Parallel Implementation of a Jacobi Linear Solver from File

Daniel Ackerson, Alex Flaherty

University of Maryland: Baltimore County

Abstract

Solving partial differential equations is valuable in many different domains. From groundwater flows to climate modelling such systems of equations are large, commonplace, and often the faster you can solve them the more accurate the solution is for the application. Therefore supercomputers are an extremely useful tool in such applications.

In this paper we propose a Jacobi Method iterative solver that will run in parallel to take advantage of the concurrent processing power of a supercomputer. Additionally, the solver will read the linear systems of equations from files instead of having data collection and storage occur entirely within the supercomputer. Our implementation will allow for a more adaptable workflow and cut down on needless time and resources used on the supercomputer.

Motivation

Matrices are powerful tools when attempting to solve linear systems of equations. A very commonly used linear solver, for sparse diagonally dominant matrices, is the Jacobi iterative solver. Unfortunately for larger matrices this method can be slow to converge so the hope is to improve the speed of convergence to a useful answer by parallelizing the Jacobi Method. From Weisstein, for each of the n equations in the linear system (Ax = B) if the ith equation is

$$\sum_{j=1}^{n} a_{ij}x_j = b_i$$

then we can simply solve for x_i assuming all other entries of x remain fixed. This is the Jacobi Method given as

$$x_i^{(k)} = \frac{b_i - \sum_{j \neq i} a_{ij}x_j^{(k-1)}}{a_{ij}}$$

and it is only guaranteed to converge for a sparse diagonally dominant matrix.

Fortunately, these types of matrices are common for solving partial differential equations. A real world example of partial differential equations being solved with the Jacobi Method is climate modelling. According to Müller, it is common for climate models to make a discrete grid instead of modelling the Earth's climate as a whole. Additionally he mentions that coarse grid equations are well suited for the Jacobi Method and will lead to rapid convergence for a solution.

When considering our implementation we also took the hardware constraints to making this Jacobi solver work for larger matrices (n > 1000) into account. Knowing that most people do not at their desk have the appropriate hardware to solve systems this large we decided to have

our solver read matrices from file(s) so that solutions could be found on supercomputers and reported back. To our knowledge this is a unique approach to solving partial differential equations.

We wanted to see if we could efficiently solve these systems on a supercomputer if the equations were simply in a file uploaded to the supercomputer. The workflow would then be for the problems to be collected and uploaded to a location in the filesystem on the remote supercomputer and have the results reported back once the solver was ran remotely.

Objectives

The first thing we wanted to see was that we could parallelize the Jacobi solver and find out if our parallel implementation is faster than the fastest serial implementation we could find. Once we had everything implemented we could compare the running times.

We were also looking to see if our file reading could keep up with the solver over many matrices and if not, at what size matrices would file reading be faster than solving. This is the part that is really unique in our approach and it will be interesting to find out if and when it is a viable strategy.

Methods

Methods for Implementation

The first thing we needed to do was find a fast serial Jacobi solver. What we went with was the JACOBI C++ library from FSU because of its simplicity. It operates on arrays of double-precision floating-point values and supplies functions for matrix-vector multiplication,

vector-norm from vector subtraction, single iteration of jacobi method, and other auxiliary functions. The test file provided performs a fixed number of iterations before stopping.

Before starting with implementation we experimented with the fastest way to read a file in Linux. We attempted to use a double buffering technique with a separately running thread constantly filling up the buffers in the background while a main thread would simply request some data from worker thread. Unfortunately due to having many edge cases to account for this was slower even than requesting from the file as needed.

Once this conclusion was reached we found the National Institute of Standards and Technology's (NIST's) Matrix Market implementation of storing to, and reading matrices from, files. Their implementation is much faster and also provided the option of using full matrix encoding or simply coordinate encoding for sparse matrices. Since the Jacobi method operates on sparse and diagonally dominant matrices we thought the coordinate encoding would be best. This method stores only the matrix coordinates and the value of the non-zero elements of an array which will save wasted processing power storing and reading the mostly zero matrices we would produce.

From there we created a matrix generator for diagonally dominant sparse matrices and stored them in the NIST Matrix Market format. This was not something we were interested in testing but merely a way to feed our final implementation the files it needed to work through.

In order to read the sparse matrix as quickly as possible from the file we did not want to waste clock cycles on zero elements of the matrix. Fortunately the decision to store in coordinate encoding paid off here. Since the "new" operator of C++ zeros out the memory allocated we could simply go line by line through the file and place every value at its location only and any

ignored locations were zero as intended. Additionally, the Matrix Market format gives us as the first line the dimensions of the array to be read so the memory can be pre-allocated.

Before converting the serial solver into a parallel solver, we changed the stopping criteria for the algorithm. Instead of stopping after a fixed number of iterations, we stop when the solution converges or after a maximum number of iterations. The metric we used for convergence is the vector-norm of the difference between the current solution and the previous solution and we stop when this value is less than some threshold:

$$\left|x_{i+1} - x_i\right| = \sqrt{\sum_{i=0}^{n} (x_{i+1} - x_i)^2} < \varepsilon$$

The norm of a vector is a measurement of the size of the whole vector. The principle behind this stopping criteria is that once the size of the vector-difference is sufficiently small, the solution will not converge further by a significant amount and therefore no additional benefit is gained from further iterations.

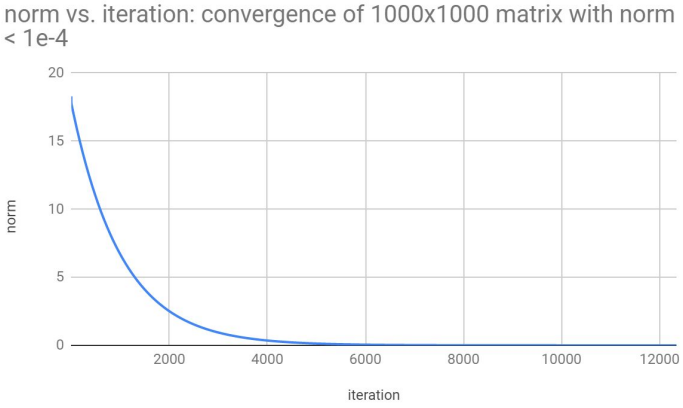


Chart C.1. *Visualization of the convergence of a solution using the Jacobi Method.*

Another change made, though it may be considered pre-optimization, was to the representation of the matrix in memory. The JACOBI library chose to address matrices in column-major order. In consideration of cache usage, this was changes to row-major ordering.

To parallelize the algorithm, we must look for operations that have few dependencies on other operations. In each iteration, every item in new solution can be computed in parallel. However, the solution cannot be updated until the whole computation is complete or else parts of the new solution will be used to calculate items within the same iteration. This is fine for a serial implementation and is essentially the Gauss-Seidel Method, but this poses problems for the Jacobi Method done in parallel. The method we used was the use of two solution buffers: one for the previous solution and one for the current solution. The current solution is calculated from the previous and then the buffers are swapped each iteration. The items in the solutions are divided as evenly as possible into contiguous ranges for each node to compute.

Methods for Analysis

The variables we considered for analyzing the runtime of the solver are matrix size, matrix sparsity (measured as percent of matrix that is zero), number of compute nodes, and solution threshold. We collected the runtime and number of iterations for solving matrices, changing each of these variables individually. Each test was done ten times on different matrices of same size to obtain both the averages and standard deviations.

To observe the effect of matrix size on runtime and number of iterations, we solved matrices with dimensions ranging from 250 to 2000 in intervals of 250. The solver ran on 4

nodes and stopped with a norm less than 1e-4. The matrices were generated with an average sparsity of 80% zeros.

To observe the effect of the number of compute nodes on runtime and number of iterations, we solved matrices using node count ranging from 1 to 8. The solver ran on matrices of size 1000 and stopped with a norm less than 1e-4. The matrices were generated with an average sparsity of 80% zeros.

To observe the effect of the solution threshold on runtime and number of iterations, we solved matrices, only stopping with a norm less than thresholds ranging from 1e-1 to 1e-10, decreasing by factors of 10. The solver ran on 4 nodes and on matrices of size 1000. The matrices were generated with an average sparsity of 80% zeros.

To observe the effect of sparsity on runtime and the number of iterations, we solved matrices that were generated with sparsities ranging from 5-95% zeros with intervals of 10%. The solver ran on 4 nodes, on matrices of size 1000, and stopped with norm less than 1e-4.

Results

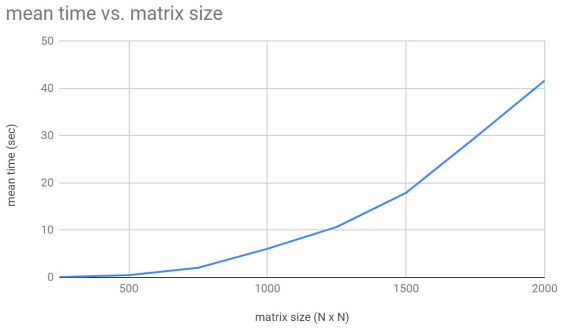
Due to our decision to use NIST's coordinate matrix encoding, the file sizes we were reading from were much smaller and reading matrices from files was always faster than solving them for matrices as small as n = 250 to as large as n = 2000.

We saw good strong scalability with the parallel solver. The runtime very clearly decreases with increasing number of compute nodes. The processor used to run these tests has four cores and eight hyperthreads. The limitations of the processor can be seen in chart ***C.4.*** where the runtime levels out at 4 or more nodes used.

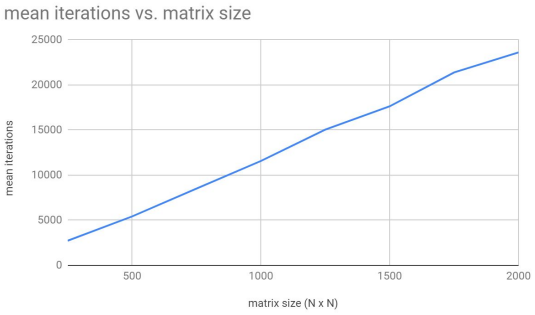
The Jacobi Method is, as most iterative solvers are, only good at solving matrices that are sparse. This is reinforced by chart ***C.8.*** which shows the runtime decreasing as the matrices become more sparse.

Lastly, we see in chart ***C.6.*** that it takes exponentially more time to increase the accuracy of the solution by a factor of 10.

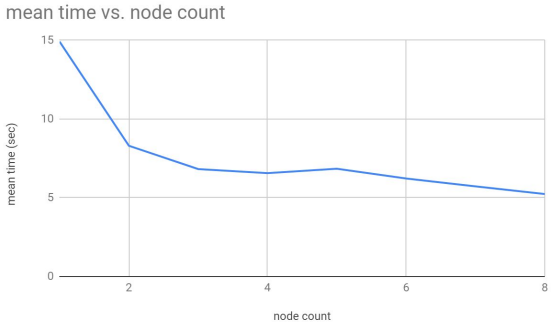
Charts



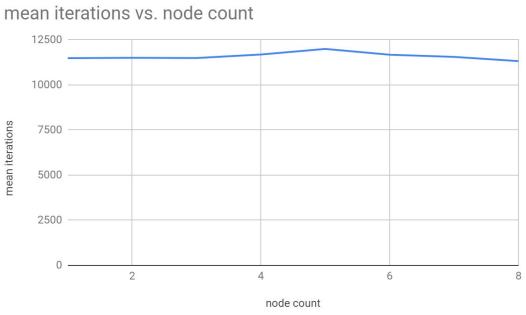
C.2. *Mean runtime increases exponentially with increasing matrix size.*



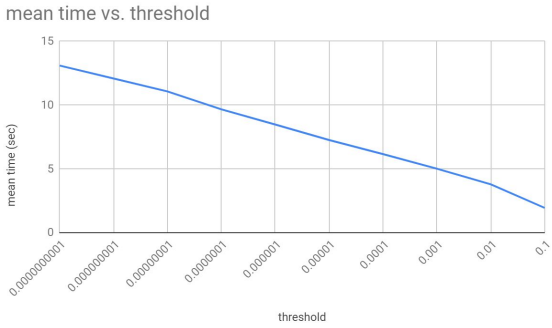
C.3. *Mean number of iterations increases linearly with increasing matrix size.*



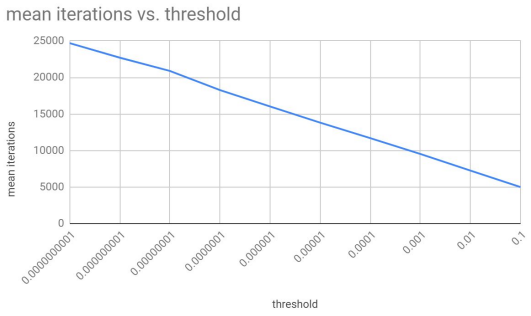
C.4. *Mean runtime is inversely related to node count.*



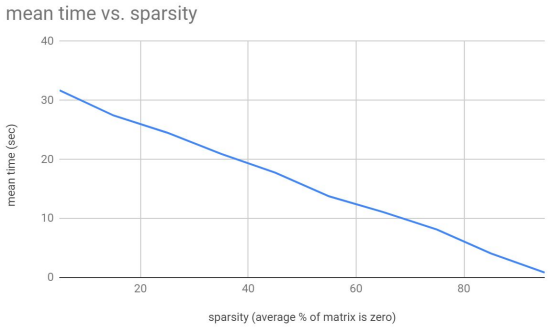
C.5. *Mean number of iterations is constant across different node counts.*



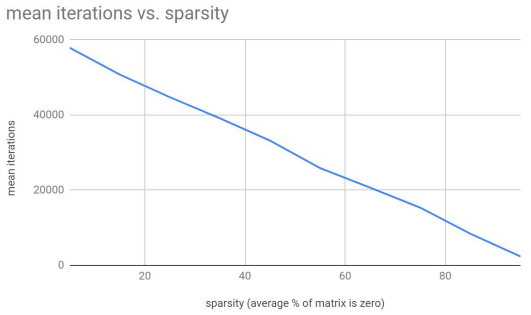
C.6 *Mean runtime is inversely related to solution threshold (or negative linear on a logarithmic scale as shown).*



C.7. *Mean number of iterations is inversely related to solution threshold (or negative linear on a logarithmic scale as shown).*



C.8. *Mean runtime decreases linearly with increasing matrix sparsity.*



C.9. *Mean number of iterations decreases linearly with increasing matrix sparsity.*

*See Appendix **A.1.** through **A.8.** for standard deviation.*

Discussion

Overall the results were very promising. While the results were mostly how we expected them to be the scaling was still surprising; the halving of runtime from one to two nodes stood out specifically.

Accepting a standard format like NIST's Matrix Market should make this implementation a viable turnkey option for anyone interested in a fast iterative solver for partial differential equations. Our addition of the tunable vector-normal threshold made this implementation even more versatile as well. We would be interested to test the solver on a climate model dataset such as those mentioned by Müller and Scheichl.

Future Work

With promising results we wanted to look forward to see what ways our implementation can be further improved. Since most supercomputers run a flavor of Linux, and since Linux treats everything as a file, it should be trivial to adapt the implementation to read matrices from a network socket instead of a locally stored file which will bypass drive access and, with direct memory access, possibly some processor cycles as well.

Additionally a more efficient matrix storage method could be achieved by storing matrices in binary instead of text format. A simple protocol would need to be agreed upon for communicating the format but reading binary instead of converting text would be an improvement.

References

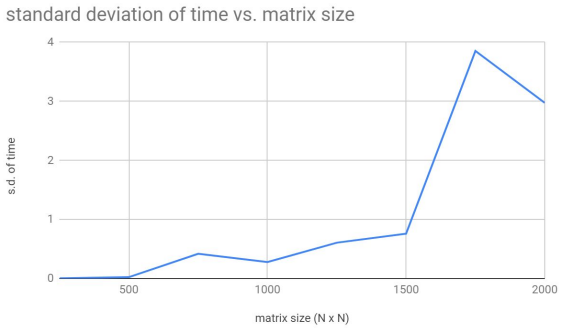
(2000). ANSI C library for Matrix Market I/O. *National Institute of Standards and Technology*, Retrieved from https://math.nist.gov/MatrixMarket/mmio-c.html

Eike Müller, and Robert Scheichl (2013). Massively Parallel Solvers for Elliptic Partial Differential Equations in Numerical Weather, 5-18, doi: 10.1002/qj.2327

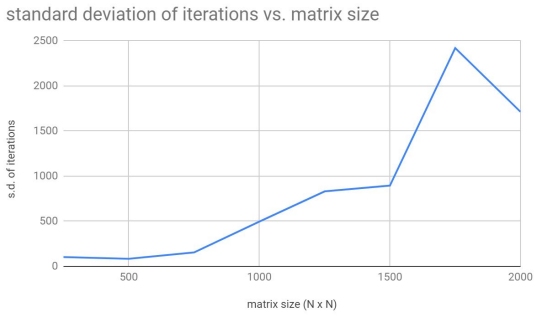
Weisstein, E. W. (2002). Jacobi method. *Wolfram Mathworld*. http://mathworld.wolfram.com/JacobiMethod.html

Appendix

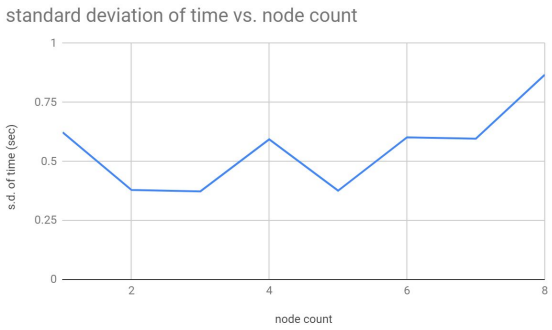
Standard Deviations



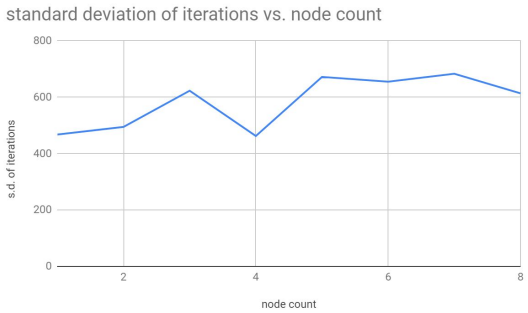
A.1.



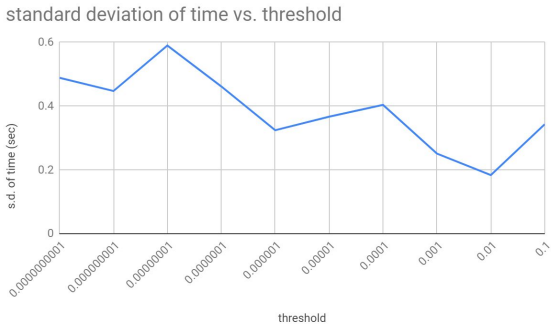
A.2.



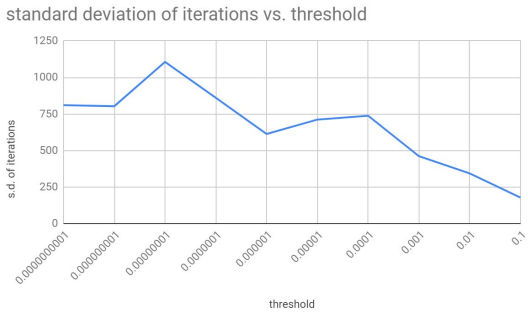
A.3.



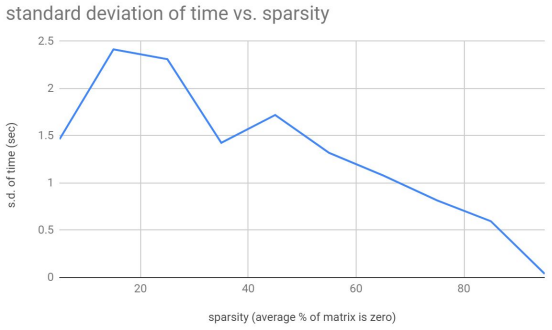
A.4.



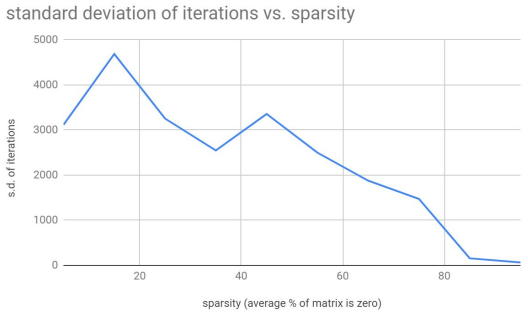
A.5.



A.6.



A.7.



A.8.