

# 2D - 3D State Heat Equation

HPC Final Project

Purvik Shah - 201401417

Maharshi Vyas - 201401414

# Problem Description:-

The physical region, and the boundary conditions of a plate, are suggested by this diagram;

$$I = 0, W=0$$

$$[0][0] \text{-----} [0][N-1]$$

$$W=100 \mid \quad \quad \quad \mid$$

$$J = 0 \mid \quad \quad \quad \mid J = N-1, W=100$$

$$\mid \quad \quad \quad \mid$$

$$[M-1][0] \text{-----} [M-1][N-1]$$

$$I = M-1, W=100$$

The region is covered with a grid of M by N nodes,  
and an M by N array W is used to record the temperature.

The correspondence between array indices  
and locations in the region is suggested by giving the indices of the four  
corners.

The steady state solution to the discrete heat equation satisfies the following condition at an interior grid point:

$$W[\text{Central}] = (1/4) * ( U[\text{North}] + U[\text{South}] + U[\text{East}] + U[\text{West}] ) \text{ (For Plate)}$$

$$W[\text{Central}] = (1/6) * ( U[\text{North}] + U[\text{South}] + U[\text{East}] + U[\text{West}] + U[\text{Upper}] + U[\text{Lower}] ) \text{ (For Cube)}$$

Assuming U is the current state and W is the next state



# Pseudo Code:-

- Allocating space for the Cube (3D array) (Old and New)
- Setting the boundary value for all 12 boundaries
- Averaging the Boundary value to come up with mean
- Initialize the interior solution to the mean value
- While epsilon is less than diff,
  - Saving the old solution in 3D array u
  - Determining new estimate of the solution by taking average of its neighbours from old solution
  - Determining difference with  $\text{diff} = \text{fabs} ( w[i][j][k] - u[i][j][k] );$



# Input, Output, Time Complexity:-

## **Input:-**

Error tolerance for a fixed plate/cube size.

Temperature values for boundaries.

## **Output:-**

Number of iterations required to reach the error tolerance, time.

Final solution at each point of the plate/cube meaning temperature of each point on the grid is calculated as output.

## **Time Complexity:-**

$O(M*N*K*No\_of\_iterations)$

Where M,N,K are dimensions of the Cube. For plate, K=1.



# Parallelization Strategy:-

## Determining the Loop to be parallelized:-

Loop Structure - We are running 3 loops iterating to 3 dimensions (For M,N and K respectively) of cube to access memory. There is no swapping between loops as this would be cache optimized because of row major architecture.

```
for( i = 0 ; i < M; i++ )
```

```
    for( j = 0; j < N ; j++ )
```

```
        for( k = 0; k < K; k++ )
```

```
            W[i][j][k] = ( u[i-1][j][k] + u[i+1][j][k] + u[i][j-1][k] + u[i][j+1][k] + u[i][j][k+1] + u[i][j][k-1] ) / 6.0;
```

Decided to parallelize outermost loop because parallelizing any other loop would have meant significantly increased overhead.

# Parallelization Strategy:-

## Parts that are parallelized:-

1. Initialization of all points on Boundary  
#pragma omp for directive to divide the task
2. Averaging boundary values as mean, for reasonable values of interior points  
#pragma omp for reduction with addition operator on *mean* variable  
#pragma omp for directive to divide the task of assigning values to interior points.  
(Strategy Described in previous slide is used here)
3. Iterate until the difference between new solution and old one is greater than error  
Determining new solution by averaging old solution: Static, Dynamic, Auto Scheduling used  
(Strategy Described in previous slide is used here)  
Finding the minimum difference between old and new solution.  
#pragma omp critical used to achieve correct result.(To handle the race condition)

# Issues in Parallelization

1. We can not normalize mean in the parallel region. Since it can only get the correct value after all the threads are adding their respective values to it. To overcome this, we create two parallel regions with normalized mean in between.
2. We can't use difference as a variable which can simultaneously be used by all threads. Therefore, we define a private variable `th_diff`. All threads have their own copy of it. We use critical section to update the value of `diff`. Which is done once per iteration.
3. Determining whether fusion or fission would be better while measuring value of new solution by taking average of the nearby points in the old solution.

**Loop Fission:-** First line cannot be optimized for cache use, while second line can. So fused loop is not cache optimized.

But after fission, second loop is cache optimized (Don't have cache misses in each iteration).

- 1)  $w[i][j][k] = u[i-1][j][k] + u[i+1][j][k] \dots\dots\dots$
- 2)  $\text{Diff} = \min(\text{diff}, \text{fabs}(w[i][j][k] - u[i][j][k]))$



# Scheduling and Load Balancing:-

There is no dependency between data handled by each thread.


No need of synchronization between threads except one race condition.

Work assigned to each threads are spatially distant. So no false sharing or cache coherence. So, there will not be cases where threads remain idle because of this.

Have to check for cases when some threads are doing more work than others i.e load imbalance.

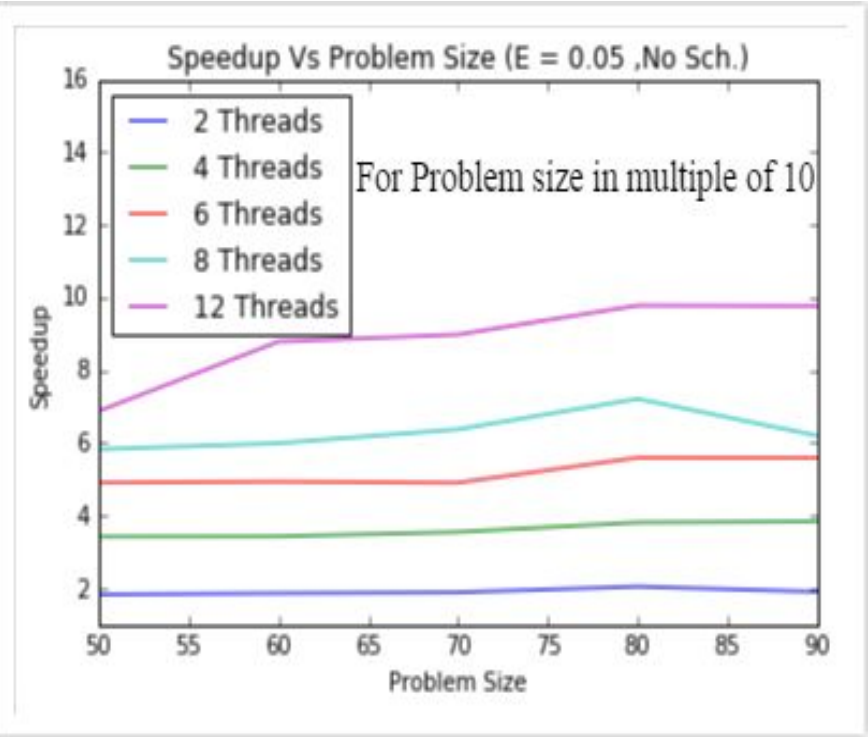
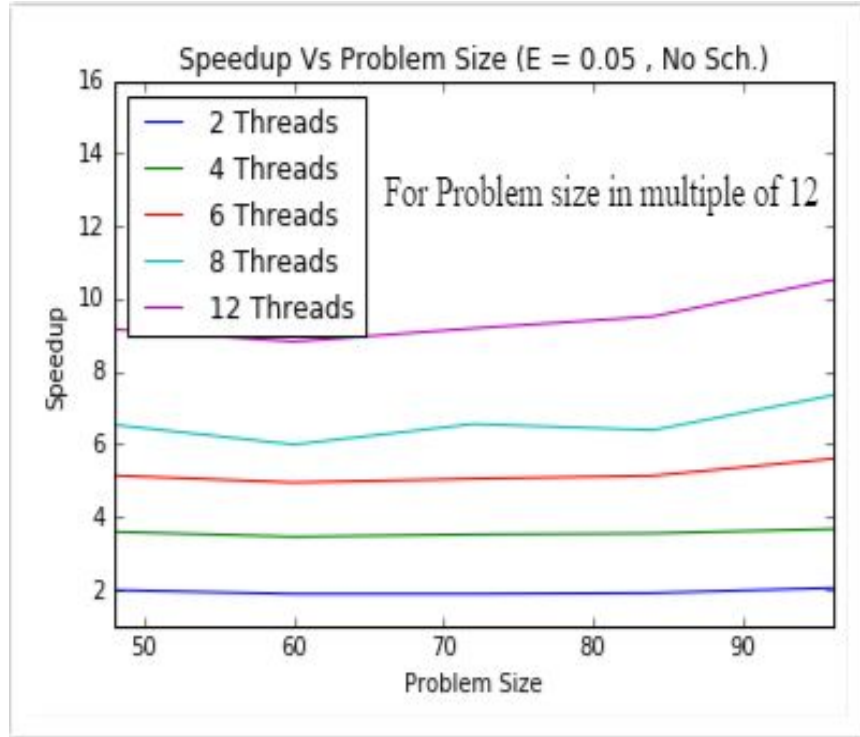
Problem size varies from 50 to 100 for cube, which is not much larger than number of threads. And each imbalanced have  $O(N \cdot K)$  amount of work, which can cause a decrease in speedup.

So we tried various scheduling and checked load imbalance with different kind of problem sizes.

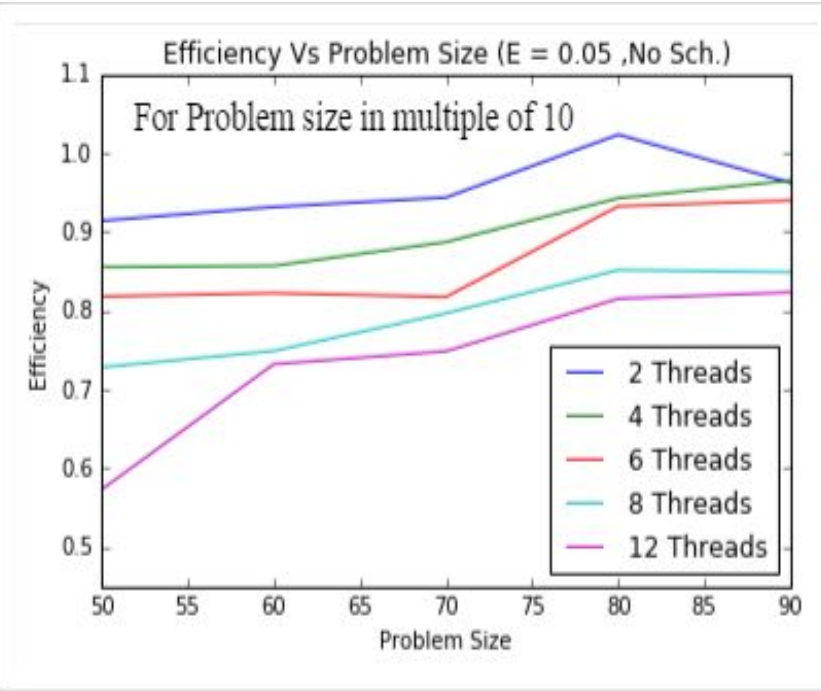
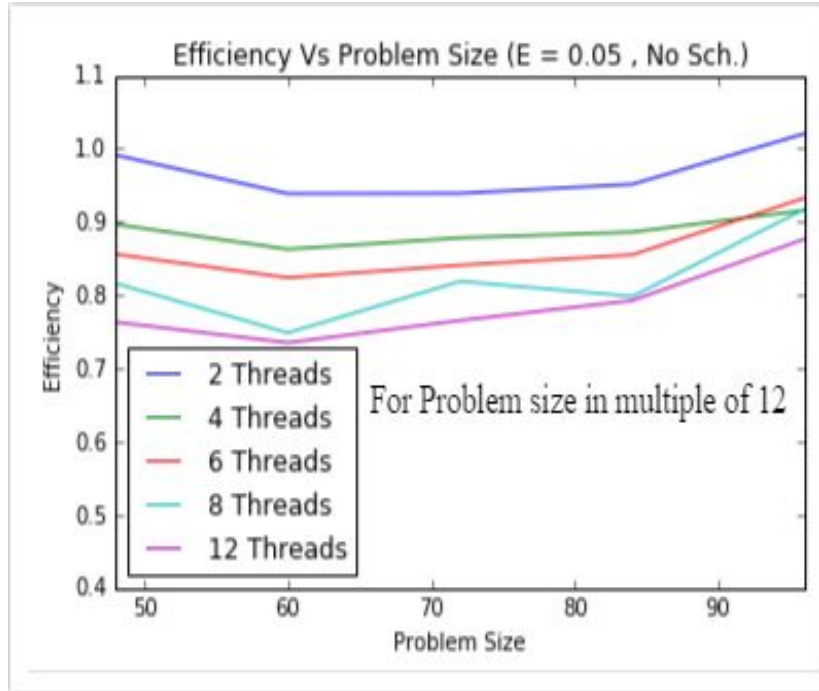




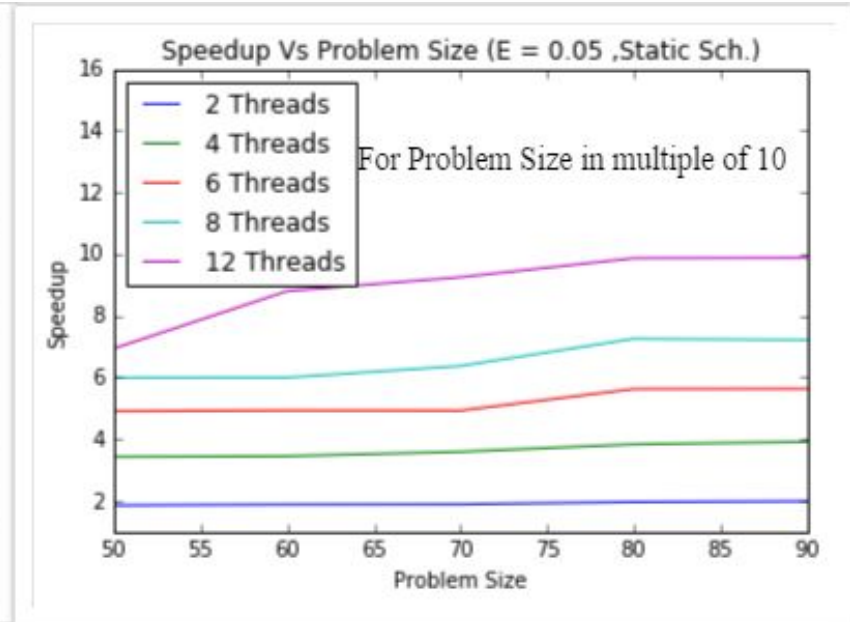
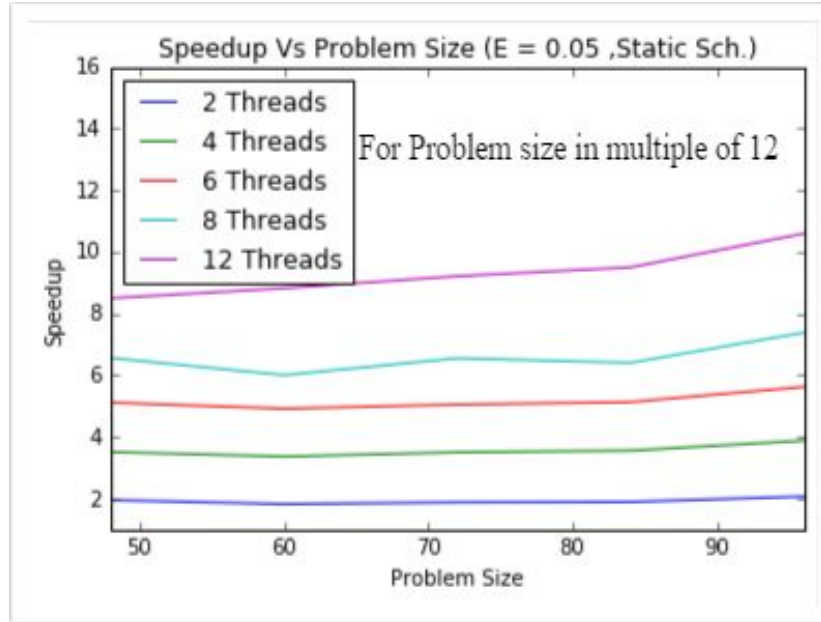
# No Scheduling Speedup



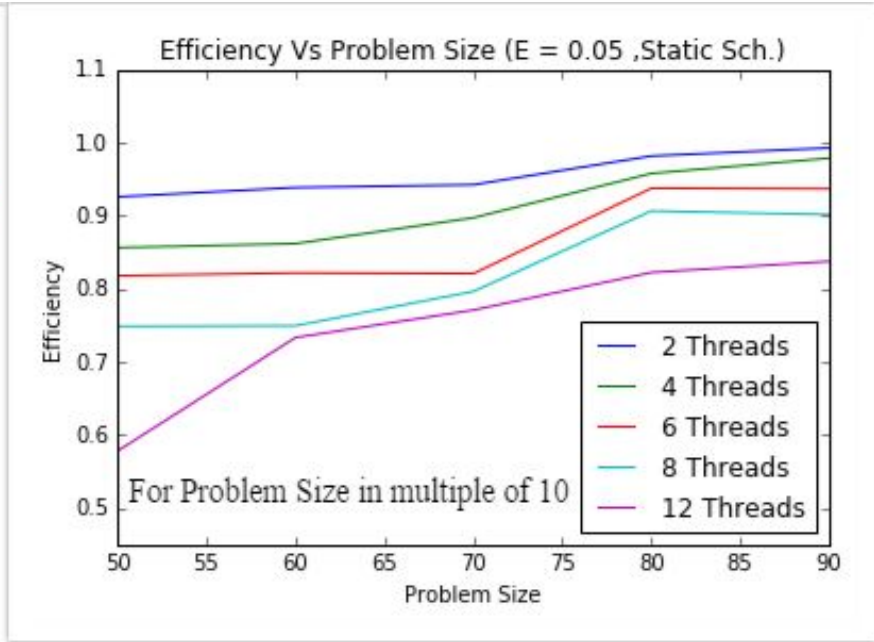
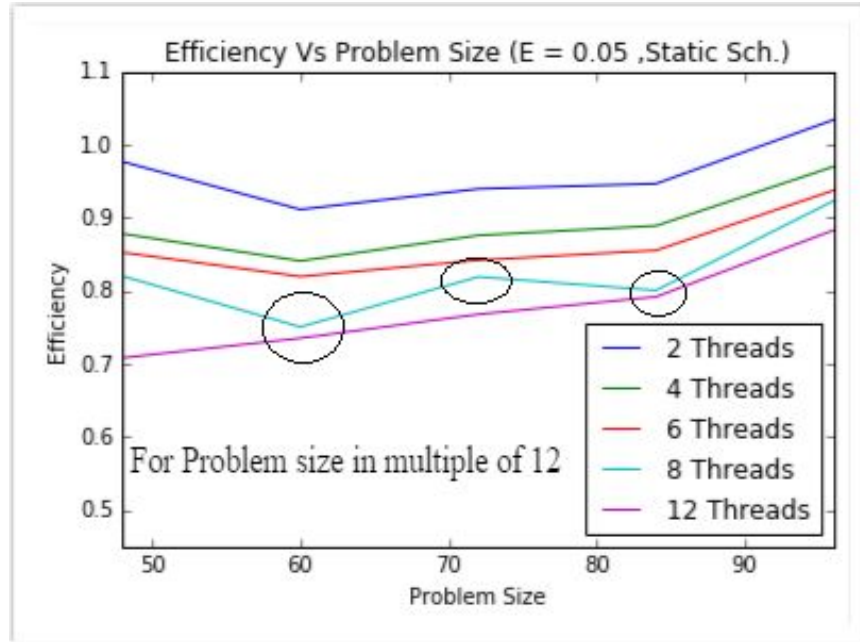
# No Scheduling Efficiency



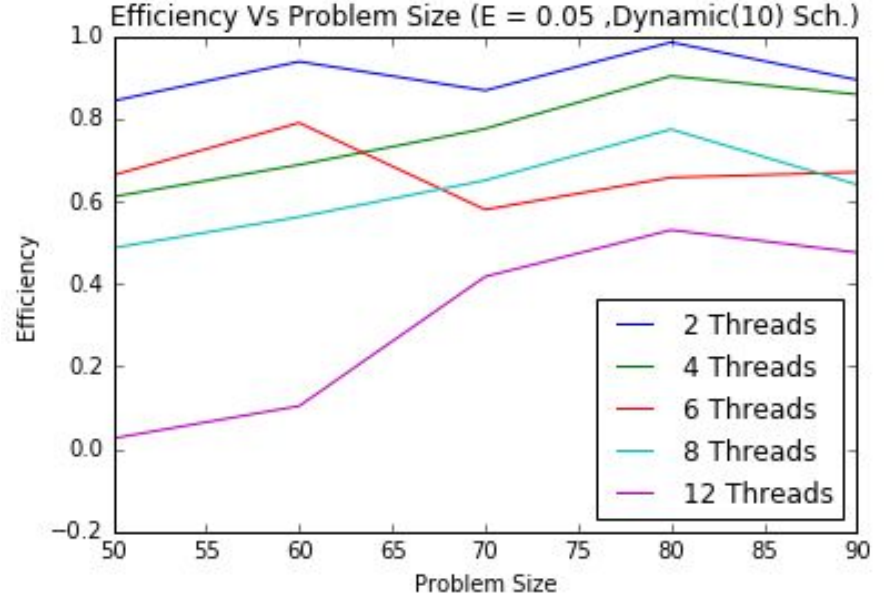
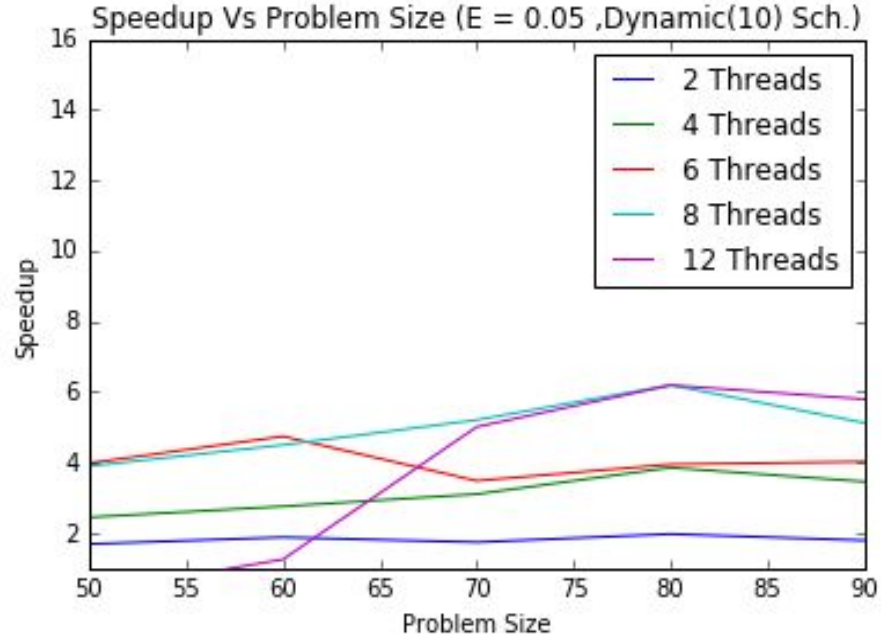
# Static Scheduling Speedup



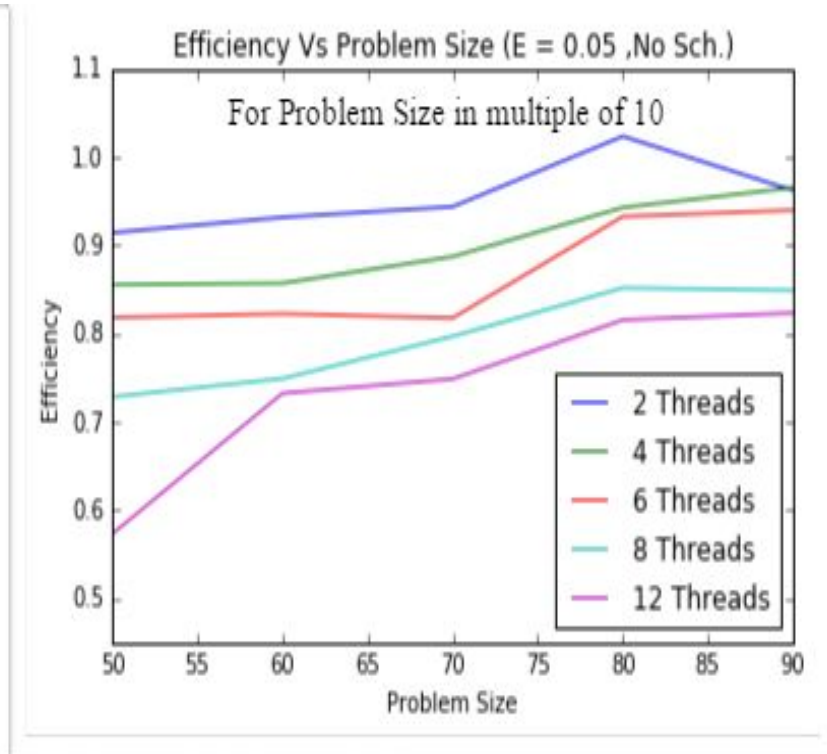
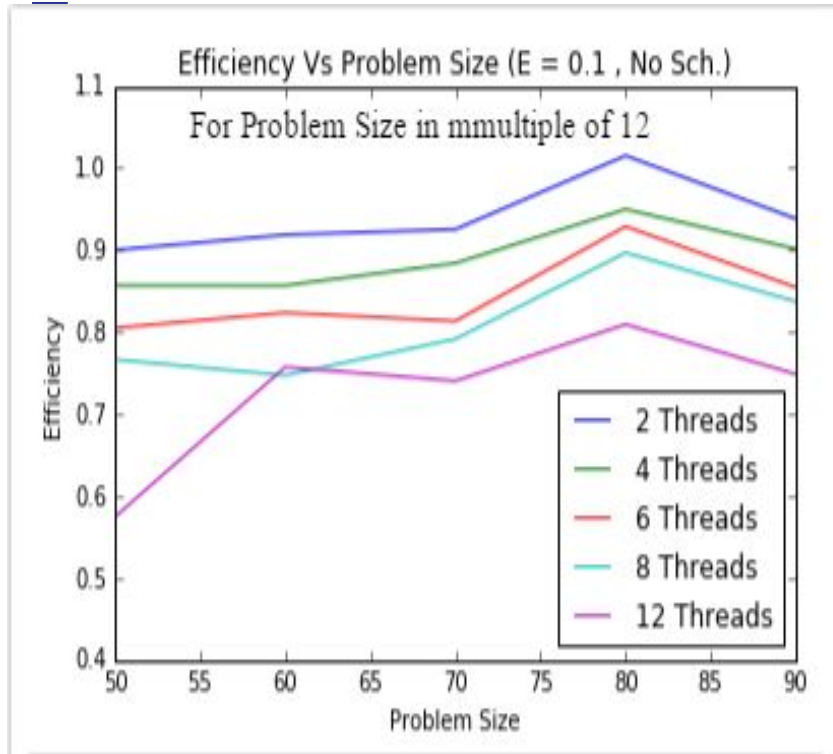
# Static Scheduling Efficiency



# Dynamic Scheduling



# Effect on Efficiency because of Variations in



# Conclusion:-

- As we lower tolerance, computation increase, scope for parallelism increases. Hence speedup too increases.
- Also, if we can divide the work equally among the threads and there is no load imbalance, it helps in speedup.
- Loop fission works better than Loop fusion in this case.
- The problem does not have any data dependency. It is also embarrassingly parallel. Hence we get high speedups for all number of threads for increasing problem size. The efficiency is almost always greater than 0.8

