2D - 3D State Heat Equation

Purvik Shah - 201401417
Maharshi Vyas - 201401414

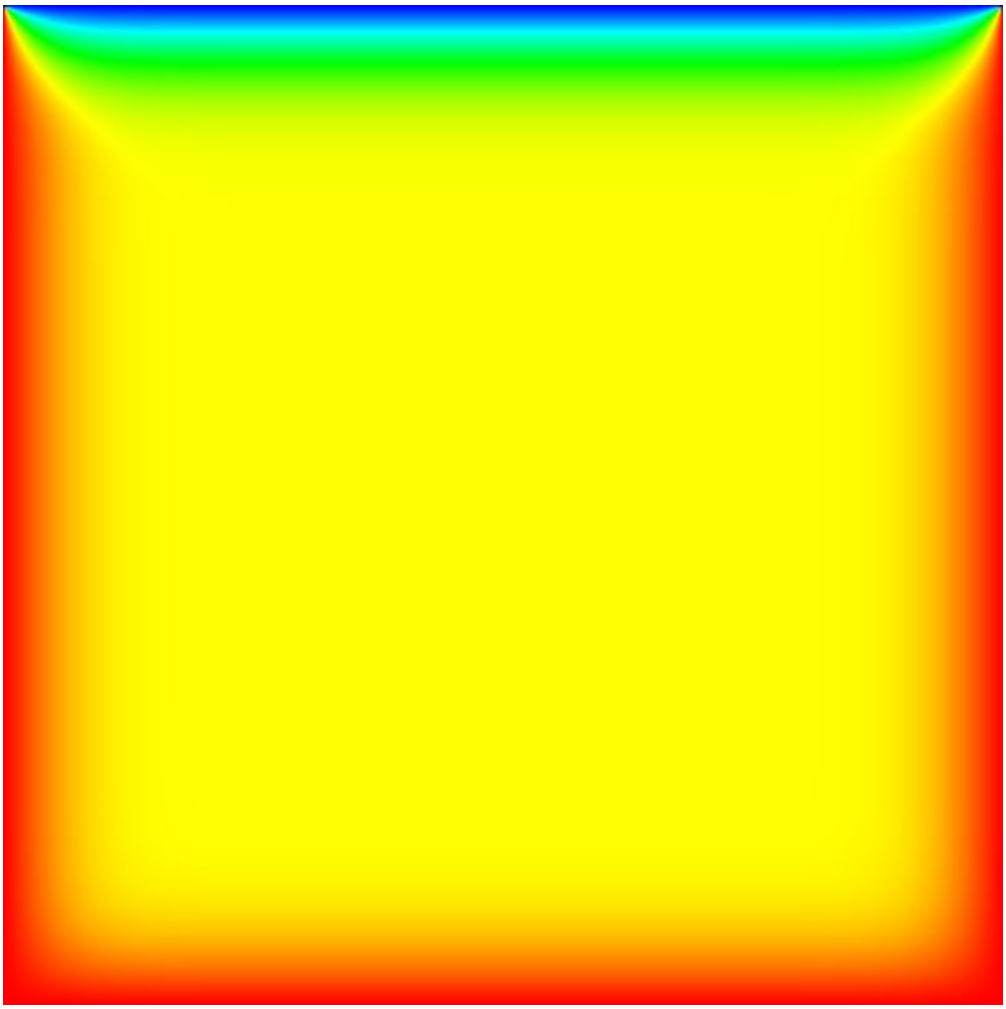
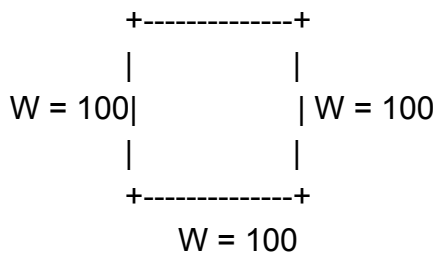


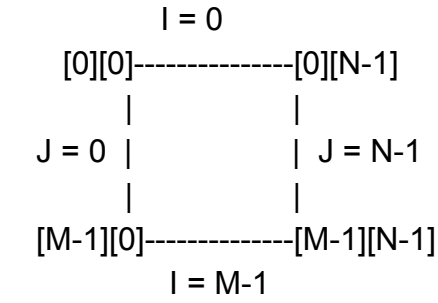
Image of Heated Plate in Steady State

Description:-

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



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:
Assuming U is the current state and W is the next state
W[Central] = (1/4) * (U[North] + U[South] + U[East] + U[West]) (For Plate)
W[Central] = (1/6) * (U[North] + U[South] + U[East] + U[West] + U[Upper]+ U[Lower]) (For Cube)

where "Central" is the index of the grid point, "North" is the index of its immediate neighbor to the "north", and so on.
Given an approximate solution of the steady state heat equation, a "better" solution is given by replacing each interior point by the average of its 4 neighbors - in other words, by using the condition as an ASSIGNMENT statement:

W[Central] <= (1/4) * (W[North] + W[South] + W[East] + W[West])

If this process is repeated often enough, until the difference between successive estimates of the solution will go to zero.
This program carries out such an iteration, using a tolerance specified by the user, and gives the final estimate of the solution.

We can achieve similar results with a Cube instead of a plate. That would change the 2D equation to 3D.
Adding a parameter of height (or depth) to the plate will be equivalent to the Cube.
3D State Heat Equation would be-
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

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 `diff = fabs ( w[i][j][k] - u[i][j][k] );`

Profiling:-

The code does not have multiple functions, so all of the time is spent in the main function.
Most of the time is spent in the while loop of (epsilon < diff) , since it contains the highest amount of computations and rest are just initializations. So we need to parallelize loops in each iteration mainly.

Input Parameters:-

Here, input parameter is the error tolerance for a fixed plate/cube size. We can change the size of the plate/cube to obtain other results. As error decreases (Accuracy increase), number of iterations taken to reach the answer will increase. So Computations increase linearly with each iteration.

Apart from that, we specify the number of threads to be used by the parallel version of the program.

We can modify the temperatures given to the edges of the cube or plate, or we can give different temperatures to some regions in the cube, but it will not affect the algorithm or the parallelization process computationally. So for the sake of simplicity (where computations and work are equal) we have given values 100 and 0 to each borders of the cube. Bottom 3 edges are at 0 for the cube and and other edges are at 100.

Output:-

We get the number of iterations required to reach the error tolerance and the time it took for that. We also get the solution which has been computed at each point of the plate/cube i.e. temperature of each point on the grid is calculated as output.

Algorithmic Complexity and how problem is computationally expensive:-

It would be O(M*N*K*No_of_iterations)
Where;
M, N, K are the size parameters of the Cube.
For plate, K=1.
No. of iterations are the iterations required to achieve the tolerance.
We can get a maximum speedup of p.

Hardware Configuration:-

Architecture: x86_64
Byte Order: Little Endian
CPU(s): 16
On-line CPU(s) list: 0-15

Thread(s) per core: 1
Core(s) per socket: 8
Socket(s): 2
CPU MHz: 2001.000
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 20480K

Memory Allocation and Memory Access:-

We have allocated our data different from regular allocation. In case of a plate(M*N dimension), where normal array is allocated linearly and contiguously, we have allocated each row of N length separately in memory. So there are M pointers pointing towards M rows of the plate. In case of cube, there are M*N rows of K length are allocated separately. So M*N pointers pointing to each row. Because we have row major architecture, contiguous blocks from each row will be taken in cache.

Using this method of allocation can cause a cache miss at each of the last point in a row, so there will be O(M) and O(M*N) more cache misses (respectively for plate and cube) then contiguous memory allocation which is not a major scale compared to O(M*N) and O(M*N*K).

```
double *u[M][N];
  for (i=0; i<M; i++)
    for(j=0; j<N; j++)
      u[i][j] = (double *)malloc(K * sizeof(double));

  double *w[M][N];
  for (i=0; i<M; i++)
    for(j=0; j<N; j++)
      w[i][j] = (double *)malloc(K * sizeof(double));
```

Parallel Part of the Code:-

Determining the Loop to be parallelized:-

Loop Structure: We are running 3 simple loops iterating to 3 dimensions(For M,N and K respectively) of cube to access memory. Memory allocation is contiguous for 3rd dimension, so memory access is done through the third loop because of the row major architecture. So there is no need of swapping loops or make transpose of the matrix or any such things. This would be the most optimized use of Cache because of row major architecture.

We decided to parallelize the outermost loop(For both cube and plate). Because,

1. Parallelizing any other loop would have meant significantly increased overhead since using #pragma omp parallel for, for inner loops initialize threads all the time(O(M*N) if third loop is parallelized) which will result in a large total of parallel overheads. And if only we keep all the threads launched and use #pragma omp for, for third loop, some part of the computations will run on all the threads uselessly.
2. Outermost loop has the most workload per iteration. It made sense for that workload to be divided rather than workload for other loops which is significantly less.

Parts that are parallelized:-

1. Initialization of all points on Boundary
2. Averaging boundary values as mean, for reasonable values of interior points
3. Iterate until the difference between new solution and old one is greater than error

Parallelization Strategies:-

For Initialization,
Using #pragma omp for directive to divide the task of assigning values among threads

For Averaging Boundary Values,
Using #pragma omp for reduction with addition operator on *mean* variable

To assign the interior values equal to *mean,* and assign values of the new solution to the old solution.
Using #pragma omp for directive to divide the task of assigning values among threads: Static, Dynamic, Auto Scheduling used

Determining new solution by averaging old solution: Static, Dynamic, Auto Scheduling used.

While difference between new solution and old one is greater than error,
Saving the old solution in variable *u:* Static, Dynamic, Auto Scheduling used

For every topic above, there is no dependency between data handled by each thread. and also there is no need of synchronization between threads. Work assigned to each thread is independent of each other by memory allocation also, because our memory allocation is not necessarily contiguous. So there will not be any false sharing or cache coherence. So, there will not be these type of cases where threads remain idle.

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. This also results in less computation.
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.
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.

Handling Race Condition:

There is a variable diff which is minimum difference between old and new solutions. So, Creating a new variable th_diff for each thread to determine the minimum difference calculated by each thread, and finally using #pragma omp critical to determine minimum among th_diff which will be diff in this case.
Here only when it comes to #critical, there may be times when a thread remains idle, but this happens only once in an iteration (no. of iterations << computational complexity), which is negligible.

```
# pragma omp critical
{
  if ( diff < th_diff )
  {
    diff = th_diff;
  }
}
```

Instead of creating a new variable, we could've kept the diff variable and put the critical section inside the third for loop. However, that would've caused a large portion of the code to run in Serial. So, we optimized it by introducing a new diff variable.

Loop Fusion and Fission:

Fused Loop:

```
for ( i = 1; i < M - 1; i++ )
{
  for ( j = 1; j < N - 1; j++ )
  {
     for ( k = 1; k < K - 1; 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;
              if ( th_diff < fabs ( w[i][j][k] - u[i][j][k] ) )
              {
                  th_diff = fabs ( w[i][j][k] - u[i][j][k] );
              }
     }
  }
}
```

We are splitting this loop here. After splitting, averaging and assigning w[i][j][k] will be done in first loop and finding difference and checking minimum will be done in second loop. Loop fission is giving better results as compared to fused loops. And that can be because:

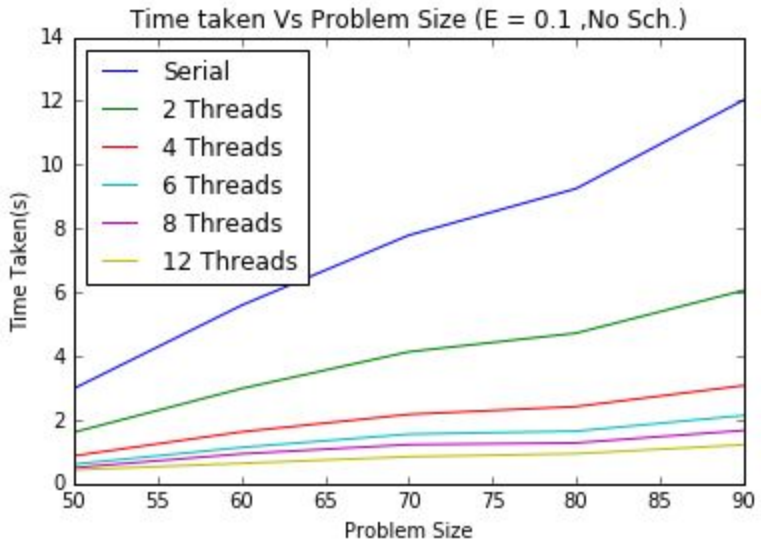
In fused loop 6 differently placed points are used. They are not spatially local, so cache use is not optimized in each iteration. But after fission, the second loop will have blocks of w in cache.So it will be cache optimizing. First loop still has cache misses, but it cannot be changed. By fission, we are saving cache misses for w[i][j][k](in the second loop) for each iteration, which saves some time.

This fused loop is used in each iteration. So optimizing this loop will affect final results as memory access time will decrease in total iterations number of times.

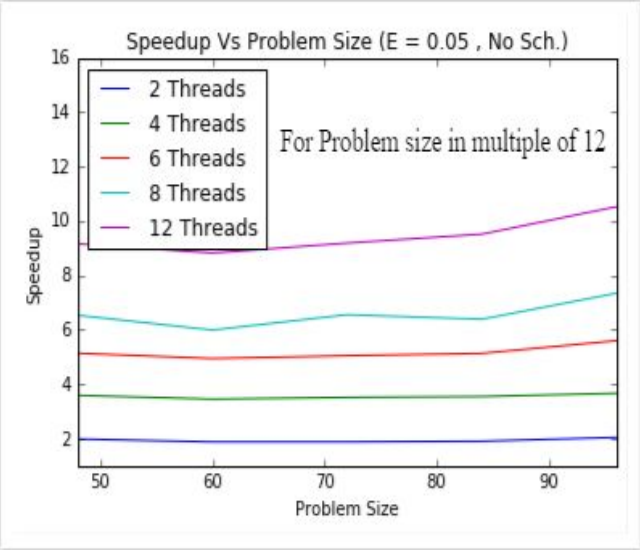
Schedulings and Load Balancing:-

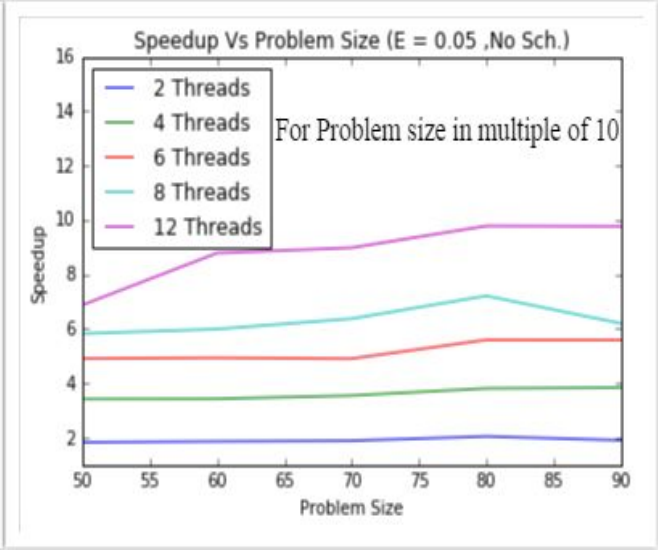
Time taken for memory access time is nearly same of each thread, and cases of threads remaining idle also cannot be optimized more. So we have to check for cases when some threads are doing more work than others i.e load imbalance. So we have done various type of scheduling and checked the cases of load imbalance. We have taken readings for a cube(i.e. 3D array) ,where M = N = K = Problem Size.
Increasing problem size(M) increases memory and computations with O(M^3). We have taken problem sizes(M) from 50 to 100,which are very much comparable to number of threads. We kept them first in multiple of 12, and then in multiple of 10, and we are observing for threads 2,4,6,8,12.
Load imbalance also happens in case of 2D,but problem size is very large than number of threads and imbalanced threads also does not have much amount of work as compared to 3D problem. So load imbalance hides behind the increased speedup. So no need to check all this for that in 2D.

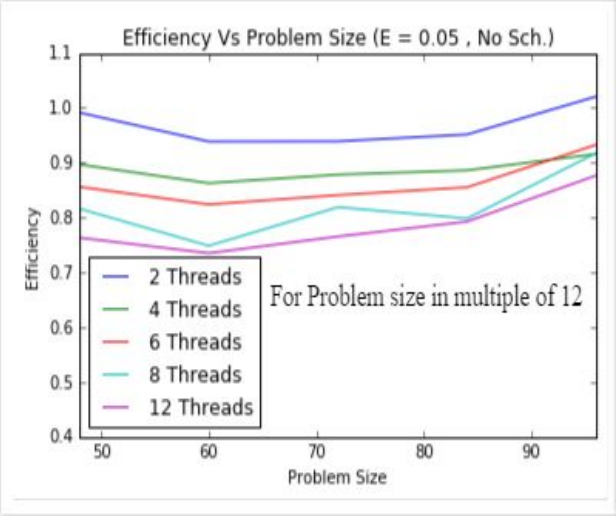
Parallelization without any Scheduling:-

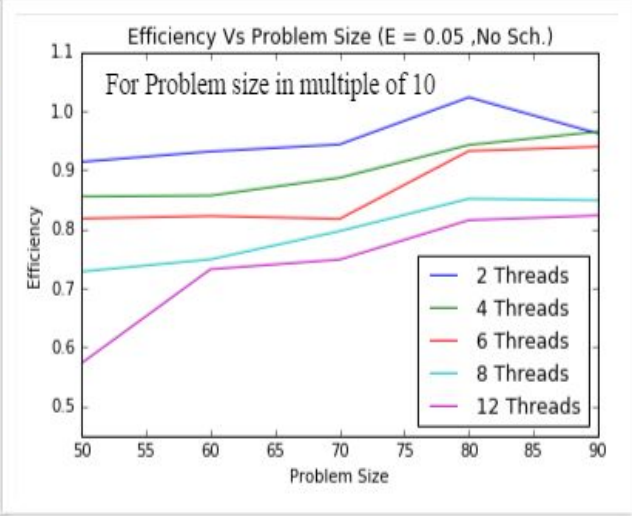


Clearly, serial time is more than the parallel time.



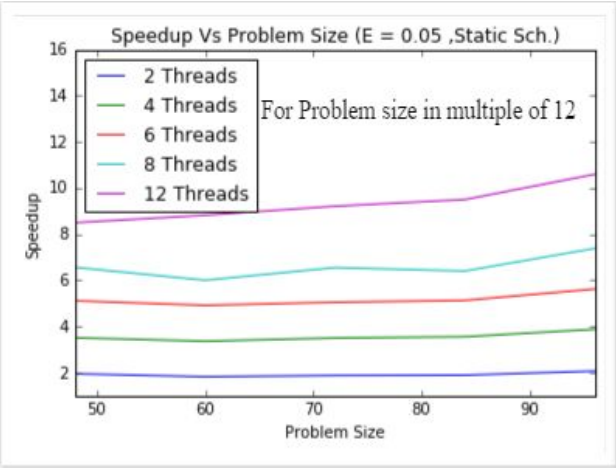


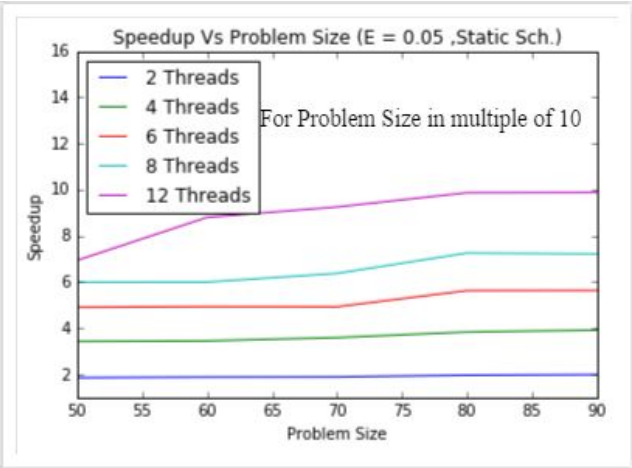


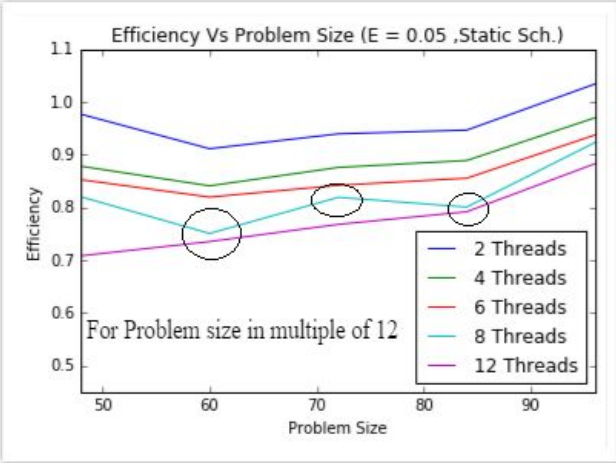


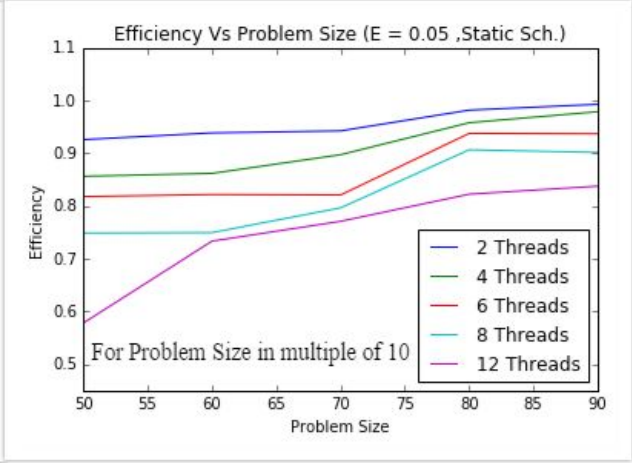
When there is no scheduling, there is normally an increase with increase in problem size for multiple of 12s. But there are some anomalies (There are more ups and downs which are clearly seen for 12 threads) for multiple of 10s. And that can be because those cannot be divided equally between each thread (For 4,6,8,12) and for each iteration there will be some imbalanced threads having work in O(M^2).

Static Scheduling:-









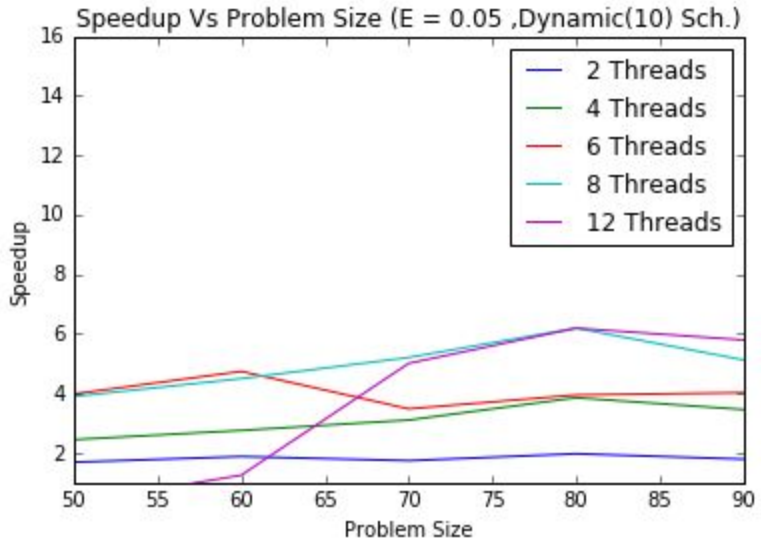
Having parallelized the outer loop, work given to each thread is always equal (i.e. O(N*K) - to handle their respective y and z dimensions.) and independent. So static scheduling gives the best results, where work is divided directly to threads at compile time. This means there is less overhead time than other types of scheduling.

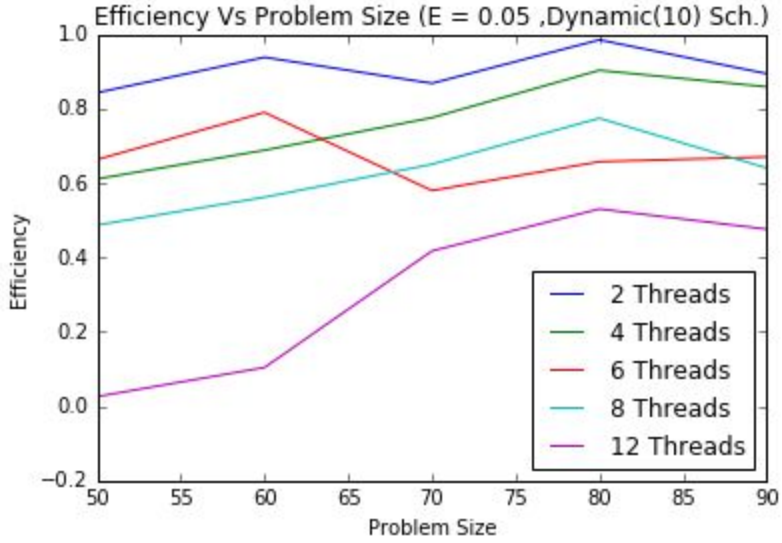
The behaviour of speedup and efficiency are same as no scheduling.

In case where problem size is of multiple 12s, 2,4,6 and 12 threads equally divide problem size, but 8 does not. So efficiency of 8 does not increase linearly, it has anomalies. Efficiency for 8 threads is higher when problem size is in multiple of 8

(48,72,96) and lower when it's not (60,84). This happens only in case of 8. So, this clearly shows that there is a load imbalance in case of 8 threads.

Dynamic Scheduling:-





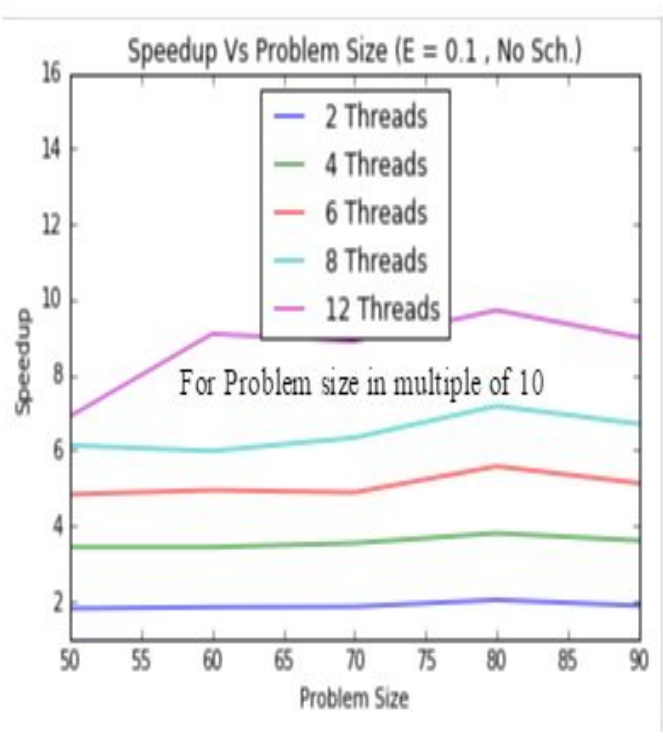
There is a significant decrease in performance of dynamic scheduling (chunk size of 10), especially in case of more threads. Dynamic scheduling gives equal results in case of 2 threads. Also, for 4 and 6 threads, results are not that much diminished. But for 12 threads, there is a significant decrease.This can be overcome if we decrease chunk size, but only for some extent. Overall performance for smaller chunk size are much lower because of the overhead related to dynamic scheduling. We don't get a proper

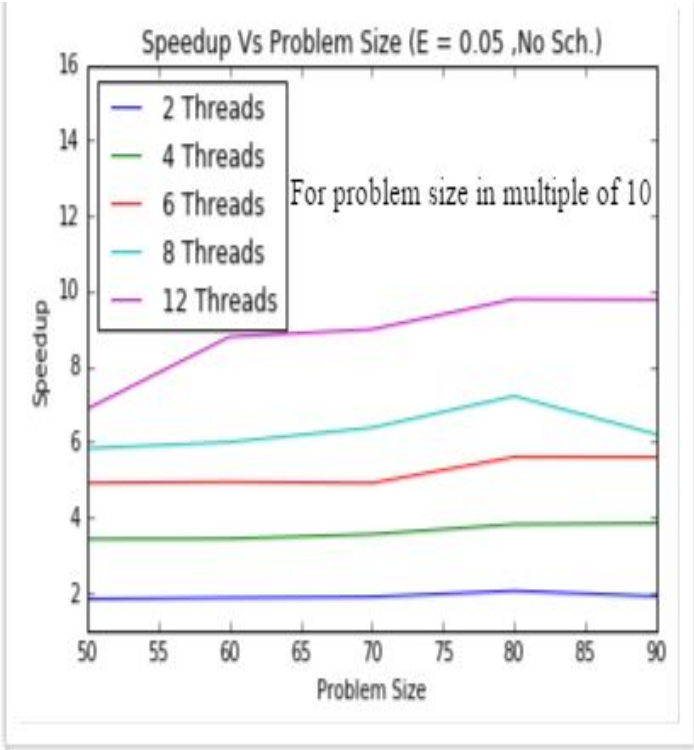
bargain for this increase in parallel overhead in terms of increase in speedup.Therefore overall static scheduling is better.

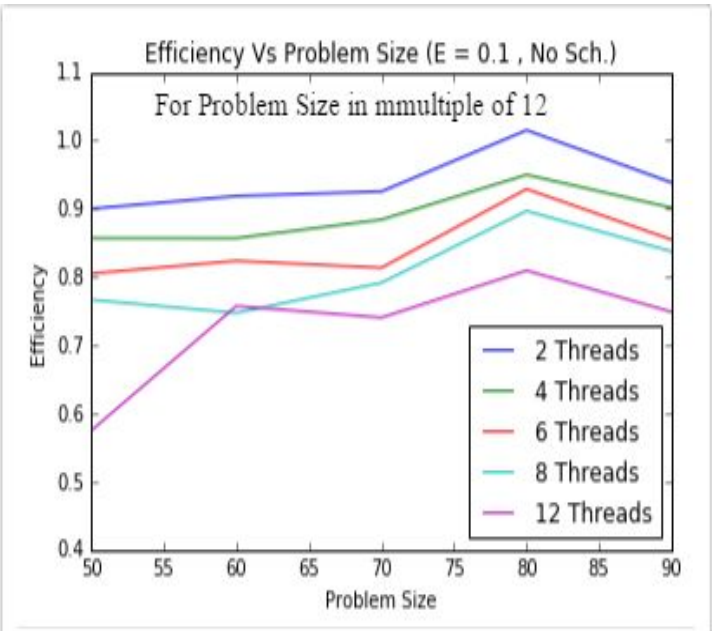
Effect on Speedup because of Variations in Error:-

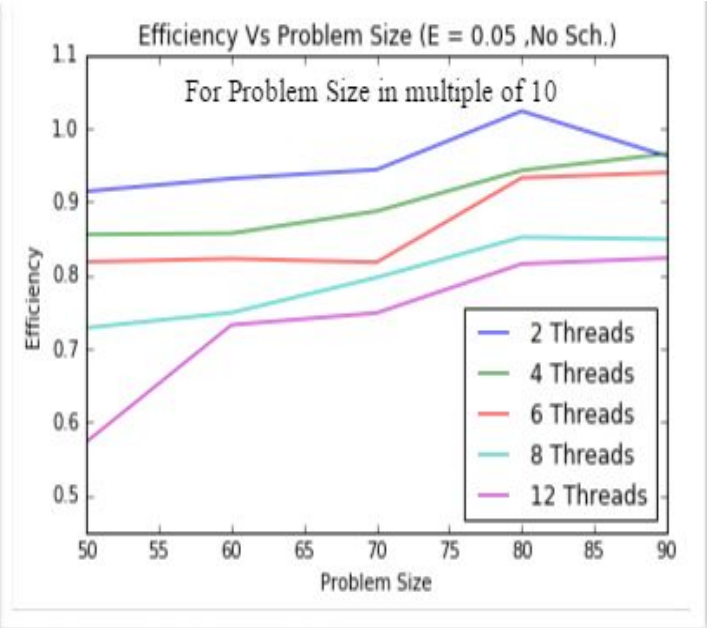
If we decrease the margin of error, total number of computations (Work Done) increases.
This is because the number of iterations required to reach the error increases significantly which has a lot of impact on time complexity. With each iteration new solutions are calculated which takes to most amount of run-time and which is parallelized.
So,the part of time that can be parallelized increases.

Hence, the speedup increases when we decrease the value of error.



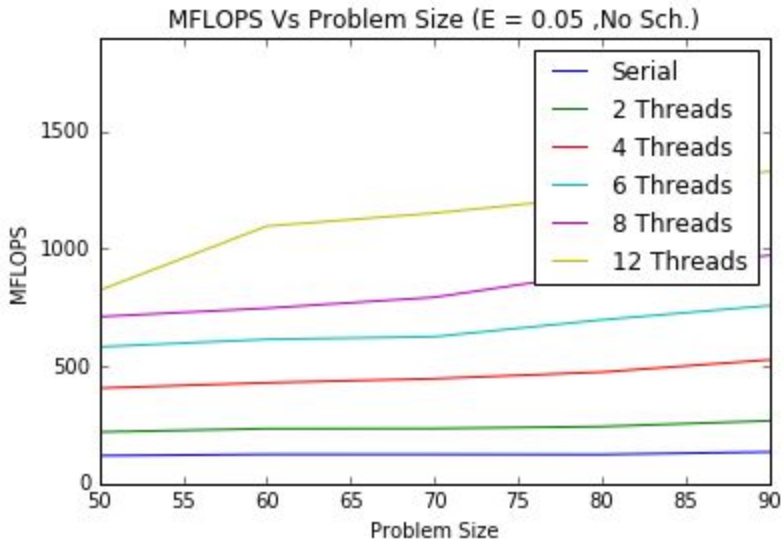






As seen from the graph,efficiency values of error 0.05 are slightly greater than efficiencies of error 0.1. This difference is greater when we decrease error to 0.001,taking more time to run because of increase in computations.

Performance measure in terms of MFLOPS:-



MFlops are increased as threads are increased.Because number of computations done per cycle increase with increase in threads.

Comparison with 2D:-

We ran the code for a plate as well. It showed same trends in speedup and efficiency. Increasing M and N gave higher speedups. Efficiency stayed above 80% as well. Results for 2D problems are attached in the final zip file.

Conclusion:-

We can see from the above plots that:-
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.
Hence we get high speedups for all number of threads for increasing problem size. The efficiency is almost always greater than 0.8