A Parallelization of QR-factorization with Modified Gram-Schmidt Orthogonalization

Vilhelm Urpi Hedbjörk

June 6, 2018

1 Introduction

QR-factorization is the decomposition of a matrix A into an orthogonal matrix Q and an upper triangular matrix R, i.e. $A = QR$. A common use of QR-factorization is to solve the least squares problem. By performing the decomposition $A = QR$, the linear equation $Ax = b$ can be written $QRx = b$ and since the matrix Q is orthogonal, this is equivalent to $Rx = Q^T b$ which in turn can be solved by simple back substitution. QR-factorization is also a fundamental part of the QR algorithm, which is used to find the eigenvalues and eigenvectors of a matrix.

In order for the QR-factorization to be of practical use, i.e. to solve large problems, it is imperative to efficiently parallelize the algorithm to be able to run it on supercomputers. This paper treats an implementation of QR-factorization using row-oriented Modified Gram-Schmidt orthogonalization with a cyclical column-wise partitioning scheme. Additionally, a performance evaluation on the UPPMAX system Rackham at Uppsala University, Sweden, is presented.

2 Problem Description

The problem consists of decomposing an input m x n matrix A into an orthogonal m x n matrix Q and an upper triangular n x n matrix R, such that $A = QR$. The Q-matrix is constructed by applying Gram-Schmidt orthogonalization to the column vectors of A. Let a_j be a column vector of A and $proj_{q_i}(a_j) = \frac{\langle q_i, a_j \rangle}{\langle q_i, q_i \rangle} q_i$, i.e. the projection of a_j onto the vector q_i. The Gram-Schmidt method is then defined as follows:

$$q_1 = \frac{a_1}{||a_1||}, \tag{1}$$

$$q_2 = \frac{a_2 - proj_{q_1}(a_2)}{||a_2 - proj_{q_1}(a_2)||}, \tag{2}$$

$$q_3 = \frac{a_3 - proj_{q_1}(a_3) - proj_{q_2}(a_3)}{||a_3 - proj_{q_1}(a_3) - proj_{q_2}(a_3)||}, \tag{3}$$

$$\vdots \tag{4}$$

$$q_j = \frac{a_j - \sum_{i=1}^{j-1} proj_{q_i}(a_j)}{||a_j - \sum_{i=1}^{j-1} proj_{q_i}(a_j)||}. \tag{5}$$

This produces a matrix Q consisting of n orthonormal column vectors q_j. The problem with the classical Gram-Schmidt method is that it is numerically unstable. Therefore the Modified Gram-Schmidt method (MGS) is used instead, where instead of as in 5, q_i is computed like so:

$$q_i^{(1)} = a_i - proj_{q_1}(a_i), \tag{6}$$

$$q_i^{(2)} = q_i^{(1)} - proj_{q_2}(q_i^{(1)}), \tag{7}$$

$$q_i^{(3)} = q_i^{(2)} - proj_{q_3}(q_i^{(2)}), \tag{8}$$

$$\vdots \tag{9}$$

$$q_i^{(i-1)} = q_i^{(i-2)} - proj_{q_{i-1}}(q_i^{(i-2)}), \tag{10}$$

$$q_i^{(i)} = \frac{q_i^{(i-1)}}{||q_i^{(i-1)}||} \tag{11}$$

which in exact-precision arithmetics is identical to the classical method, but is numerically stable.

Since q_i is a unit vector $proj_{q_i}(a_j) = \frac{\langle q_i, a_j \rangle}{\langle q_i, q_i \rangle} q_i = \langle q_i, a_j \rangle q_i$. The R-matrix consists of n column vectors r_j of length n. The i:th element of r_j gives the weight to be multiplied with q_i in order to create a_j. For $i < j$, this means that $r_{ij} = \langle q_i, a_j \rangle$. Note that in the expression for r_{ij} for the MGS method, a_j will only actually be the column vector a_j for $i = 1$. For $i = 2$, it will be the updated vector $q_j^{(1)}$ etc. as shown in 6 - 10. For $i = j$, $r_{ij} = ||q_i||$ as seen in 11. For $i > j$, $r_{ij} = 0$ which makes R an upper triangular matrix.

3 Solution Method

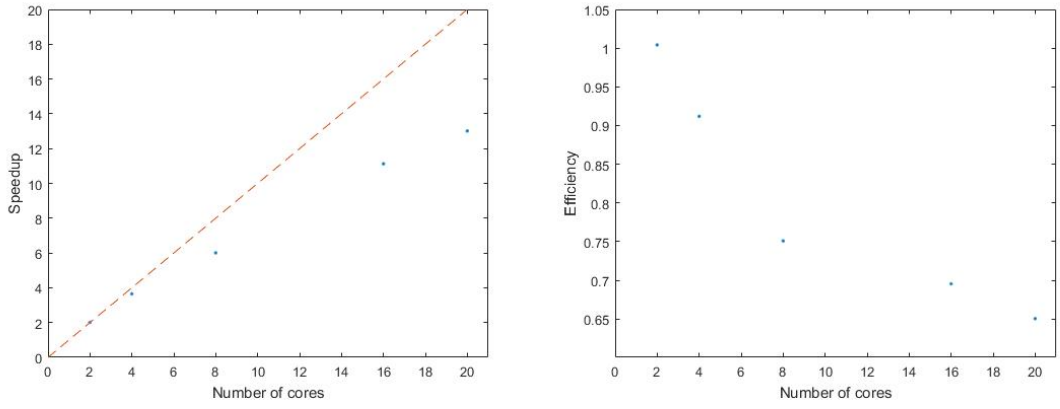
The parallelization of the MGS method is based on the observation that as soon as the first orthonormal vector q_1 is created by normalizing a_1, all other vectors $q_i^{(1)}$ $(1 < i < n)$ can be created by projecting the vector a_i onto the space orthogonal to q_1, i.e. step 6 of the MGS method[1]. At the next iteration q_2 is created and the vectors $q_i^{(2)}$ $(2 < i < n)$ can be created by projecting the vector $q_i^{(1)}$ onto the space orthogonal to q_2 (7) and so on. This way the R-matrix is created row-wise, hence the name row-oriented MGS.

The idea is to partition the task of creating the orthonormal vectors q_i (and the corresponding vectors r_i) between the processes. Since the number of vectors q_i that need to be created decrease by one every iteration, distributing the task of creating these vectors cyclically among the processes provides better load balancing compared to e.g. assigning each process a block of contiguous vectors. In the case of block partitioning, if the block size is denoted k, the first process will be idle after k iterations and the second process will be idle after $2k$ iterations etc. With cyclical (column-wise) partitioning, every process will receive k column vectors to orthogonalize, where k is the wrap-around number $(k = n/p)$ for p number of processes. That is, the first process will create q_1, q_{1+p}, q_{1+2p} etc., the second process will create q_2, q_{2+p}, q_{2+2p} and so on. This way, no processes will be idle up until the last p iterations.

4 Experiments

The parallel algorithm was implemented in *C* using the *Open MPI* library. It was run on the UPPMAX system Rackham in Uppsala, Sweden. Each node of the Rackham cluster consists of 2 Intel Xeon E5 2630 v4 at 2.20 GHz/core and the interconnect is Infiniband FDR with a theoretical bandwidth of 56Gb/s and a latency of 0.7 microseconds. The operating system used is CentOS 7. More information on the Rackham cluster is available at the UPPMAX website[2]. The compiler used was GCC 4.8.5. Three test runs were done for each set of parameters and the best timing was recorded to reduce the effect of noise on the results.

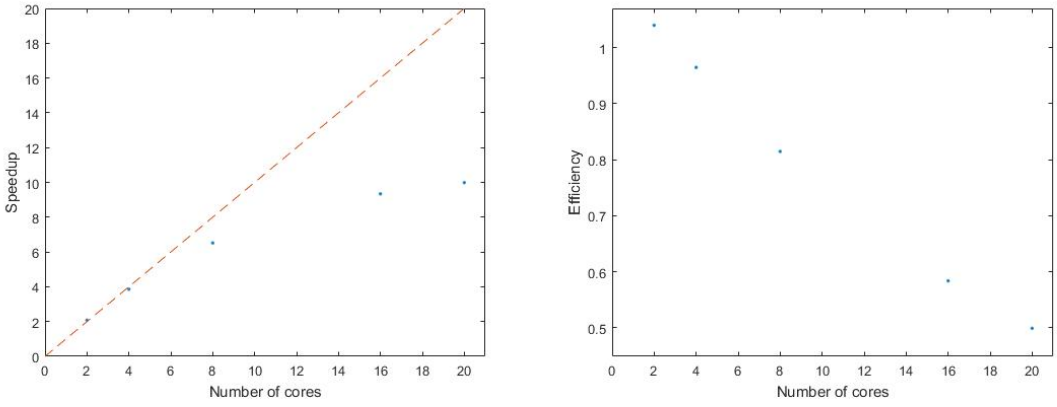
The speedup and efficiency of the algorithm for n x n input matrices, with $n = 2000, 3500, 5000$ was calculated for 1, 2, 4, 8, 16 and 20 number of cores and are presented in Figures 1 - 3. Additionally, since one common use of QR-factorization is to solve the least squares problem which generally deals with overdetermined system of equations, the performance on an m x n input matrix with $m = 25000$ and $n = 500$ was also investigated and the results are shown in Figure 4. The sizeup compared to the serial runtime of an n x n input matrix with $n = 2000$ was computed to reduce the effect of processor speed and level of optimization of the algorithm on the performance results. The result is shown in Figure 5.



(a) (b)

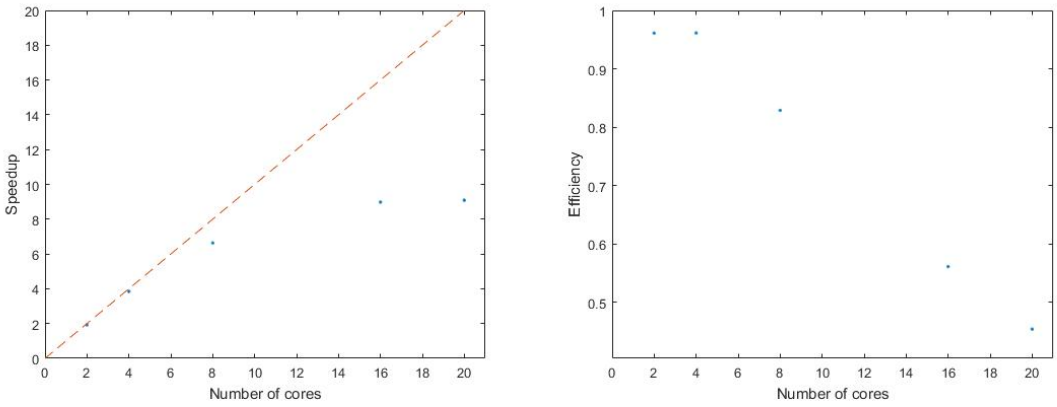
Figure 1: The (a) speedup and (b) efficiency for an n x n input matrix with $n = 2000$.

[1] Oliveira et al. Analysis of Different Partitioning Schemes for Parallel Gram-Schmidt Algorithms, 3
[2] UPPMAX website: https://www.uppmax.uu.se/resources/systems/the-rackham-cluster/



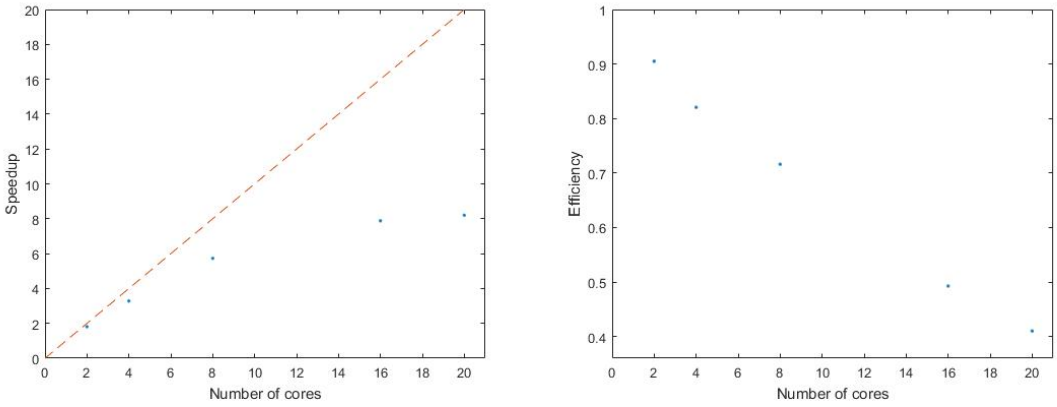
(a) (b)

Figure 2: The (a) speedup and (b) efficiency for an n x n input matrix with $n = 3500$.



(a) (b)

Figure 3: The (a) speedup and (b) efficiency for an n x n input matrix with $n = 5000$.



(a) (b)

Figure 4: The (a) speedup and (b) efficiency for an m x n input matrix with $m = 25000$ and $n = 500$.

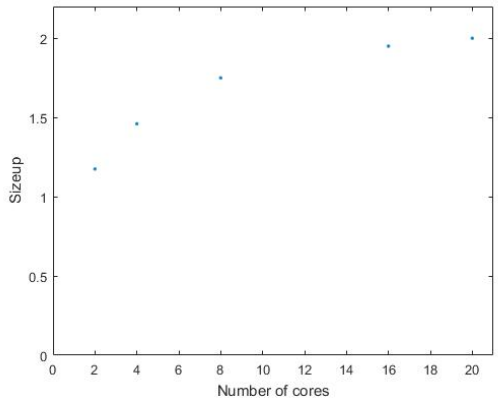


Figure 5: The sizeup compared to the serial runtime of an n x n input matrix with $n = 2000$.

5 Conclusions

The speedup of the algorithm is better for smaller square problems compared to larger ones, which can be seen by comparing Figure 1 to Figures 2 and 3. The results for the problem with $m = 25000$ and $n = 500$ (Figure 4) show similar characteristics to the larger square problems, even though the serial runtime was very similar to that of the smallest square problem (5.02 s for $m = 25000$ and $n = 500$ compared to 4.51 s for $m = 2000$ and $n = 2000$). This suggests that the scalability of the algorithm is dependent on the number of rows, m, of the input matrix. As m increases, the inner product of the vectors become the dominating factor in determining the speed of the algorithm. Since this computation is not dependent on the number of processes (each process calculates the inner product of its own vectors), this could explain the reduction in efficiency as m increases. The sizeup for square matrices compared to the serial runtime with $n = 2000$ shown in Figure 5, seems to confirm this idea. The sizeup appears to stagnate at around 2 as the number of cores are increased, likely due to that the number of rows of A become the predominant factor in determining the runtime. Again, the inner product calculation is not affected by the number of cores used which may explain this behavior.

The results imply that to improve the algorithm, the vector operations should be done more efficiently. From a cache utilization perspective, the algorithm is fairly efficient already since all the vectors are stored contiguously in memory. However, the speed at which the vector operations are calculated could be greatly improved with the use of a highly optimized basic linear algebra library (BLAS), that takes advantage of vector registers, instruction level parallelism and even better cache optimization.

Another interesting extension to the parallelization of the algorithm would be to not only partition the matrix column-wise, but also row-wise. That way, it might be possible to help the scalability of the algorithm even for large m, since several cores could be working on the same vectors. *Open MPI* offers functions that can create two-dimensional topologies of the processes and thus could enable simultaneous column and row-wise partitioning.

References

[1] Oliveira S., Borges L., Holzricther M. and Soma T. Analysis of Different Partitioning Schemes for Parallel Gram-Schmidt Algorithms. *Parallel Algorithms and Applications* 14, no.4 (2000): 3-12. doi: 10.1080/10637199808947392. http://homepage.divms.uiowa.edu/~oliveira/PAPERS/mgs-journal.pdf