High Performance Computing
Project
Parallelization of Gauss Elimination Method using OpenMP

Jahnavi Suthar(201301414) Deeksha Koul(201301435)

Hardware Details:

Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 32
On-line CPU(s) list: 0-31
Thread(s) per core: 2
Core(s) per socket: 8
Socket(s): 2
Vendor ID: GenuineIntel
CPU MHz: 1200.000
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 20480K

Problem Introduction:

Gaussian elimination is an algorithm which solves a system of n linear equations with n unknowns. All equations are non-scalar with respect to each other. This means that two equations that are multiples of each other cannot exist in the system that is to be solved.

For example:

$$
\begin{aligned}
a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + a_{03}x_3 &= b_0\\
a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1\\
a_{20}x_0 + a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2\\
a_{30}x_0 + a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3
\end{aligned}
$$

This system of linear equations can be rewritten as Ax = b in matrix notation, where 'A' is the matrix of coefficients, 'x' is the vector of unknowns for which we are solving and 'b' is the vector of RHS values.

The Gaussian Elimination algorithm converts this matrix and vector into an Upper Triangle Matrix and a vector of computed values. This permits the solving of the unknowns through a process known as *back-substitution*. This last step, back-substitution, is trivial and thus is not part of making Gaussian Elimination parallelizable.
An Upper Triangle Matrix is a matrix for which all diagonal values are '1' and all values below the diagonal are '0'.
e.g.

$$
\begin{bmatrix}
1 & * & * & * \\
0 & 1 & * & * \\
0 & 0 & 1 & * \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

Gaussian elimination converts our equation Ax = b into Ux = y, where U is an Upper Triangle Matrix and y is a new vector of equation values.
The **difficulty** in parallelizing Gaussian elimination is that calculating a new value in the upper triangle regions requires that all previous values of the upper triangle matrix be known.

Input:
Sequence of ‘n’ Linear equations with ‘n’ number of variables.
Output:
The solution of ‘n’ Linear equations i.e to find the value of ‘n’ unknown variables.

Applications:
The solving of dense and large scale linear algebraic systems lies at the core of many scientific and computational economic applications such as linear programming, computational statistics, econometrics and theory game. Many practical problems can be translated into a large scale linear algebraic system. Therefore, the parallel solving of dense and large scale linear algebraic systems is of great importance in the field of scientific computation. Other basic functions of Gauss - Elimination are:Computing determinants of a square matrix, Finding the inverse of a matrix, Computing ranks and base of any mxn matrix.

References:

1. Some Improvements of the Gaussian Elimination Method for Solving Simultaneous Linear Equations(http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6596232)
2. Parallel direct methods for solving the system of linear equations with pipelining on a multicore using OpenMP(http://users.uom.gr/~pmichailidis/jpapers/jpaper)

3. Gaussian Elimination
   (http://www.saylor.org/site/wp-content/uploads/2011/11/ME205-4.2-TEXT.pdf)

Serial Algorithm

Here is serial pseudo-code for Gaussian elimination so that the actual calculations and iterations can be examined.

```
for i=0 to N-1
        for j=i+1 to N-1
                /*Division Step*/
                cons = a[j][i]/a[i][i];
                a[j][i] = 0;
                for  k=i+1 to N+1
                        /*Elimination Method*/
                        a[j][k] = a[j][k] - cons * a[i][k];

        /*Back Substitution*/
        for i=0 to i=N-1
                x[i] =a[i][N];
               cons = 0;
                 for j=i+1 to N
                        cons += a[i][j] * x[j];
        x[i] -= cons;
        x[i] /= a[i][i];
```

Complexity:

For the elimination step, the outer most loop runs for i = 0 to N - 1, the middle loop runs (N-i-1)times and the inner most loop runs (N - i)times,thus total running time is nearly O(n^3).

In the back substitution step, both the inner as well as outer loop runs in O(n)running time respectively.Hence takes
O(n^2)time.
Thus,the total running time is of **O(n^3)**.

Scope of Parallelism and part of code that can be parallelized:

The difficulty in parallelizing Gaussian elimination is that calculating a new value in the upper triangle regions requires that all previous values of the upper triangle matrix be known. The algorithm descends a row of the equation matrix at each iteration. At this row it computes the new values of the row and its corresponding value in the solution vector. This new row and vector information is used to calculate all rows below it. Thus the difficulty in parallelizing Gaussian elimination is that the calculation of each row requires the calculation of all the rows that have come before it. Concurrency of operation is the overriding issue.

There is no parallel algorithm that can efficiently run the second part i.e back substitution as in this for calculating each variable it depends on the previous value So,there is loop dependency that can't be resolved but we try to parallelize the first part in such a way that the efficiency of algorithm increases.

For the first part , we can parallelize first section of code by primarily **implementing two methods:**

a. A trivial parallelization(Row Cyclic) of serial code done with static and dynamic scheduling(by changing the chunk size depending upon size of the input matrix)
b. A pipelined parallelization that attains higher levels of parallelization

Effect of increasing problem size on serial time :

Algorithms like the Gaussian elimination algorithm do a lot of arithmetic. Performing Gaussian elimination on an n by n+1 matrix typically requires on the order of $O(n^3)$ arithmetic operations. One obvious problem with this is that as the size of the matrix increases due to the growth in problem size ,the amount of time needed to complete Gaussian elimination(both the division and elimination step) grows as the cube of the number of rows. Hence as the problem size is increased more rows are added and hence the iteration over the elimination method as well as in back substitution increases overall thus increasing the computational work.

Parallel Algorithm

Version 1: Naive Parallelization

```
for i=0 to N-1
 #pragma omp parallel for num_threads(p) private(cons, k) shared(i)
//schedule(dynamic,bs)//schedule(static,bs)
        for j=i+1 to N-1
                /*Division Step*/
                cons = a[j][i]/a[i][i];
                a[j][i] = 0;
                for  k=i+1 to N+1
                        /*Elimination Step*/
```

```
                    a[j][k] = a[j][k] - cons * a[i][k];

/*Back Substitution*/
for i=0 to i=N-1
    x[i] =a[i][N];
    cons = 0;
    for j=i+1 to N
        cons += a[i][j] * x[j];
    x[i] -= cons;
    x[i] /= a[i][i];
```

In version 1, we can not parallelize the outermost loop using naive parallelization, because we must have done all the elimination operations on ith row before using it for further elimination. (e.g. We can not subtract 2nd row from 3rd row until we have not subtracted 1st row from the 2nd row). We can parallelize middle loop, i.e we can parallely subtract ith row from i+1th, i + 2th, … … … … , n-1th row, because it does not involve data dependency.

Version 2: Pipelined Parallelization

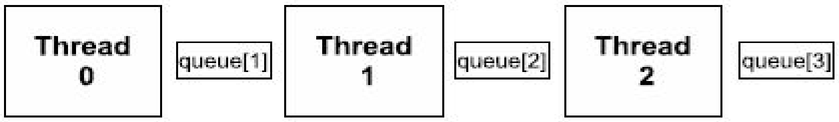


Fig. 1. Pipelining model.

An implementation of the pipelining technique shown in Fig.1 in OpenMP using the queue data structure. The queue is realized with the help of the proposed simple Get() and Put() procedures.

```
#pragma omp parallel private(cons, k, i, j) shared(a)
rank = omp_get_thread_num()
size =  n / p

if(rank != 0)
    for i= 0  to  rank * size
        row = Get(rank);
        Put(rank + 1, row)
        for j = rank * size  to (rank + 1) * size
            /*Division Step*/
            cons = a[j][row]/a[row][row];
```

```
                    for  k=row+1 to n+1
                        /*Elimination Step*/
                        a[j][k] = a[j][k] - cons * a[row][k];

 for i = rank * size  to  (rank + 1) * size
     Put(rank + 1, i)
     for j = i + 1  to (rank + 1) * size
         cons = a[j][i]/a[i][i]
         for k = i + 1 to n+1
             a[j][k] -= cons * a[i][k]
/*Back Substitution*/
for i=0 to i=N-1
    x[i] =a[i][N];
    cons = 0;
    for j=i+1 to N
        cons += a[i][j] * x[j];
    x[i] -= cons;
    x[i] /= a[i][i];
```

In the V1, threads had to wait after each iterations, for other threads to complete.

e.g. We are given system of 5 linear equations and 2 threads.
The 1st thread has already done assigned operations on 2nd and 3rd row, but 2nd thread is still doing its operations. Rather than waiting for 2nd thread to complete, the 1st thread can start subtracting 2nd row from the 3rd row, because 2nd row is not going to change in the iterations done by thread 2.

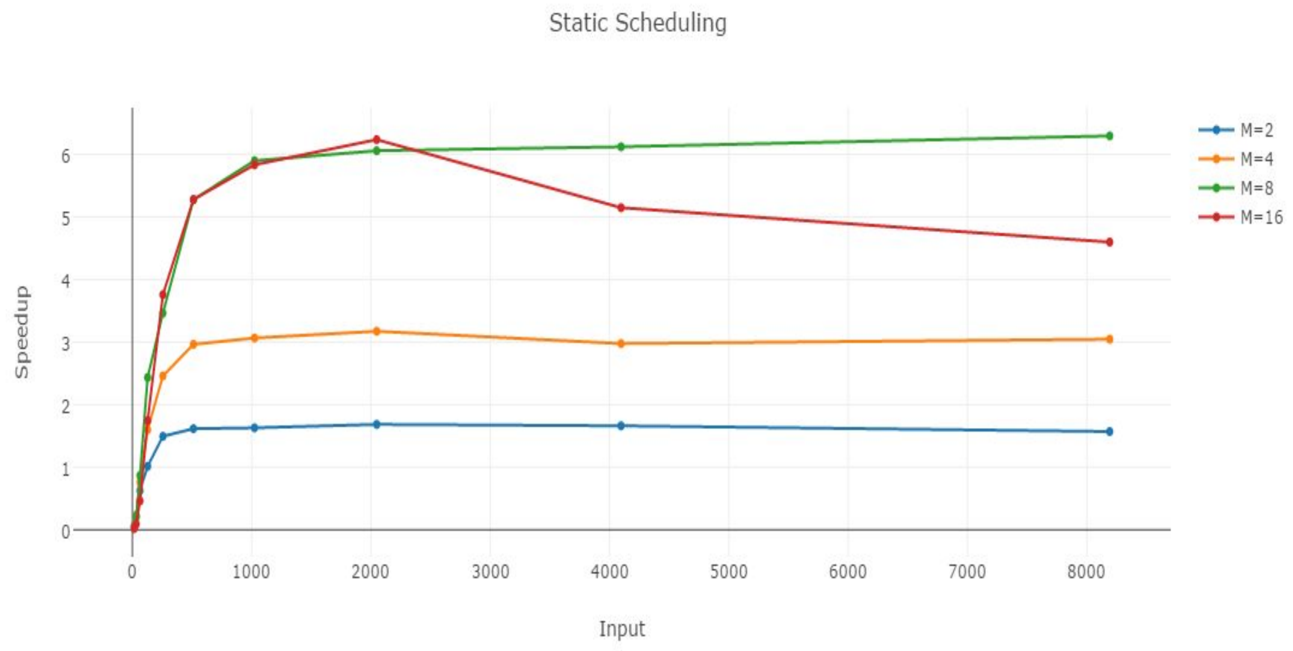
This can be realized using p queues, 1 for each thread. Thread with rank (i - 1) can enqueue and thread with rank i can dequeue items from ith queue.

In this version rows are equally divided among the threads on which they perform operations. Once the thread has done all the operations on the row assigned to it, it passes it to the next thread using queue. The thread also passes the index of rows received by it from the previous thread using the same row, and does the operations using it afterwards.

After the matrix is reduced to upper triangular matrix, we find the values of x_s using back substitution.

Results and discussion

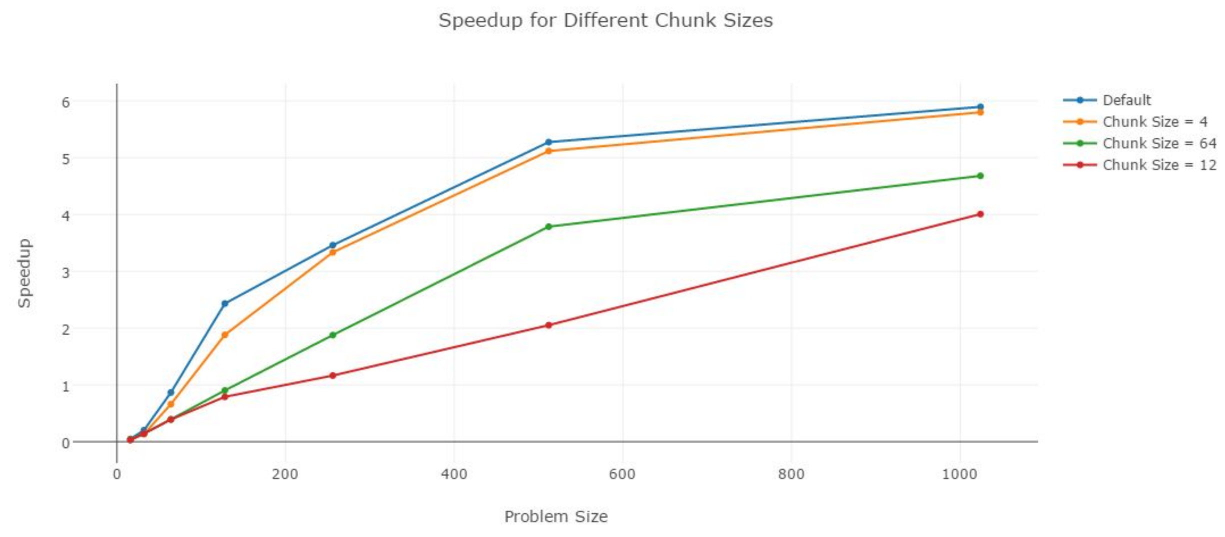
Version 1: Naive Parallelization



Problem Size vs Speedup for Default Static Scheduling

Observation

For very small problem sizes, the speedup is not much, but as the problem size increases we get more speedup. After problem size equal to 512, the speedup becomes almost constant. M=16 have less speedup than all other cores because may be the implicit synchronization cost of parallel 'for' loops (i.e. start and stop parallel execution of the threads) dominates the execution time.

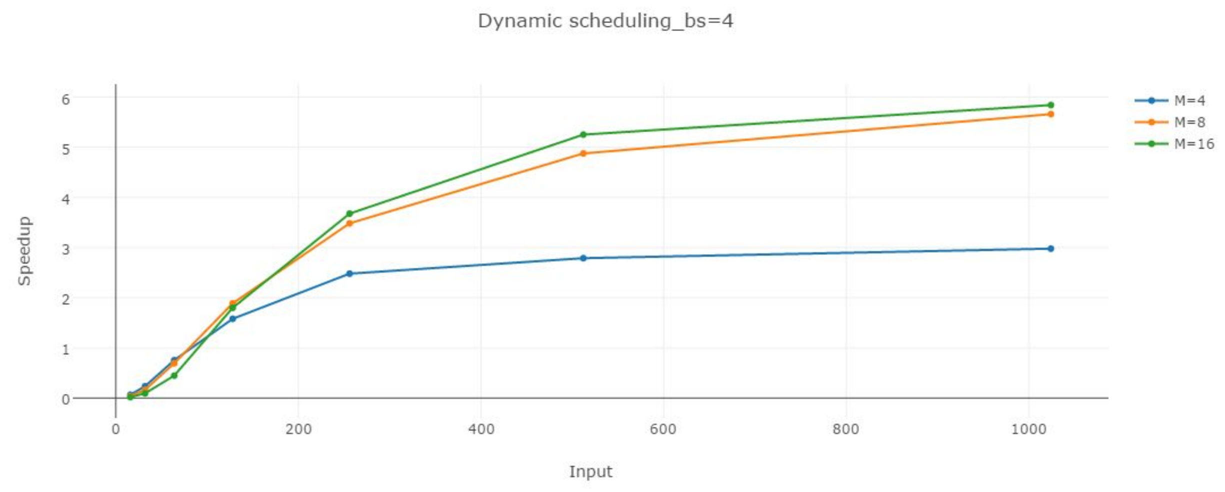


Problem Size vs Speedup for chunk sizes, = 4, 64, 128 and default chunk size

Observation

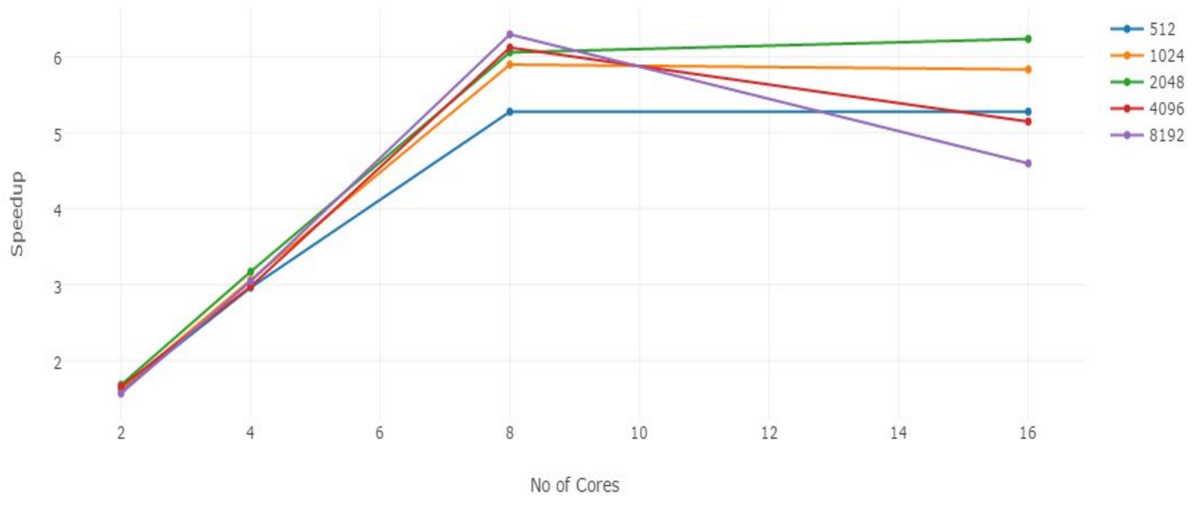
Here,we observe that when there is default scheduling the speedup attained for different sets of input is maximum. This is due to the fact that the Row Cyclic method has poor locality of reference. For this reason we have used the Row Cyclic algorithm with default scheduling and gained better speedup.

Similar is the case of dynamic scheduling with block size = 4 , 64 , 128 .



Observation:

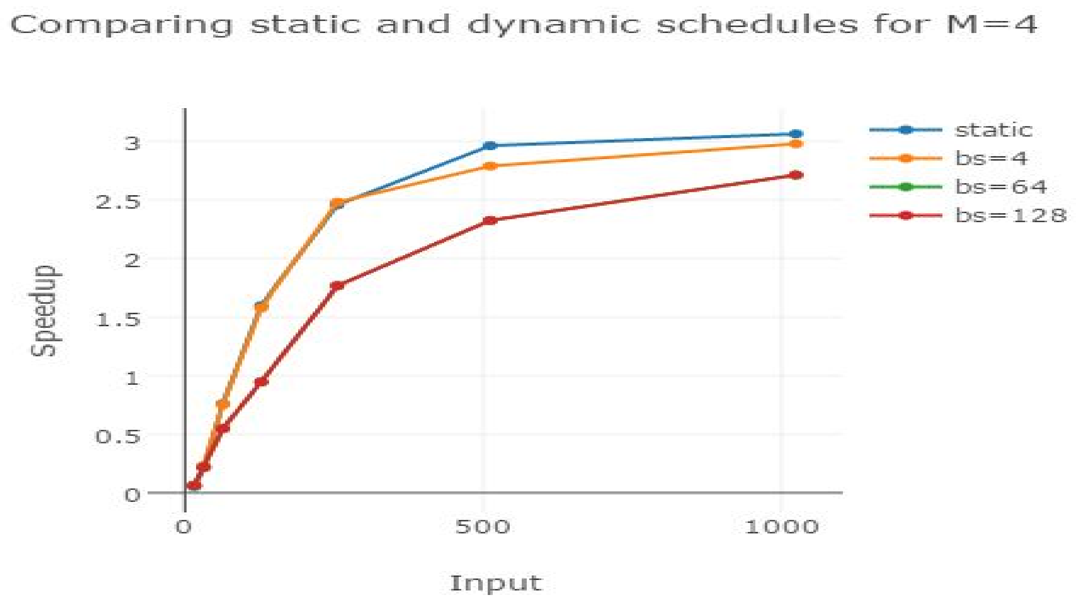
The pattern of dynamic scheduling for different number of threads is similar to above figure. In all cases of block sizes we get similar pattern to that of static scheduling.



No of Cores vs Speedup for Problem Sizes 512, 1024, 2048, 4096, 8192

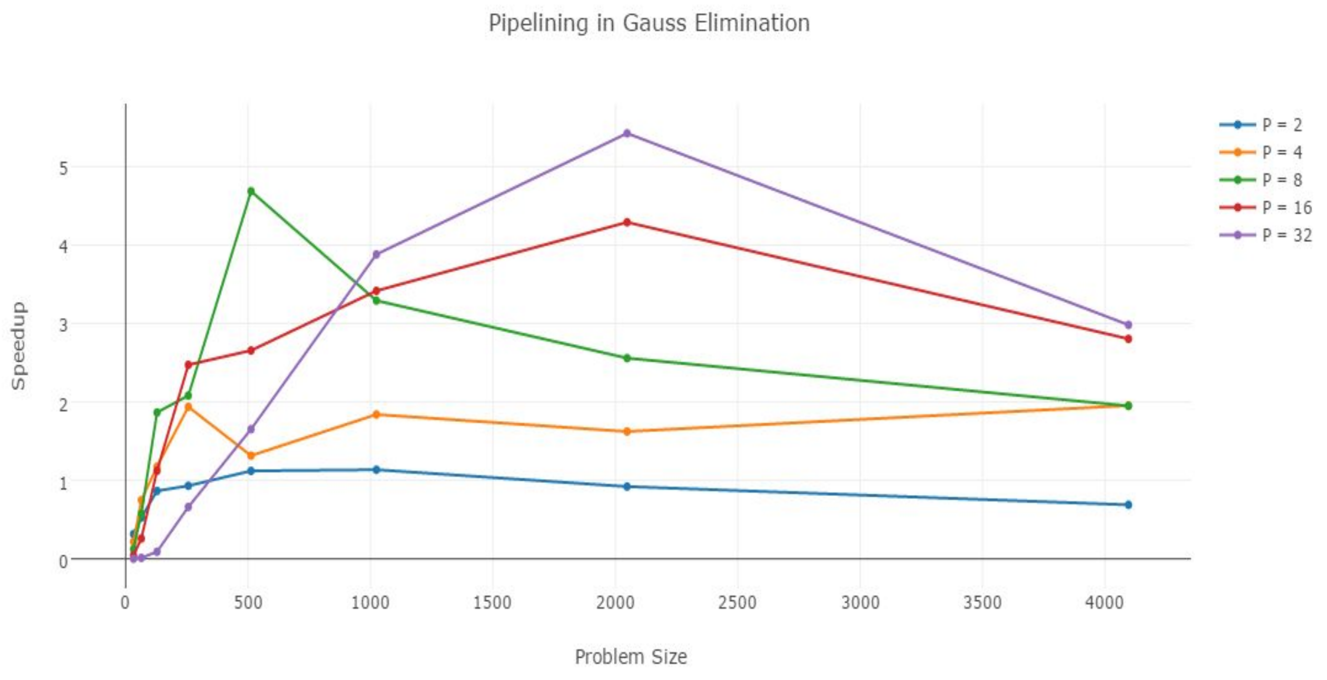
Observation:

As the number of cores increases , the speedup increases initially (till M=8) but after sometime the speedup doesn't increase but starts to decrease,the possible reason is 'parallel overhead and this dominates speedup and increases inefficiency. For each core,according to Amdahl's Law as the number of problem size increases, the speedup increases this can be clearly be seen from the figure above.(computation gradient more than communication gradient)



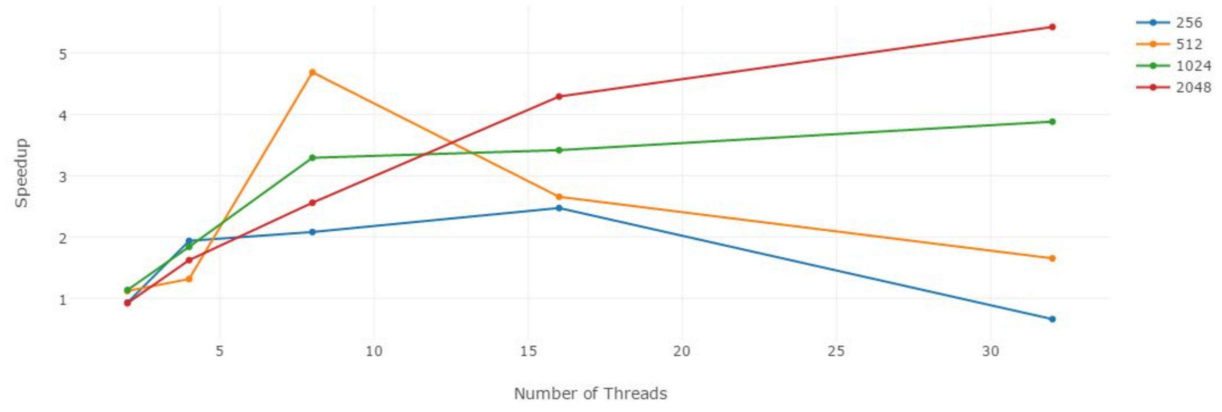
Observation:
As the problem size increases,the speedup for static scheduling (default) is same as dynamic scheduling for block Size=4 but as input tends to increases the default static as well as dynamic scheduling are better than the rest and the static schedule is better efficient than the dynamic part.

2. Pipeline algorithm

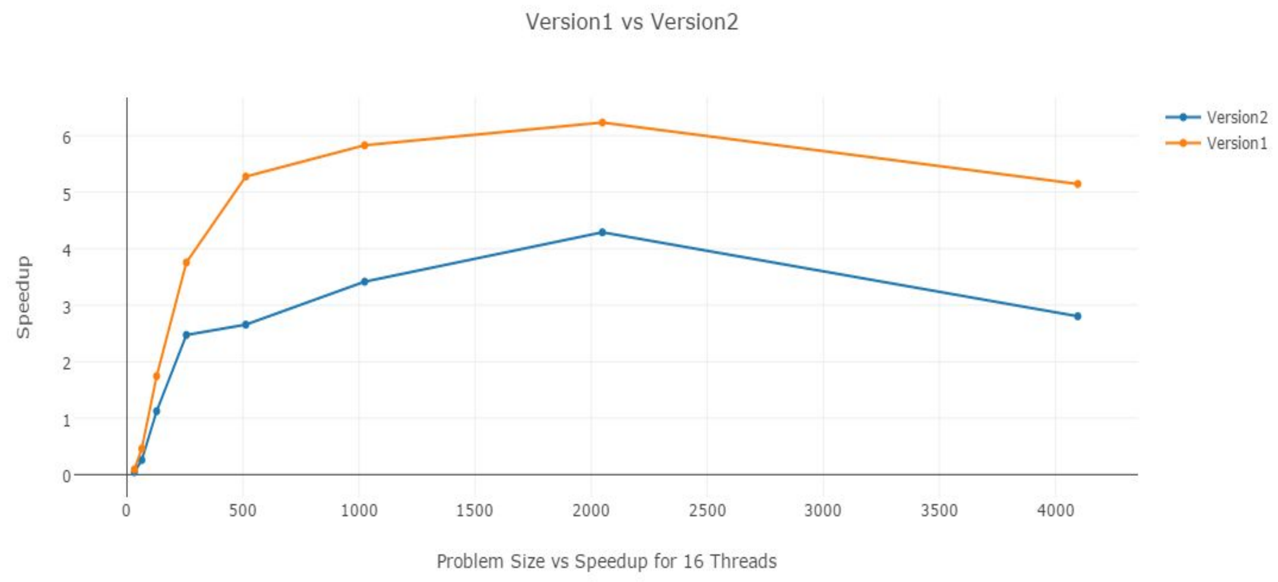


Problem Size vs Speedup for Algorithm 2

Observation:
As the problem size increases, speedup increases for a particular number of threads. But after problem size equal to 2048 it starts decreasing again. The increase is due to increase in parallelization with increase in problem size.The decrease after problem size 2048 is possibly due to the limited size of cache.



Observation:
For a constant problem size as the number of threads increases, speedup increases. But after a certain point it starts decreasing. More number of threads perform better for more number of input size. This is because the parallel overhead in the case of more problem size is compensated by the computation the threads do.



Observation:
In both the algorithms, speedup increases as the problem size increases due to increase in parallelization. After 2048, in both the cases speedup starts decreasing. The Algorithm 1 performs better than Algorithm 2 in all the cases. This is due to the extensive amount of function calls Algorithm 2 makes. If we can somehow restrict this, we might get better speedup.

Karp Flatt Metric (Experimentally determined Serial Fraction)

	Version 1			Version2		
Problem Size	P = 4	P = 8	P = 16	P = 4	P = 8	P = 16
32	5.881237	5.520005	11.66373	5.739854	9.041061	24.08769
128	0.50083	0.326985	0.544932	0.801051	0.883165	11.38934
512	0.116747	0.073777	0.135459	0.68046	0.101045	0.334967
1024	0.101896	0.050969	0.116254	0.391364	0.204329	0.245576
2048	0.087092	0.045847	0.104445	0.488438	0.303825	0.181943
4096	0.114917	0.043902	0.140621	0.349063	0.443339	0.313764

Version 1 :As the problem size increases, e decreases for nearly all cases but for larger input size (N=4096) e starts to increase.
For constant problem size, e decreases as we go from 4 threads to 8 threads, but after that it starts increasing. This is due to parallelization overhead in the case of more than 16 threads.
Version 2:As the problem size increases, e decreases initially but after N=1024, e starts to increase that is parallel overhead plays an important role.
For small problem sizes, e keeps increasing as we increase the number of threads due to parallelization overhead.

Some **problems** that occur during parallelization:

1. Poor data locality
2. Overheads of creating and destroying threads at each iteration
3. Another problem with the classic Gaussian elimination method is that it performs too many accesses to the memory where the coefficients of the system are stored. This slows down the procedure, even with compilers that perform aggressive optimizations.

4. Round-off error: The Naïve Gauss elimination method is prone to roundoff errors. This is true when there are large numbers of equations as errors propagate. Also, if there is subtraction of numbers from each other, it may create large errors.

Conclusion and scope of improvement

It is shown from the observations above that the execution time of the Row Cyclic algorithm is not affected by changes in the block size. This is due to the fact that the Row Cyclic method has poor locality of reference. For this reason we have used the Row Cyclic algorithm with a default block of size for the comparison to the other different block size Row Cyclic parallel algorithm.

As can be seen, the performance of the algorithms improved with each additional thread. We observe that the performance of the Row Cyclic algorithm increased at a decreasing rate. This is due to the fact that the implicit synchronization cost of parallel 'for' loops (i.e. start and stop parallel execution of the threads) dominates the execution time. In this case, the cost of synchronization is about $O(n^2)$. On the other hand, it is clear that the performance of the Pipe algorithm on multi cores results in better performance.With the Pipe algorithm, the decrease in performance is much slower than it is with the other algorithm. Therefore, for the Pipe algorithm there is a strong inverse relation between the parallel execution time and the number of threads, since the total communication and overhead time is much lower than the processing time on each thread. Finally, we can say that the pipeline algorithm as well Row Cyclic algorithm for Gauss Elimination works better than than naive parallel algorithm.

The gauss elimination algorithm can be done using MPI on a distributed memory and we can get better performance there because as the input increases matrix size grows and therefore due to limited cache size per process in the shared memory system the memory access time increases and therefore the communication overhead also increases as more number of cycles are wasted in accessing other memory(main memory).But in MPI every processor has its own cache and larger than in shared system therefore the performance can be enhanced there.