Trillion by Trillion Matrix Inverse: Not actually *that* crazy

Alexander Fix

Cornell University

Misha Collins

GISHWHES Institute of Technology

August 8, 2014

Abstract

A trillion by trillion matrix is almost unimaginably huge, and finding its inverse seems to be a truly impossible task. However, given current trends in computing, it may actually be possible to achieve such a task around 2040 — if we were willing to devote the the entirety of human computing resources to a single computation. Why would we want to do this? Perhaps, as Mallory said of Everest: "*Because it's there*".

1 Introduction

An intriguing computational task was recently posed in [1]: find the inverse of a trillion by trillion matrix. At face value, this seems like an insane task. A trillion by trillion matrix is almost impossibly large, containing 10^{24} elements. Using single-precision floating point numbers, this requires 4 Yottabytes of storage, rougly a thousand times the current size of the entire internet.

However, even truly ridiculous computational tasks are no match for exponential growth. As of 2007, the current total computational resources of humanity total $6 \cdot 10^{18}$ Flops, with $3 \cdot 10^{21}$ bytes of storage [2]. Assuming optimistic growth rates following Moore's law, by 2040, this will have expanded to $6 \cdot 10^{27}$ Flops, mostly in highly parallel processors such as GPUs. Storage and internet bandwidth are growing at even faster rates. As we'll see, even Yottabyte-sized matrix inverse tasks become feasible with years of increasing computing technology.

2 Background

Due to numerical stability issues, it is typical not to compute the matrix inverse A^{-1}, but instead a factorization $A = LU$ where L is lower-triangular and U us upper-triangular. Then, A^{-1} can by evaluated by $A^{-1}x = U^{-1}L^{-1}x$, with U^{-1} and L^{-1} being computed by forward and backward substitution.

Fast LU factorization algorithms are usually based on the block-Schur decomposition. If we decompose A into blocks

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \tag{1}$$

and we wish to get a decomposition

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} \tag{2}$$

then we can multiply out the right hand side to get

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{bmatrix}. \tag{3}$$

In particular, we have $A_{11} = L_{11}U_{11}$ is the LU-decomposition of A_{11}; $U_{12} = L_{11}^{-1}A_{12}$ and $L_{21} = A_{21}U_{11}^{-1}$; and $L_{22}U_{22} = A_{22} - L_{21}U_{12}$.

Therefore, we can compute the LU-decomposition of A by the following

1. Compute the LU decomposition $L_{11}U_{11}$ of A_{11} (recursively).
2. L_{11} and U_{11} are invertible, so compute $U_{12} = L_{11}^{-1}A_{12}$ and $L_{21} = A_{21}U_{11}^{-1}$.
3. Set $S = A_{22} - L_{21}U_{12}$ and recursively compute the LU-decomposition of S to get L_{22} and U_{22}.

Previous parallel algorithms for LU-decomposition, such as [3] mostly consider the case with n processors for an $n \times n$ matrix. Since we have the sum total of every computing device on the planet to work with, we'll instead consider the massively parallel case, where there are $O(n^2)$ parallel threads of execution.

3 Massively parallel algorithm

Our abstraction of this massively parallel machine will have m^2 parallel units of execution. Divide A into an $m \times m$ grid of equal-sized square blocks

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & & A_{2m} \\ \vdots & & \ddots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mm} \end{bmatrix} \tag{4}$$

Each thread (i, j) will perform all calculations regarding the block $A_{i,j}$.

The first step is for thread $(1, 1)$ to compute $A_{11} = L_{11}U_{11}$, an LU-decomposition of a single block. Using the block-Schur algorithm, we know that $U_{i1} = L_{11}^{-1}A_{i1}$ and $L_{1i} = A_{1i}U_{11}^{-1}$. So, the second step is for thread $(1, 1)$ to send L_{11} and U_{11} to each thread $(i, 1)$ and $(1, i)$, which then compute U_{1i} and L_{i2} for each $i = 2, \ldots, m$.

Then, the next step is to compute the blocks S_{ij} of S. These are each computed by $S_{ij} = A_{ij} - L_{i1}U_{1j}$. The necessary computation is for threads $(i, 1)$ and $(1, j)$ to send the results of their computation to

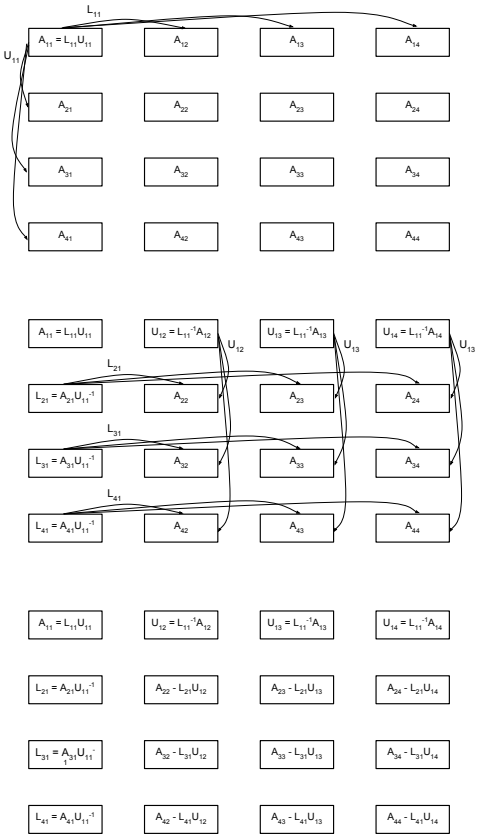

Figure 1: The 3 steps of the algorithm. (1) Thread $(1, 1)$ computes the factorization of A_{11} and passes the factors horizontally and vertically. (2) Each thread $(i, 1)$ and $(1, i)$ computes the respective blocks L_{i1} and U_{1i} and passes these results to every other thread in the same column or row (respectively). (3) Each thread (i, j) subtracts the product of $L_{i1}U_{1j}$ from A_{ij}.

thread (i, j) which can then form their product and subtract from A_{ij}. Finally, having computed S we can recurse to find L_{ij}, U_{ij} for $i, j > 2$.

This process is summarized graphically in Figure 1.

A final note: if the actual inverse matrix is desired, this can be obtained by backsolving $LUx = e_i$ for each basis vector e_i. These can be run in parallel and pipelined, taking a total time similar to finding the LU-decomposition.

3.1 Analysis

The basic operations of this algorithm are matrix operations on the individual blocks, and passing messages from one thread to another. Let $b = m/n$ be the dimension of each block. Since $b << n$, we can assume that all computations involving a single block take some constant time B to perform. Each of the 3 steps of Figure 1 involve separate matrix operations in each block, so can be solved in parallel in total time B. So, each recursion takes time $3B$ for a total of $3Bn$ for the whole algorithm.

We also need to be able to broadcast messages between the threads. The requirements are that a thread (i, j) must be able to send an identical message to each other node in its row or column. The messages are the same size $b \times b$ as the blocks, so we need total bandwith of $b \times b$ per basic step (time B) between the threads.

4 Implementation

For a trillion by trillion matrix, we'll use a block structure with blocks of size $10^6 \times 10^6$, so that the blocks also form a $10^6 \times 10^6$ grid. That is, we have a trillion blocks with a trillion elements each. Within each block, the dominating calculation is the LU-decomposition which takes $\frac{2n^3}{3}$ floating point operations, for a total of $6 \cdot 10^{17}$ operations.

With the total 2040 world computational resources of $6 \cdot 10^{27}$ Flops organized into a trillion independent units (each of which may be inherrently parallel, e.g., a GPU), we have $6 \cdot 10^{15}$ Flops per block, so each block operation B takes rougly 100 seconds. The total execution of the algorithm is $3B \cdot 10^6$ for a total of $3 \cdot 10^8$ seconds, or 9.5 years — almost reasonable, for using the entire computational resources of humanity on a single task.

4.1 Alternate implementation: Matrix inverse via the Matrix

The above analysis relies on continued growth of traditional computing resources via Moore's law, hardly a given with the several barriers to further miniturizing transistors on silicon chips. However, an even greater computational resource can be found in every single human: it's estimated that the human brain is capable of an Exa-Flop (10^18 Flops). With modest growth of the human population to 10 billion, this gives total resources of 10^30 flops, allowing the task to be completed in only 3.5 days. Harvesting of these computational resources presents a challenge, but based on the work of [4], we propose large towers of bio-vats, each holding individual humans plugged into a virtual reality environment simulating the year 1995.

References

[1] Misha Collins. personal communication, 2014.

[2] Martin Hilbert and Priscila Lpez. The worlds technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, 2011.

[3] Zhiyong Liu and D.W. Cheung. Efficient parallel algorithm for dense matrix {LU} decomposition with pivoting on hypercubes. *Computers and Mathematics with Applications*, 33(8):39 – 50, 1997.

[4] A. Wachowski and L. Wachowski. The Matrix. 1999.