Subcubic Parallelized Matrix Inversion

Through Strassen's Algorithm

Andres Valdes, Alejandro Perez, Dewey Mowris, Andy Herrera

April, 2018

Abstract

Matrix inversion is one of the most computationally complex matrix operations. This is unfortunate, as matrix inversion has many practical applications, including encryption, where the contents of a matrix are obfuscated by multiplying by a square, invertible cypher matrix and can only be retrieved my multiplying it by the inverse of that cypher.

1 Introduction

A matrix is a rectangular array of numbers or expressions, for example:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \end{bmatrix}$$

This is what is referred to as a 2×4 matrix, as it has two rows and four columns. In general, we speak about matrices as $m \times n$, where m is the number of rows and n is the number of columns. The types of matrices we will deal with are called "square" matrices, this is the case that we have an $m \times n$ matrix such that $m = n$, we refer to these also as $n \times n$ matrices. Only square, non-singular[1] matrices are invertible.

The inverse of a matrix A, is a matrix A^{-1} that when multiplied by A, yields the identity matrix, I. I_n is the square, $n \times n$ matrix with the top-leftmost to the bottom-rightmost diagonal filled in with ones, and where every other element is a zero.

$$I_n = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}_{n \times n}$$

So $AA^{-1} = I$, this is the linear algebra equivalent of $a\frac{1}{a} = 1$.

Traditionally, the inversion of a matrix involves the use of the formula

$$A^{-1} = \frac{1}{det(A)} adj(A),\ adj(A) = (C_A)^T$$

Where C_A is the coefficient matrix of A, that is, a matrix of the dimension of A with determinants filled in for each of the elements.

This is a particularly heavy computation, as the common algorithm for determinant computation is of the computation complexity $O(n!)$, with the best algorithm being $O(n^3)$, forcing us to do $O(n^5)$ computations for the coefficient matrix alone, as there are n^2 elements in A.

[1] A matrix whose determinant is nonzero.

Thankfully, we do have a way of reducing the problem for the calculation of a matrix inverse via an analytic inversion formula known as blockwise inversion[**?**]. In blockwise inversion, we split our matrix M into matrix blocks, such that

$$M^{-1} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} + A^{-1}B(D - CA^{-1}B)^{-1}CA^{-1} & -A^{-1}B(D - CA^{-1}B)^{-1} \\ -(D - CA^{-1}B)^{-1}CA^{-1} & (D - CA^{-1}B)^{-1} \end{bmatrix}$$

The advantage to this divide-and-conquer algorithm is that it can be shown that the computational complexity of the entire inversion is the same as the computational complexity of the multiplication algorithm used[**?**]. This is where Strassen's algorithm, with a complexity of $O(n^{log_2 7})$, comes into play.

Strassen describes an algorithm to multiply two matrices by splitting each into four matrix blocks of equal dimensions and performing seven multiplications between those blocks as opposed to eight[**?**].

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},\ B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},\ AB = C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

We define seven new matrices

$$\begin{aligned} M_1 &\coloneqq (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &\coloneqq (A_{21} + A_{22})B_{11} \\ M_3 &\coloneqq A_{11}(B_{12} - B_{22}) \\ M_4 &\coloneqq A_{22}(B_{21} - B_{11}) \\ M_5 &\coloneqq (A_{11} + A_{12})B_{22} \\ M_6 &\coloneqq (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &\coloneqq (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

Note that for each matrix, we multiply only once. We can now compose these matrices into C, or AB.

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

And thus, we've achieved matrix multiplication in subcubic time, and by extension, we can achieve matrix inversion in subcubic time.

2 Previous Works

There exist many algorithms for inverting matrices in cubic time via Gaussian Elimination in parallel, similar to the one implemented in many of the popular linear algebra packages like BLAS and LAPACK, and also the parallelized implementations of it.

There are also parallelized implementations of Strassen's algorithms in BLAS and elsewhere, but their implementation does not strictly tie in with the matrix inversion problem addressed in this article. The topic is particularly sparse when looking towards parallelized inversion algorithms in Go, where parallelized Strassen algorithms are scarse to begin with.

This article aims to implement an algorithm that, while not new, hasn't been parallelized for the application of inverting matrices, especially not in Go.

3 Process

Now, in order to implement the algorithm described above, we'll be using Go, Google's language written particularly for concurrency using a wonderful structure called a "goroutine."

Essentially, our algorithm can be broken up into two algorithms, both of which have aspects that can be parallelized. First and foremost, we need to split our matrix sub-blocks. We then perform multiplications across those sub-blocks using Strassen's algorithm in order to get the seven intermediate matrices, and we do so in parallel. Of course, the intermediate scalar multiplications, additions between matrices, and copying of matrices are parallelized trivially in and of themselves.

The code on the following page implements the mathematical algorithm described below:

$$
\begin{aligned}
M_1 &:= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &:= (A_{21} + A_{22})B_{11} \\
M_3 &:= A_{11}(B_{12} - B_{22}) \\
M_4 &:= A_{22}(B_{21} - B_{11}) \\
M_5 &:= (A_{11} + A_{12})B_{22} \\
M_6 &:= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &:= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

```
wg.Add(7)
go func(wg *sync.WaitGroup) {
    defer wg.Done()
    utils.copy(m1, Mult(Add(a11, a22), Add(b11, b22)))
}(&wg)
go func(wg *sync.WaitGroup) {
    defer wg.Done()
    utils.copy(m2, Mult(Add(a21, a22), b11))
}(&wg)
go func(wg *sync.WaitGroup) {
    defer wg.Done()
    utils.copy(m3, Mult(a11, Add(b12, Scale(-1, b22))))
}(&wg)
go func(wg *sync.WaitGroup) {
    defer wg.Done()
    utils.copy(m4, Mult(a22, Add(b21, Scale(-1, b11))))
}(&wg)
go func(wg *sync.WaitGroup) {
    defer wg.Done()
    utils.copy(m5, Mult(Add(a11, a12), b22))
}(&wg)
go func(wg *sync.WaitGroup) {
    defer wg.Done()
    utils.copy(m6, Mult(Add(a21, Scale(-1, a11)), Add(b11, b12)))
}(&wg)
go func(wg *sync.WaitGroup) {
    defer wg.Done()
    utils.copy(m7, Mult(Add(a12, Scale(-1, a22)), Add(b21, b22)))
}(&wg)
wg.Wait()
```

You may notice the use of "WaitGroup", these are essentially pools of goroutines that have yet to finish, the main thread will be halted until these goroutines finish.

After we've acquired these matrices, we can find compose our result matrix, C, in the following manner:

$$
\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

Via the following Go implementation:

```
wg.Add(4)
go func(wg *sync.WaitGroup) {
    defer wg.Done()
    utils.Copy(c11, Add(Add(Add(m1, m4), Scale(-1, m5)), m7))
}(&wg)
go func(wg *sync.WaitGroup) {
    defer wg.Done()
    utils.Copy(c12, Add(m3, m5))
}(&wg)
go func(wg *sync.WaitGroup) {
    defer wg.Done()
    utils.Copy(c21, Add(m2, m4))
}(&wg)
go func(wg *sync.WaitGroup) {
    defer wg.Done()
    utils.Copy(c22, Add(Add(Add(m1, Scale(-1, m2)), m3), m6))
}(&wg)
wg.Wait()
```

Strassen's multiplication, in our implementation, recurses down to a base case of a trivial 1×1 multiplication.

This is done for every step in our blockwise inversion involving multiplication, the implementation of which is far too lengthy and complex to include in this article and would be better understood within the context of the entire package, found on the GitHub repository.[2]

Once the algorithm has recursed down to the base case, the 3×3 case, the function calls close, fully inverting the matrix.

[2]Repository: https://github.com/SubcubicInversion/implementation

4 Results

The following results compare out runtime against that of BLAS, the low-level package that handles NumPy's matrix inversion.

Size	Strassen	BLAS
2×2	44.55ns	367.89ns
4×4	648.49ns	503.06ns
8×8	2.41ms	1.05ms
512×512	2min, 2s	12.49ms

Clearly, BLAS' matrix inversion implementation, despite its $O(n^3)$ time complexity, is far faster for matrices of reasonable sizes. The reason for this is that the assembly-level implementation of BLAS uses cache tricks to significantly reduce the time required to compute the inverse.

This advantage of caching over lesser complexity does dwindle as n approaches infinity, however; but the matrices of that magnitude are far too large for our tests to terminate within a reasonable amount of time.

Another reason that our algorithm may take longer to execute may be the fact that Go is a garbage-collected language, and through recursive calls, many intermediary structures are created and then simply unreferenced, forcing Go to garbage-collect a large host of memory.

5 Conclusions

While for our tests, the algorithm developed to invert a matrix fell short of the algorithms used commonly as implemented by BLAS and LAPACK, it was the first parallelized algorithm to invert a matrix via Strassen's multiplication implemented in Go. It is interesting to see how, despite Go's low-level nature and the parallelization of an algorithm of lesser complexity, it is still beaten out by clever caching tricks to avoid fully computing the inverse element by element.

This does not mean that we cannot improve on the algorithm, as it be helpful to continue to improve on the space-complexity of the problem and perhaps write the algorithm on a lower-level language, one that is not garbage-collected, like C or Assembly, as BLAS is written.

Perhaps in the future, we can run this algorithm on a computer fast enough to terminate within a passable amount of time for very large n, so that we may see the aymptotic relationships between both algorithms.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 3rd ed., MIT Press, Cambridge, MA, 2009, §28.2.

[2] Strassen, Volker, *Gaussian Elimination is not Optimal*, Numer. Math. 13, p. 354-356, 1969

[3] Zhang, F., Zhang, F. (2005). *The Schur Complement and Its Applications: Numerical Methods and Algorithms*. Boston, MA: Springer US.