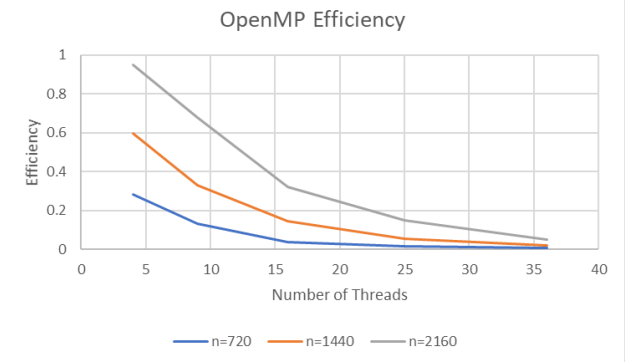
Experiment 2:

In this experiment, we execute version 1 of our code with OpenMP and Pthreads implementations. We vary the number of threads between p={4,9,16, 25, 36}. The plots for speedup and efficiency obtained are as follows:





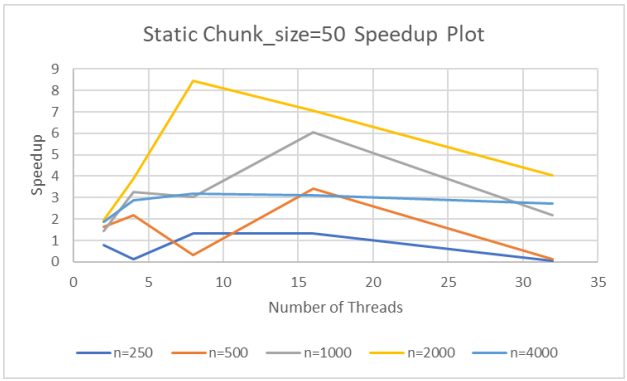


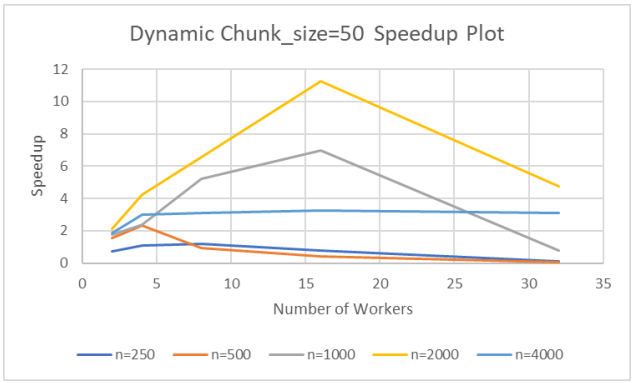


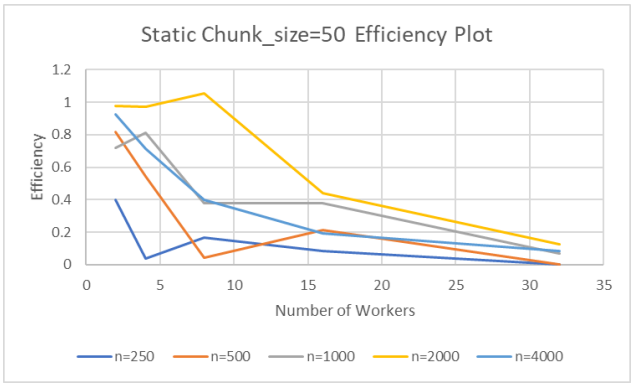
The speedup behaviour is as expected i.e. when the number of threads is small, the speedup is small because we are not creating enough parallelism and when the number of threads is too large, the speedup is small because there is a lot of overhead due to the large number of thread creation. Similarly, the behaviour of efficiency is also as expected which should decrease as we increase the number of threads. This is because increasing the number of threads does not give us a linear speedup.

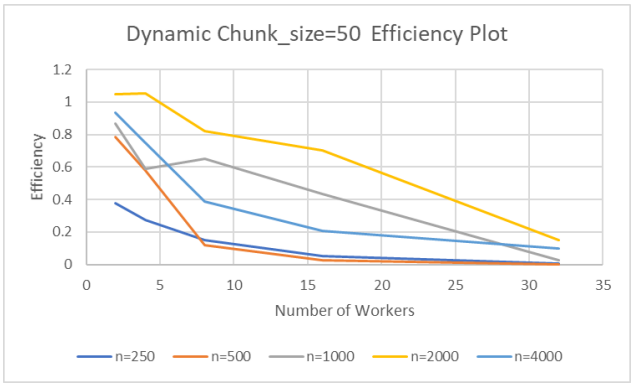
Experiment 3:

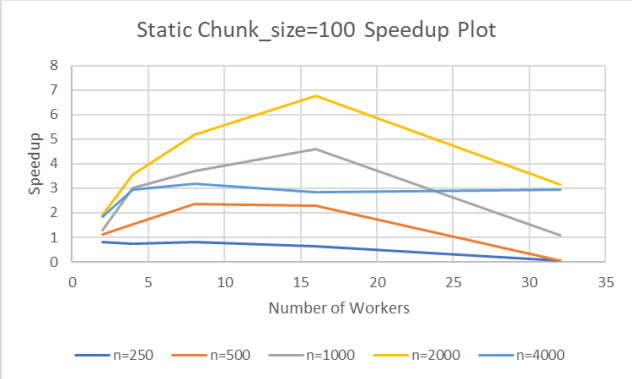
In this experiment, we run the version 1 of our code implemented using OpenMP. We vary the scheduling policy between “STATIC” and “DYNAMIC”, number of threads p={2, 4, 8, 16, 32}, n={250, 500, 1000, 2000, 4000} and chunk size b={50, 100}. The results obtained are as follows:

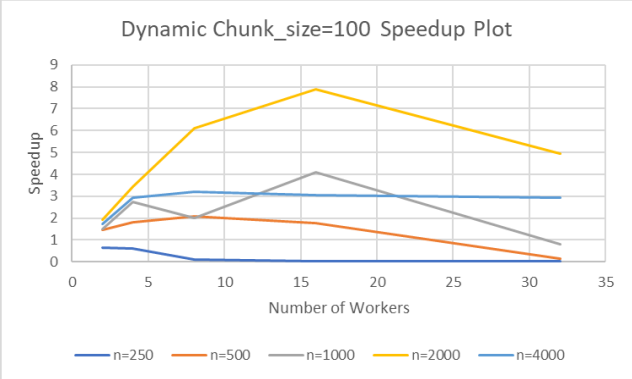


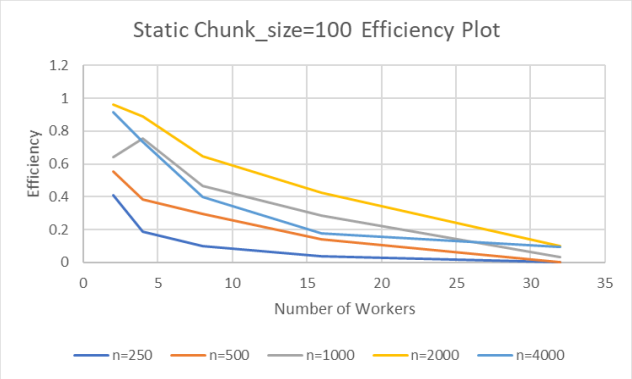


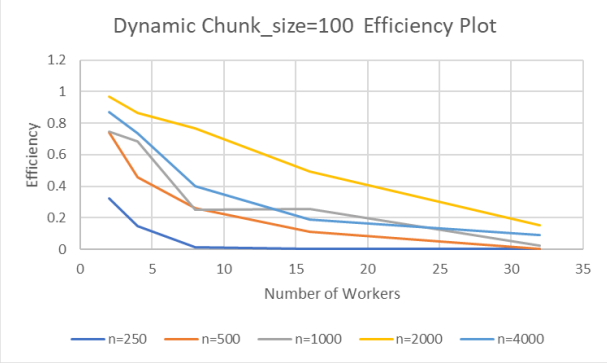












We can notice that for almost all the cases except when n is small, the speedup and efficiency for DYNAMIC scheduling is more than the STATIC scheduling. We also observe that a chunk size of 50 is better suited for our application than 100. The trends for speedup and efficiency are as expected.

Experiment 4:

In this experiment, we run version 2 and version 3 to find the best block size b which yields the fastest execution time. We fix the number of threads, p=36 and size of the matrix n=2160. For version 2 and 3, we evaluate the execution time over b={10, 20, 30, 40, 60, 90, 120, 180, 360} and b={60, 90, 120, 180, 360} respectively. The results obtained are as follows:

Version 2 (execution time in ms)

	b=10	**b=20**	**b=30**	**b=40**	**b=60**	**b=90**	**b=120**	**b=180**	**b=360**
OpenMP	207	182	171	227	150	204	241	355	776

Pthreads	1061	612	486	552	343	266	255	295	321

Version 2 (Efficiency)

	b=10	**b=20**	**b=30**	**b=40**	**b=60**	**b=90**	**b=120**	**b=180**	**b=360**
OpenMP	0.68	0.77	0.82	0.62	0.94	0.69	0.59	0.40	0.18
Pthreads	0.13	0.23	0.29	0.25	0.41	0.53	0.55	0.48	0.44

Version 2 (Speedup)

	b=10	**b=20**	**b=30**	**b=40**	**b=60**	**b=90**	**b=120**	**b=180**	**b=360**
OpenMP	24.65	28.0	29.8	22.5	34.0	25.0	21.1	14.37	6.57
Pthreads	4.8	8.34	10.5	9.24	14.88	19.18	20.0	17.3	15.9

Version 3 (execution time in ms)

	b=60	**b=90**	**b=120**	**b=180**	**b=360**
OpenMP	3032	1172	325	180	209
Pthreads	3764	1672	918	456	162

Version 3 (Efficiency)

	b=60	**b=90**	**b=120**	**b=180**	**b=360**
OpenMP	0.04	0.12	0.43	0.78	0.67
Pthreads	0.03	0.08	0.15	0.31	0.87

Version 3 (Speedup)

	b=60	**b=90**	**b=120**	**b=180**	**b=360**
OpenMP	1.68	4.35	15.70	28.35	24.42
Pthreads	1.35	3.05	5.55	11.19	31.50

Therefore, for Version 2 OpenMP and Pthreads implementation, block size of 60 and 120 performs the best respectively. For Version 3 OpenMP and Pthreads implementation, block size of 180 and 360 performs the best respectively.