The QR Algorithm for Finding Eigenvectors

Eric Mikida

December 20, 2011

QR Algorithm

- Chosen by editors at Computing in Science and Engineering as one of the 10 most influential algorithms of the 20th century
- Used for finding eigenvalues and eigenvectors of a matrix
- One of the algorithms implemented by LAPACK

Research and Motivation

- Current research uses the LAPACK sequential implementation
- Eigenvalues can tell us about the stability of solutions
- Want a higher resolution solution, which isn't feasible with a sequential implementation
- The sequential implementation also limits throughput of code

Algorithm

Let $A_0 = A$. For each $k \geq 0$:

$$A_k = Q_k R_k$$

$$A_{k+1} = R_k Q_k$$

Note that:

$$A_{k+1} = R_k Q_k = Q_k^T Q_k R_k Q_k = Q_k^T A_k Q_k = Q_k^{-1} A_k Q_k$$

So all of the A_k's are similar and therefore have the same eigenvalues
As k increases, the A_k's converge to an upper triangular matrix, and the eigenvalues are the diagonal entries

QR Decomposition

Given matrix A, using Givens Rotation we can zero out an entry in the matrix, by multiplication by orthogonal matrix G_1

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}, G_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0.4961 & 0.8682 \\ 0 & -0.8682 & 0.4961 \end{pmatrix}$$

$$G_1 A = \begin{pmatrix} 1 & 2 & 3 \\ 8.0623 & 9.4266 & 10.7910 \\ 0 & -0.3721 & -0.7442 \end{pmatrix}$$

QR Decomposition (cont)

Continuing in this fashion, find $G_2, G_3 \ldots$ such that:

$$G_3 G_2 G_1 A = R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ 0 & r_{22} & r_{23} \\ 0 & 0 & r_{33} \end{pmatrix}$$

The G's are orthagonal, so their product is orthagonal

$$Q = (G_3 G_2 G_1)^T$$

$$A = QR$$

Parallelization

In each iteration we compute Q_k, R_k, and A_{k+1}:

- Parallelize the Givens rotation matrices
  - by column
  - by row
  - ~~by row and column~~
- ~~Parallelize the matrix multiplication to get Q_k, R_k and A_{k+1}~~
  - ~~by tractor tread algorithm~~

Problem Reduction

- Focus only on the QR Decomposition portion of the algorithm to get more focused results
- Dependence between individual steps of Given's rotations raises a few complications
- Used OpenMP for the benefits of shared memory (hybrid row/column method doesn't make sense here)

Given's Rotation Independence

Trick to parallelizing is to how each entry is zeroed out.
To zero out entry $A(i,j)$ you must find row k such that $A(k,j)$ is non-zero but A has zeros in row k for all columns less than j.

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & 5 & 6 & 7 \\ 0 & 0 & 8 & 9 \\ 0 & 10 & 11 & 12 \end{pmatrix}$$

Figure: Only the second row can be used to zero out the circled entry.

Column Method

- Iterate through each entry that must be zeroed out
- Calculate the 2x2 matrix needed to zero it out
- Each thread then applies the matrix multiplication to a subset of the columns

Very naive translation of the sequential algorithm that involves a lot of bottlenecking, and little independence between threads. An MPI implementation would require a broadcast at each step so that all processes would know the 2x2 zeroing matrix. OpenMP alleviates this issue but it still creates an implicit barrier. OpenMP also allows for an OpenMP for loop instead of statically allocating columns to each thread.

Column Method

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 & 29 & 30 \\ 31 & 32 & 33 & 34 & 35 & 36 \end{pmatrix}$$

Figure: Must zero out each column sequentially.

Column Method

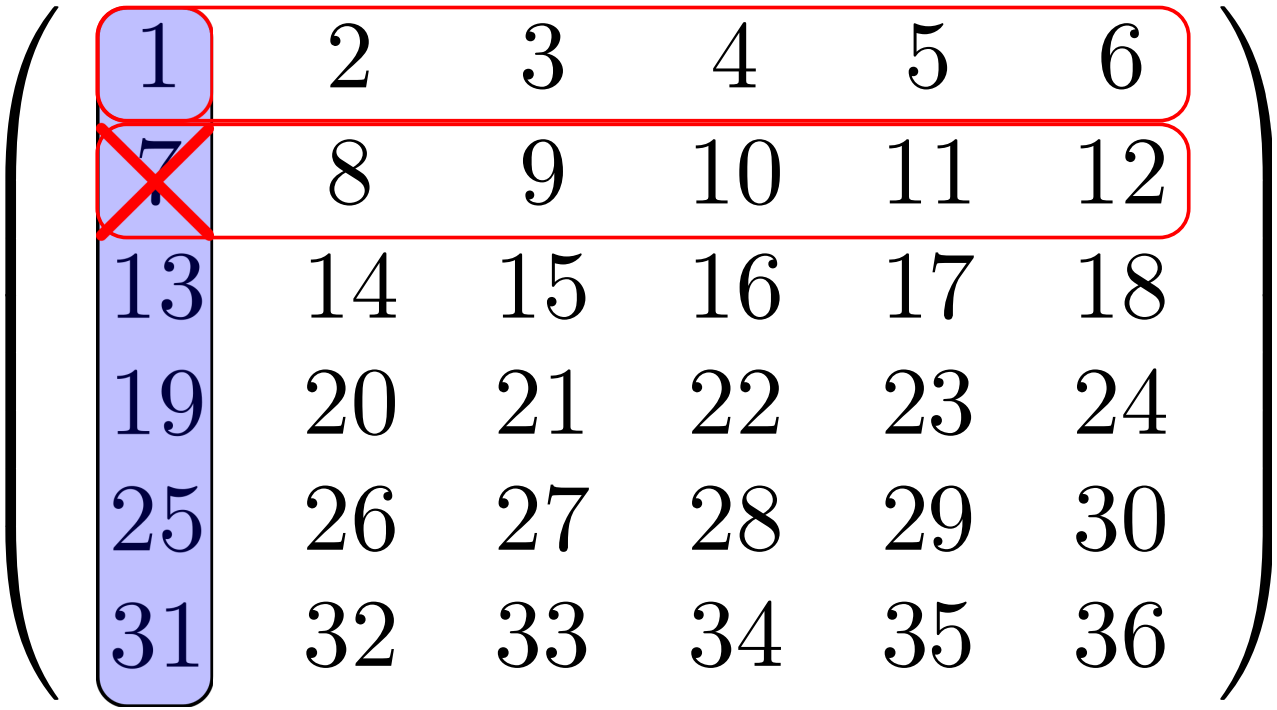

Figure: To zero out the 7, we use the 1. The first two rows will be modified by this multiplication, and each thread will apply the multiplication to a subset of the columns.

Column Method

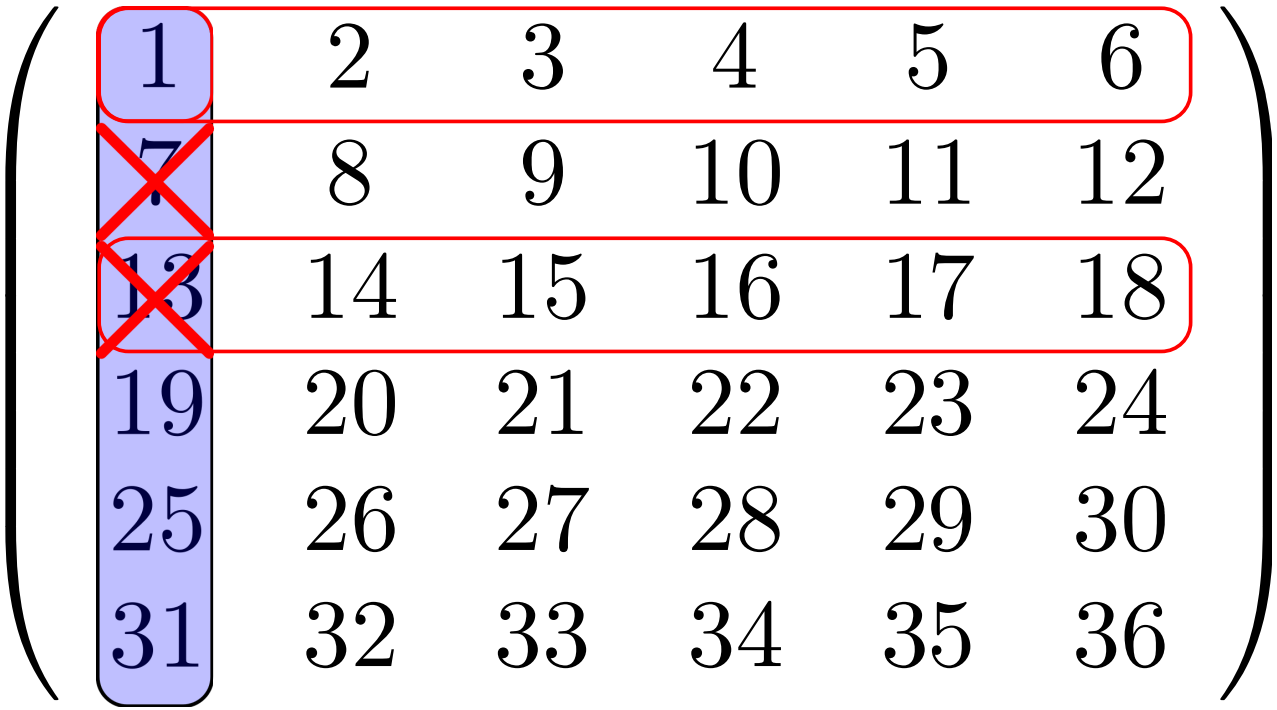

Figure: A similar procedure will zero out the 13 and modify the first and third rows.

Column Method

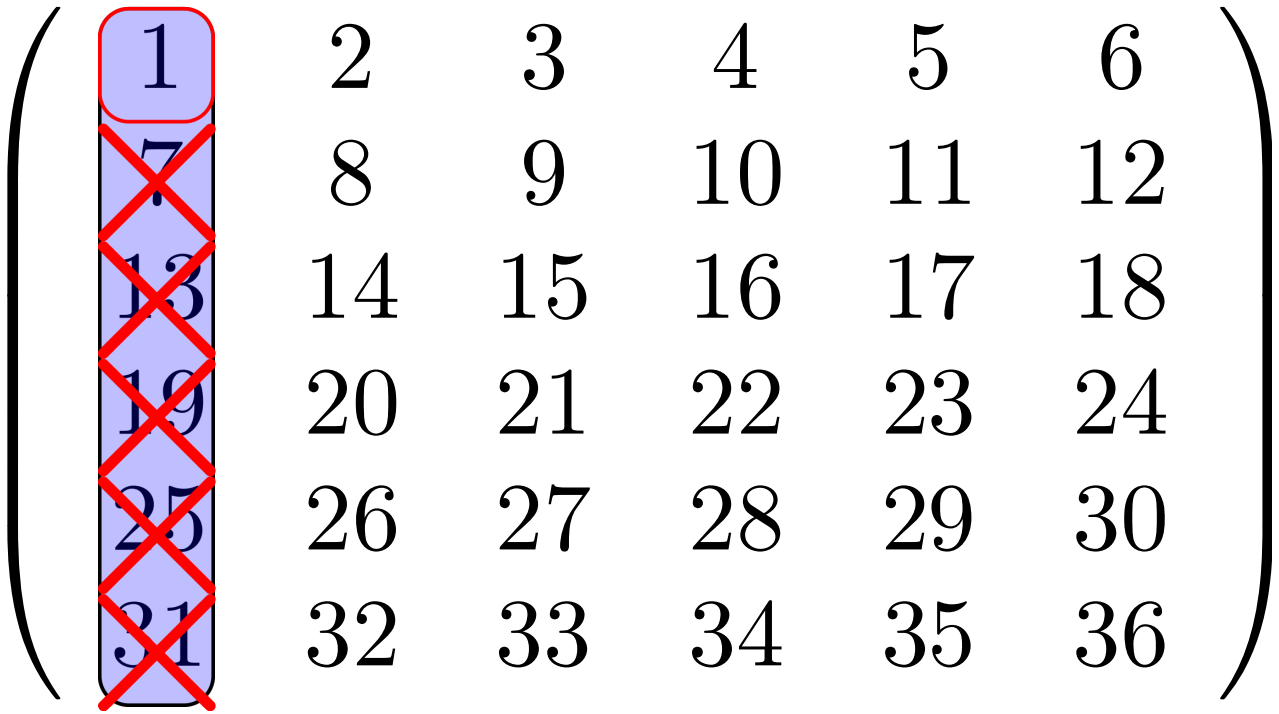

Figure: Continuing in this fashion we can zero out the whole first column.

Column Method

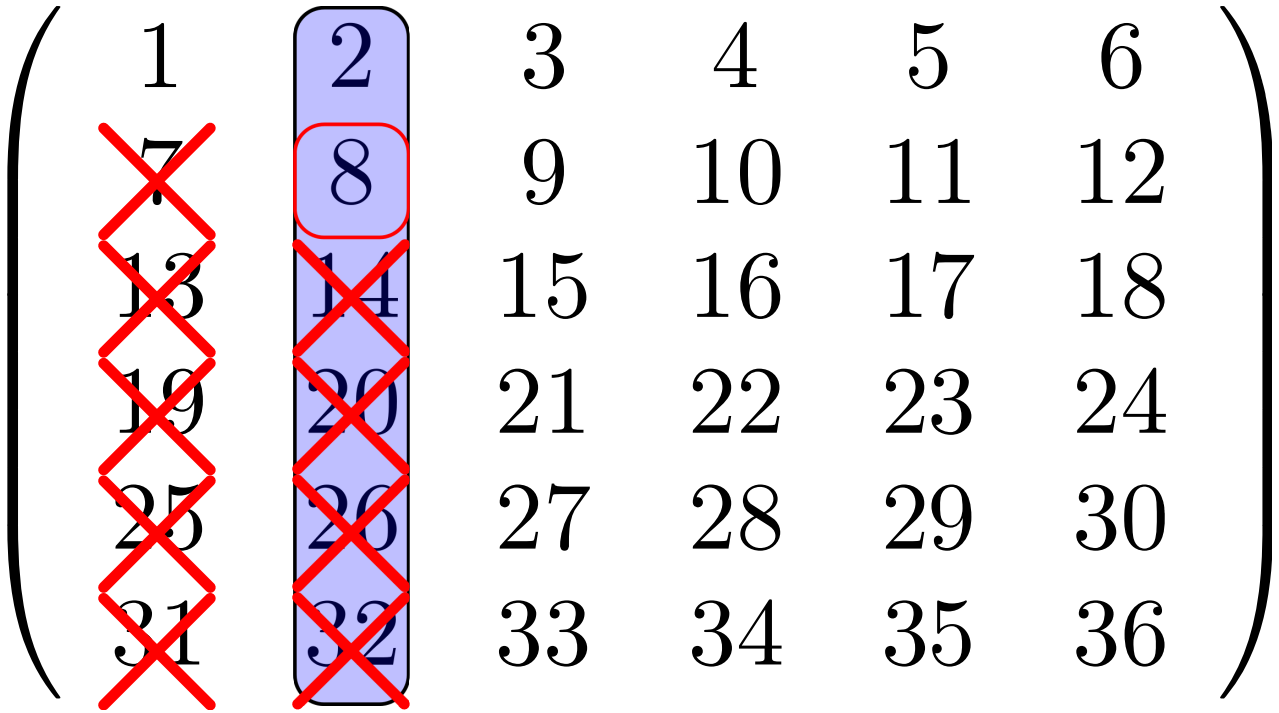

Figure: The same procedure can be applied to the second column. This time, the second row will be used to zero the entries.

Column Method

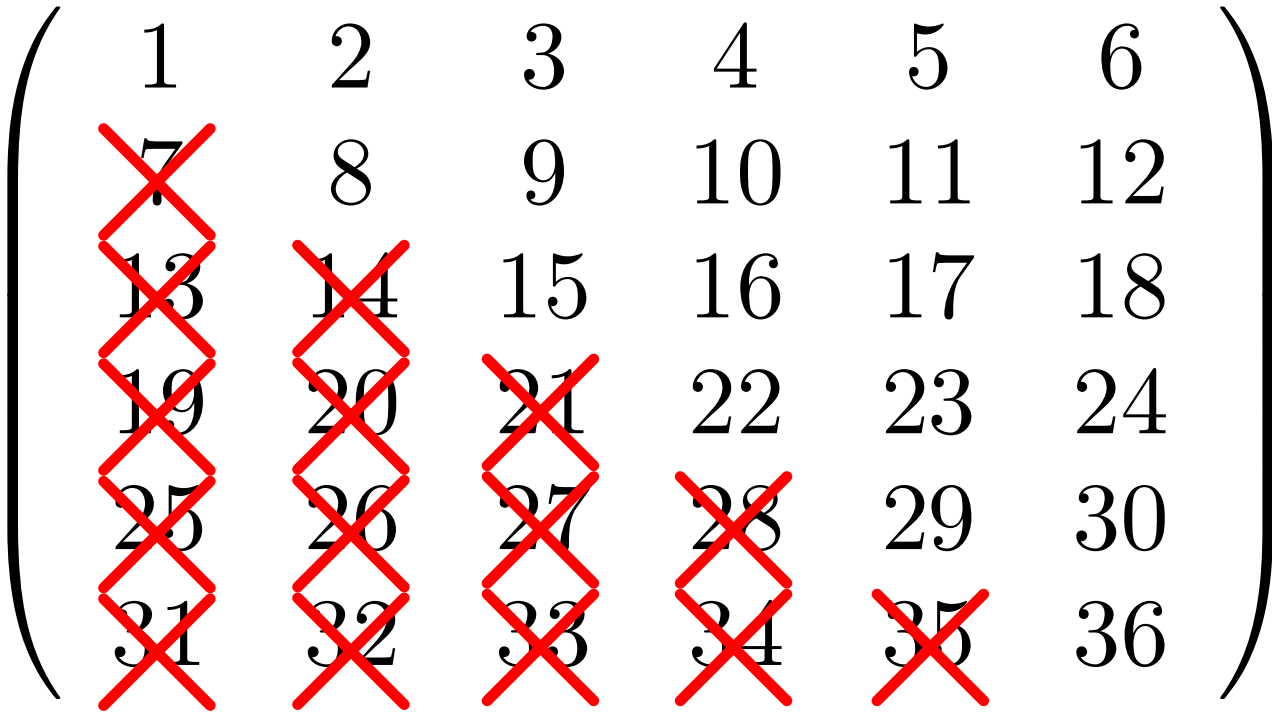

Figure: Applying this method to each column in turn will result in an upper triangular matrix as desired.

Row Method

- Each thread picks a row that still needs to be zeroed out more
- It then finds a row that it can use to zero out the next entry
- It calculates the 2x2 matrix and does the multiplication on the 2 rows

Much more independence between threads since they can each zero out an entry independently of the others. The only constraint is that when a pair of rows are being worked on by one thread, no other thread can use them. Expect much better scalability from this method, although it is more difficult to implement. The trick is minimizing the time needed for a thread to figure out what two rows to work on. Ideally it should be done in $O(1)$ time.

Row Method

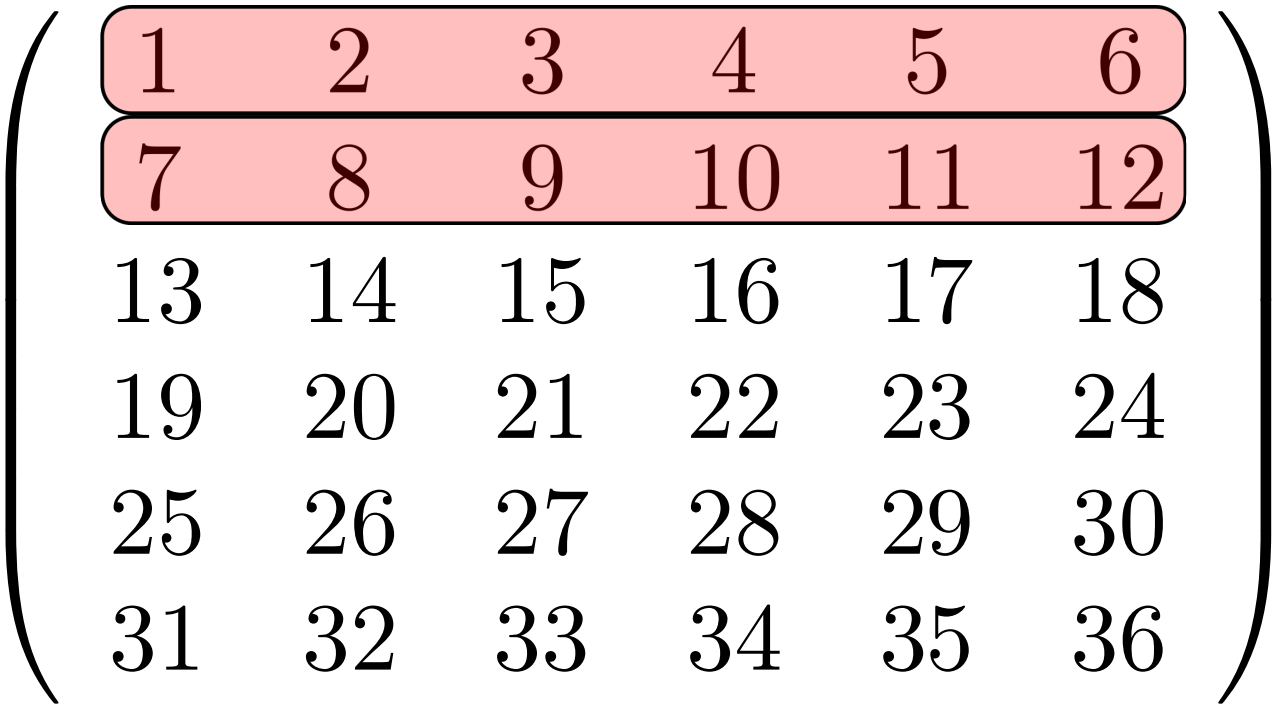

Figure: First a thread picks a pair of rows that it can work on.

Row Method

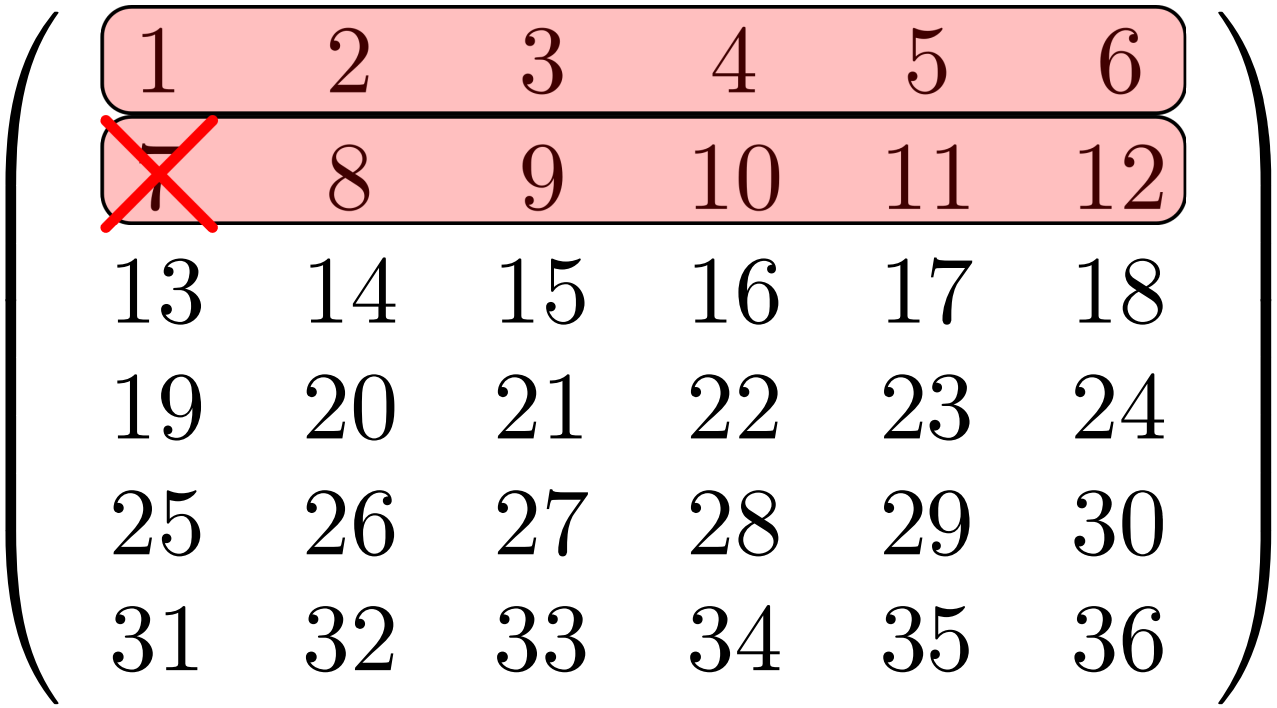

Figure: Once again, to zero out the 7, we use the 1. This time, the matrix multiplication is carried out by only one thread. The other threads are working independently on other rows.

Row Method

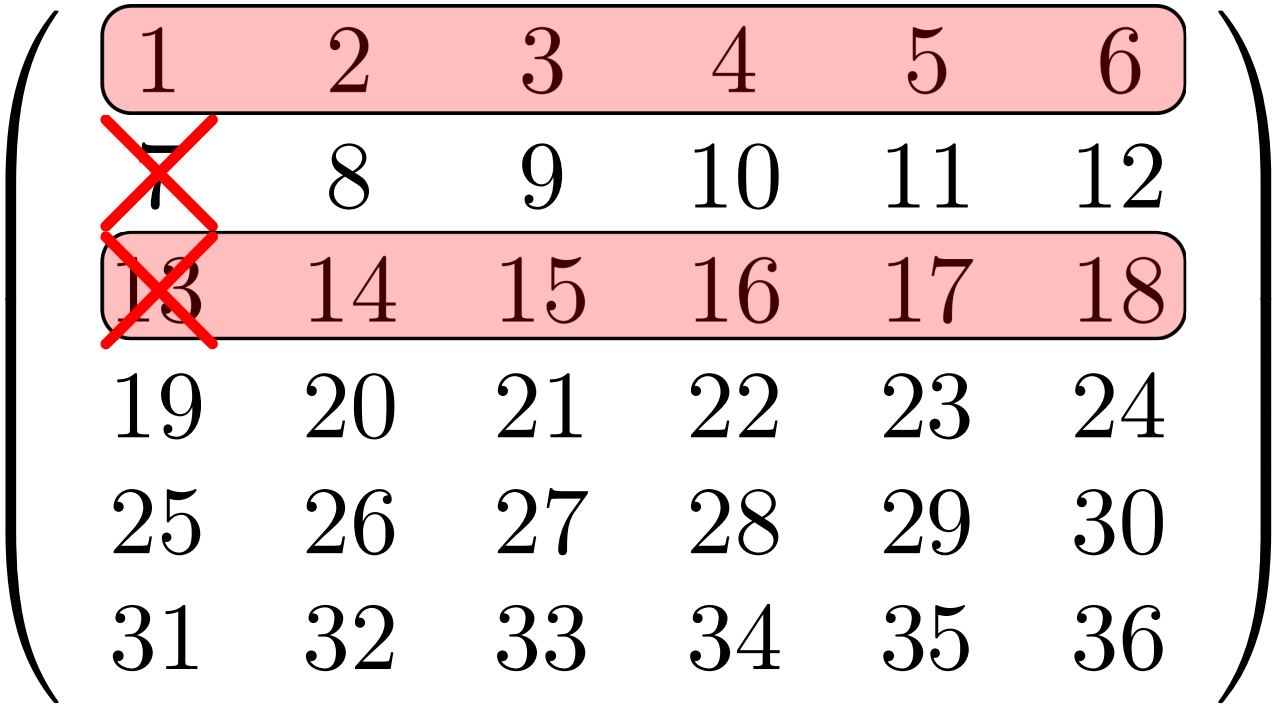

Figure: A similar procedure will zero out the 13 and modify the first and third rows.

Row Method

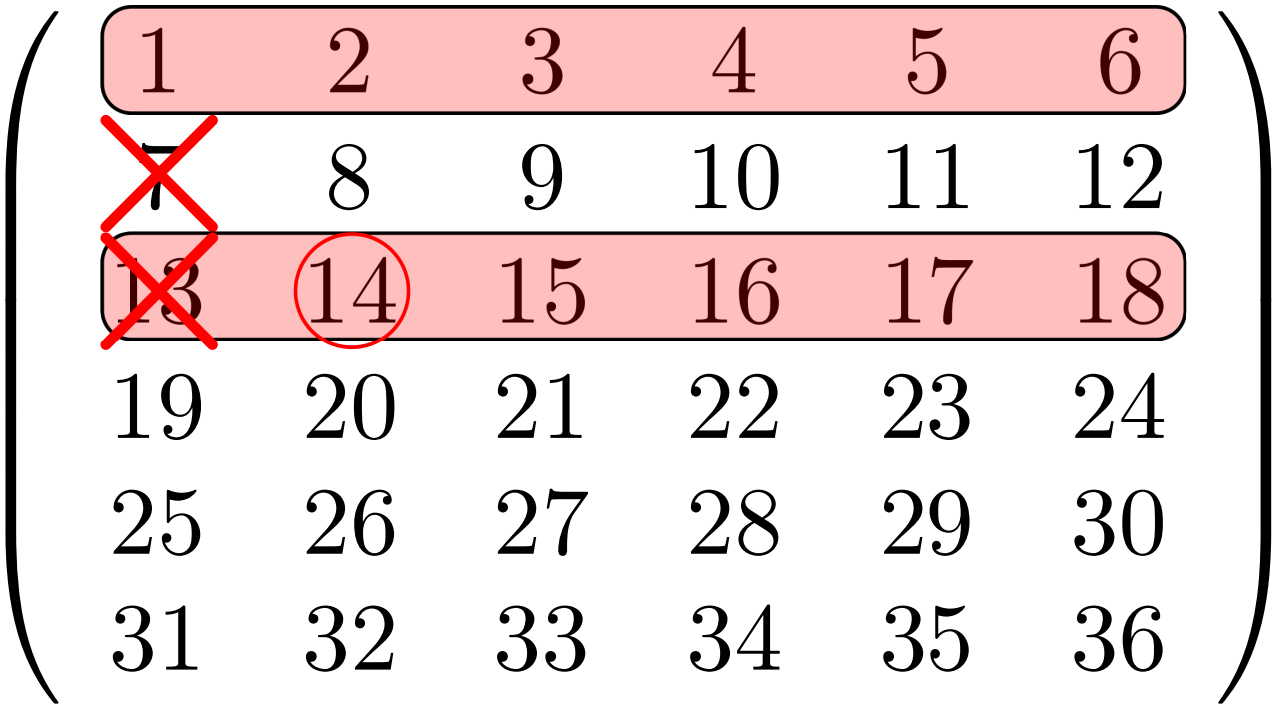

Figure: The 14, however, cannot be zeroed out with the first row.

Row Method

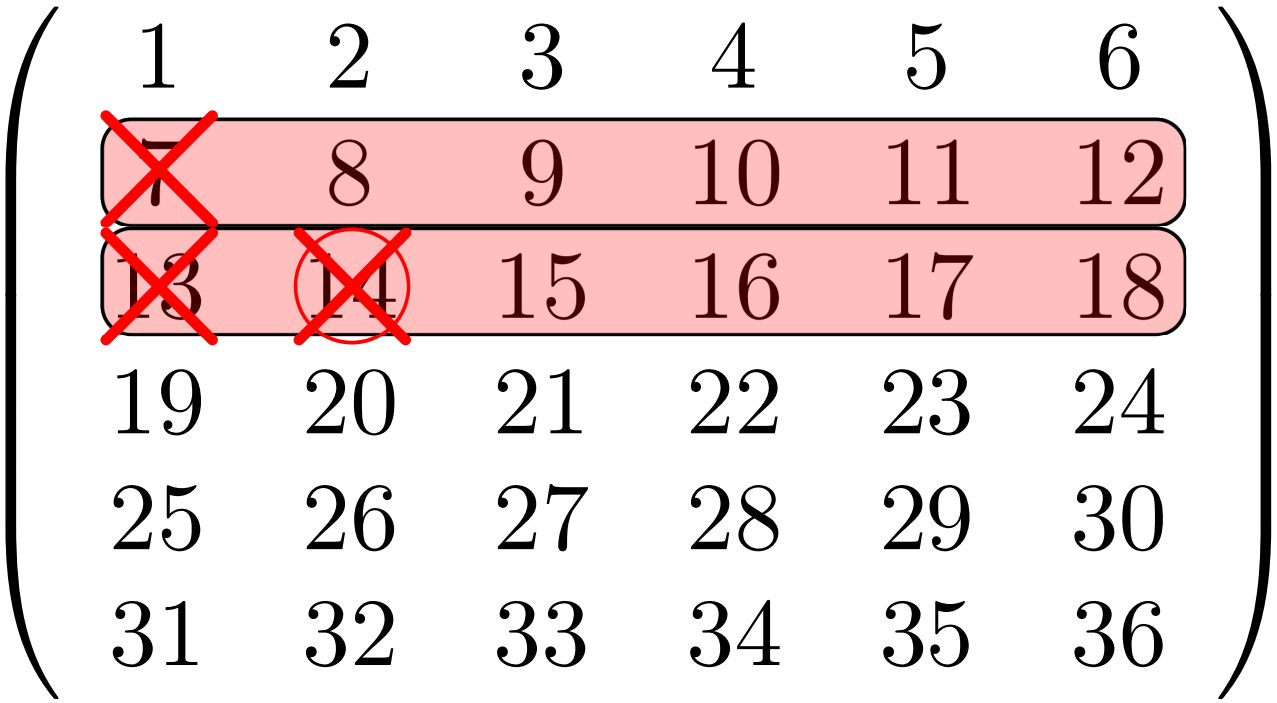

Figure: Instead it must be zeroed out with the second row, because the second row has already been partially zeroed out.

Row Method

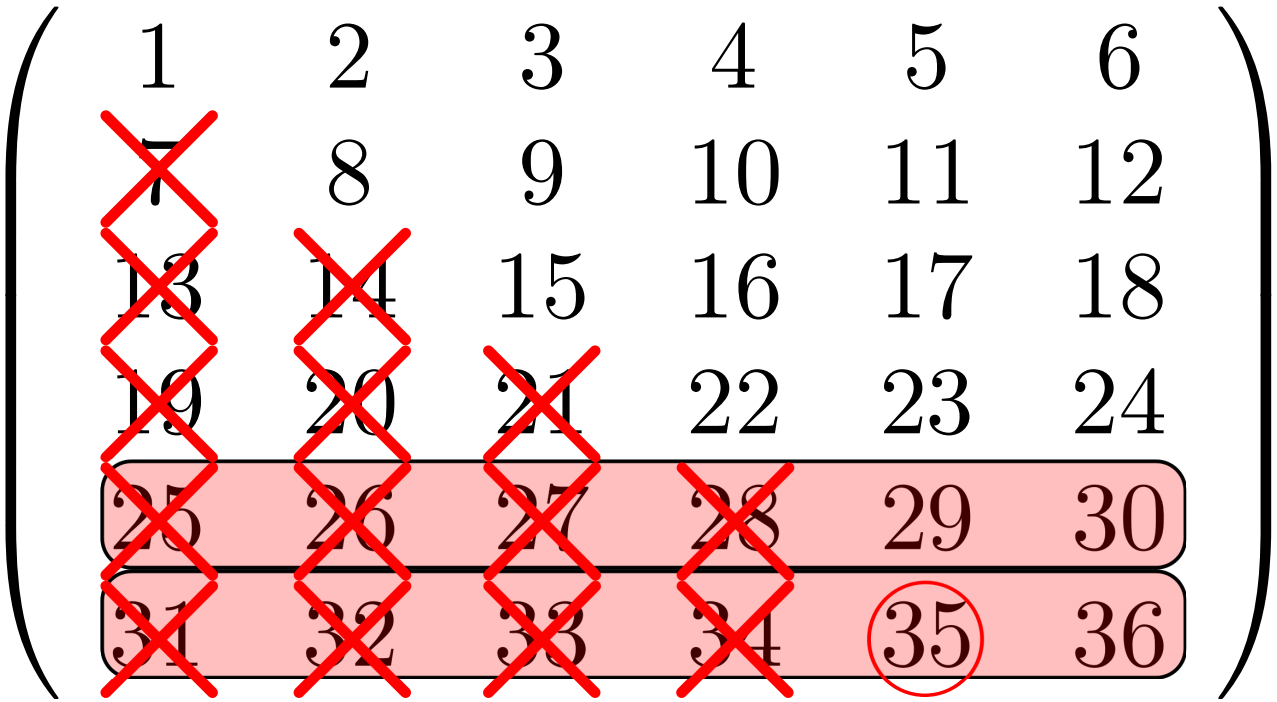

Figure: Eventually the last entry can be zeroed out and we once again end up with an upper triangular matrix.

Results

For all of the following results, the following assumptions can be made:

- Matrices were generated once uniformly at random.
- Because they are random, the matrices are also dense.
- All tests were done on one 32 core fat node, with all the cores checked out.
- Timing results are in seconds and all used the OpenMP timer.

Results

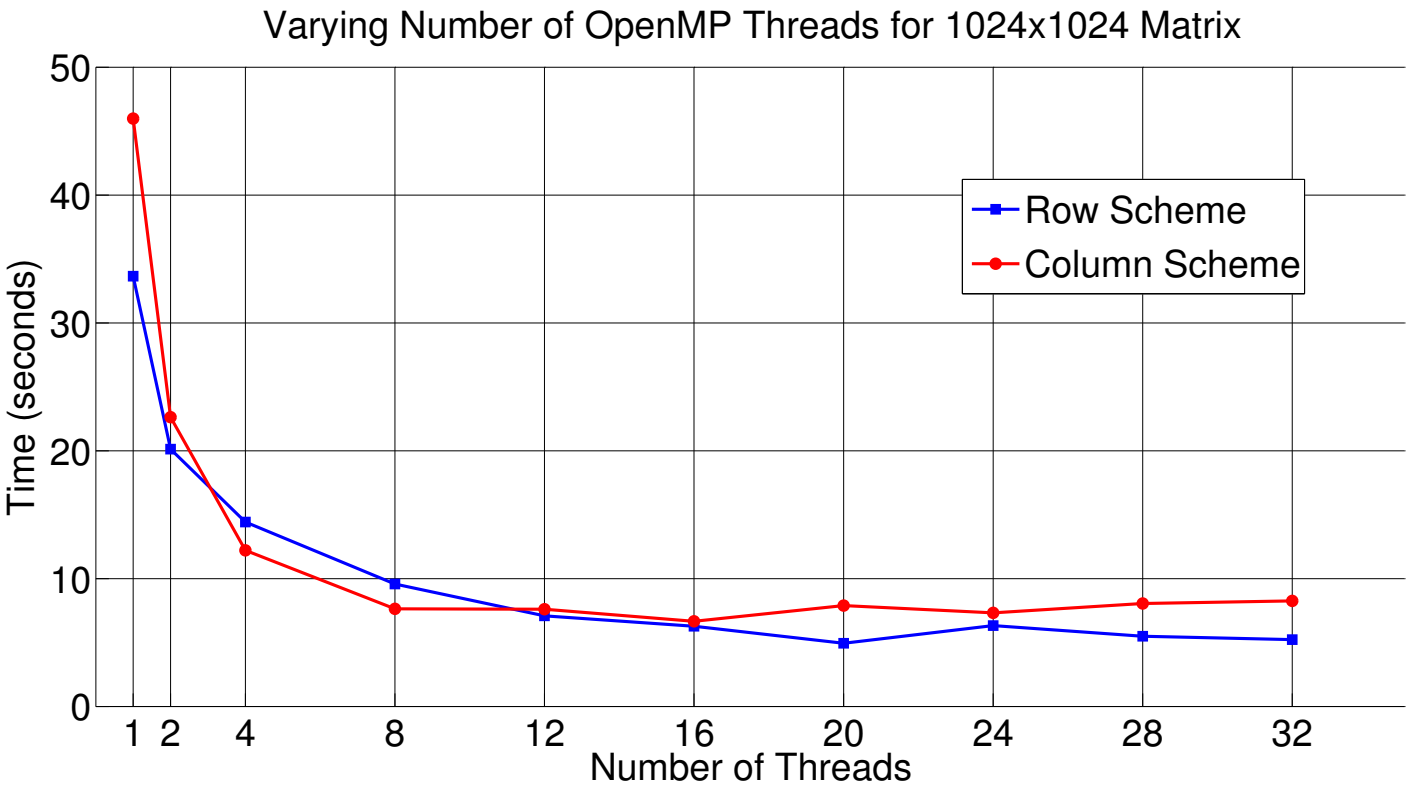

Figure: Keeping data size constant (and small). Each point is the average of 10 runs.

Results

- With the exception of two points, the row scheme outperforms the column scheme
- It would appear the turning point of the row scheme happens later, which implies that it may scale better
- Overall the schemes perform similarly
- Shared memory may reduce the penalty of "broadcasting" at each step in the column scheme which is why performance is comparable
- Hypothesize that the performance difference would be more exaggerated in an MPI scheme

Results

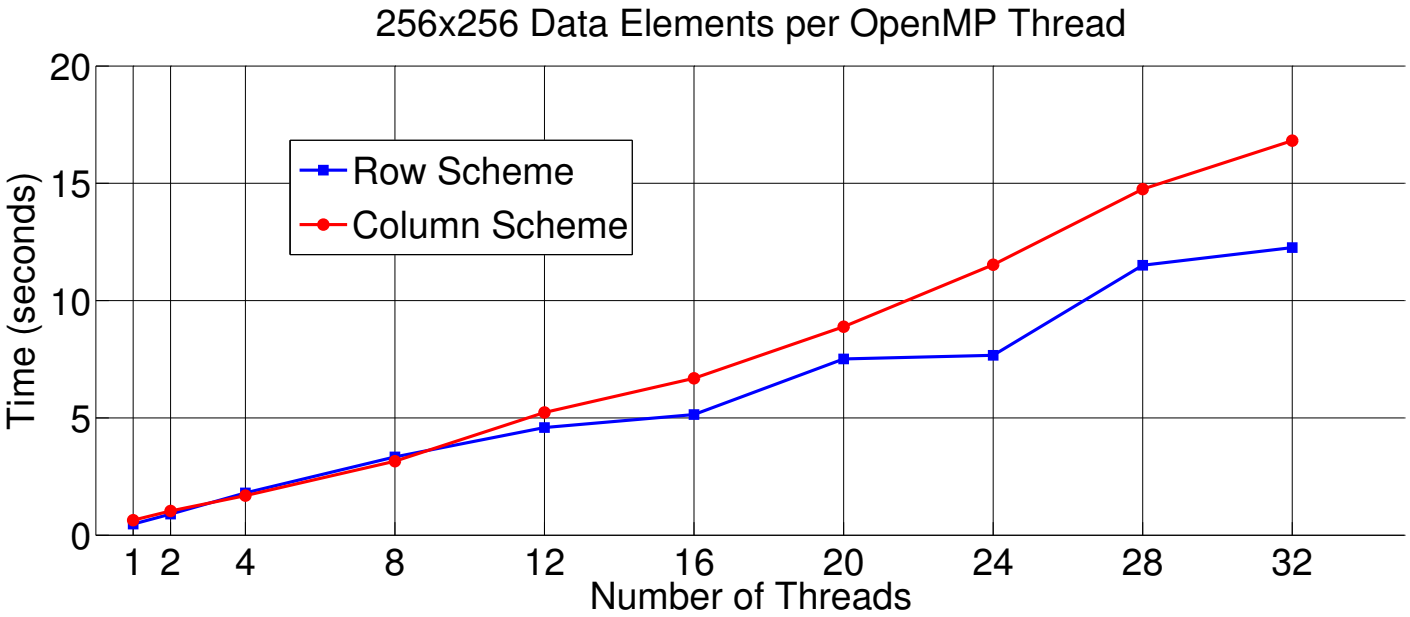

Figure: Keeping data per thread constant. Each point is the average of 10 runs.

Results

- Obviously not the ideal (but unrealistic) constant trend, but both are close to linear
- Difference between 1 thread and 32 threads is less than 20 seconds
- The row schemes performance increase is more obvious in this plot

Results

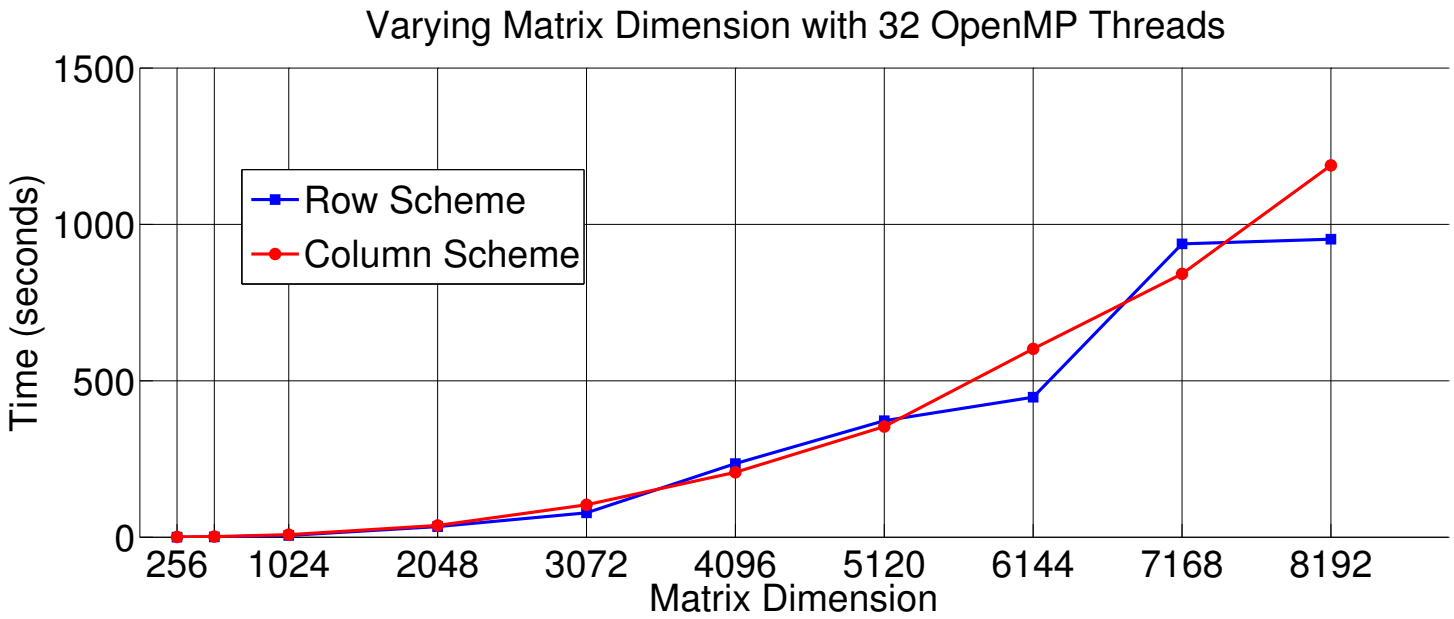

Figure: Keeping number of threads constant. Each point is the average of 10 runs.

Results

- In both cases the run time appears to scale much better than $O(n^3)$ which is the complexity of a sequential QR Decomposition

Results

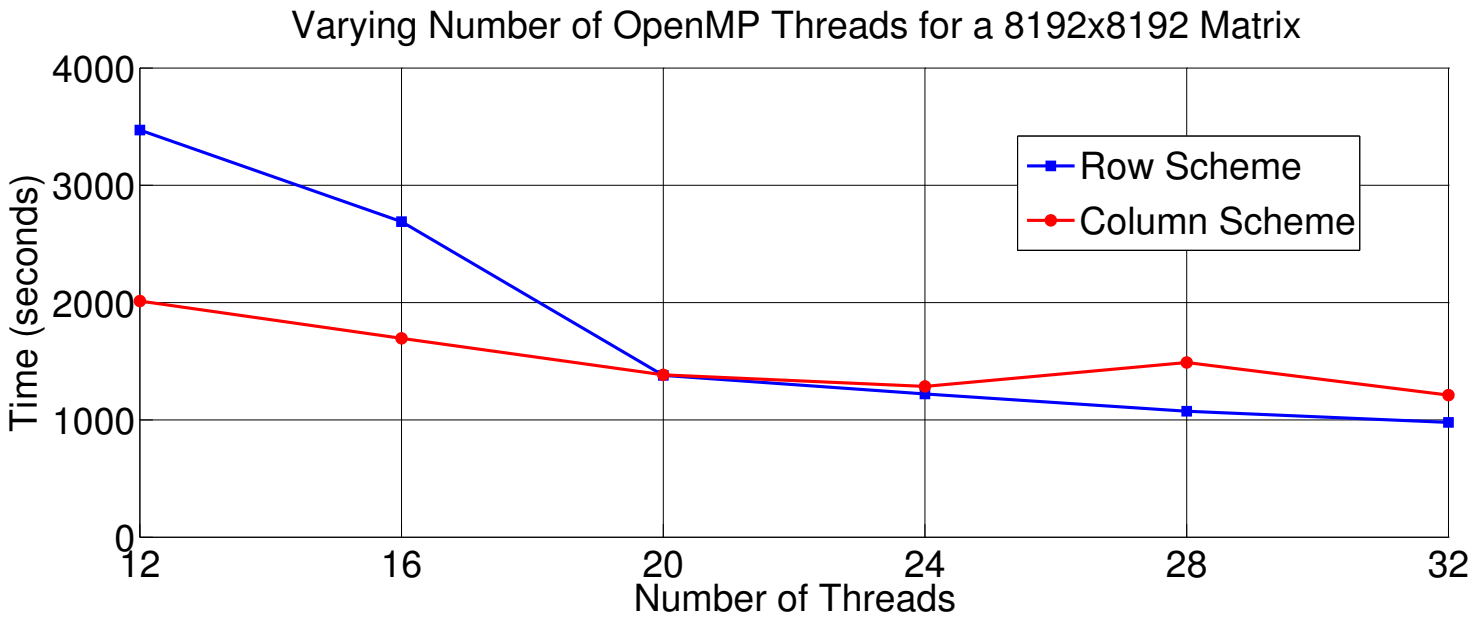

Figure: Keeping data size constant (and large). Each point is the average of 10 runs.

Results

Problems with large data sets:

- For larger matrices, the problem scales similarly to previous results
- However, when limited to 32 threads, we don't see a clear turning point in the graphs
- The benefits of the OpenMP implementation are somewhat limited by the number of cores on one node
- OpenMP's limitations become more apparent for larger data sets

Conclusions

Future Plans:

- Use CUDA to alleviate the limitations of OpenMP
- Use a hybrid MPI/OpenMP method to allow for more threads when data set gets large
- Explore a hybrid between the row and column schemes when using MPI
- Is there a point where we should finish the decomposition sequentially?
- Parallelize the matrix multiplication portion of the QR algorithm

References

- http://en.wikipedia.org/wiki/QR_Decomposition
- http://en.wikipedia.org/wiki/QR_Algorithm
- http://en.wikipedia.org/wiki/Givens_rotations
- http://www.cse.illinois.edu/courses/cs554/notes/11_qr.pdf