PARALLELIZING NON-NEGATIVE LEAST SQUARES

Allyn Muzhi Xu
Courant Institute of Mathematical Sciences
New York University
NetID: mx501
mx501@nyu.edu

ABSTRACT

In this report, we describe our parallel implementations of a non-negative least squares solver using OpenMP and MPI. Non-negative least squares is a convex optimization problem which is equivalent to linear least squares with a non-negativity constraint on the solution. We implement parallelized versions of the sequential coordinate-wise algorithm. We touch on one practical application of non-negative least squares, and describe the implementation details, performance, and challenges faced when parallelizing the non-negative least squares algorithm.

1 Introduction

Non-negative least squares (NNLS) is a linear least squares problem where the solution is constrained to be non-negative. Namely, given a matrix $A \in \mathbb{R}^{M \times N}$ and target vector $y \in \mathbb{R}^M$, we solve for vector $x \in \mathbb{R}^n$ such that

$$x = \underset{x \geq 0}{\operatorname{argmin}} ||Ax - y||_2^2. \tag{1}$$

We can equivalently reformulate this problem as optimizing,

$$\underset{x \geq 0}{\operatorname{argmin}} \frac{1}{2} x^T Q x + c^T x. \tag{2}$$

where $Q = A^T A \in \mathbb{R}^{N \times N}$ and $c = -A^T y \in \mathbb{R}^N$. As such, NNLS can be seen as a special case of semidefinite programming.

2 Applications

One application for NNLS is from my own research in using mass spectrometry in proteomics. The goal of proteomics is to identify and quantify the proteins in a given sample. Each protein produces a unique signal in the mass spectrometer, a signal which only depends on it's mass and charge. Namely, the mass spectrometer measures mass divided by charge (m/z) of each ionized molucule in the sample, and effectively outputs a 'histogram' of m/z detected in the sample.

Since each protein sequence has a fixed isotope distribution and ionization pattern, the protein produces a known fixed signal. Moreover, since the final output of the mass spectrometer is a non-negative linear combination of all the signals of all proteins, one can use NNLS to reconstruct the proteins that generated the mass spectrometer data. In particular, if we take y as the discretized mass spectrometer output, the columns of A as the signal (over m/z) of each possible candidate protein, and x as the solution to eq. (1), then the ith entry of x can be interpreted as the abundance of the ith protein.

An parallel implementation of NNLS is important in this case since the number of proteins can be $N \approx 10^7$ in which case the quadratic formulation of NNLS in eq. (2) has $Q \in \mathbb{R}^{N \times N} \approx \mathbb{R}^{10^7 \times 10^7}$.

3 Algorithms for NNLS

There are a number of commonly used algorithms for NNLS [1]. One popular method is an active-set method introduced by Lawson and Hanson. Although this algorithm has nice convergence guarantees, this approach requires solving the pseudo-inverse of a matrix every iteration [2] and so can be prohibitively expensive.

There are other solvers such as Landweber's gradient descent method, and coordinate-wise algorithms. We will be focusing on an algorithm called Sequential Coordinate-wise Algorithm (SCA) [3], which iteratively solves for the optimal value of one coordinate, while keeping all others fixed. We chose this algorithm for it's speed and simplicity, which makes implementing a parallelized version simpler and more comprehensible.

3.1 Sequential Coordinate-wise Algorithm

The Sequential Coordinate-wise Algorithm is very straightforward: for each coordinate $k \in \{1, ..., N\}$, we solve for x_k^* which minimizes $\frac{1}{2}x^T Q x + c^T x$ given the other coordinates x_i, $i \neq k$ are fixed. Since the objective function is quadratic in each coordinate, this gives us a very simple expression for the above as,

$$x_k^* = \max\left(0, x_k - \frac{(Qx + c)_k}{Q_{kk}}\right). \tag{3}$$

We can save computing $Qx + c$ on every iteration by maintaing a separate variable μ to track the value of $Qx + c$, and by updating μ accordingly when x is updated.

In other words, let $x^{(t)}$ be our approximation for the solution at step t. Let $\mu^{(t)} = Qx^{(t)} + c$. Then, our update is given by,

$$x_k^{(t+1)} = \max\left(0, x_k^{(t)} - \frac{\mu_k^{(t)}}{Q_{kk}}\right) \tag{4}$$

$$\mu^{(t+1)} = \mu^{(t)} + (x_k^{(t+1)} - x_k^{(t)})q_k \tag{5}$$

where q_k is the kth column of Q. Executing one update above will be referred to as a 'step'. The above step is executed for each coordinate $k = 1$ through $k = N$, which corresponds to one full 'iteration'. The algorithm is terminated after a fixed number of iterations are completed, or by using some other stopping critera.

Note that in the above equation, the first line (eq. 4) is a scalar update. However, the second line (eq. 5) updates the vector μ which takes $O(N)$ operations. We can speed up the iteration slightly by not updating vector μ if x did not change from the previous step (i.e. $\mu^{(t+1)} = \mu^{(t)}$ if $x^{(t+1)} = x^{(t)}$).

In pseudocode, SCA can be written as follows:

```
for iter = 1,...,max_iters:
    for j = 1,...,N:
        old_x_j <- x[j]
        x[j] <- max(0, x[k] - mu[k] / Q[k,k])

        if x[j] - old_x_j != 0:
            for i = 1,...,N:
                mu[i] = mu[i] + (x[k] - old_x_k) Q[i,j]
```

Parallelization

Although SCA is straightforward to implement sequentially, it was not clear how to best parallelize the code so it would be scalable. The main issue is that for each step, after we update one coordinate x_k, we need to update the whole vector μ. The whole vector μ is then needed to update the next coordinate x_{k+1}.

Since the coordinate updates alternate with vector updates for μ, this makes it challenging to update multiple coordinates in parallel. Moreover, if we're using multiple nodes, we need to have some level of communication to properly update μ.

One obvious core-level parallelization we can carry out is to replace the for loop that updates μ with an omp for loop. We designate this algorithm as 'SCA SEQ+OMP' (while the basic sequential algorithm is denoted as 'SCA SEQ'). Parallelizing across multiple compute units with MPI is slightly more difficult, and we outline two implementations below.

3.2 MPI Implementation 1

The main theme of multi-node parallelization of SCA is to minimize the number of communications between the nodes when updating μ. A naive implementation would be to simply have node 0 gather blocks of μ from every node, and then perform an update for coordinate x_k. A better implementation, however, would be to instead communicate the updated value of x_k and have each node update their block of μ (as every step you are communicating a scalar x_k instead of a vector μ of length N).

As such, our first MPI parallized SCA is to break Q μ, and x into blocks of size $N_{proc} = N/\text{num\_processors}$. Processor p will have access to the pth block of Q, i.e. the rows i of Q where $i \in [p \cdot N_{proc}, (p+1) \cdot N_{proc})$. Processor p will therefore be responsible for updating the pth block of μ, i.e. elements μ_i where $i \in [p \cdot N_{proc}, (p+1) \cdot N_{proc})$.

When updating coordinate $x_k^{(t)}$ at step t, processor p such that $k \in [p \cdot N_{proc}, (p+1) \cdot N_{proc})$ will be tasked with computing x_k since processor p knows the kth row of Q. After updating $x_k^{(t)}$ to $x_k^{(t+1)}$, processor p broadcasts $\delta x_k := x_k^{(t+1)} - x_k^{(t)}$ to all other nodes, and every node will update their respective block of μ.

This method of dividing the task is desirable since updating any entry of μ only requires the difference $\delta x_k = x_k^{(t+1)} - x_k^{(t)}$. Therefore, as noted above, we can accomplish each step by broadcasting the scalar δx_k, rather than scattering and gathering all pieces of μ when updating μ.

One last hiccup in the process is that the sequential version has the advantage of skipping updates of μ when $\delta x_k = 0$ (c.f. SCA sequential code). We ideally want to avoid broadcasting anything when $\delta x_k = 0$. Otherwise, the MPI implementation will have to perform a broadcast when the sequential implementation can skips a μ update, possibly hampering the effectiveness of the MPI implementation greatly. This issue can be easily circumvented by having processor p to only broadcast δx_k for which $\delta x_k \neq 0$, and by additionally broadcasting the coordinate k that was updated. This requires broadcasting two scalars, but not requires no broadcasts when a μ update is skipped.

This MPI implementation is referred to as 'SCA MPI'. Finally, this MPI implementation can still benefit from OMP by parallelizing the μ update with a OMP for loop within each compute node. This implementation with OMP for loops, is referred to as 'SCA MPI+OMP'.

3.3 MPI Implementation 2: Less Broadcasting

One issue with above implementation is that we perform a scalar broadcast whenever mu needs to be updated (i.e. when $\delta x_k \neq 0$). Since there is an overhead cost to performing a broadcast, it is somewhat wasteful to only broadcast a scalar each time. Namely, broadcasting a scalar multiple times is in practice slower than broadcasting an array once, in which the array contains all the aforementioned scalars. This leads to another possible speed up with the MPI implementation.

Rather than processor p broadcasting δx_k at every chance, we can have processor p hold off on broadcasting δx until it computes some (or all) of them. As such, for our second MPI implementation, we have processor p first compute all δx_k for all coordinates it's responsible for, i.e. for $k \in [p \cdot N_{proc}, (p+1) \cdot N_{proc})$. Then processor p broadcasts the δx array (i.e. $[\delta x_{pN_{proc}}, ..., \delta x_{(p+1)N_{proc}-1}]$) at once to all other nodes.

This clearly saves on broadcasting overhead since rather than performing multiple broadcasts for each δx_k, they are all broadcasted at once. The downside of this implementation is that all other processors are idling as they wait for processor p to finish computing every δx_k. As such, this implementation trades less broadcasting for more idling. We refer to this implementation as 'NNLS-LB' (LB for 'less broadcasting').

This implementation can be improved by including non-buffering broadcasts, and perhaps by broadcasting the δx array in smaller more regular intervals, but this was not included in the current implementation.

Performance & Discussion

We compare the performance of the five NNLS implementations: sequential, OMP, MPI, MPI-OMP, and MPI-LB. The implementations are compared by their wall-clock time and flops on randomly generated data, where we varied the size of the data set, number of compute units, and the max number of iterations. All experiments were run on the Greene cluster.

The data was generated by randomly sampling $A \in \mathbb{R}^{N \times N}$, $x_{true} \in \mathbb{R}^N$ with entries iid uniform from 0 to 1. Then y is computed as $y = Ax_{true}$, along with $Q = AA^T$ and $c = -A^T y$.

The first four implementations were tested for $N = 5000, 10000, 20000$ for 100 iterations with 10 compute units. Then they were tested for $N = 10000$ for 100 iterations with 5, 10, 20 compute units.

Lastly the MPI-LB implementation was tested separately with $N = 10000$ for 100 iterations with 10 compute units.

Flop Rate

Here we calculate the number of flops performed in NNLS to compute the flop rate.

In one iterations, we have $2N$ flops for calculating the x_k updates (c.f. eq. 3) and N flops calculating δx_k . Moreover, we have a total of $2N^2$ flops for all the μ updates in an iteration. Hence the total flops is:

$$\text{\# of flops} = \text{max\_iters} \cdot (3N + 2N^2) \tag{6}$$

We note that since all of our MPI and OMP implementation still perform the same floating-point operations, they all have the same flop calculation as the above.

Hence it suffices to only report wall-clock time when comparing these five NNLS implementations.

Varying N

We have the following wall-clock timing in seconds for 10 compute units and max_iters $= 100$:

	N		
	5000	10000	20000
SEQ	0.0223	0.0539	0.1364
SEQ+OMP	0.0196	0.0471	0.1203
MPI	0.0214	0.0501	0.0731
MPI+OMP	0.0222	0.0530	0.0747

We see that MPI and MPI+OMP perform very similarly, and they considerably outperform SEQ and SEQ+OMP for the larger problem size ($N = 20000$) due to the MPI taking advantage of the 10 compute units. We see that the performance of the MPI implementations suffer and match that of SEQ when the problem is too small ($N = 5000$) since there are not as much computation to split among the nodes and the parallelization overhead (such as broadcasting) become more dominant. We see that SEQ+OMP performs the best for $N = 5000$ and $N = 10000$) as it can make use of parallelized update to μ with little to no parallelization overhead.

Curiously, the performance of the OMP versions seem somewhat lackluster as the μ update seems to be the bottleneck of the iteration step, for which one expects a 8-core processor to compute this 8 times faster. Moreover, it seems odd that the MPI+OMP implementation performs worse than the solely MPI implementation.

Varying the Number of Compute Units

We have the following wall-clock timing in seconds for $N = 10000$ and max_iters $= 100$:

	# of Compute Units		
	5	10	20
SEQ	0.0537	0.0539	0.0530
SEQ+OMP	0.0478	0.0471	0.0472
MPI	0.1439	0.0501	0.0367
MPI+OMP	0.1515	0.0530	0.0379

We see that the sequential implementations do not vary in performance when the number of compute units are changed, since they only use one compute unit to begin with. However, we see a drastic improvement for the MPI implementations as the number of nodes is increased from 5 to 20. We note that the improvement in performance for the MPI implementations are not linear with compute units, since there are different tradeoffs between distributed computing and the overhead of broadcasting. We see that at 20 compute units, the MPI implementations finally outperform the OMP implementation. At 5 compute units, the MPI implementations are 3 times slower than the SEQ implementations, which I would attribute to the fact that broadcasting and synchronization have expensive overheads.

Again, the OMP implementations are not providing a significant (if at all) improvement to performance. I'm not sure what exactly causes this; potentially forking and unforking the processes introduces some slowdown every time.

Varying Max Iterations

Here, we ran the same experiment but with a higher max iteration of 2000. Since the max iteration is much higher, one would expect that many of the coordinates have converged (or are zero) and so many coordinate updates are skipped (since $\delta x = 0$). As such, the purpose of this experiment is to verify that the MPI implementations don't suffer when many coordinate updates are skipped. Here $N = 10000$ with max_iter $= 2000$:

```
    # of Compute Units
         10       20
SEQ     1.7774  1.7741
SEQ+OMP 1.7393  1.7505
MPI     1.8824  1.2853
MPI+OMP 2.0328  1.3906
```

We see that the MPI implementations still outperform the SEQ implementations when there are enough compute units. Comparing these numbers to those of the previous section, we see that the MPI implementation is slightly slower than with max_iter=100 (relative to the sequential baseline). Namely when the compute units is 10, the MPI implementation is slower than the sequential implementation here, unlike in the previous section. This slowdown makes sense as there is still a small overhead when skipping updates in the MPI implementation, since there are still broadcasts being made when a processor finishes updating the coordinates that it is responsible for. Nevertheless, this effect seems rather mild, demonstrating that our implementation is robust to skipping iteration updates.

MPI Less Broadcasting

Lastly, we ran our second implementation of MPI (with less less broadcasting) using $N = 10000$ and 10 compute units. We found the following wall-clock timings in seconds:

```
SEQ     0.0522
MPI-LB  0.2098
```

It seems as though the MPI implementation with less broadcasting is performing significant worse. One explanation for this is that this was poorly implemented. Namely, although the algorithm broadcasts fewer times, many of the processors are idling as they wait for one processor to computes all of its own the coordinate updates. Idling could have been reduced by using non-buffering broadcasts.

Moreover, this implementation may have been too aggressive in attempting to group all the broadcasts together; as discussed before, the broadcasts of δx could have been set to be slightly more frequent so that there the other compute units could begin computing while waiting for the next broadcast.

Conclusion

In this work, we implemented the Sequential Coordinate-wise Algorithm for solving the non-negative least squares problem, both sequentially and in parallel. We provided implementations using OMP and MPI, with a focus on minimizing the amount of communications between compute nodes in each iteration.

The performances of the implementations were tested in various settings such as different random data set size, and number of computing nodes. We found that the MPI implementation performed the best when the data set was large and many compute units were used, demonstrating that the implementation had uses for solving very large NNLS problems. The MPI implementation suffered significantly when the problem was too small as the overhead of distributed computing outweighed the benefits. Using OMP had marginal benefits, most likely due to costs of frequently forking the processes.

References

[1] Donghui Chen and Robert J. Plemmons. Nonnegativity constraints in numerical analysis. In *in A. Bultheel and R. Cools (Eds.), Symposium on the Birth of Numerical Analysis, World Scientific*. Press, 2009.

[2] Charles L. Lawson and Richard J. Handson. In *Solving Least Squares Problems*, page 291. SIAM, 1995.

[3] Vojtěch Franc, Václav Hlaváč, and Mirko Navara. Sequential coordinate-wise algorithm for the non-negative least squares problem. In André Gagalowicz and Wilfried Philips, editors, *Computer Analysis of Images and Patterns*, pages 407–414, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.