Parallel NMF Implementation for High Performance Computing

Samuel Sasaki
Dept. of Mathematics
Pomona College

Nick Crites
Dept. of Mathematics
Pomona College

James Lucassen
Dept. of Mathematics
Harvey Mudd College

***Abstract*—Due to the rising costs of inter-processor communication often associated with distributed computing networks, significant literature exists attempting to optimize local algorithms such that bandwidth cost is optimized and inter-processor communication–and thus overall computational cost– is minimized. Building off of this body of knowledge, this project implements a few of the available parallel non-negative matrix factorization (NMF) algorithms optimized to be run on multiple local processors as described by Kannan, Ballard, and Park in their paper titled *A High-Performance Parallel Algorithm for Nonnegative Matrix Factorization*. Replicating their framework, which was written in MPI/C++, this project aims to implement the three algorithms described in pseudocode in the aforementioned paper. More specifically, we provide our own implementation derived from the available pseudocode for the following algorithms: Alternating Non-negative Least Squares (ALNS), Naive-Parallel NMF, and High Performance Computing (HPC)-NMF. In particular, this project will focus on deriving these implementations, validating the significance of Kannan et. al's findings, and motivating future work in optimizing NMF algorithms at low levels of abstraction.**

***Index Terms*—nonnegative matrix factorization, parallelism, graphical processing unit, high-performance computing**

I. Introduction

Non-negative matrix factorization is a type of constrained low rank matrix factorization technique that can produce interpretable compressions of complex data. It works by approximating a non-negative input matrix $\mathbf{A}$ as a product of two low rank non-negative factor matrices $\mathbf{W}$ and $\mathbf{H}$. Despite this technique's many interesting use cases, one difficulty is that NMF is not well suited to handle very large datasets. This is largely because the highly non-convex optimization for $\mathbf{A}$ is often performed with strategies such as alternating least-squares (ALS), which alternates between holding either $\mathbf{W}$ or $\mathbf{H}$ constant and updating the other. Even when using very fast algorithms to perform these local updates such as multiplicative updates (MU) [7] or block principal pivoting (BPP) [3], the ALS algorithm can only perform local search over half the available dimensions at a time. This means it often requires many iterations to converge, making NMF inefficient to compute. In addition, because advancements in computational power are progressing much more rapidly than bandwidth and latency speeds between processors, optimization of distributed algorithms is essential for lessening such latency and bandwidth constraints. Thus, when applying NMF to large, real-world datasets to perform topic modeling, content recommendation, or video analysis, these algorithms can lead to exponentially slower run-times rendering the technique impractical for real-time applications. In response to the need for distributed NMF algorithms using both CPUs and GPUs, Kannan, Ballard, and Park propose parallelized versions of two algorithms to solve non-negative least squares (NLS) sub-problems: (a) alternating non-negative least squares (ANLS) and (b) High-Performance Computing (HPC)-NMF – an algorithm that runs an alternating least squares method in parallel while optimizing communication cost, per-iteration bandwidth cost, and local memory requirements [2]. In replicating the work of Kannan, Ballard, and Park, this analysis aims to verify the study's results, compare the performance and computational costs of Naive-Parallel NMF and HPC-NMF, and compare both Naive-Parallel NMF and HPC-NMF to simple, non-parallelized NMF.

In addition to a comparative study between the results produced in [1], this analysis offers some unique findings. First, rather than iteratively solving the NMF sub-problems with Block Principal Pivoting, this analysis employs Multiplicative Updates (MU). Next, we propose a novel implementation of HPC-NMF that we termed Vertical HPC-NMF - a slight alteration of processor communication in HPC-NMF. Finally, due to the inability to access the original source-code in [1], the implementation of the proposed algorithm is novel and may slightly differ from the original implementation. Because we do not have access to the same hardware as the original experiment, we are unable to measure this effect on overall run-time.

II. Related Work

This works both Naive-Parallel NMF and HPC-NMF based on the pseudocode noted by Kannan, Ballard, and Park. However, the methods of this study will differ due to the difference in resources between our group and that of Kannan, Ballard, and Park. In their study, they utilized a supercomputer available to them through the National Energy Research Scientific Computing Center and ran experiments on input matrices $\mathbf{A}$ with dimensions $172,800 \times 115,200$. Our implementation, instead, will be run on an NVIDIA Tesla K80 with datasets still to be determined. Despite the large difference in computing resources, the goal of this study is to validate the performance improvement claims made by Kannan, Ballard, and Park using their proposed algorithm HPC-NMF. Thus, a

comparison between non-parallelized, standard NMF implementation and HPC-NMF will suffice so long as computing power and datasets operated on are consistent between the trials. Another notable difference in methodology between our experiments and those of Kannan et. al is the choice of programming language. First, in their implementation, they utilize an MPI/C++ implementation, whereas we utilized the Python programming language. In adapting pseudo-code, the verbosity and interpretability of Python made it a more flexible tool for constructing our own implementation and provides a pathway to additionally examine performance benefits of compiler choice via the use of Cython, an optimizing static compiler that compiles Python code to C code. In addition, Kannan, Ballard, and Park performs Block Prinipal Pivoting (BPP) when solving for NMF sub-problems. This work implements Multiplicative Updates in a similar fashion to [6] (Liu, Yang, Fan). The work in which this analysis references is also built the shoulders of experts examining distributed NMF algorithms [1], [5], [6].

III. NMF Algorithms

In this paper, we exclusively examine NMF algorithms. In NMF, a non-negative input matrix $\mathbf{X} \in \mathbb{R}_{\geq 0}^{n_1 \times n_2}$ that can be decomposed into two low-dimensional non-negative matrices: $\mathbf{W}$ and $\mathbf{H}$. In addition, one must state a target dimension, or rank, that is shared by $\mathbf{W}$ and $\mathbf{H}$. This rank defines the dimensionality constraints on the reduced matrices $\mathbf{W}$ and $\mathbf{H}$. Once the input is given and the rank of the decomposition is specified, the optimization problem is outlined: $\mathbf{X} \approx \mathbf{WH}$. There exist many techniques to iteratively solve this optimization problem, but the algorithms used in this analysis employ an Alternating Non-Negative Least Squares Approach (ANLS). In such approach, $\mathbf{A}$ and $\mathbf{S}$ are updated one at a time such that the other value is held constant. This processed is defined in the equation below:

$$\mathbf{W} \approx \underset{W>0}{\operatorname{argmin}} \|\mathbf{X} - \tilde{\mathbf{W}}\mathbf{H}\|_F^2 \tag{1}$$

and

$$\mathbf{H} \approx \underset{H>0}{\operatorname{argmin}} \|\mathbf{X} - \mathbf{W}\tilde{\mathbf{H}}\|_F^2 \tag{2}$$

On a more granular level, $\mathbf{W}$ and $\mathbf{W}$ are updated element-wise for each iteration the above sub-problem. By taking the gradient of equation [1], one can arrive at a method for the entry-wise updating of both $\mathbf{W}$ and $\mathbf{W}$. The following technique to solve NMF sub-problems is known as Multiplicative Updates (MU) for $\mathbf{W}$ and $\mathbf{H}$, respectively:

$$w_{ij} \leftarrow w_{ij} \frac{(\mathbf{AH}^T)_{ij}}{(\mathbf{WHH}^T)_{ij}} \tag{3}$$

and

$$h_{ij} \leftarrow h_{ij} \frac{(\mathbf{W}^T\mathbf{A})_{ij}}{(\mathbf{W}^T\mathbf{WH})_{ij}} \tag{4}$$

A. Basic NMF

For the *Basic NMF* implementation, we consider a non-parallelized implementation of the MU algorithm. For reference, consider the pseudocode below.

Algorithm 1 [$\mathbf{W}$, $\mathbf{H}$] = Basic-NMF($\mathbf{A}$, k)

Require: $\mathbf{A}$ is an $m \times n$ matrix, k is rank of approximation.
1: Initialize $\mathbf{H}$ with a non-negative matrix in $\mathbb{R}_+^{n \times k}$.
2: **while** stopping criteria not satisfied **do**
3: Update $\mathbf{W}$ using $\mathbf{HH}^T$ and $\mathbf{AH}^T$
4: Update $\mathbf{H}$ using $\mathbf{W}^T\mathbf{W}$ and $\mathbf{W}^T\mathbf{A}$
5: **end while**

Fig. 1: Pseudocode for the Basic NMF implementation from Kannan et. al [2].

The above algorithm is an ANLS-based NMF. In this algorithm, W and H and updated iteratively and separately while holding the other constant. Updates are performed according to the equation for entry-wise multiplicative updates outlined in equations [3] and [4].

B. Naive Parallel NMF

The second algorithm implemented within [1] is Naive Parallel NMF. In this algorithm, $\mathbf{A}$ is sub-divided into mxn blocks. Optimizing for every $\mathbf{A}_{ij}$ entry, the entire $\mathbf{H}$ and $\mathbf{W}$ are updated in their entirety for each iteration of the multiplicative update. In implementing Naive Parallel NMF, the pseudo-code referenced in Fig. [3] was referenced. Below is a visual representation of the updates involved in Naive Parallel NMF. See Figure [2].

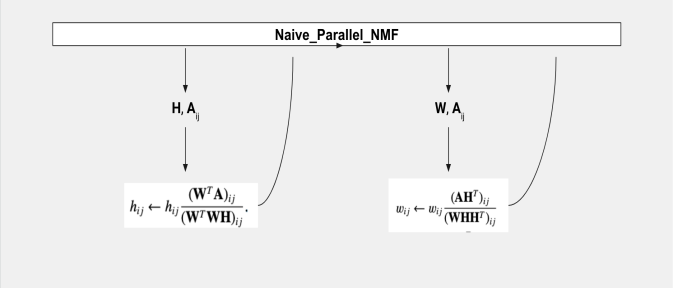

Fig. 2: A visual representation of Parallel Naive NMF.

C. Vertical HPC-NMF

Vertical HPC-NMF is this project's contribution to the distributional NMF discourse. A crossover between HPC-NMF and Naive Parallel NMF, Vertical HPC-NMF pre-computes and stores matrix operations that are used repetitively in the process of updating $\mathbf{H}$ and $\mathbf{W}$. The algorithm consists of three threads for each update of $\mathbf{H}$ and $\mathbf{W}$, respectively. Because the update for $\mathbf{H}$ and $\mathbf{W}$ are analogous, let's first look at the update for $\mathbf{H}$. In these three threads, $\mathbf{H}_{ij}$ is calculated and used to produce σ $\mathbf{H}_{ij}$ $\mathbf{H}_{ij}^T$ = $\mathbf{U}_{ij}$. Once this is completed, the thread is terminated. Next, thread two users $\mathbf{A}_{ij}$ and $\mathbf{H}_j$ to compute and store $\mathbf{A}_{ij}$ $\mathbf{H}_j^T$ $\mathbf{V}_{ij}$. This marks the end thread 2. At this point, because we have pre-computed and stored $\mathbf{A}_{ij}$, $\mathbf{U}_{ij}$, and $\mathbf{H}_{ij}$, the algorithm begins the third thread

Algorithm 2 [$\mathbf{W}$, $\mathbf{H}$] = Naive-Parallel-NMF($\mathbf{A}$, k)

Require: $\mathbf{A}$ is an $m \times n$ matrix distributed both row-wise and column-wise across p processors, k is rank of approximation.
Require: Local matrices: $\mathbf{A}_i$ is $m/p \times n$, $\mathbf{A}^i$ is $m \times n/p$, $\mathbf{W}_i$ is $m/p \times k$, $\mathbf{H}^i$ is $k \times n/p$.

1: p_i initializes $\mathbf{H}^i$
2: **while** stopping criteria not satisfied **do**
    /* Compute W given H */
3: collect $\mathbf{H}$ on each processor using all-gather
4: p_i computes $(\mathbf{H}^i)^T \leftarrow \text{SolveBPP}(\mathbf{W}^T\mathbf{W}, (\mathbf{W}^T\mathbf{A}^i)^T)$
    /* Compute H given W */
5: collect $\mathbf{W}$ on each processor using all-gather
6: p_i computes $(\mathbf{H}^i)^T \leftarrow \text{SolveBPP}(\mathbf{W}^T\mathbf{W}, (\mathbf{W}^T\mathbf{A}^i)^T)$
7: **end while**

Ensure: $\mathbf{W}, \mathbf{H} \approx \underset{\tilde{\mathbf{W}} \geq 0, \tilde{\mathbf{H}} \geq 0}{\text{argmin}} \|\mathbf{A} - \tilde{\mathbf{W}}\tilde{\mathbf{H}}\|$

Ensure: $\mathbf{W}$ is an $m \times k$ matrix distributed row-wise across processors, $\mathbf{H}$ is a $k \times n$ matrix distributed column-wise across processors

Fig. 3: Pseudocode for the Naive-Parallel NMF implementation from Kannan et. al [2].

and performs the multiplicative update for $\mathbf{H}$. An analogous process occurs for $\mathbf{H}$, but is represented by different variables. A visual representation of Vertical HPC-NMF algorithm can be seen below in figure [4].

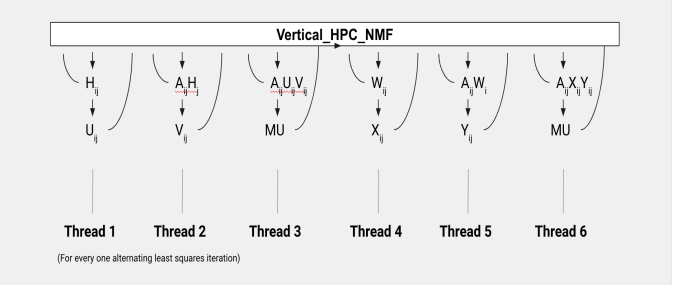

Fig. 4: A visual representation of Vertical HPC-NMF.

D. HPC-NMF

The final algorithm proposed in [1] is an ANLS-based HPC-NMF (see figure [6]). HPC-NMF utilizes the vertical threading seen in Vertical HPC-NMF, but combines threads 1, 2, and 3 into a single vertical thread. Each of these vertical threads are assigned to a given process - p. The vertical threads, represented by the number of processors p, communicate pre-computed values horizontally; this process is what we call 'horizontal threading'. In the example of the update of $\mathbf{H}$, the values for $\mathbf{HU}_{ij}$, $\mathbf{H}_j$, and $\mathbf{V}_{ij}$ are horizontally communicated back and forth between p vertical threads. In theory, using this techniques optimizes the algorithm to the number of processors, and thus the computational power, available. By employing horizontal threading, not only are unnecessary or redundant calculations of factor matrices avoided, but communication between the p processors - and thus the latency and bandwidth costs - are minimized. In addition to these advantages, communication between vertical threads also improves the 'accuracy' of each iteration therefore minimizing the number of iterations needed for convergence. The updating process for HPC-NMF can be seen in figure 5 below.

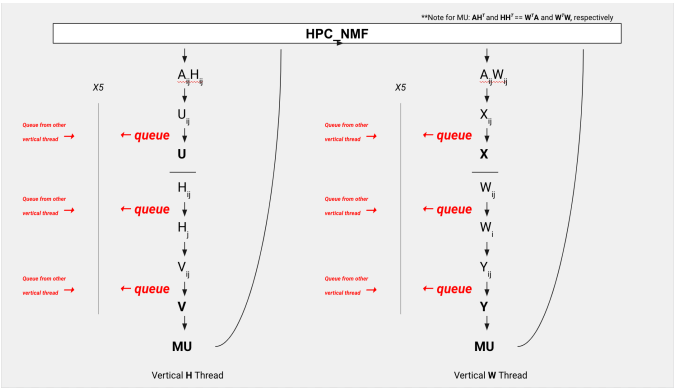

Fig. 5: A visual representation of HPC-NMF.

IV. Hypothesis

We hypothesize that the results of Kannan et. al will be found valid despite the difference in hardware and software choices when running the experiments through the realization of significant performance improvements in memory usage and runtime of the HPC-NMF implementation. Should this be true, this will motivate the development of a Python standard library to make optimized NMF a more readily accessible resource to the general machine learning community. As has been observed through this process, having to create your own version of a provably optimal technique is cumbersome and neglects one of the most powerful characteristics of the research community, collaboration.

V. Experiments

A. Datasets

In their experiments, Kannan et. al utilize four different datasets to compare their Naive-Parallel and HPC-NMF implementations. The datasets are well distributed to fit various use cases of their algorithm as they tested against data in the form of uniform Gaussian and random sparse matrices, video data, and a graph network dataset. The magnitude of size for each of these datasets was significant. For the uniform random and random sparse matrices, the dimensions were $172,800 \times 115,200$, which resulted in 24 billion elements in the matrices. The video data resulted in a $1,013,400 \times 2400$ dimensional matrix and the graph network dataset consisted of $>1,000,000$ nodes and $>3,000,000$ edges. The use of these datasets invites optimized NMF methods when combined with

Algorithm 3 [$\mathbf{W}$, $\mathbf{H}$] = HPC-NMF($\mathbf{A}$, k)

Require: $\mathbf{A}$ is an $m \times n$ matrix distributed a $p_r \times p_c$ grid of processors, k is rank of approximation.
Require: Local matrices: $\mathbf{A}_{ij}$ is $m/p_r \times n/p_c$, $\mathbf{W}_i$ is $m/p_r \times k$, $(\mathbf{W}_i)_j$ is $m/p \times k$, $\mathbf{H}_j$ is $k \times n/p_c$, and $(\mathbf{H}_j)_i$ is $k \times n/p$.

p_{ij} initializes $(\mathbf{H}_j)_i$
while stopping criteria not satisfied **do**
  /* Compute W given H */
  p_{ij} computes $\mathbf{U}_{ij} = (\mathbf{H}_j)_i(\mathbf{H}_j)_i^T$
  compute $\mathbf{H}\mathbf{H}^T = \sum_{i,j} \mathbf{U}_{ij}$ using all-reduce across all processors
  p_{ij} collects $\mathbf{H}_j$ using all-gather across proc columns
  p_{ij} computes $\mathbf{V}_{ij} = \mathbf{A}_{ij}\mathbf{H}_j^T$
  compute $(\mathbf{A}\mathbf{H}^T)_i = \sum_j \mathbf{V}_{ij}$ using reduce-scatter across proc row to achieve row-wise distribution of $(\mathbf{A}\mathbf{H}^T)_i$
  p_{ij} computes the following: $(\mathbf{W}_i)_j \leftarrow \text{SolveBPP}(\mathbf{H}\mathbf{H}^T, ((\mathbf{A}\mathbf{H}^T)_i)_j)$
  /* Compute H given W */
  p_{ij} computes $(\mathbf{X}_{ij} = (\mathbf{W}_i)_j^T(\mathbf{W}_i)_j$
  compute $\mathbf{W}^T\mathbf{W} = \sum_{i,j} \mathbf{X}_{ij}$ using all-reduce across all processors
  p_{ij} collects $\mathbf{W}_i$ using all-gather across processor rows
  p_{ij} computes $\mathbf{Y}_{ij} = \mathbf{W}_i^T\mathbf{A}_{ij}$
  compute $(\mathbf{W}^T\mathbf{A})^j = \sum_i \mathbf{Y}_{ij}$ using reduce-scatter across proc columns to achieve column-wise distribution of $(\mathbf{W}^T\mathbf{A})^j$
  p_{ij} computes the following: $((\mathbf{H}^j)^i)^T \leftarrow \text{SolveBPP}(\mathbf{W}^T\mathbf{W}, (((\mathbf{W}^T\mathbf{A})^j)^i)^T)$
end while

Ensure: $\mathbf{W}, \mathbf{H} \approx \underset{\tilde{\mathbf{W}} \geq 0, \tilde{\mathbf{H}} \geq 0}{\operatorname{argmin}} \|\mathbf{A} - \tilde{\mathbf{W}}\tilde{\mathbf{H}}\|$
Ensure: $\mathbf{W}$ is an $m \times k$ matrix distributed row-wise across processors, $\mathbf{H}$ is a $k \times n$ matrix distributed column-wise across processors

Fig. 6: Pseudocode for the HPC-NMF implementation from Kannan et. al [2].

the capable hardware as the time to configure parallelized software is far less than the run-time improvement of the multi-threaded methods described by the algorithms.

In setting out to complete this project, the goal was to validate the results that Kannan et. al were able to show. As such, it was not as significant to entirely replicate the results as it was to see similar relative performance improvements. The dataset we used in our experiments was a uniform randomly generated 1728×1152 matrix with added random Gaussian noise. The choice of dataset was made to provide the fundamentally simplest dataset to the algorithm as possible in order to see performance benefits. The dimensions of the matrix were chosen to be a number of orders of magnitude smaller than what was used in Kannan et. al in order to try to hold consistency in relative size despite the sheer difference in computing power between the experiments. More on these choices will come in section V-C.

B. Machine

Hardware specifications also play a crucial role in interpreting the results of the experiments. Kannan et. al had the liberty of using the *Edison* supercomputer housed at the National Energy Research Scientific Computing Center (NESRC). This computer boasts a 5,576 total compute nodes, each of which with dual-socket 12-core Intel Ivy Bridge processors and 64 GB of memory. In combination with this, the nodes are equipped with private 64KB L1 and 256KB L2 caches and a shared 30MB L3 cache across the sockets allowing for high performance communication between node cores.

In our experiments, we utilized an NVIDIA Tesla K80 GPU running inside of a Microsoft Azure compute instance. This Tesla K80 contains 4992 NVIDIA CUDA cores with 24 GB of memory and 128KB L1 caches [4].

C. Limitations

Because of the difference in resources between our group and Kannan et. al, we were unable to completely replicate the results observed in their paper. Both their hardware and datasets provide for more robust trials of experimentation. While attempting to meet the expectations of their datasets by creating a matrix of size $172,800 \times 115,200$, it was approximated that two of their datasets were about 366 GB in size. The requirements of computing on a dataset of this magnitude far exceeded the limits of the NVIDIA Tesla K80 we were equipped with in the cases of both system and secondary memory.

Experience has also contributed a significant limitation in our experiments. The only resource available to replicate their algorithms was pseudocode and, even then, the true implementation by Kannan et. al was composed via the Message Passing Interface (MPI) written in C++. This interface allows for even further optimization at the software level via the use of hardware-optimized software methods that specify communication between cores across a node. Due to a lack of experience, trying to even come close to replicating the algorithms defined in Kannan et. al was too much of a challenge, which contributed to the development of our true research challenge: validating the results in Kannan et. al. In our implementation of these algorithms, we opted to use the Python programming language because of its verbosity, ease-of-use, and default support for multi-threading, which provided a gateway for us to optimize our algorithm, Vertical HPC-NMF.

D. Methods

In running the experiments, 10 trials for each algorithm were run on the dataset. The runtime of the algorithm for each iteration was calculated. Additionally, the Frobenius norm was used to calculate the error of each matrix factorization from

the original data. This allows us to interpet how well each algorithm was performing with respect to the objective of NMF, which is to achieve as similar of a non-negative matrix factorization from the original data as possible. The number of threads used in the parallelized algorithms was 9.

VI. Results

It was not exactly what we predicted, but we were able to see some interesting results from the previously defined experiments. For starters, our hypothesis was somewhat correct. Vertical HPC-NMF performed significantly faster as shown in Figure 7, but the non-parallelized NMF algorithm also outperformed the two optimal, parallelized algorithms in terms of runtime.

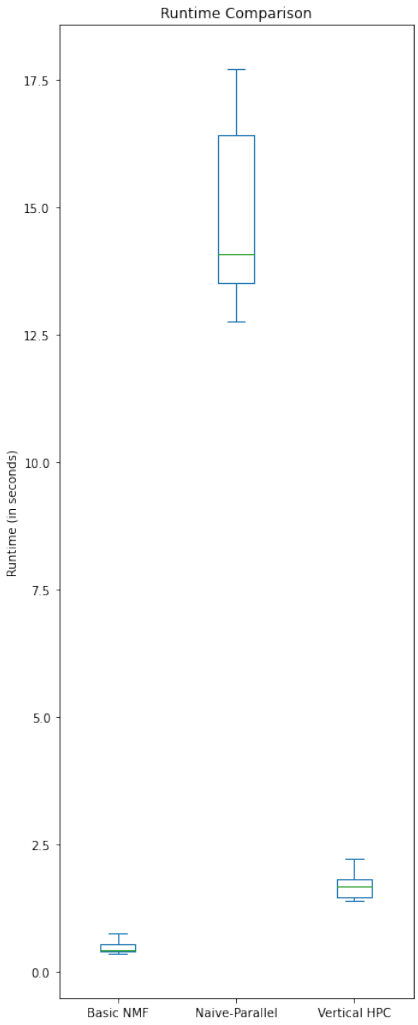

Fig. 7: Boxplots of all running times (in seconds) of the three NMF algorithms.

For a better visual of how close the non-parallelized NMF algorithm and Vertical HPC-NMF were, see Figure 8.

Also, all of the algorithms performed similarly well with regards to accuracy. They expressed similar error rates in respect to the Frobenius norm being used to compute error.

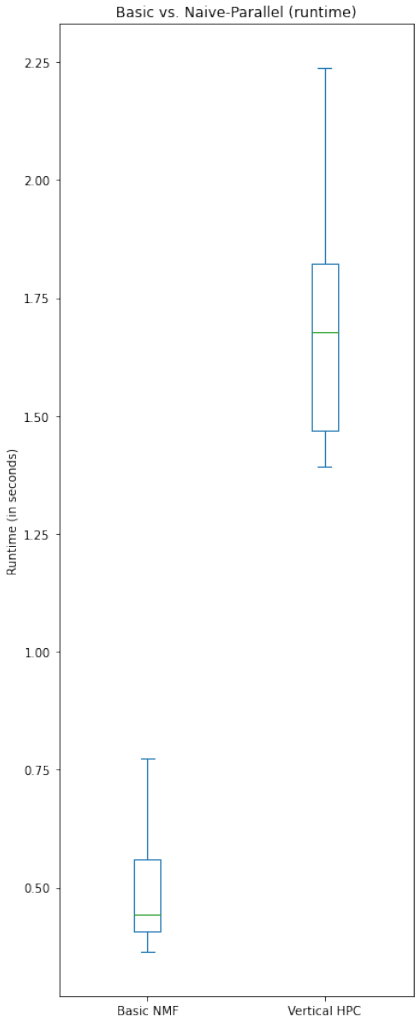

Fig. 8: Boxplots of running times (in seconds) for Basic vs. Vertical-HPC NMF.

This validates that the algorithms were in fact performing valid NMF, which was another challenging goal of the experiments. The calculated errors across iterations of the algorithms can be found in Figure 9.

VII. Conclusion

In this paper, we both replicate the results proposed in [2] and propose our own Vertical HPC-NMF algorithm. All algorithms tested however, belong to the ANLS family and thus are solved iteratively. By carefully studying and implementing these algorithms, we are able to parallelize aspects of the multiplicative update such that bandwidth and latency costs are minimized. Optimization of such costs, especially with HPC-NMF, lead to fewer iterations, a quicker runtime, and a faster convergence.

For the dataset upon which we experimented, we saw significant speed up in the Vertical-HPC NMF algorithm compared to the Naive-Parallel implementation, yet still experienced the slowness of thread management in Python. Because of this,

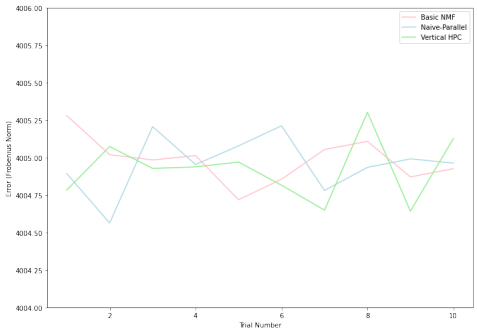

Fig. 9: Plots of the calculated errors across iterations of the three NMF algorithms.

it is apparent that in order to experience the benefits of these parallelized NMF algorithms, it is necessary to have a big data problem. If the dataset being operated on is not considerable large, then there is little point in going through the trouble of trying to implement a parallelized NMF algorithm.

Parallelizing NMF algorithms lends itself to fruitful future work. First, we would like to experiment with a further horizontalized model of HPC-NMF in which $\mathbf{U}$ and $\mathbf{V}$ are calculated in parallel under separate threads in multiplicative updates for $\mathbf{H}$. The same could be done in the computation of $\mathbf{X}$ and $\mathbf{Y}$ in the update of $\mathbf{W}$. Such an optimization would be building on the work done in [1]. Next, we would like to run this experiment using different loss functions, like the KL Divergence, for example. Using these different loss functions, and perhaps with various constraints and step sizes, we could compare the total run-time under an optimal rank and number of processors. Lastly, because the benefits of these algorithms are largely limited to computers with a nearly unlimited number of processors and only made available to individuals with prior experience with parallelization, development of a Python library to perform optimal, parallelized NMF could be extremely helpful in introducing some of the benefits of HPC-NMF to non-experts. With a roadmap towards future discovery and a library that could be accessed by anyone with Python, high-performance NMF algorithms could prove key to mending the existing disjunction between NMF and large datasets.

REFERENCES

[1] L. Gao J. Yin and Z. Zhang. Scalable nonnegative matrix factorization with block-wise updates. *Machine Learning and Knowledge Discovery in Databases*, pages 337–352, 2014.

[2] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. A high-performance parallel algorithm for nonnegative matrix factorization, 2015.

[3] J. Kim and H. Park. Fast nonnegative matrix factorization: An active set-like method and comparisons. *SIAM Journal on Scientific Copmuting*, 33(6):3261–3281, 2011.

[4] NVIDIA. Nvidia tesla k80. https://www.nvidia.com/en-gb/data-center/tesla-k80/.

[5] P. J. Haas R. Gemulla, E. Nijkamp and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. *Proceedings of the KDD, ACM*, pages 69–77, 2011.

[6] J. Guan R. Liao, Y. Zhang and S. Zhou. Cloudnmf: A mapreduce implementation of nonnegative matrix factorization for large-scale biological datasets. *Genomics, proteomics bioinformatics*, 12(1):48–51, 2014.

[7] D. Seung and L. Lee. Algorithms for non-negative matrix factorization. *NIPS*, 13:556–562, 2001.