Exploiting Learning Rates on SPMD Architectures for a Parallelized Gradient Descent

Conner Hinman

Department of Computer Science

Georgia State University

Atlanta, Georgia, United States

c@nnerhinman.com

Andrew Chang

Department of Computer Science

Georgia State University

Atlanta, Georgia, United States

Ch4ng94@gmail.com

*Abstract*—In this paper we propose a method of solving the linear least squares problem, through parallelization of the common Gradient Descent algorithm. We seek to make the case for the iterative Gradient Descent approach over a number of analytical methods. The problem of parallelizing linear least squares minimization is explored, along with analysis of current techniques. A parallel Gradient Descent solution is proposed and the methods employed in our implementation are detailed. The case for our algorithm is made through comparison of techniques for hypothetical runtime scenarios, with strengths and weaknesses of each detailed. The proposed parallel gradient descent method is implemented in MPI and is benchmarked against an analogous sequential version, to demonstrate the improved efficiency.

Keywords—Gradient Descent; Linear Least Squares; QR factorization; Householder reflection; MPI; SPMD; Message-Passing;

# Introduction

In many of the most common algorithms in statistical analysis and machine learning, such as multiple linear regression, neural networks, and many others, the minimization of an error value is implicit. A common measure of error is the SSE(Sum of Squared Error) metric. Defining the SSE as the error metric makes the purpose of such algorithms to reduce the SSE as much as possible. This is a minimization problem and is achieved by searching the space for the global minima. A number of solutions exist for this problem, and each falls in to one of two categories: iterative or analytical. Analytical methods typically have very poor performance for very large datasets, where the iterative solutions tend to perform better. However, in the world we live in today, data stores are continuing to advance and the volumes of data at hand are becoming less wieldy, thus making both minimization approaches inadequate when run in an entirely sequential manner. Parallelization is crucial for dealing with such big data.

The path for parallelizing minimization algorithms is much more straightforward in the case of the analytical techniques, as most are done using some combination of simple matrix operations such as multiplication, which lend themselves to parallelization fairly plainly. The path is not so straightforward with iterative techniques such as Gradient Descent. The inherent sequential nature seems not to lend itself to parallelization, as each step must be performed in strict order. However, a small feature of the Gradient Descent algorithms, namely the learning rate, which strongly determines the performance of the algorithm, creates a clear ingress to parallelization.

In this paper we detail a technique of implementing a parallel gradient descent using an SPMD (Single Processor Multiple Data) architecture, which will outperform parallel analytical versions for very large datasets. The technique will be explored in the context of the problem of Multiple Linear Regression.

# Multiple Linear Regression

In uncountable scenarios, the following problem arises: How do we find a way to learn the relations of a vector of independent variables () to a single dependent variable ()? In practice, the problem is solved by attempting to find a vector of coefficients (), such that the hypothesis minimizes the SSE between and y. The SSE is defined as .

The purpose is to learn a function which, when provided a new vector accurately predicts the corresponding value, within acceptable boundaries. In practice, an algorithm would be provided with a set of training data, consisting of a matrix of features and examples, and a vector of response variables. Its goal is to output the vector, with which to make subsequent predictions.

The existing algorithms for this problem are either incremental or analytical in nature. The analytical solutions solve for the values that minimize the SSE in one go, using different variations of the Normal Equation:

The incremental solutions take steps one at a time towards the minimum, evaluating their progress at each step along the way.

# Minimization Methods

In each vein of solutions a number of variations exist. Each vein has its benefits and within them there is variation in problem solving methods for varying problem type.

Solutions of the analytical type minimize by computing the normal equation. Computing the normal equation comes down to performing a number of matrix operations, which leads to large computational complexity, considering matrix multiplication itself is a operation. In practice, the normal equation is not solved for directly as the factor is pretty prohibitive. One solution is to compute a covariance matrix for X and y, and determine the eigenvalues, eigenvectors, and eigenvector of best fit. This method is used in a contemporary GPU parallelization [3]. Another solution, called a QR decomposition, is to decompose the matrix into a combination of an orthogonal matrix and an upper triangle matrix . This method is used in two more contemporary parallel algorithms for CPUs [1] and GPUs [2]. Methods that use the normal equation are very straightforward, and remain useful for smaller volumes of data, but data volumes grow the matrix operations therein become prohibitive.

Solutions of the iterative type, such as Gradient Descent, Stochastic Gradient Descent, and other GD variations minimize by computing the partial derivatives of values, with respect to the SSE, and stepping down the gradient in the direction of the minimum at each iteration. Gradient descent runs in where is the number of features, and k is the number of steps it takes to converge.

Gradient descent has the disadvantage of falling victim to suboptimal minima convergence, and in practice tends to be less precise than the analytical methods. However, in practice GD tends to produce results much faster, and it is not difficult to see why: a completely eclipses a order algorithm as data volumes grow.

# Motivations

In comparing the two methods, Gradient Descent seems like the obvious choice for training on large datasets. For gradient descent to run as poorly as the other methods, the number of steps for it to converge would have to approach , which becomes increasingly unlikely, as grows large. However, with the current sizes of data that are becoming standard for this sort of problem, neither of the solutions is sufficient in a single-process, sequential form. Parallelization is crucial, and the normal equation lends itself perfectly to it, making it the more obvious choice for parallelization.

Parallelization can offer some factor of speedup to sequential versions of algorithms, and since Gradient Descent is inherently sequential, contemporary parallel solutions implement the normal equation.

Two methods, one implemented using the MapReduce model [1] and one implemented using Nvidia’s CUDA language for use on their GPUs [2], use clever partitioning techniques to run a QR decomposition across processes. QR decomposition is a method of decomposing a matrix into a product of an orthogonal matrix , and an upper triangle matrix . The QR decomposition can be performed a number of ways, usually using a series of projections or reflections. These projection methods introduce an element of sequence that removes some of the allure of the normal equation for parallelization. Clever hacks are now necessary to run the algorithms in parallel, instead of being a perfect fit. For instance, a common method is to use Householder reflections, a process that creates a number of intermediate matrices on its way to producing Q and R. This is the method used in [2]. Using Householder reflections however introduces the necessity of producing matrices in strict sequence. The authors of the GPU implementation found a way to optimize the reflections a bit by performing the reflections in blocks, but the full benefit of the GPU is just not able to be realized for QR decompositions, and their implementation produced speedup by about a factor of 5, over the sequential version. For large problem sizes the runtime is still prohibitive.

An analytical method that does not use the QR decomposition exists and also has the ability to be implemented in parallel. A covariance matrix is computed between and , and from that, eigenvalues are determined, eigenvectors created, and the eigenvector of best fit chosen. This is the method used in another GPU and CUDA based algorithm [3]. The authors were able to achieve speedup of a factor of 1000 over the sequential version. But, the process of computing a covariance matrix alone is . Once again, as problem sizes grow huge, the effect of division by a constant starts to disappear.

So, for huge problem sizes Gradient Descent is the obvious choice. In a problem of size , for GD to run as poorly as normal equation methods, it would have to take somewhere around steps to reach its goal, an unlikely scenario. When the difference is by a factor of a trillion, any constant speedup likely to be achieved through parallelization is likely to be outweighed.

Still, the inherently sequential nature of Gradient Descent is troubling, and some means of speeding up execution through parallelization is desired. Through exploitation of the learning rate, and its rate of change, nonlinear speedup is possible.

# Method for parallelizing Gradient Descent

The update step for the Gradient Descent algorithm is as follows:

where is the size of step to be taken down the gradient.

The step size of the Gradient Descent implementation is crucial to its performance. If too large of a step size is chosen, the algorithm suffers from a re-correction problem. At minima, if the step size is too large, the algorithm will continue to overcorrect, growing further and further from proper functioning at each step, as the values and corresponding error value launch off to infinity. If the step size is chosen too small, the algorithm will take needlessly long to converge, and thus behave sub-optimally. The best choice of learning rate is one that is just large enough to not shoot off to infinity. If it avoids that problem, it will converge in fewer iterations than if it had a smaller value. Another feature of the learning rate in Gradient Descent is its adaptability constants. Gradient Descent suffers from the problem of becoming stuck in local minima. One solution to that problem is to implement an adaptive learning rate, which changes in response to the previous iteration’s behavior. If a step down the gradient in the previous iteration resulted in the error growing, the learning rate is probably too large and should be reduced by a factor. If a step down the gradient produced proper results, and the error shrunk, then it might be safe to grow the learning rate by a factor. Proper choice of these constants also greatly affects the performance of Gradient Descent. However, determining proper values for these variables ahead of time is still a topic of research. A contemporary algorithm for determining proper step size, based on the data, called FASSA (Fast Automatic Step Size Estmiation) [4] exists which does a good job of choosing as large a step size as it thinks is wise. It provides speedup by a factor of about 2-3, but those results are only confirmed experimentally, and it is not guaranteed to be an optimal value. The algorithm is also a product of the fact that with a constant step size throughout, Gradient Descent is guaranteed to converge, and thus does not account for variation of the learning rate between iterations.

With a reasonable choice for the learning rate and reasonable choices for the adaptation constants, a close to optimal implementation can be found. By varying these rates around predetermined reasonable values, across processing nodes, one is guaranteed to have better performance than the rest. As the number of nodes increases, the node which converges the fastest is more likely to have closer to optimal values, with which speedup is considerable and exists over a wide range. With a parallel model, in which nodal intercommunication is possible, the runtime of the most optimal configuration becomes the runtime of the entire group.

# Parallel implementation

Our implementation, which is written in C, using the MPI interface, provides considerable speedup over the sequential analog, also written in C.

MPI (Message Passing Interface) is an SPMD programming model that uses intermodal message passing between nodes running their own version of a program. Our algorithm requires a message passing structure, so that all nodes know to cease execution as soon as the first node converges on what it considers to be an optimal solution, and thus we chose MPI. Any parallel architecture in which all nodes have some means of passing information, whether that be through message passing or shared memory access, would suffice. The details of the algorithm are as follows.

For a single node, reasonable and un-extreme values for the learning rate, the reduction constant, and the addition constant are chosen, leaning towards slower running, to avoid divergence. No functionality is currently present for programmatic choice of values for the first node. For every node after the first, the values progress slowly toward the more extreme, based on the node’s process number. All begin their own version of gradient descent, differing only in their step values, and run simultaneously. Each node is provided with a maximum number of iterations. At each step, each node meets a barrier such that they all run in lock-step. Upon a node’s convergence, it checks that it has a lower error value than its siblings, and if so, broadcasts a halt message to all nodes in the communicator.

In the case of a single node instance of the MPI version, the values will be the same as for the sequential version and thus run similarly, minus the overhead cost of communications. Since determining the optimal values a priori is guesswork, the parallel implementation is not guaranteed to produce any speedup as nodes are added; the probability of large speedups just increases in a manner hard to predict. Thus, speedup is nonlinear and can only be confirmed experimentally.

Our parallel implementation was benchmarked against the sequential version for a few datasets ranging from 5 to 100 examples. Both versions converged on accurate solution in all examples. The results are as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Implementation | Processes | Features | Examples | Runtime(seconds) | Speedup |
| Sequential | 1 | 3 | 5 | .0138 | - |
| MPI | 2 | 3 | 5 | .00148 | 9.32 |
| MPI | 4 | 3 | 5 | .0057 | 2.42 |
| Sequential | 1 | 3 | 100 | .075 | - |
| MPI | 2 | 3 | 100 | .063 | 1.19 |
| MPI | 4 | 3 | 100 | .060 | 1.25 |

Fig.1. Comparison of runtimes between MPI and sequential Gradient Descent algorithms.

# Conclusion

Many parallel algorithms exist for obtaining a solution to the Linear Least Squares problem. However, most are modifications of analytical solutions using the normal equation. While they offer tremendous speedup over their sequential analogs, the underlying algorithms are still woefully inadequate for application to huge volumes of data. Though it is not a problem fit for satisfying parallelization, the learning rate present in the algorithm presents the possibility, and Gradient Descent should be the choice for Multiple Linear Regression. For the right size of problem, a sequential version of Gradient Descent will run more efficiently than any parallel normal equation implementation. With sufficient computing power, and proper exploitation of the learning rate, Gradient Descent can be made to run much faster.

##### References

1. Moufida Adjout Rehab and F. Boufares, “Scalable Massively Parallel Learning of Multiple Linear Regression Algorithm with MapReduce” 2015 IEEE Trustcom/BigDataSE/ISPA conference.
2. Andrew Kerr, Dan Campbell and Mark Richards, “QR Decomposition on GPUs”.
3. Jyoti B. Kulkarni, A. A. Sawant, and Vandana S. Inamdar, “Database Processing by Linear Regression on GPU using CUDA” 2011 International Conference on Signal Processing, Communication, Computing and Networking Technologies (ICSCCN 2011).
4. Yuchuan Qiao, Baldur van Lew, Boudewijn P.F. Lelieveldt, and Marius Staring, “Fast Automatic Step Size Estimation for Gradient Descent Optimization of Image Registration” IEEE Transactions on MedicalImaging, Vol.35, No.2, February 2016.