Parallel QR Decomposition by Random Projections

Ethan Nanavati

December 30th 2024

1 Introduction

1.1 QR Factorization

QR factorization is an important matrix decomposition that allows matrices to be expressed as the product of an orthogonal matrix Q, and an upper triangular matrix R. It is used in applications ranging from solving the linear least squares problem ($\min_x |Ax - b|^2$) to solving the fundamental linear equation (Ax=b). The usefulness of QR is –in part– due to its stability and the convenient structures of the Q and R matrices.

QR is typically performed in 1 of 3 ways. Either through a Modified Gram Shmidt algorithm, via a series of Householder reflections, or using Givens rotations. Unfortunately, while the first option is very simple, Modified Gram Shmidt QR introduces numerical instability that the other 2 options do not. Thus, by constructing it this way, one can remove an essential part of the decomposition's utility. Givens rotations and Householder reflections are similarly stable, but in this project, I choose to use Householder reflections as a matter of preference for comparisons.

1.1.1 Modified Gram Schmidt

The Modified Gram Schmidt algorithm is widely used to orthonormalize a set of vectors. It loops over each vector in the set, orthogonalizing them one-by-one. For each vector, it subtracts off its projection onto all vectors not yet orthogonalized in the set then normalizes it. In doing so, it ensures that no vector has any component parallel to another in the set and that all vectors are orthonormal.

This procedure can be used to find an orthonormal basis for the column space of an input matrix A and thus constructed into an orthogonal matrix Q. Each vector in A is a linear combination of the vectors in Q so A should be attainable from Q by matrix multiplication. Because of the construction of Q, we can write A=QR where R is upper triangular with elements as the inner products between vectors in Q and A. Essentially, the matrix multiplication inverts the Gram Schmidt process by adding back projections of columns in Q onto columns in A.

Gram Schmidt by itself is rather unstable so this modification which I will simply call MGS (Modified Gram Schmidt) is often used instead. The algorithm is more stable, yet still quite quick (an advantage to both GS and MGS). Nonetheless, due to subtractive cancelations inherent to MGS, it is regarded as relatively unstable when compared to other QR algorithms.

MGS can be parallelized somewhat efficiently. One can subtract projections of a vector to all other vectors in a set in parallel without having to worry about concurrent writes. One can also parallelize projections using a reduction for the inner product. However, the benefit of this can be outweighed by the costs of synchronization if the problem scale is not large enough.

There is, also an inherent sequential portion in looping over the current vector being normalized which decreases the parallel gains.

1.1.2 Householder QR

Householder QR is an elegant method by which a matrix is made upper triangular through repeated application of orthogonal matrices.

Central to the scheme is a type of orthogonal matrix called a Householder reflector. These matrices have a special property that given a vector $a \in R^n$, and a target vector $v \in R^n$, one can construct a Householder matrix

$$\begin{aligned} H &= I_{n,n} - 2ww^T \\ w &= \frac{a-v}{||a-v||_2} \end{aligned} \tag{1}$$

such that $Ha = v$.

By multiplying Householder matrices in a block format with A,

$$\begin{bmatrix} I_{i,i} & 0 \\ 0 & H_i \end{bmatrix} \tag{2}$$

one can eliminate the subdiagonal elements in column i of A while preserving all subdiagonal 0s to the left of it. This is done by having H_i be the householder matrix that takes the ith subdiagonal column (inclusive) $a_{(i)}$ to the standard basis multiple k*e_1 so that $H_i a_{(i)} = ke_1$. Multiplying all the block Householder matrices together gives an orthogonal matrix Q^T such that Q^TA=R so A=QR.

Importantly, the Householder matrix is rarely constructed. This is because when multiplying B*H for some matrix B and a householder reflector H, one can write $B * (I_{n,n} - 2ww^T) = B - (Bw)(2w^T)$ and compute in $O(n^2)$ time.

Unlike MGS, Householder QR (which I will now call HQR) experiences no subtractive cancellation. It also proceeds by multiplying orthogonal matrices which is inherently a very stable operation. Constructing the H_i and performing the matrix multiplications makes HQR meaningfully more expensive than MGS. Still, HQR is widely used for QR decompositions when precision is important.

HQR can be parallelized by parallizing the ubiquitous matrix vector multiplications in the method and matrix subtractions. Unfortunately, this is the extent of the parallelism. Each step of the HQR algorithm is dependent on the last, creating a seriality that cannot simply be parallelized. Moreover, as the algorithm progresses, there is less room for parallelism. The matrices operated on decrease in size, causing the benefits of parallelism to decay. That is, processors deployed at the start of the algorithm may become unutilized as it proceeds. This decreases the algorithm's ability to scale with the number of processors used.

1.2 Random Projections

Like many matrix decompositions, QR factorization takes $O(n^3)$ for a square matrix with n columns (regardless of which of the mentioned methods are used). This can be an intense computational burden which makes parallelism quite valuable. However, as discussed previously, standard parallelism techniques are not a perfect method to speed up QR algorithms.

By using a method of random projections (as put forward in Halko et al. 2010 [2]), we can mitigate the issues of parallel QR decomposition and speed it up considerably.

This method assumes that the matrix is approximately low-rank. This is the case in many practical applications, for example, sensor data and social-network graphs.

Suppose that a square matrix A is low rank with rank $k << n$ the number of rows and columns. Given an independent set of k vectors in R^n stacked as columns in a matrix Ω, A*Ω is almost surely full-column-rank. This is because a randomly sampled vector will not be in the null-space of A with probability 1 (and with floating point approximations it isn't too different) so each column of A*Ω will almost surely be independent of the others.

This set of k column vectors forms a basis for the column space of A so if we orthonormalize A*Ω as P, we have an orthonormal basis for the column-space of A. This means that PP^T is an orthogonal projection matrix onto the column space of A so

$$A = PP^T A \tag{3}$$

This has important implications for QR decompositions. Supposing that A is approximately low rank, we can do this process, compute

$$Q_P R_P = P^T A \tag{4}$$

With a standard QR decomposition, then

$$(PQ_P)R_P = PP^T A \approx A \tag{5}$$

so we have the QR decomposition of the original A matrix.

The costliest steps of this process are A*Ω in $O(n^2k)$, Orthogonalizing A*Ω in $O(nk^2)$, QR decomposition in $O(n^2k)$, and $P * Q_P$ in $O(n^2k)$. If $k << n$ the savings are very significant and the expense of the pre-processing step should be outweighed by overall algorithm time.

1.3 Goals

In this project, I will use random projections to reduce the amount of time for 2 different QR decompositions (MGS and HQR). I predict that these methods will have similar parallel scalability to their more standard counterparts, while also allowing for quick and reliable computation.

2 Implementation

I used OpenMP's shared memory framework for the entirety of this project.

2.1 Eigen

I used Eigen to store the matrices and perform some of the operations in this project. Eigen provided parallel outer products, matrix-matrix multiplication, and support for block matrix operations (useful for HQR). However, after testing, it became clear that Eigen's parallelism was non-uniform. Some functions had good parallel support and some did not. In particular, Matrix-vector multiplication, Matrix subtraction, and inner products were not meaningfully parallelized, so I made my own versions for them. The first 2 were done simply with a parallel for loop and the last was accomplished via a reduction.

Every object, regardless of whether it was a matrix or a vector, was represented as an Eigen MatrixXd object.

2.2 Input Matrix

I generated the input matrix for QR (A) to be approximately rank k. I did this by computing A as the sum of k distinct outer products, then adding small Gaussian noise to the elements of A. The elements of the random vectors for the outer products had elements as uniformly distributed doubles in the range [-10,10]. The Gaussian noise was substantially smaller with mean 0 and standard deviation .001. The same matrix was used by each QR method for a given parameter set (eg n=500, k=50).

2.3 MGS

I parallelized MGS with little modification to the overall algorithm. I deployed threads in parallel when subtracting projections of the current vector being normalized from all the next ones. I also tried using parallel inner products, but ended up removing them for scaling purposes (discussed in results). As a result, all inner products were done using Eigen's sequential operations.

2.4 HQR

I parallelized HQR by performing all the matrix operations (except for inner products) in parallel. This fully parallelizes a step of HQR and leaves each subsequent Householder multiplication as sequential operations.

2.5 Random Projections

I generated the Ω matrix in parallel with a thread-safe random number generator. I used Eigen to perform A*Ω in parallel.

When I was implementing MGS QR, I simply used the standard MGS to orthonormalize $A\Omega$ into P. When I was using HQR, I computed the QR decomposition of AΩ with HQR and took the Q matrix as P. This works because Q forms an orthonormal set of k columns and it has the same columnspace as $A\Omega$.

I multiplied $P^T A$, computed $Q_P R$ with HQR or MGS, then recovered $Q = PQ_P$, R=R.

3 Results and Discussion

For every experiment conducted I ran each parameter set 3 times and took the average to improve reliability of results. I found this to be quite important, as when I was using many processors, random background processes could sometimes slow down the experiment.

I ran strong scaling experiments, varying the number of processors and timing QR decompositions for each of the 4 methods (HQR, HQR with projections, MGS, and MGS with projections). I set n (number of columns of input matrix A) to 500 and approximate k (rank of input matrix A) to 50.

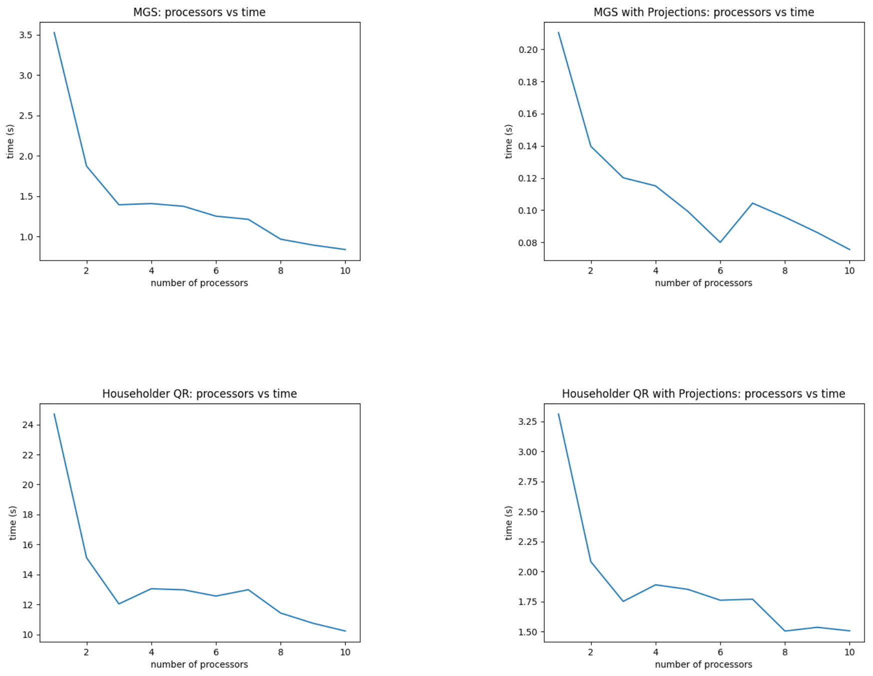

Figure 1: Strong Scaling Experiments

As the number of processors increased, standard QR and the corresponding random projections versions experienced comparable improvements in parallelism. HQR methods' times dropped by around 50% while MGS methods' times dropped by around 67%. These results were consistent across multiple runs and suggest that there is minimal reduction in parallelism when switching from standard QR methods to methods with random projections.

The difference in the level of parallelism between HQR and MGS methods is likely explained via Eigen's built-in operations. HQR leveraged more of these functions which were not all parallelized. In particular, each step of HQR used a number of matrix transposes which, from experimenting, are not parallelized. I believe that the sequential bottlenecks of each algorithm, combined with sequential operations from eigen were the leading causes for differences in parallelism.

Interestingly, when I tried to reduce the sequential portion of MGS by parallelizing inner-products, runtime grew significantly with the number of processors. I believe this is because the column-sizes (n or k) were too small to effectively reap the gains of parallelism. That is, the time for deploying and synchronizing threads was greater than the time saved through parallelism. This is consistent with my testing, where for larger vectors (on the order of 10^5) there were substantial gains to parallelizing the inner product.

I ran weak scaling experiments, varying both the number of processors and size of the problem. In order to make sure that the problem size and number of processors were in a (close to) constant ratio, I scaled the standard QR and projections QR sizes differently. Since QR is typically $O(n^3)$, I scaled the size of the input matrix (n) by $2^{\frac{1}{3}}$ each time I doubled the number of processors (p). Similarly, since QR with projections methods take $O(n^2k)$, I scaled n by $2^{\frac{1}{2}}$ each time I doubled p. Since n was likely to not be an integer, I floored it before beginning computation. n was initially set to 300 and k at 200.

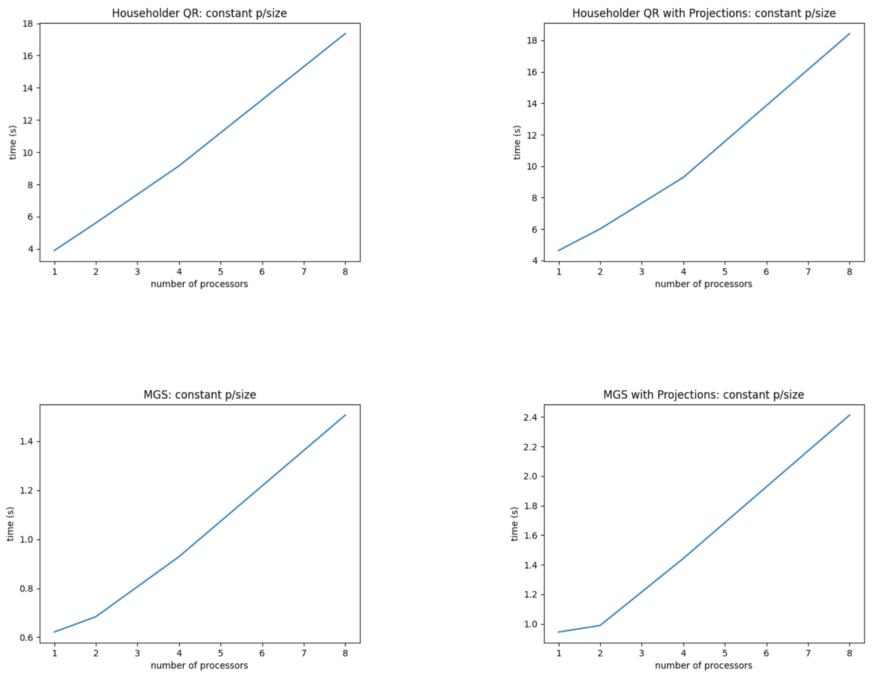

Figure 2: Weak Scaling Experiments

The results of these experiments are consistent with the strong scaling tests. Initially the effects of parallelism are stronger, but decay and are dominated as the size of the problem changes. The scaling was similar between standard QR and the corresponding QR with projections.

I also studied the effect of input matrix rank on the run-times of the QR with projections algorithms. For fixed n at 600, I varied k from 10 to 400 and timed QR decomposition for each parameter set.

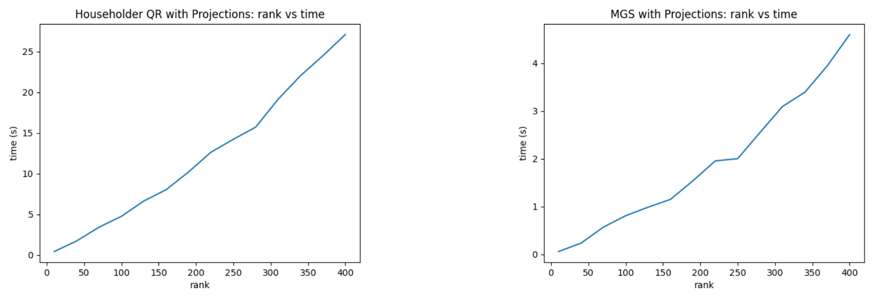

Figure 3: Effect of Rank on Time for Projections QR

As expected, the performance of projections-based QR methods varies linearly with the approximate rank of the input matrix. Also, HQR scales much worse with rank than MGS (rank changes by about 15 for 1 second difference vs k changes by around 100 for 1 second difference).

The average times for MGS experiments with n=600 was 4.05 seconds, meaning that even when $n/k < 2$, MGS with projections is substantially faster than the original. Further, the average times for HQR with n=600 was 32 seconds, meaning that even when $n/k < \frac{3}{2}$, HQR with projections is meaningfully faster than without. This indicates that for practical situations where often $k \approx \sqrt{n}$ or $k \approx log(n)$, using random projections can have an incredibly high impact on runtime.

Lastly, I computed relative error $\frac{||QR-A||_F}{||A||_F}$ for each method of QR decomposition for different sizes of input matrix. To ensure these results were accurate, I averaged with 10 runs instead of 3. n was initially 200 and k was 50.

Num Columns	200	300	400
HQR	1.5e-15	1.7e-15	2.0e-15
MGS	2.0e-6	5.6e-7	5.09e-10
PHQR	4.5e-5	5.5e-5	1.4e-3
PMGS	7.2	11.9	4.7e-5

Table 1: Relative error for QR Decompositions

Note: PHQR and PMGS are HQR and MGS with projections

Note first that all of these results were calculated using double precision. In my initial experiments, I used floats to store the results, but all relative errors were quite poor. I realized that the only way to achieve reliable results was to increase the level of precision for calculations (as Eigen does).

HQR was the more accurate of the standard QR methods (as expected). PHQR was also generally more accurate than PMGS. The performace of both PMGS and PHQR was worse than their standard counterparts.

It is understandable that methods with projections were less accurate than standard methods. After all, those methods assume that A is exactly rank k and require an additional QR decomposition. However, while the relative error for PHQR is reasonable, PMGS has absurd error. For n=200 and 300, relative error exceeds 1, indicating that the QR decomposition is wildly wrong. I reason this in 2 ways. PMGS requires 2 uses of MGS and the errors might compound between the two. This can be especially problematic as the projected matrix that the first MGS is run on is already an approximation of a larger matrix. Compounding all these errors might explain the poor performance of PMGS.

Secondly, the noise that I introduced to the input matrix, when combined with the instability of the first MGS might have introduced enough error that P was not an appropriate orthonormal basis for A. If this is, in fact, the issue, [2] Halko et al suggests that increasing the number of columns in Ω might do a better job at capturing the true range of A (adding redundancy and accounting for the shifts that perturbing A caused).

4 Conclusion

In this report, I explored the parallelism and performance of different algorithms for QR decomposition. I used Householder QR and Modified Gram-Schmidt QR as well as 2 other methods designed to reduce the problem size via random projections.

I found that the methods using projections exhibit comparable parallelism to the standard methods and scale well with increasing threads. Moreover, using projections substantially decreased runtime for both Gram-Schmidt and Householder QR.

I also determined that while Gram-Schmidt was significantly faster than Householder QR, Householder QR was far more stable than Gram-Schmidt. Indeed, in some cases the instability of Gram-Schmidt with projections rendered its outputs useless and erroneous.

Future work might seek to improve the stability of MGS with projections by increasing the number of random samples of the column-space of A by adding extra columns to the Ω matrix. This might yield a method that is faster than conventional methods, but still accurate and precise.

5 Bibliography

References

[1] J. G. F. Francis. The qr transformation a unitary analogue to the lr transformation - part 1. *Comput. J.*, 4:265–271, 1961.

[2] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions, 2010.

[3] Axel Ruhe. Rational krylov sequence methods for eigenvalue computation. *Linear Algebra and its Applications*, 58:391–405, 1984.

[1] [3] [2]