High Performance Computing

Shantanu Tyagi, Kirtan Delwadia
CS-301 201801015, 201801020
Prof. Bhaskar Chaudhry Lab Date: 10/02/2021
Due Date: 20/02/2021

Assignment 3

Hardware Details:

- Architecture: x86_64
- Byte Order: Little Endian
- CPU(s): 4
- Sockets: 1
- Cores per socket: 1
- Model name: Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 256K
- L3 cache: 6144K

Question 1: Calculating π using Trapezoidal method

Implementation Details:

1. Description about the Serial implementation

We numerically evaluate π using trapezoidal method using a single loop that runs for the entire length of the problem giving us the value of π at the end of the loop.

```
for(i=0;i<num_steps;i++)
{
        x=(i+0.5)*step;
        sum=sum+4.0/(1.0+x*x);
}
```

2. Description about the Parallel implementation

In parallel programming, we divide the entire problem into sub problems according to the number of cores/processors available and each processor calculates the result of the sub problem assigned to it. Here, we have two choices. The first is the critical approach in which we sum the individual results obtained by each thread working on the core it was assigned, to a global variable. The second is the Private approach in which we maintain a global array which stores the results obtained by individual cores which is finally summed up to get the answer. We have used 6 implementations that use either of the two cases mentioned above.

1. **Atomic**

```
#pragma omp parallel
{
    int i,id,nthrds;
    double x,sum;
    id=omp_get_thread_num();
    nthrds=omp_get_num_threads();
    if(id==0)nthread=nthrds;
    for(i=id,sum=0.0;i<num_step;i+=nthrds)
    {
      x=((double)i+0.5)*step;
      sum+=4/(1+x*x);
    }
    sum=sum*step;
    #pragma atomic
    ans+=sum;
}
```

2. **Critical**

```
#pragma omp parallel
     {
       int i,id,nthrds;
       double x,sum;
       id=omp_get_thread_num();
       nthrds=omp_get_num_threads();
       if(id==0)nthread=nthrds;
       for(i=id,sum=0.0;i<num_step;i+=nthrds)
       {
          x=((double)i+0.5)*step;
          sum+=4/(1+x*x);
       }
        #pragma omp critical
       {
           ans+=sum*step;
       }
     }
```

3. **For loop**

```
#pragma omp parallel
     {
       int i;
       double x;
        #pragma omp for
            for(i=0;i<num_step;i++)
            {
               x=((double)i+0.5)*step;
               sum+=4/(1+x*x);
            }
     }
     ans=sum*step;
```

4. **Padding**

```
#pragma omp parallel
     {
       int i,id,nthrds;
       double x;
       id=omp_get_thread_num();
       nthrds=omp_get_num_threads();
       if(id==0)nthread=nthrds;
       for(i=id,sum[id][0]=0.0;i<num_step;i+=nthrds)
       {
          x=((double)i+0.5)*step;
          sum[id][0]+=4/(1+x*x);
       }
     }
     for(i=0;i<nthread;i++)ans+=sum[i][0]*step;
```

5. **No padding**

```
#pragma omp parallel
     {
       int i,id,nthrds;
       double x;
       id=omp_get_thread_num();
       nthrds=omp_get_num_threads();
       if(id==0)nthread=nthrds;
       for(i=id,sum[id]=0.0;i<num_step;i+=nthrds)
       {
          x=((double)i+0.5)*step;
          sum[id]+=4/(1+x*x);
       }
     }
     for(i=0;i<nthread;i++)ans+=sum[i]*step;
```

6. **Reduction**

```
#pragma omp parallel
    {

        double x;
        #pragma omp for reduction(+:sum)
            for(i=0;i<num_step;i++)
            {
               x=((double)i+0.5)*step;
               sum+=4/(1+x*x);
            }
    }
    ans=sum*step;
```

Complexity:

1. Complexity of serial code:

The time varies linearly with the problem size since we just have one loop which runs for n times. When the problem size is n, we have the time complexity as $O(n)$.

2. Complexity of parallel code:

Here again we have a problem with size n, however it is divided into p processors each of which runs in parallel. This means that the loop runs $\frac{n}{p}$ times on each processor giving us a time complexity of $O(\frac{n}{p})$. This was for the critical approach. However in the private approach since we have an array of size p, we initialise it using a loop that runs p times and finally p times again while summing up the p values stored in the array. This gives us a time complexity of $O(2 \cdot p + \frac{n}{p})$. Usually, $p<<n$, so we can approximate the time complexity for larger problem sizes as $O(\frac{n}{p})$.

3. Cost of Parallel algorithm

Speedup is calculated as ratio of time taken to execute the code serially to time taken to execute the code parallelly. The expected speed up is p but this if often not achieved due to memory access, parallel overhead, and thread synchronisation.

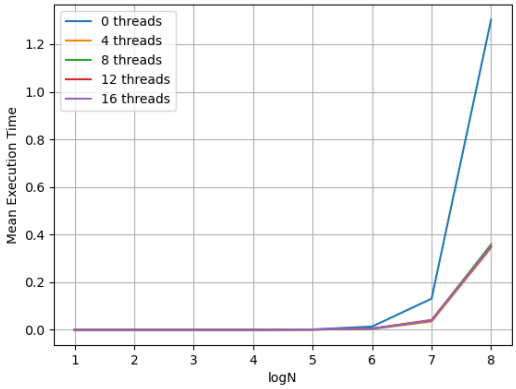
4. Theoretical Speed Up:

For a problem of size n and total p processors, we have theoretical speed up as $\frac{n}{(\frac{n}{p})} = p$
Thus for 1 processor, we get $\frac{n}{(\frac{n}{1})} = 1$
and for 4 processors, we get $\frac{n}{(\frac{n}{4})} = 4$

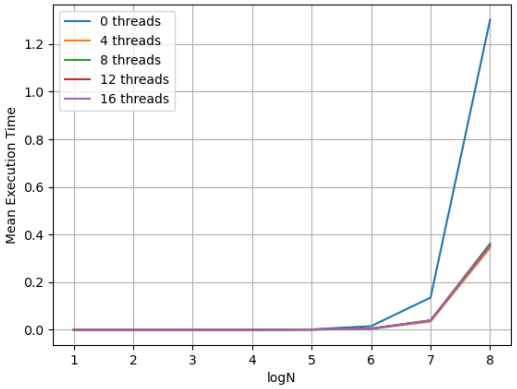
Curve Related Analysis:

1. Time Curve Related analysis:

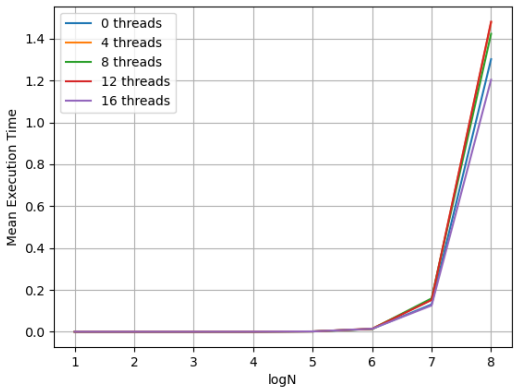
It is observed that as the number of processors on which the code runs increases the time taken for executing or calculating the result reduces. Therefore, the time for executing the code is inversely proportional to the number of processors on which the code is working since the computations get divided on more processors as opposed to less processors, hence reducing the computation time. As the problem size increases, the time taken also increases for both serial and parallel implementations. This is because with increased problem size, more calculations need to be done and the loop runs longer. However, if we increase the threads beyond a certain threshold than the cost to create and synchronize threads will become significant giving poorer results.



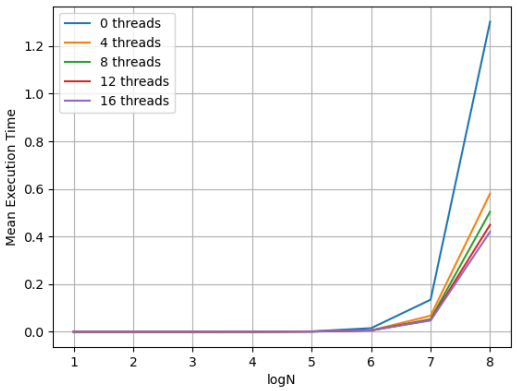
(a) Atomic



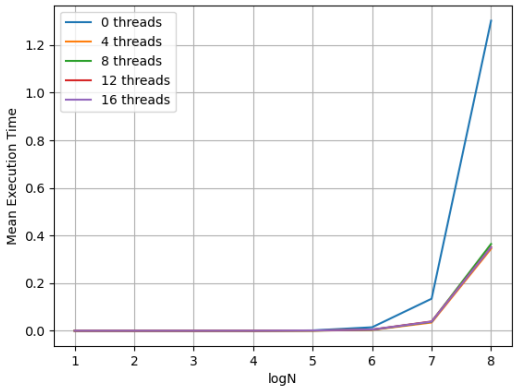
(b) Critical



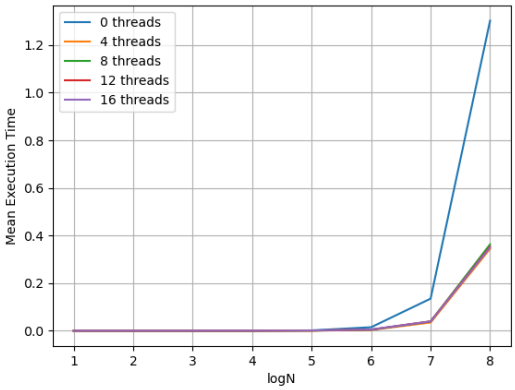
(c) For loop



(d) No pad



(e) Pad

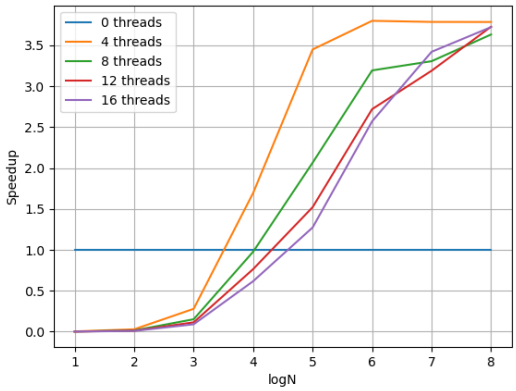


(f) Reduction

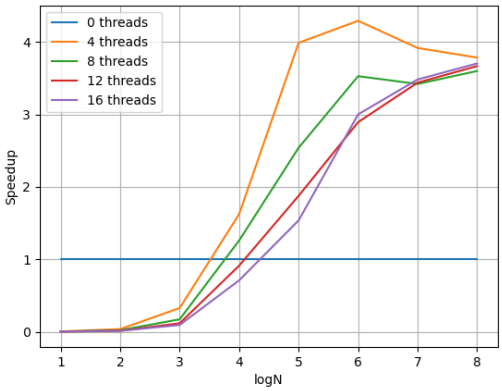
Figure 1: Mean Execution Time

2. Speedup Curve Related analysis:

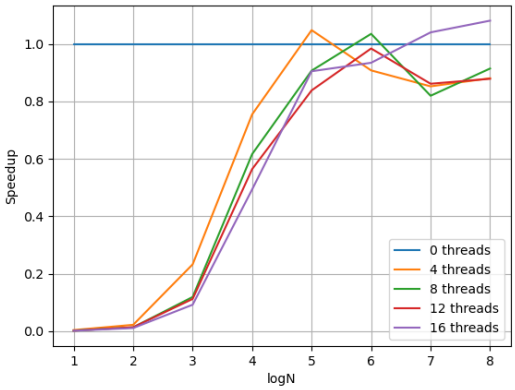
The speed up increases as the number of processes increases on which the code runs. However, if we increase the threads beyond a certain threshold than the cost to create and synchronize threads will become significant giving poorer results. The parallel code runs simultaneously on the multiple cores instead of the single core as contrast to the serial code. As the problem size increases initially the parallel threads have a speedup less than one due to parallel overhead and pipeline not being utilised efficiently, however as size increases the speed increases because the pipeline is now being used efficiently. It is observed that the speedup initially remains below 1. As the problem size increases, the speed up reaches closer to 4. The deviation in these values from the theoretical values depicts that for large problem size the memory access time also comes into consideration. It is also noted that for small problem sizes the speedup of 1 thread is greater than speedup of multiple threads which might be due to parallel overhead.



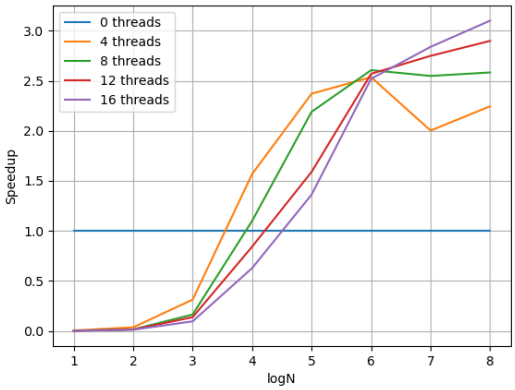
(a) Atomic



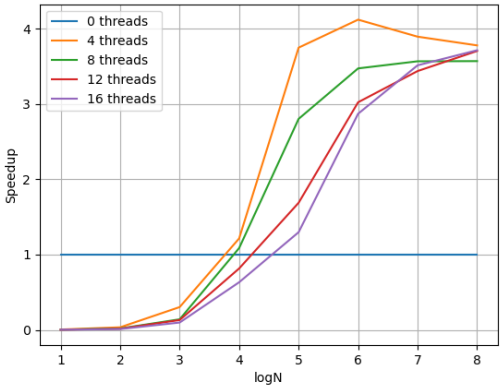
(b) Critical



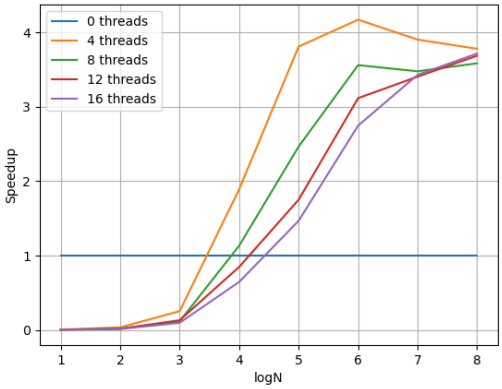
(c) For loop



(d) No pad



(e) Pad



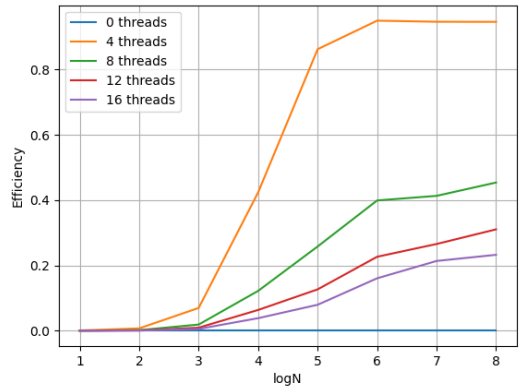
(f) Reduction

Figure 2: Speedup

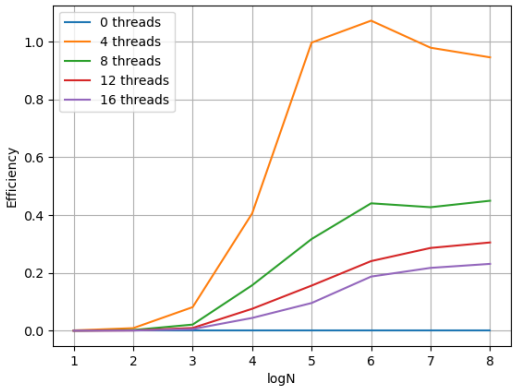
We see that Critical, Padding, and reduction give better performance than other methods.

3. Efficiency Curve Related analysis:

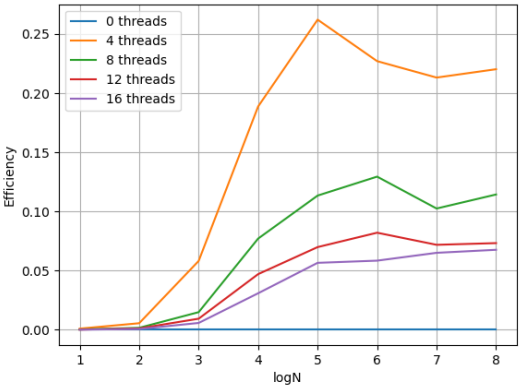
Efficiency is speedup divided by the number of threads, p. It is also observed that with greater number of processors working together the efficiency also decreases. The greater number of threads lesser will be the efficiency because managing so many threads also requires time and the parallel overhead starts dominating. Cost of creating more and more threads is not able to balance how much more performance it gives as compared to lesser number of threads. As the problem size increases the efficiency also gradually starts increasing for a given implementation and gradually reaches 1.



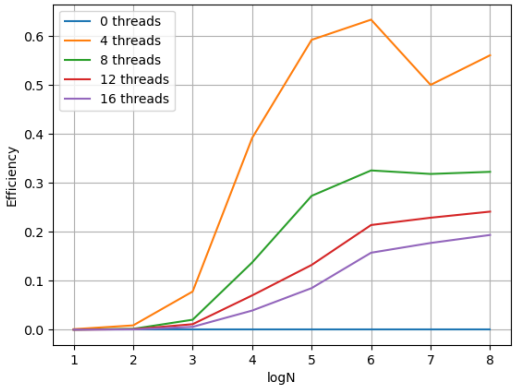
(a) Atomic



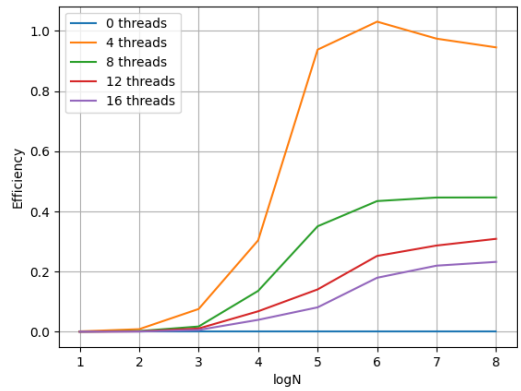
(b) Critical



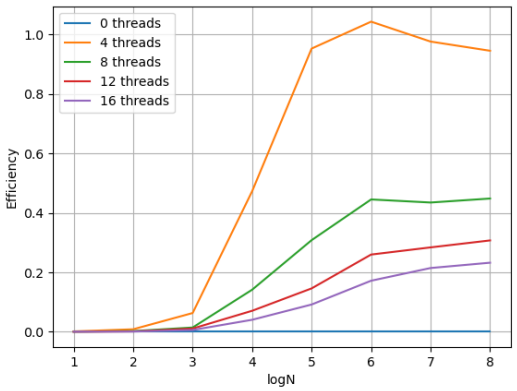
(c) For loop



(d) No pad



(e) Pad



(f) Reduction

Figure 3: Efficiency

Question 2:Vector addition, Vector triad

Implementation Details:

1. Description about the Serial implementation

We use a single loop that runs for the entire length of the problem and sum up the values stored at the indices into a result vector having size equal to the problem size. Three arrays are used. The serial code for the vector summation is written below.

```
for(i = 0; i < N ; i++)
{
        c[i] = a[i] + b[i];
}
```

The serial code for the multiplication of 2 vectors followed by summation is written below.

```
for(i = 0 ; i < N ; i++)
{
        sum = sum + a[i]*b[i];
}
```

2. Description about the Parallel implementation

In parallel programming, we divide the entire problem into sub problems according to the number of cores/processors available and each processor calculates the result of the sub problem assigned to it. Here, we maintain a global array having size equal to the problem size and this array is divided into portions that are parallel accessed by each processor.
The parallel code's snippet for vector summation problem is given below.

```
#pragma omp parallel
{
        #pragma omp for
        for(i=0;i<N;i++)
        {
                c[i]=a[i]+b[i];
        }
}
```

The parallel code's snippet for multiplication of 2 vectors followed by their summation is given below.

```
#pragma omp parallel private(temp) shared(sum)
{
    temp=0;
        #pragma omp for
        for(i=0;i<N;i++)
        {
                temp=a[i]*b[i];
        }
```

```
9   #pragma atomic
10  sum=sum+temp;
11 }
```

Complexity:

1. Complexity of serial code:

The time varies linearly with the problem size since we just have one loop which runs for n times. When the problem size is n, we have the time complexity as $O(n)$.

2. Complexity of parallel code:

Here again we have a problem with size n, however it is divided into p processors each of which runs in parallel. This means that the loop runs $\frac{n}{p}$ times on each processor giving us a time complexity of $O(\frac{n}{p})$.

3. Cost of Parallel algorithm

Speedup is calculated as ratio of time taken to execute the code serially to time taken to execute the code parallelly. The expected speed up is p but this if often not achieved due to memory access, parallel overhead, and thread synchronisation.

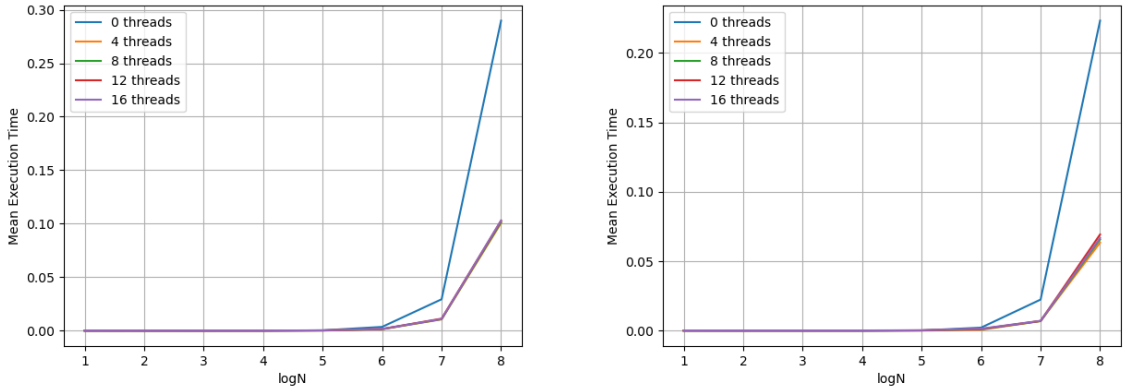
4. Theoretical Speed Up:

For a problem of size n and total p processors, we have theoretical speed up as $\frac{n}{(\frac{n}{p})} = p$
Thus for 1 processor, we get $\frac{n}{(\frac{n}{1})} = 1$
and for 4 processors, we get $\frac{n}{(\frac{n}{4})} = 4$

Curve Related Analysis:

1. Time Curve Related analysis:

It is observed that as the number of processors on which the code runs increases the time taken for executing or calculating the result reduces. As the problem size increases, the time taken also increases for both serial and parallel implementations. For smaller problem sizes the serial code is better but as the problem size increase the parallel approach gives significantly better results.



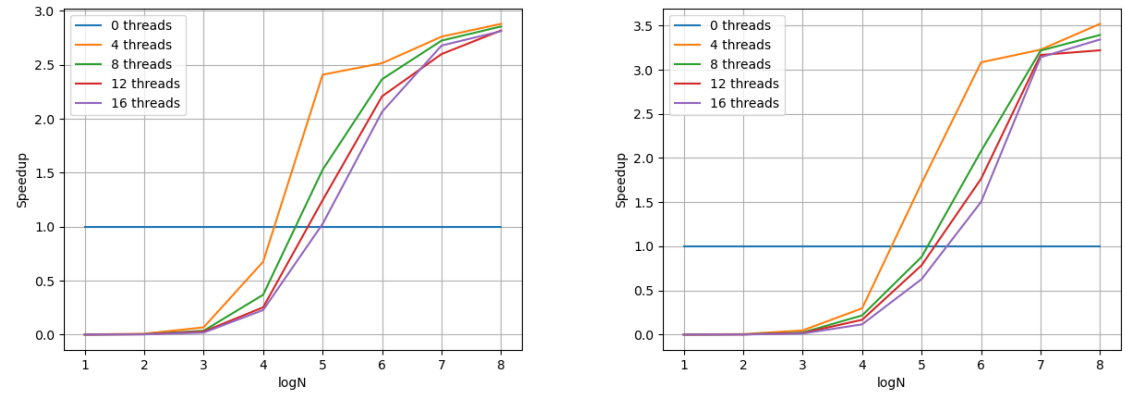
(a) For summation

(b) For atomic multiplication followed by summation

Figure 4: Mean Execution Time

2. Speedup Curve Related analysis:

The speed up increases as the number of processes increases on which the code runs. However, if we increase the threads beyond a certain threshold than the cost to create and synchronize threads will become significant giving poorer results. As the problem size increases initially the parallel threads have a speedup less than one due to parallel overhead and pipeline not being utilised efficiently, however as size increases the speed increases because the pipeline is now being used efficiently. It is observed that the speedup initially remains below 1 and serial approach is better in such cases. As the problem size increases, the speed up reaches closer to 4.



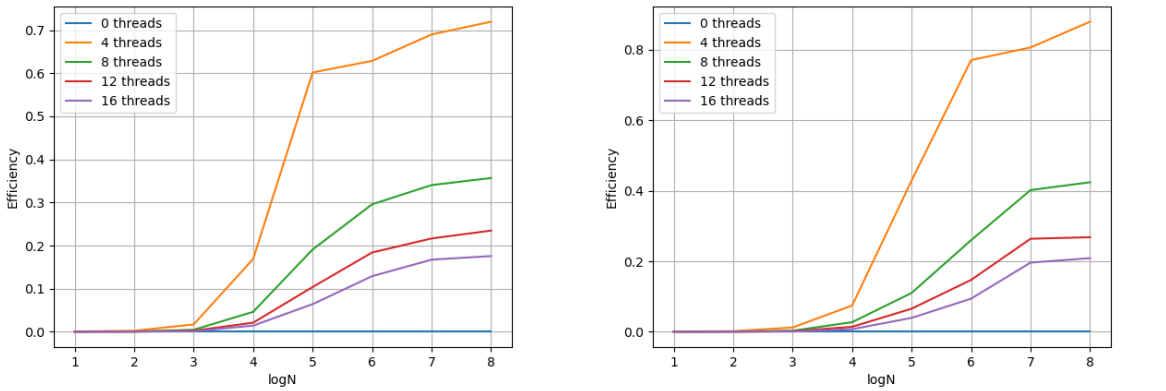
(a) For summation

(b) For atomic multiplication followed by summation

Figure 5: Speedup

3. Efficiency Curve Related analysis:

Efficiency basically tells us the Average work done per unit thread. For less number of threads, the commutation overhead is negligible and hence we observe maximum efficiency. As the problem size increases the efficiency also gradually starts increasing for a given implementation and gradually reaches 1.



(a) For summation

(b) For atomic multiplication followed by summation

Figure 6: Efficiency

Question 3: Matrix Multiplication

Implementation Details:

1. Description about the Serial implementation

We multiply two matrices by running three nested for loops, such that the outer two loops iterate to evaluate every element in the matrix whose value is the result of the third for loop which multiplies a row with a column. at the end of the three loops, we have the desired product.

1. The serial code for the normal matrix multiplication is given below.

```
for (i = 0; i < N; i++)
{
    for (j = 0; j < N; j++)
    {
        for (k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

2. The serial code for the blocked matrix multiplication is given below.

```
for (i = 0; i < N; i += block_size)
{
  for (j = 0; j < N; j += block_size)
  {
      for (k = 0; k < block_size; ++k)
          {
            for (l = 0; l < block_size; ++l)
                {
                    for (m = 0; m < N; ++m)
                        {
                          c[i + k][j + l] += a[i + k][m] * b[m][j + l];
                        }
                }
          }
   }
}
```

2. Description about the Parallel implementation

Various parallel implementations used are

1. Static and dynamic scheduling of outermost for loop

```
#pragma omp parallel shared(A,B,C) private(i,j,k)
{
    #pragma omp for  schedule(static/dynamic)
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            for (k = 0; k < N; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

2. Static and dynamic scheduling of outermost for loop with blocking

```
#pragma omp parallel shared(a,b,c) private(i,j,k,l,m)
{
    #pragma omp for schedule(static/dynamic)
    for (i = 0; i < N; i += block_size)
    {
      for (j = 0; j < N; j += block_size)
      {
          for (k = 0; k < block_size; ++k)
```

```
                {
                  for (l = 0; l < block_size; ++l)
                      {
                          for (m = 0; m < N; ++m)
                              {
                                c[i + k][j + l] += a[i + k][m] * b[m][j + l];
                              }
                      }
                }
        }
    }
}
```

3. Static scheduling of middle for loop

```
for(i=0;i<N;i++)
{
        #pragma omp parallel shared(a,b,c) private(j,k)
        {
                #pragma omp for schedule(static)
                for(j=0;j<N;j++){
                        for(k=0;k<N;k++){
                                c[i][j] += a[i][k]*b[k][j];
                        }
                }
        }
}
```

Complexity:

1. Complexity of serial code:

We are using three nested for loops which gives us a time complexity of $O(n^3)$.

2. Complexity of parallel code:

After parallisation the time complexity of the code becomes, $O(\frac{n^3}{p})$

3. Cost of Parallel algorithm

The expected speed up is p but this if often not achieved due to memory access, parallel overhead, and thread synchronisation.

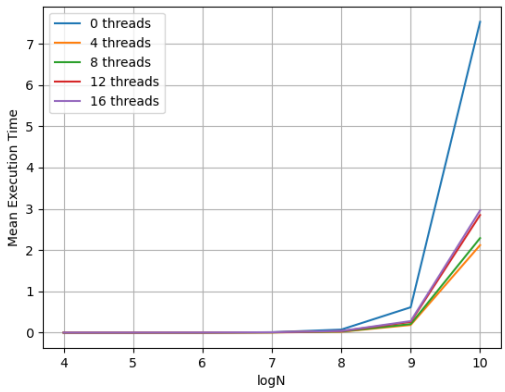
4. Theoretical Speed Up:

For a problem of size n and total p processors, we have theoretical speed up as $\frac{n}{(\frac{n}{p})} = p$
Thus for 1 processor, we get $\frac{n}{(\frac{n}{1})} = 1$
and for 4 processors, we get $\frac{n}{(\frac{n}{4})} = 4$

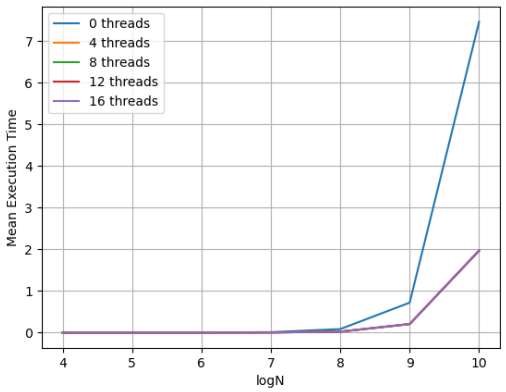
Curve Related Analysis:

1. Time Curve Related analysis:

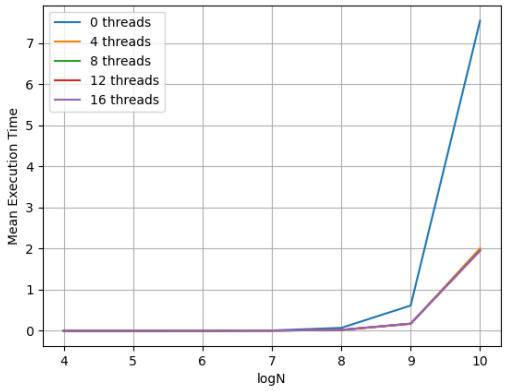
It is observed that as the number of processors on which the code runs increases the time taken for executing or calculating the result reduces. As the problem size increases, the time taken also increases for both serial and parallel implementations. For smaller problem sizes the serial code is better but as the problem size increase the parallel approach gives significantly better results.



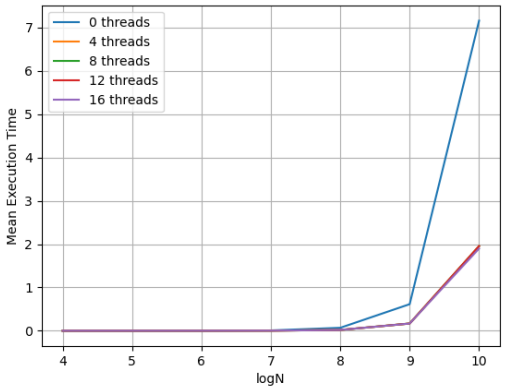
(a) Blocked Static



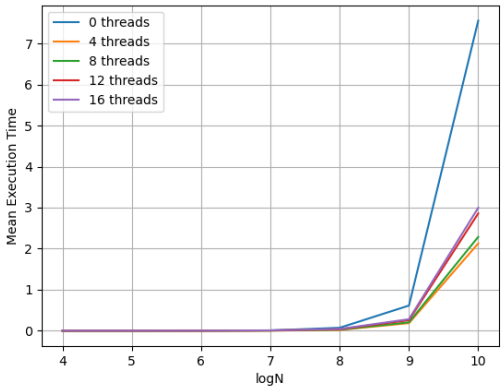
(b) Block Dynamic



(c) Outermost Static



(d) Outermost Dynamic

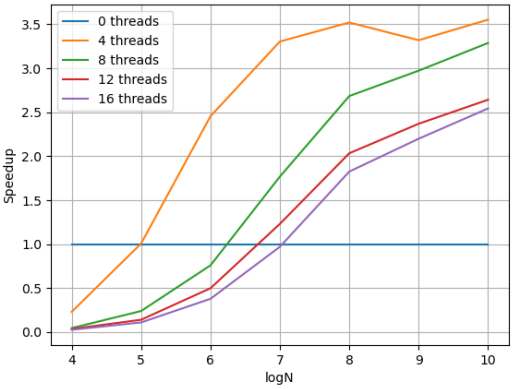


(e) Middle Static

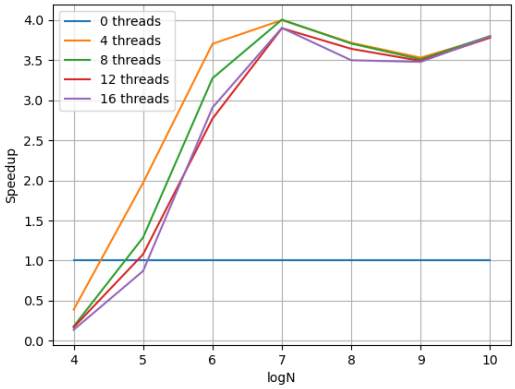
Figure 7: Mean execution time

2. Speedup Curve Related analysis:

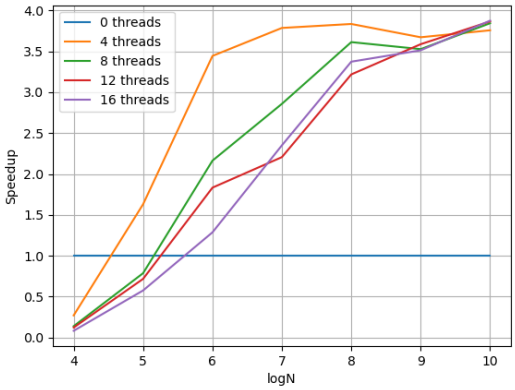
The speed up increases as the number of processes increases on which the code runs. However, if we increase the threads beyond a certain threshold than the cost to create and synchronize threads will become significant giving poorer results. As the problem size increases initially the parallel threads have a speedup less than one due to parallel overhead and pipeline not being utilised efficiently, however as size increases the speed increases because the pipeline is now being used efficiently. It is observed that the speedup initially remains below 1 and serial approach is better in such cases. As the problem size increases, the speed up reaches closer to 4.



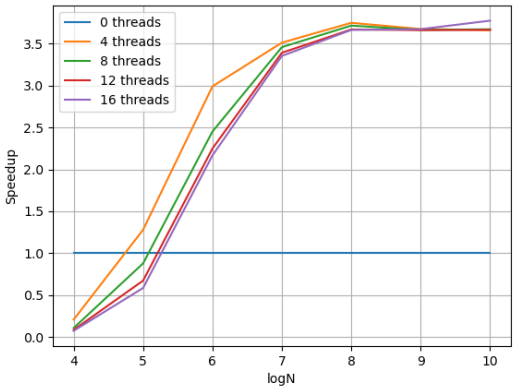
(a) Blocked Static



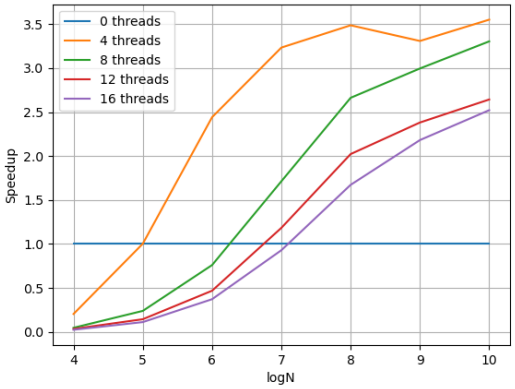
(b) Block Dynamic



(c) Outermost Static



(d) Outermost Dynamic



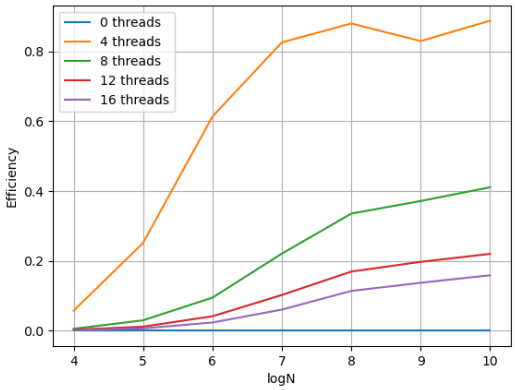
(e) Middle Static

Figure 8: Speedup

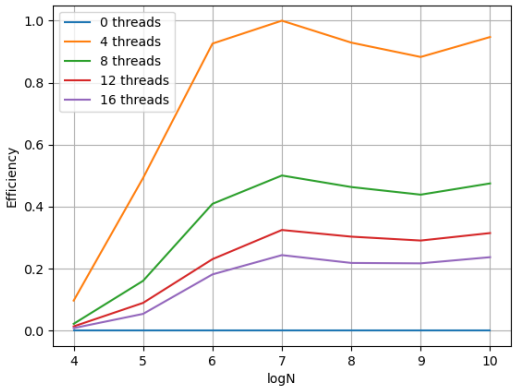
We, see that dynamic scheduling gives better speedup for the larger problem sizes because since the scheduling is done dynamically, distribution of the workload is efficient. On the other hand, for smaller problem size, static scheduling will give better result because of the significant overhead for dynamic scheduling which will reduce its performance.

3. Efficiency Curve Related analysis:

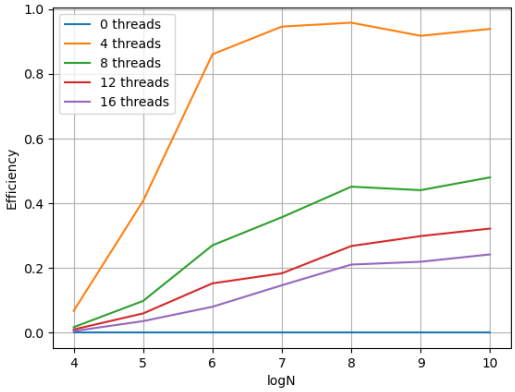
Efficiency basically tells us the Average work done per unit thread. For less number of threads, the commutation overhead is negligible and hence we observe maximum efficiency. As the problem size increases the efficiency also gradually starts increasing for a given implementation and gradually reaches 1.



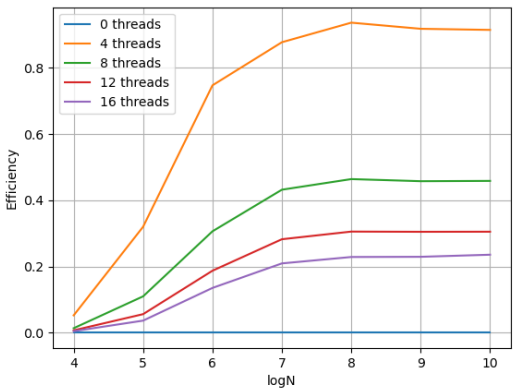
(a) Blocked Static



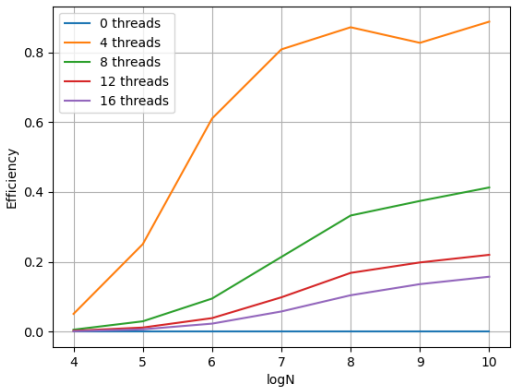
(b) Block Dynamic



(c) Outermost Static



(d) Outermost Dynamic



(e) Middle Static

Figure 9: Efficiency